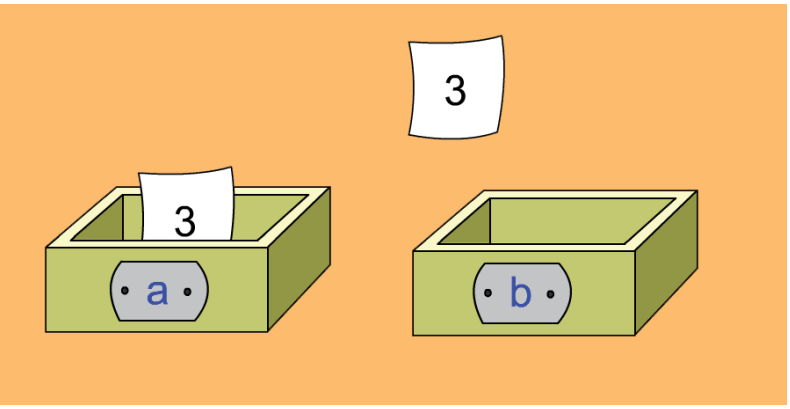
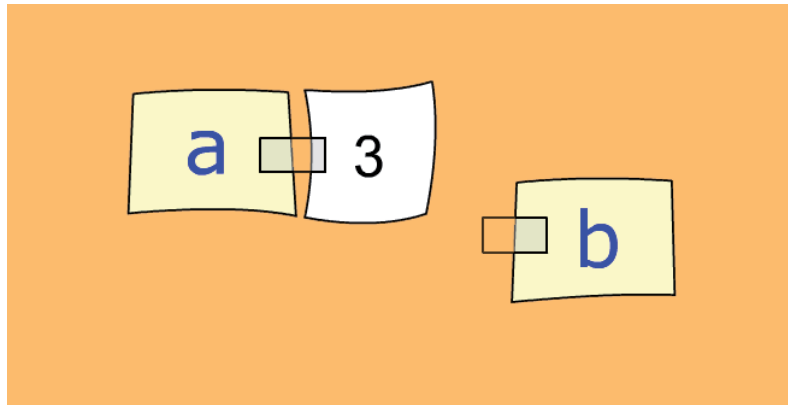
Universität Potsdam





Michael Weigend

Intuitive Modelle der Informatik

Universitätsverlag Potsdam

Michael Weigend

Intuitive Modelle der Informatik

Universitätsverlag Potsdam 2007

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über http://dnb.d-nb.de/ abrufbar.

Universitätsverlag Potsdam, Am Neuen Palais 10, 14469 Potsdam
Tel.: +49 (0) 331 977 4517 / Fax: 4625
e-mail: ubpub@uni-potsdam.de
http://info.ub.uni-potsdam.de/verlag.htm

ISBN 978-3-940793-08-9
URL http://pub.ub.uni-potsdam.de/volltexte/2008/1578/
URN urn:nbn:de:kobv:517-opus-15787
[http://nbn-resolving.de/urn:nbn:de:kobv:517-opus-15787]

Druck: docupoint GmbH Magdeburg

Institut für Informatik
Didaktik der Informatik

Intuitive Modelle der Informatik

Dissertation
zur Erlangung des akademischen Grades
"doctor rerum naturalium"
(Dr. rer. nat.)
in der Wissenschaftsdisziplin "Didaktik der Informatik"

eingereicht am 1. März 2007 an der
Mathematisch-Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Dipl.-Inform. Michael Weigend

1. Gutachter: Prof. Dr. Andreas Schwill, Universität Potsdam
2. Gutachter: Prof. Dr. Carsten Schulte, Freie Universität Berlin
3. Gutachter: Prof. Dr. Werner Hartmann, Pädagogische Hochschule Bern

Disputation am 26.11.2007

Für meinen Vater

Vorwort

Seit vielen Berufsjahren als Lehrer und Dozent interessiert mich die Frage, wie sich Menschen kognitiven Zugang zu den schwierigen abstrakten Konzepten der Informatik verschaffen. Nach vier Jahren der Forschung glaube ich, einige Facetten dieses Rätsels etwas besser zu verstehen. Die Ergebnisse sind in der vorliegenden Dissertation dokumentiert.

Mein Dank gilt Prof. Dr. Andreas Schwill (Universität Potsdam) für die vielen Ermunterungen, Anregungen und wissenschaftlichen Stellungnahmen zu meiner Arbeit. Meine Vorträge in Potsdam während den letzten Jahre waren immer eine Quelle für Motivation, Inspiration und Erkenntnisgewinn, nicht zuletzt auf Grund scharfsinniger Anmerkungen von Ralf Romeike und anderen Mitarbeitern und Gästen des Instituts für Didaktik der Informatik.

Die empirischen Erhebungen wären nicht möglich gewesen ohne die Kooperation der vielen Lehrerinnen und Lehrer, Schülerinnen und Schüler, Studentinnen und Studenten, die an Visualisierungsübungen und Workshops mit der Python Visual Sandbox teilgenommen haben und mir hoffentlich verzeihen, dass ich an dieser Stelle nur ihre Schulen bzw. Universitäten erwähnen kann: Arlington County School (Arlington, Virginia, USA), Rudolf-Diesel-Gymnasium (Augsburg), Ernst-Abbe-Gymnasium (Eisenach), Kopernikus-Gymnasium (Wissen), Gustav-Heinemann-Oberschule (Berlin), Humboldt-Gymnasium (Berlin), International American Highschool (Hongkong), International German and Swiss Highschool (Hongkong), Holzkamp-Gesamtschule (Witten), Klara-Schumann-Gymnasium (Bonn), Westfälische Wilhelms-Universität Münster, FernUniversität Hagen.

Dank auch an Prof. Dr. Ming Ming Chiu und Prof. Dr. Fong Lok Lee (Chinese University of Hong Kong) für die anregenden Korrespondenzen und persönlichen Gespräche über Metaphern und E-Learning. Verbunden fühle ich mich der Python Community – insbesondere Jeff Elkner – für die Diskussion der Python Visual Sandbox auf der PyCon 2005 in Washington D. C. Nicht unerwähnt bleiben soll Hendrik Büdding (Universität Münster), der sich die Arbeit gemacht hat, das Manuskript kritisch durchzusehen.

Witten, Dezember 2007

Michael Weigend

Inhaltsverzeichnis

Einleitung

Intuitive Modelle der Informatik sind gestaltartige, kognitiv gut verarbeitbare und als gewiss und richtig akzeptierte gedankliche Vorstellungen über informatische Konzepte. Sie werden bewusst oder unbewusst verwendet, wenn Menschen versuchen, die Arbeitsweise von Informatiksystemen zu verstehen, anderen zu erklären oder selbst – kreativ – ein solches System zu entwickeln.

Intuitive Modelle fließen innerhalb der praktischen Informatik in die Gestaltung von „Frontend-Systemen“ an der Schnittstelle zum Menschen ein. Dazu gehören z.B. höhere Programmier- und Modelliersprachen oder Entwurfsmuster. Beispielsweise ist eine Symbolik zur Beschreibung informatischer Systeme (wie z.B. UML) nur dann brauchbar, wenn sie zu schnell und sicher begreifbaren Dokumenten führt. Das Ziel ist, beim Rezipienten ein intuitives Verständnis des Gemeinten, verbunden mit dem Gefühl von Gewissheit, zu erreichen. So basieren Zustandsübergangsgraphen auf dem intuitiven Modell einer sprunghaften Bewegung von Ort zu Ort, ein Konzept, das viele Menschen schon als Kinder bei einer Reihe von Straßenspielen trainiert haben und das sie deshalb sicher beherrschen. Entwurfsmuster – wie z.B. das Iteratorkonzept – schöpfen ihren Wert für Softwareentwicklungen daraus, dass sie die prinzipielle Arbeitsweise eines Systems auf gut zugängliche, kompakte Weise in einer kohärenten Gestalt beschreiben.

Intuitive Modelle sind wichtig für das Lernen informatischer Konzepte. Aus konstruktivistischer Sicht lässt sich Lernen als Aufbau interner mentaler Modelle über die Welt auf der Basis bereits vorhandenen Wissens beschreiben. Es ist ein Vorgang, der vom Subjekt ausgeht und nicht direkt von einem Außenstehenden (z.B. einem Lehrer) beobachtet oder kontrolliert werden kann. Intuitive Modelle sind in dieser Hinsicht ein kognitionspsychologisches Phänomen (Kapitel 1). Lehrerinnen und Lehrer sind hilflos, wenn sie nicht wissen, welche gedanklichen Konzepte ihre Schülerinnen und Schüler verwenden. Erklärungen und Hilfestellungen greifen nicht, wenn sie an der Gedankenwelt des Adressaten vorbei gehen. Um Fehlvorstellungen aufzuklären und Verständnislücken zu schließen, muss man wissen wo diese liegen könnten. Intuitive Modelle sind auch die Grundlage für Medien (Texte, Bilder, Filme) und Unterrichtsaktivitäten (Rollenspiele, Programmierübungen, Explorationsaufgaben), die der Vermittlung informatischer Konzepte dienen. Um diese effizient gestalten zu können, braucht man Kenntnisse darüber, wie und wie gut und wie schnell die verwendeten Anschauungen und Analogien verstanden werden.

Die vorliegende Arbeit versucht dieses didaktische Wissen zu verbessern und verfolgt vier primäre Forschungsziele: Das Finden und Beschreiben intuitiver Modelle der Informatik, ihre Katalogisierung und Systematisierung, die Überprüfung der psychischen Realität bei Schülerinnen und Schülern und schließlich die Identifizierung allgemeiner Mechanismen der Nutzung von Modellen bei der intellektuellen Auseinandersetzung mit Programmtexten. Dabei liegt der Fokus auf Modellen zu grundlegenden Programmierkonzepten, die typischerweise im Anfangsunterricht an allgemeinbildenden Schulen thematisiert werden.[1]

Identifikation und Beschreibung

Lehrbücher, Softwaredokumentationen, Sprachreferenzen, Schülervisualisierungen und andere Dokumente enthalten mehr oder weniger explizite Repräsentationen intuitiver Modelle zur Arbeitsweise von Programmen. Diese gilt es herauszuarbeiten und zu beschreiben. So unterstützt der Satz „Die Funktion gibt eine Zahl zurück.“ unter anderem folgende intuitiven Vorstellungen: Eine Funktion ist eine aktive Entität, die z.B. etwas zurückgeben kann. Es gibt eine andere Entität, von der die Funktion etwas erhalten hat, für das sie ersatzweise jetzt etwas zurückgibt. Zahlen sind Entitäten, die man bewegen (also auch z.B. zurückgeben) kann. Wir unterscheiden zwischen einem intuitiven Modell als subjektivem gedanklichen Konstrukt (das möglicherweise unbewusst ist) und seiner Repräsentation in einer materiellen Form (Abbildung, Text etc.). Die Darstellung eines intuitiven Modells ist (durch den Rückgriff auf intersubjektive Symbolsysteme) vor allem ein kulturelles Phänomen. Kapitel 2 widmet

[1] Anzumerken ist, dass neben Modellen zu Programmierkonzepten noch viele andere intuitive Modelle existieren, die sich primär auf andere Bereiche der Informatik beziehen, wie z.B. Modelle zu Vorgehensweisen (Wasserfallmodell etc.) oder zur Untersuchung einer Domäne (Thomas 2003).

sich speziell dieser Perspektive und diskutiert verschiedene Repräsentationsformen. In dieser Arbeit werden intuitive Modelle vornehmlich durch Animationen (mit kleinen Textelementen) beschrieben. In den Details eines visuellen Vorgangs können implizite Bedeutungsfacetten einer sprachlichen Formulierung expliziert werden.

Systematisierung und Analyse

Das zweite Ziel ist, Ansätze für eine Systematik intuitiver Modelle der Informatik zu finden. In den Kapiteln 5 bis 12 werden Modelle nach inhaltlichen Kriterien eingeteilt: Allokation von Aktivität, Benennung von Entitäten, Daten, Funktionen, Verarbeitungskonzepte, Kontrollstrukturen zur Steuerung von Programmläufen, Klassen und Objekte. Grundlegende intuitive Modelle der Informatik treten meistens nicht isoliert sondern in größeren Zusammenhängen auf. Daraus ergeben sich Interdependenzen, die herausgearbeitet werden müssen. So passt das Modell einer Funktion als Fabrik, die Daten verarbeitet, gut zum Modell eigenaktiver Daten-Entitäten, die sich zum Eingang einer Funktionsentität bewegen können. Für ein und dasselbe informatische Konzept gibt es in der Regel mehrere intuitive Modelle, die miteinander verglichen werden. Je nach Kontext oder Erklärungsabsicht (Fokus) ist mal das eine und mal andere Modell besser geeignet. So ist es bei endrekursiven Prozeduren denkökonomisch, den rekursiven Aufruf als Aufforderung zur Wiederholung zu betrachten – eine Vorstellung, die bei eingebetteter Rekursion zu Fehlern führt.

Psychische Realität und kognitive Zugänglichkeit

Das dritte Ziel ist, Informationen darüber zu gewinnen, welche Modelle von Programmieranfängern beim Verstehen, Erklären und Generieren von Programmtexten tatsächlich verwendet werden. Dazu wurde mit der Python Visual Sandbox (PVS) ein technisches System entwickelt, das es erlaubt, die intellektuelle Auseinandersetzung mit Programmen wenigstens ansatzweise zu beobachten (Kapitel 4). Eng verbunden mit der Frage nach der psychischen Realität ist die Frage der kognitiven Zugänglichkeit eines Modells und damit der Eignung als Basis für ein Unterrichtsmedium. Welche visuellen Darstellungen sind verwirrend und missverständlich und welche verständlich und leicht erfassbar? In welche „Denkfallen“ tappen Programmieranfänger, wenn sie bestimmte anschauliche Vorstellungen verwenden?

Allgemeine Prinzipien der Verwendung intuitiver Modelle

Ein viertes Anliegen ist, allgemeine Prinzipien des Umgangs mit intuitiven Modellen herauszufinden. Welche Strategien und Techniken wenden Schülerinnen und Schüler an, wenn sie versuchen, die Arbeitsweise eines Programms durch intuitive Modelle zu rekonstruieren? (Kapitel 13) Welche Hindernisse gibt es auf dem Weg von einer intuitiven Lösungsidee zu einer programmtechnischen Realisierung? Wie können sie überwunden werden? (Kapitel 14)

1 Intuitive Modelle und informatisches Wissen

Intuition (lat. intuieri: ansehen, hinschauen, betrachten) wird seit der Antike in der Philosophie und Wissenschaft als Quelle für sicheres oder scheinbar sicheres Wissen beschrieben. In manchen Kontexten verwendet man den Begriff Intuition zur Bezeichnung eines Prozesses. Eine Handlung ist intuitiv, wenn sie schnell und aus einem Gefühl heraus erfolgt. Im Gegensatz dazu stehen rational durchdachte Entscheidungen, die auf logischen Schlussfolgerungen basieren und bei denen jede Handlungsalternative in Gedanken durchgespielt worden ist. In anderen Zusammenhängen wird Intuition dagegen als besondere Kategorie von deklarativem Wissen definiert. Nach Fischbein (1987) sind Intuitionen grundlegende Vorstellungen über die Beschaffenheit der Welt, die wir ohne weiteren Beweis als sicher richtig akzeptieren. Sie sind unmittelbar wie sinnliche Wahrnehmungen, gehen aber über beobachtbare Einzelereignisse hinaus. Man sieht unmittelbar an konkreten Beispielen, dass gegenüberliegende Winkel am Schnittpunkt zweier Geraden gleich sind. Die Intuition liegt in der Verallgemeinerung, dass es immer so ist. Eine Intuition im Sinne von Fischbein hat somit Theoriecharakter und verallgemeinert Einzelbeobachtungen zu einem Modell eines Wirklichkeitsaspektes.

1.1 Intuition als Wissen

Intuitive Modelle sind eine Form von Wissen. Man kann generell zwischen prozeduralem und deklarativem Wissen unterscheiden (Haberlandt 1994; Anderson 1996, 1996a). Prozedurales Wissen bezieht sich auf kognitive und motorische Fertigkeiten, deklaratives Wissen dient der Repräsentation von Objekten und Ereignissen der Umwelt. Ryle (1949) beschreibt diese beiden Wissenskategorien als „knowing how“ und „knowing that“. Die Differenzierung zwischen prozeduralem und deklarativem Wissen ist die Basis von Andersons ACT-R-Theorie (Anderson 1996a), die eine Architektur zur Simulation kognitiver Prozesse beschreibt.

1.2 Intuition als prozedurales Wissen

Prozedurales Wissen ist Wissen, wie man etwas tut. Dazu gehören motorische und kognitive Fertigkeiten wie Schreibmaschineschreiben, Rechnen, und Programmieren. In der Alltagssprache wird Intuitivität häufig prozedural gesehen. Man denkt vielleicht an den Detektiv im Kriminalroman, der sich auf seinen „Riecher“ verlässt und intuitiv das Richtige tut um den Täter zu finden. Jagdish Parikh (1994) befragte in einer internationalen Studie 1312 Manager aus neun Ländern zur Bedeutung von Intuition bei ihren täglichen Entscheidungen. Die Befragten beschrieben Intuition vor allem als kognitiven Prozess, der rationalem und logischem Denken antithetisch gegenübersteht. Weitere dominierende Attribute, mit denen Intuition charakterisiert wurde, waren „innere Wahrnehmung“, „unerklärliches Verständnis“, „Gefühl, das von Innen kommt“, „Integration vorheriger Erfahrung“, „Entscheidung ohne vollständige Datenbasis“, „sechster Sinn“ und „spontane Vision“ (Parikh 1994, S. 57). Bemerkenswert ist, dass nur 38.9 % der befragten Manager angaben, sich bei Entscheidungen mehr auf rationale Überlegungen zu verlassen als auf Intuition (Parikh 1994, S. 57).

Anderson betont, dass prozedurales Wissen unbewusst sein kann, d.h. eine Person beherrscht bestimmte Problemlösungstechniken ohne die Regeln angeben zu können, die sie verwendet. Eine versierte Schreibkraft kann zwar mit hoher Geschwindigkeit eine Tastatur fehlerfrei bedienen, aber es gelingt ihr in der Regel nicht aus dem Gedächtnis anzugeben, an welcher Stelle sich welche Taste befindet. In einem Experiment von Berry und Broadbent (1984) sollten Studenten in einem Simulationsprogramm die Produktion einer hypothetischen Zuckerfabrik steuern. Dieses gelang den Probanden auch nach einigen Versuchen. Aber keiner konnte eine Regel angeben, nach der er oder sie vorging. Die Versuchspersonen behaupteten, das Problem „intuitiv“ gelöst zu haben (nach Anderson 1996, S. 229 f.)

Leiser (2001) beobachtete, dass deklaratives und prozedurales Wissen zu einer Domäne weitgehend unverbunden nebeneinander existieren können. Er interviewte 20- bis 30-jährige Personen zu der Frage, was ein gutes Paar ausmacht. Die Befragten waren in der Lage, (deklarative) Prinzipien anzugeben, nach denen man gut zusammenpassende Menschen auswählen kann. Sie hatten zweitens kein

Problem zu entscheiden, welche Personen aus ihrem Bekanntenkreis stabile Paare bilden würden. Jedoch gerieten sie in Schwierigkeiten, als sie ihre (prozeduralen) Zuordnungen mit den zuvor genannten (deklarativen) Paarbildungsprinzipien begründen sollten, wichen von diesen Prinzipien ab oder gaben plötzlich neue an.

Halten wir fest: Intuition kann prozeduraler Natur sein. Eine prozedurale Intuition ist eine zielorientierte Aktivität, die ein Mensch sicher beherrscht. Die Besonderheit gegenüber deklarativem Wissen ist, dass die Person nicht zwingend einen Algorithmus für ihr Tun angeben kann. Das heißt nicht, dass sie nicht einem Algorithmus folgt. Aber er ist ihr möglicherweise nicht bewusst, oder es fehlen ihr die sprachlichen Mittel die eigene Problemlösung zu beschreiben. Es kann auch sein, dass sie die Aufgabe jedes Mal auf andere Weise löst. Die subjektive Gewissheit – ein Merkmal von Intuition – liegt darin, dass man sich sicher ist, die Aufgabe lösen zu können. Dagegen bleibt (unter Umständen) ungewiss, wie man das tut.

1.3 Intuitive Modelle als bedeutungsbezogenes deklaratives Wissen

Intuitive Modelle werden häufig visuell durch Bilder oder Animationen repräsentiert. Betrachtet man in einem Programmtext eine Zuweisung `x = 1`, so taucht vor dem inneren Auge vielleicht das Bild eines Kasten mit dem Etikett x auf, in den ein Zettel mit einer 1 gelegt wird. Doch eine solche bildhafte Vorstellung allein ist noch kein intuitives Modell. Anderson (1996) unterscheidet zwischen wahrnehmungsbezogenem und bedeutungsbezogenem Wissen. Ein Beispiel für wahrnehmungsbezogenes Wissen sind mentale Bilder, gedankliche Vorstellungen von früher wahrgenommenen Objekten (Anderson 1996, 103 ff). Mentale Bilder müssen nicht unbedingt mit Bedeutung belegt sein. Beispielsweise kann man einen Computer als mentales Bild, bestehend aus Monitor, Tastatur, Zentraleinheit, Kabeln etc., vorstellen, ohne die Funktion der einzelnen Komponenten zu verstehen. Die Komponenten bedeutungsbezogenen Wissens bezeichnet man meist als Konzepte (concepts). Intuitive Modelle sind bedeutungsbezogen und deklarativ. In den folgenden Abschnitten vergleichen wir Intuitionen mit drei Arten von Konzepten: Begriffe, Schemata und mentale Modelle.

1.3.1 Begriffe – Identifikation von Objekten

Nach Auffassung von Eckes (1991) sind Begriffe Bezeichnungen für Objekte oder Klassen von Objekten. In diesem Sinne verwendet man Begriffe, um Objekte oder Ereignisse der Realwelt identifizieren und unterscheiden zu können. Begriffzuordnungen spielen eine Rolle, wenn jemand ein intuitives Modell oder eine intuitive Prozedur als Grundlage für die Entwicklung eines Programms verwendet. Der erste Schritt des Entwicklungsprozesses ist meist die Belegung von Entitäten oder Einzelaktivitäten mit informatischen Begriffen (z.B. das Beschreiben einer gewissen Anzahl von Dingen einer Kategorie als Liste von Objekten einer Klasse).

1.3.2 Schemata – Speichern struktureller Merkmale

Ein kognitives Schema ist eine Anordnung von Attributen (Slots), die bestimmte Merkmale einer Klasse von Objekten benennen (Anderson 1996). Dabei ist jedem Attribut ein typischer Wert (Default) oder Wertebereich für denkbare Belegungen zugeordnet. Ein Haus z.B. ist ein Gebäude, enthält Zimmer, hat meist einen rechteckigen Grundriss, wird zum Wohnen verwendet, ist aus Holz oder Stein gebaut und hat eine Größe zwischen 10 und 1000 Quadratmetern. Diese typischen Merkmale ergeben das Schema eines Hauses. Schemata gibt es nicht nur für physische Objekte sondern auch für Ereignisse, wie stereotype Handlungsabläufe. Solche Ereignisschemata bezeichnet man als Scripts (Schank & Abelson 1977). Ein Script für einen Restaurantbesuch besteht bei vielen Menschen aus der typischen Abfolge Platz nehmen, Speisekarte lesen, bestellen, essen, bezahlen und gehen (Bower, Black & Turner 1979). Das Schema-Konzept wird in der Kognitionspsychologie vor allem verwendet, um das effizientes Abspeichern von bedeutungsvollen Inhalten im Gedächtnis zu erklären.

Wenn man erkennt, dass ein Objekt zu einer Objektklasse gehört, braucht man sich nur noch Anweichungen von den Default-Werten des Schemas zu merken. Die anderen („typischen“) Merkmale des Objektes kann man aus dem (bereits gespeicherten) Schema ableiten. Brewer und Treyens (1981) haben die Existenz von Schemata durch Erinnerungsexperimente nachgewiesen. Auch beim Pro-

grammieren greift man auf Schemata zurück. So gibt es Schemata über den Aufbau einer rekursiven Funktion (siehe Abschnitt 9.6).

Intuitive Modelle kann man als Schemata betrachten, wenn man Aspekte wie Erinnerbarkeit oder Zuordnung einer Situation (aufgrund von wahrgenommenen Merkmalen) zu einem Modell untersucht. Dies ist dann jedoch eine reduzierte Sichtweise, in der z.B. dynamische Aspekte eines Modells im Hinblick auf eine Problemlösung vernachlässigt werden.

1.3.3 Mentale Modelle

Mentale Modelle sind vereinfachte Repräsentationen realer oder hypothetischer Situationen. Sie erlauben das gedankliche Durchspielen von Abläufen und können so zur Antizipation von Ereignissen im abgebildeten Wirklichkeitsausschnitt verwendet werden. Mentale Modelle enthalten also eine dynamische Komponente. Sie werden für komplexere kognitive Aktivitäten wie Problemlösen und Erklären von Phänomenen der Realwelt verwendet.

Mentale Modelle wurden zum ersten Mal von dem schottischen Psychologen Kenneth Craik (1943) postuliert. Johnson-Laird (1983) erklärt logisches Schließen – insbesondere die Überprüfung der Gültigkeit kategorialer Syllogismen – mit seiner Theorie mentaler Modelle.

Insbesondere im Zusammenhang mit Computern wird gelegentlich zwischen konzeptuellen Modellen und mentalen Modellen differenziert. Während ein mentales Modell eine (eventuell unbewusste) gedankliche Leistung eines Individuums ist – z.B. die Vorstellung eines Computernutzers über die Arbeitsweise eines Computers –, ist ein konzeptuelles Modell von Experten (Lehrer, Wissenschaftler, Softwareentwickler etc.) bewusst gestaltet. Es stellt eine fachlich akzeptable Repräsentation eines Zielsystems dar, d.h. es ist Expertenwissen. Dagegen können mentale Modelle, die Personen im Umgang mit Informatiksystemen heranziehen, auch unangemessen sein und im Widerspruch zum Wissenstand der Fachgemeinschaft stehen (Fehlvorstellung). Konzeptuelle Modelle dienen als Grundlage für das professionelle Design von Softwaresystemen oder die Gestaltung von Medien und Aktivitäten für den Unterricht (Wu et al. 1998). Inhaltlich unterscheiden sich mentale Modelle und konzeptuelle Modelle nicht. Im Unterricht oder in Handbüchern verbreitete konzeptuelle Modelle werden oft von den Rezipienten übernommen und sind dann deren mentale Modelle. Selbst Fehlvorstellungen können von Pädagogen expliziert und in Lehrprozessen zur Abgrenzung vom „Richtigen“ und zur Vermittlung eines vertieften Verständnis eingesetzt werden.

1.4 Merkmale intuitiver Modelle

Fischbein nennt u. a. folgende Merkmale intuitiver Vorstellungen (Fischbein 1987 S. 43 ff): Selbstevidenz (self evidence), Intrinsische Gewissheit (intrinsic certaincy), Dauerhaftigkeit (Persistenz), Zwanghaftigkeit, Theoriestatus, Extrapolativität und Globalität.

Selbstevidenz

Ein intuitives Modell ist unmittelbar einleuchtend und bedarf keiner weiteren Erklärung. Selbstevidenz ist nicht unbedingt eine Folge von Vertrautheit durch Erfahrung. Die bekannte binomische Formel $a^2-b^2 = (a+b)(a-b)$ ist nicht selbstevident, obwohl niemand, der die Sozialisation des schulischen Mathematikunterrichts durchlaufen hat, an der Richtigkeit zweifelt.

Dagegen ist die Aussage „Jede ganze Zahl hat einen Vorgänger“ offensichtlich. Fischbein (1987) stellt heraus, dass eine selbstevidente Aussage unmittelbar etwas mit einem dahinter liegenden mentalen Modell zu tun hat. Sie ist gewissermaßen eine Bedeutungsfacette eines abstrakten Konzeptes. So werden ganze Zahlen über Vorgänger und Nachfolger definiert. Selbstevidente Aussagen sind so etwas wie die Atome von Argumentationen. Jeder Schritt eines nachvollziehbaren Beweises muss selbstevident sein. Anderenfalls könnte er nicht vom Leser akzeptiert werden und müsste in weitere „feinere“ Unterschritte zerteilt werden.

Intrinsische Gewissheit – Konfidenz

Eine Intuition wird vom Subjekt als sicher zutreffend akzeptiert. Gewissheit und Selbstevidenz sind nicht das gleiche. Man kann überzeugt sein, dass z.B. ein mathematischer Satz wie der Satz des Pytha-

goras richtig ist, obwohl er nicht selbstevident ist, sondern eines Beweises bedarf. Andererseits gibt es (empirisch nachgewiesene) Fälle, in denen befragte Personen eine Aussage als selbstevident einschätzen, sich aber bezüglich der Korrektheit dieser Einschätzung unsicher sind (Fischbein 1987). Selbstevident heißt lediglich, dass man keine „Beweismittel" für erforderlich oder gar vorstellbar hält, mit denen die Richtigkeit der Behauptung nachgewiesen kann. Der Begriff Gewissheit (oder Konfidenz) bezieht sich vor allem auf die affektive Komponente einer Intuition. Mit Gewissheit ist die Bereitschaft verbunden, persönliche Nachteile in Kauf zu nehmen, falls die betreffende Aussage sich unerwartet als falsch erweist. Dementsprechend kann Gewissheit z.B. dadurch gemessen werden, dass Probanden sich bereit erklären, Geld zu zahlen, falls sie sich irren. (Fischbein 1987, S. 46, Fischhoff et al. 1977, S. 559–560).

Intuitionen sind häufig trügerisch. Nicht selten vertrauen wir einer Intuition, die sich später als falsch erweist. Fischbein nennt dieses Phänomen Übervertrauen (overconfidence). Bemerkenswert ist, dass Menschen vor allem auf Gebieten, in denen sie sich nicht auskennen, zu Übervertrauen neigen.

Gewissheit ist bei informatischen Problemlösungen – z.B. bei einer Programmentwicklung – von besonderer Bedeutung. Denn mit jeder Programmzeile investiert man Arbeit in das Projekt und geht das Risiko eines semantischen Fehlers ein, der später nur mit großer Mühe lokalisiert und beseitigt werden kann, wenn er denn überhaupt gefunden wird. Ein Programmentwickler, der eine Programmzeile formuliert, verlässt sich also auf seine Intuition. Ohne (subjektive) Gewissheit, dass die gewählte Formulierung richtig ist, würde er oder sie leichtfertig die Korrektheit des Gesamtsystems gefährden und sich potenziell erhebliche Mehrarbeit (also Zusatzkosten) beim Debuggen aufbürden.

Persistenz und Zwanghaftigkeit

Fischbein (1987) ist der Auffassung, dass intuitive Modelle niemals vergessen werden, sondern als „stilles Wissen" erhalten bleiben und mehr oder weniger unbewusst das Denken eines Menschen ein Leben lang beeinflussen, auch wenn er oder sie inzwischen andere (unter Umständen geeignetere) Konzepte gelernt hat. Diese Zwanghaftigkeit und Persistenz intuitiver Modelle wurde oft im Zusammenhang mit Fehlvorstellungen beobachtet. Ein viel diskutiertes Beispiel aus der Physik ist das „Schwungmodell", das Kraft als Ursache von Bewegung beschreibt (Nakamura 1974; Clement 1982; Fischbein 1987, S. 171, S. 176 ff). Dabei handelt es sich um die Vorstellung, dass ein sich bewegender Körper immer langsamer wird und schließlich zum Stillstand kommt, wenn man nicht ständig eine Kraft in Bewegungsrichtung ausübt. Ein sich bewegendes Objekt hat einen Schwung, den es im Laufe der Bewegung allmählich „aufbraucht". Das Modell widerspricht dem ersten Newtonschen Gesetz, wonach Körper (infolge der Trägheit der Masse) ihre Bewegung bis in alle Ewigkeit beibehalten, sofern keine Kräfte auf sie wirken. Doch nach unserer alltäglichen Erfahrung hält keine Bewegung ohne unterstützende Krafteinwirkung unendlich lange an, sondern erstirbt (infolge Reibung) mit der Zeit: Eine Murmel rollt auf einer ebenen Fläche nur ein gewisses Stück weit, ein Boot verliert an Geschwindigkeit, wenn man nicht ständig rudert usw. Clement (1982) beobachtete, dass Studenten der Ingenieurwissenschaften die Kräfte, die auf eine fliegende Münze wirken, mit diesem Schwungmodell beschrieben, obwohl ihnen die Gesetze der Mechanik vertraut waren. Ueno (1993) ist der Auffassung, dass die Robustheit „naiver Erklärungen" eher auf den Einfluss der sozialen Umgebung zurückzuführen ist als auf die Persönlichkeit des Individuums (Ueno 1993, S. 244f.).

Gestaltcharakter

Intuitive Modelle repräsentieren einfache Gedanken. Sie sind in sich geschlossene, holistische Sinneinheiten. Sie haben den Charakter von Gestalten. Der Begriff „Gestalt" wurde seit den 1920iger Jahren durch die Gestalttheorie (Max Wertheimer, Kurt Koffka, Wolfgang Metzger, Rudolph Arnheim) geprägt, die damit menschliche Wahrnehmung erklärt. Nach ihrem Ansatz ist Wahrnehmung keine passive Reizaufnahme sondern ein aktiver, subjektgesteuerter Vorgang. Visueller Input wird als Anordnung von (dem Individuum vertrauten) Gestalten interpretiert.

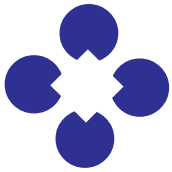


Abb. 1: Gestalt eines Quadrats

So sehen die meisten Menschen in Abbildung 1 ein auf der Spitze stehendes Quadrat, obwohl eine solche Figur nicht explizit umrandet ist. Gewissheit und Selbstevidenz können sich nur auf kohärente Sinngebilde beziehen. Wenn ich mir einer Sache gewiss bin, muss diese Sache auch benennbar und als geschlossene Ganzheit repräsentierbar sein.

Modelle von Softwaresystemen, die während einer Software-Entwicklung entstehen und etwa durch UML-Diagramme repräsentiert werden, sind in der Regel zu komplex, um intuitiv zu sein. Das gleiche gilt für Programmtext. Um ein komplexes System zu verstehen, zu erklären oder zu entwickeln, greifen Menschen häufig – quasi zeitgleich – auf ein ganzes Bündel intuitiver Modelle zurück (Modellcluster). Denn ein einzelnes Modell kann das komplexe Original nur unzureichend abbilden. Ein Problem ist, die verschiedenen Intuitionen scharf zu trennen und die verwendeten Gestalten zu verdeutlichen. Die Interpretation eines Programms ähnelt zuweilen der Betrachtung eines Vexierbildes („Kippbild"), bei dem der gleiche optische Reiz unterschiedliche Gestalt-Wahrnehmungen auslöst.

1.5 Phänomenologische Primitive

Andrea diSessa (1988, 1993, 2001) beschreibt besonders grundlegende intuitive Modelle, die auf häufig erlebten sensorischen Erfahrungen basieren. Er nennt sie phänomenologische Primitive oder kurz p-Prims. Sie sind Basismodelle bzw. Primitive in dem Sinne, dass sie die Grundlage für Erklärungen sind und selbst nicht weiter begründet werden. Das bedeutet, dass ein Modell M für einen Menschen den Charakter eines p-Prims verlieren kann, wenn er noch einfachere, grundlegendere Modelle findet, mit denen er M erklären kann.

Ein Beispiel ist das Konzept der Schwere. Jeder Mensch erlebt bereits in jungen Jahren, dass Gegenstände schwer sind und verallgemeinert diese häufig gemachte Beobachtung z einem selbstverständlichen Merkmal der Welt. Wenn jemand die Tatsache, dass Dinge zu Boden fallen, wenn man sie loslässt, damit begründet, dass sie schwer sind, verwendet er oder sie das p-Prim „Schwere". Für diese Person ist die Schwere von Gegenständen ein unmittelbar akzeptiertes Phänomen, das selbst keiner weiteren Erklärung bedarf.

Ein weiteres Merkmal von p-Prims ist ihre universelle Verwendbarkeit in verschiedenen Lebensbereichen. Die Vorstellung von Widerstand, der nur mit Anstrengung zu überwinden ist (ein von diSessa häufig erwähntes Beispiel), wird sowohl zur Erklärung elektrischer als auch mechanischer Phänomene herangezogen (Ohmsches Gesetz, Reibung etc.).

1.6 Intuitive Modelle und Fehlvorstellungen

In zahlreichen Untersuchungen sind Fehlvorstellungen (misconceptions) von Schülern und Studenten im Zusammenhang mit naturwissenschaftlichen (z.B. Griffith und Preston 1992; Clement 1982; Brown 1992), mathematischen (z.B. Rosnick 1981; Clement, Lochhead, Monk 1981; Fischbein 1987) und informatischen Themen (z.B. Bonar, Soloway 1985; Close, Dicheva 1997; Sleeman, Putman, Baxter, Kuspa 1989; Ginat 2001; Madison Gifford 2002) entdeckt und spezifiziert worden.

Intuitive Modelle werden in verschiedener Hinsicht als Quelle für Fehlvorstellungen gesehen:

- Sie können selbst objektiv falsch sein, weil die sensorische Erfahrung, auf der sie fußen, irreführend war. So ist das bereits erwähnte Schwungkonzept für Bewegungen das Resultat von Alltagserfahrungen mit reibungsbehafteten Bewegungsvorgängen.
- Sie werden auf unzulässige Weise verallgemeinert und in einem Kontext verwendet, in dem sie nicht gelten. So erklären Kindergartenkinder häufig, die Sonne gehe abends unter, weil sie müde

ist. Hier wird das intuitive Modell des Ermüdens auf Objekte der unbelebten Natur angewendet (Anthropomorphismus, Animismus).

Intuitive Modelle, wenn sie denn zu Fehlvorstellungen führen, werden von Pädagogen als Barrieren für den Erwerb wissenschaftlich fundierter Expertenkonzepte gesehen (Champagne, Gunstone und Klopfer 1985, Strike und Posner 1985). Dementsprechend sollten Misconceptions möglichst vermieden werden (z.B. Holland, Griffiths, Woodman 1997). Bereits vorhandene Fehlvorstellungen müssten dann vom Lehrer entdeckt und im Unterricht gezielt aufgegriffen werden. Bei der „Konfrontationsmethode" werden Expertenkonzepte den fehlerhaften Vorstellungen der Schüler/innen gegenüber gestellt. Der Unterricht enthält Demonstrationen und Arrangements, die die vorhandenen Fehlvorstellungen falsifizieren (Gegenbeweise). Als Ziel wird manchmal gesehen, falsche naive Vorstellungen durch Expertenkonzepte zu *ersetzen* (z.B. McClosky 1983, Brown 1992). Smith, diSessa und Roschelle (1994) bezweifeln allerdings, dass man Misconceptions einfach ersetzen kann. Der Begriff „ersetzen" impliziert, dass die falsche Vorstellung ausgelöscht wird. Doch konnte man beobachten, dass auch Experten in bestimmten Situationen auf intuitive Vorstellungen zurückgreifen anstatt abstrakte wissenschaftliche Modelle (durch Formeln beschriebene Gesetze) zu verwenden.

Jean Piaget (2003) hält Fehlvorstellungen für eine „natürliche Randerscheinung" der kognitiven Entwicklung eines Menschen. Er nennt für jede Entwicklungsstufe typische „Denkfehler". Beispielsweise kennen Kinder in der präoperationalen Stufe (Kindergartenalter) noch nicht das Prinzip der Volumenkonstanz. Wenn sie z.B. eine Flüssigkeit von einem breiten niedrigen Glas in ein schlankes hohes Gefäß gießen, glauben sie, es sei mehr geworden, weil der Flüssigkeitsspiegel höher liegt.

Smith, diSessa und Roschelle (1994) sind der Auffassung, dass die Sichtweise der Misconceptionforschung mit dem Konstruktivismus nicht vereinbar ist. Sie sprechen von einem „Lernparadox". Der Konstruktivismus beschreibt Lernen als Weiterentwicklung bereits existierender Wissensstrukturen. Damit müssten eigentlich auch Fehlvorstellungen nützliches Vorwissen sein und nicht etwas, das Dazulernen behindert. Fazit: Nicht der *Inhalt* einer Fehlvorstellung – also das intuitive Modell – ist das Problem sondern seine unangemessene Verwendung.

Jede Fehlvorstellung, die vom Subjekt als solche erkannt worden ist, bedeutet einen Wissenszuwachs. Denn sie dient dazu, die Grenzen der Anwendbarkeit zu definieren. Betrachten wir ein Beispiel aus der Informatik: Angenommen, Sandra stellt sich eine Liste `s=[1, 3, 5, 6]` als Behälter mit drei Fächern vor, in denen sich beschriftete Zettel befinden. Dann verwendet Sandra ein intuitives Modell. Sie ist den Umgang mit einem solchen Behälter gewohnt – vielleicht hat sie als Kind einen solchen Kasten zum Sortieren von Legosteinen verwendet. Für sie ist es z.B. zweifelsfrei klar, was das erste oder letzte Element ist oder was es heißt, das erste mit dem letzten Element zu vertauschen. Für viele Operationen in einem Computerprogramm, das eine solche Liste verarbeitet, ist das Behältermodell perfekt geeignet. Die Änderung des ersten Listenelements durch die Anweisung `s[0] = 2` kann z.B. so dargestellt werden, dass man den Zettel im ersten Fach durch einen anderen Zettel ersetzt. Allerdings gibt es auch Anweisungen, die mit dem Behältermodell nicht dargestellt werden können. Wenn das zweite Element durch die Anweisung `del s[1]` gelöscht wird, bedeutet das z.B. nicht, dass der zweite Zettel aus dem Behälter entfernt wird. Wer das intuitive Modell des Behälters angemessen verwendet, kennt also erstens das Modell und alle relevanten Situationen, in denen man es nicht verwenden darf. D.h. er oder sie kennt potentielle Fehlvorstellungen, die mit dieser Intuition verbunden sind.

2 Repräsentation intuitiver Modelle

In diesem Kapitel richten wir das Augenmerk auf Fragen der Repräsentation eines intuitiven Modells. Wir betrachten ein intuitives Modell als immaterialles gedankliches Konzept, das auf verschiedene Weise repräsentiert werden kann. Die Repräsentation ist physisch existent und der Wahrnehmung durch die Sinne zugänglich. Sie kann aufgeschrieben, gemalt, gedruckt, gefilmt d.h. in irgendeiner Form auf einem physischen Medium gespeichert werden. Nur über eine physische Repräsentation kann ein Modell externalisiert, kommuniziert und archiviert werden (vgl. auch Ueno 1993). Es macht Sinn zwischen dem Modell und seiner physischen Repräsentation zu differenzieren, weil letztere eine gewisse Beliebigkeit und Variabilität aufweist, während das intuitive Modell dauerhaft ist. Anders herum kann man auch sagen, dass die Annahme einer dauerhaften Intuition allein die Schlussfolgerung eines Beobachters ist, der im Verhalten einer Person gewisse Regelmäßigkeiten über die Zeit feststellt. Beobachtbar sind allein physische Repräsentationen wie z.B. ein Bild, das jemand in einer bestimmten Situation zeichnet, um einem Gedanken Ausdruck zu verleihen. Aus mehreren solcher „Äußerungen" schließt ein Beobachter auf dahinter liegende Modellvorstellungen.

Wenn man auch analytisch zwischen Modell und seiner physischen Repräsentation unterscheiden kann, so ist doch beides eng miteinander verwoben. DiSessa (2001) weist auf die Materialität von Intelligenz hin. Ohne geeignete Ausdrucksmittel können intuitive Modelle gar nicht entstehen. Manchmal ist es schwierig die Grenze zwischen Repräsentation und Modell zu ziehen. Ist ein konkretes Beispiel nur eine Repräsentation eines Modells oder ist es bereits das Modell? Es gibt verschiedene Dimensionen, die man bei einer Untersuchung von Repräsentationsformen bedenken kann.

Kodierung. Mit welcher Art von materialen Bedeutungsträgern werden intuitive Modelle dargestellt? Alan Paivio unterscheidet verbale und imaginale Kodierung von Wissen.

Metaphorisierung. Es muss eine geeignete, der menschlichen Vorstellungskraft zugängliche Domäne gefunden werden, aus der ein Modell stammt. Für Auswahl der Domäne ist ein wichtiges Kriterium wie gut man in ihr die gemeinten Modelle physisch repräsentieren kann.

Konkretisierung und Beispielbildung (Exemplarisierung). Die Repräsentation eines Modells ist immer konkret. Aus einer im Prinzip unendlichen Fülle von Möglichkeiten müssen eine oder mehrere repräsentative Exemplare auswählt werden.

Strukturorientierte und prozessorientierte Darstellungen. Eine strukturorientierte Darstellung ist statisch und gibt dauerhafte Aspekte des Gemeinten wieder. So ist ein Struktogramm oder ein Programmtext eine strukturorientierte Repräsentation eines Algorithmus. Ein Protokoll, das die Inhalte von Variablen während eines beispielhaften Laufs darstellt oder eine Animation, das die Veränderungen während eines Programmlaufs visualisiert, ist prozessorientiert. Wenn jemand sich die Arbeitsweise eines Algorithmus über einen Beispielablauf – also prozessorientiert – merkt, heißt das nicht, dass er oder sie nur über prozessuales Wissen dazu verfügt. Eine prozessorientierte Repräsentation ist kein prozedurales Wissen sondern bewusstes deklaratives Wissen über einen Prozess. Aus dem Prozessbeispiel kann in der Regel leicht eine strukturorientierte Darstellung (z.B. ein Programmtext) generiert werden und umgekehrt. Das Prinzip der Vielfalt und Flüchtigkeit (Abschnitt 2.2) gilt auch für diesen Aspekt von Repräsentationen.

2.1 Duale Kodierung von Wissen

In der Theorie der dualen Kodierung (Paivio 1971; 1986) werden zwei grundsätzliche physische Darstellungsformen für Wissensinhalte unterschieden: verbale (sprachliche, begriffliche) und nonverbale (bildhafte, imaginale) Repräsentationen. Beispiele für bildhafte Darstellungen sind Fotos, Zeichnungen, Karten und Diagramme. Sie sind analog, ikonisch und kontinuierlich. Beispiele für sprachliche Repräsentationen sind (natürlichsprachliche) Texte, mathematische Modelle oder Computerprogramme. Sie sind digital, nicht-ikonisch und diskontinuierlich. Paivio nimmt an, dass es für verbal und nonverbal repräsentierte Information getrennte aber gleichwohl kooperierende Verarbeitungssysteme gibt.

Es gibt eine Reihe von empirischen Befunden, die dieses Modell unterstützen: Doppelte Kodierung eines Inhaltes durch sprachliche und bildhafte Repräsentationen erhöht die Erinnerungswahrscheinlichkeit. So werden leicht verständliche und benennbare Bilder (z.B. ein Haus) nach einmaliger Präsentation besser behalten als Wörter. Diesen „Bildüberlegenheitseffekt“ erklärt Paivio damit, dass die Information doppelt repräsentiert wird, nämlich erstens nonverbal als Bild und zweitens verbal durch einen Begriff, der mit dem Bild assoziiert wird. Die Annahme von getrennten Verarbeitungssystemen für verbale und nonverbale Information wird durch hirnphysiologische Befunde unterstützt. EEG-Studien (Ley, 1983) zeigten, dass bei der Verarbeitung von sprachlichem Material vor allem die linke Hirnhälfte aktiv ist, während beide Hirnhälften an der Verarbeitung nonverbaler Information beteiligt sind. Sasse (1997) schlägt vor, sprachliche und bildhafte Repräsentationen als zwei Endpunkte eines Kontinuums zu betrachten. Viele Repräsentationen kombinieren sprachliche und visuelle Komponenten. Eine Zeichnung ist oft leichter verständlich – und repräsentiert das gemeinte Modell – besser, wenn sie einige Wörter oder Symbole enthält.

2.2 Vielfalt und Flüchtigkeit

Inwiefern macht es Sinn zwischen mentalem Modell und physischer Repräsentation zu unterscheiden? Man könnte ja auch sagen, das Bild eines Behälters, der einen Zettel mit einer Zahl enthält, ist bereits ein Modell.

Doch während intuitive Modelle einfache und in der Biographie eines Menschen dauerhafte Denkweisen darstellen, mit denen viele Erscheinungen der Welt erklärt und verstanden werden können, sind Repräsentationen dieser Modelle vielfältig und flüchtig.

Für ein und dasselbe intuitive Modell gibt es im Prinzip beliebig viele unterschiedliche Repräsentationen, von denen je nach Situation nur eine ausgewählt wird. Nehmen wir als Beispiel das Behältermodell für Variablen. Dieses Modell wird in unterschiedlichen Sprachäußerungen verwendet, die sich in der exakten Wortwahl unterscheiden können, aber das gleiche meinen:

„Bei einer Zuweisung geht der alte Inhalt einer Variablen verloren.“

„Bei einer Zuweisung wird der vorige Inhalt zerstört.“

In visuellen Repräsentationen des Behältermodells können Behälter, Namensschilder und Inhalte auf mannigfache Weise dargestellt werden. Als Darstellungsform kann man Zeichnungen, Fotos oder reale Gegenstände verwenden. Inhalte einer Variablen lassen sich auf unterschiedlichen Abstraktionsniveaus visualisieren. Ein Zettel mit einer Zeichenkette oder einer Zahl ist konkreter als die Abbildung eines Gegenstandes, der ein Datum repräsentiert. Man hat die Wahl, einen Wert als Literal („Haus“) oder in Form seiner Ansicht (Haus) wiederzugeben.

Intuitive Modelle werden in Kommunikationsprozessen verwendet und dabei über Repräsentationen externalisiert. Jeder, der eine Idee durch eine Grafik veranschaulicht, versucht zunächst alles möglichst einfach zu halten. Erst wenn die Gesprächspartner Nachfragen haben, werden weitere Details hinzugenommen, verbale Erläuterungen gegeben oder zusätzliche Bilder zur Darstellung spezifischer Aspekte angefertigt. Häufig werden Repräsentationen intuitiver Modelle ad hoc erfunden und auf die Bedürfnisse (Seh- und Hörgewohnheiten) des Adressaten zugeschnitten. Im Schulunterricht denken sich Lehrerinnen und Lehrer immer neue Formulierungen für ein und denselben Gedanken aus bis sie das Gefühl haben, dass sie verstanden werden. Repräsentationen intuitiver Modelle haben oft etwas Flüchtiges. Manche Bilder werden nur einmal verwendet (um einem bestimmten Menschen in einer bestimmten Situation mit bestimmten Medien etwas zu erklären) und häufig kann man sich nach kurzer Zeit nicht mehr an sie erinnern. Das dahinter stehende Konzept ist dagegen dauerhaft. Persistenz ist ja ein Merkmal intuitiver Modelle.

2.3 Repräsentation und Metaphorisierung

2.3.1 Uneigentliche Redeweise in der Informatik

Die Sprache der Informatik ist voller Formulierungen, die in der Rhetorik als Formen des uneigentlichen Sprechens (Tropen) bezeichnet werden (Baumgarten 2005). In einer Metapher (Übertragung) wird der eigentliche Ausdruck (Ziel) durch einen anderen (Quelle) ersetzt, der aus einem anderen

Sachbereich (Domäne) stammt. Eine Metapher ist ein Vergleich ohne ein tertium comparationis, d.h. ohne die Nennung der Hinsicht, in der sich die Begriffe aus Quell- und Zielbereich ähneln. Beispiele:

- Eine Klasse ist ein Bauplan (Quelldomäne ist Architektur).
- Variablen sind Behälter für Daten.

Wenn Metaphern lange genug in der Fachsprache verwendet worden sind, werden sie zu toten Metaphern, d.h. ihre ursprüngliche Bedeutung in der Quelldomäne tritt in den Hintergrund. Beispiele für tote Metaphern aus der Informatik sind Begriffe wie Botschaft, Verzweigung oder Schleife.

Bei Metonymien (Umbenennungen) wird der eigentliche Ausdruck durch einen anderen ersetzt, der aus derselben Domäne stammt. Eine Metonymie in der Alltagssprache ist z.B. die Formulierung „vor den Altar treten“ für „heiraten“. Wenn man unterstellt, dass die Hochzeit in der Kirche stattfindet, stammt der Begriff „Altar“ aus derselben Domäne.

Viele Begriffe in der Informatik sind Metaphern oder Metonymien, für die es keine eigene, eigentliche Bezeichnung gibt. So sagt man, dass eine Funktion ihr Berechnungsergebnis zurückgibt. Damit vergleicht man die Funktion mit einem Menschen, der von einer anderen Person etwas in Empfang nimmt und ihr anschließend etwas zurückgibt. Für diesen Ausdruck gibt es aber keine andere Formulierung. Ja, er wird bei vielen Programmiersprachen sogar in der Syntax als Schlüsselwort verwendet (return). Solche notwendigen Metaphern oder Metonymien bezeichnet man als Katachresen.

Eine Allegorie (Bild) ist eine komplexe Metapher, die einen abstrakten Begriff veranschaulichen soll (z.B. „divide and conquer“). Bei einem Vergleich werden Ausdrücke aus verschiedenen Domänen durch das Partikel *wie* verbunden (z.B. „Instanzen einer Klasse sind wie Häuser, die nach dem gleichen Bauplan gebaut worden sind.“) Ein Vergleich bringt eine gewisse Distanz zwischen dem verwendeten Modell und dem gemeinten Zielkonzept zum Ausdruck.

Man verwendet Tropen – insbesondere Metaphern – um die Wirkung eines Textes auf den Leser oder die Leserin zu erhöhen. Metaphern machen einen Text abwechslungsreich und interessant. Es ist ein Spiel mit der Bedeutung der Wörter. Die Spannung wird erhöht, wenn die Quelldomäne ungewöhnlich ist und thematisch weit von der Zieldomäne entfernt ist.

2.3.2 Metaphern und analoges Denken

In der kognitiven Linguistik löst man sich von dem rhetorischen Aspekt der (oberflächlichen) Redeverbesserung und betrachtet Metaphern als Vorstellungen, die substanzielles Wissen repräsentieren. Boyd (1993) zählt für verschiedene Wissenschaften theoriekonstituierende Metaphern auf. Lakoff und Núñez (1997) beschreiben grundlegende Metaphern (grounding metaphors oder conceptional metaphors) für die Mathematik. Diese Metaphern schlagen eine Brücke zwischen vertrauten Vorstellungswelten und mathematischen Domänen. Beispiele sind „arithmetic is object collection“ oder „arithmetic is object construction”. [2]

Kennzeichnend für diesen Ansatz ist, dass eine Metapher als Abbildung A → B von einer Quelle (base, source) A auf ein Ziel (target) B dargestellt wird. (Man liest eine solche Abbildung „B is A“.) Jede Metapher besteht aus einer Sammlung von „Unterabbildungen“. So gehören zur Metapher „Arithmetic Is Object Collection“ unter anderem folgende Zuordnungen:

- Zahlen sind Kollektionen von physischen Objekten gleicher Größe.
- Arithmetische Operationen sind Akte des Bildens einer Kollektion von Objekten.
- Eine Addition ist das Zusammenlegen zweier Objektkollektionen zu einer größeren Kollektion.

Das Beispiel zeigt, dass die Quelle einer Metapher intuitiv sein muss, wenn sie helfen soll, die Zieldomäne (z.B. Arithmetik) zu verstehen. Nur wenn man sich sicher ist, dass beim Zusammenlegen von zwei Kollektionen keine Objekte verloren gehen und keine aus dem Nichts dazukommen, kann

[2] Daneben gibt es auch verbindende Metaphern (linking metaphors), mit denen Konzepte aus verschiedenen Gebieten der Mathematik gekoppelt werden (Lakoff und Núñez 1997, S.34).

man Additionen verstehen. Intuitionen (subjektiv sicheres Wissen) fungieren somit als Quellkonzepte grundlegender Metaphern.

Eine Analogie ist nicht das gleiche wie eine Metapher. Während der Begriff Metapher aus der Rhetorik stammt und primär ein linguistisches Phänomen beschreibt, ist die Analogie ein Begriff aus der Logik. Analogien beziehen sich auf den Vergleich der Struktur von Systemen. Aristoteles (384–322 v. Chr.) beschreibt „geometrische Analogien" als Gleichheit zweier Verhältnisse in der Form A/B=C/D (A verhält sich zu B wie C zu D). Dabei kann der Bruchstrich als mathematische Operation (z.B. Division) aber auch nicht-mathematisch verstanden werden (Coenen 2002). Beispiel: „Ein Baum verhält sich zu einem Ast wie ein Körper zu einem Arm". In analogem Denken (oder analogem Schlussfolgern) kann Wissen über eine vertraute Quelldomäne auf eine neue noch unbekannte Zieldomäne übertragen werden. So verwendete Galileo sein Wissen über die bereits bekannte Kreisbahn des Mondes um die Erde als Basis für seine Theorie, dass sich auch die Erde bewegt (English 2004).

Ein bekanntes Analogon für Elektrizität ist die Vorstellung von Wasser, das durch Rohre fließt. Beide Systeme haben strukturelle Ähnlichkeit. Man kann Entitäten und Beziehungen des Quellsystems (Modell) auf das Zielsystem abbilden.

Fließendes Wasser entspricht sich bewegenden Elektronen. Der Wasserdruck – z.B. in Folge eines Höhenunterschiedes zwischen Anfang und Ende des Rohres – entspricht der elektrischen Spannung, der Wasserfluss – d.h. die Wassermenge, die pro Zeiteinheit einen Rohrabschnitt passiert – entspricht der Stromstärke.

Zu bedenken ist, dass nur Systeme gleicher Komplexität wirklich analog sein können. Bei intuitiven Modellen in der Informatik spielt aber häufig gerade die Reduktion von Komplexität eine Rolle. Zum Beispiel haben intuitive Modelle, die die Idee eines Algorithmus zum Ausdruck bringen, oft eine andere (einfachere) Struktur als das zugehörige Programm. Eine Programmentwicklung auf der Basis einer intuitiven algorithmischen Idee ist somit strenggenommen kein analoges Denken. Betrachten wir die folgende Iteration (Python):

```
s = ["Monika", "Tim", "Sandra"]
for i in [0, 1, 2]:
    print s[i]
```

Die Ausführung kann man sich anschaulich so vorstellen: Ein Behälter mit drei Fächern enthält mit Namen beschriftete Zettel. Eine Stecknadel springt von Fach zu Fach. Der Text auf dem Zettel in dem mit der Nadel markierten Fach wird auf ein Blatt Papier geschrieben. Dieses intuitive Modell ist keine strukturgleiche Analogie für das Programm. Es ist einfacher und enthält zum Beispiel keine Entität, die der Indexliste `[0, 1, 2]` entspricht. Gleichwohl kann die wandernde Stecknadel als Metapher für einen Namen des aktuellen auszudruckenden Items `s[i]` betrachtet werden.

2.3.3 Metaphern als Bilder für intuitive Modelle

Intuitive Modelle können mehr oder weniger abstrakt sein. DiSessas p-Prims sind besonders abstrakte Intuitionen und können deshalb in vielen unterschiedlichen Situationen angewendet werden. DiSessa weist darauf hin, dass diese Basisintuitionen nur sehr schwer in Worte gekleidet werden können. Im Zusammenhang mit der Erklärung von zusammengesetzten Bewegungen identifiziert diSessa drei relevante p-Prims: Verstärkung (reinforcement), Kompromiss (compromise) und gegenseitige Auslöschung (canceling) (diSessa 2001, S. 190ff). Die Intuition „canceling" spielt zum Beispiel eine Rolle, wenn man die Bewegung einer Person, die in einem langsam fahrenden Zug gegen die Fahrtrichtung läuft, beschreibt.

Die gleiche Intuition wird aber auch verwendet, wenn man Additionen mit negativen Zahlen verstehen will. Chiu (1996, 2001) hat in mehreren Untersuchungen festgestellt, dass Kinder und Erwachsene bei mathematischen Problemlösungen in ein und demselben Kontext mehrere unterschiedliche Metaphern quasi parallel verwenden. Dabei verfügen Erwachsene über ein reichhaltigeres Repertoire an Metaphern als Kinder, äußern diese aber nicht spontan während des Problemlösens sondern erst bei Interviews, wenn sie veranlasst werden, einen mathematischen Zusammenhang zu erklären.

Was veranlasst Menschen, zwischen unterschiedlichen Domänen hin und her zu springen? Warum äußern Erwachsene bei einer Problemlösung spontan fast keine intuitiven Modelle? Eine Erklärung ist, dass sie eigentlich *ein* einheitliches abstraktes Konzept im Sinn haben, das intern durch unterschiedliche konkretere Modelle repräsentiert wird (vgl. Abb. 2).

Problemlösen mit Hilfe von Metaphern lässt sich anscheinend eher als zweistufiger kognitiver Prozess verstehen. Primär hat man eine abstrakte Idee parat, die gleich durch ein ganzes Bündel von intuitiven Modellen aus verschiedenen Domänen repräsentiert wird. Erst im Nachhinein (z.B. bei kniffligen Detailfragen und wenn man jemandem etwas erklären muss) vergewissert man sich der Brauchbarkeit und Angemessenheit der abstrakten Idee und repräsentiert sie durch Metaphern.

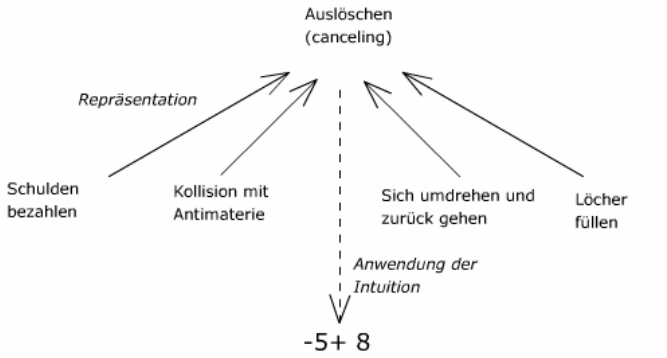


Abb. 2: Problemlösen mit dem p-Prim „Auslöschen“

Menschen greifen also innerhalb eines Kontextes auf unterschiedliche Vorstellungswelten zurück. Bei manchen bildhaften Darstellungen ist es sogar schwierig eine Domäne zu benennen, aus der sie stammen könnten. Ein Beispiel dafür ist ein Bild aus konzentrischen Bögen, das ein Schüler gezeichnet hat, der die Summe einiger Zahlen der Folge 5, 8, 11 … berechnen sollte (Presmeg 1997, S. 269). Die Zeichnung visualisiert die Formel:

$\text{summe} = n/2 * (s_1+s_n)$

Jeder Bogen stellt die (konstante) Summe zweier Folgenglieder s_i und s_{n+1-i} dar.

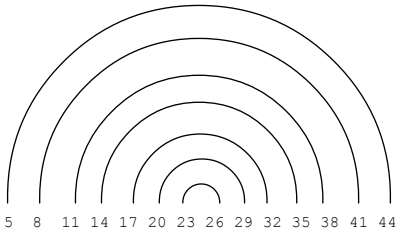


Abb. 3: Visualisierung der Idee eines Rechenalgorithmus zur Berechnung der Summe von Gliedern einer Zahlenfolge. Nach: Presmeg 1997

Der Schüler bezeichnete das Bild als „dome“ (Gewölbe), ein anderer Schüler, dem die Zeichnung vorgelegt wurde, sah darin einen Regenbogen. Man könnte also behaupten, die beiden Schüler hätten Architektur und optische Naturphänomene als Quelle (source) für eine Metaphorisierung verwendet. Allerdings: Ihre Aussagekraft gewinnen die Bögen offenbar nicht durch den Sinnkontext innerhalb der Quelldomäne (sonst wäre sie nicht so einfach austauschbar). Vielmehr ist zu vermuten, dass die Benennung der Abbildung (z.B. als Gewölbe) zwar ein effizientes Speichern ermöglicht (im Sinne der doppelten Kodierung), die erklärende Kraft aber ganz abstrakt aus der Visualisierung von symmetrisch angeordneten Zahlenpaaren durch Bögen gewonnen wird. Das heißt die Domäne ist für das Verstehen unwichtig.

Von Bedeutung werden Domänen, wenn es sich um konzeptionelle Metaphern im Sinne von Lakoff und Johnson (1980) handelt. In ihrer Theorie der konzeptionellen Metapher sehen sie Metaphorisierung als Mechanismus zum Aufbau konzeptuellen Wissens.

Norma Presmeg verwendet den Begriff des Vehikels (vehicle) für lehrreiche Metaphern (1997). Ein abstraktes Konzept (der Tenor) wird mit Hilfe einer Metapher (Vehikel) erklärt. Die Metapher ist in diesem Fall eine Lernhilfe und eröffnet einen Zugang zu Konzepten, der ohne sie verschlossen bliebe. Merkmal dieses Metapherntyps ist, dass die Konzepte der Quelldomäne intuitiv verständlich sind, nicht aber die Konzepte der Zieldomäne.

In der didaktischen Literatur werden für den Unterricht in verschiedenen Fächern Vorstellungswelten abgezirkelt, in denen lehrreiche Aktivitäten stattfinden. Dazu kann man auch die Domänen der von Lakoff (1997) zusammengestellten Metaphern aus dem Bereich der Mathematik zählen. Nehmen wir als Beispiel „arithmetic is motion along a line“. Für den Mathematikunterricht gibt es ausgearbeitete Unterrichtsreihen auf der Basis der Domäne „Bewegung entlang einer Linie“. Die Kinder verwenden einen Ausschnitt der Zahlengeraden mit dem Nullpunkt in der Mitte und führen damit verschiedene Aktivitäten aus, die Rechenoperationen mit ganzen Zahlen entsprechen (z.B. eine gewisse Schrittzahl nach links gehen bedeutet subtrahieren.) Nach der Theorie der konzeptionellen Metaphern bleiben die Kinder zunächst ganz in der Welt der „Bewegung entlang einer Linie“ und erwerben Konzepte, die sie später verwenden um arithmetische Operationen mit ganzen Zahlen zu verstehen.

Vorstellungswelten können detailliert gestaltet und z.B. als multimediale Software oder mechanisches Spielzeug implementiert werden. Man spricht dann von Mikrowelten (microworlds). Mikrowelten sind explorative Lernumgebungen (Schulmeister 2002). Es sind künstliche Welten mit sehr einfachen Regeln, in denen die Benutzer sich frei bewegen und eigenaktiv vom Designer (Pädagogen) „verstecktes“ Wissen entdecken können. Das wohl bekannteste und historisch erste Beispiel einer computerbasierten Mikrowelt ist die Turtle-Grafik der Programmiersprache Logo (Papert 1980). Die Bedeutung von Mikrowelten im Hinblick auf die Repräsentation intuitiver Modelle wird in Anhang 1.5 diskutiert.

2.4 Beispiele als Repräsentationen intuitiver Modelle

Metaphorisierung ist ein Prinzip, das bei der Repräsentation abstrakter Konzepte Anwendung findet. Die Hauptleistung liegt hier in der Auswahl eines geeigneten Sachbereichs (Domäne) und eines Modells innerhalb dieser Domäne.

Ein zweites Prinzip der Repräsentation ist die Beispielbildung. Eine physisch existente Repräsentation eines intuitiven Modells – auch eine Metapher – ist immer beispielhaft. Um ein abstraktes Konzept bildhaft darzustellen, muss man konkret werden und eine von unendlich vielen denkbaren Darstellungen auswählen. Aber es gibt auch Beispiele ohne Metaphorisierung, deren Repräsentation aus der gleichen Domäne stammt wie das Konzept, das abgebildet wird. In der Rhetorik nennt man sie Metonyme. So ist `[1, 2, 3]` ein Metonym für eine Python-Liste.

2.4.1 Prinzipien der Beispielbildung

Einfachheit. Nicht jedes Beispiel repräsentiert ein intuitives Modell. Wenn es zu komplex ist, kann man es sich nicht mehr als eine kohärente Gestalt vorstellen (z.B. komplexe Multiliste als Beispiel einer Liste). Ein Beispiel kann aber auch so stark reduziert sein, dass es nur einen Sonderfall repräsentiert (leere Liste, Liste mit drei gleichen Elementen etc.). Dennoch kann ein solches für sich alleine nicht repräsentatives Beispiel eine wichtige Rolle in einer Kollektion von Beispielen spielen.

Reichhaltigkeit. DiSessa sieht Reichhaltigkeit (richness) als Merkmal phänomenologischer Primitive. Unter Reichhaltigkeit verstehen wir hier die Eigenschaft einer Intuition, mehrere unterschiedliche Facetten eines abstrakten Konzepts wiederzugeben. Offenbar ist Reichhaltigkeit mit Komplexität verbunden. Die beiden Begriffe meinen aber nicht genau das gleiche. Hinter dem Terminus „reichhaltig“ steckt auch der Gedanke, dass ein reichhaltiges Modell stärker zu kognitiven Aktivititäten inspiriert. Es bietet mehr Anknüpfungspunkte für die interne Vernetzung mit anderen Konzepten. Ein triviales Beispiel repräsentiert ein allgemeines Prinzip häufig schlechter als ein Beispiel, das eine gewisse Komplexität besitzt. So wird von Schülern eine Liste mit drei Elementen eher als typisches Beispiel einer Liste gesehen als eine Liste mit nur einem oder gar keinem Element. Ein sehr einfacher regulärer Ausdruck wie `"a"` wird seltener als Merkbeispiel für reguläre Ausdrücke gewählt als `".*[bB]all"`. Letzterer verrät mehr über die Möglichkeiten der Konstruktion regulärer Ausdrücke

(Platzhalter, Wiederholung) und ist insofern reichhaltiger. Ein weiteres Beispiel für Modelle unterschiedlicher Reichhaltigkeit sind folgende zwei Metaphern für die Wirkungsweise regulärer Ausdrücke:

- Ein Sieb, das passende Zeichenketten von unpassenden trennt, und
- ein Kran, der passende Zeichenketten (dargestellt durch Karten mit unregelmäßig geformter Oberkante) nach dem Schloss-Schlüssel-Prinzip an ihrer „äußeren Form" erkennt.

Das Sieb visualisiert allein das Prinzip der Trennung, nämlich die Tatsache, dass mit regulären Ausdrücken bestimmte Zeichenketten aus der Menge aller möglichen Zeichenketten „herausgefischt" werden können. Das zweite Modell ist reichhaltiger. Denn zusätzlich zum Trennungsprinzip als solchem wird auch der Mechanismus der Trennung – das Schloss-Schlüssel-Prinzip – angedeutet.

Anwendungsmöglichkeit. Ein Beispiel, das man sich als intuitives Modell für ein abstraktes Konzept merkt, kann mehr oder weniger eine typische Anwendungsmöglichkeit sein. Fischbein spricht dann von einer pragmatischen Intuition. Versteht man Anwendungsmöglichkeiten als Bestandteil eines Konzeptes, so hat ein anwendungsbezogenes Modell einen größeren semantischen Wert als ein nicht anwendungsbezogenes. So kann man sich eine Python-Liste als Warteliste beim Arzt mit den Namen von Patienten vorstellen `["Maier", "Schmidt", "Schulz"]`. Eine typische Verwendung von Listen ist, gleichartige Objekte in einem Container zusammen zu fassen. Das Kriterium der typischen Anwendung kann in Konflikt mit dem Kriterium der Reichhaltigkeit treten. So geht aus obigem Beispiel nicht hervor, dass bei Python Listen auch Objekte unterschiedlicher Klassen enthalten dürfen.

Repräsentativität. Beispiele können mehr oder weniger repräsentativ für eine abstrakte Vorstellung sein. Kahneman &Tversky (1982) stellten fest, dass Schüler bei der Einschätzung von Wahrscheinlichkeiten sich davon irritieren lassen wie sehr eine gegebene Häufigkeitsverteilung ihrem Bild einer typischen zufälligen Zahlenfolge entspricht. Regelmäßige Zahlenfolgen wie `(4, 4, 4, 4, 4)` hält man für „weniger zufällig" als Folgen aus ungleichen Zahlen. Vorstellungen über die Repräsentativität der Repräsentation eines intuitiven Modells können eine kognitive Barriere darstellen und Problemlösungen behindern.

2.4.2 Prototypische Beispiele

Manchmal genügt ein einziges Beispiel zur effizienten Repräsentation eines Konzeptes. Ein solches prototypisches Beispiel kann kürzer formuliert und offenbar besser behalten werden als eine abstrakte, allgemeingültige Erklärung (Beispiel in Anhang 1.6). In Kalkulationstabellen haben Berechnungsformeln den Charakter prototypischer Beispiele. In der Formel `= A2 + B2` sind die Bezeichner `A2` und `B2` Beispiele für Referenzen auf andere Zellen der Tabelle, deren Bedeutung sich dem Leser erst durch Betrachtung der Gesamttabelle erschließt. Beim Kopieren der Formel greift das System quasi auf die abstrakte Bedeutung zurück und berechnet die Referenzen neu.

2.4.3 Repräsentation durch eine Beispielkollektion

In Kurzreferenzen, die auf wenig Platz (z.B. eine Karte im Format eines großen Lesezeichens) möglichst viel Information zu einer Programmiersprache bieten müssen, werden abstrakte Konzepte häufig allein durch konkrete Beispiele wiedergegeben. Effiziente Beispielkollektionen enthalten meist Prototypen mit besonders hoher Repräsentativität und Sonderfälle.

Wenn in einem Programmierprojekt eine Funktion oder Methode definiert werden soll, kann ihre Funktionalität durch beispielhafte Testfälle definiert werden. Jeder Testfall besteht aus einem Aufruf der Funktion mit bestimmten Argumenten und dem erwarteten Ergebnis. Die Kollektion der Testfälle repräsentiert ein Modell der Funktionalität des zu entwickelnden Programmstücks. [3]

[3] Zietsman & Clement (1997) zeigen, dass leicht nachvollziehbare Extremfall-Beispiele zum Aufbau abstrakter Modelle führen können.

2.4.4 Beispiel-basiertes Problemlösen

Beispiele sind nicht in jedem Fall Repräsentationen eines intuitiven Modells. Sie können auch direkt – ohne den „Umweg“ über ein abstraktes intuitives Modell – als Muster für eine Problemlösung verwendet werden. Beim Beispiel-basierten Problemlösen (vgl. Chiu, 2001) verwendet man ein Fallbeispiel, das mit der aktuellen Problemsituation strukturell identisch ist. Der Problemlöser ersetzt Entitäten des bekannten Beispiels durch passende Entitäten des Fallbeispiels und wendet das bekannte Lösungsmuster an. Betrachten wir eine typische Textaufgabe mit Lösung aus einem Mathematikbuch (nach Chiu 2001):

Ein Zug fährt 300 Kilometer nach Köln. Er fährt mit 150 Stundenkilometern. Wie lange braucht er bis er Köln erreicht.

Lösung: 300 km / 150 km/h = 2 Stunden

Durch Beispiel-basiertes Problemlösen kann man leicht Aufgaben wie die folgende lösen:

Anna fährt mit 60 Stundenkilometern. Wie lange braucht sie um das Haus ihrer Tante zu erreichen, das 180 km entfernt ist?

Hier braucht man nur in der Musterlösung andere Zahlen für Geschwindigkeit und Entfernung einsetzen und kann damit das neue Problem lösen. Dabei ist nicht einmal ein wirkliches Verständnis der Begriffe Entfernung und Geschwindigkeit erforderlich. Die passenden Zahlen können durch eine oberflächliche Analyse des Aufgabentextes ermittelt werden: Hinter der ersten Zahl muss das Wort „Kilometer“ oder „km“ stehen, hinter der zweiten die Zeichenkette „km/h“. Problemlösung durch oberflächliches Nachmachen – bei dem man sich nur an äußeren Merkmalen der Repräsentation orientiert ohne das dahinter liegende Konzept zu verstehen – kann schnell zu Fehlern führen, wie Anderson am Beispiel mathematischer Beweise zeigt (Anderson 1996, S. 242 f.).

2.5 Strukturorientierte und ablauforientierte Repräsentationen

In einer strukturorientierten Repräsentation eines (intuitiven) Modells werden Entitäten und Beziehungen zwischen ihnen darstellt. Beispiele sind UML-Objekt- und Klassendiagramme, Flussdiagramme oder Geschäftsprozessdiagramme. In dynamischen Visualisierungen werden Beziehungen zwischen Entitäten in Form kleiner Dramen durch typische Aktivitäten (Interaktionen) veranschaulicht. Eine strukturorientierte Repräsentation einer rekursiven Funktion ist z.B. eine Animation auf der Basis von Execution Frames, die durch Kästen visualisiert werden. Mit jedem (rekursiven) Aufruf der Funktion erscheint ein neuer Kasten. Er ist eine aktive Entität, die mit anderen Entitäten interagiert. Sobald er seine Aufgabe erledigt hat, verschwindet er wieder.

Das Beispiel deutet an, dass strukturorientierte Repräsentationen von Programmläufen schnell komplex werden können und dann schlecht memoriert werden können. Was man sich merken kann ist die Grundstruktur oder ein Grundprinzip, aus dem man bei Bedarf in Gedanken oder mit Bleistift und Papier eine detailreichere Version rekonstruieren kann.

In der Standardliteratur zu Algorithmik (z.B. Horowitz & Sahni 1981, Wirth 1986) werden zur Veranschaulichung der Arbeitsweise (und Komplexität) von Algorithmen häufig Abbildungen verwendet, die keine Struktur aus Entitäten und Beziehungen wiedergeben, sondern in irgendeiner Form ein Protokoll der Ausführung des Algorithmus darstellen. Solche Repräsentationen nennen wir ablauforientiert. Abb. 4 zeigt einen Entscheidungsbaum für die binäre Suche nach einem Element mit Schlüssel k in einem sortierten Array.

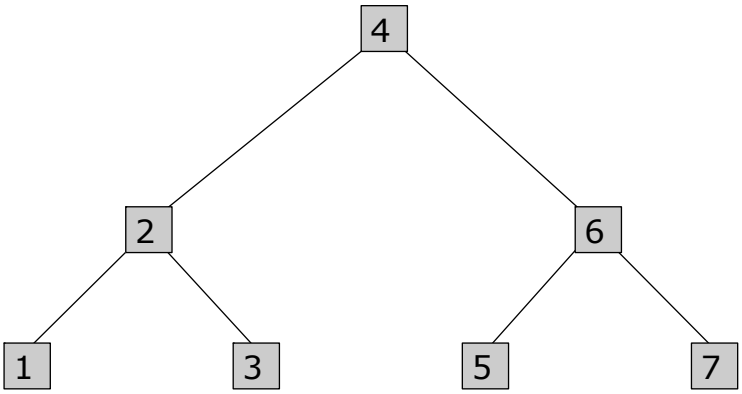


Abb. 4: Entscheidungsbaum für die binäre Suche

Jeder Knoten repräsentiert eine Entscheidung für die Weitersuche. Die Suche beginnt in der Mitte des Arrays. Wenn k kleiner ist als der Schlüssel des mittleren Elementes (hier: 4), sucht man links weiter. Wenn k größer ist, sucht man rechts weiter, sonst hat man das gesuchte Element bereits gefunden. Ein Pfad von der Wurzel zu einem Blatt repräsentiert die Entscheidungen bei einer einzelnen Suche. Der gesamte Baum repräsentiert alle denkbaren (binären) Suchprozesse. Wenn eine ablauforientierte Darstellung eine gut einprägsame Gestalt hat (wie der Entscheidungsbaum), wird man dazu tendieren, sie zur Repräsentation der algorithmischen Idee zu verwenden. In einer Problemlösungssituation muss man jedoch aus dem ablauforientierten Modell ein strukturorientiertes Modell konstruieren und letztlich zu einem Programmtext verfeinern. Typische Beispiele für ablauforientierte Repräsentationen sind auch Figuren, die durch rekursive Logo-Programme entstehen (siehe Anhang 2.7).

2.6 Modellrepräsentationen im Unterricht

Modellrepräsentationen können auf vielfache Weise im Unterricht verwendet werden. Begleitend zu den Workshops mit der PVS wurden 20 Lehrerinnen und Lehrer befragt, welche Formen der Visualisierung sie im Unterricht verwenden (Fragebogen und Details in Anhang 1.9). Das Befragungsergebnis (Abb. 5) lässt vermuten, dass Visualisierungen wie in der Python Visual Sandbox im Informatikunterricht an deutschen Schulen eine eher geringe Rolle spielen. Lediglich UML-Klassendiagramme, Zeiger, Wertetabellen und Kästchen, in die der momentane Wert einer Variablen eingetragen wird, werden im Mittel „gelegentlich“ (entspricht dem numerischen Wert 2) oder häufiger verwendet.

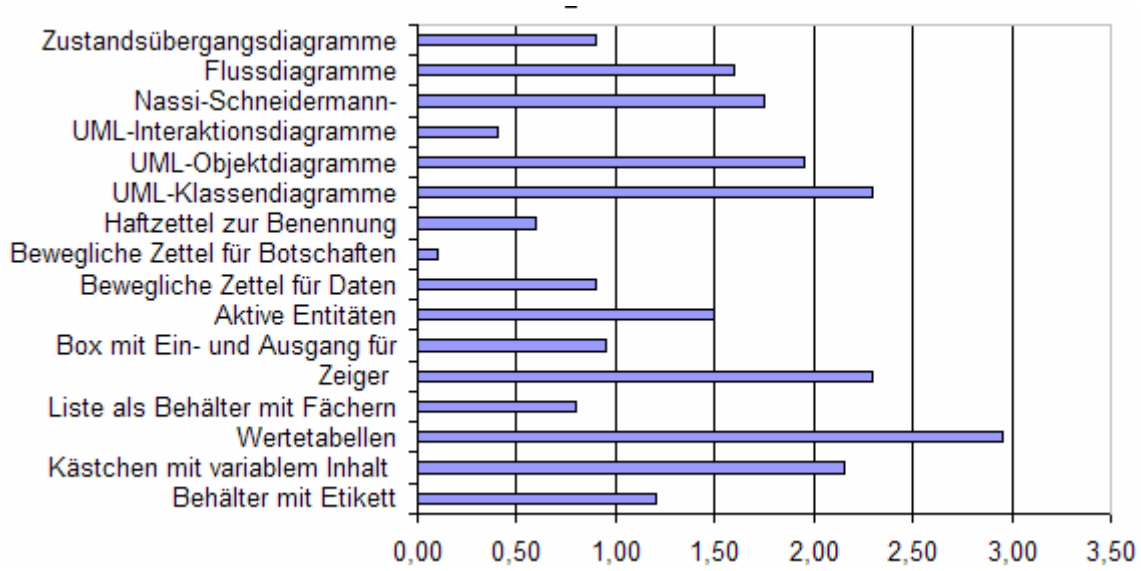


Abb. 5: Ergebnis einer Umfrage zur Häufigkeit der Verwendung von Visualisierungen im Informatikunterricht (n = 20). Dargestellt werden die Mittelwerte der numerisch codierten Antworten (0: nie, 1: selten, 2: gelegentlich, 3: häufig, 4: immer, wenn es passt). Weitere Details im Anhang 1.9.

3 Verwendung intuitiver Modelle

Welche Rolle spielen intuitive Modelle bei Softwareentwicklungen im Klassenraum? Zu welchen Zwecken werden sie verwendet? In diesem Kapitel legen wir das Augenmerk auf Verstehen, Erklären, Problemlösen und Kontrollieren.

Verstehen findet z.B. statt, wenn ein Schüler in einer Sprachreferenz die Beschreibung einer Funktion nachliest und aus abstrakten Darstellungen (semantische) Intuitionen entwickelt.

Erklären ist besonders bei gemeinsamer Softwareentwicklung im Team von Bedeutung. Erklärungen finden sich in Programmkommentaren, Dokumentationen oder Diskussionen des Programmierteams. Dabei werden intern verwendete intuitive Modelle externalisiert, geeignete Repräsentationen gefunden und die Darstellung auf das Wesentliche fokussiert.

Problemlösen ist die zentrale Aktivität bei einer Programmentwicklung. Die Suche nach einer Lösung wird häufig durch semantische Intuitionen beeinflusst. Wer z.B. eine Routine zum Sortieren einer Liste entwickelt, wird sich zunächst mit der Frage beschäftigen, was eine Liste eigentlich ist und verschiedene Modellvorstellungen ins Bewusstsein rufen, die die Bedeutung des Konzeptes „Liste“ repräsentieren. Die Lösungsidee selbst wird – in ihrer Gesamtheit – durch ein antizipatorisches intuitives Modell dargestellt.

Kontrollieren mit Hilfe intuitiver Modelle spielt beim Testen und Debuggen eines Programms eine Rolle. Bei der Suche nach semantischen Fehlern oder der Formulierung von Zusicherungen (z.B. in Form von assert–Statements) werden einfache Modelle mit besonders hoher subjektiver Gewissheit zu Plausibilitätsprüfung verwendet.

3.1 Verstehen

Wie werden intuitive Modelle verwendet, um zu einem Verständnis abstrakter Konzepte der Informatik zu gelangen? Fischbein nennt solche intuitiven Vorstellungen, die sich auf die Bedeutung eines Konzeptes beziehen, semantische Intuitionen (Fischbein 1987, S. 59 ff). Zum Beispiel gibt es für das Konzept einer geraden Linie zwischen zwei Punkten verschiedene semantische Intuitionen:

- Geometrisches Modell: Linie, die man mit einem Lineal zwischen zwei Punkten zieht.
- Pragmatisches Modell: kürzester Weg zwischen zwei Orten.
- Physikalisches Modell: Lichtstrahl von einer Lichtquelle zu einem beleuchteten Objekt.

Nun werden in der Informatik Konzepte meist durch mehr oder weniger formale Texte definiert. Typische Textdokumente für Informatiker sind Programmquelltexte, Kommentare in Programmtexten, Sprachreferenzen oder allgemeine Lehrtexte, in denen grundlegenden Ideen der Informatik erklärt werden. Im Lernalltag eines Informatik-Schülers gibt es verschiedene Anlässe für Verständnisgewinnung:

- Erklärungen der Lehrperson etwa zur Arbeitsweise einer Funktion werden von Schülern individuell nachvollzogen. Im Unterricht kann man beobachten, dass Schüler nach einem Informationsinput der Lehrperson beispielhafte Programmzeilen, die z.B. an der Tafel stehen, ausprobieren und abwandeln um das soeben Gehörte zu verarbeiten.
- In einer Diskussion des Programmierteams (z.B. zum Entwurf eines Programms) verwendet ein Kommunikationspartner einen Begriff, der den anderen nicht ausreichend klar sind. Die Diskussion weicht nun von ihrer ursprünglichen Zielrichtung ab und wendet sich nun für eine gewisse Zeit der Beseitigung der Verständnisschwierigkeit zu.
- Bei der Suche nach einem semantischen Fehler in einem Programmtext, kommt bei einer Schülerin oder einem Schüler der Verdacht auf, ein im Programm verwendetes Konzept (z.B. die Wirkungsweise einer Funktion) nicht richtig verstanden zu haben. Er oder sie unterbricht die Fehlersuche, widmet sich dem kritischen Konzept und versucht es, besser zu verstehen und Zweifel auszuräumen.

- Bei einer Programmentwicklung hat jemand die Idee, ein bestimmtes Programmierkonzept zu verwenden – z.B. eine bestimmte Standardfunktion der jeweiligen Programmiersprache – stellt aber fest, dass er oder sie zu wenig über relevante Details des Konzeptes weiß. Er hat Zweifel, dass sein Verständnis in Tiefe und Weite ausreicht und versucht – etwa durch Studium der Sprachreferenz oder kleine Testprogramme – sein Verständnis zu verbessern.
- Ein existierendes Programm soll weiterentwickelt werden. Um die notwendigen lokalen Änderungen vornehmen zu können, muss man zuerst das gesamte Programm verstanden haben.

3.1.1 Verstehen aus Sicht der Hermeneutik

Weil Texte häufig die Grundlage für Verständnisgewinnung sind, hat die Bedeutungs- und Sinnerfassung von Texten für Verstehensprozesse einen besonderen Stellenwert. Fragen des Textverstehens werden in der allgemeinen oder philosophischen Hermeneutik diskutiert. „Verstehen“ wird in der Hermeneutik seit Heidegger nicht als bloße Methode sondern als „Grundzug des Mensch-Seins“ beschrieben (Capurro 1989 S. 16). Der Mensch ist hinsichtlich seiner Verhaltensmöglichkeiten offen, die Grundlage jedes Handels ist das Verstehen der Welt, in der er sich bewegt.

Der hermeneutische Zirkel – ein Konzept, das aus der antiken Rhetorik stammt – beschreibt die Auslegung eines Textes als im Prinzip unendlichen Prozess der Auseinandersetzung mit einem Text (Capurro 1989). Das Ganze eines Textes wird aus den Einzelheiten und die Bedeutung der Einzelheiten aus dem Ganzen erfasst. Verständnis bleibt immer vorläufig und wird bei jedem „Durchlauf“ des Zirkels in Frage gestellt.

Der Begriff „Text“ ist hier im weitesten Sinne zu verstehen. Auch formale und grafische Darstellungen, in denen mathematische oder grafische Symbole verwendet werden, zählen dazu. Das entscheidende Merkmal des Textes ist, dass er „überliefert“ wird. Er ist intersubjektiv verständlich und (bei fachlichen Texten) Teil der Kultur der Fachgemeinschaft, in der sich das lesende Subjekt befindet.

In gewissem Sinne ist das Verständniskonzept der Hermeneutik dem Intuitionsbegriff von Fischbein entgegengesetzt. Die Hermeneutik betont die Unsicherheit des Verstehens. Die Auseinandersetzung mit dem Text ist niemals abgeschlossen. Dagegen ist eine intuitive Vorstellung von Selbstevidenz und subjektiver Gewissheit geprägt. Ein zweiter Unterschied betrifft die Persistenz. Intuitionen halten ein Leben lang. Nach Auffassung von Fischbein und diSessa kann man sich nicht von ihnen „befreien“. Sie sind Teil der Persönlichkeit. Dagegen wird Verstehen im Verlauf des hermeneutischen Zyklus‘ immer aufs Neue geprüft und gegebenenfalls revidiert. Das betrifft das Menschenbild der Hermeneutik: Es ist Teil der Offenheit und Freiheit des Menschen, sein Verständnis von den Dingen weiterentwickeln zu können. Man kann diesen Widerspruch zwischen Dauerhaftigkeit und Wandelbarbeit lösen, indem man folgendes annimmt: Einmal erworbene Intuitionen bleiben als solche erhalten – aber der Umgang mit ihnen, die Art und Weise, wie man sie bei der Interpretation fachlicher Texte anwendet oder dem Verstehen abstrakter Konzepte verwendet, kann sich ändern.

3.1.2 Verstehen durch intuitive Modelle

Für den Verstehensprozess sind intuitive Modelle in mehrerer Hinsicht von Bedeutung. Manchmal werden neue intuitive Modelle zur Repräsentation des im Text dargestellten Konzeptes geschaffen. „Neu“ kann zweierlei bedeuten: Der Leser erfindet selbst ein Modell, oder aber er wählt ein gegebenes Modell als Repräsentanten für das abstrakte Konzept. Das Modell wird aber erst zur Intuition, wenn es vom Subjekt als „gewiss richtig“ akzeptiert wird. Dies ist kann ein relativ langwieriger Prozess sein, der mit Erfahrungen zu tun hat. Solche Erfahrungen können das Ergebnis von Experimenten sein (z.B. Ausprobieren von Programm-Statements am Computer). Eine schnelle Form des Verstehens findet statt, wenn im Bewusstsein des Lesers bereits vorhandene Intuitionen mit dem neuen Konzept in Beziehung gesetzt werden. So kann die „Idee“ regulärer Ausdrücke durch das „Schloss-Schlüssel-Prinzip“ modelliert werden, ein Konzept, das man aus anderen Zusammenhängen kennt.

Beim erkenntnisgewinnenden Fragen werden (neue oder alte) intuitive Modelle dahingehend geprüft, ob sie geeignete Interpretationen des im Text dokumentierten Konzeptes sind. Dabei werden unpassende Intuitionen ausgegrenzt. Das kann dazu führen, dass man zu abstrakten Konzepten auch unpassende Modelle expliziert und sich dauerhaft merkt. So versteht man das Konzept der Länge einer

Liste (Anzahl der Elemente) besser, wenn man es von der physikalischen Länge eines Gegenstandes unterscheidet.

Meist wird ein Sachverhalt nicht durch ein einzelnes intuitives Modell sondern durch ein Bündel verschiedener Intuitionen verstanden. Manchmal gibt es ein „treffendes“ globales Modell, das eher vage ist aber die „Grundidee“ in einer einzigen Gestalt wiedergibt. Dazu existieren weitere „unterstützende“ Modelle, die einzelne Spezifika stärker herausstellen aber nicht mehr so repräsentativ für das gesamte Konzept sind. Häufig macht es Sinn, sich auch Spezialfälle und Ausnahmen zu merken. So gehört zum richtigen Verständnis einer Liste dazu, dass es auch leere Listen gibt.

Ein spezieller Fall ist das Verstehen konkreter Programmtexte. Untersuchungen auf diesem Gebiet gehen von der Annahme aus, dass der Leser mentale Modelle zum Programmtext konstruiert (Aschwander & Crosby 2006, von Mayrhauser & Vans 1994, Storey et al. 1997). Dabei können unterschiedliche Vorgehensweisen beobachtet werden. Bei einer bottom-up-Strategie geht der Leser vom Programmtext aus und fasst mehrere Programmzeilen zu abstrakteren Konzepten zusammen (d.h. in unserer Redeweise bildet er sie durch passende intuitive Modelle ab). Oder aber er hat bereits bestimmte intuitive Vorstellungen, wie das Programm funktionieren könnte und versucht sie beim Lesen des Programmtextes zu verifizieren (top-down).

3.2 Erklären

Erklären hat viel mit Verstehen zu tun. Man kann anderen Leuten nur etwas erklären, das man selbst verstanden hat. Erklären heißt, das eigene Verständnis, die eigene Interpretation eines Sachverhaltes anderen Menschen mitzuteilen. So verraten Erklärungen auch etwas über das Verständnis des Erklärenden. Im Unterschied zum Verstehen spielt beim Erklären das Problem der Vermittlung eine Rolle. Damit ist verbunden, dass man sich auf einen Adressaten und dessen Wissenshorizont einstellt. Zwar findet auch jedes Verstehen im „Miteinander in einer gemeinsamen Welt“ statt (Capurro 1989, S. 100 ff), aber die Zielrichtung des Verstehens ist gewissermaßen „egoistisch“, nämlich die Erweiterung des eigenen Horizontes.

Die Modelle, die dem Verständnis dienen, zielen auf eine „globale Sicht“. Das Konzept muss insgesamt richtig verstanden werden. Beim Erklären ist häufig das gezielte Ausgrenzen bestimmter ausgewählter Aspekte wichtig (Fokussierung). Entsprechend können Erklärungsmodelle sehr spezifisch sein. Das Bemühen um Verstehen geht in alle Richtungen. Neue Konzepte werden möglichst vielfältig mit bekannten Konzepten vernetzt und haben zudem Auswirkungen auf die vorhandenen mentalen Vorstellungen. Die Hermeneutik spricht von einer Horizonterweiterung des Subjekts. Erklären dagegen ist gerichtet. Es geht darum, gezielt bestimmte Aspekte durch geeignete Intuitionen darzustellen. Anlass einer Erklärung kann eine Frage sein, die von außen kommt und die es zu beantworten gilt. Dagegen ist Verstehen das (immer vorläufige) Ergebnis einer selbstgesteuerten Auseinandersetzung.

Erklärungen können Teil einer Verständnisgewinnung sein. In diesem Fall ist der Adressat der Erklärung die eigene Person. Man macht sich dann selbst einen Zusammenhang klar und versucht Gewissheit zu erlangen.

Typische Anlässe für Erklärungen mit intuitiven Modellen im Klassenraum sind:

- Darstellung der Arbeitsweise eines Programmstücks in einem Kommentar oder im Rahmen von Diskussionen des Programmierteams.
- Bei der Suche nach einem semantischen Fehler muss erklärt werden, warum das Programm eine bestimmte (nicht korrekte) Ausgabe liefert.
- Erklärungen in einer Bedienungsanleitung zu einem selbst geschriebenen Programm.

3.2.1 Fokussierung

Ein typisches Merkmal von Erklärungen ist, dass sie von einem komplexen zu erklärenden Gegenstand einzelne Aspekte herausgreifen und durch unmittelbar einleuchtende intuitive Modelle verständlich machen. Dabei treten andere Aspekte des Gegenstandes – die freilich für ein Gesamtverständnis ebenso wichtig sind – in den Hintergrund. Sie werden ignoriert oder eventuell sogar falsch dargestellt. Wir nennen diese Facette des Erklärens Fokussierung.

Modelle können sich im Fokussierungsgrad unterscheiden. Stark fokussierte Modelle werden im Kontext einer Erklärung eher in Form eines (Distanz wahrenden) Vergleiches (und nicht als Metapher) verwendet. Zur Verdeutlichung des Geheimnisprinzips in der OOP werden Objekte manchmal mit einer Festung verglichen: Im Inneren befinden sich (wie ein Schatz) die Attribute. Außen liegen schützend wie eine Burgmauer die Methoden, die den Zugriff von außen kontrollieren (siehe Balzert 1999, S. 18). Ein solches Modell veranschaulicht den Schutzaspektes, den eine Methode haben kann, ist aber für die Erklärung anderer Aspekte des Objektkonzeptes gänzlich ungeeignet.

Fokussieren ist eine Form der Vereinfachung. Ein intuitives mentales Modell muss einfach sein, um in gedanklichen Prozessen leicht verarbeitet werden zu können. Vereinfachung geht immer auf Kosten von Wirklichkeitsnähe. Von mehreren verfügbaren intuitiven Modellen wählen wir in der Regel das einfachste, d.h. dasjenige, das für die gegebene Problemstellung gerade eben noch ausreicht.

Betrachten wir als Beispiel Teilchenmodelle aus der Chemie. Zur Erklärung der unterschiedlichen Dichte von Gasen und Flüssigkeiten reicht ein einfaches Kugelteilchenmodell (große Abstände zwischen den Teilchen im gasförmigen Zustand und kleine Abstände im flüssigen Zustand). Um aber Anziehungskräfte zwischen den kleinsten Teilchen zu erklären, verwendet man ein komplexeres Modell, bei dem der Aufbau der Moleküle aus Atomen berücksichtigt wird.

Wie in den Naturwissenschaften ist Fokussierung auch in der Informatik ein wichtiger Aspekt von Erklärprozessen. Abb. 6 zeigt einen Ausschnitt aus einem animierten Modell, das die Arbeitsweise der folgenden Iteration (Python) erklärt:

```
for i in [1, 5, 4, 3, 2]:
    print i**2
```

Die Liste wird durch einen Kasten mit mehreren Fächern repräsentiert. Aus dem Kasten werden nacheinander Zettel entnommen, jeweils die darauf stehende Zahl quadriert und ein Zettel mit dem Ergebnis an eine Tafel, die die Standardausgabe darstellt, geheftet. Die Entnahme der Zettel ist suggestiv, da in jedem Moment deutlich wird, welches Listenelement als nächstes „dran ist“. Die Wiederholung ist beendet, wenn der Kasten leer ist, also keine weiteren Listenelemente mehr zur Verfügung stehen. Die Vorstellung der Entnahme von Elementen entspricht aber nicht dem Listenkonzept, da die Liste bei der Iteration nicht verändert wird. Wer diese Animation als Erklärung wählt, verzichtet auf Realitätsnähe zugunsten einer pointierten Darstellung der Idee der Iteration (Fokussierung).

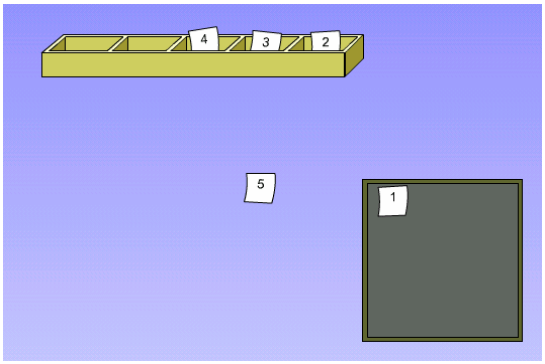


Abb. 6: Intuitive Modellierung einer Iteration über eine Liste durch Entnahme von Items.

Das Beispiel illustriert, dass Fokussierung – wie jede Vereinfachung – eine potenzielle Quelle für Fehlvorstellungen ist. Somit ist das Risiko von Fehlvorstellungen eine normale Begleiterscheinung von Erklärungen. Es ist prinzipiell nicht zu vermeiden – auch nicht durch geschickte Wahl von Modellen. Wie diSessa (2001) und Smith, diSessa und Rochelle (1994) hervorheben, liegt das Problem nicht in den Unzulänglichkeiten des intuitiven Modells selbst. Ein intuitives Modell, das in einem bestimmten Zusammenhang hervorragend für eine Erklärung geeignet ist, kann in einem anderen Kontext irreführend sein.

Das Rezept zur Vermeidung von Fehlvorstellungen durch fokussierte Modelle kann nun nicht sein, auf Vereinfachungen bei Erklärungen zu verzichten. Stattdessen müssen mehrere Erklärungsversuche mit unterschiedlichen Intuitionen vollzogen werden. Genau das liegt ja gerade im Wesen von fokussie-

renden Erklärungen: Der zu erklärende Gegenstand wird aus unterschiedlichen Perspektiven betrachtet. Jeder Blick ist begrenzt und dieser Begrenztheit muss man sich bewusst sein. Auf diesen Punkt kommen wir nun im folgenden Abschnitt zu sprechen.

3.2.2 Mehrperspektivität: Viele Modelle für eine Sache

Erklärungen sind immer Bestandteil einer Kommunikation. Ein Erklärungsversuch muss also auf den Verstehenshorizont des Gegenübers abgestimmt sein. Diese Überlegung macht zwei Probleme sichtbar:

- Es kann immer nur vermutet werden, welche intuitiven Modelle der Kommunikationspartner versteht, d.h. welche Konzepte für sie oder ihn intuitiv einleuchtend sind.
- Dokumentierte Erklärungsversuche (etwa Kommentare in Programmlistings oder Erklärungen in Bedienungsanleitungen) wenden sich an ein heterogenes Auditorium, also an Personen mit völlig unterschiedlichen Vorkenntnissen.

Beide Probleme legen die gleiche Schlussfolgerung nahe: Für Erklärungen ist es in der Regel notwendig, unterschiedliche intuitive Modelle für die gleiche Sache zu verwenden. Darüber hinaus kann es sinnvoll sein, für die Darstellung eines Modells unterschiedliche Repräsentationen zu verwenden. Dabei ist es aber kaum zu entscheiden, ob zwei unterschiedliche Darstellungen, die das gleiche Modell meinen, im Grunde nicht doch zumindest in Nuancen unterschiedliche intuitive Modelle repräsentieren, ohne dass sich der Erklärende dessen bewusst ist.

Professionelle Erklärer halten für eine Wissensdomäne eine breite Palette unterschiedlicher intuitiver Modelle bereit, auf die sie situationsspezifisch zurückgreifen (vgl. auch Chiu 2001, Ueno 1993). Bei einer Erklärung im Rahmen einer bidirektionalen Kommunikation (z.B. im Schulunterricht) wird der Erklärende (im Idealfall) so lange immer wieder neue Modelle anbieten, bis er merkt, dass er einen »Augenöffner« gefunden hat und dem Gesprächspartner die Erklärung einleuchtet. Dabei kann es notwendig sein, kreativ zu werden und ad hoc völlig neue intuitive Modelle zu erfinden.

3.3 Problemlösen

3.3.1 Antizipatorische Intuitionen

In vielen Problemlöseprozessen gibt es einen Moment der „Erleuchtung“. Nach einer Phase des Suchens und gedanklichen Durchspielens verschiedener Möglichkeiten, erscheint vor dem geistigen Auge plötzlich die Vision, wie das Problem im Prinzip gelöst werden könnte. Der französische Mathematiker Poincaré berichtete unter anderem folgendes Beispiel:

„Eines Morgens, als ich auf dem Kliff spazieren ging, kam mir innerhalb kürzester Zeit mit einer ungeheuren Plötzlichkeit und sofortigen Gewissheit der Gedanke, dass die arithmetischen Transformationen unbestimmter quadratischer Formen mit den Formen der nichteuklidischen Geometrie identisch waren.“ (Poincaré 1929, S. 388, zit. nach Anderson 1996, S. 263)

Fischbein nennt dieses Phänomen eine antizipatorische Intuition. Sie repräsentiert als zusammenhängende Gestalt die entscheidende Idee einer Problemlösung. Der Begriff antizipatorisch bezieht sich auf die Rolle des Modells in einem Problemlösungsprozess („Vorwegnahme“ der Lösung) und nicht auf den Inhalt. In der Informatik gibt eine Reihe intuitiver Modelle, die jeweils die grundsätzliche Idee eines relativ komplexen Algorithmus verkörpern. Nehmen wir die klassischen Sortierverfahren als Beispiel:

Bubblesort basiert auf der Vorstellung, dass in einer aufsteigend sortierten Liste der linke Nachbar eines jeden Elementes kleiner ist, sofern ein linker Nachbar existiert. Der Blick konzentriert sich auf eine Stelle der Liste. Ein Element wird mit seinem linken Nachbarn verglichen. Falls dieser größer ist, tauschen die beiden Elemente die Plätze. Wenn das genügend oft geschieht, ist die Liste sortiert. Auch wenn man zur Beschreibung der Idee mehrere Sätze braucht, so kann man sie sich dennoch als ein abstraktes zusammenhängendes Ganzes vorstellen.

Das Beispiel zeigt, dass antizipatorische Intuitionen durch semantische Intuitionen inspiriert und unterstützt werden können (vgl. Fischbein 1987). Bubblesort basiert auf dem Gedanken, dass in einer

aufsteigend sortierten Liste der linke Nachbar eines Elementes kleiner ist. Das ist zunächst eine semantische Intuition, die das Konzept der sortierten Liste verständlich macht, aber noch in keinem Zusammenhang mit einer Problemlösung steht. Die antizipatorische Intuition des Sortierens durch Auswahl (straight selection) wird dagegen durch das (semantische) Konzept des Minimums gestützt. In einer sortierten Liste $[s_0, s_1, \ldots, s_n]$ ist jedes Element s_i das Minimum der Teil-Liste $[s_i, s_{i+1}, \ldots, s_n]$.

3.3.2 Ansatz und Verfeinerung

John Anderson stellt den Problemlösungsprozess als Suche in einem Suchbaum von Handlungsmöglichkeiten dar (Anderson 1996, S. 233ff). Beim Problemlösen wird ein Ziel in Teilziele zerlegt, für deren Erreichen der oder die Problemlösende Operatoren besitzt. Die Anwendung eines Operators überführt das Problem von einem Problemzustand in einen neuen Problemzustand, der (hoffentlich) dem Zielzustand (Lösung) näher liegt. Der gesamte Lösungsweg (Folge von Operatoranwendungen) kann durch einen Pfad von der Wurzel (Anfangszustand) zu einem Blatt (Zielzustand) dargestellt werden (Abb. 8 links).

Das Modell des „blinden“ Suchens in einem Suchbaum ist insofern unrealistisch, als es bereits bei einfachen Problemsituationen eine unüberschaubar große Anzahl von Aktionsfolgen gibt, die in Gedanken zu prüfen sind. Im Grunde geht man davon aus, dass nur solche Operatoren in Betracht gezogen werden, die in irgendeiner Beziehung zu einer Lösungsidee stehen.

Der „erste Schritt“, die Bewegung von der Ausgangsituation (Wurzel des Suchbaumes) zu einem ersten Zwischenziel, ist mehr als nur die Auswahl eines Operators. Es ist im Grunde die Entscheidung für einen kompletten Lösungsansatz (eine antizipatorische Intuition), der dann in späteren Überlegungen nur verfeinert, d.h. in eine Folge konkreter Operationen überführt werden muss.

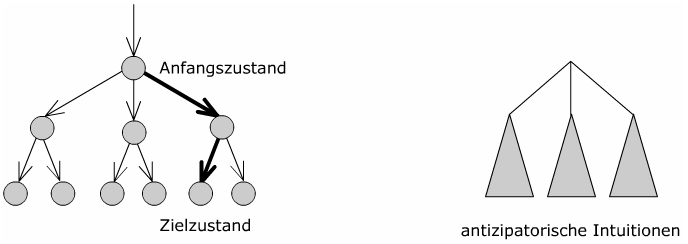


Abb. 7: Suchbaum und geschachtelte antizipatorische Intuitionen

Spohrer, Soloway und Pope (1989) verwenden zur Modellierung von Problemlösungen Und-Oder-Bäume, die sie GAP-Trees nennen (GAP = goal and plan). Ein GAP- Tree repräsentiert sämtliche Lösungsmöglichkeiten einer Aufgabe, die man sich als oberstes Ziel vorstellt (siehe Abb. 8). Zu einem Ziel gehören in der Regel mehrere alternative Pläne, über deren Realisierung das Ziel erreicht werden kann. Jeder dieser Pläne führt wiederum zu mehreren Zielen, die erreicht werden müssen, damit der Plan umgesetzt wird. Auf jedem Pfad von der Wurzel zu einem Blatt folgen immer abwechselnd Ziel- und Plan-Knoten. Bei einer konkreten Implementierung des Projekts (Lösung) wird von mehreren alternativen Plänen zu einem Ziel einer ausgewählt (oder-Verknüpfung). Die Teilziele, die zu einem Plan gehören, müssen aber alle erreicht werden (und-Verknüpfung). Es ergibt sich ein Teilbaum des GAP-Trees mit genau einem Plan pro (Sub-)Ziel. Einen solchen Teilbaum des aus der Aufgabenstellung abgeleiteten GAP-Trees erhält man z.B. durch Analyse eines Programms, das die Aufgabe löst (in der Abbildung durch dicke Linien und dunkle Knoten dargestellt).

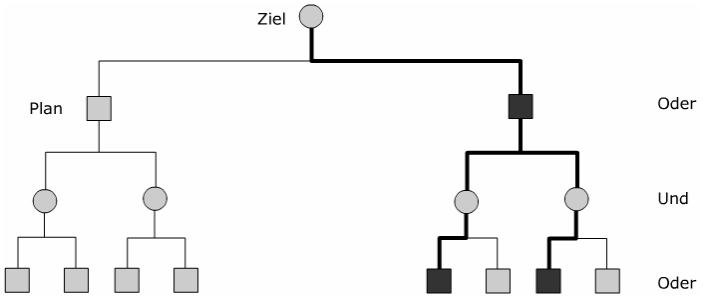


Abb. 8: Aufbau eine GAP-Trees und eines Lösungsbaums (dunkel)

Wenn jemand sich für einen von mehreren alternativen Plänen entscheidet, geht er oder sie erstens von seiner Realisierbarkeit aus und betrachtet ihn zweitens als benennbares geschlossenes Ganzes. Jeder Plan ist mit einem Ziel verbunden. Durch die Existenz eines Plans wird ein Ziel zum erreichbaren Ziel. Hinter dem Plan steht ein intuitives Modell, eine antizipatorische Intuition, die eine Lösungsidee repräsentiert (also die Lösung vorwegnimmt). Man braucht einen Plan, um überhaupt mit dem Programmieren anfangen zu können. Bei strukturierter Programmierweise wird häufig eine verbale Bezeichnung eines Ziels oder Plans zur Benennung von Funktionen oder Klassen verwendet (z.B. eine Funktion namens `eingabe()` realisiert den Plan, Möglichkeiten zur interaktiven Eingabe von Daten zu schaffen).

GAP-Trees sind ein Instrument der Analyse. Sie geben aber nicht unbedingt den Problemlösungsprozess wieder. Spohrer, Soloway und Pope (1989) untersuchten die ersten lauffähigen (also syntaktisch korrekten) Versionen von Programmen, die Anfänger zu verschiedenen Aufgaben geschrieben haben. Sie beobachteten, dass die Teilnehmer zuerst vereinfachte Versionen schrieben und Teilziele wegließen. Die fehlenden Elemente wurden als „missing"-Fehler registriert. Dies kann man so interpretieren, dass sie sich Gewissheit verschaffen wollten, dass ihr (als Ganzheit gedachter) Plan realisierbar ist. Diese ersten Versuche kann man also als Teil einer Antizipation der Lösung – verbunden mit einer Reduktion auf gewisse essentielle Elemente – sehen.

3.3.3 Paradigmatische Modelle und Software-Entwicklung

Intuitive Modelle können eine Art „Muster" für eine Problemlösung darstellen. Das ist eine etwas andere Rolle als Antizipation, bei der die Frage der grundsätzlichen Idee und Realisierbarkeit im Vordergrund steht. Fischbein nennt sie paradigmatische Modelle. Paradigmatische Modelle sind häufig (konkrete) Beispiele, die eine ganze Klasse von Problemlösungen repräsentieren. Paradigmatische Modelle spielen in der Software-Entwicklung eine große Rolle. Ein seit der Frühzeit der Informatik verwendetes Modell ist das „EVA-Prinzip". Viele Programme sind nach dem Muster Eingabe-Verarbeitung-Ausgabe gestaltet. Ein solches Programm enthält Eingabe-Funktionen zur Aufnahme von Daten, verarbeitet diese und gibt über Ausgabefunktionen das Ergebnis zurück. Die ersten interaktiven Programme, die Schüler schreiben folgen diesem Paradigma. Dieses Muster verwendet man auch in anderen Zusammenhängen:

- Arbeitsweise einer Funktion: Eine Funktion nimmt über Argumente Daten auf (Eingabe), verarbeitet diese und gibt mittels einer return-Anweisung das Berechnungsergebnis zurück (Ausgabe).
- Arbeitsweise eines cgi-Skriptes: Ein cgi-Skript verwendet einen Querystring oder ein besonderes Paket mit Daten als Eingabe, verarbeitet die enthaltenen Daten und liefert dem Kommunikationspartner (Client) die Beschreibung einer HTML-Seite als Ausgabe.

Nicht nur für Programmier-Anfänger sondern auch und vor allem für professionelle Programmentwickler gibt es einen reichhaltigen Fundus an Literatur in Form von Sprachreferenzen, Cookbooks, Pattern-Sammlungen etc. mit Hilfen zur Problemlösung. Nun wird man dort niemals ein Beispiel finden, das haargenau auf das eigene aktuelle Programmieraufgabe passt. All diese Bücher und Websites enthalten letztlich paradigmatische Modelle, die abstrakter sind als eine konkrete Lösung. Sie werden meist in Form von Programmbeispielen dargeboten. Dieser Programmtext ist dann aber nicht als pra-

xistaugliche Lösung gemeint. Vielmehr handelt es sich um ein (mehr oder weniger gelungenes) bewusst reduziertes, verständliches und auf eine Lösungsidee zugespitztes Beispiel.

3.3.4 Entwurfsmuster (Design Patterns)

Paradigmatische Modelle finden sich in der Informatik auf vielen Ebenen. Auf hohem Abstraktionsniveau versucht man strukturierte Sammlungen von so genannten Software Patterns oder Design Patterns anzulegen (Gamma, Helm, Johnson, Vlissides, 1995). Patterns werden als Musterlösungen für häufig vorkommende Problemstellungen verstanden („a solution to a recurring problem in a context"). Das Ziel ist, die Wiederverwendung von Softwarelösungen zu ermöglichen. Riehle & Züllighoven (1996) definieren den Begriff Pattern etwas abstrakter. Patterns sind in ihrer Sichtweise nicht allein auf Programmtexte beschränkt, sondern können auch der Analyse von Wirklichkeitsausschnitten dienen. Ein Beispiel für ein Pattern, das sich z.B. in Bürosoftware findet, ist die Unterscheidung von Materialien und Werkzeugen, die man auf die Materialien anwenden kann. Typische Materialien sind Textdokumente oder Terminkalender. Typische Werkzeuge sind z.B. Editoren. Die Unterscheidung von Material und Werkzeug dient nun sowohl der Analyse eines Büros (Welche realen Werkzeuge und Materialien werden in einem realen Büro verwendet? Welche Aufgaben werden mit den Werkzeugen durch Anwendung auf Materialien gelöst?) als auch der Entwicklung einer Bürosoftware (Klassen für Werkzeuge und Materialien). Freilich ist ein paradigmatisches Modell nicht dasselbe wie ein Pattern. Die Unterschiede sind folgende:

- Ein Pattern ist elaboriert und expliziert. Es ist in irgendeiner Form (z.B. durch einen Text, UML-Diagramm oder Programmcode) schriftlich fixiert. Dagegen ist ein paradigmatisches Modell Sache eines Individuums. Es wird – wie alle intuitiven Modelle – individuell unterschiedlich, manchmal auch vielfältig repräsentiert und kann unter Umständen sogar unbewusst sein. Das heißt, ein Pattern ist mehr als ein paradigmatisches Modell.
- Ein Pattern dient letztlich der Wiederverwendung von Programmcode in Softwareprojekten. Die Absicht, Ressourcen zu sparen, spielt jedoch bei paradigmatischen Modellen eigentlich keine Rolle. Ein paradigmatisches Modell verkörpert zwar eine Lösungsidee für eine Problemlösung. Diese ist aber so unspezifisch, dass man eher von Anwenden und nicht von Wiederverwenden redet. Das impliziert, dass nicht jedes paradigmatische Modell ein Pattern verkörpert.

Patterns gewinnen ihren Wert für Problemlösungen aus ihrem intuitiven Charakter, also daraus, dass sie ein zugrunde liegendes intuitives Modell besitzen. Das rasch erfassbare intuitive Modell erleichtert den Zugriff. Man kann blitzschnell den Katalog bekannter Patterns durchstöbern und ein passendes Muster finden – sofern es existiert. Oder anders herum: Ein Pattern, dem das Intuitive fehlt, ist nicht so gut geeignet, weil man während eines Problemlösungsprozesses gar nicht darauf kommt, dass es auf einen bestimmten Anwendungsfall passt.

Was macht ein Pattern intuitiv? Cooper (1998, S. 12) weist darauf hin, dass Patterns nicht einfach so erfunden werden. Sie werden eher entdeckt als geschrieben. („In fact, most such patterns are rather discovered than written."). Riehle und Züllighoven betonen, dass Patterns aus der Erfahrung erwachsen. Man kann sie also als bewährte Muster zur Strukturierung der Wirklichkeit betrachten. Manche Patterns (oder genauer: ihre Modelle) ähneln in dieser Hinsicht diSessas phänomenologischen Primitiven. Sie sind fest verwurzelt in persönlicher Lebenserfahrung. Das Material-Werkzeug-Pattern ist ein Konzept, das in sehr vielen Alltagskontexten auftaucht. Kinder lernen es bereits in jungen Jahren, z.B. wenn sie das Werkzeug „Buntstift" auf verschiedene Materialien wie Malblöcke, Tapeten, Tischoberflächen oder Haut ausprobieren.

3.3.5 Use Cases – intuitive Modelle für Funktionalität

Eine Software-Entwicklung, die dem objektorientierten Paradigma folgt, beginnt mit einer objektorientierten Analyse (vgl. z.B. Balzert 1999). Während dieser Phase werden zunächst Geschäftsprozesse (Use Cases) festgelegt, die Systemfunktionalität beschreiben. In einem Geschäftsprozessdiagramm werden Akteure festgelegt, denen bestimmte Geschäftsprozesse zugeordnet werden. Bei einem System zur Verwaltung einer Bibliothek sind typische Akteure:

Leser: Er kann typischerweise im Bestand der Bibliothek nach Büchern suchen, feststellen ob sie ausgeliehen sind etc.

Sachbearbeiter der Ausleihstelle: Er kann Bücher als ausgeliehen oder zurückgegeben vermerken, Beschädigungen dokumentieren etc.

Jeder Akteur ist ein intuitives Modell eines Bündels von zusammengehörigen Aufgaben. Allein die Gestalt eines Akteurs liefert eine ziemlich gute Vorstellung von den damit verbundenen Aufgaben. Diese Akteur-Intuitionen reichen bereits als Grundlage für vollständige Erfassung aller vom System zu leistenden Aufgaben. Das heißt: Wenn man ein sinnvolles System zueinander passender Akteure gefunden hat, besitzt man bereits einen Überblick über die gesamte Funktionalität des geplanten Programms.

3.3.6 Intuitive Modelle im agilen Programmieren – The Planning Game

Bei Software-Entwicklungen arbeiten häufig Personen mit unterschiedlichem technischem Hintergrundwissen zusammen. Das gilt insbesondere für das agile Programmieren (z.B. Extreme Programming, Beck 1999). Hier fühlen sich Softwareentwickler (Developers) und Kunden (Customers) gemeinsam für den Entwicklungsfortschritt verantwortlich. Aufgrund der Heterogenität der Gruppe spielt die Kommunikation intuitiver Modelle eine besondere Rolle. In regelmäßigen Abständen trifft sich das Team zu einem Planning Game. Die Entwicklung beginnt mit einem (einmaligen) Release Planning. Im ersten Schritt wird das Gesamtsystem durch ein einfaches intuitives Modell beschrieben („Metapher“ genannt). Dieses globale Modell wird nun entfaltet, indem die Customers in sogenannten Stories umgangssprachlich die Aufgaben des Systems beschreiben. Dabei zerlegen sie eine globale und diffuse Aufgabe in mehrere konkretere Teilaufgaben. Dies ist insofern ein Problemlösungsprozess, als die Summe der Teilaufgaben einen Weg zur Lösung der Gesamtaufgabe darstellt. Hinter jeder Story steckt ein intuitives Modell, das innerhalb des Teams kommuniziert werden muss. Dieses Modell wird in Gesprächen (negotiations) zwischen Kunden und Entwicklern geklärt. Ein wichtiger Punkt in diesen Verhandlungen ist die Sicherstellung der richtigen Komplexität. Eine Story muss einfach sein. Man muss sie in wenigen Worten beschreiben können. Ist sie zu komplex, wird sie gespalten. Für jede Story spezifizieren die Kunden einen Set von Anwendungstests, die das implementierte Programmfeature am Ende der Iteration bestehen muss (Cohn 2004, S. 24 ff.). Diese (von Nicht-Informatikern entworfenen) Tests haben nicht so sehr den Charakter softwaretechnischer Korrektheitstests, bei denen systematisch das korrekte Systemverhalten in allen möglichen Situationen geprüft wird. Vielmehr operationalisieren und präzisieren sie das intuitive Modell der Story durch Beispiele.

Damit die Entwickler im Team die Kosten für die Implementierung einer Story einschätzen können, benötigen sie eine Lösungsidee, also ein paradigmatisches Modell. Die Kosten ermitteln sie auf der Basis eines Vergleichs mit Lösungen, deren Kosten sie kennen. Falls im Repertoire des Teams kein brauchbares Modell verfügbar ist, wird zunächst eine Erkundungsstory geschrieben (Investigation Story), die schnell (zumindest sicher in der nächsten Iteration) implementiert werden kann (Spike Solution). Erst nach Auswertung eines solchen Experimentes wird dann die eigentliche Story in Angriff genommen. Bemerkenswert ist folgender Aspekt: In jedem Fall arbeiten die Entwickler mit paradigmatischen Modellen, die sie gut kennen und mit denen sie schon mehrfach gearbeitet haben, so dass sie den Zeitaufwand für eine Implementierung einschätzen können. Sie haben nur Modelle und keine Lösungen – denn sonst gäbe es ja nichts mehr zu programmieren. Die Schwierigkeit im Planning Game liegt also darin, geeignete paradigmatische Modelle zu finden.

3.4 Kontrollmodelle

Intuitive Modelle sind klein und unmittelbar einleuchtend. Sie können verwendet werden, um eine komplexere und schlecht durchschaubare Problemlösung auf Korrektheit zu prüfen. Sie bilden dann eine Art „Alarmsystem“, das anschlägt, wenn man einen Fehler gemacht hat. Ein Beispiel aus der Mathematik wird von Fischbein erwähnt (1987, S. 40).

Aufgabe: „0,65 l Fruchtsaft kosten 2 Dollar. Was ist dann der Preis für einen Liter?“

Das Problem ist hier, eine geeignete arithmetische Operation zu finden, um aus den gegebenen Zahlenwerten die Lösung zu berechnen. In diesem Fall müssen 2 Dollar durch 0,65 Liter dividiert werden um den Literpreis zu erhalten.

Als Kontrolle kann das folgende einfache intuitive Modell dienen: Man stellt sich eine kleinere Flasche mit 0,65 l und eine größere Flasche mit 1 l Fruchtsaft vor. Zweifellos muss die größere Literflasche teurer sein als die kleinere Flasche.

Das Kontrollmodell ist einfacher als das vollständige Lösungsmodell. Es erlaubt nur Aussagen über bestimmte Merkmale der Lösung (Literpreis ist größer als 2 Dollar), ist aber kein Korrektheitsbeweis. Allerdings können intuitive Kontrollmodelle „Bausteine“ eines Korrektheitsbeweises sein. Die Zuversicht in die Korrektheit einer Lösung steigt, wenn sie mehreren Kontrollen durch unterschiedliche intuitive Modelle standhält. Die ist vergleichbar mit der „Bewährung einer Theorie“ in der Popperschen „Logik der Forschung“ (Popper 1934). Die Einbettung eines Modells in ein Kontrollsystem von einfachen intuitiven Modellen kann man als Verstehensprozess auffassen. Je mehr Kontrollmodelle ich verwende, desto besser ist das Erlernte verankert.

Im Zusammenhang von Programmierprojekten verwendet man Kontrollmodelle bei der Suche nach semantischen Fehlern (Debuggen), Testen und beim Einbau von Zusicherungen in den Programmtext. Wir gehen im folgenden Abschnitt auf intuitive Modelle zur Formulierung von Zusicherungen ein.

3.4.1 Zusicherungen

Bei einer Softwareentwicklung werden intuitive Kontrollmodelle verwendet, wenn man in einen Programmtext Zusicherungen einbaut. Das sind Bedingungen, die während des Programmlaufs auf Gültigkeit geprüft werden. Falls die Bedingung nicht gilt, wird das Programm mit einer Fehlermeldung abgebrochen. Mit Zusicherungen kann ein Programm „logisch sicher“ gemacht werden. Die Technik ähnelt der Kontrolle einer Maschine durch Sensoren und Messgeräte, die etwaige Störungen anzeigen. So findet man auf dem Armaturenbrett eines Autos Warnleuchten, die z.B. Ölmangel, eine angezogene Handbremse oder überhöhte Temperatur signalisieren.

Bei Python werden Zusicherungen, durch assert-Statements realisiert, die folgende Syntax haben:

```
assert Bedingung
```

Die Bedingung ergibt sich aus einem intuitiven Modell, das einen Teilaspekt des Gesamtmodells, das hinter dem Programm steht, pointiert. Als Beispiel betrachten wir die Definition einer rekursiven Funktion (Python), die den Quicksort-Algorithmus implementiert.

```
def qsort (liste):
    s = liste[:]             # s ist eine Kopie von liste
    if s == []:
        return s
    else:
        assert len(s) >= 1                              #1
        x = s[0]
        s.remove(x)          # entferne x aus der Liste s
        s1 = []
        s2 = []
        for i in s:
            if i <= x:
                s1.append(i)
            else:
                s2.append(i)
        assert len(s1) < len(liste)                     #2
        assert len(s1) <= len(s)                        #3
        ergebnis = qsort(s1) + [x] + qsort (s2)
        assert len(ergebnis) == len (liste)             #4
        assert ergebnis[0] == min(liste)                #5
        assert ergebnis[0] <= ergebnis[-1]              #6
        return ergebnis
```

Hier sind einige assert-Statements eingefügt, die folgende Intuitionen wiedergeben:

#1: Die Liste `s` muss mindestens ein Element enthalten.

#2: Die Teilliste `s1`, die im rekursiven Aufruf von `qsort()` als Argument übergeben wird, ist kürzer als die (als Argument übergebene) zu sortierende Liste `liste`. Anderenfalls droht eine Endlosrekursion. Hier wird also eine Grundidee rekursiver Algorithmen aufgegriffen: In einem rekursiven Aufruf wird der Lösungsalgorithmus auf einen kleineren Teil des Gesamtproblems angewendet.

#3: Variante von Zusicherung 2. Da die Liste `s` um ein Element kürzer ist als die ursprüngliche Liste, kann `s1` auch die gleiche Länge wie `s` haben.

#4: Hinter dieser Zusicherung steht die intuitive Vorstellung, dass beim Sortieren die Anzahl der Elemente erhalten bleibt. Es geht nichts verloren und es kommen keine Elemente hinzu.

#5: Bei einer aufsteigend sortierten Liste muss das kleinste Element am Anfang stehen.

#6: In einer aufsteigend sortierten Liste ist das letzte Element mindestens so groß wie das erste.

Eine Gruppe von 21 Schülerinnen und Schülern, die mit dem Quicksort-Algorithmus und Python-Zusicherungen vertraut waren, mussten in einem (benoteten) Test folgende Aufgabe lösen. Gegeben war die obige Definition der Funktion `qsort()` – allerdings ohne Zusicherungen. An den Stellen, wo sich im obigen Listing die `assert`-Statements befinden, war jeweils eine Lücke gelassen. Außerdem gab es eine Liste von geeigneten und ungeeigneten assert-Statements. Die Aufgabe bestand darin mindestens drei geeignete `assert`-Anweisungen an den passenden Stellen einzufügen. Tab. 1 gibt zu jeder korrekten Zusicherung die Häufigkeit an, mit der sie (auf korrekte Weise) in das gegebene Programm eingesetzt wurde. Bemerkenswert ist, dass die Zusicherung `len(ergebnis)== len(liste)` nur von acht Teilnehmern gewählt wurde, obwohl sie doch ein sehr einfaches, intuitives Konzept verkörpert: „Beim Sortieren einer Liste verändert sich die Anzahl der Elemente nicht“. Eine mögliche Erklärung ist, dass dieser Gedanke nicht unmittelbar mit der primären Aufgabe der Funktion (Sortieren, in eine bestimmte *Reihenfolge* bringen) verbunden ist. Für diese Erklärung spricht auch, dass in 20 Fällen eine Eigenschaft einer aufsteigend sortierten Liste geprüft wurde, nämlich die Tatsache, dass das erste Element das kleinste sein muss (`ergebnis[0]== min(liste)`).

Zusicherung	**Häufigkeit**
`assert len(s) >= 1`	21
`assert len(s1) < len(liste)`	7
`assert len(s1) <= len(s)`	8
`assert len(ergebnis) == len (liste)`	8
`assert ergebnis[0] == min(liste)`	20
`assert ergebnis[0] <= ergebnis[-1]`	8

Tab. 1: Zusicherungen für eine Implementierung von Quicksort. Häufigkeit der korrekten Verwendung bei einem Test mit 21 Schülerinnen und Schülern eines Informatik-Grundkurses der Jahrgangsstufe 13.1 (Herbst 2004).

4 Empirische Erforschung intuitiver Modelle

4.1 Forschungsansätze und methodische Probleme

Viele empirische Studien zu Fehlvorstellungen und intuitiven Modellen in der Informatik folgen einem der folgenden Muster:

- Schüler oder Studenten schreiben ein Programm oder vervollständigen ein Skelett-Programm, das eine vorgegebene Funktionalität besitzen soll (z.B. Mayer 1989).
- Die Versuchsteilnehmer erhalten eine Aufgabe und verschiedene Programme, die die Aufgabe lösen sollen, zur Auswahl. Sie haben zu entscheiden, welches Programm die Aufgabe löst und welches nicht (z.B. Kahney 1989).
- Die Teilnehmer erhalten Programme und sollen antizipieren, welche Ausgabe sie liefern (Kurland, Pea 1985).

Aus den Lösungen (insbesondere Fehlern) wird auf die Verwendung bestimmter mentaler Modelle geschlossen. Dabei ist zu bedenken, dass ein vermutetes Konzept das geistige Produkt des Untersuchers ist. Das zentrale Grundproblem derartiger Laborexperimente ist aber, dass die Motivation der Teilnehmer unklar bleibt. Die Lösung eines algorithmischen Problems ist eine anspruchsvolle kognitive Leistung, die eine gewisse Anstrengung erfordert. Auf der anderen Seite ist es leicht, aus einer Auswahlliste nach dem Zufallsprinzip eine mögliche Antwort anzukreuzen, wenn man keine Lust hat sich ernsthaft mit der Aufgabe auseinanderzusetzen. Bei freien Programmieraufgaben konnte beobachtet werden, dass die Versuchsteilnehmer die ursprüngliche Aufgabe umformuliert (vereinfacht) haben und in der Versuchssituation ein Programm entwickelten, das nur einen Teil der geforderten Funktionalität besaß (z.B. Spohrer et al. 1989).

Eine Lösung des Motivationsproblems ist, die Erhebungen in reguläre Tests des normalen Unterrichts zu integrieren (z.B. Fothe 2005). Da die Schülerinnen und Schüler gute Noten erhalten wollen, sind sie motiviert, ihr Bestes zu geben. Die Schwierigkeiten bei diesem Ansatz liegen darin, dass eine enge Kooperation mit Lehrerinnen und Lehrern erforderlich ist und der Test inhaltlich und methodisch curricularen Vorgaben entsprechen muss.

Eine zweite Erhebungsmethode sind Beobachtungen und Interviews, bei denen die Teilnehmer „laut denkend“ ihre Überlegungen während eines Problemlösungsprozesses mitteilen (z.B. Chiu 1996, 2000, 2001). Ein Problem ist jedoch, dass intuitive Konzepte unbewusst sein können oder es den Versuchsteilnehmern an sprachlichen und visuellen Ausdrucksmöglichkeiten mangelt.

In „Teach back“-Befragungen werden die Teilnehmer aufgefordert, bestimmte Sachverhalte mit Hilfe von selbst angefertigten Texten und Bildern zu erklären. Mit diesem Verfahren versuchte Gerrit van der Veer (1994) in einer umfangreichen Untersuchung (607 Schülerinnen und Schüler aus drei europäischen Ländern) mentale Modelle über die Arbeitsweise von Computersystemen zu ermitteln. Untersuchungsgegenstand waren auch die verwendeten Ausdrucksmittel. Es zeigte sich, dass visuelle Darstellungsformen gegenüber Text nur eine untergeordnete Rolle spielten. So enthielten nur 25% der Teach-back-Produkte bildhafte und 28% ikonische Darstellungen.

4.2 Visualisierungsübungen

Der deutsch-amerikanische Kunstpsychologe Rudolph Arnheim ließ seine Studenten Visualisierungsversuche machen. Die Versuchspersonen sollten zu abstrakten Begriffen Zeichnungen anfertigen, die den Begriff bildhaft darstellen sollten. Dabei sollten sie für jede neue Version ein neues Blatt verwenden, damit man die Entwicklung ihrer Idee verfolgen konnte. Die Studenten brauchten viel Zeit und ein Dutzend Blätter und mehr bis sie zu einer befriedigenden Lösung kamen (Arnheim 1972, S.120 ff). Aus Arnheims Erfahrungen kann man folgendes Fazit zeihen: Eine visuelle Repräsentation für ein intuitives Modell zu finden ist erstens eine Kunst, die gelernt werden muss, und zweitens ein Entwicklungsprozess. Das muss bei der Gestaltung von Visualisierungsexperimenten mit Schülern bedacht werden. Besser als ein einmaliger Individualtest (wie bei van der Veer) erscheint deshalb die

Integration derartiger Aktivitäten in den Unterricht – und zwar so, dass Dazulernen ermöglicht wird. Im Juni 2006 habe ich mit 73 Schülerinnen und Schülern einer Wittener Gesamtschule (Grundkurse Informatik 11.2 und 12.2) eine Reihe von Visualisierungsübungen durchgeführt. Die Aufgabe war, mit Bleistift und Papier ein Storyboard zu entwickeln, das die Ausführung eines kurzen Java-Programms visualisiert. Die meisten der Schüler haben dann später auf der Basis ihrer Storyboards Flash-Animationen erstellt. Die Schüler konnten an Gruppentischen zusammenarbeiten und Ideen austauschen. Es sollte aber jeder seine eigene subjektive Vorstellung zu Papier bringen. Wie in Arnheims Übungen wurden die Schüler außerdem ermuntert, zu einer Aufgabe mehrere Lösung zu zeichnen (Aufgaben und Arbeitsblätter finden sich in Anhang 3.1).

4.3 Die Python Visual Sandbox

Die Python Visual Sandbox (PVS) ist eine Sammlung von interaktiven Online-Applikationen (Spielen) mit insgesamt etwa 150 Animationen, die informatische Konzepte veranschaulichen. Im Unterschied zu freien Visualisierungsübungen (siehe voriger Abschnitt) werden hier vorgefertigte Repräsentationen intuitiver Modelle vorgeführt, von denen vermutet wird, dass sie Schülerinnen und Schülern (bewusst oder unbewusst) verwenden. Die PVS setzt darauf, dass die Nutzer (Spieler) interne mentale Modelle wieder erkennen, die sie vielleicht selbst – mangels Visualisierungskompetenz – nicht explizieren können. Es gibt drei Typen von PVS-Applikationen. Zwei davon haben den Charakter von wettkampfartigen Spielen, bei denen man in Abhängigkeit von der Leistung Punkte gewinnen kann. Beim Python Puzzle setzen die Teilnehmer aus vorgefertigten Bausteinen Python-Programme zusammen und testen sie. Die Animationen dienen hier als Hilfe. Bei Python Quiz beurteilen die Spieler Animationen dahingehend, ob sie die Arbeitsweise eines vorgegebenen Programmstücks richtig wiedergeben. Beim dritten Typ – Python Visual – spielt der Aspekt der Leistung keine Rolle. Zu verschiedenen animierten Modellen, die einen Programmlauf veranschaulichen sollen, werden einfach nur drei Fragen gestellt. Punkte gibt es hier allein für die Teilnahme.

Eine PVS-Applikation kann von einer oder zwei Personen gleichzeitig benutzt werden. Damit wird den Arbeitsbedingungen in einer typischen Schule Rechnung getragen, wo häufig zwei Schüler sich einen Rechner teilen müssen. Zu Beginn einer Sitzung loggen sich die Teilnehmer mit einem Passwort ein, das sie zuvor bei der Registrierung spezifiziert haben. In Abhängigkeit von den Passwörtern wird die Sprache für Textelemente (Deutsch oder Englisch) gewählt. Das System registriert wichtige Aktivitäten während der Sitzung und speichert das Sitzungsprotokoll in einer Datenbank ab. Durch Auswertung der Datenbank können Erkenntnisse über die Verwendung visueller Modelle gewonnen werden. Die PVS ist aber nicht nur ein Forschungsinstrument, sondern sie hat auch die Merkmale eines Spiels und einer Lernumgebung.

4.3.1 Die PVS als Spiel

Nach Scheuerl (1975) ist Spiel freiwilliges, intrinsisch motiviertes und Freude bereitendes Verhalten. Nun ist das Lösen von Aufgaben in der PVS kein oberflächliches Amüsement, sondern anstrengende intellektuelle Aktivität. Durch spielerische Elemente kann die Motivation gesteigert werden, diese Anstrengung auf sich zu nehmen. Cailloise (1958) beschreibt vier Merkmale, die bei Spielen in unterschiedlich starken Maße auftreten können, durch lateinische und griechische Begriffe: Agôn (griech.: Wettkampf), Alea (lat.: Würfel), Mimicry (griech.: Schauspiel, Verkleidung) und Ilnix (griech.: Extase). Alea ist eine Metapher für die die Ungewissheit von Spielereignissen wie sie typischerweise bei Würfelspielen auftreten. Der Begriff Mimicry bringt zum Ausdruck, dass Spiele quasi neben der Realität stehen und in einer geschlossenen Fantasiewelt mit eigenen Regeln stattfinden. Für Rollenspiele und Adventure-Games ist die Mimicry-Komponente von zentraler Bedeutung. Ilnix schließlich meint das fast rauschartige Gefühl, das man zum Beispiel beim Achterbahnfahren empfindet.

Zwei der PVS-Applikationen haben den Charakter von Wettkampfspielen (Agôn). Beim Spielen kann man in Abhängigkeit von der Leistung Punkte gewinnen. Der Reiz des Wettkampfes ergibt sich einmal aus dem Vergleich mit anderen Spielern und zweitens aus der Möglichkeit die eigene Leistung zu beobachten und zu verbessern. Jeder Spieler kann für die Applikationen, die sie oder er wenigstens einmal gespielt hat, Highscore-Listen anfordern, die Punktzahl, Nicknames der Spieler und Datum der

besten zehn Sessions enthalten. Außerdem kann jeder Spieler einen Activity-Report abrufen, der seine erreichte Gesamtpunktzahl und Kurzbeschreibungen aller absolvierten Sessions auflistet.

Eine Sitzung mit einer PVS-Applikation enthält viele ungewisse Ereignisse (Alea), die eine gewisse Spannung hervorrufen. In einem Python Puzzle kann ein Spieler niemals sicher sein, ob sein Programm korrekt ist, bis er es getestet hat. Auf der anderen Seite ist es oftmals überraschend, dass ein kleines Programm nicht das vorhergesehene Verhalten zeigt. Bei einem Python Quiz erlebt man die Ungewissheit, wie das System die eigenen Antworten beurteilt.

Aktivität und angeregte Spannung (Ilnix) wird durch die hohe Spielgeschwindigkeit gefördert. Python Puzzles haben eine maximale Spielzeit von zehn Minuten, ein Python Quiz mit 20 bis 30 Einzelaufgaben dauert im Schnitt acht bis neun Minuten. Die PVS enthält keine Fantasieelemente. Die Mimicry-Komponente fehlt fast völlig. Hier besteht noch viel Entwicklungspotenzial. Die Verlagerung von Aktivitäten in eine Fantasiewelt kann helfen, eine entspannte Spielhaltung (Scheuerl) einzunehmen. [4]

4.3.2 Lernen mit der PVS

Empirische Untersuchungen in der Schule sind besser zu rechtfertigen, wenn für die beteiligten Schülerinnen und Schüler Profit in Form von Lernerfahrung herausspringt. Deshalb wurde die PVS auch als Lernwerkzeug konzipiert, das im Unterricht eingesetzt werden kann. Lernziele betreffen natürlich vor allem intuitive Modelle für Konzepte der Informatik:

- Erweiterung des Repertoires an intuitiven Modellen und Metaphern und damit verbunden eine Verbesserung der Kommunikationsfähigkeit.
- Unbewusste Intuitionen werden durch Visualisierungen expliziert und ins Bewusstsein gerufen.
- Intuitive Vorstellungen werden reflektiert und Fehlvorstellungen aufgedeckt.

Die Python Puzzles bieten zusätzlich die Gelegenheit, metakognitive Kompetenzen zu üben, die für eine Softwareentwicklung (und andere Bereiche des Lebens) wichtig sind wie z.B. Zeitmanagement, Selbstbeobachtung, Verwendung visueller Modelle, die eine Lösungsidee repräsentieren, und Strategien des Problemlösens.

Unterrichtsmethodisch kann die PVS zum individuellen Üben oder als Vorbereitung einer Plenumsdiskussion über intuitive Modelle eingesetzt werden.

4.3.3 Überblick über den technischen Aufbau der Python Visual Sandbox

Die PVS ist ein Client-Server-System und besteht aus folgenden Komponenten (siehe Abb. 9):

- Flash-Applikationen und interaktive HTML-Seiten, die mit einem Web-Browser über das Internet abgerufen werden und auf dem Client-Rechner ausgeführt werden,
- Dienstprogramme (Python-Skripte), die auf einem zentralen Server laufen.

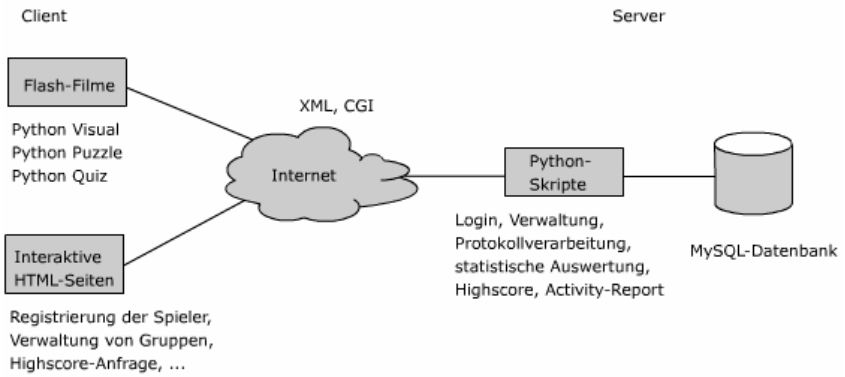


Abb. 9: Aufbau der Python Visual Sandbox

[4] Ein weitergehende Diskussion der PVS als webbasiertes Spiel findet man in Weigend 2006c

Client

Herzstück der PVS ist eine Sammlung interaktiver multimedialer Spiele (Python Visual, Python Puzzle und Python Quiz), die über den Web-Browser aufgerufen werden. Diese Applikationen führen ein Protokoll über die Benutzeraktionen während einer Sitzung und senden die Daten an ein Dienstprogramm auf dem Server, das die jeweilige Session in der Datenbank dokumentiert.

Vor der ersten Nutzung der PVS muss sich ein Spieler auf einer HTML-Seite mit Formular registrieren. Sie oder er spezifiziert einen Nickname (öffentlich) und ein Passwort (geheim) und macht folgende Angaben zur Person: Geschlecht, Alter, Beruf, Zeit, die sie oder er wöchentlich mit Programmieren verbringt. Die Webseite ist mit einem Service-Skript verbunden, das die Daten in die Datenbank einträgt.

In einem zweiten Verwaltungsbereich (HTML-Seiten mit Formularen), können Lehrpersonen (Coaches) Gruppen definieren und jeder Gruppe Personen (Nicknames) zuordnen. Erhobene Gruppenmerkmale sind z.B. Größe der Stadt, Programmiervorkenntnisse, minimales und maximales Alter der Gruppenmitglieder. Einige dieser Gruppenmerkmale können verwendet werden, um die Korrektheit der Registrierungsdaten der Gruppenmitglieder zu prüfen (Validitätskontrolle).

Server

Die Daten zu den Spielern, zu den verwendeten Modellen und Protokolle der Sessions werden in einer zentralen MySQL-Datenbank gespeichert (Details in Anhang 3.2). Für die Python Puzzles gibt es einen Online-Interpreter, der die vom Spieler aus Bausteinen zusammengesetzten Python-Skripte empfängt, ausführt und das Ergebnis (Ausgabe des Skriptes) zurückgibt. Zu Beginn eines Spiels, nachdem die Benutzer ihre Passwörter eingegeben haben, wird ein Dienst aufgerufen, der die Validität der Passwörter prüft und im positiven Fall die zugehörigen Nicknames zurückgibt, die dann im Spiel verwendet werden. Für jede Spielkategorie gibt es ein Dienstprogramm, das die Protokolldaten (z.B. die Zeit, wie lange sich ein Spieler eine Animation angesehen hat) zu einer Spielsession aufnimmt und in die Datenbank einträgt. Die Kommunikation erfolgt über XML-Dokumente. Passwort-geschützte Auswertungsprogramme liefern formatierte und bebilderte Dokumente mit statistischen Angaben zu den Sitzungen mit der PVS. Ein öffentliches Auswertungsprogramm berechnet eine HTML-Seite mit einem allgemeinen Überblick über die Nutzung der PVS (Anzahl registrierter Spieler, Anzahl der Sitzungen etc.). Weitere Service-Skripte dienen der Verwaltung von Gruppen, Ermittlung von Highscores und Activity-Reports.

4.4 Python Visual

Bei einem Python Visual sehen die Spieler einen kurzen Programmtext oder eine kurze Beschreibung eines allgemeinen Konzeptes der Informatik (z.B. reguläre Ausdrücke). Danach folgen drei bis vier Animationen, die das Programm bzw. Konzept erläutern. Die Animationen können beliebig oft abgespielt und zwischenzeitlich gestoppt werden. Am Ende werden den Spielern drei Fragen gestellt, die sie beantworten müssen. Im Unterschied zu Python Quiz gibt es keine richtigen oder falschen Antworten. Für das erstmalige Bearbeiten eines Python Visuals gibt es 20 Punkte für den Activity-Report. In der Regel wird eine gewisse Mindestbearbeitungszeit verlangt, um auf ernsthaftes Beantworten der Fragen hinzuwirken.

Am Ende einer Session wird ein XML-Paket mit den Betrachtungszeiten und gewählten Antworten der Spieler an das zugehörige Service-Programm geschickt, das die Datenbank aktualisiert.

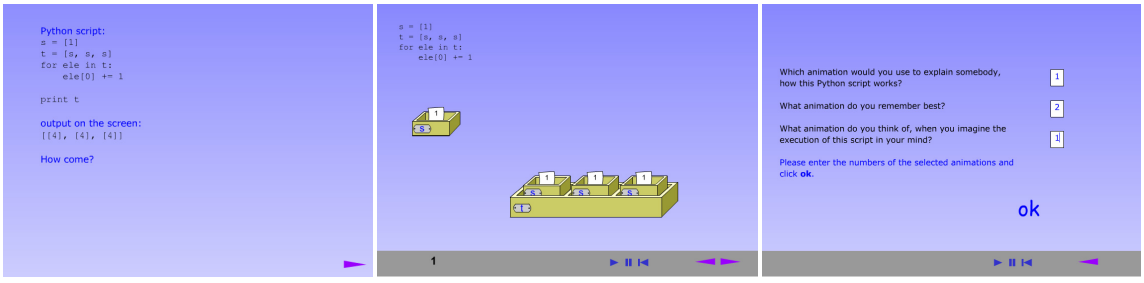


Abb. 10: Screenshots aus dem Python Visual „Multilists“

Die Antworten der Mitglieder einer Gruppe (z.B. Teilnehmer/innen eines Workshops) können mit einem (online verfügbaren) Werkzeug ausgewertet werden. Es liefert in einer HTML-Seite zu jedem Modell folgende Informationen (siehe Abb. 11):

- Einen Screenshot aus der Animation als Erinnerungshilfe.
- Zu jeder Frage den Anteil der Personen aus der betreffenden Gruppe, die dieses Modell gewählt haben (numerisch und grafisch als farbigen Balken).
- Zeitliche Dauer der Animation und mittlere Betrachtungszeit in der Gruppe.

In Workshops mit der PVS wurden diese Übersichten als Anlass für eine Plenumsdiskussion über die Modelle und ihre Schwächen verwendet. Durch die Einbettung in einen sozialen Kontext sollte (neben dem Lerneffekt) auch eine gewisse Ernsthaftigkeit bei der Bearbeitung der Python Visuals bewirkt werden.

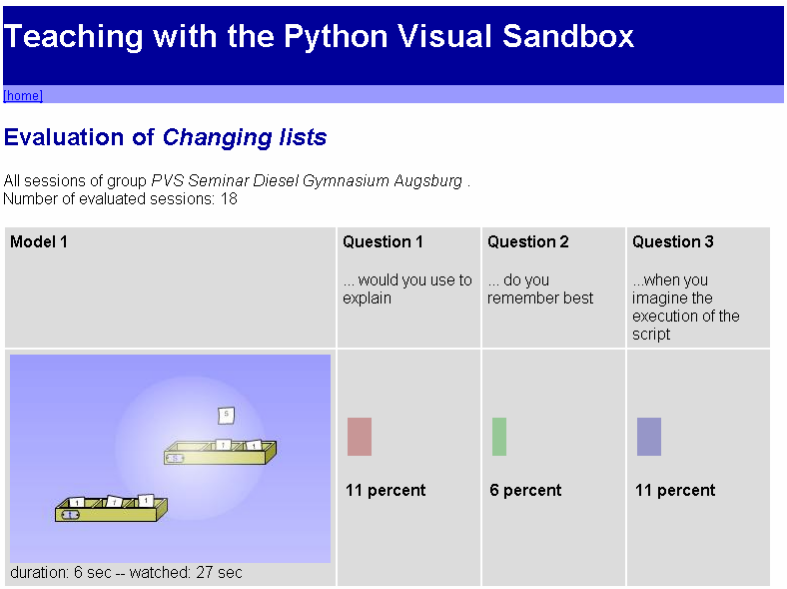


Abb. 11: Auszug aus der Evaluation der Antworten zum Python Visual „Changing lists“

4.5 Python Puzzle

Bei einem Python Puzzle bauen die Spieler aus vorgegebenen Programmtext-Zeilen („Bausteine“) unter Zeitdruck ein lauffähiges Python-Skript zusammen. Dabei können sie Tipps abrufen, die bei der Problemlösung helfen. Das zusammengebaute Programm kann getestet werden. Erst wenn das Programm fehlerfrei läuft und die korrekte Ausgabe liefert, kommt man zur nächsten Aufgabe.

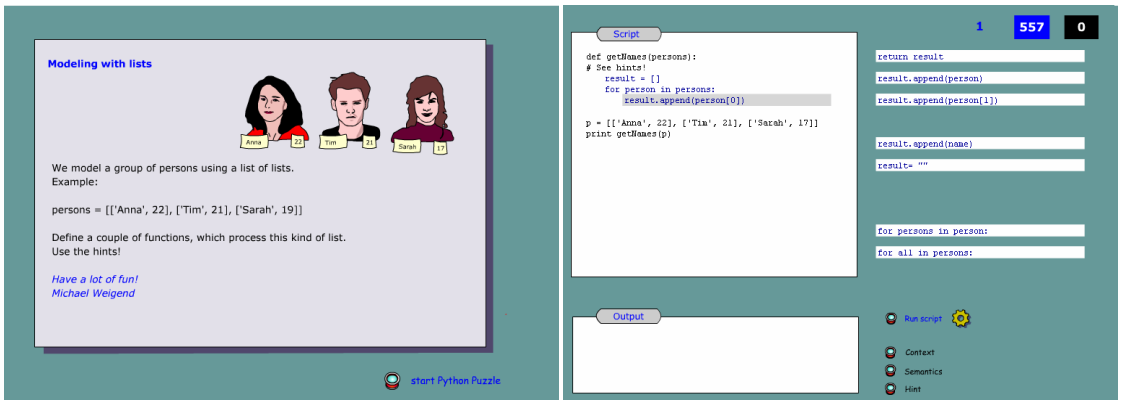


Abb. 12: Screenshots aus einem Python Puzzle. Links die Seite mit der Problemstellung (kann immer wieder hervorgeholt werden), rechts die Editorseite.

Es gibt eine gewisse Anzahl von Aufgaben, die innerhalb einer fest vorgegebenen Zeit bearbeitet werden müssen. Für jede Aufgabe gibt es je nach Schwierigkeitsgrad eine gewisse Punktzahl. Ziel ist es, möglichst viele Punkte zu sammeln. Wer alle Aufgaben gelöst hat, bekommt Bonuspunkte zugeschlagen, die aus der Anzahl der verbleibenden Sekunden berechnet werden. Bei den meisten Aufgaben soll eine Funktion definiert und getestet werden.

Abb. 12 zeigt zwei Screenshots aus dem Python Puzzle „Modeling a group“. Darin wird eine Gruppe von Personen durch eine Liste modelliert, deren Elemente Listen aus Name und Alter sind:

```
p = [['Anna', 22], ['Tim', 21], ['Sarah', 19]]
```

In Aufgabe 1 soll eine Funktion definiert werden, die zu einer solchen Liste eine Liste mit den Namen der Gruppenmitglieder liefert. Zu dieser Funktionsdefinition ist nur der Funktionskopf vorgegeben:

```
def getNames(persons):
```

Die Anweisungen des Funktionskörpers müssen eingefügt werden. Angeboten werden Puzzlestücke mit korrekten und falschen Anweisungen, die mit der Maus in das Editorfeld verschoben werden (siehe Abb. 12). Die korrekte Lösung lautet:

```
def getNames(persons):
    result = []
    for person in persons:
        result.append(person[0])
    return result
```

Durch die Auswahl an Bausteinen, die zur Verfügung stehen, wird erzwungen, einen bestimmten Algorithmus zu implementieren. Es gibt also im Prinzip (wie bei einem richtigen Puzzle) nur eine mögliche Lösung. Lediglich einige wenige Marginalien sind variabel (z.B. Reihenfolge einiger Anweisungen). Ein wesentlicher Vorteil der Verwendung von Bausteinen liegt darin, dass syntaktische Flüchtigkeitsfehler (wie vergessene Kommata oder Klammern), die für wissenschaftliche Fragestellungen weitgehend irrelevant sind, ausgeschlossen werden.[5] Es kann sich um ein „geschlossenes Puzzle“ handeln, in dem alle Programmbausteine verwendet werden oder ein „offenes Puzzle“, in dem auch falsche oder nicht benötigte Bausteine angeboten werden. Mit offenen Puzzles kann man untersuchen, für welche Fehler Programmentwickler besonders anfällig sind. Es werden dann in Konkurrenz zu einer richtigen Anweisung verschiedene fehlerhafte Statements mit unterschiedlichen Merkmalen als Puzzlesteine angeboten.

Zu Beginn einer Sitzung erscheint (nach dem Einloggen) eine Seite, in der der Problemkontext der Puzzleaufgaben erklärt wird (Abb. 12 links). Nach einem Klick auf den Knopf rechts unten wird die Uhr gestartet und man gelangt auf die Editorseite (Abb. 12 rechts).

Im Editorfeld (Überschrift „Script“) befindet sich oben die Kopfzeile der Funktion und unten eine print-Anweisung, in der die Funktion zum Testen aufgerufen wird. Ein Kommentar enthält eine Beschreibung der Funktionalität („Was soll die Funktion leisten"). In einigen Fällen wird auf diese verbale Beschreibung komplett verzichtet. Stattdessen wird die Funktionalität in einer Animation beschrieben. Dann steht im Kommentar nur „See hints“.

Wenn der Spieler den Test-Knopf betätigt, wird das Skript aufbereitet, an den Server geschickt, die Ausgabe und eventuell Fehlermeldung im Ausgabefenster dargestellt und eine Rückmeldung für den Spieler generiert. Liefert das Skript die korrekte Ausgabe, erscheint ein „next task“-Knopf., der den Übergang zur nächsten Aufgabe ermöglicht.

Falls ein Laufzeitfehler auftritt, wird die Antwort des Python-Interpreters vom Client-Programm (Flash) auf die eigentliche Fehlermeldung (z.B. `SyntaxError`) verkürzt und erscheint im Ausgabefeld (Ausgabe). Der Testlauf wird nun vom System analysiert und eine Rückmeldung für den Spieler generiert. Insbesondere wird der Testlauf mit Punkten bewertet.

[5] Solche Flüchtigkeitsfehler (engl. „flaws“) dominieren sehr stark bei realen Programmentwicklungen und ihre Beseitigung kostet gerade Programmieranfängern viel Zeit (vgl. z.B. Anderson & Jeffries 1985).

4.5.1 Dokumentation einer Session

Zu Beginn einer Session wird ein XML-Paket an das zugehörige Serviceprogramm geschickt. Es enthält die IDs (Passwörter) der Spieler, sowie Primärschlüssel (IDs) für das Spiel und die Session. Vom Serviceprogramm wird ein Datensatz für die Beschreibung der Session angelegt, als Spieldauer 0 Sekunden und als erreichte Punktzahl –200 eingetragen. Ein vorzeitiger Spielabbruch wird also durch Minuspunkte „bestraft“. Damit soll vermieden werden, dass ein Spieler bei einem schlechten Start sofort „die Flinte ins Korn wirft“. Für die wissenschaftliche Auswertung sind gerade die ersten – noch völlig unvoreingenommenen – Spieldurchgänge interessant.

Am Ende einer Session wird ein XML-Dokument mit dem Sessionprotokoll an das Service-Programm geschickt. Es enthält globale Informationen über die Session (Spieler, erreichte Punktzahl etc.) und zu jeder bearbeiteten Aufgabe folgende spezifischen Angaben:

- Bearbeitungsbeginn, Bearbeitungszeit, erreichte Punktzahl, Anzahl der Testläufe.
- Für jedes als Hinweis verwendete Modell: Modell-ID, Zeitpunkt und Dauer der Betrachtung, Bewertung (hilfreich, etwas hilfreich, nicht hilfreich).
- Jede falsche und richtige verwendete Programmzeile (Puzzlestück) und Zeitpunkt der Verwendung (erstmaliges Verschieben in das Editorfeld).
- Das Serviceprogramm berechnet aus den Zeitstempeln die Reihenfolge, in der richtige und falsche Programmzeilen verwendet worden sind (für jedes Stück gibt es einen Rang) und aktualisiert die Datenbank.

4.5.2 Auswertung der Python Puzzle Sessions

Ein spezielles Dienstprogramm wertet die in der Datenbank gespeicherten Sessionprotokolle aus. Es zielt vor allem auf die Beantwortung folgender Fragen ab:

(1) In welcher Reihenfolge werden korrekte Programmzeilen in das Programm eingebaut?

(2) Welche fehlerhaften Programmelemente (bei unvollständigen Puzzles) werden verwendet?

(3) Wie lange und wie häufig werden die angebotenen Tipps betrachtet?

(4) Wie werden die angebotenen Tipps von den Spielern beurteilt?

Als Beispiel betrachten wir einen Auszug aus dem Bericht des Auswertungsdienstes zu Aufgabe 1 aus dem Puzzle „Modeling a group“ (Anhang 3.6.3 zeigt den vollständigen Bericht).

Die folgenden Tabellen beziehen sich auf die Auswertung der Sessions von 20 Schülern und 4 Hochschulstudenten, die die Aufgabe erfolgreich gelöst haben. Bis zur korrekten Lösung wurden im Schnitt 3.3 Testläufe und insgesamt eine Zeit von 320 Sekunden benötigt.

Korrekte Programmzeile	**Verwendung (Prozent)**	**Mittl. Rang (Std.-Abw.)**
`result = []`	24 (100.00 %)	1.38 (0.88)
`return result`	24 (100.00 %)	2.96 (0.95)
`for person in persons:`	24 (100.00 %)	2.67 (1.05)
`result.append(person[0])`	24 (100.00 %)	3.00 (0.78)

Tab. 2: Verwendung korrekter Programmzeilen in Python Puzzle „Modeling a group“

Tab. 2 gibt Aufschluss darüber in welcher Reihenfolge korrekte Programmzeilen in die Funktionsdefinition eingebaut worden sind. Der Rang einer korrekten Programmzeile ist eine Zahl zwischen 1 und der Anzahl aller einzufügenden Programmzeilen. Er gibt – für eine bestimmte Session – den relativen Zeitpunkt, also die Position in der Folge aller korrekten Einfügungen, wieder. In der rechten Spalte der Tabelle wird zu jeder korrekten Programmzeile der mittlere Rang angegeben. So erkennt man, dass meist die Initialisierung der leeren Liste zuerst vorgenommen wurde. Auch bei anderen Puzzles konnte festgestellt werden, dass die Spieler mit der Initialisierung von Variablen begannen.

Dagegen hat Samurçay (1989) beobachtet, dass Anfänger häufig die Initialisierung von Variablen weglassen.[6] Aus den Ergebnissen mit der PVS kann man also schließen, dass hinter dem Weglassen einer Initialisierung keine grundlegende Fehlvorstellung sondern allein Flüchtigkeit steht, die vielleicht das Resultat einer Fokussierung auf den algorithmischen Kern des Programms ist. Bei den anderen Programmzeilen gibt es keine deutliche Tendenz in der Reihenfolge der Verwendung. Das hängt unter anderem damit zusammen, dass vor der korrekten Programmzeile (in unterschiedlichem Maße) zunächst fehlerhafte Anweisungen verwendet wurden. [7]

Tab. 3 gibt einen Überblick über den Gebrauch falscher Puzzle-Stücke. Die Zahlen bestätigen Beobachtungen, die auch in anderen Studien gemacht wurden, nämlich dass Anfänger in Programmtexten alltagssprachliche Formulierungen verwenden (Bonar & Soloway 1985). So hat die Zeile

```
for all in persons:
```

als englischer Text durchaus Sinn, ist aber kein gültiger Python-Programmtext.

Falsche Programmzeile	**Verwendung (Prozent)**	**Mittl. Rang (Std.-Abw.)**
`for all in persons:`	8 (33.33%)	2.62 (1.69)
`for persons in person:`	11 (45.83 %)	2.82 (1.66)
`result.append(name)`	7 (29.17 %)	2.14 (1.35)
`result.append(person)`	14 (58.33 %)	1.71 (1.14)
`result.append(person[1])`	6 (25.00 %)	3.33 (1.03)
`result= ""`	5 (20.83 %)	2.60 (1.82)

Tab. 3: Verwendung falscher Programmzeilen in Python Puzzle „Modeling a group".

Zu jedem als Tipp angebotenen Modell beschreibt der Bericht des Dienstprogramms die gesamte Betrachtungszeit, die Häufigkeit der Betrachtung und die Bewertung. Die Spieler sind gezwungen, die betrachtete Hilfe zu bewerten, bevor sie mit der Programmedition fortfahren können.

4.5.3 Verwendung von Tipps

Ein zentrales Anliegen der Python Puzzles war, Erkenntnisse über die Verwendung von visuellen Modellen als Hilfe bei Programmieraufgaben zu finden. Bedauerlicherweise wurden die angebotenen visuellen Modelle nur in geringem Maße genutzt, selbst wenn die Spieler erhebliche Probleme mit der Aufgabe hatten und keine Lösung finden konnten. Möglicherweise empfanden die Spieler die Auseinandersetzung mit den Modellen als Zeitverlust. Sie bevorzugten, die knappe Zeit allein dem Testen von Programmvarianten zu widmen.

In den Workshops zeigte sich außerdem, dass Spielstrategien und insbesondere die Verwendung der Modelle erst durch mehrfaches Spiel gelernt werden müssen. Eine sinnvolle Vorgehensweise ist z.B. folgende:

- Betrachte eine Animation zuerst als Hilfe, um zu verstehen, *was* die zu implementierende Funktion leisten soll.
- Wenn du nicht sofort auf eine Lösung kommst, analysiere die angebotene Animation dahingehend, *wie* das Programm arbeitet. Was passiert zuerst? Welche Programmzeilen passen zu den dargestellten Aktionen?

[6] Johnson (1990) hält ausgelassene Initialisierungen für so typisch für Anfänger, dass sein didaktisches Debugging-System PROUST, in solchen Fällen lehrreiche Hinweise gibt.

[7] Aus der undeutlichen Reihenfolge kann man also nicht schließen, dass die Spieler uneinheitliche Vorgehensweisen zur Lösung der Aufgabe verfolgten.

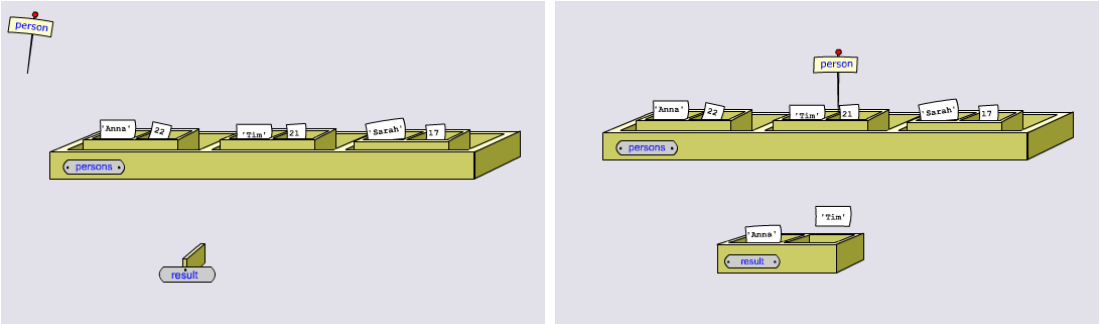


Abb. 13: Zwei Screenshots aus dm „Tipp“ des Python Puzzles „Multilists“.

In zukünftigen Versionen müsste der Abruf von Tipps auf geschickte Weise belohnt werden, damit ein Anreiz geschaffen wird, sich überhaupt auf die Verwendung paradigmatischer Visualisierungen einzulassen und in einen Lernprozess einzusteigen.

4.6 Python Puzzle assert

Python Puzzle Assert ist eine Variante von Python Puzzle. Hier ist ein fertiger Programmtext vorgegeben, in den noch Zusicherungen (assert-Anweisungen) durch Verschieben mit der Maus eingebaut werden können. Eine assert-Anweisung hat folgenden Aufbau:

`assert` *`Bedingung`*

Wenn die Bedingung nicht erfüllt ist, wird die Ausführung des Programms mit einer Fehlermeldung abgebrochen. Mit assert-Statements kann man die logische Korrektheit eines Programms absichern. Hinter jeder Zusicherung steht eine kontrollierende Intuition. Je mehr Assert-Statements eingebaut worden sind, desto mehr Punkte gibt es, sofern jedes assert-Statement logisch sinnvoll ist.

Für jedes Programm stehen mehrere assert-Statements zur Verfügung. Sie befinden sich am rechten Bildschirmrand und können mit der Maus zwischen beliebige Zeilen des Programmtextes geschoben werden. Manchmal wird in einem Spiel der gleiche Programmtext mehrfach verwendet, aber jeweils unterschiedliche Sets von assert-Statements angeboten. Für jedes Programm, das mit Zusicherungen logisch abgesichert werden soll, gibt es drei Testläufe. (Das ist ein Unterschied zu Python Puzzle.) Somit kann ein Spieler z.B. zunächst nur ein assert-Statement ausprobieren und bei späteren Testläufen weitere hinzufügen. Das System verlangt, dass mindestens eine Zusicherung eingebaut worden ist, bevor es einen Testlauf zulässt.

Nach einem Testlauf gibt es folgende Möglichkeiten einer Rückmeldung:

- Alle Zusicherungen sind logisch richtig. In diesem Fall gibt es für jede Zusicherung 5 Pluspunkte.
- Mindestens eine Zusicherung ist logisch fehlerhaft. Dann gibt es 10 Minuspunkte. Logisch fehlerhaft heißt, dass die Zusicherung einen `AssertionError` auslöst, obwohl das Programm korrekt ist.
- Mindestens eine Zusicherung hat einen Fehler ausgelöst, der kein `AssertionError` ist (z.B. falsche Einrückung). Dann gibt es 5 Minuspunkte und einen zusätzlichen Testlauf.

Generell wurde versucht die Punktevergabe so zu gestalten, dass es sich einerseits lohnt, möglichst viele Zusicherungen zu verwenden, man aber andererseits logisch falsche Zusicherungen meiden möchte.

Die Kommunikation mit dem Server und die Dokumentation der Sessions sind genauso wie bei den anderen Python Puzzles.

4.7 Python Quiz

Python Quiz ist ein System, mit dem Zuordnungen von visuellen Modellen zu Programmtexten beobachtet werden können. Bei jeder Aufgabe ist ein kleines Stück eines Programms (eine oder zwei Zeilen) optisch hervorgehoben. Dazu werden nacheinander mehrere Modelle angeboten (zufällige Reihenfolge). Für jedes Modell entscheidet der Spieler, ob es die Arbeitsweise des Programmstücks

richtig wiedergibt oder nicht. Dabei kann sie oder er 0, 5 oder 10 Punkte setzen. Liegt der Spieler richtig, erhält er oder sie die gesetzte Punktzahl als Pluspunkte. War die Entscheidung falsch, wird die dreifache Punktzahl abgezogen. Entscheidungen nach dem Zufallsprinzip werden also drastisch „bestraft“. Es ist strategisch günstiger, null Punkte zu setzen, wenn man die Antwort nicht weiß. Schnelle Entscheidungen (Zeitbedarf weniger als 10 Sekunden) führen zu einer Verdoppelung der erhaltenen Plus- oder Minuspunkte. Somit ist die gesetzte Punktzahl ein Maß dafür, wie sicher der Spieler ist, die richtige Entscheidung getroffen zu haben. Bei 10 gesetzten Punkten kann man als zweites Maß für die subjektive Sicherheit die Zeit betrachten, die für die Entscheidung benötigt wurde.

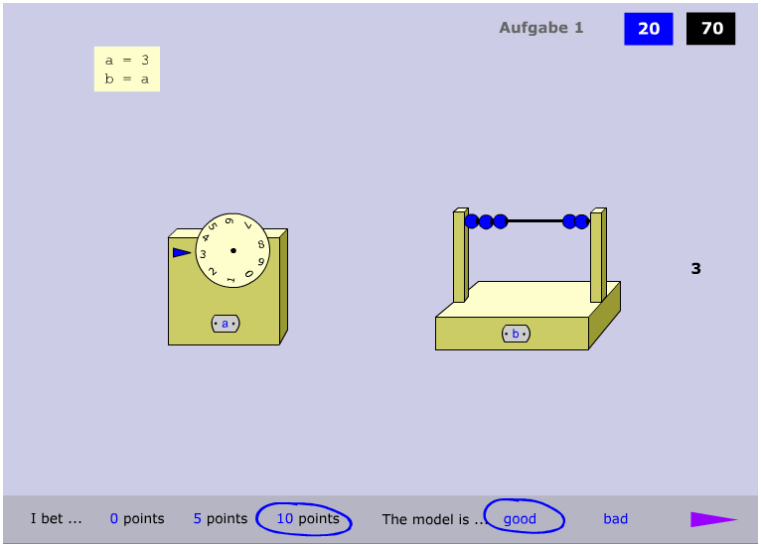


Abb. 14: Screenshot aus einem Python Quiz.

4.7.1 Bewertung der Antworten

Eine Animation kann unterschiedlich interpretiert werden. Strukturelle Abweichungen des Modells vom Original können als Fehler oder als Vereinfachung gesehen werden. Generell reagiert das System sehr tolerant. Nur eindeutig falsche Bewertungen werden mit Minuspunkten bestraft. Die strittigen Fälle werden immer zu Gunsten des Spielers bepunktet.

Merkmale eines eindeutig falschen Modells sind:

- Es enthält symbolische Bezeichnungen, die mit dem Programmtext nicht übereinstimmen (z.B. – statt +)
- Es enthält strukturelle Abweichungen vom Programmtext, die nicht als Vereinfachung interpretiert werden können. Beispiel: Die Zuweisung `x=1` wird durch einen Behälter modelliert, der das Etikett „1“ trägt und als Inhalt einen Zettel mit der Aufschrift „x“ erhält.

In Workshops mit der PVS zeigte sich, dass manche Spieler nicht alle Systembewertungen, die mit Punktverlust verbunden waren, hinnahmen und in Einzelfällen sogar aggressiv reagierten. Um das Risiko eines derartigen Abrückens von einer gelassenen „spielerischen Attitüde“ (im Sinne von Scheuerl) zu verringern, könnte man bei zukünftigen Entwicklungen weitere Fantasieelemente in Quizähnliche Applikationen einbauen. Beispielweise könnte man die Figur eines Schiedsrichters integrieren, der zwar fehlbar ist, aber dessen Entscheidungen man – wie beim Fußball – dennoch akzeptieren muss.

4.7.2 Dokumentation einer Session

Wie bei Python Puzzle wird zu Beginn einer Session ein XML-Paket an ein Service-Programm geschickt, das einen Datensatz für die Beschreibung der Session angelegt, als Spieldauer 0 Sekunden und als erreichte Punktzahl –200 einträgt, um vorzeitigen Spielabbruch unattraktiv zu machen. Am Ende einer Session wird ein XML-Paket mit dem Sessionprotokoll (Beurteilungen der gezeigten Modelle,

gesetzte Punktzahl und Entscheidungszeiten) an ein zweites Service-Programm gesendet, das die Datenbank aktualisiert.

4.7.3 Schlussfolgerungen zur Intuitivität von Modellen

Aus dem Verhalten der Spieler kann man ablesen, in welchem Maß sie die dargestellten Modelle als intuitiv empfinden:

(1) *Geschwindigkeit der Reaktion.* Je schneller das Modell analysiert und bewertet wird, desto intuitiver ist es. Bei einer schnellen Reaktion kann man davon ausgehen, dass der Spieler bereits früher einmal mit einem ähnlichen Modell in Gedanken operiert hat. Bei einer langsamen Reaktion ist eher zu vermuten, dass diese Art der Interpretation für sie oder ihn neu ist.

(2) *Gesetzte Punktzahl.* Je mehr Punkte gesetzt werden, desto sicherer sind die Spieler, dass ihre Einschätzung korrekt ist.

(3) *Korrektheit.* Fehlerhafte Bewertungen – insbesondere wenn ein eindeutig falsches Modell als richtig eingestuft wird – können verschiedene Ursachen haben. Vielleicht wurde das Modell gar nicht durchschaut und es wurde nach dem Zufallsprinzip geantwortet. Oder ein eindeutig falsches Detail des Modells (z.B. falscher Bezeichner) wurde übersehen. In jedem Fall weist eine fehlerhafte Bewertung auf eine geringere Vertrautheit mit dieser Art der Darstellung und damit auf geringere Intuitivität hin.

4.8 Workshops mit der PVS

In den Jahren 2005 und 2006 habe ich insgesamt 14 Workshops mit der Python Visual Sandbox an Schulen und Universitäten in Deutschland und Hongkong (China) durchgeführt. Daran haben insgesamt 241 Personen teilgenommen, davon kamen 187 aus Deutschland, 54 waren weiblichen und 187 männlichen Geschlechts. Unter den Teilnehmern waren 195 Schüler/innen, 24 Universitätsstudierende und 22 Lehrer/innen.

Jeder Workshop war ein Unikat und orientierte sich inhaltlich an den Vorkenntnissen und Interessen der Teilnehmer. Thematische Schwerpunkte (z.B. Rekursion, Sortieren, Modellieren mit Listen oder nur allgemeine Grundlagen der Programmierung) wurden vorher mit den Verantwortlichen an der jeweils besuchten Bildungsinstitution abgesprochen.

Die Workshops waren in eine Spielsituation eingebunden. Es galt, in den Übungen mit der PVS möglichst viele Punkte zu sammeln. Die Person mit den meisten Punkten erhielt am Ende einen Preis. Alle Workshops hatten folgende gemeinsame Struktur:

- Kurze Einführung in die Idee der PVS, das Ziel und den Ablauf des Workshops.
- Registrierung der Teilnehmer.
- Vor jedem Übungsblock mit der PVS wurden in einer Plenumsdiskussion kurz die relevanten Aspekte der Python-Syntax angesprochen, um sicher zu stellen, dass die Teilnehmer nicht völlig überfordert wurden.
- Vor der Nutzung einer PVS-Applikation wurde das Handling präsentiert und insbesondere darauf hingewiesen, dass der Abbruch einer Session zu Punktverlust führt.
- Übungen mit der PVS nahmen den größten Teil der Zeit ein.
- In den meisten Workshops wurden an einer Stelle die Ergebnisse von Python Visuals präsentiert und diskutiert.
- Am Ende gab es die Preisvergabe und ein kurzes Debriefing, bei dem die Teilnehmer allgemeine Anmerkungen und Kritik äußern konnten.

Generell konnte man bei den Workshops ein hohes Maß an Aktivität und Kommunikation beobachten. Die meisten Teilnehmer arbeiteten allein an einem Rechner, diskutierten aber kritische Spielsituationen mit ihren Nachbarn. Dies war – im Hinblick auf den Lerneffekt –erwünscht. Die Frage ist, inwieweit dadurch die Validität der Beobachtungen beeinträchtigt wird. Systematisches Mogeln, z.B. indem man einfach die Ergebnisse von seinem Nachbarn übernimmt, war nur selten zu beobachten. Wegen der Vielfalt der Aufgaben – die zudem von den Applikationen zum Teil in zufälliger Reihen-

folge angeboten werden – und der hohen Geschwindigkeit der Spiele war zu einem Zeitpunkt jeder Teilnehmer gerade mit einer anderen Aufgabe beschäftigt. Die meisten Diskussionen entstanden bei einem Python Quiz, wenn das System ein Modell anders beurteilt hatte als der Spieler. Es ging dann z.B. um die Frage, was an dem betrachteten visuellen Modell falsch ist. Eine solche Diskussion ist für die Beteiligten lehrreich – weil intuitive Modelle reflektiert werden – im Hinblick auf die Validität des Sessionprotokolls aber unschädlich, da sie *nach* der Entscheidung erfolgt ist.

4.9 Systematisierung intuitiver Modelle der Informatik

In den folgenden Kapiteln werden eine Reihe von intuitiven Modellen der Informatik inhaltlich beschrieben, ihre Herkunft und Probleme (Anwendungsgrenzen, Konflikte mit anderen Modellen etc.) diskutiert und – soweit möglich – ihre psychische Realität insbesondere bei Programmieranfängern quantitativ eingeschätzt. Dabei setzen wir den Fokus auf einfache, grundlegende und deshalb häufig verwendete Konzepte. Sie werden im Alltag der Programmierung nur selten in aller Ausführlichkeit visuell oder verbal expliziert, sondern häufig nur stillschweigend verwendet oder sind sogar gänzlich unbewusst. Nicht Gegenstand dieser Arbeit sind speziellere Modelle wie z.B. Architekturen für große Software-Systeme oder Modelle für Vererbungshierarchien in der Objektorientierten Programmierung.

Die hier untersuchten intuitiven Modelle ordnen wir folgenden Themen zu: Akteure, Namen, Daten, Funktionen, Verarbeitung, Steuerung, Klassen und Objekte. Da intuitive Modelle Gestaltcharakter haben und Aspekte aus verschiedenen Bereichen der Programmierung in einer geschlossenen Figur vereinigen, sperren sie sich oft gegenüber einer strengen taxonomischen Einteilung. So kommt es zwangsläufig zu gewissen Überschneidungen. Zum Beispiel enthält das Zeigermodell für Variablen eine besondere Art der Benennung von Objekten und wird in Abschnitt 6.3 mit anderen Benennungmodellen verglichen. Gleichzeitig impliziert dieses Modell bestimmte Vorstellungen zur Datenverarbeitung (z.B. Entstehungs- und Vernichtungskonzepte), die in Abschnitt 10.2 diskutiert werden.

Wie findet man intuitive Modelle? Aus der subjektiven Sicht der Menschen, die sich mit Programmtexten intellektuell auseinander setzen, gibt es unter anderem folgende Quellen für intuitive Modelle:

- Programmiersprachen (textuell oder visuell)
- Entwicklungsumgebungen (z.B. BlueJ, Delphi)
- Visuelle und verbale Erklärungen in der Literatur (z.B. Schulbücher, Sprachreferenzen, Klassendokumentationen)
- Erklärungen anderer Menschen im Rahmen von face-to-face-Kommunikationsprozessen (Programmierteam, Lehrer etc.)
- Alltagserfahrungen mit informatischen Phänomenen (z.B. Umgang mit Behältern, Büchern, Notizzetteln)

Wer informatische Intuitionen aufspüren und erfassen will, kann grundsätzlich diese Quellen untersuchen. Welche intuitiven Modelle werden von Programmiersprachen wie Visual Basic oder LabView unterstützt? Welche informatischen Konzepte stecken hinter dem Gebrauch von selbstklebenden Notizzetteln? Auf welche veranschaulichenden Analogien wird in Lehrbüchern zurückgegriffen?

Intuitive Modelle der Informatik – die man für sich als kulturelle Phänomene dokumentieren kann – haben für Denkprozesse im Zusammenhang mit Computerprogrammen unterschiedliche Relevanz. Zur quantitativen Einschätzung ihrer psychischen Realität werden in dieser Arbeit vor allem die automatisch aufgenommenen Protokolle der PVS herangezogen.

Die Signifikanz der Daten aus den Sitzungen mit Python Visual wurde durch Vergleich der beobachteten Antwortmuster mit einer Gleichverteilung geprüft. An den entsprechenden Stellen wird das Signifikanzniveau (p-Wert) des χ^2-Tests angegeben, das die Wahrscheinlichkeit zum Ausdruck bringt, dass das Ergebnis durch zufälliges Antworten zu Stande gekommen ist. Beim Vergleich von Modellbewertungen in Python-Quiz-Sitzungen wurde der zweiseitige exakte Fisher-Test (Summe kleiner p-Werte) angewendet. Rechnungen wurden mit MS Excel (χ^2-Test) und SISA (Uitenbroek 2000) durchgeführt.

5 Akteurmodelle

Wenn man über die Arbeitsweise eines Programms spricht, unterscheidet man häufig aktive und passive Entitäten. Man stellt sich vor, dass es sich um ein strukturiertes System handelt, in dem verschiedene Akteure am Werk sind, die in irgendeiner Weise kooperieren und gemeinsam eine Aufgabe lösen. Sichtbar werden Akteurkonzepte bei sprachlichen Formulierungen wie

- „An dieser Stelle macht der Computer dies und das."
- „Das Objekt a schickt eine Botschaft an Objekt b"
- „Funktion f ruft Funktion g auf."
- „Die Funktion gibt den Wert x zurück."

Betrachten wir folgendes Programm:

```
def quadrat(n):
    print n*n
zahl = input()
quadrat(zahl)
```

Ein Beispiel für eine typische verbale Erklärung ist folgender Text:

„Das Programm wartet auf eine Eingabe. Sobald der Benutzer über die Tastatur eine Zahl eingegeben hat und <ENTER> gedrückt hat, wird die Funktion `quadrat()` aufgerufen. Sie berechnet das das Quadrat und gibt das Ergebnis auf dem Bildschirm aus."

Hier kann man zumindest drei Akteure erkennen, die zum System gehören:

- Ein Akteur wird als „Programm" bezeichnet. Dieser Akteur wartet zunächst auf eine Eingabe und ruft später eine Funktion auf.
- Die <ENTER>-Taste (betätigt von einem Menschen, den wir nicht zum System zählen) löst die Fortsetzung des Programms aus
- Die Funktion `quadrat()` wird vom Programm aufgerufen, verarbeitet den übergebenen Wert und sorgt dafür, dass etwas auf dem Bildschirm ausgegeben wird.

Daneben gibt es in diesem Modell auch passive Elemente:

- Die eingegebene Zahl
- Das Rechenergebnis der Funktion

Es sei darauf hingewiesen, dass dieses Modell mit der technischen Realität wenig zu tun hat. Aus technischer Perspektive gibt es eine klare Hierarchie von Akteuren: Das Betriebssystem ruft den Interpreter auf und startet damit einen Prozess. Der Interpreter führt das Programm aus und ist damit Akteur. Bei der Abarbeitung werden Betriebssystem-Befehle aufgerufen, damit wird das Betriebssystem wieder Akteur, das Betriebsystem startet wiederum Mikroprogramme des Prozessors, damit wird dieser aktiv.

5.1 Daten als Akteure – Datenflüsse

In vielen intuitiven Modellen der PVS, die die Arbeitsweise eines Programms darstellen, werden Daten durch fliegende Zettel dargestellt, die sich selbstständig ihren Weg durch das System suchen. Ein solcher Zettel kann spontan entstehen (z.B. als Folge einer Eingabe oder als Kopie eines bereits existierenden Zettels) bewegt sich dann auf magische Weise zu einer Box, die eine Funktion darstellt, so als wisse er, was nach dem Algorithmus als nächstes zu tun ist. Im Unterschied etwa zu einer Funktion oder eines Objektes im Sinne der OOP hat eine solche Daten-Entität primär mit sich selbst zu tun (z.B. den richtigen Weg finden) und nur in wenigen Fällen Einfluss auf seine Umgebung. Einer dieser Fälle ist, dass das Datum eine Funktion aktivieren kann, sobald es ihren Eingang passiert.

In Datenflussdiagrammen im Sinne des funktionalen Programmierparadigmas (Hubwieser 2004, S. 93; Sommerville 1997, S.275 ff.) bewegen sich Daten auf festen Bahnen von Funktionsblock zu Funktionsblock. Solche Darstellungen ähneln Materialflusssystemen, die eine chemische Fabrik modellieren. Im Grund ist hier der entscheidende Akteur das Transportsystem. Das Datenflussmodell mit festen Wegen (Transportsystem) ist konzeptuelles Herzstück einer Reihe von visuellen Programmiersprachen bzw. Entwicklungsumgebungen wie VIPER (Sanner et al. 2002), LabView (www.labview.com), DRLP (Anjaneyulu & Anderson 1992) oder Alligator (Mosconi et al. 2003). Das Problem dieser Sprachen ist, dass sie eine vollständig deterministische Definition eines Programms ermöglichen. Deshalb müssen sie z.B. auch Steuerungselemente bereitstellen, mit denen man Programmverzweigungen darstellen kann. Modelle mit Daten-Entitäten als Akteuren, die selbst ihren Weg finden, können von dem Problem der Steuerung abstrahieren und sind deshalb einfacher.

Datenbewegungen werden gelegentlich bei spielerischen Aktivitäten (ohne Computer) zur Erarbeitung eines Algorithmus verwendet. Beispielsweise beim Sortieren von Spielkarten werden Karten auf dem Tisch hin und her geschoben und dabei versucht eine algorithmische Beschreibung des intuitiven Vorgehens (innerhalb der Mikrowelt sich bewegender Datenobjekte) zu finden. Diese kann später die Grundlage für die Formulierung einer Sortierfunktion sein. Neben der Bewegung gibt es noch weitere Formen der Aktivität von Daten-Entitäten:

- Verarbeitungsprozesse werden manchmal so beschrieben, dass Daten sich verändern (Metamorphose, siehe Abschnitt 10.3).
- Viele Menschen stellen sich vor, dass bei einer Zuweisung eine Datenentität die Verbindung zu einem Namen sucht (siehe Abschnitt 10.5.3).

5.2 Namen als Akteure

Manche Verarbeitungsprozesse können dadurch visualisiert werden, dass ein Namensschild, eine Stecknadel, ein Zeiger oder sonst eine benennende Entität sich von Objekt zu Objekt bewegt. Ein Beispiel ist die Darstellung einer Iteration über eine Liste der Form (Python):

```
for i in [1, 2, 3]:
   tue etwas
```

Man kann sich vorstellen, dass während der Ausführung der Iteration eine Markierung (etwa ein Etikett mit der Aufschrift *i*) sich nacheinander an die Zahlen der Liste heftet und so das aktuelle Item benennt.

5.3 Funktionen

Funktionen können als Akteure, die z.B. Daten empfangen und Daten produzieren (zurückgeben) können, modelliert werden. Ein Funktionsaufruf wird dann als Delegation einer Aufgabe an einen anderen Akteur verstanden. Wir gehen später detailliert auf Modelle für die Ausführung von Funktionen ein.

5.4 Allmächtige Steuerungsentität – monoaktive Systeme

Manche intuitiven Modelle implizieren eine übergeordnete Entität, die „willkürlich“ die anderen Objekte verändern kann. Diese allmächtige Entität ist dann der einzige Akteur, die anderen Entitäten sind passiv und warten auf Manipulationen „von oben“.

Ein Beispiel für eines solches Modell ist die Visualisierung der Auswertung eines arithmetischen Terms durch sukzessive Ersetzungen. Zuerst sieht man einen komplexen Term, in dem nach und nach Teile „wie von Geisterhand“ ersetzt oder entfernt werden.

```
(2+1)*4
(3) * 4
3 * 4
12
```

In einer Animation kann die allmächtige Entität auch explizit visualisiert werden (etwa durch einen Greifer, der von oben ins Bild kommt und Objekte manipuliert) oder auch nur implizit enthalten sein.

Im obigen Beispiel ist sie implizit vorhanden, weil man die Auslösung einer Veränderung (Ersatz eines Subtermes) keinem anderen Objekt des Systems zuordnen kann.

Im Metaphernsystem von Lakoff und Núnez gibt es für jede Domäne einen „mathematical agent“, der alle Aktionen vornimmt. Wenn zum Beispiel arithmetische Operationen als Manipulation von Objekten konzeptionalisiert werden, ist der mathematical agent der Akteur, der Objekte verschiebt. Für die Addition 2 + 3 könnte er z.B. zwei Äpfel und drei Äpfel zu einer Kollektion von fünf Äpfeln zusammenlegen (Lakoff & Núnez 1997, S. 33).

Monoaktiv sind auch die einfachen Computermodelle („notional machines“), die bei Programmieranfängern beobachtet werden, wenn sie versuchen die Arbeitsweise eines Programmes zu erklären (du Boulay 1989). Das Bild des Computers als *einer* Entität, die Anweisungen eines Programms ausführt, kommt in anthropomorphen Redeweisen wie „Er versuchte gerade, …“, „Er dachte, du meintest …“ zum Ausdruck.

In einem monoaktiven System wird ein Funktionsaufruf nicht als Delegation einer Aufgabe an einen eigenen Akteur betrachtet, sondern als Kopieren und Einschieben von Programmtext.

5.5 Objekte

Objekte im Sinne der objektorientierten Programmierung sind komplexe Akteure, die über einen internen Datenbestand, den man abfragen und verändern kann, und über ein Repertoire von Operationen verfügen. Sie reagieren auf Botschaften, durch die sie veranlasst werden, aktiv zu werden.

Im Hinblick auf intuitive Modellierung haben Objekte den Vorteil, dass sie viel Information in einer kohärenten Gestalt zusammenfassen können. Abb. 15 zeigt eine „Summierungsmaschine“. In der Animation hüpft sie in der Liste von Element zu Element und summiert die über ihren Sensor „abgetasteten“ Werte. Sie ist insofern ein Objekt als sie ein Attribut (`summe`) besitzt, dessen Wert zu jeder Zeit abgelesen werden kann. Sie beherrscht zumindest zwei Operationen, die in der Animation angedeutet werden, nämlich das Aufaddieren eines Wertes zum aktuellen Wert des Attributs Summe und das Zurücksetzen der Summe auf null.

Ein solches intuitives Objekt kann zur Interpretation eines nicht objektorientierten Programms verwendet werden, in dem nur eine Variable namens `summe` vorkommt, die initialisiert und in einer Iteration verändert wird.

```
summe = 0
for i in liste:
    summe += i
```

Das Modell ist intuitiv und in sich gut verständlich, aber „strukturell entfernt“ von dem Programmtext, der damit erklärt werden soll.

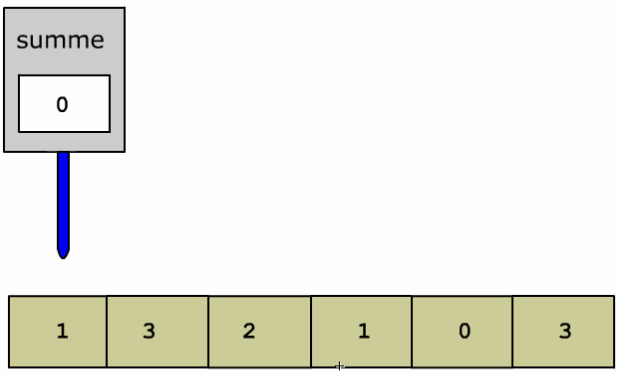


Abb. 15: Ein Objekt als Akteur zur Veranschaulichung der Idee eines nicht objektorientierten Programms.

6 Benennung von Entitäten

Namen für Entitäten der Realwelt werden in früher Kindheit gelernt. Sie ermöglichen die interne gedankliche Repräsentation der externen Welt – nach Piaget der Beginn der Intelligenz. Namen haben verschiedene Funktionen. Sie identifizieren eine Entität, ermöglichen den Zugriff (Adressierung), bringen ihre Funktion innerhalb eines Kontextes zum Ausdruck und repräsentieren somit auch Beziehungen zwischen Entitäten. Namen können explizit aber auch implizit sein. So ist die Position eines Objektes in einer Sequenz (Index) ein Name für das Objekt. Eine Stecknadel auf einer Landkarte markiert (benennt) einen Ort. Wir diskutieren nun einige intuitive Modelle, die sich um die Benennung von Entitäten drehen.

6.1 Behältermodell und Referenzierungsmodell

Eine bekannte Intuition für Variablen ist die Vorstellung eines Behälters für Daten (Behältermodell). Der Behälter – z.B. eine Schachtel – ist mit einem Etikett versehen, das den Variablennamen trägt. Eine Zuweisung der Form

```
x = 1
```

wird so interpretiert, dass der Behälter mit Namen (Etikett) `x` mit einem neuen Inhalt, einer Repräsentation der Zahl `1` gefüllt wird. Der vorige Inhalt wird dabei vernichtet. Der Behälter zusammen mit seinem Etikett dient der Identifikation des momentan „gespeicherten" Objektes (Inhalt) und ermöglicht den Zugriff.

Ein alternatives intuitives Modell verzichtet auf das Behälterkonzept und sieht eine Zuweisung allein als Benennung eines Objektes. Der Variablenname wird als Name eines Objektes interpretiert (Referenzierungsmodell). Man sagt auch: Der Name ist an das Objekt gebunden. Im obigen Beispiel wird der Zahl `1` Name `x` zugeordnet. Anschaulich kann man sich eine Benennung so vorstellen: Man zeichnet einen Pfeil von dem Namen zum Objekt oder etikettiert ein Objekt mit einem Zettel, das einen Namen trägt. Es gibt noch weitere Formen, auf die wir später zu sprechen kommen.

Was ist der Unterschied zwischen beiden intuitiven Modellen? Das Behältermodell stellt den Namen einer Variablen in den Vordergrund. Der Behälter ist eine dauerhafte Entität, sein Inhalt ist flüchtig und austauschbar. Dagegen ist beim Referenzierungsmodell das Objekt die dauerhafte Entität. Der Name kann geändert werden. Im Referenzierungsmodell kann ein Objekt erst benannt werden, wenn es existiert. Unter Umständen ist es anonym und hat keinen expliziten Namen. Dagegen kann ein Behälter auch leer sein. Insofern unterstützen dynamische Programmiersprachen wie z.B. Python, die ohne Variablendeklarationen auskommen, eher das Namensmodell. Denn erst mit einer Zuweisung wird ein neuer Name eingeführt. Dagegen kann man sich eine Variablendeklaration bei Java oder Pascal so vorstellen, dass ein zunächst leerer Behälter bereitgestellt wird, der erst später mit einem Objekt gefüllt wird. Gravierende Konsequenzen werden sichtbar, wenn man folgende Anweisungsfolge interpretiert.

```
x = 1
y = x
```

Im Namenmodell erhält das Objekt `1` nun einen zweiten Namen, nämlich `y`. Das entspricht vollkommen dem Alltagsgebrauch von Namen. Für ein und dasselbe Objekt werden häufig verschiedene Namen verwendet. Wendet man das Behältermodell korrekt an, so ergibt sich folgende Interpretation: Der Inhalt der Variablen `x` wird kopiert und in der Variablen `y` gespeichert.

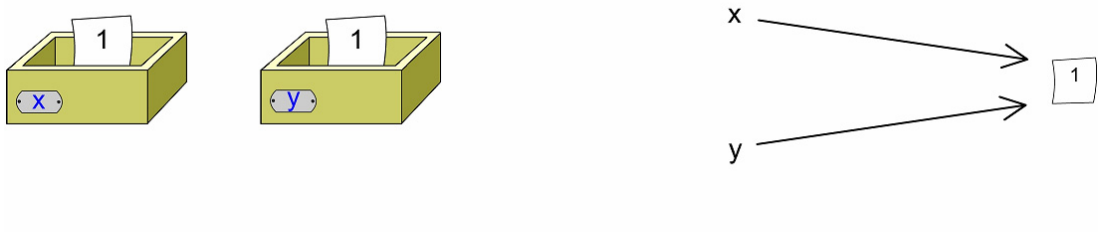


Abb. 16: Modelle für Mehrfachnamen

Bei Programmieraufgaben deutet die Bevorzugung des Behältermodells für Variablen und der Interpretation einer Zuweisung als Kopiervorgang auf eine Tendenz zur Vermeidung von Mehrfachnamen hin. Man arbeitet manchmal lieber mit doppelten Datenentitäten als mit doppelten Namen für ein und dasselbe Objekt.

6.2 Erscheinungsmodelle

Bei unveränderbaren Objekten wie z.B. Zahlen ist das Behältermodell unproblematisch. Schwierig wird es, wenn es sich um änderbare Objekte handelt wie Listen (Python) oder Instanzen selbst definierter Klassen. Betrachten wir die folgende Python-Anweisungsfolge aus der PVS-Applikation *Changing Lists*:

```
s = [1, 1, 1]
t = s
s[0] = 5
print t
```

Die Ausgabe auf dem Bildschirm lautet `[5, 1, 1]` und nicht etwa `[1, 1, 1]`. Zur Erklärung dieses Verhaltens werden vier Visualisierungen angeboten:

(1) Die erste Animation zeigt ein komplett ungeeignetes Modell (Abb. 17). Liste `s` wird als Behälter mit 3 Fächern dargestellt, in denen sich Zettel mit der Zahl `1` befinden. Es erscheint ein zweiter zunächst leerer Behälter gleicher Bauart mit der Beschriftung `t`. Von jedem Zettel in s wandert eine Kopie in `t`. Der erste Zettel in Behälter s wird durch einen Zettel mit der Zahl `5` ersetzt. Behälter `t` bleibt unverändert. Hier müsste die Anweisung `print t` die Ausgabe `[1, 1, 1]` liefern.

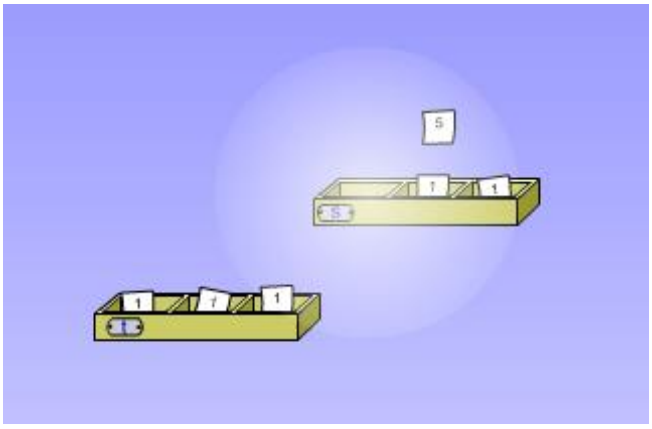


Abb. 17: Ungeeignetes visuelles Modell für das Programm

(2) Auch die zweite Animation verwendet ein Behältermodell und interpretiert die Zuweisung `t = s` als eine Art Kopiervorgang. Es entsteht ein neuer Behälter mit Etikett `t`, der den gleichen Inhalt wie `s` hat (Abb. 18). Allerdings ist der Behälter t keine Kopie im üblichen Sinne. Stattdessen handelt es sich um zwei Erscheinungen desselben Objektes. Jede Veränderung von `s` – wie z.B. die Neubelegung der ersten Kammer mit der Zahl 5 – wird auf magische Weise ebenfalls mit `t` ausgeführt.

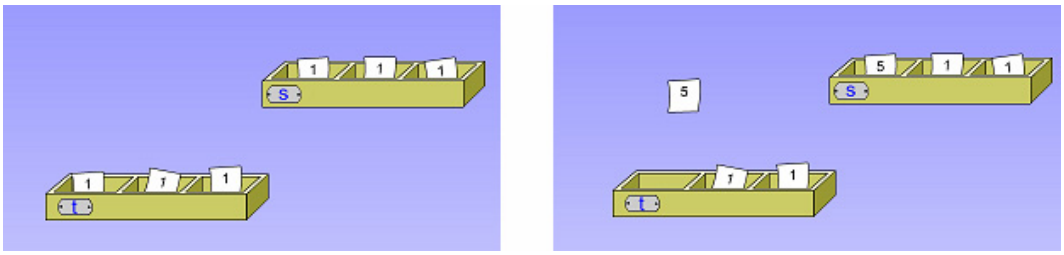


Abb. 18: Behältermodell für die Veränderung einer Liste mit zwei Namen (Erscheinungsmodell).

(3) Die dritte Animation stellt Liste `s` als Behälter dar, der ein Schild mit der Aufschrift `s` trägt. Es erscheint ein zweites Schild mit der Aufschrift `t`, das neben dem alten Schild an dem Kasten befestigt wird. Dem Listenobjekt wird also ein neuer (zweiter) Name zugeordnet (Abb. 19).

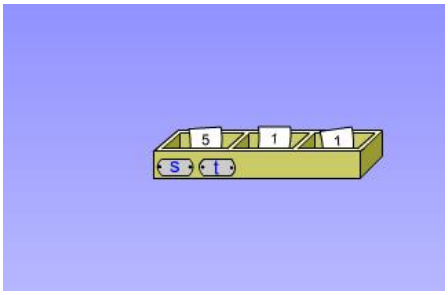


Abb. 19: Modell einer Liste mit zwei Namen

(4) Die vierte Animation schließlich visualisiert das Programm mit einem Zeigermodell (Abb. 20). Die Liste `s` wird durch ein Brett mit drei Zeigern dargestellt, die auf einen einzigen Zettel mit der Zahl `1` zeigen. Ein Pfeil mit der Beschriftung `s` zeigt auf das Brett. Die Zuweisung `t = s` wird dadurch visualisiert, dass ein zweiter Pfeil mit der Beschriftung `t` erscheint, der auf dasselbe Brett verweist. Schließlich erscheint ein neuer Zettel mit Aufschrift 5. Der erste Pfeil auf dem Brett, das die Liste darstellt, löst sich von dem Zettel mit der `1` und zeigt auf die `5`.

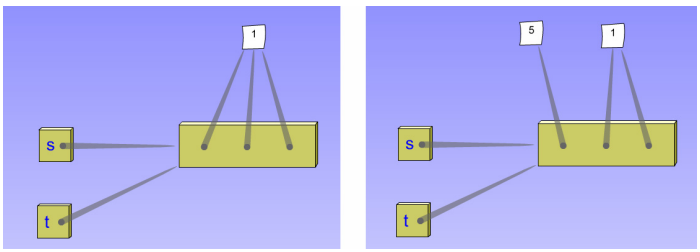


Abb. 20: Konsistentes Zeigermodell für die Veränderung einer Liste mit zwei Namen

In dem Python Visual wurden folgende Fragen gestellt:

- Welche Animation würden Sie verwenden um jemandem zu erklären, wie das Python-Skript funktioniert?
- An welche Animation können Sie sich am besten erinnern?
- An welche Animation denken Sie, wenn Sie sich die Ausführung des Skriptes vorstellen?

Tab. 4 zeigt die Antworten von 70 Schülerinnen und Schülern, die an einem PVS-Workshop teilgenommen haben. Die Verteilung weicht signifikant von einer Gleichverteilung ab ($p = 0.000$), die bei zufälligem Antworten zu erwarten wäre. Die Schülerinnen und Schüler bevorzugten das Erscheinungsmodell (Modell 2). Dagegen wird das konzeptionell einfachere Zeigermodell abgelehnt und sogar von weniger Personen gewählt als das falsche Modell 1.

Wie in Abschnitt 13.10 erläutert wird, kann die Annahme von Erscheinungen als Exhaurierung des Behältermodells gesehen werden. Wenn dieses Konzept trotz seiner (unnötigen) Komplexität von Anfängern zur Interpretation von Programmen verwendet wird, ist das ein Indiz für die hohe Intuitivität des Behältermodells.

n = 70	**Falsch**	**Erscheinung**	**Zweites Schild**	**Zeiger**
Erklären	15	40	11	4
Sich erinnern	12	23	16	19
Sich vorstellen	10	34	17	9
Summe	37	97	44	32

Tab. 4: Wahl verschiedener Modelle zur Veranschaulichung eines Programms. Ergebnisse von 70 ersten Sitzungen während Workshops mit der PVS. Teilnehmer: 56 Schüler und 14 Schülerinnen aus Deutschland, sie beschäftigten sich im Schnitt 3.9 Stunden pro Woche mit Programmierung und hatten ein Durchschnittsalter von 17.0 Jahren.

6.3 Wem gehört ein Name? Zeiger versus Etiketten

Wenn ich bei einem Abendessen am Tisch einen Menschen mit seinem Namen anspreche, um ihm eine Botschaft zukommen zu lassen, gehe ich davon aus, dass die Angesprochene den Namen kennt. Der Name, der im Personalausweis steht, wird als Eigentum des Benannten gesehen. Er ist Teil der persönlichen Daten und das Besitzrecht ist sogar durch Gesetze geschützt.

Wir nennen diese Intuition *Etikettmodell*. Sie geht davon aus, dass der Name einer Entität zur benannten Entität gehört und gewissermaßen Teil von ihr ist. Typische Beispiele aus dem Alltag sind Etiketten auf Weinflaschen, Schulheften und anderen Produkten.

Andererseits existieren im sozialen Alltag auch Namen, die der benannten Person unbekannt sind. Zum Beispiel erfinden Schüler hinter dem Rücken ihrer Lehrer Spitznamen für sie. Diese Namen gehören den Individuen, die sie verwenden. Sie eignen sich zur Identifikation eines Objektes, aber nur mit Einschränkungen zur Kommunikation mit ihm. Wir nennen dieses intuitive Modell *Zeigermodell*. Beispiele für Zeiger im Alltag sind Wegweiser, die sich in einer gewissen Entfernung vom durch sie benannten Zielort befinden.

In Modellen zur Veranschaulichung von Computerprogrammen können Namen durch Pfeile oder Etiketten visualisiert werden. Das Etikett klebt an einem Objekt und gehört ihm damit. Andererseits kann man nicht mehr erkennen, wer das Etikett angeklebt und so die Benennung vorgenommen hat. Beim Pfeil dagegen sind Name und Objekt räumlich getrennt. Der Pfeil beginnt bei einem Namen, und dieser Name kann z.B. Attribut eines anderen Objektes sein. Hier wird also der Besitzer des Namens mit dargestellt.

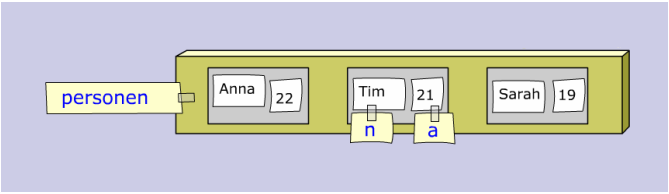


Abb. 21: Etikettenmodell

Besitz eines Namens bedeutet nicht zwangsläufig auch Kontrolle über den Namen. Wer entscheidet über Benennungen und Umbenennungen? Das benannte Objekt oder eine andere Entität? Im Alltag nimmt man Beziehungen zu anderen Menschen auf, indem man sich Ihnen vorstellt. Dabei wählt man selbst den Namen, mit dem man angeredet werden möchte. Im Französischen wird diese Kontrolle über den eigenen Namen durch die Formulierung „Je m'appel …“ (wörtlich: „ich nenne mich …“) besonders deutlich zum Ausdruck gebracht. Etiketten dagegen werden von anderen Entitäten vergeben und geändert. Die Vorstellung, dass ein Objekt seine Namen kontrolliert, verwenden viele Schüler bei der Interpretation von Zuweisungen. Abb. 22 zeigt einen Screenshot aus einer Visualisierung, in der ein Objekt (Zahl 3) sich selbst Namen sucht.

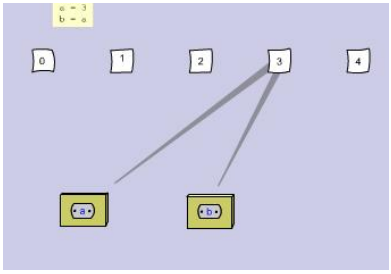


Abb. 22: Zahl als aktive Entität, die ihre Namen kontrolliert

6.4 Vermischung von Namensmodellen

Das Python Quiz „Modeling a group“ enthält Animationen, die die Arbeitsweise einer Iteration der folgenden Form (Python) modellieren (Abb. 23):

```
for n, a in personen:
   tue etwas
```

Das erste Modell verwendet konsequent das Referenzierungsmodell mit Zeigern. In der zweiten Animation werden alle Variablen durch Behälter dargestellt. Nach und nach werden Kopien der Daten aus dem Behälter `personen` herausgenommen und in die Behälter `n` und `a` gelegt. Die dritte Animation verwendet eine Kombination aus Zeigern und Behältern. Die Zeiger `n` und `a` „wandern“ über den Listenbehälter und zeigen nach und nach auf die Zettel in den Fächern. In der vierten Animation schließlich wird eine Variante des Behältermodells für Listen in Kombination mit Etiketten verwendet.

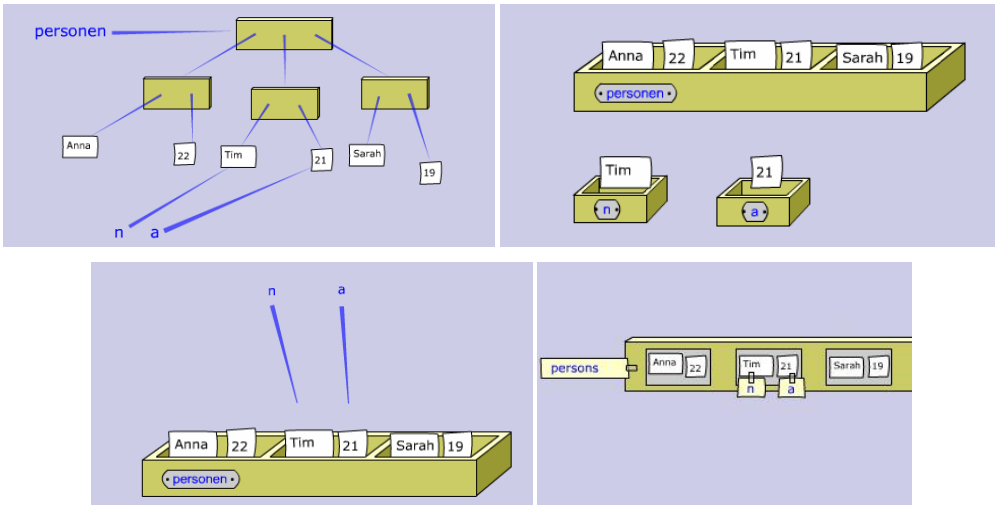


Abb. 23: Modelle für eine Iteration über eine Liste von Paaren

Die Spieler sollten die Modelle als passend oder unpassend qualifizieren. Tab. 5 zeigt, dass von 68 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben, die meisten alle vier Modelle akzeptierten. Deutliche Bevorzugungen sind nicht zu beobachten. Offenbar hat in diesem Beispiel die Konsistenz des Modells keinen Einfluss auf die Intuitivität. Das inkonsistente Modell 3 gibt eine typische Iteration im Alltag wieder: Man durchsucht z.B. Bücher im Bücherregal (Behälter für Bücher), indem man nacheinander einen Blick auf jedes Buch wirft (Neuorientierung eines Zeigers), ohne dabei irgendein Buch zu bewegen.

Bereits Grudin (1989) hat im Zusammenhang mit Benutzungsoberflächen beobachtet, dass es nicht immer auf die interne Konsistenz des konzeptuellen Modells ankommt. Analogien zu Objekten der Realwelt sind häufig wichtiger für die Qualität eines User Interfaces hinsichtlich Erlernbarkeit und „intuitive“ Anwendbarkeit in neuen Situationen.

n = 68	Dauer der Animation	Entscheidungs-zeit (Stdabw.)	Als passend beurteilt von	Konfidenz (Stdabw.)
1. Nur Zeiger (pq_list_a2_3)	4 s	19.4 s (65.9 s)	57 (84 %)	75% (39%)
2. Nur Behälter mit Kopien (pq_list_a2_6)	10 s	11.2 s (10.5 s)	49 (72%)	75% (40%)
3. Zeiger und Behälter (pq_list_a2_8)	7 s	10.3 s (11.1 s)	54 (76.0 %)	80% (33%)
4. Etiketten und Behälter (pq_list_a2_7)	9 s	12.4 s (17.7 s)	54 (79%)	76% (36%)

Tab. 5: Beurteilung von Modellen zur Veranschaulichung von Zuweisungen. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 68 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

6.5 Namen als Bezeichnungen für Rollenträger

Namen haben Bedeutungen und sie werden in guten Programmen so gewählt, dass die Bedeutung erkennbar ist. In einem Funktions*aufruf* können als Argumente (Positionsargumente) andere Namen verwendet werden als in der Liste der formalen Parameter in der Funktions*definition*.

```
def quadrat (n):
   return n*n

seitenlaenge = input()
flaeche = quadrat (seitenlaenge)
```

Mehrfache Namen kommen auch im Alltag vor. Die selbe Person besitzt neben ihrem Namen, der im Personalausweis eingetragen ist, Spitznamen von Freunden und weitere Namen, die nur in bestimmten sozialen Kontexten verwendet werden. Verschiedene Namen für ein Objekt bringen meist zum Ausdruck, dass ein und dasselbe Objekt verschiedene Rollen spielt. Das Konzept des Namens ist eng mit dem Rollenkonzept verbunden, das heißt mit der Tatsache, dass ein Objekt in verschiedene „soziale“ Kontexte eingebunden sein kann.

Im obigen Programmbeispiel wird ein Zahlenobjekt, das zur Laufzeit über die Tastatur eingegeben wird, mit zwei unterschiedlichen Namen belegt. Im Hauptprogramm repräsentiert es die Seitenlänge eines Quadrats. Innerhalb der Definition der Funktion `quadrat()` wird das gleiche Objekt durch den Namen `n` referenziert. Hier wird das Objekt nicht mehr als Seitenlänge einer Fläche gesehen, sondern als abstraktes mathematisches Objekt. Es steht in einem anderen Kontext. Man könnte sagen, es spielt eine andere Rolle.

Namensgebung ist mit der Spezifikation von Rollen verbunden und somit ein wichtiger Teil der Modellbildung (siehe Abschnitt 13.1). Betrachtet man ein Programm als System von miteinander verwobenen Minimodellen, so wird über Mehrfachbenennung ein und dasselbe Objekt zum Bestandteil mehrerer Minimodelle.

6.6 Implizite Namen

Unter impliziten Namen verstehen wir solche Namen, die sich aus dem Kontext ergeben, aber nicht explizit „ausgesprochen“ werden. Beispiele aus dem Alltag sind folgende:

- Eine Stecknadel auf einer Landkarte markiert einen Ort (z.B. Städte, die man schon einmal besucht hat, Tatorte von Banküberfällen im Zuständigkeitsgebiet einer Kriminalkommission).
- Eine Unterstreichung markiert eine besonders wichtige Textstelle.

Abb. 24 zeigt Screenshots verschiedener Animationen aus der Python Visual Sandbox, die die „in place“-Sortierung einer Liste nach dem Verfahren der direkten Auswahl (straight selection) veranschaulichen.

```
s = [10, 4, 1, 3]
for i in range(len(s)):
```

```
for j in range(i+1, len(s)):
   if s [j] < s[i]:
      s[i], s[j] = s[j], s[i]
```

Die Modelle unterscheiden sich in der Implizität der verwendeten Namen. Die Laufvariablen `i` und `j` sind so genannte Stepper in Sajaniemis Systematik von Visualisierungen (Sajaniemi 2002, siehe auch Abschnitt 13.1). Sie werden im ersten Modell entsprechend seinem Vorschlag durch Papierstreifen mit Zahlen dargestellt, auf denen die aktuelle Zahl durch ein „Sichtfenster“ hervorgehoben wird. In den beiden folgenden Animationen werden i und j durch Zeiger und Marken repräsentiert. Ihre Rolle ergibt sich allein aus dem Kontext der Animation. Die vierte Animation geht noch einen Schritt weiter. Hier wird die Existenz von Steppern nicht explizit erwähnt sondern nur dadurch angedeutet, dass immer zwei Karten ein Stück aus der Kiste herausgezogen sind. Der bereits sortierte Teil der Liste wird mit einem Glaskasten „zum Schutz vor Zugriff“ abgedeckt.

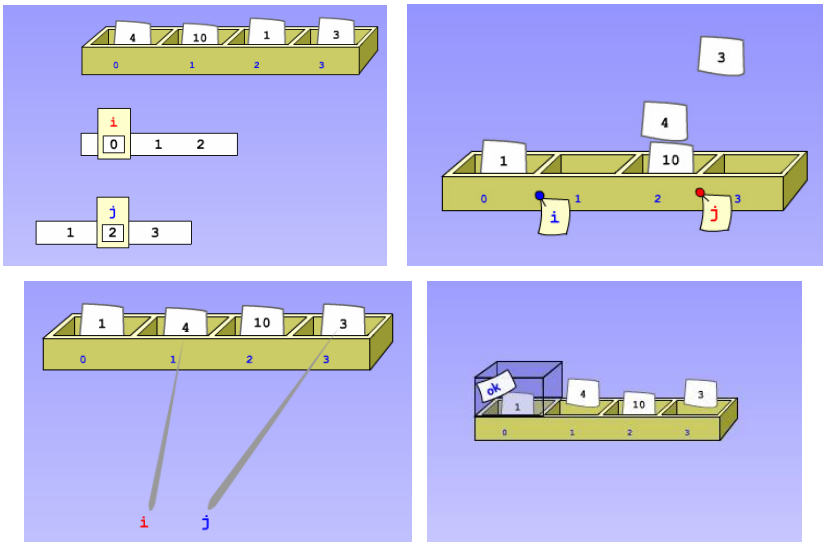


Abb. 24: Modelle für die Sortierung einer Liste nach dem Algorithmus straight selection (Screenshots)

In dem Python Visual wurden folgende Fragen gestellt:

- Welche Animation würden Sie verwenden um jemandem zu erklären, wie das Python-Skript funktioniert?
- An welche Animation können Sie sich am besten erinnern?
- An welche Animation denken Sie, wenn Sie sich die Ausführung des Skriptes vorstellen?

Tab. 6 zeigt die Antworten von 16 Personen, die an einem PVS-Workshop teilgenommen haben. Die Verteilung weicht signifikant von einer Gleichverteilung ab, die bei zufälligem Antworten zu erwarten wäre (χ^2-Test, $p = 0.0005$). Demnach wird das vierte Modell mit impliziten Namen bevorzugt. Zu beachten ist, dass diese Animation eine Information expliziert, die im Programmtext nicht direkt enthalten ist. Der bereits sortierte Teil der Liste wird nämlich markiert und mit „ok“ gekennzeichnet. Eine solche Benennung fehlt im Programmtext und muss aus der Semantik erschlossen werden. Möglicherweise ist der Erfolg von Modell 4 auf diese Zusatzinformation, die ein wichtiges Element der Idee des Algorithmus darstellt, zurückzuführen.

Das erste Modell, das die beiden Indexe i und j explizit durch eigene Ziffern darstellt, hat offenbar den geringsten Nutzen bei einer intellektuellen Auseinandersetzung mit dem Programm.

n = 16	Streifen	Stecknadeln	Zeiger	anheben
Erklären	2	5	0	9
Sich erinnern	1	8	2	5
Sich vorstellen	1	5	3	7
Summe	4	18	5	21

Tab. 6: Wahl verschiedener Modelle zur Veranschaulichung eines Programms.
Ergebnisse von 16 ersten Sitzungen während Workshops mit der PVS. Teilnehmer: 15 Schüler und ein Student aus Deutschland, 3 weiblich und 13 männlich, sie beschäftigten sich im Schnitt 2.5 Stunden pro Woche mit Programmierung und hatten ein Durchschnittsalter von 18.06 Jahren.

Um die Komplexität eines Modells zu reduzieren und seine Gestaltartigkeit zu erhalten, werden implizite Namen verwendet, die dann bei einer Formalisierung durch Programmtext expliziert werden müssen. Eine Aufgabe bei der Lösung einer Programmieraufgabe ist es, implizite Namen in einem intuitiven Lösungsmodell zu entdecken.

6.7 Indirekte Namen

Manchmal wird als Name für ein Objekt keine neue Wortschöpfung verwendet, sondern es wird aus bereits bekannten und verwendeten (belegten) Namen ein Name konstruiert. Der indirekte Name ist ein Name „über einen Umweg“ Er gibt einen Weg an, wie man das gemeinte Objekt identifizieren und ansprechen kann. Im Alltag wird z.B. die Phrase „Die älteste Tochter meines Bruders Andi“ zur Bezeichnung einer Person verwendet. Beispiele für indirekte Benennungen in Programmtexten sind

- Funktionsaufrufe, z.B. `sqrt(2)`
- Verwendung von Indizes bei Listen oder Schlüsseln bei Dictionaries, z.B. `s[2]` oder `englisch["Haus"]`
- Verwendung von Literalen.

6.7.1 Funktionsaufrufe und mathematische Terme

Bei Alltagsrechnungen wird die Wurzel aus zwei nicht explizit als Literal angegeben (was auch gar nicht geht) sondern der Name √2 verwendet. In Animationen, die eine komplexe Rechnung veranschaulichen, kann ein Zettel mit der Aufschrift `f(3)`, der an eine Karte mit einem Wert geheftet ist, eine Erinnerung sein, dass dieser Wert von der Funktion `f` nach Eingabe von `3` zurückgegeben worden ist. Beim Nachvollziehen der Arbeitsweise einer rekursiven Funktion kann es hilfreich sein, rekursive Funktionsaufrufe als Namen für ein Objekt zu verstehen, das zum aktuellen Zeitpunkt noch nicht bekannt ist. Betrachten wir die folgende Anweisung aus einer rekursiven Funktion, die die Fakultät berechnet:

```
return n* fak(n-1)
```

Es ist denkökonomisch, die Anweisung so zu verstehen: `fak(n-1)` ist ein Name für die Fakultät von `n-1`, also eine Zahl. Diese Zahl wird mit n multipliziert um, die Fakultät von `n` zu erhalten. Der einzige dynamische Aspekt bei dieser Interpretation ist die Multiplikation zweier Zahlen. Wenn ich dagegen `fak(n-1)` nicht als Name sondern dynamisch als Funktionsaufruf deute, muss ich in meiner Vorstellung unter Umständen viele rekursive Aufrufe durchspielen – eine gewaltige Denkleistung.

In der Mathematikdidaktik (Crowley et al. 1994, Knuth et al. 2006) wurde beobachtet, dass zuweilen mathematische Objekte ohne innere Dynamik dynamisch interpretiert werden. So fassen Kinder eine Gleichung vom Typ

```
x = a + b
```

als Rechenvorschrift auf, also als Anweisung, eine Addition auszuführen. Analog zu dieser Dynamisierungstendenz kann man in unserem Zusammenhang die Verwendung indirekter Namen als *Dedynamisierung* verstehen. Wiederum ist mit der Dedynamisierung eine Reduktion der Komplexität verbunden. Indem ich ein Objekt als `n!` (die Fakultät von `n`) bezeichne, erspare ich mir die Vorstellung den Wert der Fakultät tatsächlich ausrechnen zu müssen. Ich blende also einen Teil der Mechanik des

Modells aus. Der verbleibende Rest ist einfacher und intuitiver. Untersuchungen mit der PVS im Zusammenhang mit Rekursion zeigen jedoch, dass Funktionsaufrufe und Terme (trotz des kognitiven Vorteils) eher selten als Benennung verstanden werden (vgl. Abschnitte 8.8.4, 10.6).

6.7.2 Benennung durch Literale

Literale sind Zeichenketten, die Objekte repräsentieren. 1, 1.0, 1.00000, 1.0E0, 01 sind unterschiedliche Literale für das gleiche Objekt, nämlich die Zahl 1. Im Unterschied zum Literal ist eine Zahl ein ideelles mathematisches Objekt, das auch dann existiert, wenn es niemals aufgeschrieben oder sonst wie materiell repräsentiert wird. Programmiersprachen erlauben in der Regel, dass man Objekte über ein Literal ansprechen kann. In diesem Fall fungiert das Literal als indirekter Name. Beispiel: Mit `"klein geschrieben".upper()` wird an das Objekt, das durch das Literal `"klein geschrieben"` repräsentiert wird, eine Botschaft geschickt (Aufruf der Methode `upper()`). Gegenüber anderen Benennungsverfahren weisen Literale folgende Besonderheiten auf:

- Ein Literal für ein Objekt muss nach einem vorgegebenen Verfahren gebildet werden, das in der Grammatik der Programmiersprache spezifiziert ist.
- Ein Literal kann Informationen über den Typ des Objektes enthalten, das es bezeichnet. Bei Python z.B. bezeichnet ein Literal aus Dezimalziffern, das mit einer Ziffer ungleich null beginnt, immer eine ganze Zahl (Typ `int`).
- Explizite Namen können privates Wissen darstellen. Literale dagegen müssen von allen Objekten eines Systems interpretiert werden können. Sie sind immer öffentlich.

6.8 Assoziationen

In den bisher erwähnten Benennungsmodellen ist der Name etwas grundsätzlich anderes als das Objekt, das er benennt. Alle gängigen imperativen Programmiersprachen (Programmiersprachen, die Zuweisungen verwenden) gehen von dem Grundsatz aus, dass ein Bezeichner ein Objekt spezifiziert, aber nicht selbst ein Objekt ist. Bei Zuweisungen steht links vom Zuweisungsoperator ein Name (Bezeichner) `n` und rechts davon irgendeine Spezifikation eines Objektes `o`, die man sich als das Objekt selbst vorstellt. Nach Ausführung der der Zuweisung ist `n` ein Name für `o`. Die rechte Seite einer Zuweisung kann auch ein Name sein. Aber nicht dieser Name sondern das gemeinte Objekt wird neu benannt. Nach den Zuweisungen

```
a = 3
b = a
```

ist `b` nicht ein Name für den Namen `a` sondern ein Name für das Objekt `3`. Visualisierungsübungen mit Schülern zeigten jedoch, dass Zuweisungen manchmal als Herstellung einer Assoziation interpretiert werden. In Abb. 25 wird die Zuweisung

```
wort = "Hallo"
```

mit Vokabelnlernen verglichen. Das Ergebnis ist eine gedankliche Verbindung zwischen zwei Wörtern.

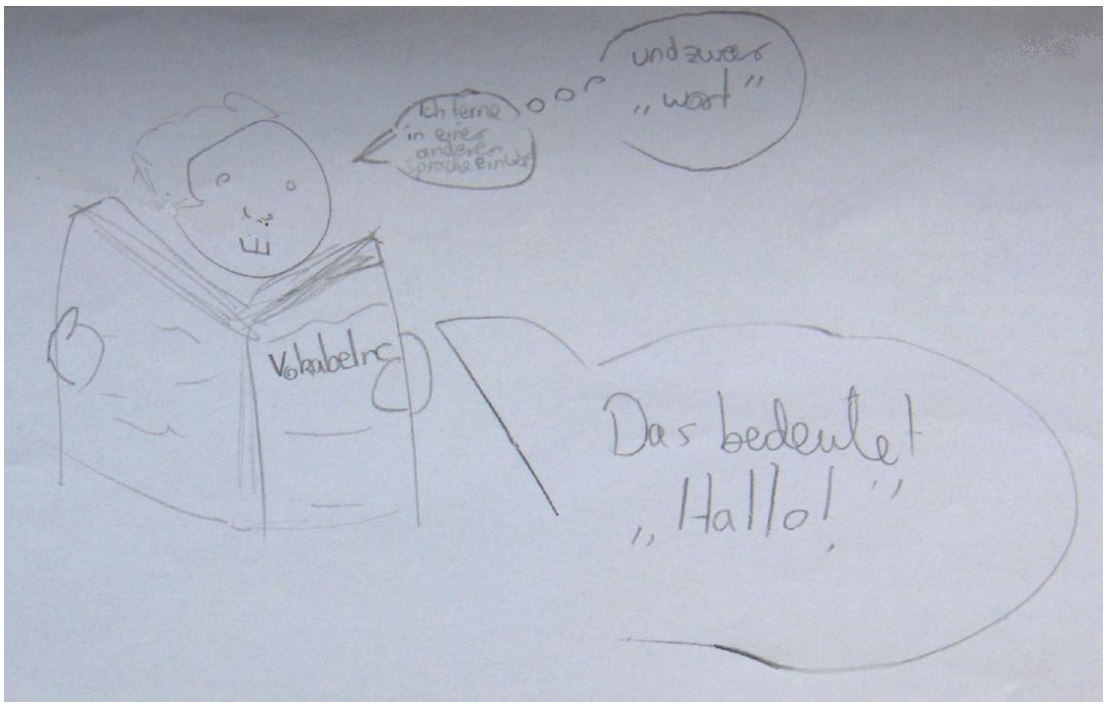


Abb. 25: Illustration des 17-jährigen Schülers T. (Jahrgangsstufe 11)

Die Unterschiede zwischen Assoziationen und Benennungen im Sinne einer Zuweisung sind folgende:

(1) Die assoziierten Objekte repräsentieren Inhalte und sind in diesem Sinne gleichartig. D.h. es wird nicht zwischen Namen und benanntem Objekt unterschieden. Die Vorstellung der Gleichartigkeit von Name und benanntem Objekt wird z.B. verwendet, wenn Personen Zuweisungsfolgen durch Kettenbildung modellieren. In diesem Fall wird der Zustand nach den Zuweisungen

```
a = 3
b = a
```

durch die Assoziationskette b → a → 3 dargestellt (siehe Abschnitt 10.5.3).

(2) Die Assoziation ist symmetrisch. Wenn `a` mit `b` assoziiert ist, dann auch `b` mit `a`. Bei Workshops mit der PVS konnte beobachtet werden, dass viele Schüler „umgedrehte Zeigermodelle" mit Zeigern, die vom Objekt zum Namen verweisen, für passende Visualisierungen von Zuweisungen halten (siehe Abschnitt 10.5.3).

(3) Assoziationen werden kumuliert. Das bedeutet zweierlei: Erstens wird mit der Entstehung einer neuen Assoziation eine bereits existierende nicht gelöscht. Zweitens werden durch eine neue Assoziation gleichzeitig noch weitere Einheiten miteinander vernetzt. Wenn `a` mit `b` assoziiert ist und `a` mit `c`, dann ist auch `b` mit `c` assoziiert. Dagegen besteht bei zwei Zuweisungen der Form

```
a = b
a = c
```

keine Verbindung zwischen den Objekten `b` und `c`. Außerdem wird mit der zweiten Zuweisung die Bindung von `a` an `b` aufgehoben.

6.9 Benennung als Unterordnung

Eine Zuweisung `wort = "Hallo"` wird manchmal auch als Unterordnung interpretiert (siehe Abb. 26). Name und benanntes Objekt stehen in einem hierarchischen Verhältnis zueinander. Damit wird dem Namen die Bedeutung eines Oberbegriffs zugemessen. Die untergeordneten Objekte sind Exemplare einer Kategorie, die im Namen zum Ausdruck kommt. So ist `"Hallo"` ein Beispiel für ein Wort.

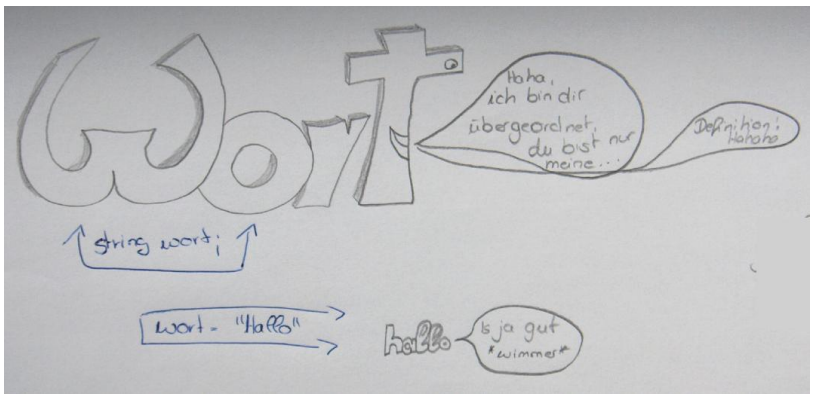


Abb. 26: Illustration der 17-Jährigen Schülerin J. zur Visualisierung der Unterordnung eines Objektes unter einen Namen

Die Interpretation einer Zuweisung als Unterordnung kann als Spezialfall einer Rückmodellierung interpretiert werden. Darunter verstehen wir den Versuch zu einem Programmtext einen Wirklichkeitsausschnitt (Original) zu finden, der durch den Programmtext modelliert wird. In der Tat modellieren Zuweisungen häufig Subsumierungssituationen. Betrachten wir als Beispiel die Aufgabe, einen Würfel zu modellieren. Ein Programm soll eine Zufallszahl zwischen 1 und 6 ausgeben. Etwas formaler ausgerückt lautet eine Teilaufgabe des Problems: „Erzeuge ein Objekt, das in die Kategorie *Zufallszahl zwischen 1 und 6* fällt." Das folgende Python-Programm löst die Aufgabe:

```
from random import *
zufallszahl = randint(1,6)
print zufallszahl
```

Das Objekt, das der Funktionsaufruf in der zweiten Zeile zurückgibt, ist eine ganze Zahl. Die Zuweisung ist hier tatsächlich das Modell der Unterordnung eines Objektes (Zahl) unter einen Begriff.

7 Daten

Unter Daten verstehen wir Information, die so repräsentiert ist, dass sie vom Computer verarbeitet werden kann. Aus Sicht der OOP kann man ein Datum als Zustand eines Objektes verstehen, der durch die Belegung der Attribute gegeben ist. Wir verwenden den Begriff Wert synonym für Datum (Singular von Daten).

7.1 Ansicht versus Literal

Daten werden in Programmtexten durch Literale dargestellt. Für die Bildung von Literalen gibt es Syntaxregeln in der Grammatik der Programmiersprache. Man kann zwischen der Ansicht eines Wertes und seiner Beschreibung durch ein Literal unterscheiden. Beide Begriffe beziehen sich auf die Repräsentation eines Wertes. Im Unterschied zum Literal ist die Ansicht eines Wertes unabhängig von der Programmiersprache. Ansichten sind Teil unserer Vorstellungswelt. Sie zeigen Daten, wie sie von uns wahrgenommen und verstanden werden. Sie repräsentieren die Bedeutung eines Literals. Sie sind somit kein informationstechnisches sondern ein psychologisches und kulturelles Phänomen.

Bei Zahlen ist häufig die Unterscheidung zwischen Literal und Ansicht schwierig. Mit 1 kann man sowohl ein abstraktes mathematisches Objekt (Ansicht) als auch ein Literal zur Repräsentation dieser Zahl meinen. Gleichwohl kann die Zahl 1 (Ansicht) auch durch andere Literale dargestellt werden, in Python z.B. `01`, `0x1`, `1.0`.

In der Interpretation von Programmtexten durch intuitive Modelle verwendet man Ansichten von Daten an Stelle der Literale, wenn dadurch die Intuitivität gesteigert werden kann. Hier spielen vor allem Abstaktion und Gestaltbildung eine Rolle. Die Ansicht eines Datums abstrahiert von Fragen der technischen Repräsentation und sie betont die Bedeutung, was die Bildung einer sinnhaften Gestalt erleichtert.

7.2 Verwechseln von Wert (Datum) und Literal

Der Unterschied zwischen Literal und Wert ist sehr feinsinnig. Bei numerischen Objekten ignoriert man ihn häufig ohne dass dies zu Problemen führt. Intuitiv setzt man Dezimalzahlen (Literale) mit den repräsentierten mathematischen Werten gleich und tut z.B. so, als ob es von einer Zahl mehrere Exemplare geben kann.

„An der Tafel stehen Zahlen, von denen einige gleich sind: 1, 2, 4, 1, 1“

„Schreibe eine Drei auf.“

In manchen Formulierungen wird jedoch tatsächlich zum Ausdruck gebracht, dass bestimmte Objekte (z.B. Zahlen) einmalig sind: „Nach der deutschen Rechtschreibung beginnt das Wort *Schule* mit einem großen Buchstaben.“ (Es gibt nur ein Wort *Schule*.) Es gibt Situationen, in denen die Differenzierung zwischen Literal und Wert von Bedeutung ist. Mathematische Operationen sind für Werte definiert und nicht für Literale. In den folgenden Beispielen, die in Python formuliert sind, liefert der Gleich-Operator == jedes Mal den Wert `True`, obwohl die Literale sich unterscheiden:

```
"Hall\0xf6" == "Hallo"
"Wort" == 'Wort'
1.0 == 1 == 0.1E1
010 == 8
```

7.3 Figürliche Ansichten

Intuitive Modelle von Programmen sind häufig kleine Beispielprogrammläufe, die die algorithmische Idee repräsentieren. Daten, die von dem Programm verarbeitet werden, sind austauschbar und kommen im zu interpretierenden Programmtext unter Umständen gar nicht vor.

Eine Möglichkeit die Intuitivität eines Modells zu verbessern ist die Verwendung von vertrauten, kognitiv gut zugänglichen Objekten zur Repräsentation von Daten. Wir nennen sie figürliche Ansich-

ten. Ein Beispiel ist, wenn man zur Visualisierung eines Sortieralgorithmus Figuren unterschiedlicher Größe als Platzhalter für Zahlen, Tupel oder andere Daten verwendet (Abb. 27).

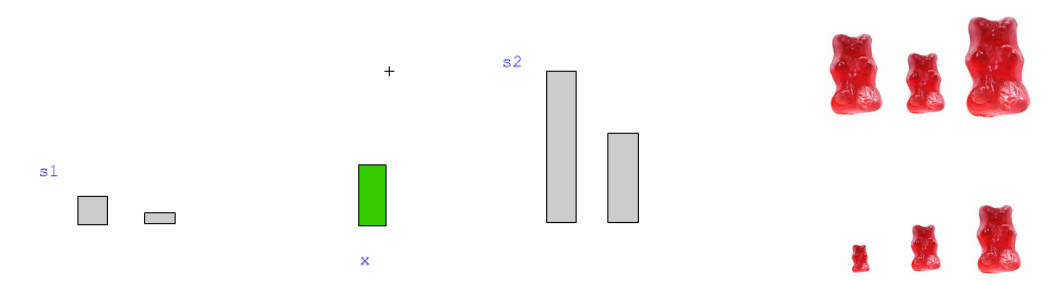


Abb. 27: Figuren als Platzhalter für Zahlen

7.4 Nichts

Ein besonderes Thema im Zusammenhang mit der Repräsentation von Daten ist die Repräsentation des Nichts. Leere Objekte können in verschiedenen Zusammenhängen eine wichtige Rolle spielen:

- Eine Prozedur kann als Funktion aufgefasst werden, die ein leeres Objekt (bei Python das Objekt `None`) zurückgibt. Andererseits gibt eine Funktion, die nicht terminiert, auch nichts zurück, doch dieses Nichts kann nicht durch ein Objekt „materialisiert" werden.
- Leere Objekte (z.B. Listen, Dictionaries oder Mengen) werden bei der Initialisierung eines Containers verwendet, in dem z.B. in einer Iteration bestimmte Daten gesammelt werden.
- Für eine leere Zeichenkette gibt es keine eindeutige Ansicht. Sie kann nur durch ein Literal (z.B. `""`) von einer Zeichenkette aus Whitespaces (Leerzeichen, Tabulatorzeichen etc.) unterschieden werden.

Beobachtungen mit der PVS zeigen, dass Schülerinnen und Schüler Probleme mit der Beurteilung von Visualisierungen leerer Objekte haben. Abb. 28 zeigt einige passende und unpassende Modelle für die leere Liste `[]` mit Länge 0. Das erste Modell ist bei Anwendung der sonst in der PVS verwendeten Bildersprache eine Liste mit einem leeren Element (z.B. leerer String) und damit unpassend (eine leere Liste enthält kein einziges Element). Das zweite Modell beschreibt die Python-Liste `[None, None, None, None, None]`, also eine Liste der Länge 5 mit fünf leeren Objekten und ist damit ebenfalls ungeeignet. Das dritte Bild kann als Behälter mit null Fächern interpretiert werden und ist passend. Das vierte Bild stellt einen Zeiger dar, der ins Leere (auf ein unbestimmtes Nichts) zeigt, ist somit weniger spezifisch, verstößt aber nicht wie die ersten beiden Modelle gegen Merkmale einer leeren Liste.

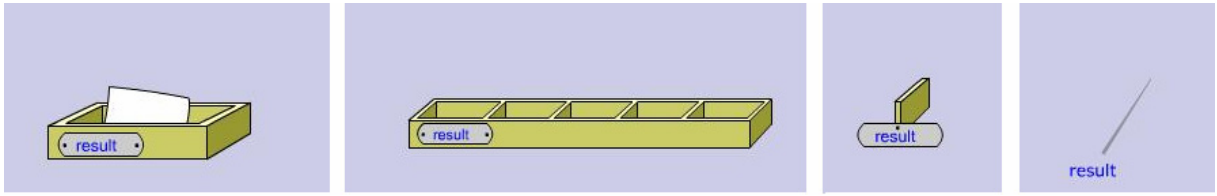


Abb. 28: Geeignete und ungeeignete intuitive Modelle zur Darstellung einer leeren Liste

Tab. 7 gibt die Einschätzungen von 68 Schülerinnen und Schülern wieder. Die Beurteilungen passender und unpassender Modelle unterscheiden sich kaum.

n = 68	Als passend beurteilt von	Konfidenz (Stdabw.)	Als unpassend beurteilt von	Konfidenz (Stdabw.)
Behälter mit leerem Zettel (pq_list_a1_1)	41 (60%)	61% (40%)	27 (40%)	68% (42%)
Behälter mit leeren Fächern (pq_list_a1_2)	47 (69%)	62% (45%)	21 (31%)	50% (45%)
Behälter ohne Fach (pq_list_a1_4)	41 (60%)	56% (45%)	27 (40%)	54% (46%)
Unbestimmtes Nichts (pq_list_a1_5)	44 (65%)	52% (46%)	24 (13%)	73% (39%)

Tab. 7: Beurteilung von Modellen zur Darstellung einer leeren Liste. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 68 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

7.5 Platzhalter für variable Teile in Dokumenten

In vielen Zusammenhängen verwenden wir die Vorstellung eines Dokumentes (z.B. Text), das aus konstanten (unveränderlichen) und variablen Teilen besteht. Die variablen Teile werden auf bestimmte Weise durch Platzhalter markiert und während der Interpretation des Dokumentes durch andere Zeichenfolgen ersetzt. Im Verlaufe dieses Ersetzungsprozesses entsteht sukzessive eine vollständig interpretierte Ausprägung des Dokumentes, in der keine Platzhalter mehr enthalten sind.

Die Programmiersprache PHP wird verwendet um „dynamische HTML-Seiten“ zu generieren. Die Idee ist, dass ein Webseite durch (statischen) HTML-Text definiert wird, der einige Stellen enthält, die erst zur Betrachtungszeit durch den PHP-Präprozessor mit Inhalt (d.h. HTML-Text) gefüllt werden. Im eigentlichen PHP-Programmtext wird algorithmisch festgelegt, durch welche Werte (Zeichenketten, die der HTML-Syntax entsprechen) die Platzhalter zum Zeitpunkt der Betrachtung ersetzt werden sollen.

Programmiersprachen wie C oder Python erlauben die Definition „formatierter Zeichenketten“. Anstatt eine Zeichenkette durch Konkatenation aus mehreren Teilen zusammenzusetzen, definiert man eine Zeichenkette mit Platzhaltern. Man gibt sozusagen ein Muster vor, von dem durch Einsetzungen unterschiedliche konkrete Ausprägungen erzeugt werden. Beispiel (Python):

```
>>> text1 ="%s geht nach %s." % ("Axel", "Berlin")
>>> print text1
Axel geht nach Berlin.
```

Das Beispiel zeigt eine Möglichkeit, wie variable Zeichenketten mit Python mit Platzhaltern spezifiziert werden können. Äußerlich ist ein Formatierungsplatzhalter bei Python eine Zeichenkette, die mit dem Prozentzeichen % beginnt. Danach kann in Klammern ein Variablenname folgen. Der Platzhalter endet schließlich mit einer Angabe zum Format der Texteinfügung. Dabei bedeutet s, dass es sich um einen String handeln muss. Insbesondere bei numerischen Werten (z.B. float) kann man weitere Angaben zur textuellen Darstellung machen wie z.B. Anzahl der Nachkommastellen oder Gesamtzahl der Stellen, die für den Wert reserviert werden sollen.

Die Auswertung eines Ausdrucks in einem Computerprogramm stellt man sich häufig so vor, dass sukzessive Subterme mit Variablen, Operatoren und Funktionsbezeichnern durch Literale ersetzt werden (siehe Abschnitt 5.4).

Platzhalter sind keine Namen für Objekte. Sie können aber Namen für Objekte enthalten. Der Name dient dazu, einen Wert zu identifizieren, durch den der Platzhalter bei der Interpretation ersetzt wird. Beispiel (Python):

```
>>> text2 = "%(name)s geht nach %(stadt)s." % vars()
>>> print text2
Axel geht nach Berlin.
```

Es gibt auch Platzhalter ohne explizite Namen. Im ersten Beispiel dieses Abschnitts ist die Position des Platzhalters im Dokument entscheidend für die Wertzuweisung.

Eine Verwechslung von Platzhaltern mit Namen liegt vor, wenn jemand glaubt, bei einer Zuweisung werde ein Name durch einen Wert ersetzt oder bei einer Neuzuweisung werde erst ein Objekt einschließlich seines Namens vernichtet (siehe Abschnitt 10.2.3).

7.6 Daten als Entitäten oder Zustände von Objekten

Ein Datum (Wert) kann man als eigene Entität begreifen (Entität-Modell), oder als Zustand eines Objektes (Zustand-Modell). Abb. 29 zeigt Screenshots aus drei Animationen der PVS, die die Ausführung der beiden Zuweisungen

```
a = 3
b = a
```

veranschaulichen. Im ersten Modell wird der Wert `3` der Variablen a durch einen beschrifteten Zettel repräsentiert. Bei die Zuweisung `b = a` entsteht eine Kopie des Zettels in Behälter a und fliegt in den Behälter b, der kurz zuvor ins Bild gekommen ist. Hier sind Daten separate Entitäten, die – wie materielle Dinge – kopiert und bewegt werden können.

In den beiden anderen Modellen wird ein Wert als Zustand eines Objektes dargestellt. Im mittleren Modell (pq_assign_a1_1) wird der Wert `3` durch die Position einer Drehschreibe an den Objekten a und b repräsentiert. Der Wert ist also keine eigene Entität, sondern immateriell und integraler Bestandteil eines Objektes. Die Zuweisung `b = a` wird dadurch visualisiert, dass Objekt b durch einen übergeordneten Akteur (Greifarm, der von oben ins Bild kommt und die Drehscheibe von `b` bewegt) in den gleichen Zustand wie Objekt a gebracht wird. Im rechten Modell (pq_assign_a1_3) werden zwei unterschiedliche Visualisierungen für den Zustand „3“ verwendet.

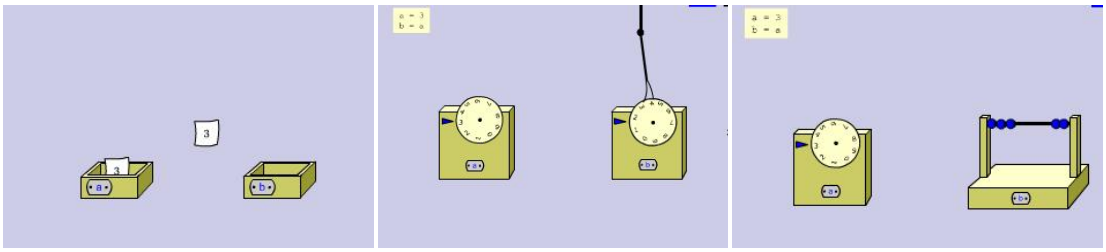


Abb. 29: Screenshots aus drei Animationen zur Visualisierung von Zuweisungen. Von links nach rechts: pq_assign_a1_5 (Entität-Modell), pq_assign_a1_1 (Zustand-Modell) und pq_assign_a1_3 (Zustand-Modell).

Die Mehrheit der beobachteten Schülerinnen und Schüler, die an PVS-Workshops teilgenommen haben, sehen alle drei Animationen als geeignete Visualisierungen (siehe Tab. 8). Allerdings hat die Animation, die ein Entitätsmodell für Daten verwendet, eine deutlich höhere Akzeptanz (86.4%) als die beiden Animationen mit Zustandsmodellen ($p < 0.0001$).

n = 154	**Dauer der Animation**	**Entscheidungszeit (Stdabw.)**	**Als passend beurteilt von**	**Konfidenz (Stdabw.)**
Wert (Datum) als eigene Entität (pq_assign_a1_5)	6 s	9.06 s (7.39)	133 (86.4%)	8.77 (2.76)
Wert als Zustand eines Objektes (pq_assign_a1_1)	14 s	19.93 s (10.96)	100 (64.9%)	8.30 (3.03)
Wert als Zustand eines Objektes (pq_assign_a1_3)	14 s	15.92 s (11.04)	103 (66.7%)	8.83 (2.54)

Tab. 8: Beurteilung von Modellen zur Veranschaulichung von Zuweisungen. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 154 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

Ein Problem des Zustand-Modells für Daten ist, dass bei einer Zuweisung der Form `b = a` der Bezug zwischen den beiden Variablen `a` und `b` nicht so einfach abgebildet werden kann. Beim Entität-

Modell (linke Animation in Abb. 30) entsteht eine Kopie des Inhalts von `a`, die zu b wandert. Dagegen wird bei den beiden Animationen, die ein Zustand-Modell verwenden, der Zustand von `b` auf „3“ eingestellt ohne dass expliziert wird, woher dieser Wert stammt.

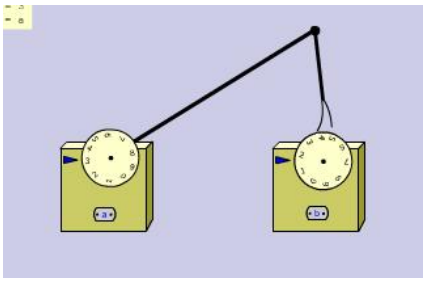


Abb. 30: Zuweisung b = a als Manipulation des Zustandes von Objekt b durch Objekt a.

Abb. 30 zeigt eine Möglichkeit, diesen „Datentransport“ von `a` nach `b` mit dem Zustand-Modell wiederzugeben. In dieser Animation manipuliert Objekt `a` mit Hilfe eines Greifarms den Zustand von `b`. D.h. die Zuweisung `b = a` wird als Auftrag an Objekt `a` interpretiert, seinen eigenen Zustand auf Objekt `b` zu übertragen. 108 (70.1 %) der 154 beobachteten Schülerinnen und Schüler hielten dieses Modell für geeignet.

8 Funktionen

Funktionen ermöglichen die strukturelle Zerlegung eines komplexen Problems in weniger komplexe Probleme und realisieren damit eine fundamentale Idee der Informatik (Schwill 1993). In der Mathematik definiert man eine Funktion als eine eindeutige Zuordnung der Elemente einer Menge A zu den Elementen einer Menge B. Jedem Element von A darf höchstens ein Element von B zugeordnet sein. In der Informatik wird eine Funktion meist als Akteur beschrieben, der eine Aufgabe lösen kann. In Programmtexten werden Funktionen als eigenständige Objekte oder Methoden einer Klasse definiert und können an anderer Stelle auf verschiedene Weise aufgerufen werden:

- Als eigenständiges aufrufbares Objekt, z.B. in der Anweisung `x = sqrt(2)`,
- als Botschaft an ein Objekt, z.B. `x = rechner.sqrt(2)`,
- als statische Methode einer Klasse, ohne dass zuvor ein Objekt instanziert worden ist, z.B. `x = Math.sqrt(2)`,
- über einen Operator in einem mathematischen Term, z.B. `x = y + 2`.

Aufrufe von Methoden können auch als Aufrufe eigenständiger Funktionen und nicht als Botschaften an Objekte betrachtet werden. Der Name des Objektes (oder bei statischen Methoden der Name der Klasse) wird dabei einfach als Teil des Funktionsnamens gesehen.

In der Definition einer Programmiersprache sind Syntax und Semantik von Funktionsaufrufen eindeutig festgelegt. Dennoch verwenden Programmierer bei Problemlösungen, bei der Interpretation von Programmtexten oder in der Kommunikation über Programme je nach Kontext unterschiedliche Cluster von intuitiven Modellen, die jeweils bestimmte Aspekte des „Phänomens“ Funktion repräsentieren. Das Statement

```
x = f(2)
```

kann z.B. auf folgende Weise gelesen und verstanden werden:

„Die Funktion mit dem Namen `f` wird aufgerufen und erhält als Eingabe die Zahl `2`. Dieser Wert wird von `f` verarbeitet und das Ergebnis (ein Wert) zurückgegeben. Dieser Wert wird der Variablen `x` zugewiesen.“

In dieser verbalen Beschreibung werden zumindest drei intuitive Modelle über Funktionen verwendet.

- Die Funktion wird aufgerufen. Hier wird unterstellt, dass die Funktion ein Akteur ist, der schon vor der Ausführung der Anweisung existiert und darauf wartet, aufgerufen zu werden. Etwa so wie ein Schraubenzieher in einer Werkstatt zur Benutzung bereit liegt oder ein Handwerker, den man zur Erledigung eines Auftrags anrufen kann.
- Der Funktion wird ein Wert übergeben. Dieses Konzept der Übergabe ist intuitiv und aus dem Alltag bekannt. Man übergibt einen Zettel an jemanden. Der übergebene Wert hat – wie ein Gegenstand – den Besitzer gewechselt und die Funktion kann als neuer Besitzer „damit machen was sie will“.
- In der Formulierung „zurückgeben“ steckt die Annahme, dass die Funktion selbst (und niemand anderes) dafür sorgt, dass das Berechnungsergebnis an einen bestimmten Akteur des Systems (nämlich den Auftraggeber) übergeben wird. Das impliziert, dass sie sich ihren Auftraggeber merken muss.

In anderen Zusammenhängen dagegen (z.B. bei der Interpretation rekursiver Funktionen) sieht man die Ausführung des Textfragments `f(2)` eher als Generierung eines neuen Akteurs (Prozess), der vorher noch nicht existierte und erst im Moment des Aufrufs nach dem Bauplan der Funktionsdefinition erstellt und aktiviert wird.

8.1 Funktion als Box mit Ein- und Ausgang für Daten

Eine populäre Metapher beschreibt eine Funktion als statische Einheit mit Ein- und Ausgang. Über den Eingang nimmt die Funktion Werte auf, verarbeitet diese und liefert über ihren Ausgang ein Ergebnis zurück. Ein- und Ausgang sind die einzigen Verbindungen (Schnittstellen) zur Umgebung. Die Datenübergänge werden analog zu einem Materialtransport dargestellt. Sobald die Ausgabedaten die Funktion verlassen haben, ist der Verarbeitungsprozess der Funktion beendet. Sie kümmert sich z.B. nicht mehr darum, ob die produzierten Daten den Empfänger erreichen.

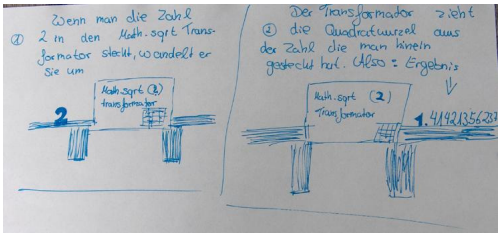


Abb. 31: Funktion (eigentlich statische Methode einer Klasse) als Materialverarbeitungseinheit mit Ein- und Ausgang. Visualisierung der 17-jährigen C.

Dieses Modell wird in datenflussorientierten visuellen Programmiersprachen wie LabView oder VIPER oder im Datenflussmodell der funktionalen Modellierung verwendet (siehe Abschnitt 5.1).

Eine (einfachere) Variante ist die Vorstellung eines „magischen Bechers“, der ein Objekt aufnimmt und das Berechungsergebnis aus der gleichen Öffnung wieder ausgibt. Diese Vorstellung klingt in der lexikalischen Metapher „zurückgeben“ an.

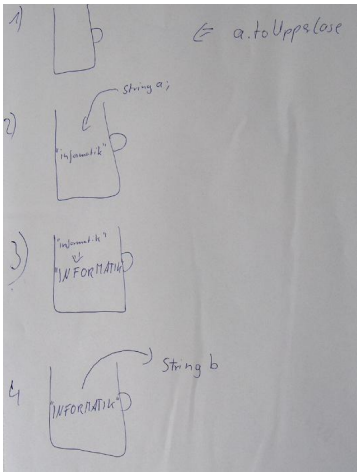


Abb. 32: Funktion (eigentlich Aufruf einer Methode) als „magischer Becher“, der Objekte „verwandelt“. Visualisierung des 17-jährigen Schülers M.

Man verwendet sie z.B. in folgenden sprachlichen Äußerungen:

- „Der Funktion `quadrat()` wird die Zahl `3` übergeben, sie liefert als Ergebnis die Zahl `9`“.
- „Beim Aufruf `f(3)` erhält die Funktion `f` den Wert `3`.“
- „Die Funktion `random()` liefert eine Zufallszahl zwischen `0` und `1`, wenn sie ohne Argument aufgerufen wird.“

Es gibt eine Reihe von Alltagsanalogien für das Datenflussmodell:

- Fahrscheinautomat: Man gibt Münzen ein und erhält eine Fahrkarte
- Toaster: Eingabe von Weißbrot, Ausgabe von heißem Toast.

Das Modell ist kompatibel mit dem Behältermodell für Variablen (Variablen als Behälter für Daten). Die Funktion nimmt einen Wert auf (z.B. Kopie eines Variableninhalts), behält diesen (er wird zu ihrem Eigentum) und liefert einen neuen Wert.

In einer Variante des Datenflussmodells bleiben die Eingabedaten im Eingang stecken. Auf diese Weise kann man später noch erkennen, mit welchen Werten die Funktion arbeitet. Das spielt bei dramatischen Modellen eine Rolle, in denen mehrere Funktionen interagieren und die Ausführung einer Funktion relativ lange dauert.

Das Datenflussmodell mit Ein- und Ausgang kann auch auf Prozeduren angewendet, also auf Funktionen, die nichts zurückgeben, und zwar in zweifachem Sinne:

- Eine Prozedur wird als Funktion interpretiert, die ein leeres Objekt zurückgibt.
- Wenn eine Prozedur ein Objekt verändert, so kann man sich das ursprüngliche Objekt als Eingabe und das veränderte Objekt als Ausgabe vorstellen. Die folgende Abbildung zeigt ein Datenflussmodell für eine Prozedur, die alle Elemente einer Liste von Zahlen quadriert (Python):

```
def quadriereListe (liste):
    for i in range(len(liste)):
          liste[i] =  liste[i] **2
```

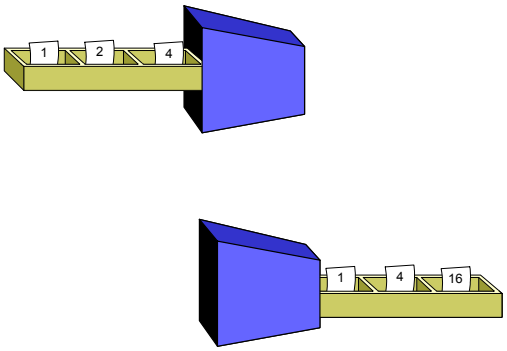


Abb. 33: Visualisierung einer Prozedur als Box mit Ein- und Ausgang

8.2 Dateneingabe über „Sensoren“

Zur Darstellung des Eingabemechanismus kann die Metapher eines Messgerätes mit Sensoren herangezogen werden. Alltagsanalogien sind:

- Ein Stück Indikatorpapier, das durch seine Farbe den pH-Wert einer Lösung anzeigt (einstellige Funktion)
- Ein Spannungsmessgerät, das die Potenzialdifferenz zwischen zwei elektrisch geladenen Gegenständen erfasst (zweistellige Funktion).
- Ein Finger, der die Temperatur eines Gegenstandes erfühlt (einstellige Funktion).

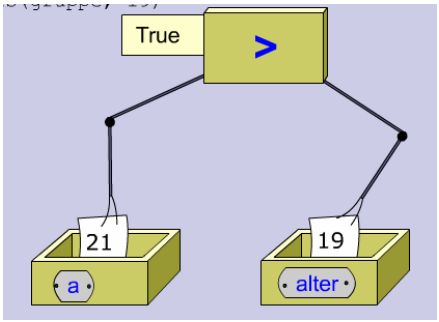


Abb. 34: Eingabe über Sensoren (PVS)

Der Eingabemechanismus dieses Modell verzichtet auf die Intuition eines Transports von Daten. Eíngabeobjekte bleiben außerhalb der Box. Über einen Greifarm oder ähnliches wird eine Verbindung

von der Funktion zum Objekt hergestellt. Der Eingabemechanismus ist ein Sensor, der einen Wert abtastet.

8.3 Übergabe von Referenzen bei der Eingabe

Funktionen können auf Objekte zugreifen, ohne sie sich einzuverleiben. Die Eingabeobjekte (die in der Parameterliste aufgeführt werden) bleiben außerhalb der Box.

Sofern die Funktion ein referenziertes Objekt auch verändert, *be*arbeitet sie es, anstatt es zu *ver*arbeiten. Der Zugriff kann z.B. durch Manipulatoren visualisiert werden, Greifer, die z.B. aus einer Liste Elemente herausnehmen und verändern. Insbesondere der konkurrente Zugriff mehrerer Funktionen auf ein Objekt kann auf diese Weise veranschaulicht werden. Eine Alltagsanalogie ist das Schnitzen, d.h. das Bearbeiten eines Holzstückes mit unterschiedlichen Messern.

Bei der Eingabe wird spezifiziert, auf welche Objekte die Funktion zugreifen kann („call by reference“). Diese Bindung kann auf unterschiedliche Weise visualisiert werden:

- Die Funktion wird in die Nähe des zu bearbeitenden Objektes gebracht (Analogien: Schnitzmesser, Messgerät, Zauberstab, Mauszeiger und Mausklick)
- Zu bearbeitende Objekte werden durch zusätzlichen Namen (Etikett, Pfeil) markiert. Eine Analogie aus der Forstwirtschaft ist die Markierung eines Baumes, der gefällt werden soll.
- Die Referenz wird durch eine Linie dargestellt.

Wenn Eingabeobjekte einfach sind, kann eine Referenz mit der Vorstellung des Abtastens von Werten gekoppelt werden. In der Mikrowelt Logotron (aus Lehotská, 2006) werden Referenzen durch gestrichelte Linien dargestellt, die von einem (numerischen) Objekt zu einem Eingang (kleine Raute) eines Operators führen (Abb. 35). Man kann sich vorstellen, dass über diese „Leitung“ der momentane Wert des „angeschlossenen“ Objektes „übertragen“ wird.

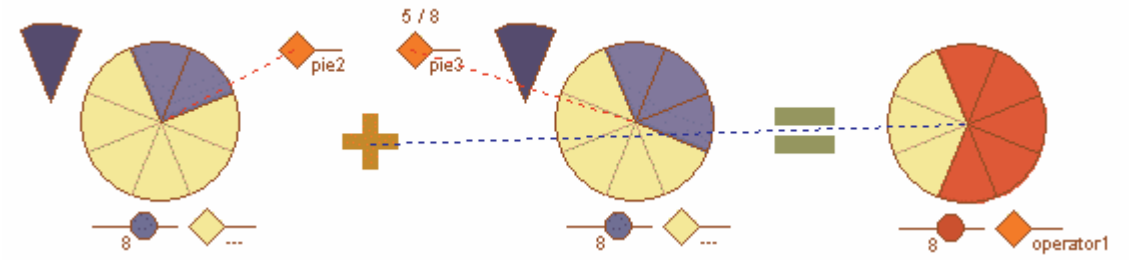


Abb. 35: Verknüpfung von Operatoren (Funktionen) und Objekten bei Logotron (aus Lehotská 2006)

8.4 Ursprung der Eingabespezifikation

Ein weiterer Aspekt bei der Modellierung von Eingabe ist die Frage, welche Entität die Argumente spezifiziert. Programmtechnisch ist der Fall klar: Beim Aufruf einer Funktion werden in Klammern die Argumente aufgeführt. Die Aktivität geht also vom aufrufenden Akteur aus. In Modellen zur Visualisierung von Programmabläufen kann das jedoch auch genau anders herum dargestellt werden: Die Funktion holt sich selbst die Eingabedaten. Dieser Fall liegt beispielsweise vor, wenn eine boolesche Funktion, die einen Größer-Vergleich realisiert, als Akteur mit Sensoren dargestellt wird und dieser Akteur über die Szenerie wandert und gezielt bestimmte Objekte abtastet und vergleicht.

8.5 Vergleich von Eingabemechanismen

Im Python Quiz *Modeling a group* geht es in Aufgabe 4 um die Visualisierung des Funktionsaufrufes

```
olderThan(group, 19).
```

Dabei beschreibt das erste Argument `group` eine Personengruppe durch eine Liste von Tupeln der Form `(name, alter)`. Das zweite Argument ist eine Altersangabe. Die Funktion liefert eine Liste

von Namen aller Personen, die älter sind. Es werden unter anderem folgende Modelle angeboten (siehe Abb. 36):

(1) Die erste Animation (pq_list_a5_1) verwendet ein Behältermodell für die Liste `group` und stellt die Funktion `olderThan()` als Kasten mit einem Greifarm dar. Diese Entität ist eigenaktiv und nimmt mit dem Greifarm zunächst eine Kopie aller Zettel aus der Liste group heraus und stopft sie in ihr Inneres. In gleicher Weise verleibt sie sich einen Zettel mit der Altersangabe `19` ein. Schließlich zieht der Greifarm aus dem Inneren des Kastens das Ergebnis (zwei Zettel mit Namen) heraus.

(2) In der zweiten Animation (pq_list_a5_4) wird die Funktion durch einen Kasten mit beweglichen Sensoren visualisiert. Die Sensoren nehmen mit der Liste und einem Zettel, der die Altersangabe `19` trägt, Kontakt auf. Danach kommen unten aus dem Funktionsobjekt zwei Zettel mit Namen heraus.

(3) Hier wird ein Eventmodell (Blitz) für den Funktionsaufruf verwendet. Aus dem Kasten, der die Liste darstellt, steigen Kopien der enthaltenen Zettel heraus. Sie treffen auf einen Zettel mit der Altersangabe `19`, mit einem Blitz verschwinden diese Zettel und es erscheinen stattdessen zwei neue Papierstücke mit den Namen der gesuchten Personen.

(4) Die Animation stellt die Funktion `olderThan()` durch ein Ein-Ausgang-Modell (umgedrehter Pyramidenstumpf) dar. In das Funktionsmodell fliegen Kopien aller Zettel aus der Liste und ein Zettel mit der Altersangabe `19`. Unten kommt das Ergebnis heraus, zwei Zettel mit Personennamen (Datenfluss).

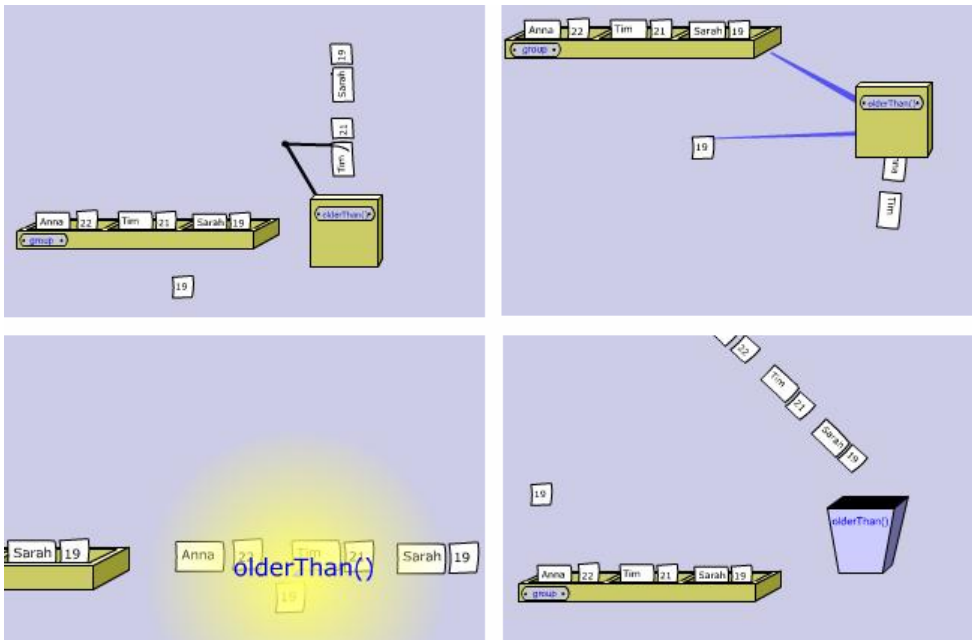


Abb. 36: Modelle mit unterschiedlichen Eingabemechanismen zur Visualisierung eines Funktionsaufrufs

Tab. 9 zeigt die Bewertungen dieser Modelle (passend/unpassend) von 68 Schülerinnen und Schülern, die an einem PVS-Workshop teilgenommen haben. Alle vier Modelle wurden von der Mehrheit akzeptiert, das besonders eigenaktive Modell mit Greifarmen sogar von 81 % der Spieler, allerdings ließ sich keine signifikante Bevorzugung gegenüber den anderen Modellen nachweisen.

n = 68	Dauer der Animation	Entscheiungszeit (Stdabw.)	Als passend beurteilt von	Konfidenz (Stdabw.)
Box mit Greifarmen (pq_list_a5_1)	14 s	12.75 s (12.56)	55 (81%)	76% (37%)
Box mit Sensoren (pq_list_a5_4)	9 s	7.75 s (6.09)	48 (71%)	72% (39%)
Ereignis (pq_list_a5_5)	6 s	8.01 s (6.32)	49 (72 %)	67% (40%)
Datenfluss (pq_list_a5_5)	9 s	10.37 s (9.02)	46 (68%)	73% (40%)

Tab. 9: Beurteilung von Modellen mit unterschiedlichen Eingabemechanismen für Funktionen. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 12 Schülerinnen und 56 Schülern, die an Workshops mit der PVS teilgenommen haben. Das Durchschnittsalter beträgt 17.15 Jahre. Die Teilnehmer haben sich im Durchschnitt 3.81 Stunden pro Woche mit Programmierung beschäftigt.

8.6 Übergabe von Referenzen bei der Ausgabe

Die Ausgabe einer Funktion kann darin bestehen, dass sie der aufrufenden Entität ein Objekt, das außerhalb der Funktion existiert, zeigt. Ein typisches Beispiel ist eine Suchfunktion, die aus einem großen Datenbestand ein bestimmtes Objekt heraussucht. Zurückgegeben wird nicht eine Kopie des gesuchten Objekts sondern eine Referenz darauf, so dass gezielt dieses Objekt verändert werden kann. Dieses Ausgabekonzept kann z.B. folgendermaßen visualisiert werden:

- Die aufrufende Entität hält ein Etikett (Namen) bereit, das von der Funktion (z.B. mit Hilfe eines Greifers) an das zurückzugebende Objekt geheftet wird. Das Problem ist, dass die Übergabe des Etiketts als weitere Eingabe missverstanden werden kann.
- Die aufrufende Entität hält einen Zeiger bereit, der mit einem Namen versehen ist und zunächst „ins Leere“ zeigt. Er wartet darauf, an ein Objekt gebunden zu werden. Wenn die Funktion ihre Rechnungen abgeschlossen hat, „manipuliert“ sie den Pfeil und sorgt dafür, dass er auf das richtige Objekt zeigt.

Beide Modelle für die Rückgabe einer Referenz basieren auf der Idee wartender Namen. D.h. die aufrufende Entität hält einen Namen bereit, der schon existiert, bevor die Funktion eine Referenz geliefert hat. Wenn man diesen wartenden Namen nicht voraussetzt, kommt man in logische Schwierigkeiten. Es ist dann so, als ob man jemandem etwas zeigen will, aber dieser Jemand gar nicht existiert.

Dies kann als Widerspruch zum Namenskonzept gesehen werden. Denn streng genommen muss ein Name immer an ein Objekt gebunden sein (sonst wäre es kein Name). Die Annahme eines wartenden Namens ist ein weiteres Beispiel für eine Überstrukturierung. Ein an sich atomarer Vorgang – die Bindung eines Namens an ein Objekt – wird aufgeteilt in zwei Vorgänge: Generierung des Namens und Bestimmung des zugehörigen Objektes (durch eine andere Entität).

Die PVS enthält eine Applikation (Python Visual *What happens, when a function returns something?*), in der das Problem der Rückgabe einer Referenz thematisiert wird. Gegeben ist ein Python-Programm mit einer Multiliste (Liste von Listen).

```
s = [[4], [5], [1]]
x = min(s)
x[0] = 10
print s
```

Es liefert folgende Ausgabe auf dem Bildschirm:

```
[[4], [5], [10]]
```

Das heißt, die Funktion `min()` gibt eine Referenz auf die kleinste Subliste (letztes Element), der in der zweiten Programmzeile der Name `x` zugeordnet wird. In der dritten Zeile wird das erste und einzige Element dieser Liste überschrieben.

Zur Erklärung dieses Verhaltens – und insbesondere des Rückgabemechanismus – werden vier visuelle Modelle angeboten. In allen Modellen wird die Multiliste als Behälter mit Behältern dargestellt. Die Funktion stellen wir durch eine Box dar, die die Liste mit Hilfe eines „Sensors“ (grauer Zeiger) „abtastet“. Die Modelle unterscheiden sich in der Art und Weise, wie die Funktion ihr Ergebnis zurückgibt.

(1) Die Funktion holt aus ihrem Inneren eine Nadel mit einem Zettel, der die Aufschrift `x` trägt und setzt ihn auf den Behälter der Unterliste mit dem kleinsten Inhalt (Abb. 37 oben links). Das Modell ist in sofern unpassend, als die Funktion den Namen `x`, mit dem sie ein Objekt markiert, ja gar nicht kennt.

(2) Die Funktion produziert eine Kopie der Unterliste mit dem kleinsten Inhalt und gibt sie zurück. Sowohl in der Kopie als auch im Original wird der Zettel mit der Zahl `1` gelöscht und durch einen Zettel mit der Zahl `10` ersetzt (Erscheinungsmodell, Abb. 37 oben rechts).

(3) Bevor die Funktion erscheint, ist bereits die Variable `x` im Bild angedeutet (Name `x` und Pfeil, der ins Leere zeigt). Aus der Funktionsbox kommt ein Greifarm, der die Spitze des Zeigers mit dem Namen x auf das kleinste Element der Liste zieht (Abb. 37 unten links).

(4) Bevor die Funktion erscheint, ist bereits die der Name `x` im Bild (Schild). Aus der Funktionsbox kommt ein Greifarm, der dieses Schild auf dem Behälter der kleinsten Unterliste befestigt (Abb. 37 unten rechts).

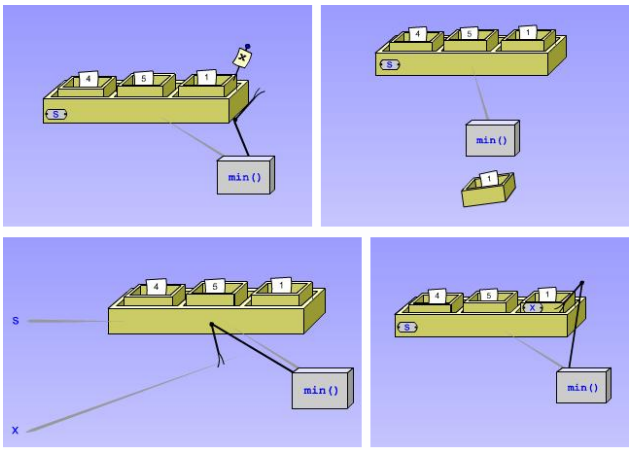


Abb. 37: Visuelle Modelle für die Rückgabe einer Referenz auf ein Objekt

Den Spielern wurden folgende Fragen gestellt:

- Welche Animation würden Sie verwenden, um jemandem zu erklären, wie das Python-Skript funktioniert?
- An welche Animation können Sie sich am besten erinnern?
- Welche Animation war am schwierigsten nachzuvollziehen?

Tab. 10 zeigt die Verteilung der Antworten von 23 Workshop-Teilnehmern. Es lässt sich aufgrund der geringen Teilnehmerzahlen keine deutliche Bevorzugung oder Ablehnung nachweisen (der χ^2-Test ergibt keine signifikante Abweichung von einer Gleichverteilung).

n = 23	Nadel mit x	Kopie der Liste	Zeiger	Neues Schild x
Erklären	6	6	6	5
Sich erinnern	5	6	4	8
Schwierig	10	7	3	3

Tab. 10: Wahl verschiedener Modelle zur Veranschaulichung der Rückgabe einer Referenz. Ergebnisse von 23 ersten Sitzungen während Workshops mit der PVS. Teilnehmer: 16 Schüler, ein Student und sechs Lehrer aus Deutschland (vier weiblich und 19 männlich), sie beschäftigten sich im Schnitt 5.5 Stunden pro Woche mit Programmierung und hatten ein Durchschnittsalter von 22,5 Jahren.

8.7 Ausgabe als Signal

Boolesche Funktionen (Tests) liefern einen Wahrheitswert, der meist zur Steuerung des Programmlaufs (in Programmverzweigungen oder Schleifen) verwendet wird. Die Besonderheit ist hier, dass der Name des ausgegebenen Objektes in der Regel irrelevant ist. In Programmtexten wird in solchen Fällen auch kein expliziter Name für das Testergebnis verwendet wird. Beispiel:

```
if a < b:
    …
```

Hier ist der Informationsgehalt des zurückgegebenen Objektes minimal. Das Ausgabeobjekt selbst tritt in den Hintergrund. Viel wichtiger ist der Aspekt, dass (im Beispiel, wenn das Ergebnis das Objekt `True` ist) eine bestimmte Aktionsfolge ausgelöst wird. Die Ausgabe wird als Signal zur Steuerung des Geschehens gesehen und kann dementsprechend veranschaulicht werden (z.B. als Schild, das für kurze Zeit sichtbar ist).

Abb. 38 zeigt Screenshots aus Animationen der PVS, die eine Testfunktion als Ereignisauslöser darstellen und sich auf folgende Python-Anweisung beziehen:

```
if a > age:
    …
```

Links wird die Testfunktion „größer als“ als Kasten dargestellt, der über „Sensoren“ die Inhalte zweier Variablen abfragt. Das Signal wird durch ein Schild (in diesem Fall mit der Aufschrift True) visualisiert, das aus dem Kasten herauskommt, für kurze Zeit sichtbar ist und dann wieder verschwindet (Boxmodell mit Sensoren und Signalausgabe).

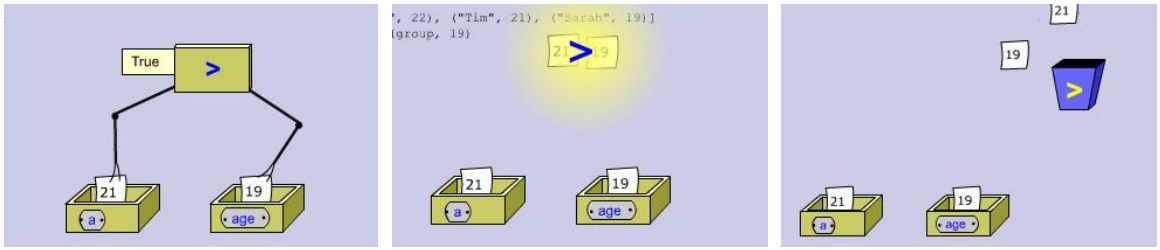


Abb. 38: Vergleich zweier Zahlen als steuerndes Ereignis. Screenshots aus Python Quiz „Modeling a group“.

Die zweite Animation zeigt die Funktionsausführung als Ereignis (Blitz), bei dem aus Kopien der Inhalte der Variablen `a` und `age` eine Karte mit einem Wahrheitswert entsteht.

Das dritte Bild zeigt ein Boxmodell mit Ein- und Ausgang. Bei den letzten beiden Modellen wird das ausgegebene Objekt (eine Karte mit einem Wahrheitswert) zu keiner Entität weitergeleitet, sondern bleibt eine kurze Weile sichtbar und verschwindet wieder. Es hat damit den Charakter eines Signals.

Tab. 11 zeigt, dass Schülerinnen und Schüler alle drei Modelle in etwa mit gleicher Häufigkeit als passend beurteilen. Insbesondere kann man (vorsichtig) den Schluss ziehen, dass die explizite Modellierung eines Signals wie im ersten Modell keinen kognitiven Vorteil gegenüber Funktionsmodellen

bietet, die ein Objekt ausgeben, das im Prinzip auch an einen Akteur zur Verarbeitung weitergegeben werden kann – also per se kein Signal darstellt.

n = 68	Dauer der Animation	Entscheidungs-zeit (Stdabw.)	Als passend beurteilt von	Konfidenz (Stdabw.)
1. Boxmodell mit Sensoren (pq_list_a3_4)	8 s	10.65 s (7.18)	58 (85.3%)	74.1% (30.8%)
2. Blitz (pq_list_a3_2)	7 s	7.88 s (5.58)	58 (85.3%)	76.7% (37.7%)
3. Boxmodell mit Ein- und Ausgang (pq_list_a3_1)	9 s	14.84 s (25.25)	56 (82.35 %)	72.3% (4.04%)

Tab. 11: Beurteilung von Modellen für Testfunktionen, deren Ausgabe als Signal interpretiert wird. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 68 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

Ein Modell mit einer booleschen Funktion, die einen Wahrheitswert liefert, kann nur zweiseitige Entscheidungen darstellen. Bei verschachtelten Entscheidungsstrukturen muss ein Ereignismodell alle vorkommenden Fälle als logische Aussagen explizieren. Es gibt eine Entität, die den Systemzustand prüft und ein Ereignis (z.B. `a < b` oder `a == b` oder `a > b`) generiert, auf das jeweils ein Akteur reagiert.

8.8 Durchlässigkeit der Systemgrenze

8.8.1 Geschlossene Box

Ein geschlossenes Modell einer Funktion lässt einen Datenaustausch mit der Umgebung nur über „kontrollierte“ Ein- und Ausgänge zu. Probleme mit dem geschlossenen Ein-/Ausgabemodell treten dann auf, wenn man einen Zugriff der Funktion auf Objekte darstellen will, die außerhalb der Funktion existieren (z.B. Zugriff auf globale Variablen oder Aufruf von anderen Funktionen).

8.8.2 Box mit „Seitentür“

Um die eben erwähnten Probleme zu lösen, kann das geschlossene Box-Modell um eine „Seitentür“ erweitert werden, über die die Maschinerie der Funktion mit ihrer Umgebung Kontakt hat. Sie stellt neben Ein- und Ausgang einen weiteren Kommunikationskanal dar. Wenn eine Funktion eine andere Funktion aufruft, so kann man dies so darstellen: Über die Seitentür „wandert“ ein Objekt – das Argument des Funktionsaufrufs – nach draußen in den Eingang der aufgerufenen Funktion. Sie liefert (über ihren Ausgang) ein Ergebnis, das durch den Seiteneingang in das Innere der ersten Funktion gelangt und dort in die weitere Rechnung einbezogen wird (Abb. 39 links). Boxen mit Seitentüren werden in er PVS als Veranschaulichung einer rekursiven Funktion zur Berechnung von Fibonacci-Zahlen verwendet (siehe Abschnitt 8.9).

8.8.3 Direkter Zugriff auf externe Objekte

Seiteneffekte und die Verwendung globaler Variablen kann man durch die Metapher eines „Manipulators“ darstellen, der Objekte verändern kann, die sich außerhalb der Funktionsbox befinden. Damit ist die Systemgrenze sehr durchlässig. Kooperierende Akteure, die auf ihre Umgebung Einfluss nehmen, kann man beobachten, wenn eine Gruppe von Menschen gemeinsam eine handwerkliche Aufgabe mit verteilten Rollen löst. Betrachten wir als Beispiel die Situation nach einem Essen. Jemand räumt das Geschirr vom Tisch, eine andere Person spült, eine dritte räumt ab und eine vierte Person räumt das Geschirr in den Schrank. Die von den verschiedenen Akteuren verarbeiteten Objekte befinden sich sämtlich in einer gemeinsamen, öffentlichen Sphäre.

Kommen wir wieder zurück zu Programmtexten. Die Manipulatormetapher kann auch dann für Funktionen verwendet werden, wenn überhaupt kein Zugriff auf externe Objekte erfolgt. Die PVS enthält ein Beispiel zur Visualisierung einer rekursiven Funktion, die eine Zeichenkette spiegelt. Obwohl auf programmtechnischer Ebene in jedem rekursiven Aufruf ein separates String-Objekt verar-

beitet wird, geht die Animation von einem einzigen String aus (Folge von Buchstabenkarten), dessen Zeichen von den verschiedenen Akteuren (die Funktionsaufrufe repräsentieren) bewegt werden (siehe Abb. 39 rechts).

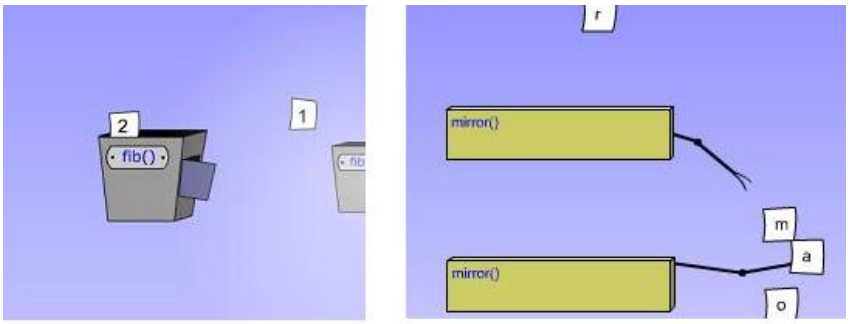


Abb. 39: Modelle für Funktionen mit offener Systemgrenze

8.8.4 Vergleich von offenen und geschlossenen Funktionsmodellen

Stellt man eine Funktion als geschlossene Box mit Ein- und Ausgang dar, so muss bei einem internen Funktionsaufruf der zugehörige Akteur sich im Inneren befinden. Er ergibt sich eine geschachtelte Struktur von Boxen, die andere (kleinere) Boxen enthalten (siehe Abb. 40). Das Problem einer solchen Visualisierung ist, dass die inneren Boxen und schon bei geringer Rekursionstiefe nicht mehr sichtbar sind. Andererseits werden geschachtelte Strukturen (wie Russian Dolls) mit Rekursion assoziiert. Bei offenen Modellen mit durchlässiger Systemgrenze können gleichgroße Modelle für Funktionen verwendet werden.

In Python Visual Mirrors wird die Ausführung einer rekursiven Funktion veranschaulicht, die eine Zeichenkette spiegelt (aus `roma` wird `amor`). In den Animationen werden offene und geschlossene Modelle einander gegenüber gestellt (Abb. 40):

```
def mirror (w):
   if w== "": return w
   else: return mirror(w[1:]) + w[0]

print mirror "roma"
```

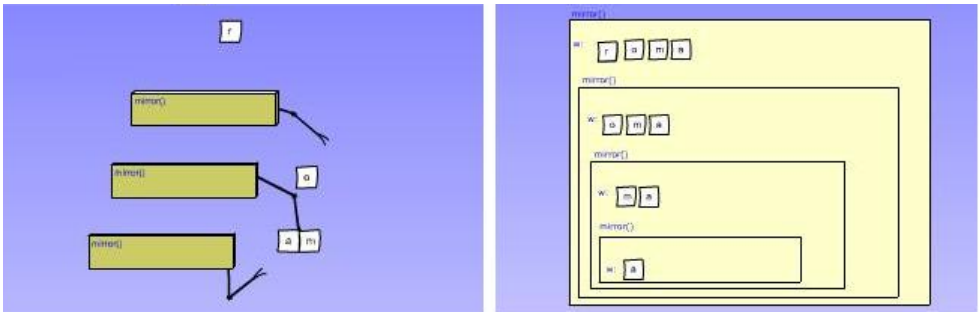


Abb. 40: Offene und geschlossene Modelle zur Visualisierung der Ausführung einer rekursiven Funktion.

(1) Die erste Animation zeigt nur einen Rekursionsschritt des Programmlaufs. Der zu einem Funktionsaufruf gehörende Prozess wird als Brett mit der Aufschrift `mirror()` visualisiert. Es besitzt einen Greifer, der Zettel manipuliert. Ein derartiger Akteur erhält vier Zettel mit den Buchstaben des Wortes `"roma"`. Der Greifer nimmt die hinteren drei Zettel (`"oma"`) und übergibt sie einem zweiten Akteur (rekursiver Aufruf der Funktion). Dieser vertauscht die Buchstaben, so dass ihre Reihenfolge umgekehrt wird und gibt das Ergebnis zurück (`"amo"`). Der erste Akteur nimmt diese drei Zettel in Empfang, setzt den Buchstaben `"r"` hinten an und gibt das Ergebnis (`"amor"`) nach oben zurück.

(2) Das zweite Modell entspricht dem ersten, stellt jedoch die vollständige Rekursion (vier Funktionsaufrufe) bis zum Rekursionsabbruch dar (Abb. 40 links).

(3) Das dritte Modell stellt die vollständige Rekursion (vier Funktionsaufrufe) bis zum Rekursionsabbruch dar. Rekursive Funktionsaufrufe werden durch geschachtelte Kästen (Rahmen) visualisiert. Zettel mit Buchstaben fliegen in die Kästen (Eingabe) und werden innnerhalb des Kastens verarbeitet. Das Ergebnis fliegt wieder heraus und der betreffende Rahmen verschwindet.

(4) Die letzte Animation stellt wie Modell 3 den Prozess, der zu einem Funktionsaufruf gehört, durch einen Rahmen dar. Allerdings wird hier nur ein rekursiver Funktionsaufruf explizit visualisiert.

Den Spielern wurden folgende Fragen gestellt:

- Welche Animation würden Sie verwenden, um jemandem zu erklären, wie das Python-Skript funktioniert?
- Welche Animation gibt am besten die Idee einer rekursiven Funktion wieder?
- Welche Animation war am schwierigsten nachzuvollziehen?

Tab. 12 zeigt das Ergebnis aus 30 Sessions, von denen aber nur 22 im Rahmen von Workshops mit der PVS stattfanden. Es weicht signifikant von einer Gleichverteilung ab (χ^2-Test, $p = 0.0003$). Offene und geschlossene Modelle wurden etwa gleich bewertet. Auffällig ist, dass die vollständige Darstellung der Funktionsausführung bis zum Erreichen des Trivialfalls und Abbruch der Rekursion (Modelle 2 und 3) einer verkürzten Darstellung vorgezogen wird – ein Punkt, auf den wir in Abschnitt 9.6 im Zusammenhang mit Rekursion zurückkommen werden.

n = 30	**offen, ein Schritt**	**offen, volle Rekursion**	**geschlossen, volle Rekursion**	**geschlossen, ein Schritt**
Erklären	3	12	11	4
Idee	2	11	14	3
Schwierig	10	7	4	9

Tab. 12: Wahl verschiedener Modelle zur Veranschaulichung der Rückgabe einer Referenz. Ergebnisse von 30 ersten Sitzungen, davon 22 während Workshops mit der PVS. Teilnehmer: 22 Schüler, vier Studenten und vier andere Personen, davon 24 aus Deutschland. Es gibt 26 männliche und 4 weibliche Teilnehmer; sie beschäftigten sich im Schnitt 4.5 Stunden pro Woche mit Programmierung und hatten ein Durchschnittsalter von 21.2 Jahren.

8.9 Dynamische und statische Funktionsmodelle

Ein statisches Funktionsmodell betrachtet eine Funktion als Entität, die während der gesamten Lebenszeit des Programms (Prozesses) existiert und in Bereitschaft ist, aufgerufen zu werden. In Datenflussmodellen (siehe Abschnitt 5.1) geht man davon aus, dass jede benötigte Funktion genau einmal existiert. Es gibt einen gewissen Vorrat an vorgefertigten Funktionen, die bereits zu Beginn des Programmlaufs sichtbar sind.

Ein Problem tritt z.B. bei rekursiven Funktionen auf. Von rekursiven Funktionen müssten in dieser statischen Sichtweise im Prinzip unendlich viele Exemplare existieren. Im „Schleifenmodell“ endrekursiver Funktionen (Kahney 1984, Close & Dicheva 1997) stellt man sich eine statische Funktion vor, die immer wieder (mit neuen Parametern) aufgerufen wird.

Ein dynamisches Funktionsmodell repräsentiert im Prinzip den Prozess der Funktion, der durch einen Aufruf gestartet wird. Das Funktionsmodell erscheint also erst bei Aufruf der Funktion in der Szene und verschwindet wieder nach Beendigung.

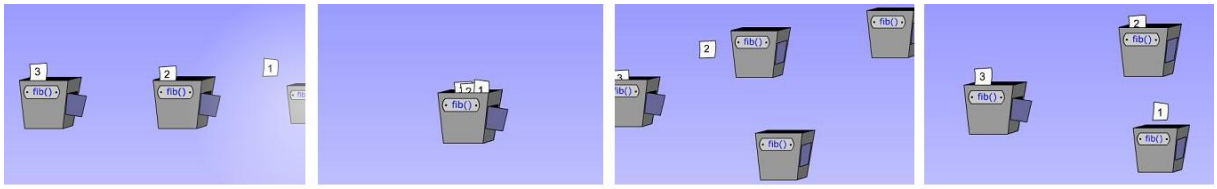


Abb. 41: Funktionen als Boxen mit Ein/Ausgang und Seitentüren (PVS)

Abb. 41 zeigt Screenshots aus Animationen der PVS, die die Ausführung einer rekursiven Funktion zur Berechnung von Fibonacci-Zahlen veranschaulichen und Boxen mit Eingang (oben), Ausgang (unten) und Seitentüren verwenden. Das Argument des Funktionsaufrufes wandert von oben in den Eingang, bleibt aber dort stecken bis der Berechnungsvorgang abgeschlossen ist. Alle Animationen fokussieren also auf die Struktur der rekursiven Aufrufe und lassen interne Berechnungen weg. Das vorgegebene Python-Programm lautet

```
def fib(n):
    if n<= 1: return n
    else: return fib(n-1) + fib(n-2)

print fib(3)
```

(1) Die erste Animation (links) enthält ein dynamisches Akteurmodell. Bei jedem rekursiven Funktionsaufruf entsteht mit einem Blitz ein neuer Akteur. Durch die Seitentür der aufrufenden Entität wandert eine Karte mit einer Zahl in den Eingang der soeben entstandenen Box. Diese führt Berechnungen aus, muss dabei wieder Funktionsaufrufe durch ihre Seitentür initiieren und gibt schließlich über ihre untere Öffnung (Ausgang) eine Karte mit dem Ergebnis aus, das durch die Seitentür in die Box der aufrufenden Funktion wandert. Sobald eine Box ein Ergebnis geliefert hat, verschwindet sie wieder. Alle Aktionen laufen im ersten Modell streng seriell. Das heißt beim ersten Aufruf `fib(3)` wird zuerst rekursiv `fib(2)` aufgerufen (Karte mit Aufschrift 2 wandert durch die Seitentür nach draußen). Erst wenn das Ergebnis (Karte mit Aufschrift 2) durch die Seitentür zurückgekommen ist, wird der zweite rekursive Aufruf `fib(1)` ausgelöst d.h. eine Karte mit Aufschrift `1` wird durch die Seitentür verschickt und durch eine neue Box verarbeitet und so weiter.

(2) In der zweiten Animation gibt es nur einen einzigen Akteur, der die Funktion `fib()` repräsentiert. Ein rekursiver Funktionsaufruf wird so dargestellt, dass eine Karte mit dem Argument durch die Seitentür die Box verlässt und nach oben zum Eingang wandert. Da die Karten mit den Argumenten der Funktionsaufrufe erst verschwinden, wenn eine Berechnung abgeschlossen ist, sammeln sich am Eingang Karten mit Zahlen und erinnern an noch laufende Berechnungsprozesse.

(3) Im dritten Modell sind die Funktionsakteure von Beginn an vorhanden (statisches Akteurmodell) und warten auf ihren Einsatz. Von der Funktion `fib()` gibt es also einen unerschöpflichen Vorrat von Exemplaren. Jede Zahlenkarte (Argument eines Aufrufs der Funktion `fib()`) sucht sich ihren Weg zu einer noch freien Box, die daran erkennbar ist, dass in ihrem Eingang noch keine Karte steckt. Die Ausführung der rekursiven Aufrufe ist wieder seriell wie beim ersten Modell.

(4) Die vierte Animation verwendet wie Modell 1 ein dynamisches Akteurmodell, die rekursiven Aufrufe werden aber parallel ausgeführt. D.h. aus der Seitentür der Box für den Aufruf `fib(3)` wandern kurz hintereinander Karten mit den Argumenten für die rekursiven Aufrufe `fib(2)` und `fib(1)` und werden in quasi gleichzeitig erscheinenden neuen Akteuren nebenläufig verarbeitet.

Den Spielern wurden die gleichen Fragen gestellt wie im Python Visual aus Abschnitt 8.8.4. Tab. 13 zeigt die Verteilung der Antworten von 21 Workshop-Teilnehmern. Sie weicht signifikant von einer Gleichverteilung ab (χ^2-Test, $p = 0.05$). Die Daten zeigen eine Bevorzugung dynamischer und Ablehnung statischer Varianten des Seitentür-Modells. Insbesondere das (realitätsferne) Modell, das eine parallele Ausführung rekursiver Funktionsaufrufe darstellt, wurde für Erklärungen als besonders geeignet angesehen.

n = 21	**Dynamisch, seriell**	**Statisch, ein Akteur**	**Statisch, viele Akteure**	**Dynamisch, parallel**
Erklären	5	3	3	10
Idee	8	4	3	6
Schwierig	5	6	8	2

Tab. 13: Wahl verschiedener Modelle zur Veranschaulichung einer rekursiven Funktion. Ergebnisse von 21 ersten Sitzungen während Workshops mit der PVS. Teilnehmer: 19 Schüler, ein Student und ein Lehrer aus Deutschland (7 weiblich und 14 männlich), sie beschäftigten sich im Schnitt 1,6 Stunden pro Woche mit Programmierung und hatten ein Durchschnittsalter von 18.2 Jahren.

8.10 Auslösemechanismen

Wann wird eine Funktion aktiv und beginnt mit der Verarbeitung der ihr übergebenen Daten? Modelle für Funktionen unterscheiden sich in der Art und Weise, wie sie die Aktivierung einer Funktion veranschaulichen (Auslösemechanismen). Oft wird eine implizite Auslösung verwendet. Die Funktion beginnt „automatisch“ mit der Verarbeitung, sobald sie alle Daten empfangen hat. Über andere Formen der Auslösung braucht man nur bei statischen Funktionsmodellen nachdenken, die keine Eingabedaten verarbeiten (nullstellige Funktionen). Ein Beispiel für eine Funktion ohne Eingabedaten ist ein Generator für Zufallszahlen.

Die visuelle Darstellung einer Funktion wird häufig allein auf ihre Auslösung reduziert. Die Ausführung der Funktion wird als Ereignis dargestellt, das z.B. als Darstellung als Blitz, der eventuell noch mit dem Funktionsnamen beschriftet ist, visualisiert werden kann. Solche Ereignisse bewirken dann die Umwandlung der Argumente des Funktionsaufrufs in das Ergebnis. Eine ähnliche Vorstellung verwendet man bei der Erklärung chemischer Reaktionen: Ausgangsstoffe „verwandeln sich“ unter bestimmten Bedingungen in neue Stoffe. Auch hier reduziert man die Darstellung eines an sich komplexen Ablaufs auf seine Auslösung.

9 Kontrolle – Steuerung

Kontrollstrukturen regeln die Reihenfolge, in der Operationen in einem Programm ausgeführt werden. Im Grunde ist ein Programmtext mit Kontrollinformation vollkommen durchdrungen. In einem imperativen Programm ist jede Anweisung zumindest Teil einer Sequenz, wird also vor oder nach einer anderen Anweisung ausgeführt. Nancy Pennington (1987) beschreibt Kontrolle als eine „Abstraktion" eines Programmtextes, die neben den Abstraktionen Datenfluss und Funktionalität eine Dimension des Verstehens eines Programms ist[8].

9.1 Handhabung von Kontrolle: Kontrollfluss und Kontrollübergabe

Kontrolle kann man sich als Entität vorstellen, die auf geregelte Weise von Akteur zu Akteur bewegt wird. Nur die Entität, die gerade die Kontrolle besitzt, ist aktiv. Im Bereich der Datenkommunikation sind Kontroll-Entitäten expliziter Bestandteil von Algorithmen, die den Zugriff eines Senders auf das Übertragungsmedium regeln. Im Token-Bus (IEEE 802.4) und Token-Ring (IEEE-802.5) wandert ein Token (eine bestimmte Bitfolge) von Station zu Station. Wer das Token besitzt, darf senden. Auch im Alltag gibt es Gegenstände, die Kontrolle repräsentieren. Bei einem Staffellauf markiert das Staffelholz den Läufer, der gerade aktiv ist.

Im Kontrollflussmodell wandert die Kontrolle auf vorgegebenen Bahnen. Kontrollflüsse werden in Flussdiagrammen oder bestimmten visuellen Programmiersprachen wie Labview (National Intruments) verwendet. Bei der Interpretation eines Flussdiagramms geht man davon aus, dass zu einem Zeitpunkt nur eine Entität aktiv sein darf, also nur eine bestimmte Anweisung ausgeführt wird. Man kann den Kontrollfluss nachspielen, indem man mit dem Finger den Bahnen des Flussdiagramms folgt.

Das Kontrollflusskonzept ist grundlegend und wird in intuitiven Modellen für Sequenzen, Schleifen und Programmverzweigungen zur Modellierung des Programmlaufs verwendet.

Den Aufruf einer Funktion dagegen kann man durch Übergabe der Kontrolle von einem Akteur an einen anderen modellieren. Nach der Abgabe wartet der aufrufende Akteur. Wenn die aufgerufene Funktion terminiert, gibt sie die Kontrolle an die aufrufende Einheit zurück. Hier wird die Bewegung der Kontroll-Entität nicht durch ein räumliches Gebilde (eine Bahn) dargestellt. Stattdessen beschreibt man die Handhabung der Kontrolle als soziales Phänomen. Geben und Annehmen von Kontrolle ist Teil der Aktivität interagierender Entitäten.

Es gibt auch Darstellungsformen für Steuerungsprozesse, die keine explizite Kontroll-Entität enthalten wie das Ereignismodell (Abschnitt 9.3.3) und datengesteuerte Modelle für Iterationen (Abschnitt 9.4).

9.2 Anweisungssequenzen

In der Redeweise des imperativen Programmierparadigmas ist eine Sequenz eine Folge von Anweisungen, die in einer festgelegten Reihenfolge hintereinander ausgeführt werden. In textuellen Programmiersprachen sind die Anweisungen einer Sequenz hintereinander und untereinander geschrieben. Bei der Programmausführung wird aus der räumlichen Anordnung ein zeitliches Nacheinander.

Sequenzen werden durch Verwendung bestimmter Sprachkonstrukte (bei Pascal `begin` und `end`, bei Java und C durch geschweifte Klammern und bei Python durch Einrückung) zu Blöcken zusammengefasst. Häufig – insbesondere bei „gutem Programmierstil" – enthält ein Block eine Sequenz von logisch zusammengehörigen Anweisungen, die ein abstrakt beschreibbares Stück Programmfunktionalität enthalten. In Flussdiagrammen und Struktogrammen (Nassi-Shneidermann-Diagrammen) wird ein Block durch einen Kasten dargestellt.

---

[8] Pennington spricht von Kontrollfluss (control flow). Ein Begriff, der an dieser Stelle vermieden wird, weil er ein bestimmtes intuitives Modell impliziert.

In Visualisierungsübungen mit Schülern kann man beobachten, dass von der durch den Programmtext gegebenen Sequenzialität in einem Block abgewichen wird. Ein Block von Einzelanweisungen kann als sinnvolle Ganzheit interpretiert werden, bei der die Reihenfolge der Einzelaktionen im Detail unwichtig ist. Abb. 42 zeigt vier Screenshots aus einer Flash-Animation, die eine 17-jährige Schülerin zur Visualisierung der folgenden Anweisungsfolge (Java) angefertigt hat:

```
int a;
int b;
a = 2;
b = a + 3;
```

Der Ablauf ist folgender: Zwei Behälter, etikettiert mit `a` und `b` fliegen von rechts und links in die Mitte der Bildfläche (erstes Bild). Anschließend bewegen sich Karten mit der Aufschrift `2` bzw. `a + 3` in die beiden Behälter (zweites Bild), die Karte `2` fliegt aus Behälter `a` in Behälter `b` und verdeckt den Buchstaben `a` auf der Karte `a + 3` (drittes Bild), die daraufhin verschwindet und durch eine Karte mit Aufschrift `5` ersetzt wird (viertes Bild).

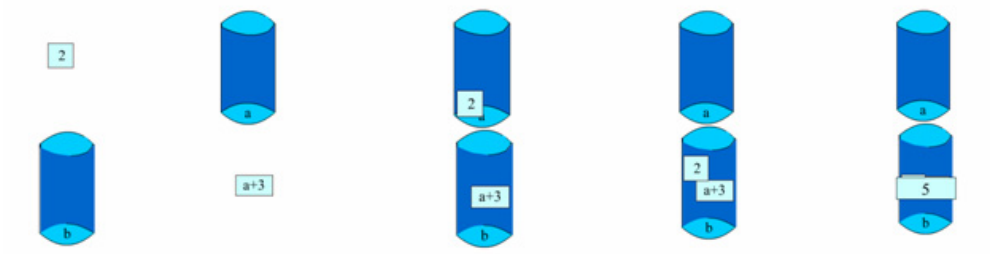


Abb. 42: Visualisierung der Ausführung einer Sequenz durch konkurrente Prozesse

In einem Interview wurde sichergestellt, dass die Schülerin die Reihenfolge der Ereignisse sich genauso vorstellte, wie sie in der Animation dargestellt hatte[9]. Offenbar gab es einige Abweichungen von der im Programmtext festgelegten Sequenzialität:

Die beiden Zuweisungen beginnen gleichzeitig (und unabhängig voneinander), obwohl sie laut Programmtext nacheinander geschehen müssten. Die Parallelisierung verkürzt die Ablaufzeit, macht das Modell kompakter und verbessert den Gestaltcharakter.

Zur Darstellung der zweiten Zuweisung wird zuerst der Term in den Behälter b bewegt und dann ausgewertet. In einer exakten Schritt-für-Schritt-Interpretation des Programmtextes müsste aber zuerst der Term ausgewertet werden und dann das Ergebnis in den Behälter gegeben.

9.3 Bedingte Anweisungen

In diesem Abschnitt diskutieren wir drei Arten intuitiver Modelle zur Repräsentation von bedingten Anweisungen (if-Anweisungen): Verzweigung des Kontrollflusses oder Datenflusses und Ereignismodelle.

9.3.1 Verzweigung des Kontrollflusses

Häufig verwendet man die Begriffe „Verzweigung“ oder „Programmverzweigung“ als Synonyme für bedingte Anweisungen. Gemeint ist dann meist die Verzweigung des Kontrollflusses. Es gibt einen Akteur (Steuereinheit), der eine Bedingung auswertet und in Abhängigkeit von dem Ergebnis die Kontrolle an unterschiedliche Akteure (die üblicherweise jeweils einen Block des Programmtextes repräsentieren) weitergibt. Verzweigungen des Kontrollflusses werden in Flussdiagrammen und Nassi-Shneidermann-Diagrammen verwendet.

[9] Sie wurde gefragt, ob in ihrer Vorstellung tatsächlich die beiden Behälter gleichzeitig Erscheinung treten. Außerdem wurde ihr Hilfe angeboten, falls sie am zeitlichen Ablauf noch etwas ändern wollte.

9.3.2 Kontrolle von Datenflüssen – Datenweichen und Datensperren

Im Datenflussmodell wandern Daten von einer Verarbeitungseinheit zur nächsten. Zur Modellierung von if-Anweisungen wird ein Mechanismus benötigt, der eine Daten-Entität (in Abhängigkeit von der Gültigkeit einer Bedingung) einer Verarbeitungseinheit zuführt („Weiche“). Dieser Mechanismus ist keine Funktion, die definitionsgemäß ein eindeutiges Ergebnis liefern muss. Die visuelle Programmiersprache DRLP (Dataflow Representation Language for Programming, Anjaneyulu & Anderson 1992) verwendet Sperren für Datenflüsse (ENABLE), die geöffnet oder geschlossen werden können (vergleichbar mit steuerbaren Ventilen).

9.3.3 Ereignismodell – Steuersignale

Die Animationen der PVS zur Visualisierung von Programmausführungen verwenden meist ein Ereignismodell um Entscheidungen darzustellen. Ein solches Modell besteht aus zwei Komponenten: Anstelle einer Steuereinheit, die Kontroll- oder Datenflüsse regelt, gibt es eine boolesche Funktion, die eine Bedingung prüft und einen Wahrheitswert ausgibt (Testfunktion). Diese Ausgabe wird als Ereignis interpretiert, das bestimmte Aktionen auslöst. Das ist ein Unterschied etwa zur Vorstellung, dass eine Funktion einen Wert zurückgibt. Die Ausgabe der Testfunktion ist ein Signal, d.h. die Daten (boolesche Werte) werden nicht einem bestimmten Akteur zugeführt sondern sind allgemein sichtbar. Der Begriff Signal betont den Kommunikationsaspekt eines Ereignisses, d.h. durch Signale werden Ereignisse mitgeteilt. Man beachte, dass das eigentliche Ereignis das Zutreffen einer logischen Aussage ist (z.B. `a < b`), die aus der Funktionalität der Testfunktion und ihrem Ergebnis zusammengesetzt ist. Zum Beispiel ergibt sich das Ereignis `a < b`, wenn eine Testfunktion, die prüft ob `a >= b` zutrifft, den Wert `False` signalisiert. Die zweite Komponente des Ereignismodells sind dementsprechend Akteure, die ihre Umgebung beobachten, Ereignisse (dargestellt durch Signale) wahrnehmen und dann darauf reagieren.

Die Idee der Signale ist sehr alt. In China wurden etwa 800 v. Chr. (in der Westlichen Zhou-Zeit) Jahren so genannte Fengsui-Türme gebaut (feng: Feuer, sui: Rauch), von denen aus durch Feuer- und Rauchsignale im Falle drohender Gefahr die Armee alarmiert wurde. Im modernen Alltag kennt man in verschiedenen Zusammenhängen Steuerung durch Signale wie etwa Ampeln an Kreuzungen, Schiedsrichterpfiffe auf dem Fußballplatz oder die Rhythmisierung eines Schultages mit dem Schulgong.

Zu beachten ist, dass moderne Programmiersprachen Konstrukte enthalten, mit denen die Verarbeitung von Ereignissen explizit implementiert werden kann. Bei der Programmierung von grafischen Benutzungsoberflächen etwa bindet man asynchron auftretende Events (z.B. Mausklicks) an Prozeduren (Eventhandler), die darauf reagieren (vgl. z.B. Weigend, 2006 S. 542ff). Eine andere Programmiertechnik, die Ereignisse verarbeitet, ist das Abfangen von Ausnahmen (Exceptions) während der Laufzeit eines Programms etwa in `try...except`- Anweisungen (Python) oder Zusicherungen (`assert`-Anweisungen bei Python). An dieser Stelle geht es jedoch darum, dass eine Entscheidung, die programmiersprachlich als if-Anweisung dargestellt wird, in der Welt gedanklicher Vorstellungen als Ereignisverarbeitung interpretiert wird.

9.4 Iterationen – datengesteuerte Wiederholungen

Eine Iteration ist das Durchlaufen (iter: lat. „Marsch“) einer aufzählbaren (eventuell unendlichen) Kollektion von Items. Programmiersprachen enthalten iterierbare Standard-Objekte, die oft auch als Container bezeichnet werden. Im einfachsten Fall sind das Listen, Zeichenketten oder andere Sequenzen. Es gibt aber auch Container, deren Items keine bestimmte Reihenfolge haben, bei Python z.B. Dictionaries oder Mengen (Weigend 2006). Mit der Iteration verbunden ist das Konzept des Iterators, eines der von Gamma et al. (1995) vorgeschlagenen Entwurfsmuster (design pattern). Darunter versteht man einen Akteur, der zu einer Kollektion nach und nach alle Elemente liefert (siehe auch van Rossum & Yee 2001).

Iterationen sind ein wichtiges Konzept der Alltagsalgorithmik. Folgende Aufgaben können vermutlich von den meisten Menschen intuitiv durch eine Iteration gelöst werden:

- Iss alle Pralinen aus der Pralinenschachtel.
- Begrüße alle Partygäste.
- Klebe auf jede Urlaubskarte eine Briefmarke.

Bei einer Iteration geht die Steuerung des Programmlaufs letztlich von einer Ansammlung von Daten aus (Kollektion). Für jedes Element der Kollektion wird etwas getan. Wenn die Kollektion, über die die Iteration läuft, bekannt ist, kann man auch die Anzahl der Wiederholungen vorhersehen. Selbst wenn die Iteration (etwa bei der Suche nach einem bestimmten Objekt der Kollektion) vorzeitig abgebrochen werden kann, ist doch zumindest die *maximale* Anzahl der Durchläufe im Vorhinein bekannt.

Programmtechnisch werden Iterationen durch for-Anweisungen realisiert. Im Falle von Python hat eine solche Anweisung das Format

```
for i in kollektion:
   anweisungsblock
```

Bei anderen Programmiersprachen wie Java oder C kann für eine Iteration nicht einfach ein beliebiges Container-Objekt angegeben werden. Stattdessen wird eine Sequenz (andere Arten von Kollektionen kann man nicht verwenden) indirekt durch eine Art Generatorausdruck spezifiziert. Das Format einer `for`-Anweisung mit Generatorausdruck (in runden Klammern) ist bei Java:

```
for (i = Startwert; Bedingung, die für i erfüllt sein muss;
     Berechnungsvorschrift für den Nachfolger von i)
    {Anweisungsfolge}
```

Bei der Ausführung einer for-Anweisung wird ein impliziter Iterator verwendet, der das jeweils aktuelle Element der Kollektion liefert. Im Python Visual *Analogies for Iterations* wurde die Arbeitsweise des Iterators in folgendem Python-Programm durch verschiedene intuitive Modelle beschrieben.

```
s = [1, 5, 4, 3, 2]
for i in s:
    print i*i
```

(1) Im ersten Modell (Abb. 43 links) werden aus einem Behälter mit Fächern nach und nach Items (Karten mit Zahlen) entnommen und verarbeitet (Entnahme-Modell). Das Modell lässt sich für alle Arten von Kollektionen verwenden. Im Falle einer Sequenz (wie hier), in der die Items eine bestimmte Reihenfolge haben, ist das nächste Element dasjenige, das im Behälter am weitesten links steht. Durch die Entnahme wird garantiert, dass jedes Item der Kollektion nur ein Mal verarbeitet wird. Wenn der Behälter leer ist, ist die Iteration beendet. Das Entnahme-Modell ist das einfachste und prägnanteste der hier betrachteten Iterationsmodelle. Es wird im Alltag in vielen Situationen verwendet, etwa beim Ausräumen einer Spülmaschine oder wenn man alle Pralinen aus einer Schachtel nimmt und aufisst. Allerdings impliziert das Entnahmemodell, dass die Liste während der Iteration verändert wird, was nicht der realen Arbeitsweise des Programms entspricht. Gleichwohl ist das Entnahmemodell nicht als Fehlvorstellung zu werten, wenn es im Rahmen eines Modellclusters verwendet wird und man sich gleichzeitig bewusst macht, dass beim modellierten Programmlauf tatsächlich die Liste erhalten bleibt.

(2) Im zweiten Modell (Abb. 37 zweites Bild) wird der Iterator überhaupt nicht expliziert. Der Betrachter muss selbst in Gedanken Buch führen, welches Item als nächstes verarbeitet wird. Die Liste wird während der Iteration nicht verändert. Es entstehen nach und nach Kopien der Karten im Behälter, die dann verarbeitet werden.

(3) Im dritten Modell (Markierungsmodell) zeigt ein roter Punkt das aktuelle Item an. Er wandert durch die Liste und repräsentiert den Namen i im Programmtext. Wiederum wird die Liste als Behälter mit Karten dargestellt (Abb. 37 drittes Bild).

(4) Das vierte Modell schließlich verwendet eine andere Metapher für die Liste. Sie wird als Zahlenkolumne auf einem Blatt Papier dargestellt. Von den Zahlen werden nach und nach Kopien auf Karten angefertigt und verarbeitet. Jede bereits verarbeitete Zahl wird durch einen Haken markiert. Dieses Modell der Markierung bereits verwendeter Elemente („Abhaken“) wird häufig bei Iterationen im All-

tag benutzt – zum Beispiel beim Einkauf mit einer Einkaufsliste. Im Unterschied zur wandernden Markierung im dritten Modell hat das Abhaken keine Entsprechung im Programmtext.

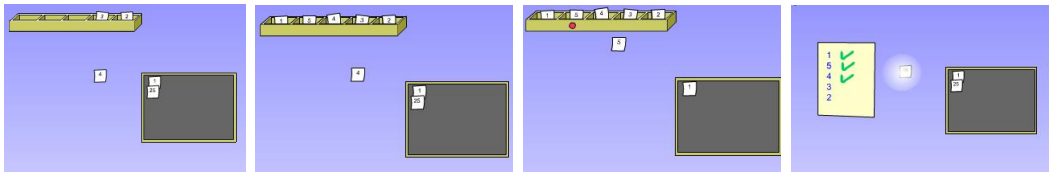


Abb. 43: Visualisierung einer Iteration. Screenshots aus dem Python Visual „Iteration“

Am Ende der Sitzung wurden folgende Fragen gestellt:

- Welche Animation würden Sie verwenden, um jemandem zu erklären wie das Python-Skript funktioniert?
- An welche Animation können Sie sich am besten erinnern?
- An welche Animation denken Sie, wenn Sie sich die Ausführung des Skriptes vorstellen?

Tab. 14 zeigt die Verteilung der Antworten von 66 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben. Die Angaben zu den Fragen unterscheiden sich sämtlich signifikant von einer Gleichverteilung (χ^2-Test, $p = 0.00033$, $p = 0.00013$, $p = 0.00012$). Nach diesen Ergebnissen wurde das Entnahmemodell (in Konkurrenz zu den anderen Modellen) überwiegend abgelehnt, obwohl es das einfachste ist. Der Grund liegt vermutlich darin, dass es im Gegensatz zur Semantik des Programms die Iterationssequenz (Liste) verändert. Bevorzugt werden die beiden letzten Modelle (wandernde Markierung und Abhaken) mit unterschiedlichem Schwerpunkt hinsichtlich der Verwendung (Erklären, Memorieren, in Gedanken nachspielen).

n = 66	**Entnahme von Karten aus einem Behälter**	**Kein Iterator (Entnahme von Kopien)**	**Wandernde Markierung**	**Abhaken bereits verwendeter Elemente**
Erklären	6 (9%)	13 (20%)	30 (45%)	17 (26%)
Erinnern	8 (12%)	8 (12%)	20 (30%)	30 (45%)
In Gedanken vorstellen	9 (14%)	7 (11%)	30 (45%)	20 (30%)

Tab. 14: Wahl verschiedener Modelle zur Veranschaulichung einer Iteration. Ergebnisse von 66 ersten Sessions in Workshops mit der PVS. Teilnehmer: 66 Schülerinnen und Schüler, davon 52 aus Deutschland, 12 weiblich und 54 männlich, sie beschäftigten sich im Schnitt 5.05 Stunden pro Woche mit Programmierung und hatten ein Durchschnittsalter von 16.64 Jahren.

9.5 Wiederholungen mit nicht antizipierbarem Ende

Neben Iterationen gibt es noch einen zweiten Typus von Wiederholungen, der in Programmiersprachen wie C, Pascal, Java oder Python durch eine `while`-Anweisung implementiert wird. Sie hat im Falle von Python folgendes Format:

```
while bedingung:
   Anweisungsblock
```

Hier geht die Steuerung vom Test einer Bedingung aus. Solange sie erfüllt (wahr) ist, wird der Anweisungsblock ausgeführt. Dabei ist – im Unterschied zur Iteration – ohne tiefgehende Analyse der Semantik des Programms nicht vorhersehbar, ob und wann die Bedingung nicht mehr erfüllt ist und die Wiederholung abgebrochen wird.

Eine programmtechnische Variante sind `repeat…until`- oder `do…while`-Anweisungen, die von einigen Programmiersprachen angeboten werden. Der Unterschied ist marginal und liegt allein darin, dass der Test der Bedingung *nach* Ausführung des Anweisungsblocks im Innern erfolgt. Wir beschränken unsere Betrachtungen hier auf den zuerst genannten Fall von `while`-Anweisungen.

Die steuernde Bedingung der while-Anweisung ist formal eine Aussage, die wahr oder falsch sein kann. Für die intuitive Modellierung ist jedoch die Bedeutung wichtig. Generell ist die Bedingung eine Aussage über den momentanen Systemzustand. Man kann folgende Interpretationen unterscheiden:

- Die Bedingung ist die negative Formulierung eines Ziels. Solange die Bedingung erfüllt ist, ist das Ziel noch nicht erreicht. Der Anweisungsblock hat die Aufgabe, das System dem Ziel ein Stück näher zu bringen. Diese Interpretation wird z.B. bei der algorithmischen Darstellung von Näherungsverfahren (z.B. Wurzelberechnung nach Heron) verwendet. Das Ziel ist hier eine gewisse Genauigkeit des berechneten Näherungswertes.
- Die Bedingung beschreibt einen Systemzustand, in dem es keinen Sinn mehr macht, den Anweisungsblock auszuführen. Der Anweisungsblock kann im Prinzip beliebig oft ausgeführt werden. Es gibt keine Annäherung an ein Ziel. Ein Beispiel ist der fetch-execute-Zyklus einer Benutzungsschnittstelle. Der Anweisungsblock wird solange wiederholt, bis der Benutzer (aus welchen Gründen auch immer) das Programm beendet. Ein Spezialfall dieser Interpretation sind Endloswiederholungen, bei denen die steuernde Bedingung immer wahr ist (`while True: …`)

Die beiden Interpretationen der while-Bedingung korrespondieren in etwa mit zwei unterschiedlichen intuitiven Modellen für while-Anweisungen, die im Folgenden diskutiert werden: Die kontrollierte Wiederholung und die Schleife.

9.5.1 Kontrollierte Wiederholung einer holistischen Aktivität

Nassi-Shneiderman-Diagramme (Struktogramme, DIN 66261) unterstützen die Vorstellung eines Akteurs, der die Ausführung eines untergeordneten Programmblocks (den man sich als eigenen Akteur vorstellen kann) steuert. Dieser Steuerungsakteur prüft die Einhaltung der while-Bedingung und regelt letztlich, wie oft der angeschlossene Anweisungsblock ausgeführt wird.

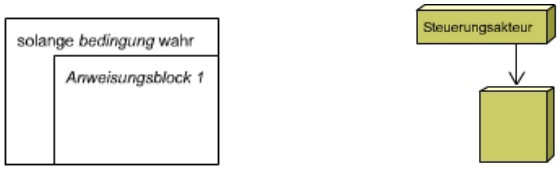


Abb. 44: Nassi-Shneiderman-Diagramm (Sinnbild nach DIN 66261) und Modell eines Steuerungsakteurs

Diese Intuition wird z.B. von LabView (National Instruments, www.ni.com/labview/) unterstützt. Hier ist es möglich, eine Anordnung in einen Rahmen zu packen und mit einem Wiederholungsicon zu versehen.

Eine Besonderheit gegenüber dem Schleifenmodell (siehe nächster Abschnitt) ist, dass der wiederholte Anweisungsblock als eine einzige holistische Einheit vorstellbar ist. Das heißt man kann von Einzelheiten insbesondere auch von der zeitlichen Reihenfolge, in der atomare Einzelanweisungen des Blocks ausgeführt werden müssen, abstrahieren.

Die Verwendung dieses Modells kommt in folgenden Formulierungen aus der „Algorithmik des Alltags“ zum Ausdruck:

- Solange noch Erbsen in der Asche liegen, nimm eine Erbse heraus.
- Solange das Auto noch nicht steht, bremse.
- Solange du die Aufgaben noch nicht schnell genug lösen kannst, übe.

Der zweite Unterschied zum Schleifenmodell liegt in der pointierten Modellierung der Steuerungskomponente als eigenem Akteur, der die Ausführung des (gesamten als geschlossene Gestalt gedachten) Anweisungsblocks überwacht und steuert.

Mit dieser Intuition der Wiederholungskontrolle ist eine mögliche Fehlvorstellung verbunden, nämlich dass die wiederholte Aktion sofort abgebrochen wird, wenn die while-Bedingung nicht mehr erfüllt ist.

9.5.2 Schleifen

Im Programmierjargon ist der Begriff Schleife (loop) für Wiederholungen außerordentlich gut etabliert. Dahinter steht die Vorstellung eines kreisförmigen Kontrollflusses. Die zeitliche Abfolge von Aktivitäten wird durch einen zyklischen gerichteten Graphen visualisiert. Typischerweise denkt man bei einer Schleife nicht an eine einzige holistische Aktion, die wiederholt wird, sondern an eine Sequenz von mindestens zwei Aktionen, die immer wieder in der gleichen Reihenfolge hintereinander ausgeführt werden. Diese Sequenz von Aktionen bildet das Herzstück des Modells. Beispiele von Aktionszyklen im Alltag sind:

- Der Wechsel von Tag und Nacht im Verlauf eines Tages.
- Der Zyklus der Jahreszeiten.
- Das Wechselspiel von Frage und Antwort in einem Interview.

Schleifen werden z.B. in Programmablaufplänen (PAP, Flussdiagramm) visualisiert. PAPs enthalten kein besonderes Sinnbild für Wiederholungen. Der Austritt aus der Schleife wird durch eine Verzweigung des Kontrollflusses (wie bei einer if-Anweisung) dargestellt. Die schleifenförmige Struktur wird erst durch Betrachtung der Gesamtstruktur sichtbar. In Anhang 4.2 wird als Beispiel der Fetch-execute-Zyklus eines interaktiven Systems diskutiert.

Mit dem Schleifenkonzept ist die Vorstellung von Sprüngen der Kontrolle verbunden. Beim Programmtext in einer üblichen Programmiersprache (C, Java, Pascal Python) werden Anweisungen von oben nach unten ausgeführt. Nach der letzten Anweisung des wiederholten Blocks, springt die Kontrolle wieder zur ersten Anweisung der wiederholten Sequenz.

Schleifen werden in der graphischen Programmiersprache für Robotersteuerungen von Lego-RoboLab (www.lego.com) verwendet. Da hier der Kontrollfluss standardmäßig immer von links nach rechts gerichtet ist, werden zur Darstellung der Schleife Sprungmarken verwendet.

Mit Macromedia Flash werden komplexe Animationen als Aggregate von zyklisch ablaufenden Filmen (Movie Clips) konstruiert. Jeder Film ist eine Folge von Bildern (Frames), die von links nach rechts durchlaufen werden. Die Wiederholung wird durch einen Sprung vom letzten an das erste Bild realisiert.

9.6 Rekursion

Eine rekursive Funktion ist eine Funktion, in deren Definition mindestens ein Aufruf derselben Funktion vorkommt. Die Ausführung einer Funktion kann als Prozess beschrieben werden. Beim Aufruf wird ein Objekt (Execution Frame) angelegt, in dem der aktuelle Zustand der Funktionsausführung gespeichert ist. Bei jedem rekursiven Aufruf erzeugt das Laufzeitsystem einen neuen Prozess mit neuem Execution Frame. Der aufrufende Prozess wartet, bis der rekursive Funktionsaufruf abgearbeitet ist und fährt dann fort.

Gegenseitige Rekursion entsteht, wenn zwei Funktionen sich abwechselnd gegenseitig aufrufen.

Rekursion ist ein Konzept zur Steuerung von Programmläufen. Rekursion ist aber auch ein allgemeines Prinzip, das zur Beschreibung von Dingen in der Welt herangezogen werden kann. Manche Phänomene des Alltags legen eine rekursive Beschreibung nahe:

- Vorfahren: Meine Vorfahren sind meine Eltern und deren Vorfahren.
- Matroschka (Russian Doll): Eine Matroschka ist entweder massiv oder sie besteht aus einer hohlen Puppe, die eine Matroschka enthält (Fothe 2005).

Solche „rekursiven Phänomene“ (insbesondere die Russian Dolls) werden gelegentlich als Modelle für Rekursion bezeichnet (Wu et al. 1998). Fothe (2005) verwendet sie zum Testen von Wissen über

Rekursion. Levy und Lapidot (2001) beschreiben kognitive Konzepte, die mit Rekursion assoziiert werden, wie z.B. Regelmäßigkeit, Gradualität, Periodizität und Zirkularität (siehe Anhang 4.3).

9.6.1 Rekursion als Schleife

Eine Reihe von Autoren bezeichnen das „Schleifenmodell“ („loop model“) als typische Fehlvorstellung von Anfängern über die Arbeitsweise rekursiver Funktionen (Kahney 1984, Kurland & Pea 1985, Bhuiyan et al. 1994, Close & Dicheva 1997).

Im Schleifenmodell wird die Ausführung einer rekursiven Funktion als Wiederholung gesehen. Der rekursive Aufruf wird als Aufforderung gesehen, die (bisherige) Sequenz von Anweisungen noch einmal von vorne zu beginnen – allerdings unter Verwendung eines anderen Argumentes). Der rekursive Aufruf wird als Sprungbefehl in Kombination mit der Veränderung von Variablen (Argumente) interpretiert. Insofern liegt tatsächlich eine Schleife vor.

Das Schleifenmodell führt bei endrekursiven Prozeduren nicht zu Widersprüchen. Bei diesem Spezialfall rekursiver Funktionen wird kein (nichtleeres) Objekt zurückgegeben und der rekursive Aufruf erfolgt ganz am Ende der Anweisungsfolge. Man beachte, dass jede Funktion, die ein Ergebnis zurückgibt (also keine Prozedur ist) nicht endrekursiv sein kann, weil die Rückgabe des Ergebnisses die letzte ausgeführte Anweisung ist und der rekursive Aufruf vorher erfolgt.

Bhuiyan et al. (1994) sind der Auffassung, dass es zum Schleifenmodell kommt, wenn Programmieranfänger vorher erworbenes Wissen über Iterationen auf die Interpretation und Generierung von rekursiven Funktionen anwenden. Auf der anderen Seite konnten Anzai & Uesato (1982) für den Bereich der Mathematik nachweisen, dass Schüler rekursive Funktionen wie die Fakultätfunktion leichter formulieren konnten, wenn sie zuvor mit iterativen Funktionsdefinitionen experimentiert hatten. Kessler und Anderson (1989) beobachteten, dass Übungen mit iterativen Programmen einen positiven Transfereffekt auf anschießendes Erstellen rekursiver Programme hatten. Dagegen führte ein Training mit rekursiven Programmen bei nachfolgender iterativer Programmierung zu einer Leistungsverschlechterung (negativer Transfereffekt). Die Autoren schlossen daraus, dass bei Übungen mit iterativen Programmen relativ gut grundlegende mentale Modelle zur Steuerung von Programmabläufen gelernt werden, die später als Basis zum Verständnis rekursiver Funktionen dienen können.

9.6.2 Rekursion als Selbstaufforderung

Was in der Literatur als „Schleifenmodell“ bezeichnet wird, kann man – zumindest im Zusammenhang mit endrekursiven Prozeduren – auch als eigenständige Intuition betrachten, die sich von der Schleife unterscheidet. Wir nennen sie das Modell der rekursiven Selbstaufforderung. Es lässt sich durch folgende Merkmale charakterisieren:

- Es gibt nur einen Akteur, der für die Ausführung verantwortlich ist. Darin unterscheidet sich das Modell von Delegationsmodellen (siehe nächster Abschnitt), bei denen bei rekursiven Aufrufen neue Akteure ins Spiel kommen.
- Es gibt ein holistisches Konzept eines Handlungsablaufs, der komplett wiederholt wird (falls es notwendig ist).
- Die gesamte Aktivität ist eigentlich nicht von vornherein auf Wiederholung angelegt (siehe Alltagsbeispiele unten). Es kann durchaus sein, dass eine einmalige Ausführung ausreicht. Dagegen geht man bei der Planung einer typischen Wiederholung mit while immer von vielen Durchgängen aus. Typische Schleifen wie der Fetch-Execute-Zyklus eines interaktiven Systems sind von der Idee her sogar Prozesse mit beliebig vielen Wiederholungen. Das heißt innerhalb des Systemdesigns gibt es keinen Grund, die Aktivität abzubrechen.

Im Alltag gibt es viele Beispiele für rekursive Selbstaufforderungen:

- Lösen einer Mathematikaufgabe: Löse die Gleichung und mache die Probe. Wenn sich die Lösung als falsch erweist, löse die die Aufgabe noch einmal.

- Brennen einer CD: Nachdem die CD gebrannt ist, fragt das System, ob man noch eine weitere CD mit dem gleichen Inhalt beschreiben möchte. Wenn man eine neue CD einlegt und den entsprechenden Button anklickt, wird der Vorgang wiederholt.
- Sich zum Bahnhof durchfragen: Frage jemanden nach dem Weg zum Bahnhof und folge den Anweisungen. Wenn du dann noch nicht am Bahnhof bist (z.B. weil die Beschreibung unvollständig oder teilweise falsch war), frage dich weiter durch.

Zu beachten ist, dass das Modell der rekursiven Selbstaufforderung nur bei endrekursiven Algorithmen Sinn macht, weil es nur hier eine Gestalt der zu wiederholenden Aktivität gibt. Bei eingebetteter Rekursion gibt es zumindest zwei Blöcke, einer vor und einer nach dem rekursiven Aufruf).

Ein Beispiel für eine endrekursive Prozedur ist das Logo-Programm aus Abb. 45. Es wurde mit Microworld EX (www.microworlds.com), einer modernen Variante der Logo-Turtle-Mikrowelt, erstellt.

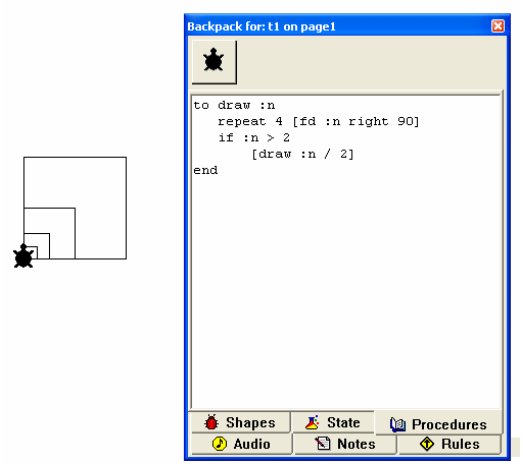


Abb. 45: Screenshot aus einer Sitzung mit Microworld EX. Links das Arbeitsblatt mit der Turtle `t1`, rechts der Backpack der Turtle mit einer rekursiven Prozedur.

Die Logo-Turtle-Grafik unterstützt das Modell der Selbstaufforderung, weil es nur einen Akteur gibt (die Turtle), dem die in der Prozedur definierte Aktivität zugeordnet wird. Microworld EX verwendet die Metapher eines Rucksacks, den die Schildkröte trägt. Der Rucksack enthält z.B. aktuelle Attributbelegungen (z.B. Zeichenstift angehoben oder gesenkt), die den Zustand definieren, und Aktivitäten (Procedures), die die Schildkröte über die fest eingebauten Operationen (wie vorwärts gehen, drehen etc.) hinaus beherrscht.

In diesem Fall der Prozedur `draw` zeichnet die Turtle ein Quadrat mit einer als Parameter übergebenen Seitenlänge n. Sofern die Seitenlänge nicht zu klein ist, fordert sich die Turtle am Ende selbst auf, ein weiteres Quadrat mit der halben Seitenlänge zu zeichnen usw. Alles macht dieselbe Turtle ganz allein. Es kommen keine weiteren Akteure ins Spiel.

9.6.3 Fehlerhafte Verwendung des Modells der Selbstaufforderung

Die Vorstellung eines einzigen Akteurs, der sich immer wieder selbst auffordert, dieselbe Aktivität (mit geänderten Parametern) durchzuführen, ist so einleuchtend, dass sie in der wohl gängigsten Kurzdefinition rekursiver Funktionen verwendet wird: „Eine rekursive Funktion ist eine Funktion, die sich selbst aufruft“. Doch ist dieses Modell nur für den Spezialfall endrekursiver Prozeduren geeignet. Wenn der rekursive Aufruf nicht die letzte Aktivität ist, kann er nicht als Wiederholung eines ansonsten abgeschlossenen Vorgangs betrachtet werden. Man spricht dann von eingebetteter Rekursion. Untersuchungen von Dicheva und Close (1996) belegen, dass Schüler bei der Interpretation von Logo-Programmen das Modell der Selbstaufforderung auch für eingebettete Rekursion verwenden (Details in Anhang 4.4).

9.6.4 Delegationsmodell

Im Delegationsmodell wird deutlich zwischen der Definition und Ausführung einer Funktion differenziert. Eine Funktion wird nur einmal definiert, kann aber durch mehrere Akteure (die Prozesse repräsentieren und sich zu einem Zeitpunkt in unterschiedlichen Zuständen befinden) ausgeführt werden. Beim Aufruf einer Funktion (durch einen Akteur) wird eine Aufgabe an einen anderen Akteur delegiert, der entweder schon existiert oder in diesem Moment erzeugt wird. Das Delegationsmodell ist nicht nur auf rekursive Aufrufe beschränkt, sondern kann für jede Art von Funktionsaufruf verwendet werden. Bei einem rekursiven Aufruf ist allein die Besonderheit, dass der neue Akteur gemäß derselben (einmaligen) Funktionsdefinition arbeitet wie der aufrufende Akteur. Das Konzept der Wiederholung spielt hier keine Rolle. Der aufrufende Akteur (Prozess) wartet bis die delegierte Aufgabe erledigt ist und arbeitet dann (eventuell unter Verwendung eines zurückerhaltenen Ergebnisses) weiter. Das Delegationsmodell (ohne Schachtelung mit statischen Akteuren) kann in Rollenspielen veranschaulicht werden, indem mehrere Personen gemeinsam einen rekursiven Algorithmus ausführen und jeweils Teilaufgaben an andere Personen delegieren. Visualisierungen, die auf dem Delegationsmodell basieren werden in Anhang 4.5 diskutiert.

Wie das Beispiel aus Abschnitt 9.6.2 zeigt, unterstützt die Logo-Mikrowelt nicht das Delegationsmodell, weil die gesamte rekursiv beschriebene Aktivität ein und demselben Akteur (Turtle) zugeschrieben wird. Ebenso passt das Delegationsmodell nicht zum Paradigma der objektorientierten Programmierung. Beim rekursiven Aufruf innerhalb einer rekursiven Methode wird kein neues Objekt der Klasse instanziert, sondern dasselbe Objekt erneut beauftragt.

9.6.5 Protokoll-Modelle für rekursive Algorithmen

Visualisierungen der Arbeitsweise rekursiver Funktionen wie die in Abb. 41 (Abschnitt 8.9, Fibonacci-Zahlen) stellen die Mechanik des Zusammenspiels verschiedener Akteure in den Mittelpunkt der Modellierung. Sie fokussieren auf eine Veranschaulichung der Semantik eines rekursiven Funktionsaufrufs. Dabei bleibt die „Gesamtidee“ des Algorithmus im Hintergrund.

Protokollmodelle (siehe Abschnitt 2.5) dagegen abstrahieren weitgehend von der Steuerungsmechanik und geben ein als Gestalt erfassbares und gut merkbares Bild vom Ablauf eines rekursiven Algorithmus. So liefert ein Suchbaum eine holistische Vorstellung der (rekursiven) binären Suche. Jeder Pfad von der Wurzel zu einem Blatt ist ein Protokoll eines Suchvorgangs. Eine Matroschka (Russian Doll) ist ein Abbild ihres rekursiven Herstellungsprozesses.

Viele „rekursive Phänomene“ (Levi & Lapidot 2000, Levi et al. 2001) oder Abbildungen aus Logo Art Galleries (z.B. Tuzova & Katz 2001) können als Protokoll des Ablaufs einer rekursiven Funktion interpretiert werden. D.h. das Ergebnis (das Bild) liefert gleichzeitig eine Vorstellung von seiner (rekursiven) Herstellung.

9.6.6 Schema einer rekursiven Funktion und Dedynamisierung

Anderson, Pirolli und Farell (1988) sehen die Schwierigkeit rekursiver Programmierung darin, dass es Menschen schwer fällt, die Ausführung einer rekursiven Funktion in Gedanken nachzuspielen: „… recursive mental procedures are very difficult – perhaps impossible – for humans to execute” (S.160). Ist damit Rekursion eine Denkweise, für die der Mensch nicht geschaffen ist?

Versuchen wir eine differenzierte Betrachtungsweise. Das Delegationsmodell zur Erklärung eines rekursiven Aufrufs *als solches* ist einfach und intuitiv. Dass Aufgaben an andere delegiert werden können, ist eine Alltagserfahrung und Funktionsaufrufe werden in nicht-rekursiven Zusammmenhängen leicht verstanden. Dagegen ist es eine völlig andere, äußerst anstrengende Aktivität, die konkrete Ausführung einer bestimmten rekursiven Funktion in Gedanken nachzuspielen – zu simulieren. Schließlich muss *ein* Individuum Protokoll über alle beteiligen Prozesse führen und ihre Zwischenzustände memorieren.

Mehr noch: Für das Verstehen oder Kreieren eines rekursiven Algorithmus kann der Gedanke an die konkrete Ausführung geradezu schädlich sein. Denn der Versuch des Nachspielens ist eine unnötige mentale Belastung und lenkt von der eigentlichen Aufgabe ab.

Durch rekursive Funktionen können Probleme nach dem Prinzip des „divide and conquer“ gelöst werden, nach Schwill (1993) eine fundamentale algorithmische Idee der Informatik. Ein divide-and-conquer-Algorithmus ist nach folgendem Muster aufgebaut:

Wenn das Problem einfach genug ist, löse es direkt (Elementarfall).
Sonst:
Spalte das Problem in kleinere Teile,
Löse die Teilprobleme durch einen oder mehrere rekursive Funktionsaufrufe
Generiere aus den Teillösungen, die von rekursiven Aufrufen geliefert werden,
eine Gesamtlösung

Es handelt sich nicht um ein mentales Modell, sondern um ein Schema zur Beschreibung des (sprachlichen) Aufbaus der Definition einer rekursiven Funktion. Es gibt einen festen Rahmen vor und enthält Slots für variable Teile, die an die jeweilige Problemstellung angepasst werden müssen. Genau genommen gibt es mehrere Schemata für das divide-and-conquer-Muster, die sich aber nicht in den verwendeten Konzepten sondern nur in der Reihenfolge ihrer Anwendung unterscheiden. Eine besonders einfache Konkretisierung ist die folgende rekursive Funktion zur Berechnung des Spiegelbildes einer Zeichenkette (Python):

```
def mirror (w):
   if w== "": return w
   else: return mirror(w[1:]) + w[0]
```

Ein wichtiger Punkt ist, dass in dem divide-and-conquer-Schema keine Wiederholungen vorkommen. Sie entstehen erst, wenn ein Interpreter – der aber nicht Gegenstand der Modellierung ist – das Programm ausführt. Die Besonderheit rekursiver Funktionen liegt im rekursiven Aufruf, im Beispiel `mirror(w[1:])`. Anderson et al. (1988) vergleichen ein solches Stück Programmtext mit einem Neckerschen Würfel, einer Abbildung, für die es zwei Interpretationen gibt (Kippbild). Man kann `mirror(w[:1])` einmal als Aufruf einer Funktion (deren Ausführung man in Gedanken nachvollziehen möchte) oder zum anderen einfach als Name für einen Wert (in diesem Fall die gespiegelte Version des Teilstrings `w[1:]`) auffassen.

Mit letzterem Modell wird von der komplexen Mechanik der Ausführung praktisch komplett abstrahiert. Offenbar ist es denkökonomisch, bei einer Analyse einer rekursiven Funktion – wie bei einem Kippbild – gewissermaßen „innerlich umzuschalten“ und die Betrachtungsweise zu ändern: Man beginnt mit dem schrittweisen Nachvollziehen der Anweisungen des Programmtextes. Doch beim rekursiven Aufruf bricht man diese Aktivität ab und nimmt das Ergebnis des rekursiven Aufrufs als gegeben hin. Dieser Wechsel der Sichtweise scheint jedoch Schwierigkeiten zu bereiten. Das ergeben Untersuchungen mit der PVS. In verschiedenen Python Visuals wurden Teilnehmer mit verschiedenen Animationen zur Erklärung der Arbeitsweise rekursiver Funktionen konfrontiert. Die meisten Teilnehmer der PVS bevorzugten die längeren und komplexeren Animationen, die die Rekursion vollständig bis zum Abbruch im Elementarfall nachvollziehen, gegenüber Darstellungen, die nur den ersten rekursiven Aufruf zeigen (Details im Anhang 4.6).

10 Verarbeitung

Ein Computer wird häufig als Gerät beschrieben, das Daten verarbeitet. In diesem Kapitel wird untersucht, wie elementare intuitive Modelle zur Entstehung, Vernichtung, Veränderung und Bewegung verwendet werden, um Verarbeitungsprozesse darzustellen.

10.1 Entstehen

Modelle für Verarbeitungsvorgänge implizieren häufig die Entstehung neuer Entitäten. Entstehung kann als Erschaffung (Kreation) oder als Auswahl gesehen werden.[10]

10.1.1 Entstehen von Daten

Die Anweisung `b = a` wird bei Verwendung eines Behältermodells für Variablen als Kopieren des Inhalts von a beschrieben. Dabei wird eine neue Datenentität (die Kopie) geschaffen. Das Referenzierungsmodell dagegen korrespondiert mit der Vorstellung einer Entstehung durch Auswahl. Die Zuweisung `a = 3` wird hier so interpretiert, dass aus einem konstanten Reservoire von Zahlen, die Zahl `3` ausgewählt und mit `a` benannt wird.

10.1.2 Entstehung von Namen

In einem Programm, das in einer typisierenden Programmiersprache formuliert worden ist, wird die Erzeugung einer Variablen in zwei Teile aufgeteilt: Deklaration (d.h. Zuordnung des Namens zu einem Datentyp) und erste Zuweisung (Initialisierung). Man kann sich die Deklaration als Entstehung eines leeren Behälters mit Aufschrift `a` vorstellen oder als Reservierung eines Namens, der aber zunächst nichts benennt.

Die üblichen höheren Programmiersprachen lassen es zu, dass man sich die Definition eines Namens als Kreation vorstellen kann. Die Menge möglicher Namen wird durch die Grammatik der Sprache begrenzt, ist aber dennoch riesig. Zu gutem Programmierstil gehört es, dass Variablennamen aussagekräftig sind und dem Leser die Rolle der Variablen im Programm verraten (z.B. `summe`, `minimum`). Subjektiv kann somit die Schaffung eines Namens sogar mit dem Entstehen von Algorithmik verbunden sein.

Es konnte beobachtet werden, dass Schüler Variablen Eigenschaften zumessen, die sie nicht besitzen. Sie verwenden damit ein überzogenes („magisches“) Entstehungskonzept und glauben, dass durch Schaffung eines Namens auch Funktionalität kreiert wird. Wenn z.B. eine Variable `smallest` genannt wird, glauben diese Schüler, dass beim Lesen von Zahlen aus einem Datenstrom dieser Variablen automatisch der kleinste vorkommende Wert zugewiesen wird (Putnam et al. 1989).

Bei einem Tabellenkalkulationsprogramm dagegen wählt man eine Zelle (d.h. einen Namen der Form *SpalteZeile* für einen Wert) aus einem relativ kleinen Angebot von Zeichenkombinationen aus. Die Namen dienen allein der Adressierung und sind keine Bedeutungsträger. Ähnliches gilt für Mikroprogramme eines Prozessors mit gegebenem Set von Registern. Allerdings sind hier die Register mit bestimmten Möglichkeiten der Verarbeitung verbunden.

10.2 Vernichtung

10.2.1 Implizite Vernichtung bei Zuweisungen

Programmiersprachen enthalten spezielle Konstrukte zur Vernichtung. Beispielsweise gibt es bei Python das Kommando `del` zum Löschen von Objekten. Python-Listen besitzen die Methode `remove()` zur gezielten Entfernung von Elementen. Es gibt aber auch Programmteile, die mit Vernichtung verbunden sind, ohne dass dies – durch eine besondere Anweisung – expliziert wird. So impliziert die Überschreibung einer Variablen die Vernichtung des vorigen Inhalts.

[10] Zu Entstehungskonzepten im Alltag siehe Anhang 5.1

Aufgabe 2 des Python Quiz *Modeling assignments* enthält Animationen, die folgende Anweisungsfolge visualisieren:

```
today = "Monday"
today = "Tuesday"
today = "Wednesday"
```

In den von den Spielern zu bewertenden Animationen wurden drei verschiedene Modelle zur Darstellung der Variablen `today` verwendet:

- Notizzettelmodell. Ein Blatt Papier trägt als Überschrift (größerer Schrifttyp und blaue Schriftfarbe) den Variablennamen `today`. Darunter werden mit einem Stift die aktuellen Werte geschrieben.
- Behältermodell. In eine Box mit Etikett `today` werden Zettel mit dem aktuellen Wert gegeben.
- Zeigermodell. Von einem Brett mit dem Namen `today` führt ein Zeiger zu dem aktuellen Wert.

Mit diesen Grundmodellen werden in den Animationen Zuweisungen veranschaulicht und dabei verschiedene Vernichtungskonzepte verwendet:

(1) Als ungültig markieren: Auf einem Notizzettel mit Überschrift `today` werden sukzessive neue Werte (`Monday`, `Tuesday`, …) geschrieben, nachdem der alte Wert in der Zeile darüber durchgestrichen worden ist (Abb. 46 links).

(2) Zerstörung (1): Auf einem Notizzettel werden sukzessive neue Werte geschrieben, nachdem jeweils der alte Wert ausradiert worden ist (Abb. 46 zweites Bild).

(3) Zerstörung (2): Die Variable wird durch einen Behälter mit der Aufschrift `today` repräsentiert, in den sukzessive neue Zettel mit den Wochentagen wandern. Kurz bevor ein neuer Zettel in den Behälter gelangt, wird der alte Zettel mit einem Blitz zerstört (Abb. 46 drittes Bild).

(4) Verlust: Bevor in den Behälter mit der Aufschrift `today` ein neuer Zettel gelangt, bewegt sich der vorige Inhalt heraus, so dass sich immer nur ein Zettel in dem Behälter befindet (Abb. 46 viertes Bild).

(5) Lösen von Bezügen: Die Variable wird durch ein Zeigermodell repräsentiert. Die Daten-Entitäten (Zettel mit den Wochentagen) bleiben während der Zuweisungen erhalten. Allein der Zeiger wechselt von einem Zettel zum nächsten. Dabei löst sich jeweils der Bezug des Variablennamens zum vorigen Wert (Abb. 46 rechts).

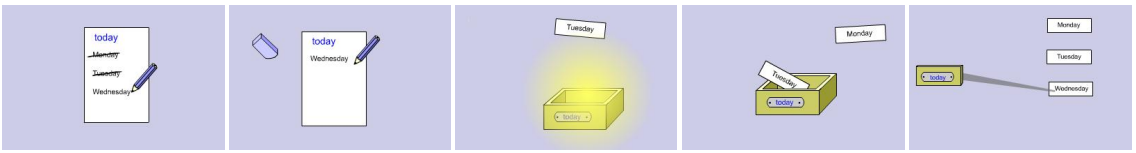


Abb. 46: Darstellung von Zuweisungen mit unterschiedlichen Vernichtungsmodellen.

Tab. 15 zeigt, dass alle fünf Varianten von der Mehrheit der beobachteten Schülerinnen und Schüler spontan als gültige Modelle für sukzessive Zuweisungen gewertet werden. Allerdings wird das erste Modell, bei dem der vorige Wert als ungültig markiert wird, signifikant seltener als korrekt erkannt als die beiden letzten Modelle (exakter Fisher-Test, $p = 0.03$ bzw. $p = 0.004$).

n = 154	Dauer der Animation	Entscheidungs-zeit (Stdabw.)	Als passend beurteilt von	Konfidenz (Stdabw.)
1. Als ungültig markieren (pq_assign_a2_5)	8 s	8.12 (6.71)	102 (66.2 %)	91.7% (23.4%)
2. Ausradieren des vorigen Inhalts (pq_assign_a2_6)	10 s	9.28 s (6.22 s)	111 (72.1%)	94.6% (15.6%)
3. Wegsprengen des vorigen Inhalts (pq_assign_a2_3)	7 s	10.14 s (11.01)	117 (76.0 %)	89.3% (25.3%)
4. Verlust des vorigen Inhalts (pq_assign_a2_2)	7 s	9.45 s (12.45)	120 (77.9%)	91.2% (24.0%)
5. Lösen eines Bezuges (pq_assign_a2_9)	6 s	7.77 s (5.88)	125 (81.2%)	93.6% (17.9%)

Tab. 15: Beurteilung von Modellen zur Veranschaulichung von Zuweisungen. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 154 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

10.2.2 Sukzessive Zuweisungen ohne Vernichtung

Eine „Fehlvorstellung“ bei der Interpretation von sukzessiven Zuweisungen ist die Annahme, dass die vorherigen Werte erhalten bleiben und sozusagen die Geschichte der Variable aufgezeichnet wird[11]. Diese Vorstellung entspricht der Art und Weise, wie Menschen neue Information aufnehmen. Als aktueller Wert wird dann die neuste Information übernommen und die alten in den Hintergrund gedrängt. Man erfährt, wer der der neue Bundeskanzler ist und speichert den Namen unter der Rubrik „Bundeskanzler“ ab, vergisst aber nicht die Namen der vorigen Amtsinhaber. Es ist fraglich, ob man dieses Zuweisungskonzept unbedingt als fehlerhaft einstufen muss. Denn auch wenn vergangene Werte zu sehen sind, ist doch der aktuelle („neuste“) Wert erkennbar. Sajaniemis Visualisierungsvorschlag für Stepper (Streifen mit Werten, von denen einer als „aktuell“ markiert ist) bezieht auch vergangene (und zukünftige) Werte ein (Sajaniemi 2002).

Aufgabe 2 des Python Quiz *Modeling assignments* enthält Filme mit den drei oben genannten Grundmodellen für Variablen, die sukzessive Zuweisungen ohne Vernichtung des vorigen Wertes visualisieren. Zuweisungen führen zu Akkumulation von Datenentitäten. Beim Notizzettelmodell (Abb. 47 links) werden Werte untereinander geschrieben. Der untere Schriftzug ist der aktuelle Wert. Im Behältermodell wird das neue Datum vor das zuletzt aktuelle gesteckt (Abb. 47 Mitte). Beim Zeigermodell entsteht bei jeder Zuweisung ein neuer Zeiger unter dem zuletzt hinzugefügten Zeiger (Abb. 47 rechts).

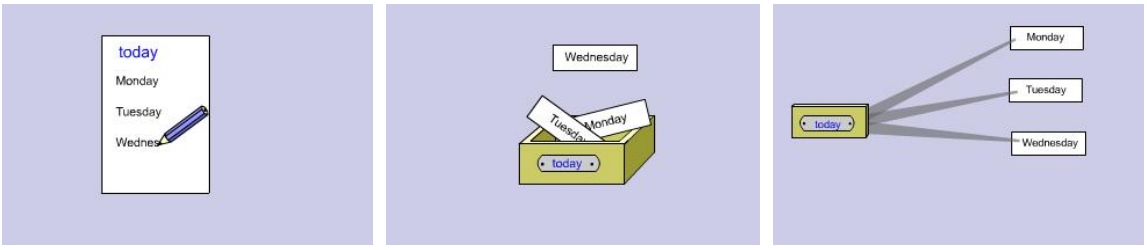


Abb. 47: Sukzessive Zuweisungen ohne Vernichtung

Die Bewertungsergebnisse aus Tab. 16 zeigen, dass nur eine Minderheit der beobachteten Schülerinnen und Schüler vernichtungsfreie Zuweisungsmodelle für geeignet halten. Dabei gibt es einen signifikanten Unterschied zwischen der Einschätzung des Notizzettelmodells und des Zeigermodells ($p = 0.028$), den man darauf zurückführen könnte, dass die PVS-Spieler mit dem Notizzettelmodell vertrauter sind (vgl. Umfrage zu Visualisierungen im Informatikunterricht Abschnitt 2.6).

[11] In mehreren Studien konnte beobachtet werden, dass Schüler glauben, eine Variable könne gleichzeitig mehrere Werte enthalten (vgl. Putnam et al. 1989, Ben-Ari 2001).

n = 154	Dauer der Animation	Entscheidungs-zeit (Stdabw.)	Als passend beurteilt von	Konfidenz (Stdabw.)
1. Notizzettel (pq_assign_a2_4)	8 s	10.67 s (10.80)	40 (26.0%)	8.38 (3.08)
2. Behälter (pq_assign_a2_1)	6 s	7.92 s (4.65)	54 (35.1%)	8.70 (2.94)
3. Zeiger (pq_assign_a2_8)	6 s	11.38 s (24.53)	59 (38.3 %)	8.81 (2.84)

Tab. 16: Beurteilung von Zuweisungsmodellen ohne Vernichtung. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 154 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

10.2.3 Totale Vernichtung

Bei einer Zuweisung geht zwar der vorige Inhalt einer Variablen verloren, aber der Variablenname wird nicht zerstört. Für die vollständige Vernichtung eines Objektes (Wert einschließlich des zugehörigen Namens) enthalten Programmiersprachen besondere Anweisungen. Bei Python z.B. bewirkt die Anweisung

```
del a
```

die Entfernung des Namens a aus dem Namensraum. Die Folge ist z.B., dass anschließend kein Zugriff mehr auf a möglich ist und eine Anweisung wie

```
print a
```

zu einem Laufzeitfehler führt. Variablennamen sind also – im Unterschied etwa zu Daten – stabile Elemente in der Mechanik eines Programmlaufs. Sie existieren von der Deklaration oder ersten Zuweisung bis zum Ende des Programmlaufs. Datenflüsse – nach Pennington (1987) eine wichtige Abstraktion eines Programmtextes – oder das Konzept der Rollen (Sajaniemi 2002) implizieren die Dauerhaftigkeit von Variablen. Dennoch können sich viele Menschen vorstellen, dass bei einer Zuweisung

```
today = "Tuesday"
```

zuerst das komplette existierende Objekt mit dem Namen `today` zerstört und dann ein neues Objekt mit gleichem Namen geschaffen wird. Im Python Quiz „Modeling assignments“ (Aufgabe 2) akzeptierten 96 von 154 beobachteten Schülerinnen und Schülern ein solches Modell.[12] Dieses Ergebnis zeigt, dass Anfänger häufig nicht deutlich zwischen dem Namen und dem Wert einer Variablen differenzieren, sondern beides als eine Einheit sehen.

10.3 Veränderung (Metamorphose)

Wenn ein Baum wächst, ein Auto neu lackiert oder ein Pudel frisiert wird, verändern sich Objekte, aber sie behalten in unserer Vorstellung ihre Identität. Es ist immer noch derselbe Baum, dasselbe Auto und derselbe Pudel, nur einzelne Aspekte der Erscheinung haben sich geändert. Aus dem Alltag sind uns radikale Wechsel der äußeren Erscheinung bekannt:

- Aus einer Raupe wird ein Schmetterling (Metamorphose), aber es bleibt dasselbe Individuum.
- Wenn Wasser verdampft, bleibt es derselbe Stoff. Nur der Aggregatzustand ändert sich.
- Wenn Eisen und Schwefel chemisch reagieren, entsteht ein neuer Stoff mit neuen Eigenschaften (Eisensulfid). Aber die Atome bleiben erhalten. Im Eisensulfid sind die Atome der Ausgangsstoffe nur anders angeordnet.

Moderne Programmiersprachen unterstützen das Konzept der Veränderung. Man kann komplexe Objekte wie z.B. Listen definieren, die veränderbar sind. Bei einer Liste können z.B. Elemente ent-

[12] Die Vorstellung einer totalen Vernichtung erwies sich sogar als besonders hartnäckig. Schüler, die das Spiel mehrfach spielten, bewerteten diese Visualisierung immer wieder als passend, obwohl das System jedes Mal mit Punktabzug reagierte (Einzelheiten in Anhang 5.3)

fernt, ersetzt oder neue Elemente eingefügt werden. Dabei behalten sie ihre Identität. Aber auch unabhängig von den Konstrukten, die die Programmiersprache bereithält, verwenden wir in intuitiven Modellen zur Datenverarbeitung verschiedene Vorstellungen von Veränderungen.

10.3.1 Datenumwandlungen

In manchen intuitiven Modellen zum Erklären und Verstehen von Programmen verändern sich Entitäten, die Daten repräsentieren. Abb. 48 zeigt ein Beispiel. Hier visualiert der 17-jährige Schüler S. die Java-Anweisung

```
b = a.toUpperCase()
```

durch einen Akt der Magie. Nach der Berührung mit einem Zauberstab verwandelt sich ein Wort aus Kleinbuchstaben in ein Wort aus Großbuchstaben. Man könnte sagen, es ist das gleiche Wort geblieben nur die Schreibweise hat sich geändert. In dieser Betrachtungsweise erhält man die Kontinuität der Entität, indem man sich von der Ebene der Daten (Zeichenkette) auf die Ebene des Wissens, das durch die Daten repräsentiert wird, begibt.

In einer detailgenauen Interpretation dieses Programmstücks passiert eigentlich folgendes: Das Stringobjekt `a` produziert mit Hilfe seiner Methode `toUpperCase()` einen zweiten String der aus `a` berechnet wird. Dieser Aspekt, dass das Original eigentlich unversehrt bleibt und ein zweites, neues Objekt entsteht, wird bei diesem Beispiel im Konzept der Umwandlung einfach ignoriert.

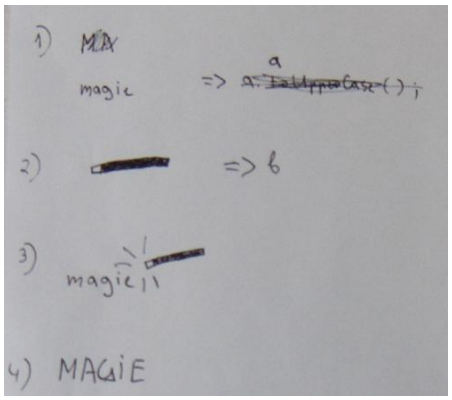


Abb. 48: Visualisierung der Ausführung eines Methodenaufrufs durch die Metamorphose einer Datenentität

Weitere Beispiele für Umwandlungen von Datenentitäten sind (mehr dazu in Anhang 5.4):

- Interpretation von cast-Anweisungen als Verändern des Typs einer Datenentität.
- Die Vorstellung der „Zerlegung“ einer Zahl in Faktoren.
- Verändern von Listen (Löschen oder Hinzufügen von Elementen).

10.4 Namenumwandlungen

In Kapitel 6 haben wir Namen als Mechanismus beschrieben, Entitäten zu identifizieren und zu adressieren. Intuitive Modelle zur Arbeitsweise von Programmen enthalten manchmal Umbenennungen. Wenn eine Funktion einen Wert als Parameter übernimmt, ist dies mit einer Umbenennung verbunden. Stellt man sich die Funktion als Box mit Ein- und Ausgang vor, dann wird der Funktion als Eingabe ein Objekt übergeben, das im Innenraum der Funktion unter einem anderen Namen gehandhabt wird als in der Umgebung (visuelles Beispiel in Anhang 5.5).

Ein zweiter Zusammenhang, in dem Namensumwandlungen eine Rolle spielen, sind Rechenprotokolle zur Darstellung der Arbeitsweise einer Funktion. Ein mathematischer Term oder Funktionsaufruf kann als Name für einen (noch unbekannten) Wert interpretiert werden.

Viele Menschen machen sich die Arbeitsweise einer solchen Funktion klar, indem sie – mit einem Funktionsaufruf beginnend – eine Folge von Termen aufschreiben, die ein Rechenprotokoll darstellt. Beim Übergang von einem Term zum nächsten wird entsprechend den Anweisungen der Funktionsde-

finition ein Subterm durch einen anderen (äquivalenten) Term ersetzt. Beispiel (rekursive Berechnung der Fakultät):

```
fak(4)
4 * fak (4-1)
4 * fak(3)
4 * 3 * fak(3-1)
usw.
```

Am Ende eines solchen Rechenprotokolls steht ein Wert. Alle vorhergehenden Terme kann man als Namen dieses Wertes auffassen. Jeder einzelne Umformungsschritt ist somit eine Modifikation eines (indirekten) Namens. Beobachtungen mit der PVS haben gezeigt, dass viele Menschen Rechenprotokolle gegenüber strukturorientierten Visualisierungen bevorzugen (siehe Anhang 5.5 und 5.6).

10.5 Bewegen

Menschen sind mobile Wesen. In der Entwicklungsgeschichte der Menschheit war für Jahrmillionen die Beherrschung von Bewegungsvorgängen (z.B. im Zusammenhang mit Fliehen, Jagen und Standortverlegungen) für das Überleben entscheidend. In der Alltagssprache werden Veränderungen häufig durch Bewegungsmetaphern beschrieben. Man redet vom „Aufstieg in die erste Bundesliga", „Höhenflug der Aktienkurse" oder beobachtet „Bewegung auf dem Arbeitsmarkt". In der Mathematikdidaktik gibt es eine lange Tradition, arithmetische Operationen durch Bewegungen auf der Zahlengeraden darzustellen (Lakoff & Núnez 1997).

Mit Bewegungen werden auch in der Informatik Verarbeitungsprozesse veranschaulicht. In diesem Abschnitt untersuchen wir Modelle, die Bewegungen von Daten und Namen verwenden. Vorstellungen zur Bewegung von Botschaften in der Objektorientierten Programmierung werden im Zusammenhang mit Objekten in Abschnitt 12.5 behandelt. Modelle für Funktionsaufrufe können mit einer Bewegung der Funktionsentität verbunden sein, nämlich dann, wenn wir uns vorstellen, dass die Funktion wie ein Werkzeug zu einer zu bearbeitenden Daten-Entität gebracht wird.

10.5.1 Bewegung von Daten

Wenn Daten-Entitäten mit einem Aufenthaltsort assoziiert werden, können Verarbeitungsprozesse durch Bewegungen – also Ortswechsel der Daten – dargestellt werden.

- In einem Venn-Diagramm wird die Zugehörigkeit zu einer Menge dadurch festgelegt, dass ein Element sich innerhalb eines (z.B. durch eine geschlossene Linie) markierten Bereichs aufhält. Beispielsweise kann das Entfernen eines Elementes (Daten-Entität) aus der Menge dann so dargestellt werden, dass es aus diesem Bereich herausgeholt wird.
- Modelle von Listen oder Arrays, bei denen Items nebeneinander liegen oder in einem Behälter mit mehreren Fächern aufbewahrt werden, können sortiert werden, indem die Elemente die Plätze wechseln.
- Eine Funktion wird aufgerufen, indem eine Daten-Entität in eine Box wandert, die die Funktion (genauer den Prozess, der beim Aufruf einer Funktion generiert wird) repräsentiert (Datenflussmodell).
- Eine Zuweisung der Form `b = a` kann durch Bewegung einer Daten-Entität (oder einer Kopie) von Behälter a nach Behälter b visualisiert werden.
- Ein Baum, in dem die Eltern-Kind-Beziehung durch räumliche Nähe der Knoten dargestellt wird, kann durch Verschieben der Knoten umstrukturiert werden.

Obwohl Graphen räumliche Darstellungen sind, können sie nur in Ausnahmefällen durch (alleinige) Bewegung der Knoten verändert werden. Da bei einem Graphen jeder Knoten mit beliebig vielen anderen Knoten über Kanten verbunden sein kann, spielt der Ort des Knoten keine Rolle. Nur in sehr einfachen Strukturen (Bäume, lineare Strukturen oder Ringe) lassen sich die Kanten durch räumliche Nähe (Nachbarschaft) darstellen (s. Abb. 49). Allein dann kann der Graph durch Datenbewegung umstrukturiert werden.

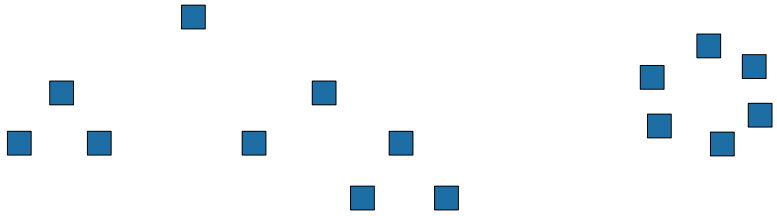


Abb. 49: Visualisierung von Graphen (Baum, Ring) ohne explizite Repräsentation der Kanten

10.5.2 Modellierung von Zuweisungen durch Datenbewegung

Bewegungsmodelle entstehen, wenn man Materialbewegung aus dem Alltag (Transport von Gegenständen, Flüsse) auf die Verarbeitung von Daten überträgt. Ein bekanntes Beispiel ist die Visualisierung der Zuweisung `b = a` durch Transport des Inhalts von `a` nach `b`.

In der PVS-Applikation Python Visual *Assign* wurden verschiedenen Modelle mit Datenbewegungen zur Visualisierung der Anweisungsfolge

```
a = 3
b = a
```

angeboten (s. Abb. 50).

(1) Naive Bewegung einer Datenentität von `a` nach `b`.

(2) Bewegung einer Kopie des Inhalts von `a` nach `b`.

(3) Der Behälter `b` holt sich mit Hilfe eines Greifarms eine Kopie des Inhalts von Behälter `a`.

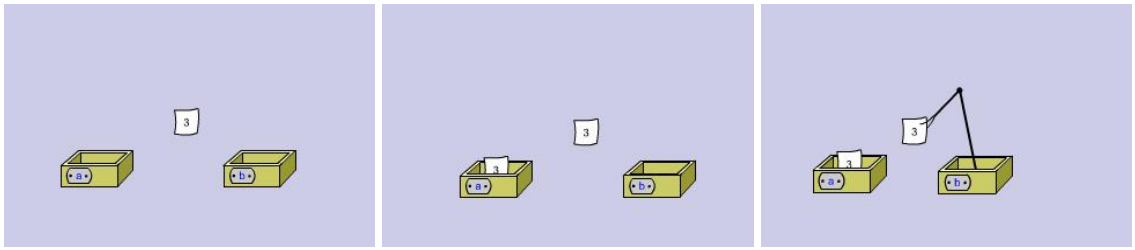


Abb. 50: Modelle mit Datentransport zur Visualisierung einer Zuweisung: Naiver Transport (pq_assign_a1_4), Transport einer Kopie (pq_assign_a1_5) und Holen einer Kopie (pq_assign_a1_7)

Das naive Bewegungsmodell wurde von etwa der Hälfte der Teilnehmer (48.1 %) akzeptiert, aber die Modelle, die die Bewegung einer Kopie darstellten, signifikant bevorzugt (Fisher-Test, p = 0.000). Im dritten Modell ist der Behälter mit Aufschrift b der Akteur, der den Datentransport bewerkstelligt. Offenbar können sich die meisten Schüler (81.2%) eine Zuweisung auch als „Daten holen“ vorstellen. Die Beurteilungen der beiden letzten Modelle unterscheiden sich nicht signifikant. Dies lässt sich so interpretieren, dass die Frage des Akteurs bei der Interpretation von Zuweisungen keine Rolle spielt. Insbesondere gibt es keinen „kognitiven Vorteil“ (mehr Intuitivität), wenn man sich eine Zuweisung so vorstellt, dass eine Entität (Variable der linken Seite der Zuweisung) sich einen neuen Wert von einer anderen Entität (Variable der rechten Seite der Zuweisung) holt.

n = 154	Dauer der Animation	Entscheidungs-zeit (Stdabw.)	Als passend beurteilt von	Konfidenz (Stdabw.)
Naïve Bewegung (pq_assign_a1_4)	6 s	12.02 s (8.19)	74 (48.1%)	81.8% (33.7%)
Bewegung einer Kopie (pq_assign_a1_5)	6 s	8.81 s (6.22)	133 (86.4%)	87.7% (27.6%)
Holen einer Kopie (pq_assign_a1_7)	9 s	12.68 s (11.56)	125 (81.2%)	86.4% (24.4%)

Tab. 17: Beurteilung von Modellen zur Veranschaulichung von Zuweisungen der Form `b = a`. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 154 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

Weitere Beispiele für die Bewertung von Modellen mit Datenbewegungen werden in Anhang 5.7 diskutiert.

10.5.3 Modellierung von Zuweisungen durch Namenbewegungen

Namenbewegungen finden statt, wenn Daten als unveränderbare, „inerte“ Entitäten betrachtet werden und bestimmte Datenentitäten durch Zuordnung eines neuen Namens markiert und damit in einen neuen Sinnzusammenhang gestellt werden.

Namenbewegungen erlauben einfache Modelle für Iterationen über Sequenzen und Bäumen. Dabei wandert z.B. der Name des aktuellen Elementes über die Entitäten der Kollektion. Im Modell einer Suchoperation (z.B. Suche nach einem Minimum in einer unsortierten Liste) kann der „most-wanted-holder“ (bei einer Minimumsuche der kleinste bisher gefundene Wert, vgl. Sajaniemi 2002) durch einen weiteren beweglichen Namen gekennzeichnet werden.

Betrachten wir nun die Rolle von Namenbewegungen bei der Interpretation von Zuweisungen. Abb. 51 zeigt Screenshots aus Animationen der PVS, die sich auf die beiden Anweisungen

```
a = 3
b = a
```

beziehen.

(1) Im ersten Modell (pq_assign_a1_9) schweben Karten mit Zahlen durch den Raum. Ein Klebezettel mit Aufschrift a fliegt ins Bild, bewegt sich zur Karte mit der Zahl `3` und bleibt an ihr kleben (`a = 3`). Ein zweiter Klebezettel mit Aufschrift `b` erscheint und heftet sich an den Zettel mit Aufschrift a. Dieses Modell ist inkonsistent (und wird vom System als ungeeignet bewertet), weil einmal der Klebezettel als Name eines Datums (3) fungiert und das zweite Mal als Name eines Namen (`b` als Name für `a`). Dies ist ein Beispiel einer Kettenbildung (siehe Abschnitt 6.8).

(2) In der zweiten Animation (pq_assign_a1_6) wird das Behältermodell für Variablen verwendet. Zur Darstellung der zweiten Zuweisung entsteht eine Kopie des Namensschildes `a` und bewegt sich in den Behälter mit Aufschrift `b`. Dieses Modell ist in sofern ungeeignet (und wird auch vom System als solches bewertet), als ein Kasten einmal als Behälter für Daten (Zahlen) und einmal als Behälter für Namen dargestellt wird. Wiederum handelt es sich um eine Namenskette, die der Semantik des Programms nicht entspricht.

(3) In der dritten Animation (pq_assign_a1_10) wird die zweite Zuweisung durch Anbringen eines zweiten Namensschildes (`b`) an den Behälter mit Inhalt `3` visualisiert. Dieses Modell ist (im Unterschied zu den vorigen Modellen) insofern akzeptabel, als die Schilder a und b gleichermaßen als Name für die Zahl `3` dargestellt werden. Gleichwohl ist dieses Modell problematisch, wenn man für komplexere Programme verwenden möchte. Was passiert, wenn anschließend die Anweisung `a = 3` ausgeführt werden soll? Das Modell suggeriert die Vorstellung, dass dann in dem Behälter die `2` durch eine `3` ersetzt würde. Dies ist aber falsch, da Variable `b` ihren alten Wert (`2`) behält. Stattdessen müsste man annehmen, dass ein neuer Behälter entsteht, der das neue Datum `3` aufnimmt und das Namensschild a erhält.

(4) Die vierte Animation (pq_assign_a1_8) schließlich enthält eine konsistente und korrekte Visualisierung von Zuweisungen durch Namenbewegungen. Der Name `b` bewegt sich zunächst in Richtung Kle-

bezettel `a` und heftet sich dann an die gleiche Datenentität (Karte mit der Zahl `3`), an der sich bereits der Name `a` befindet.

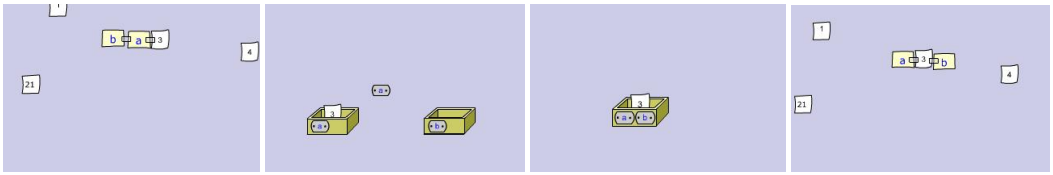


Abb. 51: Modelle mit Namenbewegung zur Visualisierung einer Zuweisung: Namenskette (pq_assign_a1_9) Transport der Kopie eines Namens (pq_assign_a1_6), zweiter Name für Behälter (pq_assign_10) und zweiter Name für Datum (pq_assign_a1_8)

n = 154	**Dauer der Animation**	**Entscheidungs-zeit (Stdabw.)**	**Als passend beurteilt von**	**Konfidenz (Stdabw.)**
Namenskette (pq_assign_a1_9)	9 s	13.75 s (11.58)	78 (50.7%)	78.2% (30.7%)
Transport der Kopie eines Namen (pq_assign_a1_6)	6 s	10.50 s (7.75)	98 (63.6%)	87.2% (24.1%)
Zweiter Name für Behälter (pq_assign_a1_10)	6 s	17.31 s (18.40)	103 (66.9%)	77.6% (31.3%)
Zweiter Name (pq_assign_a1_8)	9 s	14.05 s (12.56)	119 (77.27%)	82.1% (31.1%)

Tab. 18: Beurteilung von Modellen zur Veranschaulichung von Zuweisungen der Form `b = a`. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 154 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

Am unproblematischsten gibt das vierte Modell Zuweisungen durch Namensbewegungen wieder. Fast vier Fünftel (119) der 154 Schüler, die *Python Visual Assign* gespielt haben, waren bereits beim ersten Spieldurchgang überzeugt, dass es eine geeignete Veranschaulichung ist (Tab. 18). Allerdings schnitt es signifikant schlechter ab als das Behältermodell (Abb. 50 Mitte), bei dem die Zuweisung `b = a` durch Transport einer Kopie des Inhalts von `a` nach `b` visualisiert wurde (Fisher-Test, $p = 0.04$).

Betrachten wir nun die beiden ersten Animationen, die ungeeignete Namenbewegungen mit Kettenbildung enthalten. Beide Modelle wurden von den meisten der 154 Schülerinnen und Schüler positiv bewertet. Doch zeigte sich, dass das falsche Behältermodell (pq_assign_a1_6) eher als richtig akzeptiert wurde (63.6 %) als das falsche behälterfreie Modell (pq_assign_a1_9 mit 50.7%). Der Unterschied ist signifikant ($p = 0.029$).

Bei Vielspielern konnte zudem beim behälterfreien Modell ein höherer Lerneffekt beobachtet werden (Abb. 52 links). Im dritten Spiel haben 90% der 41 Schüler dieses Modell als ungeeignet eingestuft im Vergleich zu nur 76% beim Behältermodell.

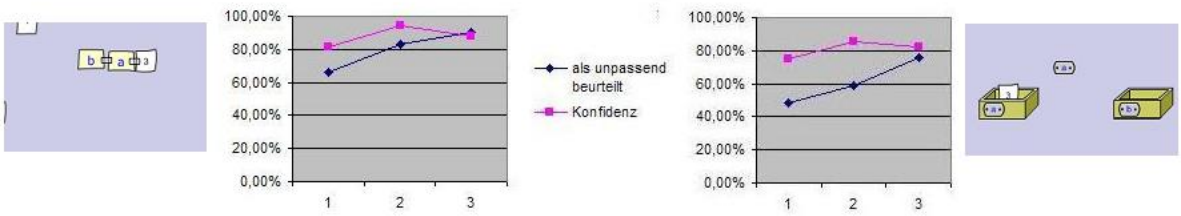


Abb. 52: Beurteilung unpassender Modelle mit Namenbewegungen für Zuweisungen. Lernkurven bei Vielspielern (n = 41) für ein behälterfreies Modell (pq_assign_a1_9, links) und ein Modell mit Behälter (pq_assign_a1_6, rechts)

Das Behältermodell mit falschen Namenbewegungen ist persistenter als das behälterfreie Modell. Trotz Feedback, Reflektion und in der Regel auch Diskussion mit Parallelspielern erkennen ein Viertel der Vielspieler auch im dritten Durchgang nicht, was an dem Modell nicht stimmt. Das wirft ein gewisses Licht auf das Modell „Variablen sind Behälter für Daten“ als solches. In diesem Zusammenhang scheint es zu falschen Vorstellungen „zu verführen“. Der Fehler der Kettenbildung wird dagegen im Etikettmodell leichter erkannt und in späteren Spieldurchgängen vermieden.

Zum Schluss dieses Abschnitts vergleichen wir noch verschiedene Zeigermodelle zur Darstellung von Zuweisungen (Abb. 53). Zeiger sind wie Etiketten bewegliche Namen für Objekte, besitzen allerdings in ihrem Ursprung (breiteres Ende) einen ruhenden Punkt. Bei Zeigerbewegungen bleibt also der Ursprung mit der Bezeichnung des Namens (hier: `a` und `b`) ortsfest und die Spitze wandert – unter gleichzeitiger Verformung des Pfeils – zur benannten Entität.

(1) Das erste Modell ist ein inverses Zeigermodell. Der Ablauf ist folgender: Zunächst entsteht ein Zeiger aus der Karte mit der `3`, der sich zum Namensschild `a` hin entwickelt, danach „wächst“ ein zweiter Zeiger zum Namensschild `b`. Die Animation bringt damit die Vorstellung zum Ausdruck, dass die Aktivität bei der Namenszuordnung vom benannten Objekt ausgeht. D.h. das Objekt `3` sucht über den Pfeil, der aus ihm herauswächst, selbst die Verbindung zu seinen Namen. Es handelt sich eigentlich nicht um eine Namenbewegung, eher ähnelt es einer Datenbewegung, da die Datenentität aktiv ist.

(2) Das zweite Modell ist ein Beispiel für eine Kettenbildung. Aus dem Namensschild `b` wächst ein Pfeil, der auf das Namensschild `a` zeigt. Da in der Semantik des Programmtextes sowohl `a` als auch `b` Namen für eine Zahl sind, ist dieses Modell unpassend.

(3) Das dritte Modell ist eine plausible Anwendung des Zeigermodells. Aus dem Namensschild `a` wächst ein Zeiger zur Karte `3`. Bei der Visualisierung der zweiten Zuweisung ist der genaue Bewegungsvorgang von Bedeutung. Die Zeigerspitze von `b` bewegt sich zuerst zum Namensschild `a` und „tastet“ sich am Zeiger von `a` entlang bis zur Karte `3`.

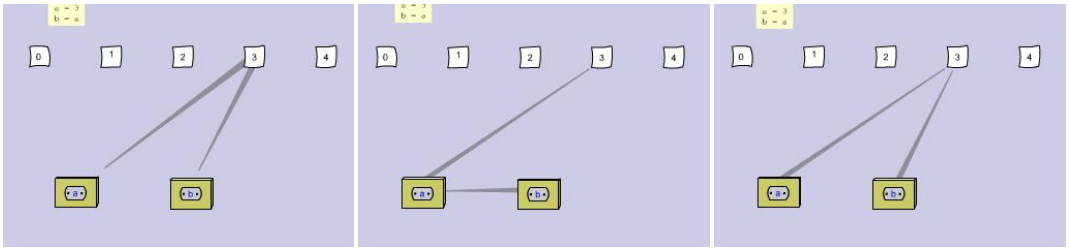


Abb. 53: Modelle mit Zeigerbewegung zur Visualisierung von Zuweisungen: Zeiger vom Objekt zum Namen (pq_assign_a1_13) Zeigerkette (pq_assign_a1_12) und zwei Zeiger zur Zahl (pq_assign_11).

Fast die Hälfte aller beobachteten Schülerinnen und Schüler (70 von 154) hielten das inverse Zeigermodell (Modell 1) für korrekt. Die Lernkurve von Vielspielern (Abb. 53 links) indiziert eine hohe Persistenz dieser Vorstellung. Das inverse Zeigermodell wurde vom System als unpassend gewertet. Nur wer sich dieser Einschätzung anschloss, erhielt Pluspunkte. Trotz „Bestrafung“ durch das System haben im zweiten Durchgang nur wenig mehr Personen das inverse Zeigermodell als unpassend eingeschätzt. Erst im dritten Durchgang stieg ihr Anteil auf etwa 71% an.

Die Verkettung von Zeigern in Modell 2 entspricht der bereits diskutierten Verkettung von Namen (Abschnitt 6.8) und ist unpassend. Dennoch wurde es von mehr als der Hälfte der Schülerinnen und Schüler (87 von 154) akzeptiert. Im Unterschied zum inversen Zeigermodell konnte man aber eine steile Lernkurve beobachten. Bereits in der zweiten Sitzung bewerten 85% der Vielspieler das Modell als unpassend (Konfidenz 90%). Dies kann man so deuten, dass an diesem Modell der „Denkfehler" einer Namensverkettung besonders leicht erkannt wird.

Modell 3 ist ein von der Fachgemeinschaft allgemein akzeptiertes Modell und wurde von den Schülern in der ersten Sitzung gegenüber den beiden anderen den beiden anderen Zeigermodellen signifikant häufiger akzeptiert.

n = 154	**Dauer der Animation**	**Entscheidungs-zeit (Stdabw.)**	**Als passend beurteilt von**	**Konfidenz (Stdabw.)**
Zeiger vom Objekt zum Namen (pq_assign_a1_13)	5 s	13.35 s (13.77)	70 (45.5%)	76.0% (35.8%)
Zeigerkette (pq_assign_a1_12)	5 s	12.07 s (13.75)	87 (56.49%)	88.6% (24.6%)
Zwei Zeiger zur Zahl (pq_assign_a1_11)	6 s	13.36 s (11.38)	119 (77.27%)	83.1% (29.8%)

Tab. 19: Beurteilung von Modellen zur Veranschaulichung von Zuweisungen mit Zeigern. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 154 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

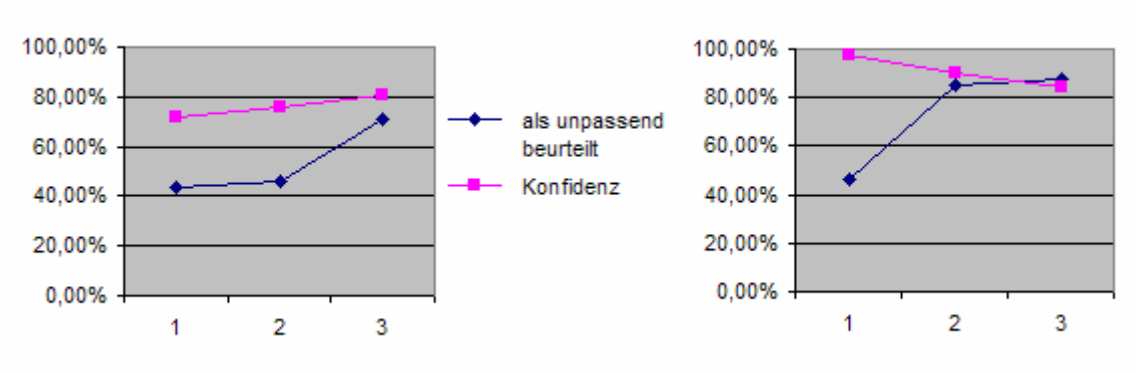


Abb. 54: Beurteilung als unpassend eingestufter Modelle mit Zeigerbewegungen. Lernkurven bei Vielspielern (n = 41) für Zeiger vom Objekt zum Namen (pq_assign_a1_13, links) und Zeigerkette (pq_assign_a1_12, rechts)

11 Klassen

11.1 Intuitive Modelle in der Objektorientierten Programmierung

Ein objektorientiertes Programm ist ein System von Objekten, die Botschaften austauschen. Jedes Objekt besitzt Attribute, die Wissen repräsentieren, und beherrscht Operationen, mit denen das gekapselte Wissen verarbeitet werden kann. Eine Besonderheit der OOP ist, dass das mentale Modell von interagierenden Entitäten Bestandteil des Paradigmas ist und mit der Begrifflichkeit der Fachsprache beschrieben wird. Objektorientierte Programme sollen häufig Realitätsausschnitte modellieren (Hubwieser 1999, 2004; Schubert & Schwill 2004) sind aber in der Regel nicht völlig strukturtreu. Das Hauptmotiv der informatischen Modellierung ist Denkökonomie. Das heißt, das Design ist so angelegt, dass möglichst gut verständliche Systeme entstehen. Nun gibt es einen seltsamen Doppelbezug. Einerseits soll häufig das informatische System tatsächlich ein Modell eines Realitätsausschnitts sein. In diesen Fällen ist die Grundlage des Entwurfs ein Fachkonzept, das sich mit der Realität (z.B. die Verwaltungsabläufe in einer Bibliothek) beschäftigt und sie analysiert. In dieser Realität liegt der praktische Nutzen des Programms (z.B. Verwaltung von Büchern in einer Bibliothek). Auf der anderen Seite aber dienen Teile dieser Realität (Bücher, Leser, ...) als Metaphern für das geschaffene Programm und helfen, das System übersichtlich und durchschaubar zu machen. Diese Metaphern unterscheiden sich allerdings erheblich von den realen Objekten. Sie sind keine Modelle im naturwissenschaftlichen Sinn.

11.2 Klassenbegriff

Der Begriff Klasse (class) bezeichnet ein allgemeines Muster für Objekte (Instanzen, Inkarnationen) mit gemeinsamen Attributen und Methoden. Nach dem Paradigma der OOP besteht Programmierung letztlich aus der Definition von Klassen. Ein konsequent objektorientiertes Programm besteht aus einem einzigen Objekt einer Klasse, das aus anderen Objekten anderer Klassen zusammengesetzt ist. Je nach Anwendungskontext werden völlig unterschiedliche intuitive Modelle für Klassen verwendet, wie in diesem Abschnitt erläutert werden wird.

Die Idee der Klasse ist sehr alt und kann bis in die griechische Philosophie der Antike zurückverfolgt werden. Aristoteles (384 – 322 v. Chr.) betrachtete es als das Hauptanliegen seiner insgesamt 170 Werke, die Dinge der Welt zu klassifizieren (nach Taivalsaari 1996). Objekte gehören zur selben Kategorie, wenn sie gemeinsame Eigenschaften besitzen. Die Kategorie selbst wird durch eine Gruppe von Eigenschaften definiert. Durch Spezifikation gemeinsamer und unterschiedlicher Eigenschaften können aus bekannten Kategorien neue gewonnen werden. Der aristotelische Klassenbegriff ist die philosophische Grundlage für das Klassenkonzept objektorientierter Programmiersprachen (Taivalsaari 1996).

11.3 Klasse als Bauplan

Wer sich eine Klasse als Bauplan vorstellt, begibt sich in die Rolle eines Architekten, der spezifiziert, wie ein Objekt einer Kategorie später aussehen soll. Der Bauplan ist die Dokumentation aber noch nicht die Sache selbst. Um aus einem Bauplan ein Objekt zu generieren, braucht man einen Akteur (Konstruktor) und Baumaterial. In OO-Programmiersprachen wie Java oder Python enthalten Klassendefinitionen eine besondere Methode zur Initialisierung eines Objektes, die man als Konstruktor bezeichnet. Es macht aber keinen Sinn, diese Methode als Kreationsoperation zu begreifen, mit der ein Objekt erzeugt wird. Denn wie soll ein Objekt eine Operation ausführen, wenn es noch gar nicht existiert? Vielmehr enthalten diese Methoden einen Mechanismus, der variable Teile eines Objektes mit Anfangswerten belegt (Initialisierung). Die eigentliche Schaffung eines Objektes muss durch einen Akteur außerhalb des geschaffenen Objektes bewerkstelligt werden. Die Bauplanmetapher spielt eine große Rolle beim Entwurf der Architektur eines Softwaresystems im Sinne eines Software-Engineerings (z.B. Sommerville 1997). Hier werden die technischen Komponenten eines Systems vollständig erfasst und in einen strukturellen Zusammenhang gebracht, der z.B. in einem UML-Klassendiagramm dokumentiert wird.

11.4 Klasse als Fabrik für Objekte

Bei der Fabrik-Metapher stellt man sich eine Klasse als Akteur vor, der Objekte bestimmter Art produzieren kann. Im Unterschied zur Bauplan-Metapher wird das Augenmerk auf die Instanzierung von Objekten – also die „Produktion" – gelegt[13].

Die Programmiersprache Python unterstützt die Fabrik-Metapher, da Klassen – so wie Funktionen – aufrufbare Objekte („callable objects") sind, die ein Objekt generieren und zurückgeben.

11.5 Klasse als Menge von Objekten

Eine Klasse kann als Menge von Objekten eines bestimmten Typs vorstellen. In der Philosophie spricht man von der Extension einer Kategorie. So verbindet man häufig mit numerischen Klassen wie `int` oder `float` (Python) den Gedanken an die Menge der ganzen Zahlen oder die Menge der Gleitkommazahlen. Das Mengenkonzept wird auch im Zusammenhang mit Klassenattributen verwendet. Ein Klassenattribut kann als gemeinsames Attribut aller denkbaren (aber unter Umständen noch gar nicht existierenden) Objekte der Klasse gesehen werden.

11.6 Klasse als Prototyp

Im Prototyp-Modell stellt man sich eine Klasse durch ein konkretes Objekt vor, das gewissermaßen als Stellvertreter für alle Inkarnationen der Klasse fungiert. Im Grunde gibt es zwei Konzepte für Prototypen, nämlich die Vorstellung eines typischen oder eines unfertigen Objektes. In der Begriffspsychologie (Rosch 1973, 1975; vgl. Anhang 6.4) ist ein Prototyp ein typisches Exemplar eine Kategorie. In Anlehnung an dieses Konzept kann man eine Klasse durch ein typisches Objekt beschreiben, dem alle Objekte dieser Klasse ähneln. Bei einer Instanzierung werden Abweichungen des neu geschaffenen Objektes vom (in der Klassendefinition festgelegten) Standard spezifiziert.

Eine Klasse kann man sich auch als unfertiges Objekt – als vereinfachte, vorläufige Version – vorstellen, dem bei der Instanzierung fehlende Teile hinzugefügt werden. Beide Intuitionen können auch gleichzeitig verwendet werden.

Nun gibt es spezielle Programmiersprachen, bei denen das Prototyp-Konzept expliziter Bestandteil des Programmierparadigmas ist (Smith & Ungar 1995; Taivalsaari 1992, 1996, siehe auch Anhang 6.5). Aber auch wenn man in einem Projekt eine der etablierten objektorientierten (und nicht prototyporientierten) Programmiersprachen wie Java oder Python verwendet, kann das Prototyp-Modell von Bedeutung sein. Betrachten wir zunächst das „Programmieren im Großen".

Im Software Engineering ist Prototyping eine Methode der Software-Entwicklung (Sommerville 1997, S. 167 ff.). Hier versteht man unter einem Prototyp eine Software, die ein vereinfachtes Modell der Zielsoftware darstellt und schnell zu implementieren ist. Entscheidend ist, dass der Prototyp lauffähige Software ist, die evaluiert werden kann, etwa um Schwächen oder Lücken der Systemspezifikation festzustellen. Eine besondere Rolle spielt das Prototyp-Konzept beim agilen Programmieren. Hier ist der Entwicklungsprozess in eine Folge von Iterationen aufgeteilt, an deren Ende immer ein lauffähiges Programm steht, das mit den Kunden diskutiert werden kann. Ausgehend von einem minimalen System wird mit jeder Iteration das Produkt dem Zielprodukt immer ähnlicher. Man geht davon aus, dass zu Beginn des Prozesses die genaue Funktionalität des Zielproduktes noch gar nicht bekannt ist. Es existiert erst eine vage Vorstellung, die sich im Laufe der Entwicklung – parallel zur Evolution der Software – präzisiert.

Auch auf der Ebene des „Programmierens im Kleinen" kann die Entwicklung einer Klasse mit einer objektorientierten Programmiersprache nahezu völlig prototyporientiert sein. Anfänger, die in der Syntax und Semantik einer Programmiersprache noch unsicher sind, folgen häufig einer experimentellen Strategie, die dem Test Driven Development (TDD) des Extreme Programming (Beck 2003) ähnelt. Sie definieren in ihrem Programmtext zunächst eine minimale Klasse und ein paar Zeilen, die eine Instanzierung beinhalten und dem Testen dieser Klasse dienen. Damit haben sie einen Prototyp, ein unfertiges konkretes Objekt, das als Stellvertreter für alle Objekte der Klasse dient. In der weiteren

[13] Schülerzeichnungen mit visuellen Modellen für Klassen befinden sich in Anhang 6.2

Entwicklung wird die Klassendefinition sukzessive erweitert und alle Zwischenversionen getestet und debuggt. Obwohl man formal eine Klassendefinition schreibt, modelliert man doch – aus psychologischer Sicht – einen Prototyp. Alle kognitiven Aktivitäten drehen sich um ein einzelnes Objekt, das die Klasse repräsentiert[14].

11.7 Klasse als Behälter für Funktionen (Toolbox)

Objektorientierte Programmiersprachen (z.B. Java oder Python) erlauben die Definition statischer Methoden. Eine statische Methode kann aufgerufen werden, ohne ein Objekt der Klasse zu instanzieren. Beispiel (Java):

```
a = Math.sqrt(2)
```

Die Klasse `Math` stellt hier einen Behälter für Funktionen dar (Toolbox) dar. Statische Methoden verändern keine Objektzustände, verhalten sich also wie Funktionen im Sinne des funktionalen Programmierparadigmas. Man kann sie sich als eigene Entität – völlig losgelöst von der Klasse – vorstellen. Der Klassenname, der im Funktionsaufruf vorkommt, wird dann nicht Bezeichnung eines Objektes, an das eine Botschaft geschickt wird, interpretiert, sondern eher als Präfix eines Funktionsnamens.

[14] Ein konkretes Beispiel für diesen Prozess findet sich in Anhang 6.6.

12 Objekte

12.1 Zustand eines Objektes – Datenbesitz oder holistische Befindlichkeit

In der objektorientierten Programmierung definiert man den Zustand eines Objektes durch die Belegungen seiner Attribute (Instanzvariablen), sofern es sich um „einfache Attribute“ handelt, denen ein einfacher Wert (Datum) zugeordnet wird. Komplexe Objekte (z.B. Aggregate aus mehreren Teilen) besitzen darüber hinaus Attribute, die selbst wieder Objekte sind. Hier ergibt sich der Zustand des komplexen Objektes aus den Zuständen der verbundenen Objekte (Balzert 1999). Letztlich kann man sich den Zustand eines Objektes als eine Kombination von Daten vorstellen. Sich in einem Zustand befinden, heißt in dieser Sichtweise: Daten besitzen. Die Veränderung eines einzigen Werts impliziert die Veränderung des Gesamtzustandes des Objektes.

Neben dieser Vorstellung „Zustand als Datenbesitz“ gibt es noch ein zweites, holistisches Konzept, mit dem der Zustand eines Objektes als *Befindlichkeit* begriffen wird. Meist geht man von einer kleinen Menge von Zuständen aus, die in einem Sinnkontext von Bedeutung sind und deshalb in irgendeiner Weise mit der Funktionalität des Objektes verknüpft sind.

Bei einem Objekt, das eine Verkehrsampel modelliert, könnten diese Zustände die Ampelphasen sein: rot, rot-gelb, grün, gelb. Bei einer realen Ampel gibt es nur diese Zustände. Sie sind jeweils mit einer verkehrsrechtlichen Bedeutung verknüpft (Sinnkontext). Sie sind insofern mit Funktionalität verknüpft als nur bestimmte Übergänge von einem Zustand zum nächsten erlaubt (und sinnvoll) sind. So hat die Gelbphase die Bedeutung „Achtung, gleich kommt Rot.“ Autofahrer, die dieses Signal sehen, müssen entscheiden ob sie Gas geben oder anhalten. Angesichts dieses Sinnkontextes muss der Folgezustand die Rotphase sein, alles andere wäre sinnlos. In einem Ampel-Objekt kann die Folge der durchlaufenen Zustände durch eine Liste – also ein einziges Attribut – dargestellt werden. Hier wird das holistische Zustandskonzept verwendet. Alternativ könnte ein Ampelobjekt für jede der drei Leuchten (rot, gelb, grün) ein eigenes Attribut besitzen, das mit den booleschen Werten `True` oder `False` belegt ist, je nachdem ob die Leuchte ein- oder ausgeschaltet ist. Diese Modellierung orientiert sich stärker am technischen Aufbau einer realen Verkehrsampel (höhere Strukturtreue). Es sind nun aber auch Zustände möglich, die in der Realität nicht vorkommen und denen keine Semantik zugeordnet ist (z.B. gleichzeitiges Aufleuchten von Rot und Grün).

Das holistische Konzept der Befindlichkeit wird in der zustandsorientierten Modellierung (Hubwieser 1999, 2004) und insbesondere in Zustandsübergangsgraphen für endliche Automaten verwendet (Albert & Ottmann 1983). Hier repräsentiert man einen Zustand durch einen Knoten in einem gerichteten Graphen. Ein Zustand wird als Ort im Raum und ein Zustandwechsel als Bewegung zu einem anderen Ort (über eine Kante des Graphen) visualisiert. Ein Ort ist eine geschlossene Gestalt. Eine Entität kann nicht an mehreren Orten gleichzeitig sein. Sie kann sich nicht mit einem Aspekt ihres Seins an dem einen und mit einem anderen Aspekt an dem anderen Ort befinden.

Im Unterschied zum Modell des Datenbesitzes werden im Befindlichkeitsmodell Zustände mit aussagekräftigen Namen versehen. Ein Prozess kann sich z.B. in den Zuständen „aktiv“ oder „wartend“ befinden. Für ein Objekt, das ein Bankkonto modelliert, gibt es aber keinen eigenen Namen für den Zustand, der sich ergibt, wenn der Besitzer „Monika Gabel“ heißt und sich 1200 EUR auf dem Konto befinden.

Holland et al. 1997 haben beobachtet, dass Studenten häufig glauben, dass Objekte nur ein einziges Attribut besitzen dürfen. Sie führen dies auf bloßes Nichtwissen zurück und empfehlen einführende Beispiele mit mehreren Attributen. Die Ursache für diese Tendenz könnte aber auch im holistischen Zustandskonzept liegen. Wenn man von einem zustandsorientierten Modell für das Verhalten einer Entität ausgeht, ist es nahe liegender, die möglichen Zustände in einem einzigen Attribut zu speichern. Eine verfeinerte, weiter gehende Modellierung impliziert, dass man die Gestalt des Zustandes aufbricht und die Struktur des zu modellierenden Originals analysiert. Teilaspekte seiner Beschaffenheit müssen identifiziert und durch Attribute repräsentiert werden.

12.2 Instanzierung von Objekten – Produktion oder Auswahl

Intuitive Modelle zur Instanzierung von Objekten korrespondieren mit Vorstellungen über Klassen. Vor dem Hintergrund der Ausführungen in Kapitel 11 kann man zwei grundsätzliche Instanzierungsmodelle unterscheiden:

- Herstellung eines neuen Objektes (Klasse als Fabrik, Bauplan oder Prototyp). Argumente des Konstruktoraufrufs werden als Individualmerkmale des neuen Objektes interpretiert.
- Auswahl eines Objektes aus einem Vorrat (Klasse als Menge). Argumente des Konstruktoraufrufs werden als Auswahlkriterien interpretiert. Damit sucht der Konstruktor ein passendes, bereits existierendes Objekt heraus.

12.3 Instanzierungsmodelle in der PVS

In der PVS-Applikation Python Quiz „Objects“ werden verschiedene intuitive Modelle aus dem Bereich der OOP thematisiert. Grundlage ist folgende Klassendefinition (Python):

```
class Container (object):
    def __init__(self, max):
        self.max = max
        self.content = 0

    def fill (self, volume):
        self.content += volume
        if self.content > self.max:
            self.content = self.max    # overflow

    def empty (self):
        volume = self.content
        self.content = 0
        return volume
```

Die Klasse `Container` modelliert Behälter mit einem begrenzten Fassungsvermögen, z.B. Flaschen für Flüssigkeiten. Mit der Methode `fill()` wird ein Behälter mit einem gewissen Volumen befüllt und mit `empty()` wird er vollständig entleert.

Ein typisches objektorientiertes Programm repräsentiert eine „magische Welt“, in der Objekte agieren können, die in der Realität völlig passiv sind. So kann im Programmtext dieses Beispiels sich eine Flasche selbst befüllen und Materie (Inhalt einer Flasche) aus dem Nichts entstehen. Die Vorstellung einer Welt aus Flaschen, deren Inhalte umgefüllt werden, ist eine Analogie, eine anschauliche Konkretisierung des formalen Programmtextes. Wie wir später sehen werden, kann es dabei vorkommen, dass auf unzulässige Weise Merkmale des modellierten Realitätsausschnitts in die Interpretation des Programmtextes einbezogen werden (siehe Abschnitt 12.4).

Alle Aufgaben verwenden diese Klassendefinition. Bei den ersten drei Aufgaben soll der folgende Programmtext (Zeile für Zeile) interpretiert werden:

```
bottle = Container (0.7)
bottle.fill(0.4)
bottle.empty()
```

Bei der vierten und fünften Aufgabe geht es um die beiden letzten Zeilen des folgenden Programms:

```
bottle = Container (0.7)
vase = Container (1.5)
bottle.fill(0.4)
vase.fill(bottle.empty())
```

Bei jeder Aufgabe ist ein Stück des Programmtextes mit einem gelben Hintergrund versehen und auf diese Weise optisch herausgehoben. Die angebotenen visuellen Modelle sollen die Arbeitsweise dieser Programmpassage veranschaulichen. Die nachfolgend verwendeten Daten zur Bewertung der Modelle als passend oder unpassend beziehen sich auf 23 erste Sitzungen im Rahmen von Workshops mit der PVS. Teilnehmer waren 21 Schüler, ein Hochschulstudent und ein Lehrer, darunter vier weibliche und 19 männliche Personen. Das mittlere Alter betrug 18,4 Jahre und die wöchentliche Zeit, in der man sich mit Programmierung beschäftigte, 3,2 Stunden (weitere Details im Anhang 6.8).

Wir vergleichen zunächst einige Modelle, die die Instanzierung eines Objektes der Klasse Container in der Anweisung `bottle = Container(0.7)` visualisieren.

(1) Das erste Modell (pq_objects_a1_2) visualisiert die Klasse `Container` als Tablett mit einem Vorrat an Flaschen unterschiedlicher Größe dar. Dem Klassenobjekt wird von oben ein Zettel mit Aufschrift `0.7` übergeben. Ein Manipulatorarm ergreift diesen Zettel, legt ihn ab, sucht eine Karaffe passender Größe heraus und gibt sie zurück. In diesem Modell produziert die Klasse also kein Objekt, sondern die Instanzierung wird als Auswahl aus einem Vorrat dargestellt (Abb. 55 erstes Bild).

(2) Die zweite Animation (pq_objects_a1_3) verwendet die Fabrikmetapher und stellt die Klasse Container als Box mit der Aufschrift `Container()` dar. Ein Zettel mit der Zahl `0.7` schwebt ins Bild, aus der Box kommt ein Manipulatorarm, und ergreift den Zettel. Die Box generiert eine Flasche (Objekt der Klasse `Container`), die die Box verlässt und an die ein Zettel mit dem Namen `bottle` geheftet wird.

(3) Das dritte Modell (pq_objects_a1_6) basiert ebenfalls auf dem Herstellungskonzept. Allerdings wird die Instanzierung eines Objektes als Ereignis durch einen Blitz mit Beschriftung `Container()` dargestellt. Die Klasse selbst wird nicht als eigenständige Entität veranschaulicht. Ein Zettel mit Aufschrift `0.7` schwebt ins Bild. Mit einem Blitz verschwindet der Zettel und stattdessen ist eine Glaskaraffe zu sehen. An sie wird ein Zettel mit dem Namen `bottle` geheftet. Die Instanzierung wird also auf die Verarbeitung eines Zahlenwertes (Metamorphose) reduziert.

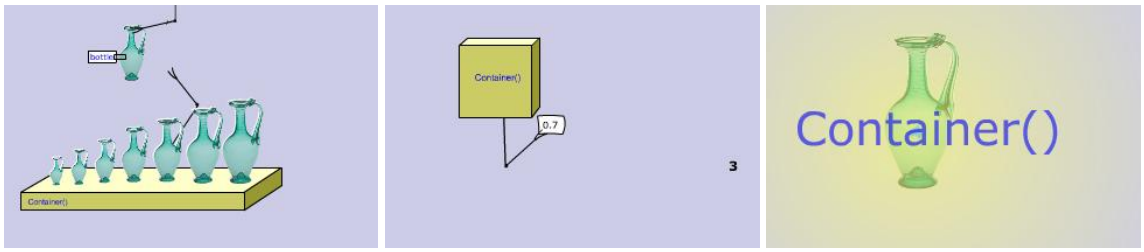


Abb. 55: Screenshots aus Animationen zur Veranschaulichung von Instanzierungen (aus Python Quiz Objects)

Tab. 20 kann man entnehmen, dass alle drei Modelle überwiegend als passend beurteilt wurden. Es lässt sich keine signifikante Bevorzugung des einen oder anderen Modells feststellen.

n = 23	**Dauer der Animation**	**Entscheidungszeit (Stdabw.)**	**Als passend beurteilt von**	**Konfidenz (Stdabw.)**
1. Auswahl (pq_objects_a1_2)	6 s	25 s (28)	17 (74%)	74% (40%)
2. Fabrik (pq_objects_a1_3)	12 s	15 s (11)	18 (78%)	80% (35%)
3. Metamorphose (pq_objects_a1_6)	6 s	11 s (12)	18 (78%)	83% (24%)

Tab. 20: Beurteilung von Modellen für Instanzierungen. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 23 Personen (darunter 21 Schülerinnen und Schülern), die an Workshops mit der PVS teilgenommen haben.

Eine weitere Animation (pq_objects_a1_5) ist eine Variante des oben beschriebenen Fabrikmodells. Wiederum wird die Klasse `Container` durch eine Box dargestellt. Der Unterschied zu Modell pq_objects_a1_3 ist, dass sie neben einem Zettel mit Aufschrift `0.7` noch einen zweiten Zettel mit dem Objektnamen `bottle` als Eingabe erhält. Anschließend verlässt eine Karaffe (Objekt) mit

Klebezettel `bottle` die Box. Hier wird die Instanzierung so interpretiert, dass die Klasse benannte Objekte produziert. Das widerspricht insofern der Semantik des Programms, als die Benennung durch die Zuweisung erfolgt und nicht bei der Initialisierung eines Objektes in der Klassendefinition. (Das Objekt kann ja später umbenannt werden.) Immerhin 14 Spieler (61 %) hielten im ersten Durchlauf dieses Modell für passend. Das Ergebnis lässt sich so interpretieren, dass die Fabrikmetapher gewissermaßen „ausstrahlt“. Sie wird nicht allein zur Beschreibung der Instanzierung verwendet sondern als Modell für die gesamte Zuweisung (Instanzierung plus Benennung). Damit verbunden ist auch die Vorstellung, dass der Name Bestandteil des Objektes ist.

12.4 Interaktion von Objekten – Verarbeitung von Botschaften

In der Objektorientierten Programmierung ist die Vorstellung interagierender Entitäten expliziter Bestandteil des Paradigmas. Objekte senden und empfangen Botschaften. Dieser Abschnitt widmet sich dem Botschaftskonzept. Wir gehen dabei auf folgende Fragen ein: In welchem Maß wird das durch eine Botschaft beauftragte Objekt als Akteur gesehen? Wie gelangt eine Botschaft zu ihrem Adressaten?

12.4.1 Objekte als Verursacher von Aktivität

Nach dem Paradigma der OOP empfängt ein Objekt eine Botschaft, wertet sie aus und führt die spezifizierte Operation aus. Der Ursprung der Aktivität liegt also im beauftragten Objekt.

Das Python Quiz „Objects“ enthält Animationen, die die Aktivität bei der Ausführung eines Auftrags auf unterschiedliche Weise auf das Objekt und seine Umgebung verteilt. Wir betrachten Modelle, die die Ausführung der Anweisung `bottle.fill(0.4)` veranschaulichen.

(1) Im ersten Modell wird die Botschaft (Oval mit Programmtext bottle.fill(0.4)) an die Karaffe (Objekt der Klasse Container) gesendet. Anschließend füllt sich die Karaffe auf magische Weise von alleine (Abb. 56). Dies ist eine passende aber weniger naturalistische Veranschaulichung des Programmtextes.

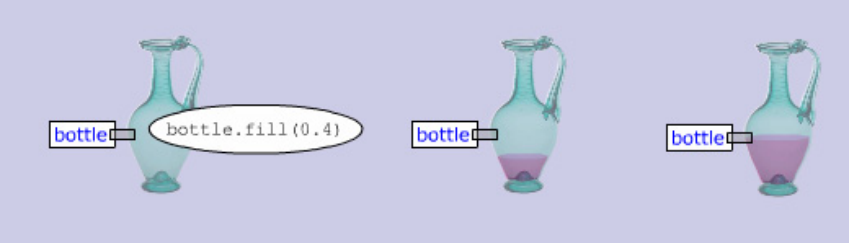


Abb. 56: Objekt als Akteur. Screenshots aus der Animation pq_objects_a2_4

(2) Das zweite Modell ist naturalistisch und stellt die Befüllung der Flasche mit einer „Abfüllstation“ (Rohr mit Hebel zum Öffnen und Schließen) dar. Die Botschaft wandert zur Abfüllstation. Diese wird nun aktiv, das Rohr bewegt sich zur Karaffe und befüllt sie. Die Karaffe selbst (Objekt der Klasse Container) bleibt völlig passiv, was dem Paradigma der OOP widerspricht (Abb. 57).

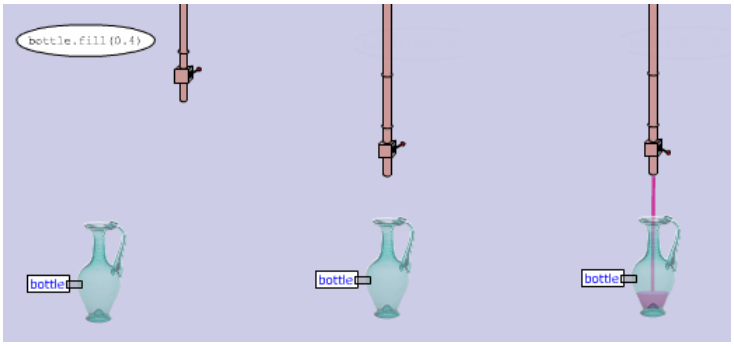


Abb. 57: Umgebung als Akteur. Screenshots aus der Animation pq_objects_a2_3

(3) In der Animation pq_objects_a2_1 wird die Botschaft durch zwei Entitäten dargestellt. Ein Oval mit der Beschriftung `fill` wird an eine Karaffe gesendet und veranschaulicht die Auswahl der Methode.

Nachdem die Karaffe diese Botschaft empfangen hat, bewegt sie sich zu einer Abfüllstation. Nun fliegt ein Zettel mit Aufschrift `0.4` (Parameter des Methodenaufrufs) zum Schließmechanismus der Abfüllstation und löst die Befüllung der Karaffe aus (Abb. 58). Die Aktivität bei der Ausführung der Operation wird hier auf das Objekt und seine Umgebung verteilt.

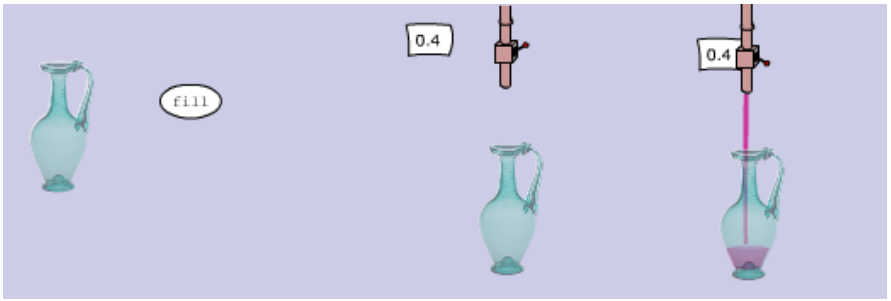


Abb. 58: Gesplittete Aktivität. Screenshots aus der Animation pq_objects_a2_1

(4) Im vierten Modell wird wiederum die Botschaft durch zwei Entitäten (Oval mit Methodenbezeichnung und Zettel mit Parameterwert) dargestellt. Das Oval mit der Beschriftung `fill` wird an die Karaffe gesendet. Diese bewegt sich sie zur Abfüllstation und bringt mit einem Greifarm das Rohr in die richtige Position. Die Karaffe empfängt nun einen Zettel mit Aufschrift `0.4` (Parameter) und betätigt mit ihrem Greifarm den Steuermechanismus der Abfüllvorrichtung. Im Unterschied zum letzten Modell ist hier allein das Container-Objekt aktiv.

Die Ergebnisse in Tab. 21 deuten an, dass alle Modelle überwiegend akzeptiert werden. Sie wurden von 70% bis 90% der Teilnehmer mit hoher Konfidenz als passend beurteilt. Dagegen waren Personen, die die Modelle ablehnten, eher unsicher (geringere Konfidenz). Aufgrund der geringen Teilnehmerzahl, können allerdings keine signifikanten Unterschiede zwischen den Modellen festgestellt werden (Fisher-Test).

n = 23	**Als passend beurteilt**	**Konfidenz (Stdabw.)**	**Als unpassend beurteilt von**	**Konfidenz (Stdabw.)**
1. Eigenbefüllung (1) (pq_objects_a2_4)	21 (91%)	81% (34%)	2 (9%)	25% (35%)
2. Naturalistische Befüllung (pq_objects_a2_3)	16 (70%)	78% (32%)	7 (30%)	76% (37%)
3. Gesplittete Aktivität (pq_objects_a2_1)	16 (70%)	97% (13%)	7 (30%)	64% (38%)
4. Eigenbefüllung (2) (pq_objects_a2_2)	20 (87%)	90% (21%)	3 (13%)	0% (0%)

Tab. 21: Beurteilung von Modellen zur Ausführung eines Auftrags mit unterschiedlicher Gewichtung der Eigenaktivität des Objekts. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 23 Personen (darunter 21 Schülerinnen und Schülern), die an Workshops mit der PVS teilgenommen haben.

Abb. 59 zeigt einen Screenshot aus einer Visualisierung der Anweisung

```
bottle.empty()
```

Aus dem Oval, das die Botschaft repräsentiert, erscheint ein Greifarm, der die Karaffe auskippt. Hier geht die Aktivität von der Botschaft aus und das Objekt bleibt passiv. Dieses Modell wurde von 19 der 23 Workshop-Teilnehmer mit einer mittleren Konfidenz von 87% als passend beurteilt.

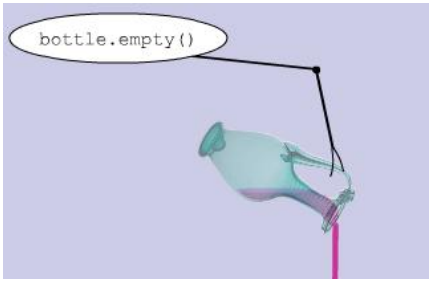


Abb. 59: Botschaft als Akteur

Die Auswertung der letzten Aufgabe (verschachtelte Botschaften) befindet sich in Anhang 6.9.

12.5 Übermittlung von Botschaften

Die Übermittlung von Botschaften vom Sender zum Empfänger ist ein Akt der Kommunikation. Wie gelangt eine Botschaft von Objekt A zu Objekt B? Wie findet es seinen Adressaten? Die PVS enthält einige Visualisierungen die man als Beispiele für folgende drei Routingmodelle betrachten kann.

(1) Der Sender der Botschaft ist für die korrekte Zustellung zuständig. Im Alltag verwenden wir dieses Modell z.B., wenn wir jemandem bei einem Gespräch in die Augen sehen und so signalisieren, dass die nun folgende Sprachäußerung für ihn bestimmt ist.

(2) Die Botschaft kennt den Adressaten und findet ihn selbsttätig. Dieses Modell wird im Alltag verwendet, wenn ein Brief von einem Boten transportiert wird.

(3) Die Botschaft wird wie eine Rundfunksendung ausgestrahlt (Broadcasting). Alle Objekte können sie empfangen und müssen selbst entscheiden, ob sie für sie selbst bestimmt ist. Dies entspricht der üblichen Gesprächssituation mit mehreren Personen in einem Raum. Jeder kann alles hören und muss selbst an ihn gerichtete Botschaften erkennen.

In Animationen zur Visualisierung der Anweisung `bottle.fill(0.4)` werden diese unterschiedlichen Vorstellungen aufgegriffen.

Zwei Animationen verwenden das Modell der direkten Zustellung durch den Sender. Im Modell pq_objects_a4_3 wird zunächst ein „Leitstrahl“ (blaue Linie) auf das Empfänger-Objekt gerichtet. Über diese Linie wandert anschließend die Botschaft (Abb. 60 erstes Bild). Im zweiten Modell betätigt ein Manipulatorarm einen Knopf mit der Beschriftung `fill()` auf einem Kasten, der zur visuellen Repräsentation des Objektes bottle gehört. Offenbar wählt die aufrufende Entität zunächst die Methode aus, die ausgeführt werden soll. Durch diesen Akt des „Anfassens“ wird das Objekt zum Empfänger der Botschaft gemacht. Im nächsten Schritt fliegt dann ein Zettel mit dem Parameter 0.4 zum angesprochenen Objekt (Abb. 60 zweites Bild).

In der Animation pq_objects_a4_2 schwebt ein Oval mit der Beschriftung `bottle.fill(0.4)` ins Bild, bewegt sich zunächst zu der Vase und schließlich zur Karaffe, verschwindet mit einem Blitz. Darauf hin füllt sich die Karaffe auf magische Weise mit einer gewissen Menge Flüssigkeit (Abb. 60 drittes Bild). Hier sucht sich also die Botschaft selbst ihren Weg zum Empfänger.

Im letzten Modell schließlich schweben von oben viele Ovale mit der Botschaft über die Bildfläche. Eines davon trifft auf das Objekt `bottle` und löst die Aktivität (Befüllung) aus (Abb. 60 rechtes Bild).

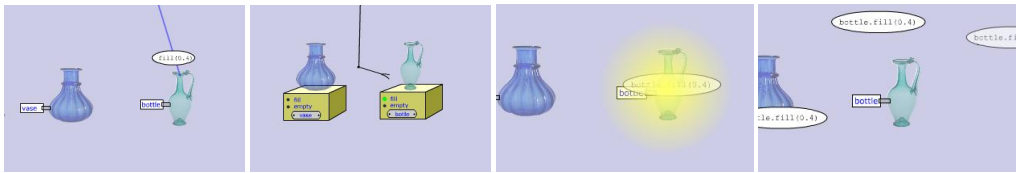


Abb. 60: Modelle mit unterschiedlicher Darstellung des Transports einer Botschaft zum Empfänger: Leitstrahl, Anfassen, selbstständige Suche und Broadcasting

Die Beurteilungen dieser Modelle durch 23 Teilnehmer von Workshops mit der PVS (Tab. 22) lassen erkennen, dass alle drei intuitiven Modelle eine gewisse psychische Realität besitzen. Allerdings wurde das Broadcasting-Modell gegenüber den Modellen „Anfassen“ und „Eigenständige Suche“ signifikant seltener als passend beurteilt (Fisher-Test, $p = 0.036$ bzw. $p = 0.006$). Die vier Modelle standen in unmittelbarem zeitlichem Kontext innerhalb einer Aufgabe. Sie wurden innerhalb einer Minute von den Spielern wahrgenommen, so dass davon auszugehen ist, dass sie von den Betrachtern verglichen worden. Vor diesem Hintergrund ist bemerkenswert ist, dass gerade das Modell „Anfassen“, welches als einziges die Botschaft nicht als eigene Entität modelliert, besonders häufig als passend beurteilt akzeptiert wurde.

n = 23	**Dauer der Animation**	**Entscheidungszeit (Stdabw.)**	**Als passend beurteilt von**	**Konfidenz (Stdabw.)**
Leitstrahl (pq_objects_a4_3)	7 s	11.3 s (12.8)	14 (61%)	78% (32%)
Anfassen (pq_objects_a4_4)	7 s	12.7 s (10.8)	19 (82%)	79% (25%)
Eigenständige Suche (pq_objects_a4_2)	8 s	11.6 s (12.4)	17 (74%)	91% (26%)
Broadcasting (pq_objects_a4_1)	8 s	9.6 s (6.8)	9 (39%)	78% (36%)

Tab. 22: Beurteilung von Modellen zum Routing von Botschaften. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 23 Personen (darunter 21 Schülerinnen und Schülern), die an Workshops mit der PVS teilgenommen haben.

12.6 Schlussfolgerungen

Ziehen wir – mit angemessener Vorsicht – einige Schlussfolgerungen aus den Beobachtungen mit der PVS. Anscheinend spielt das Botschaftskonzept in der Vorstellungswelt von Menschen, die objektorientierte Programme zu verstehen versuchen, keine große Rolle. Veranschaulichende Modelle der PVS wurden von den Spielern vor allem dahingehend beurteilt, ob der Effekt, den die Ausführung der Programmanweisungen auf den Zustand der Objekte hat, in der Visualisierung richtig wieder gegeben wird. Aber der „Weg zum Ziel“, die im Paradigma der OOP festgelegte Mechanik des Entsendens von Botschaften und deren Interpretation durch eigenaktive Objekte, scheint (zumindest für die Spieler der PVS) nur von geringer Bedeutung zu sein. Damit wird die Bedeutung des „objektorientierten Denkens“, das z.B. von Cecile Crutzen (1995) betont wird, etwas relativiert. Man beachte jedoch, dass wir mit der PVS die Konzepte der Objektorientierung lediglich auf der Ebene der Implementierung untersucht haben („Programmieren im Kleinen“). Objektorientierung ist aber auch und vor allem ein Paradigma für die Analyse- und Entwurfsphase einer Software-Entwicklung, also des „Programmierens im Großen“ (Balzert 1999; Sommerville 1997, S. 247 ff). Im Zusammenhang der objektorientierten Modellierung auf hohem Abstraktionsniveau hat die Botschaftsmetapher einen ganz anderen Stellenwert.

13 Intuitive Modellierung

In Kapitel 3 haben wir vier Anwendungen intuitiver Modelle unterschieden: Verstehen, Erklären, Problemlösen und Kontrolle. In allen diesen Kontexten spielt das Ringen um Intuitivität eine Rolle. In diesem Kapitel geht es um die Identifizierung einiger kognitiver Aktivitäten, die bei der intellektuellen Auseinandersetzung mit Programmtexten eine Rolle spielen. Wir bezeichnen als „intuitive Modellierung" den Versuch ein gegebenes formales Programm durch ein Cluster intuitiver Modelle zu repräsentieren. Dabei gehen wir von folgender Annahme aus:

Das Hauptziel bei einer gedanklichen Modellierung im Rahmen der Erklärens oder Verstehens eines Programms ist die Gewinnung von Intuitivität. Bei der Rekonstruktion eines Programmtextes durch eine Kombination intuitiver Modelle werden diese so gewählt, dass die Intuitivität maximal wird[15]. Dabei steht die Suche nach Gewissheit im Vordergrund.

Querliegend zur informatisch-inhaltlichen Dimension werden in den folgenden Abschnitten kognitive Aktivitäten oder Mechanismen beschrieben, die man beim intuitiven Modellieren beobachten kann. Dazu gehören etwa das Identifizieren von Entitäten, Abstraktion, Gestaltbildung, Animismus oder das Bemühen um Konsistenz. Diese Aktivitäten der Intuitionsgewinnung können – je nach Situation – eng zusammenhängen (und quasi gleichzeitig stattfinden) oder aber auch im Konflikt zueinander stehen. So steht z.B. das Streben nach logischer Konsistenz manchmal im Widerspruch zur Gestaltbildung.

Viele der hier beschriebenen Vorgänge kann man als Metaphorisierung betrachten, weil Konzepte einer vertrauten Domäne (Source) auf eine noch unvertraute oder zu erklärende Zieldomäne (Target) angewendet werden. Es gibt aber auch Prozesse, die keine Metaphorisierung sind. So werden bei einer Dramatisierung, d.h. der Einbindung eines Programmfragments in eine sinnvolle Geschichte, Fantasieelemente hinzugefügt, die keine Entsprechung im modellierten Programmtext (Ziel) haben.

In Anlehnung an Piagets Entwicklungsmodell könnte man sagen, dass man sich bei Nachbildung von formalem Programmtext durch intuitive Modelle auf die Denkstufe konkreter Operationen begibt. Nach diSessa ist dies ist kein negativ zu bewertender Rückfall sondern etwas völlig normales, wenn man sich auf noch unvertrautem Wissenterrain bewegt.

Die hier skizzierten kognitiven Prozesse des intuitiven Modellierens haben letztlich als Ziel, das Verständnis des Programmtextes zu verbessern. Insofern sind sie sinnvoll und haben sich zumindest in der Biographie der Personen, bei denen man diese Prozesse beobachten kann, bewährt. Gleichwohl können sie zu Fehlvorstellungen führen bzw. ihre Ergebnisse lassen sich als Fehlvorstellungen interpretieren. Damit sind sie aber noch keine „bug generators", die man generell ausmerzen muss. Es sind normale kognitive Vorgänge, die beim Versuch, die Welt zu verstehen, ihren Stellenwert haben.

13.1 Identifizierung von Entitäten

Bei der intellektuellen Auseinandersetzung mit Programmtexten streben Menschen nach gedanklichen Einheiten, die sie beherrschen können. Sie suchen nach vertrauten Entitäten in einem (zunächst) undurchschaubaren Ganzen, die im Hinblick auf die Gesamtidee des Systems wichtige Rollen spielen. So wird man beispielsweise bei der Analyse eines iterativen Algorithmus vielleicht folgende Entitäten identifizieren:

- Ein Container (z.B. eine Liste) mit mehreren Elementen.
- Das „aktuelle Element", das während eines Durchlaufs der Iteration verarbeitet werden soll.
- Eine Funktion, die das aktuelle Element verarbeitet.

[15] Hier wird der Begriff Intuitivität in einem steigerungsfähigen Sinn gebraucht. Erläuterungen dazu finden sich in Anhang 7.1.

Sajaniemi (2002) hat ein System von „Rollen“ entwickelt, die Variablen in einem Programm spielen können, und gibt für jede Rolle eine Visualisierung an.

Beispiele für Rollen sind

- Konstante, d.h. eine Variable, die nur einen Wert annehmen kann, der niemals geändert wird.
- Stepper: Eine Variable, die mit einem Anfangswert initialisiert wird, und die im Laufe einer Rechnung eine bestimmte Folge von Werten durchlaufen kann. Ein Zähler, der in einer Schleife inkrementiert wird, ist ein Beispiel für einen Stepper.
- Most-wanted-holder: Eine Variable, die den besten bisher gefundenen Wert enthält. Zum Beispiel bei der Suche nach dem Minimum in einer Sequenz braucht man einen Most-wanted-holder für das bisher gefundene kleinste Objekt.

Die Rolle definiert ein Verhaltensmuster und Möglichkeiten der Interaktion mit anderen Entitäten des Programms. Wenn man einer Variablen eine Rolle zuordnet, bindet man sie an Funktionalität. Eine Variable kann (wie ein menschliches Individuum) in verschiedenen Kontexten unterschiedliche Rollen spielen (das ist freilich schlechter Programmierstil), andererseits können unterschiedliche Variablen die gleiche Rolle annehmen.

In fast allen Fällen ist eine Rolle ein intuitives Konzept, das Bestandteil einer Problemlösung ist. Wenn ich einen Algorithmus für die Suche nach dem Minimum in einer unsortierten Sequenz entwickle und auf die Idee komme, einen Most-wanted-holder zu verwenden, habe ich bereits einen Teil der Lösung gefunden. So wie eine Rolle in sozialen System nur Sinn im Kontext komplementärer Rollen hat. Eine Rollenbeschreibung eines Koordinators in einem Programmierteam, der z.B. Aufgaben verteilen und Termine festsetzen soll, definiert implizit auch zu einem großen Teil die Funktionsweise des gesamten Teams.

Sajaniemi hat auch Programm-Beispiele in Anfänger-Büchern nach dem Vorkommen verschiedener Konzepte (Rollen) untersucht. Erstaunlich ist, dass etwa 80 bis 90 % der Variablen den drei oben erwähnten Rollen zuzuordnen sind: Konstante, Stepper und Most-wanted-holder. Sajaniemi verwendet sein Rollensystem für Programmvisualisierungen („role based program animation“). Variablen (und ihre Werte) werden je nach ihrer Rolle durch verschiedene Abbildungen dargestellt. Ein Stepper z.B. wird durch ein Band mit fortlaufenden Nummern repräsentiert, auf dem die aktuelle Nummer eingerahmt ist. Generell sollen die Visualisierungen typische Eigenschaften der Rolle veranschaulichen. Im Fall des Steppers wird die Vorhersehbarkeit der Variableninhalte (Vergangenheit und Zukunft) durch das Nummernband verdeutlicht. Hinter diesem Visualisierungsansatz steht eine „Verstehensstrategie“: Untersuche zunächst die Rollen der Variablen um die Idee des Programms zu verstehen.

13.2 Abstrahieren

Abstraktion (lat. abstrahere: abziehen, entfernen) ist der Gegenbegriff zu Konkretisierung und bezeichnet in der Philosophie das Absehen von bestimmten Aspekten konkreter Entitäten oder Prozesse in der Welt. Automatische Visualisierungsversuche – z.B. mit Sajaniemis Rollenansatz – scheitern daran, dass sie *jede* im System vorkommende Entität darstellen, was zu komplexen und schwer verständlichen Mechaniken führt. [16]Es fehlt die Unterscheidung von Wichtigem und Unwichtigem. Ein intuitives Modell eines Programmtextes muss einfach sein. Intuitive Modellierung beinhaltet deshalb auch Abstraktion im Sinne des Absehens von Unwichtigem:

- Bezeichner (z.B. Variablennamen) aus dem Programmtext werden weggelassen.
- Entitäten werden nur implizit angedeutet und nicht explizit dargestellt (z.B. Blitz für die Ausführung einer Funktion).
- Daten werden durch figürliche Platzhalter ersetzt. Sie sind einfacher und heben nur die für den Algorithmus wichtigen Aspekte der repräsentierten Daten hervor. So kann ein Papierstreifen eine Liste von Daten repräsentieren, wenn es nur auf die Länge der Liste ankommt.

[16] Wie in Abschnitt 6.6 gezeigt wurde werden abstraktere Visualisierungen einer Iteration, die auf eine Explizierung der Laufvariablen verzichten, gegenüber denen mit expliziter Laufvariablen bevorzugt.

- Umfangreiche Daten (z.B. Listen) werden unter Verwendung von Ellipsen (Auslassungen) dargestellt (Abb. 61).[17]
- Determinismus wird durch Nichtdeterminismus ersetzt. Beispielsweise kann man manchmal die exakte Sequenzialität von Ereignissen vernachlässigen und Aktionen, die nach dem Programmtext eigentlich nacheinander erfolgen müssten, als nebenläufige Prozesse darstellen (vgl. Abschnitt 9.2).

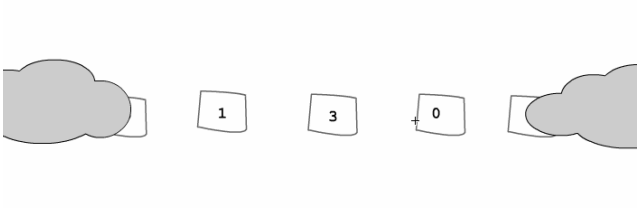


Abb. 61: Verwendung von Auslassungen bei der Darstellung einer langen Liste

13.3 Gestaltbildung

Ein intuitives Modell ist eine Gestalt, ein einfaches in sich geschlossenes Ganzes. Die gedankliche Nachbildung eines Programms bei der intuitiven Modellierung schließt deshalb auch das Streben nach einer Gestalt ein. Damit verbunden ist oftmals auch das Weglassen von Details, dennoch ist Gestaltbildung nicht das gleiche wie Abstraktion. Ein wichtiger Aspekt einer Intuition als Gestalt ist die gedankliche Vertrautheit. Häufig verwendet man Analogien aus dem Alltag, z.B. Gegenstände oder Arrangements von Gegenständen, deren Funktionsweise oder Gebrauchsmöglichkeiten wohl bekannt sind. Eine Analogie für ein Array ist z.B. ein Kasten mit mehreren Fächern, in den sich beschriftete Zettel befinden. Mit diesem materiellen Arrangement sind eine Reihe von möglichen Operationen verbunden: Man kann Zettel nacheinander herausnehmen, sie lesen oder neu beschreiben etc. Alles zusammen ergibt eine kohärente Gestalt. Das Modell des Kastens ist komplexer und besitzt einen geringeren Abstraktionsgrad als z.B. ein Arrangement von nebeneinander liegenden Zetteln. Es hat aber eher Gestaltcharakter.

Um eine Gestalt „abzurunden" kann ein intuitives Modell auch strukturelle Elemente enthalten, die im Zielprogramm nicht explizit vorkommen. In Abschnitt 9.4 wurde z.B. eine Visualisierung des In-Place-Sortierens durch direkte Auswahl beschrieben, bei der ein Glaskasten den bereits sortierten Teil der Liste bedeckt. Damit wurde für die Hauptidee dieses Sortierverfahrens – der erste Teil der Liste ist fertig und wird nicht mehr verändert – eine holistische Vorstellung gefunden, die im Programmtext nicht einmal anklingt.

13.4 Animieren

Gestaltbildung kann auch mit Animismus verbunden sein. Animismus ist die Vorstellung, dass alle Dinge beseelt (lat. animus: Seele) und lebendig sind. Sie haben einen Willen und können sich ohne Außensteuerung aus sich heraus verhalten. Nach Piaget ist Animismus eine typische Denkweise von Kindern in der präoperationalen Stufe (z.B. Flavell 1963).

In der PVS ist eine häufig vorkommende Animation ein fliegender Zettel, der selbst seinen Weg findet. Die Eigenaktivität des Objektes macht die Vorstellung eines steuernden Mechanismus überflüssig und vereinfacht das Modell.

Animismus ist ein Feature des objektorientierten Programmierparadigmas. Ein Objekt im Sinne der OOP ist eine Entität, in der Daten und Operationen auf diesen Daten zu einer Gestalt zusammengefasst sind. Das Objekt verwaltet seine Daten selbst und interagiert eigenständig mit seiner Umgebung. Es

[17] In Texten werden Auslassungen unwichtiger oder nahe liegender Passagen durch Ellipsen dargestellt. Das sind im Deutschen, Englischen und in den meisten anderen Sprachen drei aufeinander folgende Punkte.
Beispiele: „Montag, Dienstag, …, Sonntag", A = {1, …, 10}. Die Ellipse ist also eine Möglichkeit, einen Text zu kürzen ohne seine Aussagekraft zu verringern.

gibt – zumindest auf konzeptioneller Ebene – keine übergeordnete Verwaltung. Dies kann zu einfacheren und deshalb intuitiveren Modellen führen.

Objektorientiert kann man sich den Algorithmus „Sortieren durch Vertauschen“ („Bubblesort“) folgendermaßen vorstellen: Wir haben eine Liste von Objekten. Jedes Objekt prüft, ob es größer als sein linker Nachbar ist. Wenn das nicht der Fall ist, tauscht es mit ihm den Platz. Dies macht das Objekt so lange, bis keine Vertauschungen mehr notwendig sind. Wenn sich alle Objekte der Liste so verhalten, ist sie irgendwann sortiert. Die Idee des gesamten Algorithmus kommt in der relativ einfachen Gestalt eines einzelnen Objektes zum Ausdruck.

Ein Spezialfall von Animismus ist Anthropomorphismus, d.h. die Übertragung spezifisch menschlicher Merkmale auf Elemente des Zielbereichs. Die Verwendung von Anthropomorphismen in wissenschaftlichen Erklärungen wird in der Didaktik kontrovers diskutiert (vgl. z.B. Kattmann 2005). In der Anthropologie existiert die Vermutung, dass die Fähigkeit, soziale Ereignisse (in irgendeiner frühen Form) sprachlich zu beschreiben und auf andere Phänomene anzuwenden, biologisch determiniert ist. Mithen und Boyer (1996) sind der Auffassung, dass die Fähigkeit zu anthropomorphem Denken etwa vor 100 000 bis 40 000 Jahren sich gegenüber anderen frühen Menschformen (z.B. Neandertaler) als evolutionärer Vorteil herausgestellt hat. Erste archäologische Hinweise auf anthropomorphes Denken sind z.B. Abbildungen von Menschen mit Tierköpfen aus der Zeit vor 30 000 Jahren.

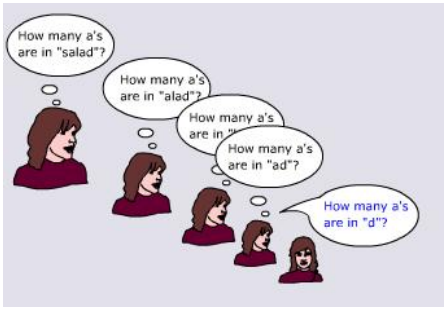


Abb. 62: Anthopomorphes Modell aus der PVS für die Ausführung einer rekursiven Funktion

Der informatische Fachjargon ist voller anthropomorpher Redewendungen, die zwischenmenschliche Interaktionen wiedergeben: Eine Funktion „übernimmt Argumente“ und „gibt etwas zurück“, ein Prozess „schläft“ oder wird „geweckt“. Da Menschen erheblich komplexer sind als die Komponenten einfacher Computerprogramme, bedeutet derartige Modellierung durch Anthropomorphismen keine strukturelle Vereinfachung. Der Vorteil von anthropomorphen Modellen liegt vermutlich darin, dass sie aus einer sehr vertrauten (und für das Überleben wichtigen) Erfahrungssphäre stammen und mit einem hohen Maß an Gewissheit verknüpft sind. Jeder hat eine ziemlich präzise Vorstellung, was es heißt zu schlafen, obwohl dies – bei genauerer Betrachtung – ein relativ komplexes Phänomen ist (man lebt noch, aber man ist inaktiv, bis man wieder aufwacht). Abb. 62 zeigt einen Screenshot aus einer Animation der PVS, in der die Ausführung einer rekursiven Funktion auf anthropomorphe Weise durch soziale Ereignisse, nämlich als Wechselspiel von Fragen und Antworten, modelliert wird.

13.5 Clusterbildung und Fokussierung

Intuitive Modelle sind einfach. Das hat zur Folge, dass ein Modell das Original nur unvollständig abbildet. Dieses Problem der Begrenztheit von Intuition löst man dadurch, dass man ein Cluster von Modellen verwendet. Insbesondere wenn es darum geht, (anderen) ein Programm zu erklären, fokussieren die verwendeten Modelle auf bestimmte Aspekte. Das Verständnis des Gesamten ergibt sich gewissermaßen aus der Summe der herangezogenen Intuitionen.

Die folgende Abbildung zeigt einen Cluster von Intuitionen zur Repräsentation einer Liste.

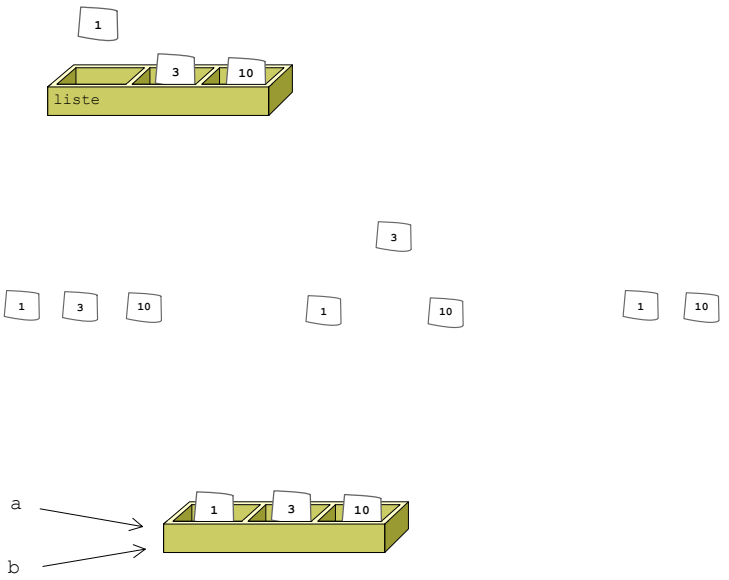


Abb. 63: Cluster intuitiver Modelle für eine Liste

Oben wird die Liste als Behälter, dargestellt, der Zettel enthält. Die Zettel kann man herausnehmen oder ersetzen. Dieses Modell fokussiert folgende Aspekte einer Liste:

- Behälter: Die Liste enthält Elemente (Items)
- Entität: Die Liste ist eine eigene Entität, sie ist mehr als die enthaltenen Elemente
- Iterierbarkeit: Man kann die Elemente nacheinander herausnehmen, um sie zu verarbeiten
- Direkter Zugriff: Man kann ein Element über das Fach, in dem es sich befindet, erreichen.

In der Mitte wird eine Liste durch eine lineare Anordnung der enthaltenen Elemente dargestellt. Deren Zusammengehörigkeit wird allein durch räumliche Nähe dargestellt. Hier liegt der Fokus auf:

- Lückenlosigkeit: Eine Liste hat keine Leerstellen. Wenn ich ein Element entferne, wird die entstehende Lücke sofort geschlossen.
- Dynamische Länge: Die Länge einer Liste kann sich während der Verarbeitung ändern.

Die untere Darstellung betont, dass (bei Python) die Liste ein änderbares Objekt ist. Für ein und dasselbe Objekt können mehrere Namen existieren. Man beachte, dass die Modelle eines Clusters in logischem Widerspruch zueinander stehen können. So kann in der Behälterdarstellung der Liste sehr wohl eine Lücke (leeres Fach) entstehen.

Clusterbildung tritt manchmal „blitzschnell“ und kaum wahrnehmbar auf. Ein Spezialfall von Clusterbildung ist die Verwendung von Kontrollmodellen. Wenn man z.B. versucht sich die Arbeitsweise eines Sortierprogramms klar zu machen, zieht man in Gedanken viele Intuitionen heran und prüft sie wechselseitig auf Übereinstimmung und Widersprüchlichkeit:

- Die sortierte Liste muss genauso lang sein wie die unsortierte Liste.
- Das erste Element einer aufsteigend sortierten Liste ist immer das kleinste.
- Eine Liste mit einem Element ist immer sortiert.

Die Intuitionen eines Clusters brauchen nicht konsistent zu sein und sind es häufig auch nicht. DiSessa verwendet den Begriff „scattered knowledge“ zur Beschreibung von Wissenstrukturen aus locker verbundenen Ideen. Sein Ansatz des „knowledge in pieces“ (1988) ist gewissermaßen eine Gegenposition zu Theorien, die von einer konsistenten in sich widerspruchsfreien kognitiven Struktur ausgehen. Die Animationen der PVS repräsentieren in der Regel nicht einzelne geschlossene Intuitionen sondern jeweils Cluster. Für ein einzelnes intuitives Modell sind sie häufig zu komplex. In ein und derselben Animation können z.B. unterschiedliche Modelle für die Benennung von Objekten vorkommen.

13.6 Überstrukturierung

Überstrukturierung liegt vor, wenn ein gegebenes System feiner strukturiert wird, als es eigentlich notwendig oder zulässig ist. Durch Überstrukturierung können z.B. logische Implikationen atomarer Programmelemente verdeutlicht werden.

So kann eine Animation mit einem Behältermodell für Variablen den atomaren Vorgang einer Zuweisung der Art `x = 2` in die Zerstörung des vorigen Inhalt und das Einfüllen eines neuen Inhalts aufteilen. Mit der visuellen Explizierung der Zerstörung (z.B. durch eine Explosion) wird hervorgehoben, dass jede Zuweisung den Verlust der vorigen Variablenbelegung impliziert. Dies ist eine rein logische Überlegung, die nichts mit der technischen Realisierung in einem realen Computer zu tun hat[18].

Ein Funktionsaufruf innerhalb einer Zuweisung der Form

```
x = f(y)
```

kann folgendermaßen visualisiert werden. Zuerst wird ein Name `x` generiert, der für ein neues Objekt bestimmt ist („wartender Name"). Dann startet die Funktion f und erzeugt ein Objekt, dem schließlich der Name x zugewiesen wird. Der auf sein Objekt wartende Name ist eigentlich überflüssig. Er wurde hinzu erfunden, um die Rückgabe des Objektes zu verdeutlichen. Er veranschaulicht, dass der aufrufende Prozess darauf wartet, dass die Funktion ein Objekt zurückgibt, für das dieser Name vorgesehen ist.

Die Interpretation von Zuweisungen als Eingabe, die bei Anfängern gelegentlich beobachtet wird, kann als Überstülpen des EVA-Prinzips (also Hinzufügen zusätzlicher Struktur) verstanden werden (eine Schülervisualisierung als Beispiel findet sich in Anhang 7.2).

Manchmal ist es für das Verständnis der Semantik einer Anweisung sinnvoll, wenn man in die Tiefe der Programmiersprache dringt und implizite technische Aspekte, die mit der Ausführung der Anweisung zusammenhängen, expliziert. Hierzu einige Beispiele:

Viele Programmiersprachen verwenden implizites Casting bei Operationen mit Objekten unterschiedlicher Typen. Die Ausführung der Operation `1*2.3` kann man sich z.B. so vorstellen, dass zuerst das Objekt `1` vom Typ `integer` in ein Objekt vom Typ `float` „umgewandelt" wird, bevor die Multiplikation ausgeführt wird.

Bei Python kann man Iterationen über Container-Objekte (z.B. Listen, Mengen oder Dictionaries), der Form

```
for i in container:
    Anweisungsfolge
```

besser verstehen, wenn man sich klar macht, dass bei der Ausführung „hinter den Kulissen" ein Iterator ins Leben gerufen wird, der bei jedem Durchlauf das nächste Element des Containers heraussucht und eine Ausnahme auslöst (und damit das Ende der Iteration bewirkt), wenn alle Items besucht worden sind. Dieser Iterator ist implizit, das heißt, er tritt im Programmtext nicht in Erscheinung, aber er könnte bei einer erklärenden Animation visualisiert werden.

13.7 Einbeziehung der Umgebung

Gelegentlich werden nicht sichtbare Teile der Umgebung des zu erklärenden oder zu verstehenden Programmtextes in die intuitive Modellierung einbezogen. Zur Umgebung eines Programms gehören z.B. das Betriebssystem, Compiler und Interpreter. Betrachten wir folgendes Python-Programm, das ein Fenster mit zwei Labeln darstellt.

```
from Tkinter import *
fenster = Tk()
label1 = Label(fenster, text="Oben")
```

[18] Beispielsweise beim Überschreiben einer Speicherzelle im Arbeitsspeicher, wird diese *nicht* zuerst gelöscht und bleibt dann für einen kurzen Moment „leer", bevor der neue Inhalt kommt.

```
label2 = Label(fenster, text="Unten")
label1.pack()
label2.pack()
fenster.mainloop()
```

Formal ist die Methode `pack()` eine Operation der Klasse `Label`. In einer strukturtreuen Erklärung würde man sagen, dass das Label-Objekt *sich selbst* in bestimmter Weise im Anwendungsfenster positioniert. Doch bei Erklärungen eines solchen Programms (z.B. in Sprachreferenzen) wird meist das Konzept des Layout-Managers herangezogen. Man sagt z.B., dass durch den Methodenaufruf `pack()` der Packer – einer von mehreren verfügbaren Layout-Managern – aktiviert wird, der in diesem Fall dafür sorgt, dass die beiden Labels untereinander stehen. Mit dem Packer wird ein Akteur herangezogen, der im Programmtext nicht sichtbar ist. Ein Layout-Manager ist Teil der Systemumgebung, die zur Laufzeit die geometrische Anordnung von Elementen der grafischen Oberfläche des Programms auf dem Bildschirm verwaltet. Die Einbeziehung der Umgebung kann auch als Möglichkeit der Gestaltbildung gesehen werden. Ein unvollständiges Stück wird durch die Hinzunahme eines Teils zu einem abgerundeten und leichter vorstellbaren Ganzen.

13.8 Dekorieren und Dramatisieren

Als Dekorierungen bezeichnen wir gestalterische Elemente, die nicht in dem abzubildenden Programmtext aber auch nicht in seiner technischen Umgebung vorkommen. Solche kreativen Add-ons können den Charakter von Kommentaren haben, die die Verständlichkeit durch Hinzunahme zusätzlicher Information erhöhen. Sie können sogar dazu dienen aus einem Programmtext eine Geschichte zu machen (Dramatisierung). In der Visualisierung in Abb. 64 unterhalten sich zwei Variablen und erklären dadurch ihren Zustand nach einer Zuweisung.

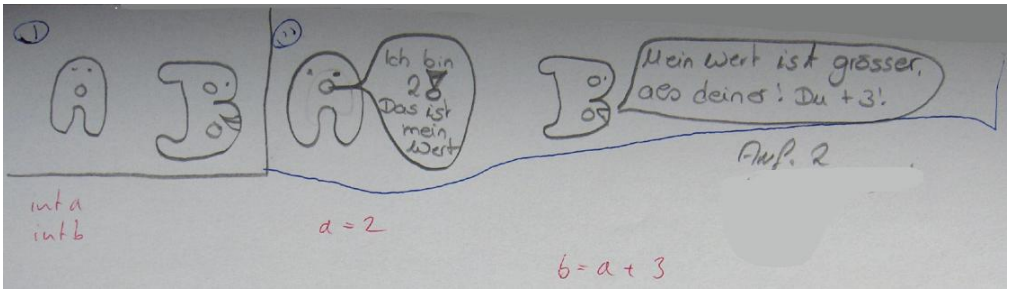


Abb. 64: Dramatisierung einer Zuweisung. Visualisierung einer 17-jährigen Schülerin

Im Unterschied zur „Einbeziehung der Umgebung“ haben die hinzugefügten Elemente nichts mit dem Kontext des Programmfragments in der Zieldomäne zu tun. Die technische Umgebung, in der das Programm läuft, das in Abb. 64 visualisiert wird, hat z.B. keinen Mechanismus, über den eine Variable über ihre Belegung berichten könnte.

13.9 Rückmodellierung

Unter Rückmodellierung verstehen wir den Versuch, zu einem Programmtext (oder Programmtextfragment) einen Wirklichkeitsausschnitt oder eine praktische Problemstellung zu finden, für die das Programm ein Modell oder eine Lösung darstellt. Abb. 66 zeigt eine Visualisierung der Java-Anweisungsfolge

```
String wort;
wort = "Hallo";
```

Hier hat der Schüler mit viel Fantasie eine Begrüßungssituation erfunden, in der ein Computer auf dem Monitor ein Gesicht zeigt und über Lautsprecher den Gruß „Hallo“ ausgibt. Das Programm, das eine solche Aufgabe bewältigt, enthält sicherlich die beiden obigen Anweisungen – aber noch viele mehr.

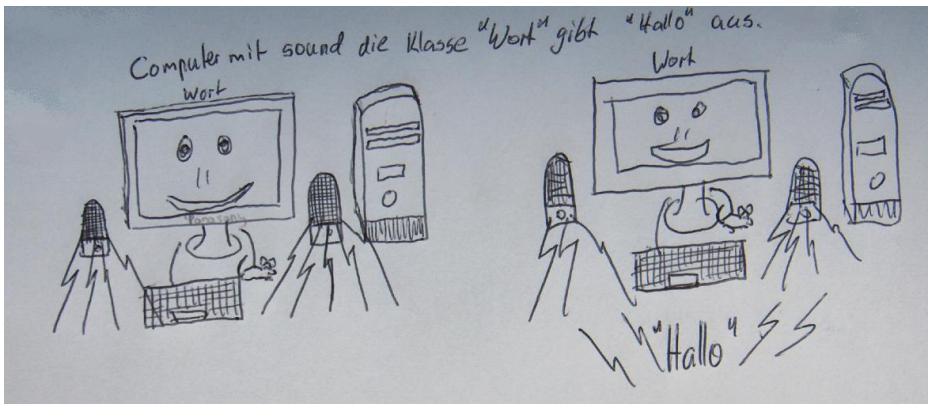


Abb. 65: Visualisierung als Rückmodellierung (Zeichnung eines 17-jährigen Schülers der Jahrgangsstufe 11)

Rückmodellierung kann dem Dramatisieren und Dekorieren ähneln, wenn Erfundenes hinzugefügt wird. Die hinzugefügten Elemente entstammen jedoch einem Realitätsausschnitt, der vom Programm modelliert wird. Dabei wird angenommen, dass in dem Programm (wie bei jeder Modellierung) gewisse Aspekte der Realität weggelassen worden sind. Diese werden beim Rückmodellieren wieder hinzugefügt. Als Motiv für Rückmodellierung kann man in erster Linie Sinnschaffung annehmen. Nackter Programmtext wird als sinnvolles Modell gedeutet.

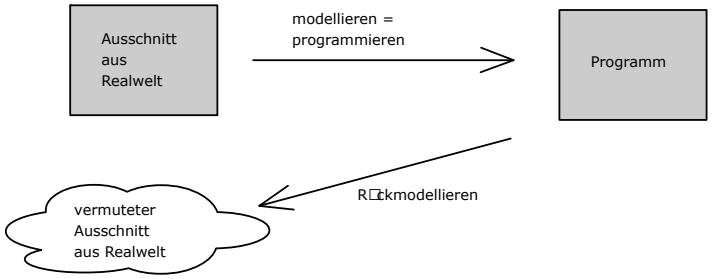


Abb. 66: Rückmodellieren

Rückmodellierung heißt, zu einem gegebenen Programm die Funktionalität herauszufinden. Wozu dient das Programm? Was leistet es? Bei diesen Fragen verwendet man den Funktionsbegriff der Systemtheorie, betrachtet das Programm als System und sucht nach dem Nutzen im Hinblick auf Ziele, die selbst außerhalb des Systems liegen. Wer die Funktion (im Sinne eines Zwecks) eines gegebenen Programms beschreibt, muss deshalb kreativ werden und einen Kontext erfinden, innerhalb dessen es sinnvoll verwendet werden könnte.

Nach der Theorie von Pennington (1987) ist die Suche nach Funktionalität (neben der Auseinandersetzung mit Kontrollmechanismen, Datenflüssen und den Zuständen, die bei der Ausführung des Programms durchlaufen werden) einer von mehreren kognitiven Prozessen („Abstraktionen"), die zum Verstehen eines Programmtextes führen. Beim funktionalen Verstehen werden Programmstücke Zielen zugeordnet. Es entsteht ein Baum mit dem globalen Ziel des gesamten Programms als Wurzel und darunter immer weiter ausdifferenzierten Teilzielen.

Nach den Ergebnissen von Davies (2000) ist das schnelle Erkennen von Funktionalität mit Expertise verbunden. In einem Experiment zeigte er 16 Anfängern und 16 Programmierexperten für kurze Zeit (entweder 2 Sekunden oder 10 Sekunden lang) kleine C++-Programme und stellte anschließend Verständnisfragen. Im Fall der Zwei-Sekunden- Präsentationen schienen die Experten einen Schwerpunkt auf die Erkennung der Funktionalität zu legen. Sie konnten signifikant mehr Fragen zur Funktionalität (54%) als zu Kontroll- und Datenfluss (26% bzw. 23%) beantworten. Novizen dagegen hatten eher Schwierigkeiten.

13.10 Exhaurierung

In der Wissenschaftstheorie versteht man unter Exhaustion folgendes: Wenn sich mit einer Hypothese Beobachtungen nicht erklären lassen, verwirft man sie nicht sofort, sondern „schöpft sie aus"

(exhauriert sie), indem man versucht mit geeigneten Zusatzannahmen die widersprechenden Daten hypothesengemäß zu deuten (Dingler 1913). Ein bekanntes Beispiel für die Anwendung dieses Verfahrens stammt aus der Frühzeit der Chemie. Der deutsche Alchemist Georg Ernst Stahl stellte zu Beginn des 18. Jahrhunderts die Phlogistonhypothese zur Erklärung von Verbrennungsvorgängen auf (Ströker 1967, S. 115 ff.). Danach ist die Verbrennung ein Vorgang, bei dem der Brennstoff einen „Feuerstoff" abgibt, den Stahl Phlogiston nannte. Nach dieser Hypothese mussten alle brennbaren Stoffe Phlogiston enthalten. Nun konnte man aber beobachten, dass die Verbrennungsprodukte (Oxide) schwerer waren als die Ausgangsstoffe der Verbrennung. Die Exhaustion der Phlogistonhypothese bestand darin, dass Stahl ihr ein negatives Gewicht zuschrieb. Wie das Beispiel zeigt, kann Exhaurierung zu einer Komplizierung des Erklärungsmodells führen.

Bei der intuitiven Modellierung von Programmtexten werden Verfahren angewendet, die der Exhaustion entsprechen. Ein Beispiel ist die Exhaurierung des Behältermodells, das Zuweisungen der Form `a = b` als Kopieren von Daten interpretiert. Wenn man dieses Modell auf änderbare Objekte (z.B. Python-Listen) anwendet, so steht man vor folgendem Problem: Eine Änderung des Objektes in Behälter `a` wirkt sich gleichzeitig auf das Objekt in Behälter `b` aus. Eine Exhaurierung kann nun darin bestehen, dass man das Objekt in Behälter `a` nicht als eigenständige Entität sondern als „Geist", als zweite Erscheinung des Objektes in Behälter `b` betrachtet. Eine Veränderung des Objektes in Behälter b wird dann auch in seinem Geist sichtbar.

Eine Funktion kann man sich als geschlossene Maschine mit Ein- und Ausgang vorstellen. Die Maschine nimmt über den Eingang Daten auf, verarbeitet sie und gibt über den Ausgang das Ergebnis aus. Probleme gibt es mit diesem Modell, wenn die Funktion andere Funktionen aufruft. Eine mögliche Exhaurierung ist die Annahme, dass die Funktionsbox eine Seitentür besitzt, über die sie Daten an andere Funktionen senden und die Ergebnisse empfangen kann (vgl. Abschnitt 8.8.2).

13.11 Konsistenzwahrung

Für die Auswahl der Modelle eines Clusters zur Erklärung eines Programmtextes kann deren Konsistenz ein Kriterium sein. Konsistent sind z.B. das Behältermodell für Variablen und das Modell einer Funktion als Box, die über ihren Eingang Daten aufnimmt, sie verarbeitet und über einen Ausgang das Ergebnis zurückgibt. Ein Cluster konsistenter intuitiver Modelle ist gedanklich einfacher zu handhaben als ein gleich großes nichtkonsistentes Cluster. Denn wenn es Widersprüche enthält, muss man sich jeden Widerspruch vergegenwärtigen oder bei Erklärungen seinen Kommunikationspartnern mitteilen. Wer bei Erklärungen konsistente Intuitionen verwendet, versucht „im Bild zu bleiben". D.h. sie oder er sucht nach Metaphern, die dieselbe Quelldomäne besitzen. In der Mathematikdidaktik gibt es regelrechte Systeme konsistenter Metaphern, die als Lernhilfen für Unterrichtsreihen eingesetzt werden (vgl. z.B. Lakoff & Núnez 1997).

Konsistenz wird andererseits oft durch höhere Komplexität erkauft. So kann man Programme einer objektorientierten Programmiersprache strikt objektorientiert deuten und sich alle Objekte als Akteure vorstellen, die Botschaften senden und entgegen nehmen können. Die arithmetische Operation

```
2*(b + 1)
```

muss man dann ungefähr so beschreiben: „Das Objekt, das die Zahl 2 repräsentiert, erhält die Botschaft, eine Multiplikation mit einer Zahl durchzuführen, die durch ein anderes (anonymes) Objekt repräsentiert wird. Dieses Objekt wird folgendermaßen ermittelt: Das Integer-Objekt mit dem Namen b erhält die Botschaft zu seinem Wert die Zahl 1 zu addieren. Zurückgegeben wird ein Objekt, das die Summe repräsentiert". Das ist ein sehr komplexer Gedanke. Cluster von intuitiven Modellen zur Erklärung eines Programmtextes sind deshalb häufig inkonsistent, um einfach zu bleiben. Ein großer Teil der intuitiven Modellierkunst besteht darin, Inkonsistenzen zu handhaben. Dazu gehört, sich ihrer bewusst zu sein, um Fehlvorstellungen zu vermeiden.

14 Von der Intuition zum Programm

Im Zusammenhang mit Programmierung kann man zwischen zwei Typen von Problemlösungen unterscheiden:

- Bei Problemlösungen erster Art ist allein die Problemstellung bekannt. Das Subjekt verwendet einen großen Teil seiner Aktivität darauf, zunächst eine unformale intuitive Lösung zu finden, von deren Korrektheit er oder sie überzeugt ist. Im zweiten Schritt wird die intuitive Lösung zu einem lauffähigen Programm formalisiert.
- Bei einer Problemlösung zweiter Art ist die intuitive Lösung bereits bestens bekannt. Die Hauptaktivität liegt in der Programmierung. Das heißt, mit bekannten oder zu recherchierenden syntaktischen Mitteln der Programmiersprache wird das intuitive Modell nachgebildet und ein lauffähiges und korrektes Programm generiert.

Beispiele für Probleme, die zu Problemlösungen zweiter Art führen, sind:

- „Nimm alle roten Gummibärchen aus der Tüte."
- „Suche im Telefonbuch die Nummer von Monika."
- „Sortiere Karten mit Namen und Telefonnummern in einem Karteikasten aufsteigend nach den Nachnamen."

Im Schulalltag spielen Problemlösungen der zweiten Art die größere Rolle. Man kann davon ausgehen, dass Schüler Aufgaben wie in den obigen Beispielen intuitiv lösen können. Aber selbst Schüler, die die notwendigen Elemente der Programmiersprache kennen und über Programmiererfahrung verfügen, haben Schwierigkeiten ein entsprechendes Programm zu formulieren und scheitern häufig sogar. Auf dem Weg von der Intuition zum Programm gibt es einige kognitive Barrieren, die zu überwinden sind (vgl. Weigend 2006d).

14.1 Prozedurale Intuition und fehlendes deklaratives Wissen

Im schlimmsten Fall kann die Vorgehensweise bei der intuitiven Lösung vollkommen unbewusst sein (prozedurale Intuition, vgl. Abschnitt 1.2). Zur Überwindung dieser Barriere muss der Programmentwickler sich selbst bei der intuitiven Problemlösung beobachten oder sie sich in Gedanken vergegenwärtigen, um zu deklarativem Wissen über eine Vorgehensweise (einem antizipatorischen oder paradigmatischen Modell) zu gelangen.[19]

14.2 Aufbrechen der Gestalt

Eine zweite Hürde ist das Aufbrechen der Gestalt eines paradigmatischen oder antizipatorischem Modells („Lösungsidee"). Im Modell des GAP-Trees von Spohrer, Soloway und Pope (1989) muss ein gestaltartiger Plan in Ziele aufgeteilt werden (siehe Abschnitt 3.3.2). Nachdem das erste Stück herausgelöst ist, hat aber der Gesamtplan seine Intuitivität verloren. Man ist sich zwar sicher, dass der Plan irgendwie realisierbar ist, aber bei der Festlegung der Ziele wird man spezifischer und unsicher, ob die aufgelisteten Ziele in ihrer Summe den übergeordneten Plan wirklich vollständig beschreiben. Vielleicht hat man ein Detail vergessen. Bonar und Solway (1985) beobachteten bei programmierenden Schülern Explizierungslücken. Sie verwendeten bei Programmierübungen die Programmiersprache als wäre sie natürliche Sprache. Ein Beispiel ist die Verwendung einer Anweisung der Art

```
sum = sum + i
```

in einer `for`-Schleife als Realisierung der umgangssprachlichen Anweisung „Addiere den nächsten Wert zur Summe". Dabei wurde unterschlagen, dass der „nächste Wert" erst noch gelesen werden

[19] Moser (2003) beschreibt Workshops, in denen über eine Analyse von Metaphern (wie z.B. „Wissen als Bibliothek" oder „Wissen als umkämpfter Schatz") in Gruppendiskussionen implizites (unbewusstes) Wissen der Teilnehmer über eine Domäne expliziert und bewusst gemacht wird.

muss. Die Variable `i` wird einfach als Behälter für den nächsten Wert gesehen. Die umgangssprachliche Darstellung „verkürzt“ das Procedere um das Einlesen.

Werfen wir einen genaueren Blick auf die kognitiven Prozesse bei der Analyse einer antizipatorischen Lösungsidee. Eine wesentliche Aktivität ist dabei die Suche nach Elementen, die man mit programmiersprachlichen Mitteln beschreiben kann. Ein einzelner Punkt dabei ist die Identifikation und Benennung von Entitäten. Das Problem ist, dass das intuitive Modell als Gestalt einfach ist und auf explizite Benennung der enthaltenen Entitäten verzichtet. Nehmen wir als Beispiel die Suche nach einer Telefonnummer in einer Liste, die auf ein Blatt Papier geschrieben ist. Jede Zeile enthält ein Paar im Format (*Name*, *Telefonnummer*). Um nach der Nummer zu suchen, geht man mit dem Finger von oben nach unten die Namensspalte hinunter und kontrolliert in jeder Zeile ob der aufgeführte Name mit dem gesuchten übereinstimmt. Falls dies der Fall ist, geht man mit dem Finger nach rechts und liest die zugehörige Telefonnummer. Eine Schwierigkeit liegt darin zu erkennen, dass das Zeigen mit dem Finger eine Benennung ist. Für die Zeile, auf die der Finger zeigt, müsste im Programmtext ein Name eingeführt werden, über den der Zugriff auf das Paar (*Name*, *Telefonnummer*) ermöglicht wird.

14.3 Fehlende Verbindungen zwischen intuitiven Vorstellungen und Programmkonstrukten

Das letzte Beispiel verweist bereits auf eine dritte Barriere, nämlich fehlende Verbindungen zwischen intuitiven Vorstellungen und Programmierkonzepten. Es kann z.B. sein, dass jemand auf der einen Seite Alltagsaktivitäten wie das Zeigen auf ein Objekt oder das Aufschlagen von Seite 20 in einem Buch beherrscht und auf der anderen (programmtechnischen) Seite Zuweisungen und den Zugriff auf ein Item eines Arrays kennt, aber diese Konzepte aus zwei verschiedenen Domänen nicht verbinden kann. Dieser Person kommt dann nicht in den Sinn, dass das Zeigen auf ein Objekt einer Benennung bzw. einer Zuweisung der Art `x = objekt` entspricht.

Diese Art von Wissen wird durch Programmiererfahrung erworben, d.h. durch viele Programmierexperimente, in denen intuitive Modelle zu Programmtext in Beziehung gesetzt werden (vgl. auch Weigend 2006c). Perkins et al. (1989) beobachteten, dass erfolgreiche Programmentwickler die Fähigkeit besitzen, genau zu erklären bzw. nachzuvollziehen, was jede Zeile eines Programmtextes bewirkt. Sie nennen das „close tracking of code“. Sie untersuchten auch, welche Arten von Unterstützungsmaßnahmen helfen, Blockaden im Problemlösungsprozess zu überwinden. Als besonders wirksam stellte sich heraus, Schüler, die nicht weiterkommen, aufzufordern, Zeile für Zeile zu erklären, was ihr (bisher geschriebener) Programmtext leistet. Dabei stießen sie von allein auf ihre Probleme und fanden Ansätze, ihren Programmtext weiterzuentwickeln. Python Puzzles der PVS enthalten als Hilfemechanismus Animationen, die die Arbeitsweise einer zu definierenden Funktion darstellen. Die Nutzung dieser visuellen Modelle kann eine besonders intensive Art des „close trackings“ sein, nämlich dann wenn man die einzelnen Entitäten und Aktivitäten mit Elementen des Programmtextes in Beziehung setzt. [20]

14.4 Schwierige intuitive Modelle

Eine vierte Barriere bei Problemlösungen sind schwierige intuitive Modelle. Eine Lösungsidee kann grundsätzlich für eine direkte Implementierung ungeeignet sein. Genauer gesagt, sie kann Aspekte enthalten, zu deren Implementierung der Person die programmiersprachlichen Konzepte fehlen. Als Beispiel betrachten wir einen Ansatz zur Lösung des Sortierproblems. Als Einstieg in einer Unterrichtsreihe könnten Schüler die Aufgabe erhalten, sich der Größe nach in einer Reihe an der Wand aufzustellen. Eine intuitive Lösungsgestalt (Bubblesort) ist:

„Jeder sorgt dafür, dass links neben ihm niemand steht, der kleiner ist.“

[20] Fehlende Verbindungen von Programmkonstrukten und intuitiven Modellen in Problemlösungssituationen sollten eigentlich mit der PVS untersucht werden. Manche Python Puzzles enthalten als Hilfesystem eine Reihe von visuellen Modellen, die auf die Prüfung (und Schaffung) solcher Assoziationen abzielen. Leider gab es in den bisherigen Workshops mit der PVS zu wenige Sitzungen mit Python Puzzles und darin eine zu geringe Nutzung der Hinweise um zu dieser Thematik irgendwelche Erkenntnisse zu gewinnen.

Es ist unmittelbar einleuchtend, dass eine Gruppe von Personen auf diese Weise nach der Körpergröße sortiert werden kann. Angenommen, jemand verwendet diese Lösungsidee als paradigmatisches Modell für die Sortierung einer Liste von Zahlen, so stieße er oder sie auf massive Probleme. Jede Zahl müsste durch einen eigenen Prozess repräsentiert werden, der mit den anderen Prozessen kommuniziert. Es müssten relativ komplexe Protokolle erarbeitet und obendrein beachtet werden, dass das System nicht in Deadlocks oder Endloswiederholungen gerät. Die intuitive Lösungsidee ist also (zumindest für Anfänger) für eine direkte Umsetzung ungeeignet. Die Kunst liegt darin, das zu erkennen und entweder nach einem völlig neuen Ansatz zu suchen oder aber die Lösungsgestalt als abstraktes (das heißt von einer Implementierung weit entferntes) Modell zu begreifen, die nur auf nichtnaheliegende Weise implementiert werden kann (siehe Abschnitt 15.2).

14.5 Fehlvorstellungen

Fehlvorstellungen über die Bedeutung von Programmtext sind das fünfte und letzte in diesem Abschnitt betrachtete Hindernis auf dem Weg vom intuitiven Modell zum Programmtext. Eine Fehlvorstellung kann man in diesem Zusammenhang als falsche Verbindung zwischen einer intuitiven Vorstellung und einem Programmkonstrukt betrachten.

Verschiedene Mechanismen machen Fehlvorstellungen zur Barriere. Erstens können sie zu heimtückischen semantischen Fehlern im Programmtext führen, da sie nicht durch Inspektion zu finden sind. Fehlersuche durch Inspektion setzt voraus, dass man die Semantik der Programmiersprache vollständig beherrscht. Logische Fehler, die auf Fehlvorstellungen basieren, können nur durch systematisches Eingrenzen in Programmierexperimenten (Tracing, Testen, Experimente mit vereinfachten Programmversionen etc.) gefunden werden. Um dabei erfolgreich zu sein, braucht man ein generelles Misstrauen gegenüber der eigenen Intuition. Man muss die Möglichkeit in Betracht ziehen, dass – gegen die eigene Überzeugung – jedes Programmstück falsch sein kann.

Heimtückischer noch ist der zweite Mechanismus. Wenn man eine falsche Vorstellung über ein Programmkonstrukt hat, kommt man vielleicht gar nicht auf die Idee es zu benutzen, weil man es irrtümlicherweise für untauglich hält.

Interessant ist in diesem Zusammenhang auch das Phänomen, dass Fehlvorstellungen häufig *keine* Behinderung von Problemlösungen und speziell Programmentwicklungen darstellen. Fehlvorstellungen sind ja kognitive Konzepte, die sich in der Bewältigung des Alltags bewährt haben. Jahrtausendelang kam die Menschheit problemlos mit der Fehlvorstellung zurecht, dass sich die Sonne um die Erde dreht. Auch heute verwenden naturwissenschaftlich gebildete Menschen Formulierungen wie „Die Sonne steht hoch am Himmel“ und denken bei der Betrachtung eines Sonnenuntergangs nicht daran, dass sich eigentlich die Erde dreht. Beobachtungen mit der PVS haben gezeigt, dass die meisten beobachteten Informatikschüler/innen für Zuweisungen der Art `a = b` unkorrekte intuitive Modelle verwenden (siehe Abschnitt 10.5). Geht man davon aus, dass diese Personen in der Lage sind, korrekte einfache Programme mit Zuweisungen der beschriebenen Art zu formulieren, so scheinen diese Fehlvorstellungen keine kognitiven Barrieren darzustellen.

15 Pädagogische Implikationen

15.1 Intuitive Modelle und Lernen formaler Programmierkonzepte

Programmiersprachen sind so konzipiert, dass sie bestimmte Denkmuster (Paradigmen) unterstützen. Die Programmierparadigmen korrespondieren mit bestimmten „Leitintuitionen“. Wie bereits an verschiedenen Beispielen erläutert worden ist, *determiniert* die Programmiersprache jedoch *nicht* die gedanklichen Vorstellungen, die man bei einer Programmentwicklung verwendet. Das imperative Programmierparadigma etwa unterstützt die Vorstellung, dass ein Programm aus „Befehlen“ besteht, die von einem einzigen Akteur ausgeführt werden. Eine im Programmtext definierte Funktion wird in diesem Denkmuster als komplexer Befehl gesehen, der nicht im vorgegebenen Repertoire des Akteurs vorrätig ist, sondern zunächst „gelernt“ werden muss. Auf der anderen Seite stellen sich viele Anwender einer imperativen Programmiersprache eine Funktion als eigenen Akteur vor, der durch einen Funktionsaufruf aktiviert wird.

Intuitive Modelle der Informatik können hinsichtlich ihres Abstraktionsgrades oberhalb, auf gleichem Niveau oder unterhalb einer Programmiersprache liegen. Man kann sie sie in Anlehnung an die Redeweise in der Chemie molekulare, atomare und subatomare Modelle nennen.

15.1.1 Molekulare Modelle

Programmierer verwenden intuitive Modelle, die ganze Programme oder Programmabschnitte – also größere logische Einheiten- in einer kohärenten Gestalt zusammenfassen. Ein Beispiel ist das Modell eines Behälters mit mehreren Fächern und springender Markierung zur Darstellung der Verarbeitung einer Liste in einer Iteration (zur psychischen Realität siehe Abschnitte 6.6 und 9.4). Solch ein Modell lässt Details der Implementierung weg und bringt Denken „oberhalb der Programmiersprache“ zum Ausdruck. In didaktischer Hinsicht sind Modelle oberhalb der Programmiersprache z.B. wichtig, um algorithmische fundamentale Ideen der Informatik (Schwill 1993) zu verstehen, zu behalten, zu kommunizieren und als antizipatorische Intuitionen für Problemlösungen zu verwenden. Die Abstraktion, die mit der Gestaltbildung verbunden ist, bringt eine Entfernung von den Konstrukten der Programmiersprache mit sich. Die Kunst (und Schwierigkeit) der Implementierung liegt vor allem im Aufbrechen der Gestalt in Fragmente, denen man Programmkonstrukte zuordnen kann. Beispielsweise muss man wissen, dass eine „springende Markierung auf einem Behälter mit Fächern“ eine Laufvariable in einer Iteration repräsentieren kann.

15.1.2 Atomare Modelle

Betrachten wir als nächstes intuitive Modelle, deren Abstraktheit in etwa auf dem Niveau einer Programmiersprache liegen und die deshalb mehr oder weniger unmittelbar formale Programmkonstrukte abbilden können. Eine Vielfalt von Ausdrucksformen zur Veranschaulichung von Programmkonstrukten kann im Hinblick auf kreatives Problemlösen nützlich sein. Wer weiß, dass man eine Zuweisung der Form `x = 2` auch als Markierung eines Objektes interpretieren kann, dem sollte es leichter fallen, ein abstrakteres intuitives Modell für eine Iteration über eine Liste, das Markierungen verwendet („springende Markierung auf einem Behälter mit Fächern“), „aufzubrechen“. Diese Person kommt dann eher auf die Idee, für die Markierung eine Variable (Name für einen Listenindex) einzuführen.

Atomare Modelle auf gleichem Abstraktionsniveau wie das modellierte Konstrukt sind die Basis von „didaktisch durchdachten“ Quelldomänen für Metaphern, die in der Mathematik bereits eine lange Tradition haben (siehe Abschnitt 2.3.2). Automatische Visualisierungssysteme wie die Jeliot-Familie (Moreno & Myller 2003) verwenden zusammenpassende Sets atomarer Modelle, um die Arbeitsweise von Programmen zu veranschaulichen. Ein Visualisierungssystem ist von Fachdidaktikern designt. Den verwendeten Modellen liegt eine von der Fachgemeinschaft akzeptierte Interpretation von Konstrukten der Bezugs-Programmiersprache zugrunde. Typische Aktivitäten im Klassenraum sind:

- Lernen neuer Sprachelemente: Die Lehrperson demonstriert die Wirkungsweise eines neu eingeführten Programmkonstrukts. Die Ausführung des Programmtextes kann mit Hilfe des Modells nachvollzogen werden.
- Debugging: Wenn ein Programm semantische Fehler enthält, kann eine kritische Passage Schritt für Schritt nachvollzogen werden. Die Funktion atomarer Modelle ist dann, die (ansonsten unsichtbare) Wirkung eines atomaren Programmschritts zu visualisieren und der kritischen Überprüfung (Vergleich mit dem erwarteten Effekt) zugänglich zu machen.

Bei derartigen Aktivitäten ist die 1:1-Beziehung – die direkte Zuordnung von Elementen der Quelldomäne (visuelle Komponenten auf dem Bildschirm) zu Elementen der Zieldomäne (sprachliche Einheiten des Programmtextes) – von Bedeutung.

Automatische Visualisierungsysteme haben jedoch prinzipiell zwei Einschränkungen. Zum einen enthalten sie keine Repräsentationen für typische falsche intuitive Modelle, die in den Köpfen der Lernenden existieren können. Sie liefern naturgemäß nur richtige Interpretationen vor Programmtexten und können nicht die Konsequenzen einer ungeeigneten Intuition vor Augen führen. Zum zweiten kanalisieren sie die Vielfalt möglicher (im gegebenen Kontext ebenfalls akzeptabler) Modelle auf jeweils ein einziges. Dabei ist es erstens fraglich, ob im Hinblick auf den Lernkontext (algorithmische Idee, Vorkenntnisse der Lernenden) die beste Darstellungsform gewählt worden ist. Zweitens können in einem abstrakteren, molekularen Modell für eine algorithmische Idee, die in dem gegebenen Programm implementiert werden soll, verschiedene Typen atomarer Modelle gemischt sein (z.B. Behältermodelle und Referenzierungsmodelle).

Zusammenfassend kann man sagen, dass atomare Modelle vor allem dem Erlernen und Einüben einer Programmiersprache dienen. Die Auswahl geeigneter Modelle für Unterrichtsgespräche ist eine didaktische Entscheidung, die nur schlecht automatisiert werden kann, da viele Faktoren berücksichtigt werden müssen. Die Beherrschung einer Vielfalt atomarer Modelle für Elemente einer Programmiersprache erleichtert die Implementierung einer algorithmischen Idee.

15.1.3 Subatomare Modelle

Für das Verstehen von elementaren Programmkonstrukten ist es oft hilfreich und notwendig mentale Vorstellungen zu verwenden, deren Abstraktionsgrad unterhalb dem Niveau der Programmiersprache liegt. Wir sprechen dann von subatomaren Modellen. In Abschnitt 13.6 wurden sie bereits unter dem Stichwort „Überstrukturierung“ diskutiert.

Wie im Zusammenhang mit Verarbeitungskonzepten diskutiert worden ist (vgl. Kapitel 10) spielen bei der Interpretation elementarer Programmanweisungen häufig mehrere grundlegende Intuitionen, die auch im Alltag verwendet werden, zusammen. Beispielsweise enthalten Modelle für Zuweisungen Vorstellungen über Entstehung, Vernichtung, Transport und Metamorphose von Entitäten. In der Regel sind die Modelle, die man zur Abbildung formaler Programmkonstrukte heranzieht, nicht hundertprozentig passend. Um jedoch die Unterschiede und Grenzen der Anwendbarkeit herauszufinden und zu explizieren, muss man sie genauer analysieren und auf subatomare Komponenten zurückgreifen.

15.2 Kompetenzen im Umgang mit intuitiven Modelle

Die Beherrschung intuitiver Modelle der Informatik ist eine Form von Fachwissen, weil die Modelle Fachwissen repräsentieren. Wie in Kapitel 3 herausgearbeitet worden ist, dienen intuitive Modelle dem Verstehen, Erklären (z.B. in Dokumentationen und Diskussionen) und Kreieren informatischer Systeme – formal repräsentiert meist durch Programmtext. Über das Fachliche hinaus gibt es eine Reihe von metakognitiven Kompetenzen, die sich auf den Umgang mit intuitiven Modellen beziehen. Nun könnte man einwenden, dass metakognitive Kompetenzen unabhängig von einem inhaltlichen Bezug sind, also in den anderen Wissenschaften wie Chemie oder Mathematik ebenso wichtig sind und deshalb in diesem Informatik-orientierten Zusammenhang nicht diskutiert zu werden brauchen. Dem muss man aber entgegen halten, dass Modellierung in der Informatik einen besonderen Stellenwert hat. Denn ein wesentlicher Teil der Informatik dreht sich um Techniken, neue Systeme zu erfinden und ihre Komplexität zu beherrschen. Pointiert gesprochen geht es in den anderen Wissenschaften

vor allem darum Modelle für Realitätsausschnitte zu finden. In der Informatik wird der Modellierungsprozess selbst thematisiert (vgl. z.B. Hubwieser 1999, 2004).

15.2.1 Abstraktionsgrad erkennen

In Abschnitt 15.1 wurde herausgearbeitet, dass intuitive Modelle für Programmtexte sich im Abstraktionsgrad unterscheiden können. Der Abstraktionsgrad eines Modells muss aber erkannt werden, damit es nicht zu Fehlvorstellungen und Blockaden bei Problemlösungen kommt. So ist es eine wichtige Einsicht für Schüler/innen, dass Computerprogramme in der Feinstruktur oft ganz anders funktionieren als die anschauliche (holistische) Vorstellung einer Problemlösung, die man im Kopf hat. Damit verbunden ist die Erkenntnis, dass es für ein und dieselbe antizipatorische Intuition unterschiedliche Implementierungen geben kann. Das heißt, man muss den Abstand sehen, den ein abstraktes Modell zu möglichen Implementierungen hat. Wenn man z.B. die Abstraktheit der Lösungsidee für Bubblesort („Jedes Element prüft, ob der linke Nachbar kleiner ist, und tauscht gegebenenfalls mit ihm den Platz“) verkennt, versucht man vielleicht die Elemente der Liste als konkurrent agierende Objekte zu implementieren und scheitert an dieser komplexen Aufgabe.

Am unteren Ende der Abstraktionsskala stehen subatomare Modelle, die eine bloße logische Facette einer Programmanweisung zum Ausdruck bringen. Auch sie müssen als solche erkannt werden. Anderenfalls sucht man überflüssigerweise nach Programmkonstrukten, die gar nicht notwendig sind oder gar nicht existieren. Wer nicht merkt, dass bei einer Zuweisung im Behältermodell die Zerstörung des vorigen Inhaltes nur eine Bedeutungsfacette ist, kommt vielleicht auf die Idee, in einem Programm müsse vor einer neuen Zuweisung der Art `x = 2` zuerst eine spezielle Löschoperation ausgeführt werden.

15.2.2 Fokus und Grenzen wahrnehmen

Intuitive Modelle der Informatik sind in der Regel ungenau und geben ein informatisches Konzept nur unzureichend wieder. Sie fokussieren auf bestimmte Features des modellierten Konzeptes und lassen andere außer Acht. In der Fokussierung und der damit verbundenen Einfachheit liegt die Eingängigkeit und Ausdrucksstärke eines Modells. Wer dies nicht würdigt und deshalb keinen Mut zur Vereinfachung hat, ist nicht in der Lage, selbst Modelle für Erklärungen zu erfinden.

Andererseits muss man sich der mit der Fokussierung verbundenen Grenzen der Anwendbarkeit bewusst sein und sie auch explizieren können (z.B. wenn man das Modell in einer Diskussion verwendet). Das impliziert zweierlei: Die Grenze der Anwendbarkeit eines Modells kennt man nur, wenn man Kontexte benennen kann, in denen es untauglich ist. Zum zweiten kann man das fokussierende Modell trotz aller Ungenauigkeiten nur dann als tauglich akzeptieren, wenn man es als Teil eines Clusters von Modellen zur gleichen Thematik sieht. Man muss sich klar machen, dass die Genauigkeit und Schärfe einer Erklärung sich erst aus der Zusammenschau mehrerer Modelle ergibt.

15.2.3 Medien- und Kommunikationskompetenz

Intuitive Modelle müssen in irgendeiner Weise (verbal oder visuell) dokumentiert („materialisiert“) sein, um sie sich selbst ins Bewusstsein zu rufen und anderen mitzuteilen. Solche Repräsentationen sind also Medien und spielen eine zentrale Rolle im „situated learning“ in einer „community of practice“ (vgl. z.B. Ben-Ari 2004). Zu den Kompetenzen, die sich primär auf den Aspekt der Mediengestaltung beziehen, gehören:

- Dekorative Elemente eines Modells erkennen (und erfinden), die die Gestalt des Analogons abrunden aber kein Element des abgebildeten Konzepts repräsentieren.
- Den (kognitiven) Nutzen eines Modells für das eigene Verstehen beurteilen.
- Für eine Zielgruppe (z.B. Kunden in einem Softwareprojekt oder Kollegen in einem Entwicklungsteam) und einen sozialen Kontext (z.B. Dokumentation einer Software, Vorstellen einer Idee in der Entwurfsphase eines Projekts) geeignete Modellrepräsentationen auswählen.

15.3 Intuitive Modelle und Scaffolding

Eine typische Lehraktivität im Informatikunterricht ist die Unterstützung einer eigenständigen Problemlösungsaktivität (Scaffolding)[21]. Wenn ein Schüler mit einer Programmieraufgabe nicht weiterkommt oder einen semantischen Fehler nicht finden kann, erhält er oder sie einen Tipp, der eine Barriere überwinden helfen soll, aber nicht die Lösung vorwegnimmt. Würde eine Lehrperson die Lösung verraten, sabotiert sie damit den ursprünglich geplanten Lernprozess. Wenn die betroffenen Schüler/innen nicht in der Lage sind, das Problem zu lösen, es also zu schwierig für sie ist, ist offenbar an dieser Stelle besonders intensives Lernen notwendig. Durch ein Verraten der Lösung wird die gesamte Anbahnung des Lernprozesses (Auseinandersetzung mit der Aufgabenstellung, Ausprobieren verschiedener Ansätze etc.) zunichte gemacht und die Lernenden des Erfolgserlebnisses einer eigenständigen Lösung beraubt.

Ein geeigneter Tipp kann an intuitive Modelle anknüpfen, die Schüler verstehen. Um gute Tipps gestalten zu können, braucht man Wissen über mentale Modelle von Anfängern zur Interpretation von Programmtexten und typische fehlerhafte Anwendungen dieser Modelle.

Hierzu einige Beispiele:

- Wenn die Lernenden „Startschwierigkeiten“ haben und nicht in der Lage sind, auch nur einen Ansatz für das Programm zu finden, könnte zunächst eine eingängige metaphorische Beschreibung einer Problemlösung unter Verwendung von Modellen erarbeitet werden.
- Möglicherweise können Schüler/innen einen Algorithmus beschreiben, aber es gelingt ihnen die Implementierung nicht, weil die gedankliche Verbindung zwischen Komponenten ihres mentalen Modells und formalen Programmkonstrukten fehlen. In diesem Fall kann eine Hilfestellung darin bestehen, diese Verbindung herzustellen.
- Ein semantischer Fehler in einem Programmtext, den die Schüler alleine nicht entdecken, kann verdeutlicht werden, indem man die kritische Programmpassage an einem Modell durchspielt. Hierbei können Fehlvorstellungen (bzw. unangemessene Verwendungen des Modells) entdeckt und expliziert werden.

15.4 Diskussion und Reflektion intuitiver Modelle

Es gibt viele Möglichkeiten, die Explikation und Reflektion intuitiver Modelle in den Informatikunterricht einzubringen. Darunter sind Visualisierungsübungen, Rollenspiele und Aktivitäten mit neuen Medien wie Mikro- und Nanowelten. Das Grundmuster einer solchen Lerneinheit besteht aus den drei Schritten Hinführung, Schüleraktivität und Reflektion.

15.4.1 Visualisierungsübungen

In einer Visualisierungsübung erhalten Schüler/innen die Aufgabe, ein Programmstück zu visualisieren. In der Hinführungsphase wird klargestellt, dass die Visualisierung anderen Personen die Arbeitsweise des Programms erklären soll. Es geht also um Kommunikation. Eventuell kann sogar die Zielgruppe spezifiziert werden (Fachleute, erwachsene Nichtfachleute wie z.B. Kunden in einem Softwareprojekt, Kinder). Hinsichtlich des geforderten Ergebnisses sind verschiedene Varianten denkbar:

- Storyboard mit Bleistift und Papier
- Trickfilm z.B. mit Modellen aus Lego
- Animationen z.B. mit Adobe Flash, Mediator oder Präsentationssoftware
- Minidrama unter Verwendung eines Sets vorgegebener Requisiten („Szenokasten“) wie z.B. Karten und Behälter.

[21] Die Unterstützung problemlösender Aktivitäten durch Tipps wird z.B. im pädagogischen Ansatz des cognitive apprenticeship kultiviert. Vgl. dazu Collins et al. 1989.

In der Hinführungsphase sollte ein (potentieller oder realer) Anwendungskontext für das Produkt umrissen werden. Beispielsweise könnte eine Animation Teil der Schul-Website werden. Ein Trickfilm könnte in eine Datenbank für Unterrichtsmaterialien eingespeist werden. Ein Storyboard kann die Grundlage für eine spätere Implementierung mit Flash sein.

Die von den Schülern eigenaktiv entwickelten Produkte werden von ihnen präsentiert und erläutert. Die Präsentationen sind Anlässe für anschließende Diskussionen über die verwendeten intuitiven Modelle (Reflektionsphase). Sie können auf verschiedene Aspekte fokussieren und durch entsprechende Fragen der Lehrperson eingeleitet werden:

- Welche Elemente des Modells sind frei erfunden?
- Was macht das Modell anschaulich und gut verständlich?
- An welchen Stellen weicht die Visualisierung von informatischen Konzepten ab?

15.4.2 Rollenspiele und Regelspiele

Bell et al. (1998) beschreiben Spiele mit Requisiten (z.B. Behälter mit einer Balkenwaage nach ihrem Gewicht sortieren) und Rollenspiele (z.B. das „Orange Game“, in dem es um Fragen der Datenkommunikation geht), die im Informatikunterricht durchgeführt werden können und dazu dienen, bestimmte Konzepte der Informatik auf enaktive Weise zu präfigurieren.

Für den Bereich der objektorientierten Programmierung gibt es Rollenspiele, bei denen die Teilnehmer in die Rolle eines Objektes schlüpfen und die Übergabe und Verarbeitung von Botschaften nachspielen (Adrianoff & Levine 2002; Jimenez-Diaz et al. 2005). Der Unterschied zu Dramen im Rahmen von Visualisierungsübungen (siehe voriger Abschnitt) liegt darin, das hier das Szenario professionell gestaltet ist und die Aktivitäten durch Regeln und Rollenbeschreibungen weitgehend festgelegt sind. Es geht hier nicht um die spontane Kreation von Darstellungsformen und Externalisierung interner mentaler Modelle sondern um das Nachvollziehen vorgegebenen Gedankenguts und körperliches Erleben mit allen Sinnen. Ben-Ari (1997) beschreibt mehrere Übungen, bei denen die Schüler „Roboter spielen“ und umgangssprachlich formulierte rekursive Algorithmen nachspielen (z.B. Schokolade essen). Adrianoff & Levine (2002) schlagen ein Spiel vor, bei dem Schüler sich in die Rolle eines Akrobaten (Objekt der Klasse `Akrobat`) versetzen, der auf Zuruf bestimmte Dinge tut. Wenn ein „Akrobat“ z.B. die Botschaft „Clap“ gefolgt von einer ganzen Zahl `n` erhält, soll er `n` Mal in die Hände klatschen. Wichtig ist, dass diese Aktivitäten tatsächlich (physisch) ausgeführt werden und so das Rollenspiel zu einem Erlebnis wird, das man nicht vergisst. In der Reflektionsphase („Debriefing“) geht nach Empfehlung von Adrianoff & Levine (2002) die Lehrperson einen vorgegebenen Katalog von informatischen Konzepten durch (Objekt, Klasse, Botschaft, Parameter usw.), auf die das Rollenspiel zugeschnitten ist, und klärt mit den Teilnehmern, wie diese Konzepte im Spiel zum Ausdruck gekommen sind und wo die Grenzen und Schwächen dieser Modellierung lagen.

15.4.3 Gestaltete Medien und Mikrowelten

Intuitive Modelle bilden den Hintergrund für bildhafte Darstellungen in Büchern, Unterrichtsfilmen und gegenständlichen Modellen zum Anfassen. Für die Designer solcher Medien ist folglich Wissen über geeignete und ungeeignete intuitive Vorstellungen wichtig. Die (professionell gestaltete) Explikation einer typischen Fehlvorstellung kann genauso lehrreich sein wie eine fachlich korrekte Darstellung.

Auch Abbildungen auf der Basis einer standardisierten Modellbeschreibungssprache (z.B. UML) fußen auf intuitiven Vorstellungen. Ein Zustandsübergangsgraph etwa beschreibt einen Zustandswechsel als Bewegung. In der Industrie werden solche Diagramme zur Dokumentation von Software-Entwürfen oder Analyse-Ergebnissen verwendet. Im Unterrichtsbereich besitzen sie einen gewissen „Eigenwert“ und werden für das Erlernen von Konzepten – unabhängig von einem Software-Projekt –

eingesetzt. So hält Hubwieser die Modellierung durch UML-Diagramme auch ohne Implementierung für lehrreich[22].

Eine besonders weit entwickelte Form medialer Lernhilfen sind Mikrowelten wie die verschiedenen Implementierungen des Turtle-Konzeptes oder „Karel the Robot“[23].

Ein Medium ist noch kein Unterricht. Damit es zu Lernprozessen kommt, muss das Medium in elaborierende Aktivitäten eingebunden sein, die üblicherweise von der Lehrperson im Unterricht (z.B. durch geeignete Aufgabenstellungen) initiiert werden. Teil einer solchen Unterrichtssequenz kann und sollte auch eine kritische Reflektion der im medialen Material enthaltenen oder sogar thematisierten intuitiven Modelle sein[24].

15.5 Informatik im Kontext

Die Belegung der Sitze in einem Auto kann durch eine Multiliste beschrieben werden. Ein Beispiel in Python-Syntax ist:

```
auto = [["Anna", "Karl"], ["Tim", "Sandra"]]
```

Dabei repräsentiert jede Unterliste eine Sitzreihe. Nun ist die Vorstellung der Sitze in einem Auto einerseits ein prototypisches Beispiel einer Multiliste und hat den Charakter eines intuitiven Modells für ein abstraktes Konzept. Andererseits ist die Multiliste ein informatisches Modell eines Realitätsausschnitts. Informatische Modellierung und Rekonstruktion abstrakt-formaler Konzepte durch intuitive Modelle sind also ganz ähnliche kognitive Prozesse. Lediglich die Zielsetzung ist unterschiedlich: Generieren eines neuen Programms versus Verstehen oder Erklären eines Programms. In der Tat tendieren Schüler beim Versuch, Programmtexte zu erklären, zur „Rückmodellierung“, d.h. sie erfinden zum Programmtext passende Realitätsausschnitte (siehe Abschnitt 13.9).

Wer ein gewisses Repertoire intuitiver Modelle zum Verstehen und Erklären von Programmkonstrukten erworben hat, dem müsste eigentlich in vielen Fällen auch umgekehrt die Modellierung von Realitätsausschnitten besser gelingen, weil er oder sie Ähnlichkeiten der neuen Situation mit gespeicherten Prototypen in Betracht ziehen kann.

In der Naturwissenschaftsdidaktik ist seit Mitte der 1980iger Jahre ein Unterrichtsmodell entwickelt worden, das unter dem Namen „Science in Context“ oder „Salters Approach“ bekannt geworden ist (vgl. z.B. Bennet et al. 2002; Bennet & Lubben 2006; Huntemann et al. 1999). Kontexte sind alltagsrelevante und lebensweltbezogene Fragestellungen, die Sachstrukturen erkennen lassen und fachwissenschaftliche Inhalte in einen für Schüler einsichtigen Sinnzusammenhang stellen. Kontexte für die Chemie der Kohlenwasserstoffe (fachliches Konzept) sind z.B. die Gegenstände „Autokraftstoffe“ oder „Fleckenentferner“. In Lehrbüchern und Kurscurricula, die diesem Ansatz folgen, sind Kontexte mit zugehörigen (systematisch eingeführten) fachwissenschaftlichen Konzepten verbunden.

Alltagsrelevante Kontexte in der Informatik sind aber nicht nur Anwendungsbereiche für abstrakte Konzepte sondern auch Quelldomänen für intuitive Modelle. Wer etwa in einem Programmierprojekt sich mit der Modellierung von Sitzbelegungen in einem Auto beschäftigt, verwendet diesen Wirklichkeitsausschnitt als Domäne für metaphorische intuitive Vorstellungen zum informatischen Konzept der Liste. Beispiel: Ein Auto mit drei Sitzreihen, in dem sich nur Susanne als Fahrerin befindet, lässt sich durch die Multiliste `[["Susanne"], [], []]` modellieren. Eine leere Sitzreihe wird hier durch eine leere Liste modelliert. Das ist etwas anderes als eine nicht vorhandene Liste (Sitzreihe). Bei der Auswahl der Lebensweltbereiche, die in einem kontext-orientierten Informatikunterricht aufgegriffen werden, sollte ihre Eignung als „Materialbasis“ für intuitive Modelle beachtet werden.

---

[22] „Im Hinblick auf die Allgemeinbildung kommt es uns dabei hauptsächlich auf die Modellierung an. Die ojektorientierte Programmierung bietet lediglich eine einfache und effiziente Möglichkeit, die erarbeiteten Modelle zu implementieren, liefert aber für sich genommen kaum Beiträge zur Allgemeinbildung.“ (Hubwieser 2004, S. 94)

[23] Eine Übersicht über gängige informatische Mikrowelten liefern z.B. Henriksen & Kölling (2004).

[24] Beispiele für Animationen aus dem Bereich der Chemie mit Explorationsaufgaben, die auf den Modellierungsaspekt abzielen, findet man in der Website „Die Welt der kleinsten Teilchen“ (Weigend 2004)

15.6 Schluss und Ausblick

Unterricht an allgemeinbildenden Schulen richtet sich an junge Menschen, die sich in einem wichtigen Abschnitt ihrer Persönlichkeitsentwicklung befinden. In der Schule bilden sich nicht nur kognitive und soziale Kompetenzen sondern auch persönliche Interessen und Einstellungen zu beruflichen Perspektiven. Intuitive Modelle der Informatik repräsentieren Wissen, das ein Individuum beherrscht. Weil sie mit einem Gefühl der subjektiven Sicherheit verbunden sind, inspirieren sie zum Weiterdenken (einschließlich einer kritischen Revision), beflügeln die Fantasie und motivieren junge Menschen, sich mit der Informatik zu beschäftigen. „Je mehr Wissen jemand in seiner Entwicklung erwirbt, desto neugieriger wird er." (Oerter, Montada 1995, S. 769). Motivierender Unterricht knüpft deshalb an vorhandenes (intuitives) Wissen an und schafft so Zugänge zu neuem Wissen.

Als Medien materialisierte intuitive Modelle, anschauliche und leicht verständliche Abbilder (schwieriger) informatischer Konzepte werden in Schulbüchern, Museen, Zeitschriften und Fernsehsendungen verbreitet. Sie sind damit „Aushängeschilder", die das Bild der Informatik in der Öffentlichkeit prägen. Eine Aufgabe der Fachdidaktik ist es, solche Vermittlungsmodelle zu schaffen, um einem breiten Publikum den Zugang zur Informatik zu ermöglichen. Zu den konkreten Zielen in dieser Richtung gehört, den Katalog identifizierter (geeigneter und ungeeigneter) intuitiver Modelle zu erweitern, zu verfeinern, auf komplexere Themen auszudehnen und neue Medien und dazu passende Unterrichtsformen zu entwickeln, in denen diese Modelle elaboriert werden.

Literatur

Andrianoff, Steven K.; Levine, David B. (2002): Role playing in an object-oriented world. In: *ACM SIGCSE Bulletin* 2002 S. 121–125.

Aharoni, Dan (2000): Cogito, Ergo Sum! Cognitive Processes of Students Dealing with Data Structures. In: *ACM SIGCSE Bulletin*, S. 26–30.

Albert, Jürgen; Ottmann, Thomas (1983): *Automaten, Sprachen und Maschinen für Anwender*. Mannheim Wien Zürich (BI).

Anderson, John R. (1996): *Kognitive Psychologie*. Heidelberg Berlin Oxford (Spektrum Akademischer Verlag).

Anderson, John R. (1996a): ACT – A Simple Theory of Complex Cognition. In: *American Psychologist*, 51/4, S. 355–365.

Anderson, John R.; Jeffries, Robin (1985): Novice LISP Errors: Undetected Losses of Information from Working Memory. In: *Human-Computer Interaction*, 1, S. 107–131.

Anderson, John R.; Piroll, Peter; Farell Robert (1988): Learning to Program Recursive Functions. In: Chi, M.; Glaser, R., Farr, M. (Hrsg.): *The Nature of Expertise*. Hilsdale, USA, S. 153–184.

Anderson, John R.; Bothell, Daniel; Byrne, Michael D. (2004): An Integrated Theory of the Mind. In: *Psychological Review*, 111/4, S. 1036–1060.

Anjaneyulu, K. S. R.; Anderson, John R (1992): The Advantages of Data Flow Diagrams for Beginning Programming. In: *Intelligent Tutoring Systems*, S. 585–592

Arnheim, Rudolph (1972): *Anschauliches Denken*. Köln (DuMont Schaumberg)

Anzai, Y.; Uesato, Y. (1982): Learning Recursive Procedures by Middleschool Children. In: *Proceedings of the Fourth Annual Conference of the Cognitive Science Society*, Ann Arbor, Michigan.

Aschwanden, Christoph; Crosby, Martha (2006): Code Scanning Patterns in Program Comprehension. In: *Symposium on Skilled Human-Intelligent Agent Performance. Mesurement, Application and Symbiosis. Hawaii International Conference on Systems Science*, Januar 2006, Kauai, Hawaii.

Balzert, Heide (1999): *Lehrbuch der Objektmodellierung: Analyse und Entwurf*. Spektrum Akademischer Verlag, Heidelberg Berlin.

Baumann, Rüdeger (1990): *Didaktik der Informatik*. Stuttgart 1990.

Baumann, Rüdeger (1994): Der Weg vom Konkreten zum Abstrakten. In: *LOG IN* 14 (1994) Heft 1, S. 10 ff.

Baumgarten, Hans (2005): *Compendium Rhetoricum*. 2. Aufl. Göttingen (Vandenhoeck und Ruprecht).

Beck, Kent (2003): *Test Driven Development*. Boston u.a. (Addison Wesley) 2003.

Beck, Kent (1999): *Extreme Programming Explained*. Boston u.a. (Addison Wesley).

Ben-Ari, Mordechai (1997): Recursion: From Drama to Program. In: *Journal of Computer Science Education* 11/3, S. 9–12.

Blackwell, Alan Frank (1998): *Metaphor in Diagrams*. Dissertation University of Cambridge.

Ben-Ari, Mordechai (2001): Constructivism in Computer Science Education. In: *Journal of Computers in Mathematics and Science Teaching*, 20 /1, S. 45–73.

Ben-Ari, Mordechai (2004): Situated Learning in Computer Science Education. In: *Computer Science Education*, 14/2, S. 85-100.

Bennet, Judith; Holman, John; Lubben, Fred; Nicolson, Peter; Prior, Christine (2002): *Science in Context: The Salters Approach*. Contribution to the 2nd IPN-YSEG-Symposium.

Bennett, Judith; Lubben, Fred (2006): Context-based Chemistry: The Salters Approach. In: *International Journal of Science Education*, 28/9, S. 999–1015.

Bell, Tim; Witten, Ian H.; Fellows, Mike (1998): *Computer Science Unplugged ... off-line activities and games for all ages*. 1998. http://unplugged.canterbury.ac.nz, Zugriff am 15. 3. 2005.

Bernitzke, Fred Heinz (1987): *Mastery-Learning-Strategie als Unterrichtsalternative*. Frankfurt a.M. Bern New York Paris (Lang).

Bhuiyan, Shawkat; Greer, Jim E.; McCalla, Gordon I. (1994): Suppporting the Learning of Recursive Problem Solving. In: *Interactive Learning Environments*, 4/2, S. 115–139.

Blackwell, Alan F.; Whitley, Kirsten N.; Good, Judith; Petre Marian (2001): Cognitive Factors in Programming with Diagrams. In: *Artificial Intelligence Review* 15/1-2, Springer Netherlands, S. 95–114.

du Boulay, Benedict (1989): Some difficulties of learning to program. In: Soloway & Spohrer 1989, S. 283–301.

Bonar, Jeffrey; Soloway, Elliot (1985): Preprogramming Knowledge: A Major Source of Misconceptions in Novice Programmers. In: *Human–Computer Interaction*, Bd. 1 Nr. 2, S. 133 -161.

Bower, G.H.; Black, J.B.; Turner, T.J. (1979): Scripts in memory for texts. In: *Cognitive Psychology*, 11, S. 177–220.

Brewer, W.F.; Treyens,. J.C. (1981): Role of schemata in memory for places. In: *Cognitive Psychology*, 13, S. 207–230.

Brown, D.E. (1992): Using examples and analogies to remediate misconceptions in physics: Factors influencing conceptual change. In: *Journal of Research in Teaching Science*, 29 (1992) S. 17–34.

Byrnes, James P. (1996): *Cognitive Development and Learning in Instructional Contexts*. Boston London Toronto Sydney Tokyo Singapur.

Chiu, Ming Ming (1996): Exploring the origins, uses and interactions of students intuitions. In: *Journal for Research in Mathematical Education*, 27, S. 478–504

Chiu, Ming Ming (2000): Metaphoric Reasoning: Origins, uses, development and interactions in mathematics. In: *Educational Journal* 28/1, S. 13–46.

Chiu, Ming Ming (2001): Using Metaphors to understand and solve arithmetic problems: Novices and experts working with negative numbers. In: *Mathematical Thinking and Learning*, 3/3, S. 93–124.

Clement, John (1982): Students preconceptions in introductory mechanics. In: *American Journal of Physics* 50 S. 60–71.

Clement, John; Lochhead, John; Monk, George (1981): Translation difficulties in Learning Mathematics. *American Mathematical Monthly,* 88 (April 1981), S. 286–290.

Close, John; Dicheva, Darina (1997): Misconceptions in Recursion: Diagnostic Teaching. In: *Proceedngs of the Sixth Eurologo Conference "Learning and Exploring with Logo"*. Budapest, Ungarn 1997, S. 132–140.

Coenen, Hans Georg (2002): *Analogie und Metapher. Grundlegung einer Theorie der bildlichen Rede.* Berlin New York (Walter de Gruyter).

Collins, A. M. & Qulian, M. R. (1969): Retrieval time from semantic memory. In: *Journal of verbal Learning and Verbal Behaviour*, 8, S. 240–247.

Collins, A; Brown, J. S.; Newman, S.E. (1989): Cognitive apprenticeship: Teaching the crafts of reading, writing and mathematics. In: Resnick, L. B. (Hrsg.): *Knowing, learning and instruction: Essays in honor of Robert Glaser*, Hilsdale, NJ, S. 453–494

Collins, Allan; Brown, John Seely; Holum, Ann (1991): Cognitive apprenticeship: Making thinking visible. In: *American Educator*, 6/11, S. 38-46.

Craik, Kenneth (1943): *The nature of explanation*. Cambridge.

Crowley, Lillie; Thomas, Michael; Tall, David (1994): Algebra, Symbols and Translation of Meaning. In: *Proceedings of the 18th International Conference for the Psychology of Mathematics Education (PME)*, S. 240–247.

Crutzen, Cecile K.M.; Hein, Hans-Werner (1995): Objektorientiertes Denken als didaktische Basis der Informatik. In: Sigrid Schubert (Hrsg.): *Innovative Konzepte für die Ausbildung/ 6. GI-Fachtagung Informatik und Schule- INFOS ´95, Chemnitz, 25. – 28. September 1995*. Berlin Heidelberg New York London Paris Tokyo Hong Kong; Barcelona Budapest (Springer).

Dicheva, Darina; Close, John (1996): Mental Models of Recursion. In: *Journal of Educational Computing Research*, 14/1, S. 1–23.

Dewdney, A.K. (1995): *Der Turing Omnibus. Eine Reise durch die Informatik mit 66 Stationen*. Berlin Heidelberg New York.

Dingler, H. (1913): *Die Grundlagen der Naturphilosphie*, Leibzig.

diSessa, Andrea A.(1988): Knowledge in Pieces. In: George Forman & Peter B. Pufall (Hrsg.): *Constructivism in the Computer Age.* Hillsdale (Lawrence Erlbaum), S. 49–70.

diSessa, Andrea A. (1993): Toward an Epistemology of Physics. In: *Cognition and Instruction* 10/3, S. 105-225.

diSessa, Andrea A.: *Changing Minds. Computers, Learning, and Literacy*. Cambridge, Massachusetts (MIT Press) 2001.

Duell, M. (1997): Non-Software Examples of Software Design Patterns. In: *Object,* 7/5.

Du Boulay, Benedict (1989): Some difficulties of learning to program. In: Soloway & Spohrer 1989, S. 283–301.

Eckes, Thomas (1991: *Psychologie der Begriffe. Strukturen des Wissens und Prozesse der Kategorisierung*. Göttingen Toronto Zürich.

English, Lyn D. (1997) (Hrsg.): *Mathematical Reasoning. Analogies, Metaphors, and Images*. Mahwah New Jersey, London (Lawrence Erlbaum Associates, Publishers) 1997.

English, Lyn D. (2004): Mathematical and Analogical Reasoning of Young Learners. Mahwah (Lawrence Erlbaum).

Fach, Peter W.; Strothotte, Thomas (1994): Cognitive Maps: A Basis for Designing User Manuals for Direct Manipulation Interfaces. In: Tauber, M. J.; Mahling, D.E.; Arefi, F. (Hrsg.): *Cognitive Aspects of Visual Languages and Visual Interfaces*. Amsterdam London New York Tokyo (North-Holland) 1994, S. 133 ff.

Fischbein, Efraim (1987): *Intuition in Science and Mathematics*. Dordrecht Boston Lancaster Tokio (Reidel).

Flavell, John H. (1963): *Developmental Psychology of Jean Piaget*. Princeton (D. Van Nostrand).

Fothe, Michael (2005): Rekursion und Iteration: Voruntersuchung zu einem Test. In: Friedrich, Steffen: *Unterrichtskonzepte für informatische Bildung. Proceedings der INFOS 2005 in Dresden*, LNI – Proceedings, Bonn 2005, S. 207–218.

George, Carlisle E. (2000): EROSI- Visualising Recursion and Discovering New Errors. In: *ACM SIGCSE 2000*, S. 305–309.

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J. (1995): *Design Patterns -Elements of Resuable Object-Oriented Software*. Reading, MA (Addison Wesley).

Ginat, David (2001): Misleading Intuition in Algorithmic Problem Solving. In: *ACM SIGCSE Bulletin 2001*, S. 21–25.

Götschi, Tina (2003): *Mental Models of Recursion*. Masterarbeit an der Faculty of Science der University of the Witwatersrand, Johannisburg.

Griffith, A.K.; Preston, K.P. (1992): Grade-12-students‘ misconceptions relating to fundamental characteristics of atoms and Molecules. In: *Journal of Research in Science Teaching*, 29, S. 611–628.

Greening, Tony (1997): Examining Student Learning of Computer Science. In: *ACM SIGCSE 1997,* S. 63-66

Grudin, J. (1989): The case against user interface consistency. In: *Communications of the ACM,* 32/ 10, 1989, S. 1164–1173.

Haberlandt, Karl (1994): *Cognitive Psychology*. Boston London Toronto Sydney Tokyo, Singapore (Allyn and Bacon).

Hammer, David (1996): Misconceptions or P-Prims – How May Alternative Perspectives of Cognitive Structures Influence Instructional Perceptions and Intentions? In: *Journal of the Learning Sciences*, 5/2, S. 97 – 128

Henriksen, Poul; Kölling, Michael (2004): Greenfoot: Combining Object Visualization with Interaction. In: *Companion to the 19th annual ACM SIGPLAN conference on Object-orented programming systems, languages and applications (OOPSLA)* Vancouver, Kanada, November 2004, S. 73-82.

Hershkowitz, R. und Vinner, Sh. (1982): Basic geometric concepts – definitions and images. In: Vermandel, A. (Hrsg.): *Proceedings of the Sixth International Conference for the Psychology of Mathematics Education*. Antwerpen (Universitaire Instelling) 1982, S. 18–23.

Holland, Simon; Griffiths, Robert; Woodman, Mark (1997): Avoiding Object Misconceptions. In: *Proceedings ACM SIGCSE 1997*, S. 131–134.

Holmboe, Christian; McIver, Linda; George, Carlisle (2001): Research Agenda for Computer Science Education. In Kadoda, G. (Hrsg.): PPIG 13, Bournemouth, UK, April 2001.

Hubwieser, Peter (1999): Modellierung in der Schulinformatik. In: *Log In 19/1* (1999), S. 24 –29.

Hubwieser, Peter (2004): *Didaktik der Informatik. Grundlagen, Konzepte, Beispiele*. 2. Aufl. Berlin u.a. (Springer) 2004.

Huntemann, Heike; Paschmann, Antje; Parchmann, Ilka; Ralle, Bernd (1999): Chemie im Kontext – ein neues Konzept für den Chemieunterricht? In: *Chemikon 6/4*, S. 191-196.

Jeong, Allan; Lee, Mihwa; Lehrer, Richard (1999): Reflective Teaching of Logo. In: *Journal of the Learning Sciences*, 8/2 S. 245-288.

Jimenez-Diaz, Guillermo; Gomez-Albarran, Mercedes; Gomez-Martin, Marco A.; Gonzalez-Calero, Pedro A. (2005): Understanding Object-Oriented Software through Virtual Role-Play. In: *Fifths International Conference on Advanced Learning Technologies (ICALT '05)*, S. 875-877.

Johnson, W. Lewis (1990): Understanding and Debugging Novice Programs. In: *Artificial Intelligence*, 42, S. 51-97.

Johnson-Laird, Philip N. (1983): *Mental Models: Toward a Cognitive Science of Language, Inference and Consciousness*. Cambridge, MA (Harvard University Press).

Johnson-Laird, Philip N. (1996): *Der Computer im Kopf. Formen und Verfahren der Erkenntnis.* München (dtv).

Johnson-Laird, Philip N.; Girotto, Vittorio; Legrenzi, Paopo (1998): *Mental models: a gentle guide for outsiders*. URL: http://www.si.umich.edu/ICOS/gentleintro.html, Zugriff am 20.2.2007.

Kahneman, D.; Tversky, A. (1982): Subjective probability: A judgement of representativeness. In: Kahneman, D.; Slovic, P.; Tversky, A. (Hrsg.): *Judgement under Uncertainty: Heuristics and Biases*. Cambridge (Cambridge University Press) 1982, S. 32–47.

Kahney, Hank (1984): Modeling novice programmer behaviour. In: Yazdani, M.: *New horizons in educational computing*. EllisHorwood Ltd. 1984, S. 101–118.

Kahney, Hank (1989): What do novice programmers know about recursion? In: Soloway & Spohrer 1989, S. 209–228.

Karlgren, Klas; Ramberg, Robert (1996): Language Use & Conceptual Change in Learning. In: Paul Brna, Ana Paiva & John Self (Hrsg.): *The European Conference on Artificial Intelligence in Education (EuroAIED)* Lissabon, S. 45–51.

Kattmann, Ulrich (2005): Lernen mit anthropomorphen Vorstellungen? – Ergebnisse von Untersuchungen zur Didaktischen Rekonstruktion in der Biologie. In: *Zeitschrift für Didaktik der Naturwissenschaften* 11 (2005), S. 165–174.

Kessler, Claudius M.; Anderson, John R. (1989): Learning Flow of Control: Recursive and Iterative Procedures. In: Soloway & Spohrer 1989, S. 229–260.

Knuth, Eric J.; Stephens, Ana C.; McNeil, Nicole M.; Alibali, Martha W. (2006): Does Understanding the Equal Sign Matter? Evidence from Solving Equations. In: *Journal for Research in Mathematics Education*, 37/4, S. 297–312.

Kurland, D. Midian; Pea, Roy D. (1985): Children's mental models of recursive Logo programs. In: *Journal of Educational Computing Research*, Vol. 1(2), 1985, S. 235–243.

Lakoff, G. (1987): *Women, fire and dangerous things: what categories reveal about the mind*. University of Chicago Press.

Lakoff, George; Núnez, Rafael E. (1997): The Metaphorical Structure of Mathematics: Sketching Out Cognitive Foundations for a Mind-Based Mathematics. In: English, Lyn D. (Hrsg.): *Mathematical Reasoning. Analogies, Metaphors, and Images*. Mahwah New Jersey, London (Lawrence Erlbaum Associates, Publishers) S. 21–92.

Lehotská, Daniela (2006): Visual fractions in teacher training. In: Deryn Watson & David Benzie (Hrsg.): *IFIP WG 3.1, 3.3 & 3.5 Joint Conference*, Alesund, Norwegen 2006 Proceedings.

Lakoff, G.; Johnson, M. (1980): *The metaphors we live by*. Chicago (The University of Chicago Press).

Levy, Dalit; Lapidot, Tami (2000): Recursively Speaking: Analyzing Students' Discourse of Recursive Phenomena. In: *Proceedings of the ACM SIGCSE 2000,* Austin, Texas, USA S. 31–319.

Levy, Dalit; Lapidot, Tami; Paz, Tamar (2001): „It's just like the whole picture, but smaller": Expressions of gradualism, self-similarity, and othe pre-conceptions while classifying recursive phenomena. In: Kadoda, G. (Hrsg.): *Proc. PPIG 13*, Bournemouth UK, April 2001, S. 249–262

Ley, R.G. (1983): Cerebral laterality and imagery. In A. A. Sheikh (Hrsg.): *Imagery: Current theory, research and application*. New York (Wiley).

Linn, Marcia C.; Dalbey, John (1989): Cognitive Consequences of Programming Instruction. In: Soloway & Spohrer 1989, S.58 ff.

Logo Computer Systems Inc. (2004): *Microworlds JR*. Ressource Book with Extended Reference Guide.

Madison, Sandra; Gifford, James (2002): Modular Programming: Novive Misconceptions. In: *Journal of Research on Technology in Education*. 34, 3 S. 217ff.

Manis, V. & Little, J. (1995): *The Schematics of Computation*. Prentice Hall.

Mayrhauser, A. von; Vans, A. M. (1994): *Program Understanding – A Survey*. Technical Report CS-94-120, Department of Computer Science, Colorado State University USA.

Mayer, Richard E. (1989): The Psychology of How Novices Learn Computer Programming. In: Soloway & Spohrer, 1989, S. 129–159.

McCloskey, M. (1983): Naïve theories of motion. In: Gentner, D. und Stevens, A. (Hrsg.): *Mental models*. Hillsdale, NJ (Lawrence Erlbaum Associates, Inc.), S. 299–324.

Messaris, Paul (1993): Analog, Not Digital: Roots of Visual Literacy and Visual Intelligence. In: Beauchamp, Darell G. et al. (Hrsg.): *Selected readings of the IVLA Annual Conference "Visual Literacy in the Digital Age"*. Rochester New York 1993, S. 307–315.

Metzler, J. und Shepard R.N. (1974): Transformation Studies of the Internal Representations of Three Dimensional Objects. In: R.L. Solso (Hrsg.) *Theories of Cognitiv Psychology: The Loyola Symposium*. Hillsdale 1974, S. 147–201.

Miller, John Alexander (2004): *Promoting Computer Literacy through Programming Python*. Dissertation University of Michigan.

Mithen, Stephen; Boyer, Pascal (1996): Anthropomorphism and the evolution of cognition. In: *Journal of the Royal Anthropological Institute*, 2/4 S. 717 ff.

Moreno, Andrés & Myller, Niko (2003): Producing an Educationally Effective and Usable Tool for Learning, The Case of the Jeliot Family. In: *Proceedings of the International Conference on Networked e-learning for European Universities*, Granada, Spain.

Mosconi, Mauro, Porta; Marco (1999): Testing the Usability of Visual Languages: A Web-Based Methodology. In: Hans-Jörg Bullinger & Jürgen Ziegler (Hrsg.): *Proceedings of the 8th International Conference on Human-Computer Interaction (HCI'99)*, Band 1, München, S.1053–1057.

Mosconi, Mauro; Ottelli, Davide; Porta, Marco (2003): Alligator, a Web-based Distributed Visual Programminp Environment. In: *Proceedings WWW 2003*, Budapest, Ungarn, 20-24. Mai 2003, Poster.

Moser, Karin S. (2003): Mentale Modelle und ihre Bedeutung – kognitionspsychologische Grundlagen des (Miss-)Verstehens. In: U. Ganz-Blättler & P. Michel (Hrsg.) *Sinnbildlich schief: Missgriffe bei Symbolgenese und Symbolgebrauch*. Schriften zur Symbolforschung, Vol. 13), Bern (Peter Lang), S. 181–205.

Nakamura, T. (1974): The learning of motion and force. In: Takahashi, K. und Hosoya, K. (Hrsg.): *Introduction to methods of Kyokuchi: Modern science education*. Tokio.

Navarro-Prieto, Raquel; Cañas, Jose J. (2001): Are visual programming languages better? The role of imagery in program comprehension. In: Journal Human-Computer Studies (2001) 54, S. 799–829.

Odell, James J. (1994): Six Different Kinds of Composition. In: *JOOP* 5/8.

Oerter, Montada (1994) (Hrsg.) *Entwicklungspsychologie*. Weinheim (Beltz).

Oviedo, Maria Cecilia Núnez; Clement, John (2003): Model Competition: A Strategy Based on Model Based Teaching and Learning Theory. In: *Procedings of NARST*, Philadelphia 23-26. März 2003.

Paivio, Allan (1971): Imagery and language: In: S. J. Seagl (Hrsg.): *Imagery: Current cognitive approaches*. New York (Holt, Rinehardt & Winston), S. 7-32.

Paivio, Allan (1986): *Mental representations – a dual coding approach*. Oxford (Oxford University Press).

Papert, Seymour (1980): Teaching Children Thinking. In: R. Taylor (Hrsg.): *The Computer in School: Tutor, Tool, Tutee*. New York (College Press), S. 161–176.

Parikh, Jadhish (1994): *Intuition: the new frontier of management*. Oxford (Blackwell).

Pennington, Nancy (1987): Stimulus Structures and Mental Representations in Expert Comprehension of Computer Programs. In *Cognitive Psychology* 19, S. 295–341.

Perkins, D.N.; Hancock, Chris; Hobbs, Renee; Martin, Fay (1989): Conditions of learning in novice programmers. Educational Technology Center, Harvard University. In: Soloway & Spohrer 1989.

Piaget, Jean (2003): *Meine Theorie der geistigen Entwicklung*. Herausgegeben von Reinhard Fatke. Weinheim, Basel, Berlin (Beltz).

Poincaré, Henri (1905): *Science and Hypothesis*. London (Walter Scott Publishing).

Popper, Karl R. (1934): *Logik der Forschung*. Wien (Verlag Julius Springer).

Putnam, Ralph T.; Sleeman, D.; Baxter Juliet A.; Kuspa, Laiani, K. (1989): A summary of misconceptions of high school basic programmers. In: Soloway & Spohrer, 1989, S. 301–314.

Riehle, D.; Züllighoven, H. (1996): Understanding and Using Patterns in Software Development, In: Karl Lieberherr & Roberto Zicari (Hrsg.): *Theory and Practice of Object Systems*, Band 2, Nr. 1, S. 3-13, 1996.

Roger, J.M.; Cisero, C.A.; Carlo, M.S. (1993): Techniques amd Procedures for Assessing Cognitive Skills. In: *Review of Educational Research*, 63/2, S. 201–243.

Reed, David (1998): Incorporating Problem-solving Patterns in CS1. In: *Proc. ACM SIGCSE*, S. 6-9.

Rosch, Eleonor (1973): On the internal structure of perceptual and semantic categories. In: T.H. Moore (Hrsg.): *Cognitive development and the acquisition of language*. New York (Academic Press).

Rosch, Eleonor (1975): Cognitive representations of semantic categories. In: *Journal of Experimental Psychology*, 104, S. 192–223.

Rosnick, Peter (1981): Some Misconceptions Concerning the Concept of Variable. Are you careful about defining your variables? In: *The Mathematics Teacher*, 74 (6), September 1981, S. 418–420, 450.

Samurçay, R. (1989): The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. In: Soloway & Spohrer, 1989, S. 161–178.

Sanner, M. F.; Stoffler, D.; Olson, A.J. (2002): ViPEr, a visual programming environment for Python. In: *Proceedings of the 10th international Python Conference*, Februar 2002, S. 103–115.

Sasse, Martina Angela (1997): *Eliciting and Describing Users' Models of Computer Systems*. Dissertation. Birmingham (University of Birmingham).

Schank, R.C.; Abelson, R. (1977): Scripts, plans, goals and understanding. Hillsdale.

Sajaniemi, Jorma (2002): Visualizing Roles of Variables to Novice Programmers. In: Kuljis, J.; Baldwin, L.; Scoble, R. (Hrsg.): *Proc PPIG* 14, Brunel University.

Sleeman, D.; Putman, D.; Baxter, R.T.; Kuspa, L. (1989): A summary of misconceptions of high school Basic programmers. In: Soloway & Spohrer 1989, S. 301–314.

Schubert, Sigrid; Schwill, Andreas (2004): *Didaktik der Informatik*. Heidelberg Berlin (Spektrum).

Schulmeister, Rolf (2002): *Grundlagen hypermedialer Lernsysteme*. Oldenbourg Verlag, München u.a.

Schwank, Inge (2003): Einführung in prädikatives und funktionales Denken. In: *Zeitschrift für Didaktik der Mathematik*, 35/3, S. 70–78.

Schwank, Inge (2005): Maschinenintelligenz: ein Ergebnis der Mathematisierung von Vorgängen – Zur Idee und Geschichte der Dynamischen Labyrinthe. In C. Kaune, I. Schwank & J. Sjuts: *Mathematikdidaktik im Wissenschaftsgefüge: Zum Verstehen und Unterrichten mathematischen Denkens*. Osnabrück (Forschungsinstitut für Mathematikdidaktik).

Schwill, Andreas (1993): Fundamentale Ideen der Informatik. In: Zentralblatt der Mathematik 20 (1993) S. 20–31.

Schwill, Andreas (2001): Ab wann kann man mit Kindern Informatik machen? Eine Studie über informatische Fähigkeiten von Kindern. In: Reinhard Keil-Slawik; Johannes Magenheim (Hrsg.): *Informatikunterricht und Medienbildung, INFOS 2001, 9. GI-Fachtagung Informatik und Schule, 17.-20. September 2001 in Paderborn*. LNI 8 GI 2001, S. 13–30.

Smith, John P.; diSessa, Andrea A.; Roschelle, Jeremy (1994): Misconceptions Reconceived: A Constructivist Analysis of Knowledge in Transition. In: *Journal of the Learning Sciences*, Vol. 3 (1993/94) Nr. 2.

Soloway, E.; Spohrer, J. C. (1989) (Hrsg.): *Studying the novice programmer*. Hilsdale (Lawrence Erlbaum Associates).

Sommerville, Ian (1997): *Software Engineerung*. 5. Auflage, Harlow, England (Addison-Wesley).

Spohrer, James C., Soloway, Elliot, Pope, Edgar (1989): A Goal/Plan Analysis of Buggy Pascal Programs. In: Soloway & Spohrer, 1989, 355–399.

Stary, Joachim (1997): *Visualisieren: eine Studien- und Praxisbuch*. Berlin.

Stern, Linda; Naish, Lee (2002): Visual Representations for Recursive Algorithms. In: *Proceedings of the SIGCSE 2002*, Covington, Kentucky, USA 2002, S. 196–200.

Storey, M.-A. D.; Wong, K.; Müller, H. A. (1997): How Do Program Understanding Tools Affect How Programmers Understand Programs? In: *Proceedings of the Fourth Working Conference on Reverse Engineering*, S. 12–21.

Ströker, Elisabeth (1967): *Denkwege der Chemie*. Freiburg München.

Sutcliffe, Alistair G.; Darzentas, Jenny S. (1994): Use of Visual Media in Explanation. In: Tauber, M. J.; Mahling, D.E.; Arefi, F. (Hrsg.): *Cognitive Aspects of Visual Languages and Visual Interfaces*. Amsterdam London New York Tokio (North-Holland). 1994, S.105 ff.

Smith, Randall B.; Ungar, David (1995): Programming as an Experience: The Inspiration for Self. In: *Proceedings of ECOOP '95*, Lecture Notes on Computer Science 952 (Springer).

Taivalsaari, Antero (1992): Kevo – a prototype-based object-oriented language based on concatenation and module operations. Technical report DCS-197-IR, University of Victoria,B.C., Kanada, Juni 1992.

Taivalsaari, Antero (1997): Classes vs. Prototypes. Some Philosphical and Historical Observations. In: JOOP 10(7) 1997, S. 44–50.

Temple, David M.; Guest, Stece P. (1994): Diagrammatic Techniques for the Visualisation of Object Oriented Programming. In: Tauber, M. J.; Mahling, D.E.; Arefi, F. (Hrsg.): *Cognitive Aspects of Visual Languages and Visual Interfaces*. Amsterdam London New York Tokyo (North-Holland) 1994, S.259 ff.

Thomas, Marco (2003): Informatische Modelle zur Strukturierung von Anfangsunterricht. In: Peter Hubwieser (Hrsg.): *Informatische Fachkonzepte im Unterricht, INFOS 2003, GI-Fachtagung Informatik und Schule*, 17.-19.September 2003 in Garching bei München. LNI 32 GI, S. 155–164.

Tuzova, Olga; Katz, Yekuda (2001): *Logo Art Gallery*. http://www.geocities.com/CollegePark/Lab/2276/ 2001. Zugriff am 1. Juli 2006.

Ueno, Naok (1993): Reconsidering P-Prims Theory From the Viewpoint of Situated Cognition. In: Cognition and Instruction. 10, 3, S. 239 ff.

Ueding, Gert (2005): *Klassische Rhetorik*. 4. Aufl. München (Beck).

Uitenbroek, Daan (2000): *SISA Fisher*, Hilversum, http://home.clara.net/sisa/fisher.htm.

Van der Veer, Gerrit C. (1994): Mental Models of Computer Systems: Visual Languages in the Mind. In: Tauber, M. J.; Mahling, D.E.; Arefi, F. (Hrsg.): Cognitive Aspects of Visual Languages and Visual Interfaces. Amsterdam London New York Tokyo (North-Holland), S. 3 ff.

Vossenkuhl, Wilhelm (1998): „Verstehen" verstehen. In: Kanitschneider & Wetz (Hrsg.): *Hermeneutik und Naturalismus*. Tübingen.

Van Rossum, Guido; Yee, Ka-Ping (2001): *Iterators* (PEP 234), 30.1.2001. http://www.python.org/peps/pep-0234.html. Zugriff am 1. Mai 2006.

Weigend, Michael (1999): Roboter im Internet (1). In: *Informatik betrifft uns*, 1999/4.

Weigend, Michael (2005): Intuitive Modelle in der Informatik. In: Friedrich, Steffen (Hrsg.) *Unterrichtskonzepte für informatische Bildung. INFOS 2005*. Lecture Notes in Informatics (LNI) – Proceedings. Bonn (GI) 2005 S. 275–284.

Weigend, Michael (2005a): Objektorientierte Programmierung. In: Markus Nix (Hrsg.); Martin Grimme; Torsten Marek; Michael Weigend; Wolfgang Weitz: *Exploring Python*. Frankfurt (entwickler.press).

Weigend, Michael (2006): *Python-Gepackt*. Dritte Auflage. Heidelberg (Redline) 2006

Weigend, Michael (2006a): *Objektorientierte Programmierung mit Python*. Dritte Auflage. Heidelberg (Redline).

Weigend, Michael (2006b): Experimental Programming. In: Deryn Watson & David Benzie (Hrsg.): *IFIP WG 3.1, 3.3 & 3.5 Joint Conference*, Alesund, Norwegen 2006 Proceedings.

Weigend, Michael (2006c): Design of web-based educational games for informatics classes – some insights from workshops with the Python Visual Sandbox. In: *Workshop „GML – Grundfragen multimedialer Lehre"*, Potsdam 14. – 15.3.2006, im Druck.

Weigend, Michael (2006d): From Intuition to Programme. In: Roland T. Mittermeir (Hrsg.): *Informatics Education – The Bridge between Using and Understanding Computers*. ISSEP 2006 Vilnius, Litauen Proceedings, Berlin Heidelberg (Springer), S. 117–126.

Wertheimer, Max (1925): Über Gestalttheorie. In: *Philosopgische Zeitschrift für Forschung und Aussprache*, 1, S. 39-60.

Wittgenstein, Ludwig (1953): *Philosophische Untersuchungen*. 3. Auflage, Frankfurt a.M. 1982 (zuerst erschienen 1953).

Wu, Cheng-Chih; Dale, Neil B.; Bethel, Lowell J. (1998): Conceptual Models and Cognitive Learning Styles in Teaching Recursion. In: *Proceedings of the 29th SIGCSE technical symposium on computer science education*, Atlanta Georgia, USA S. 223-227.

Zachary, Joseph L. (1997): The Gestalt of Scientific Programming: Problem, Model, Method, Implementation, Assessment. In: *ACM SIGCSE 1997*, S. 238-242.

Zietsman, Aletta; Clement, John (1997): The Role of Extreme Case Reasoning in Instruction for Conceptual Change. In: *Journal of the Learning Sciences*, 6/1, S. 61-89.

Anhang

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

ACM	Association for Computing Machinery
DL	Dynamisches Labyrinth
GI	Gesellschaft für Informatik
ICOS	
IFIP	International Federation for Information Processing
IVLA	International Visual Literacy Association
JOOP	Journal of Object Oriented Programming
LNI	Lecture Notes in Informatics
OOP	Objektorientierte Programmierung
PPIG	
p-Prim	phänomenologisches Primitiv (diSessa)
PVS	Python Visual Sandbox
SIGCSE	Special Interest Group Computer Science Education
SIGPLAN	Special Interest Group
UML	Uniform Modeling Language

1 Ergänzungen zur Repräsentation intuitiver Modelle

1.1 Verwendung unterschiedlicher Metaphern beim mathematischen Problemlösen

Chiu (2001) ging in einer Untersuchung mit 12 Erwachsenen (18 bis 25 Jahre) und 12 Kindern (12-13 Jahre) der Frage nach, in welchem Umfang Metaphern bei Problemlösungen und zur Erklärung arithmetischer Operationen verwendet werden. Die Untersuchung bestand aus zwei Teilen. Im ersten Teil sollten die Probanden Profite aus einem Börsenhandel berechnen und anschließend ihre Rechnung begründen. Die Rechnungen bestanden aus Additionen und Multiplikationen mit positiven und negativen Zahlen. Im zweiten Teil sollten die Versuchspersonen Fragen der Art „How do you make sense of `-5+8`?“ beantworten. Alle schriftlichen, mündlichen und gestischen Äußerungen wurden protokolliert und analysiert. In den Antworten wurde nach metaphorischen Erklärungen mit Konzepten aus verschiedenen Bereichen (Quell-Domänen) gesucht[25], unter anderem

(1) Bewegung („arithmetic is motion along a line“): In dieser Domäne werden positive Zahlen durch Schritte nach rechts und negative Zahlen durch Schritte nach links vom Ursprung dargestellt. Die Rechnung `-5+8` wird so interpretiert, dass man zuerst 5 Schritte nach links und dann 8 Schritte nach rechts geht. Dabei wird der Weg, den man zuerst nach links gegangen ist, gewissermaßen annulliert.

(2) Manipulation von Objekten („arithmetic is manipulating objects”): Natürliche Zahlen werden durch eine entsprechende Anzahl von Objekten repräsentiert, negative ganze Zahlen durch Löcher, die man mit Objekten füllen („auslöschen“) kann, oder Objekte aus Antimaterie.

(3) Soziale Transaktionen (“arithmetic is a social transaction”). Positive Zahlen stellt man sich als Besitz vor und negative Zahlen als Schulden.

Die Probanden äußerten die Metaphern spontan. Wenn im ersten Teil keine spontane Erklärung kam, sondern nur das Ergebnis genannt wurde, fragte der Interviewer nur ein einziges Mal nach. Bemerkenswert sind folgende Ergebnisse:

Erwachsene verwendeten bei den Erklärungen im zweiten Teil deutlich mehr Metaphern (Mittelwert 7.0, Standardabweichung 2.7) als Kinder (Mittelwert 2.25, Standardabweichung 0.35). Erwachsene verwendeten außerdem mehr unterschiedliche Metaphern und bevorzugten die beiden ersten Quelldomänen, während die Kinder fast ausschließlich auf die Bewegungsdomäne zurückgriffen.

Zur Erklärung der Problemlösungen im ersten Teil der Untersuchung verwendeten (genauer: äußerten) dagegen die Kinder mehr Metaphern (Mittelwert 3.67) als Erwachsene (Mittelwert 0.25).

Die Verwendung multipler Intuitionen zeigte auch eine frühere Untersuchung von Chiu (1996). Er stellte 16 Schülerinnen und Schülern im Alter von 12 bis 14 Jahren die Aufgabe, die Längen verschieden geformter Wege zwischen zwei Punkten abzuschätzen. Er interviewte sie während der Problemlösung und registrierte die Intuitionen, die sie in ihren Erklärungen verwendeten. Es zeigte sich, dass eine Person im Schnitt 2.63 unterschiedliche Intuitionen erwähnte. Insgesamt verwendete sie 5.31 intuitive Vorstellungen. Vielfalt und Häufigkeiten sanken nur unwesentlich, nachdem der Person ein Algorithmus zur Lösung der Aufgabe beigebracht worden ist (4.94 Äußerungen 2.18 verschiedenen Intuitionen).

1.2 Repräsentation von Sprachkonzepten durch eine Beispielkollektion – die Python-Kurzreferenz von O’Reilly

Tab. 23 zeigt einen Auszug aus einer Kurzreferenz für Python von O’Reilly, die auf ein Lesezeichen passt. Jedes Beispiel verdeutlicht unterschiedliche Aspekte von Tupeln. Kurzreferenzen dienen in der Regel als Erinnerungshilfe für Leute, die die Programmiersprache bereits beherrschen. Aber auch

[25] Eine systematische Übersicht über Metaphern für arithmetische Operationen findet man in Lakoff et al. 1997

erfahrene Programmierer, die die betreffende Sprache nicht kennen, gewinnen bereits einen Einblick. Beides deutet darauf hin, dass Menschen auch intern Programmiersprachenkonzepte durch Beispielkollektionen repräsentieren können.

Operation	Interpretation
`()`	An empty tuple
`t1 = (0,)`	A one-item-tuple (not an expression)
`t2 = (0, 'Ni', 1.2, 3)`	A four-item tuple
`t[i]`	Indexing
`t[i:j]`	Slicing

Tab. 23: In der Python-Kurzreferenz von O'Reilly wird allein durch Beispiele erklärt, was ein Tupel ist.

Effiziente Beispielkollektionen enthalten meist Prototypen mit besonders hoher Repräsentativität und Sonderfälle. Die ersten drei Beispiele in Tab. 23 stellen einen Prototyp (Beispiel 3) und zwei Sonderfälle (die ersten beiden Beispiele) dar. In die Sonderfälle kann auch Wissen über andere Konzepte einfließen. So wird im Beispiel des Python-Tupels (0,) zum Ausdruck gebracht, dass sich durch das Komma ein einelementiges Tupel von einem Klammerausdruck unterscheidet (vgl. Tab. 23). Derartige Abgrenzungen können auch durch Gegenbeispiele expliziert werden. So ist das Literal `(1)` kein Tupel, sondern ein Ausdruck und das Statement `t[0] = 2` ist nicht erlaubt, wenn `t` ein Tupel ist. Denn bei Python sind Tupel unveränderbare Objekte.

1.3 Von der Schwierigkeit intuitive Modelle zu visualisieren

In einer umfangreichen Untersuchung von van der Veer (1994) mit 607 Personen aus drei europäischen Ländern wurde versucht, mentale Modelle über die Arbeitsweise von Computersystemen zu ermitteln. Die Aufgabe war, mit Hilfe beliebiger Ausdrucksmittel (Text, Zeichnungen etc.) auf einem Blatt Papier zu erklären, wie man einen gespeicherten Text sucht, ihn auf dem Bildschirm darstellt, eine Kopie speichert und ihn ausdruckt („teach back"). Aus den Ergebnissen erhoffte man sich Erkenntnisse über verwendete Modellvorstellungen. Wie die Analyse der abgelieferten Erklärungsversuche zeigte, wurden zwar häufig mehrere Darstellungsformen kombiniert, aber es dominierten eindeutig verbale Repräsentationen (Tab. 24). Bildhafte oder ikonische Darstellungen spielten eine untergeordnete Rolle.

Ausdrucksmittel	Anteil
Text	91%
Bild	25%
Ikonische Darstellung	28%
Regeln („Wenn…, dann..")	6%
Programmtext („Pseudocode")	34%

Tab. 24: Verwendete Ausdrucksmittel bei Teach-back-Versuchen (van der Veer 1994)

Um das Problem mangelnder Darstellungsfähigkeit zu umgehen, werden in der Python Visual Sandbox (PVS) deshalb nicht spontane Visualisierungsversuche der Teilnehmer gesammelt, sondern stattdessen eine Vielzahl von bildlichen Darstellungen vorgegeben. Sie sollen mit Programmtexten in Beziehung gesetzt und beurteilt werden (Python Visual und Python Quiz) oder können als Hilfe abgerufen werden und müssen dann im Hinblick auf ihre Brauchbarkeit mit einer dreistufigen Skala bewertet werden (Python Puzzle). Teilnehmer von Workshops mit der PVS äußerten tatsächlich, dass sie in einigen Visualisierungen eigene Vorstellungen treffend wiederentdeckt hätten. Gleichzeitig räumten sie ein, dass sie wahrscheinlich selbst nicht in der Lage gewesen wären, spontan eine derartige Repräsentation zu erfinden.

1.4 Beispiele für Tropen in der Informatik

Metaphern

- Bauplan (für Klasse)
- Absturz (für den Übergang eines Prozesses in einen irregulären Zustand)
- Verzweigung (für bedingte Anweisung, if-Anweisung)
- Schleife (für Iteration, for- oder while-Anweisung)
- Botschaft (für Aufruf einer Methode eines Objektes)
- Inhalt (für Wert, der einem Variablennamen zugewiesen ist)

Metonymien

- Tabelle (für Relation in einer relationalen Datenbank)
- Zeile (für Tupel einer Relation in einer RDB)

Katachresen

- zurückgeben (eine Funktion gibt das Ergebnis zurück)
- Objekt
- Anweisung

Allegorien

- Divide and conquer (rekursiver Algorithmus, bei dem ein Objekt aufgeteilt und rekursiv zunächst die Teile bearbeitet werden)
- Tiefensuche (Algorithmus für das Durchlaufen eines Baumes, bei dem man zuerst in die Tiefe geht)
- Sweep (Verfahren in der Algorithmischen Geometrie, das von der Vorstellung eines Scheibenwischers ausgeht, der Regentropfen von der Windschutzscheibe wegwischt)

Vergleiche

- Instanzen einer Klasse sind wie Häuser, die nach dem gleichen Bauplan gebaut worden sind.

1.5 Mikrowelten als einheitliche Domänen für konzeptionelle Metaphern

Vorstellungswelten können detailliert gestaltet und z.B. als multimediale Software oder mechanisches Spielzeug implementiert werden. Man spricht dann von Mikrowelten (microworlds). Mikrowelten sind explorative Lernumgebungen (Schulmeister 2002). Es sind künstliche Welten mit sehr einfachen Regeln, in denen die Benutzer sich frei bewegen und eigenaktiv vom Designer (Pädagogen) „verstecktes“ Wissen entdecken können.

Das wohl bekannteste und historisch erste Beispiel einer computerbasierten Mikrowelt ist die Turtle-Grafik der Programmiersprache Logo (Papert 1973, 1980). Die Firma Logo Computer Systems Inc. (LCSI) bietet verschiedene Mikrowelten für verschiedene Altergruppen an, die jeweils die Logo-Turtle enthalten. Die Turtle (Schildkröte) ist ein virtuelles Wesen, das wenige „angeborene“ Operationen beherrscht (auf der Stelle um einen bestimmten Winkel nach rechts drehen, eine gewisse Anzahl von Schritten nach vorne gehen). Bei ihrer Bewegung hinterlässt sie auf dem Untergrund eine Spur, so dass man durch Steuern der Turtle Bilder malen kann. Weitere Aktionsmöglichkeiten kann sie durch Kombination bereits beherrschter Operationen lernen. Insofern kann man das Steuern und Trainieren der Schildkröte als Metapher für prozedurales Programmieren bezeichnen.

Ist die Turtle eine Metapher, wie es gelegentlich unterstellt wird (z.B. Clements & Samara 1997)? Sie ist primär ein künstliches Wesen, das für sich selbst steht. Zwar könnte man sie aus Sicht der Informatik grob als Metapher für ein Objekt im Sinne der Objektorientierten Programmierung bezeich-

nen, also eine abstrakte Entität, die sich in einem Zustand befindet und Methoden beherrscht[26]. Aus Sicht der Kinder, die mit Logo arbeiten, ist dieser Aspekt jedoch weitgehend irrelevant.

Die Turtle ist aber eine geschlossene Gestalt, die man sich gut vorstellen kann. Ihre Bewegungsmöglichkeiten (gehen und drehen) und das Zeichnen von Linien durch Bewegung eines Stiftes sind einzelne getrennte Konzepte, die bereits Kindern im Vorschulalter vertraut sind. Die Kombination dieser Merkmale zu einer Einheit ist jedoch rein künstlich und existiert nicht in der Realität und erst recht nicht in der Erfahrungswelt der Kinder, die zum ersten Mal mit ihr arbeiten. Bei genauerem Hinsehen stellt man fest, dass die Komponenten des Konzeptes „Turtle" aus unterschiedlichen Erfahrungsbereichen kommen. So können sich echte Schildkröten und andere Tiere (einigermaßen) geradeaus durch den Sand bewegen und dabei eine Spur hinterlassen. Sie können auch die Richtung ändern, machen dies aber typischerweise kontinuierlich und bewegen sich dabei in Kurven. Drehungen auf der Stelle beobachtet man eher bei völlig anderen Objekten, die auf einer Achse angebracht sind (Wasserhahn, Kamera auf einem Stativ, Klavierhocker etc.). Das Konzept des Schrittes passt eher zu Wesen, die auf zwei Beinen laufen. Bei einem Vierfüßer – wie einer echten Schildkröte – ist der Schritt eines einzelnen Beins nicht unbedingt mit einer Vorwärtsbewegung verbunden. Der Schritt selbst ist eine Metapher für eine diskrete Längeneinheit.

Das Erfolgsgeheimnis der Mikrowelt liegt darin, dass sie ein kohärentes Set von Repräsentationen für unterschiedliche intuitive Modelle bereitstellt. Jedes für sich könnte auch auf andere Weise repräsentiert werden. Aber durch das kohärente Design entsteht eine künstliche Welt, eine Domäne, in der man sinnvolle zielgerichtete Aktivitäten ausführen kann. So erlaubt die Turtle-Welt Aktionen, die innerhalb dieser Welt Sinn machen aber die es in der Realwelt im Grunde nicht gibt:

Gehe 10 Schritte nach vorne

Drehe dich um 90 ° nach rechts

Setze einen Stift auf den Untergrund

Gehe 20 Schritte nach vorne

ist ein Bewegungsablauf, der in keinem realen Kontext ausgeführt wird. Innerhalb der Turtle-Welt ist das jedoch eine sinnvolle zielgerichtete Aktion, die eine bestimmte antizipierbare Wirkung hat. Diese Möglichkeit des zielgerichteten Handelns ginge verloren, wenn man bei Bewegungsspuren an Schildkröten im Sand, bei Schritten an eine Wanderung und bei Drehbewegungen an einen Wasserhahn denken würde. Beobachtungen haben gezeigt, dass die Logik der Turtle nicht intuitiv ist, sondern von den Kindern erst gelernt und geübt werden muss (Clements & Samara 1997). Zum Beispiel wird die Anweisung „drehe nach rechts" mit „gehe nach rechts" verwechselt. Manchmal vollziehen die Kinder einzelne Bewegungsabläufe mit dem eigenen Körper nach, um sie sich besser vorstellen zu können.

Die Turtle ist selbst im Wesentlichen keine Metapher. Erst durch die Aktivitäten in der Logo-Mikrowelt entstehen die konzeptionellen Metaphern im Sinne Lakoffs, die den Lerneffekt bringen. Im Falle der Turtle-Grafik sind das vor allem Konzepte der Geometrie der Ebene (vgl. z.B. Clements und Samara 1997). Wenn ein Kind eine Folge von Logo-Anweisungen (oder gar eine Prozedur) schreibt, die die Turtle ein Rechteck auf den Bildschirm zeichnen lässt, bringt es damit ein mentales Modell eines Rechtecks zum Ausdruck. Dieses Modell basiert auf Bewegung. Ein Rechteck entsteht nämlich, wenn man x Schritte vorwärts, dann sich um 90° nach rechts dreht, dann y Schritte vorwärts geht, dann sich um 90° nach rechts dreht usw. Kritische Punkte sind der Drehwinkel von 90° und die Tatsache, dass die erste und die dritte sowie die zweite und vierte Vorwärtsbewegung (also die gegenüber liegenden Seiten) gleich lang sein müssen. Sonst sieht das Ergebnis nicht wie ein Rechteck aus. (Wenn man dagegen ein Rechteck mit einem Geodreieck konstruiert, verwendet man andere Intuitionen. Hier ist vor allem von Bedeutung, dass gegenüber liegende Seiten parallel sind.)

[26] In manchen Details weicht die Turtle jedoch vom Objektkonzept ab. Insbesondere das „Dazulernen" ist untypisch für Objekte im Sinne des OOP. Hier sind Objekte in der Regel Instanzen von Klassen, in denen die Methoden bereits vollständig beschrieben sind (Balzert 1999)

Als Beispiel einer Mikrowelt, die primär nicht computerbasiert ist, sei auf die dynamischen Labyrinthe (DL) von Inge Schwank hingewiesen (Schwank 2003, 2005). Die Bezeichnung Labyrinth ist allerdings etwas irreführend, denn es gibt keine Wegegabelungen mit Entscheidungsfreiheit und man kann man sich nicht verlaufen. Im Gegenteil: Es ist sogar ein wichtiges Strukturmerkmal, dass der Weg von Start zum Ziel determiniert ist. In einem DL werden verschiedene informatische Intuitionen verwendet.

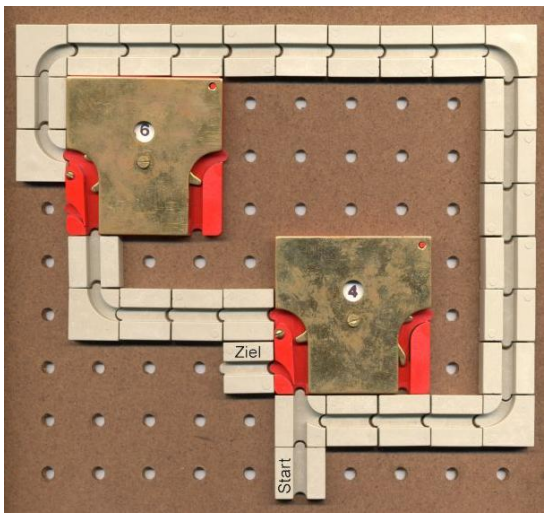


Abb. 67: Dynamisches Labyrinth (Schwank 2005), das eine Subtraktion (6−4) repräsentiert. Quelle: www.ikm.uni-osnabrueck.de/aktivitaeten/dl/dynamische-labyrinthe.htm

Abb. 67 stellt eine Maschine dar, die von einer größeren Zahl eine kleinere abziehen kann. Ihre Arbeitsweise kann durch folgendes Python-Programm beschrieben werden:

```
x = 6
y = 4
while y > 0:
    y -= 1
    x -= 1
print x
```

Zuerst stellt der Spieler an den beiden Zählern die Operanden der Subtraktion ein. Das Display des oberen Zählers zeigt den Wert des Subtrahenden x und das Display des unteren Zählers den Subtraktor. Dann setzt er oder sie einen Stab auf die Startposition und führt ihn solange – stets in der gleichen Richtung – durch die Rinne bis das Ziel erreicht ist. Die runden Nuten und Federn der Rinnenbausteine sind als Richtungspfeile zu interpretieren. Immer, wenn der Stab einen Zähler passiert, wird er um eins herabgesetzt. Die Zähler sind mit einer Weiche gekoppelt. Sie wird nach links gestellt, wenn der zugehörige Zähler auf null steht. In diesem Fall durchläuft der Führungsstab eine Schleife bis y gleich null ist und gelangt dann zum Ziel.

Dieses DL repräsentiert eine Reihe von intuitiven Modellen der Informatik. Am augenfälligsten ist die Bahn, die zusammen mit der Weiche die Intuition des Kontrollflusses eines Programmlaufs visualisiert. Die Bahn determiniert, in welcher Reihenfolge Anweisungen ausgeführt werden.

Das Zähler-Bauteil repräsentiert in Kombination gleich mehrere intuitive Konzepte. Es stellt eine Variable dar, deren Wert modulo n inkrementiert oder dekrementiert werden kann – je nachdem welches der beiden Rinnensegmente verwendet wird (links dekrementieren, rechts inkrementieren). Das linke Rinnensegment enthält eine Weiche, die den Weg nach links führt, wenn der Zähler null anzeigt.

Der Charme des DL liegt darin, dass die mechanischen Bauteile in ihrer Arbeitsweise vollkommen transparent sind und bis auf den Grund durchschaut werden können. Dagegen enthalten Mikrowelten in Form multimedialer Software „magische Elemente“, deren Arbeitsweise versteckt ist.

1.6 Prototypische Beispiele

Es gibt Fälle, in denen bereits ein einziges Beispiel zur Repräsentation eines Konzeptes ausreicht. In site@school, einem Content-Management-System für Grundschulen, wird in einem einzigen Beispiel erklärt, wie man einen Kalendereintrag vornimmt. Auf jede abstrakte Erklärung im Sinne einer Gebrauchsanweisung wird verzichtet.

Abb. 68: Beispiel für die Spezifikation eines Kalendereintrags bei site@school

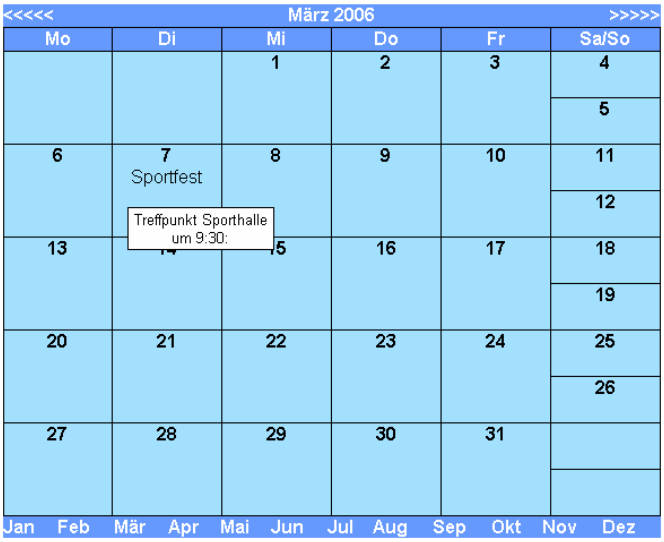


Abb. 69: Ansicht des Kalenderblattes nach Spezifikation wie im Beispiel aus Abb. 68

Probiert man das Beispiel aus und betrachtet die Wirkung auf das Kalenderblatt (Monatsübersicht), wird die Bedeutung offensichtlich. Das Beispiel hat den Charakter eines Prototyps. Die Abstraktion zu einer allgemeingültigen Anleitung kann vom Leser vorgenommen werden. Sie lautet in diesem Fall etwa folgendermaßen:

Die erste Zahl vor dem Unterstrich repräsentiert den Tag des Monats. Die anschließende Zeichenkette bis zum Komma ist der Text, der in der Monatsübersicht (Kalenderblatt) erscheint, und der dann folgende Text wird in einer Box sichtbar, wenn man mit der Maus den Cursor auf das Feld im Kalenderblatt bewegt. Dieser Text ist erheblich komplexer und vermutlich schlechter zu behalten, als das prototypische Beispiel.

Eine besondere Rolle spielen Beispiele in der Tabellenkalkulation. Nehmen wir an, wir haben eine Kalkulationstabelle mit drei Spalten. Die ersten beiden Spalten enthalten irgendwelche Zahlen. Die dritte Spalte soll in jeder Zeile die Summe der der beiden links daneben stehenden Zahlen enthalten.

Dazu gibt ein geübter Anwender der Tabellenkalkulation in die oberste Zelle der Summenspalte eine Formel ein – in diesem Fall in Zelle C2 die Formel

```
= A2 + B2.
```

Diese Formel kopiert sie oder er anschließend in die anderen Zellen der Summenspalte. In der Kalkulationstabelle erscheint jeweils das Ergebnis der Rechnung. Mit `A2` und `B2` werden die Zellen referiert, die die Summanden enthalten. Der Clou der Tabellenkalkulation ist nun, dass die Namen `A2` und `B2` relative Adressen spezifizieren. Gemeint sind eigentlich nicht die Zellen mit diesen Namen sondern die beiden Zellen links neben `C2`, also der Zelle mit der Formel. Die Bedeutung der Formel ist eigentlich:

Addiere zum Inhalt der Zelle, die sich in der gleichen Zeile und zwei Spalten links neben der aktuellen Zelle befindet, den Inhalt der Zelle, die sich in der gleichen Zeile und eine Spalte links neben der aktuellen Zelle befindet.

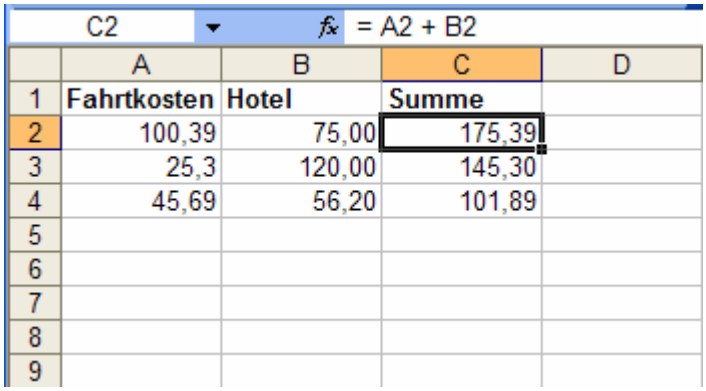


Abb. 70: Formel in einer Kalkulationstabelle

Beim Kopieren wird diese abstrakte Version der Formel übertragen und dann in der Zielzelle wieder mit konkreten Referenzen wiedergegeben. In der Zelle C3 erscheint z.B.

```
= A3 + B3
```

Der entscheidende Punkt ist, dass ein Anwender die Formel mit relativen Adressen nicht abstrakt, sondern als Beispiel mit konkreten Zellennamen definiert. Nun hat man bei jeder Beispielbildung im üblichen Sinne die Freiheit, aus mehreren (eventuell unendlich vielen) Konkretisierungsmöglichkeiten eine auszuwählen. Im Fall der Tabellenkalkulation liegt die Freiheit allein darin, aus der Menge von Zellen, in denen eine gleichartige Rechnung ausgeführt werden soll, eine einzige für die exemplarische Formulierung der Formel auszuwählen. In die anderen Zellen wird dann die Formel (in ihrer abstrakten Bedeutung) kopiert.

1.7 Beispiele für ablauforientierte Repräsentationen

Grafiken, die durch eine rekursive Prozedur erzeugt worden sind, können selbst als Protokoll des Programmlaufs interpretiert werden und ein Modell der Prozedur darstellen. Abb. 71 zeigt die Bildschirmausgabe des folgenden Python-Programms, das Turtle-Funktionen verwendet:

```
from turtle import *

def baum (n):
    if n >1:
        forward (n)
        left (60)
        baum(n/2)
        right (120)
        baum (n/2)
        left (60)
```

```
        backward(n)

baum(100)
```

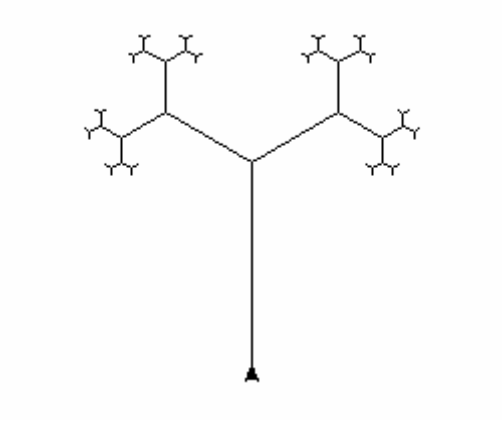


Abb. 71: Binärbaum, der durch eine Turtle-Prozedur generiert worden ist

Aus der Grafik kann die Idee der rekursiven Prozedur „abgelesen“ werden. Der Baum besteht aus einem Stamm an dessen oberem Ende je ein etwas kleinerer Baum nach schräg links und schräg rechts abgeht. Die Turtle muss also zuerst den Stamm (Linie) zeichnen, sich dann z.B. nach links drehen, einen kleineren Baum zeichnen (rekursiver Aufruf), dann nach rechts drehen wieder einen kleineren Baum zeichnen (zweiter rekursiver Aufruf) und schließlich wieder an den Ausgangspunkt zurückkehren. Der Baum ist sozusagen eine „Spur“ der rekursiven Prozeduraufrufe und dokumentiert die Arbeitsweise des Programms.

1.8 Darstellung intuitiver Modelle in der Python Visual Sandbox

1.8.1 Warum Animationen?

Oft ist ein statisches Bild wenig aussagekräftig. Bestimmte Features eines Modells werden erst sichtbar, wenn man das Modell in Aktion sieht. Zum Beispiel darf im Behältermodell für Variablen ein Behälter nur einen einzigen Zettel mit einer zusammenhängenden Dateneinheit (z.B. Zahl) enthalten. Dieser Aspekt kann durch die Abarbeitung zweier Anweisungen der Form

```
a = 1
a = 2
```

sichtbar gemacht werden. Bei der zweiten Zuweisung wird der vorige Inhalt des Behälters zerstört.

Ziel der PVS ist, Detailaspekte intuitiver Vorstellungen sichtbar zu machen. Ein Problem jeder Analogie oder Metapher ist, dass nur einige Aspekte auf den Zielbereich („das Gemeinte“) übertragen werden können und andere nicht. Fehlvorstellungen entstehen häufig wenn die Grenze der Anwendbarkeit überschritten wird bzw. Einschränkungen nicht erkannt werden. So ist das Behältermodell nur unter der Einschränkung brauchbar, dass der Behälter nur ein Objekt enthalten kann. Reale Behälter können dagegen können viele Objekte enthalten. Um Vorstellungen über die Grenzen besser ausloten zu können, werden Visualisierungen intuitiver Modelle in einem detailreichen Kontext verwendet.

Somit ist eine Animation der PVS in der Regel nicht die Repräsentation einer Intuition sondern sie enthält häufig mehrere solche Repräsentationen in einem relativ kleinen mehr oder weniger abgeschlossenen Sinnzusammenhang.

1.8.2 Entfernung und Nähe

Die Animationen der PVS beziehen sich auf Programmtexte. Im Python Puzzle stellen sie die Arbeitsweise des zu erstellenden Programms dar. In Python Visual und Python Quiz beziehen sie sich auf gegebene Programme oder Programmfragmente.

Ein äußerliches Merkmal einer Visualisierung in der PVS ist die Nähe zum Bezugsprogrammtext. Eine programmnahe Visualisierung enthält viele explizite Bezüge zum Programmtext. Tab. 25 führt einige Merkmale auf.

Kriterium	**nah**	**fern**
Daten	Die konkreten Daten des Programms (z.B. Zahlen) tauchen auch in der Visualisierung auf (z.B. Zettel mit Zahlen)	Daten werden abstrakt dargestellt z.B. zu sortierende Objekte durch Kästen oder gegenständliche Objekte
Namen	Namen aus dem Programmtextes (z.B. Variablennamen) werden auch in der Visualisierung verwendet (z.B. Etiketten)	Namen aus dem Programmtextes tauchen nicht auf
Struktur	Die Visualisierung hat die gleiche Struktur wie das Programm.	Die Visualisierung stellt nur die Kernidee des Programms dar und lässt unwichtige Details weg.

Tab. 25: Merkmale naher und entfernter Visualisierungen von Programmen

Als Maß für die Entfernung verwenden wir in der Dokumentation der Modelle den Anteil an Namen aus dem Bezugsprogramm in Prozent. Zu den Namen werden Namen von Funktionen (einschließlich Operatorsymbole), Klassen, Objekten, Variablen gezählt, nicht aber Schlüsselwörter wie `if`, `else` oder Literale.

1.8.3 Grafische Elemente der Python Visual Sandbox

Nun sind die Möglichkeiten einen Sachverhalt bildhaft darzustellen praktisch unbegrenzt, weil es keine allgemeinverbindliche Syntax gibt. Um die Verständlichkeit zu verbessern, wurde für die PVS eine Art Bildersprache entwickelt und die Visualisierung zu einem großen Teil normiert. Die Animationen enthalten wieder kehrende grafische Elemente, deren Bedeutung zum Teil aus dem Alltag bekannt ist oder aus dem Zusammenhang erschlossen werden kann. Einheitliche Formen und Farben erleichtern das Wiedererkennen. Bei Workshops mit der PVS erklärten Teilnehmer allerdings, dass sie etwas Zeit brauchten, um die Bedeutung der Abbildungen zu verstehen und zu lernen.

Behälter (dreidimensionaler Kasten)

Ein einfacher dreidimensionaler Kasten stellt eine Variable als Behälter dar, die Daten aufnehmen kann. Inhalt ist meist ein beschrifteter Zettel.

Lange Kästen mit mehreren Fächern repräsentieren Listen. Manchmal sind die Fächer mit Nummern (0, 1, 2, …) beschriftet (Listenindexe). Sonderfälle sind Kästen mit einem Fach für einelementige Listen, die mit normalen Variablen verwechselt werden können, sowie Darstellungen leerer Listen. Eine leere Liste kann durch ein einfaches Brett visualisiert werden, auf dessen Schmalseite man blickt (Kasten mit null Fächern).

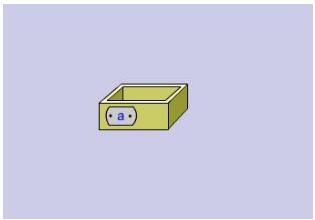


Abb. 72: Ein Kasten mit Schild repräsentiert eine Variable als Behälter für Daten.

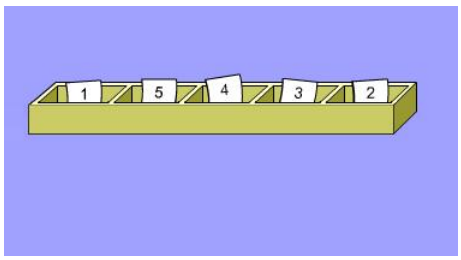


Abb. 73: Kasten mit mehreren Fächern zur Darstellung einer Python-Liste

Karten (Zettel)

Auf weißen beweglichen Zetteln werden Daten (Zahlen oder Zeichenketten) mit schwarzer Schrift wiedergegeben. Sie bewegen sich manchmal von alleine oder werden durch Greifarme transportiert.

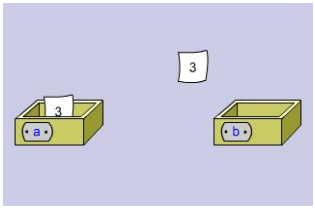


Abb. 74: Beschriftete Zettel repräsentieren Daten

Ovale für Botschaften

Beschriftete Ovale repräsentieren Botschaften, die an Objekte geschickt werden und Aktionen auslösen. Im Unterschied zu Zetteln enthalten sie zumindest die Angabe der Methode, die vom Empfänger ausgeführt werden soll.

Abb. 75: Bewegliches Oval, das eine Botschaft an das Objekt `bottle` repräsentiert

Namenszettel

Namen für Objekte werden gelegentlich durch Klebezettel (Zettel mit einem Stück Klebeband) an die benannten Objekte geheftet. Bei dieser Zuordnung eines Namens zu einem Objekt fehlt dann komplett der Behälter. Es kommt aber auch vor, dass ein Namenszettel an einen Behälter für ein Objekt geklebt wird. Inkonsistent ist, wenn ein Namenszettel an einem anderen Namenszettel befestigt wird. Denn ein Name ist kein Objekt.

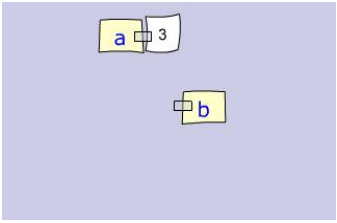


Abb. 76: Haftzettel als Namen für Objekte

Stecknadeln

Iterationen werden manchmal in der Weise visualisiert, dass eine Stecknadel von Fach zu Fach eines Behälters mit mehreren Fächern fliegt. Die Stecknadel markiert dann das aktuelle Element der

Iteration über eine Sequenz. Stecknadeln repräsentieren also Namen für Objekte. Manchmal hängt an ihnen ein kleiner Zettel mit einer konkreten Namensbezeichnung (z.B. `i` oder `person`).

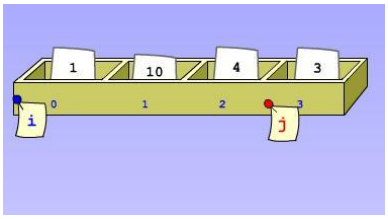


Abb. 77: Stecknadeln mit Schildern repräsentieren Referenzen auf Elemente einer Liste

Zeiger

Graue, transparente sich zu einem Endpunkt verjüngende Linien, stellen den Bezug zwischen einem Namen und dem benannten Objekt her (Zeiger). Der Name befindet sich am dickeren Ende der Linie. Die Zeiger sind häufig an der Spitze beweglich und können zunächst auf ein Objekt und dann auf ein anderes zeigen. Es stellte sich heraus, dass einige Nutzer der PVS diese Linie zunächst nicht als Pfeil mit Start und Endpunkt sondern als Linie in dreidimensionaler Darstellung interpretierten. Das dickere Ende befindet sich im Vordergrund und das dünnere im Hintergrund.

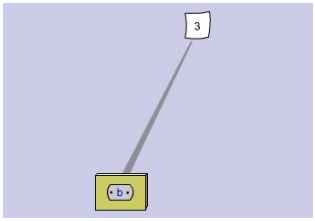


Abb. 78: Zeiger

Namensschilder

Behälter in den Animationen der PVS tragen meist Namensschilder. Sie sehen immer gleich aus: Graue Rechtecke mit abgerundeten Seiten und Bohrlöchern zum Festschrauben. Sie sind optisch leicht von Daten (Zetteln) zu unterscheiden. Manchmal bewegen sich auch Namensschilder. Zum Beispiel kann an einen Behälter ein zweites Namensschild angebracht werden um eine Zuweisung der Form `a = b` zu erklären. Dagegen ist es inkonsistent, wenn ein Behälter ein Namensschild als Inhalt aufnimmt.

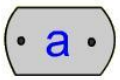


Abb. 79: Namensschild

Bretter

Sequenzen werden häufig durch flache Quader dargestellt (Bretter). Auf ihrer Oberfläche sind helle Bereiche aufgetragen, die die Listenelemente visualisieren. In diesen Bereichen befinden sich Zettel mit Daten oder graue Punkte, von denen ein Zeiger ausgeht.

Greifer, Manipulatoren

In manchen Animationen wird die Frage thematisiert, welche Entitäten in der Maschinerie eines laufenden Programms eigentlich aktiv sind. Aktivität wird durch Greifarme mit Gelenken visualisiert, die andere Entitäten (meist Zettel mit Daten) bewegen.

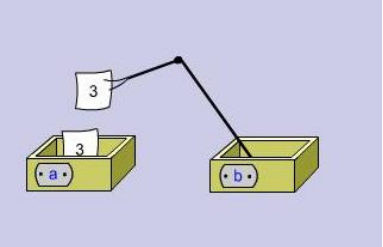


Abb. 80: Greifer repräsentieren Aktivität einer Entität

Funktionsboxen mit Ein- und Ausgang

Ein Ein/Ausgabe-orientiertes Modell einer Funktion wird in der PVS durch einen hohlen Pyramidenstumpf mit quadratischem Querschnitt dargestellt, der sich nach unten verjüngt (Funktionsbox). Die obere (größere) Öffnung ist der Eingang. Sobald dieses Gerät genügend Daten empfangen hat, wackelt es ein paar Mal (um interne Aktivität anzudeuten) und gibt dann über die untere Öffnung ein Ergebnis aus. In einigen Fällen (bei rekursiven Funktionen) besitzen die Funktionsboxen seitliche Öffnungen, um die Datenübergabe in internen Funktionsaufrufen darstellen zu können.

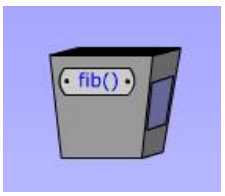


Abb. 81: Darstellung einer Funktion als Box mit Eingang (oben) und Ausgang (unten).

Blitze

Ereignisse wie die Ausführung einer Operation werden manchmal durch „Explosionen“ mit einer Art Blitzerscheinung (heller, sich rasch ausbreitender und dann wieder verschwindender Lichtfleck) visualisiert. Der Blitz tritt an einer oder mehreren Entitäten (meist Zettel) auf und führt zu einer Veränderung. Manchmal sind die Blitze mit dem Namen der ausgeführten Operation beschriftet. Beispiel: Zwei Zettel mit Zahlen fliegen aufeinander zu, bis sie sich berühren. Es erscheint ein Blitz mit einem Pluszeichen. Anschließend sind die beiden Zettel verschwunden und statt dessen sieht man einen neuen Zettel mit der Summe der beiden Zahlen.

Weitere grafische Elemente

Nicht alle grafischen Elemente der PVS sind standardisiert. So sind Boxen ein relativ universelles grafisches Element und werden zur Darstellung unterschiedlicher Arten von Entitäten verwendet. Eine Box kann einen Konstruktor repräsentieren, der ein Objekt generiert. Boxen stellen manchmal Funktionen dar, die Werte über „Sensoren“ abtasten und ein Ergebnis liefern.

Sensoren werden meist durch keilartige sich zu einer Seite verjüngende Linien (wie Zeiger) dargestellt. Sie tasten Werte ab und blinken manchmal, wenn sie aktiv sind. Gelegentlich werden aber auch Greifer mit Gelenken und andere Bilder zur Darstellung von Sensoren verwendet.

Darüber hinaus gibt es eine Reihe naturalistische Bildelemente wie Gummibärchen, Drehscheiben mit Zahlen, Vasen oder Gesichter.

1.9 Verwendung von Visualisierungen im Informatikunterricht

Im Zeitraum von Oktober 2005 bis zum August 2006 wurden 3 Lehrerinnen und 17 Lehrer zur Verwendung von Visualisierungen im Informatikunterricht befragt. Die befragten Personen hatten im

Mittel 7.5 Jahre Unterrichtserfahrung im Fach Informatik (Standardabweichung 7.44 Jahre, Median 5.5 Jahre). Zehn Personen besaßen eine staatlich kontrollierte Zusatzqualifikation (außer Staatsexamen) und neun Personen ein Staatsexamen im Fach Informatik. Eine Person war Autodidakt.

Die folgenden Abbildungen zeigen den Fragebogen.

Umfrage zu Visualisierungen im Informatikunterricht

Alle personenbezogenen Angaben werden vertraulich behandelt und nicht an Dritte weitergegeben. Das Ergebnis der Umfrage sende ich Ihnen im September 2006 zu.

Herzlichen Dank fürs Mitmachen!
Michael Weigend

Zur Person

Geschlecht

○ männlich
○ weiblich

Informatik-Ausbildung

Welche Aussage trifft am ehesten auf Sie zu?

○ Ich habe alle Programmierkenntnisse autodidaktisch erworben und keinerlei organisierte Ausbildung oder Weiterbildung erfahren.

○ Ich habe einen oder mehrere Lehrgänge (z.B. Lehrerweiterbildung, Zertifikatskurse, Kurse in einem Universitätsstudium) zur Informatik absolviert aber keinen Hochschulabschluss.

○ Ich habe ein erstes Staatsexamen (oder Erweiterungsprüfung) in Informatik abgelegt.

Seit wie vielen Jahren unterrichten Sie Informatik?

Seit ☐ Jahren

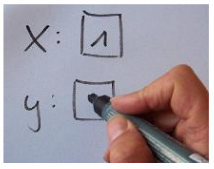
E-Mail-Adresse: ☐

Zur Verwendung von Visualisierungen

Wie häufig verwenden Sie im Unterricht - z.B. in Tafelbildern, Arbeitsblättern, gegenständlichen Lernhilfen, Lehrbuchabbildungen, Rollenspielen - die nachfolgend aufgeführten bildlichen Darstellungen?

1 Behälter mit Etikett zur Darstellung von Variablen (Kästen, Marmeladengläser etc.)
○ bisher noch nie
○ bisher insgesamt ein oder zwei Mal
○ gelegentlich
○ häufig (aber nicht immer, wenn es passen könnte)
○ praktisch immer, wenn es zum aktuellen Unterrichtsstoff passt

2 Kästchen an der Tafel als "Behälter" für Daten
○ bisher noch nie
○ bisher insgesamt ein oder zwei Mal
○ gelegentlich
○ häufig (aber nicht immer, wenn es passen könnte)
○ praktisch immer, wenn es zum aktuellen Unterrichtsstoff passt



3 Wertetabellen für Variablen
○ bisher noch nie
○ bisher insgesamt ein oder zwei Mal
○ gelegentlich
○ häufig (aber nicht immer, wenn es passen könnte)
○ praktisch immer, wenn es zum aktuellen Unterrichtsstoff passt

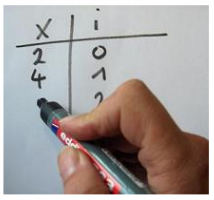
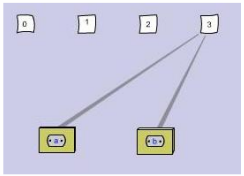


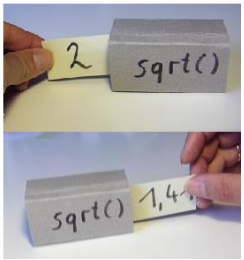
Abb. 82: Fragebogen zur Verwendung von Visualisierungen (1)

4 Behälter mit Fächern zur Veranschaulichung einer Liste bzw. eines Arrays
○ bisher noch nie
○ bisher insgesamt ein oder zwei Mal
○ gelegentlich
○ häufig (aber nicht immer, wenn es passen könnte)
○ praktisch immer, wenn es zum aktuellen Unterrichtsstoff passt

5 Zeiger oder Linie zur Darstellung der Verbindung eines Namens zu einem benannten Objekt oder Datum
○ bisher noch nie
○ bisher insgesamt ein oder zwei Mal
○ gelegentlich
○ häufig (aber nicht immer, wenn es passen könnte)
○ praktisch immer, wenn es zum aktuellen Unterrichtsstoff passt



6 Box mit Ein- und Ausgang zur Visualiserung einer Funktion, die einen Wert oder ein Objekt über ihren Eingang aufnimmt, verarbeitet und ein Ergebnis über den Ausgang ausgibt.
○ bisher noch nie
○ bisher insgesamt ein oder zwei Mal
○ gelegentlich
○ häufig (aber nicht immer, wenn es passen könnte)
○ praktisch immer, wenn es zum aktuellen Unterrichtsstoff passt



7 Aktive Entitäten (z.B. Schüler in einem Rollenspiel), die etwas in Empfang nehmen und/oder einem anderen Akteur geben können.
○ bisher noch nie
○ bisher insgesamt ein oder zwei Mal
○ gelegentlich
○ häufig (aber nicht immer, wenn es passen könnte)
○ praktisch immer, wenn es zum aktuellen Unterrichtsstoff passt

8 Bewegliche Zettel, auf denen etwas geschrieben steht, zur Darstellung von Datenflüssen.
○ bisher noch nie
○ bisher insgesamt ein oder zwei Mal
○ gelegentlich
○ häufig (aber nicht immer, wenn es passen könnte)
○ praktisch immer, wenn es zum aktuellen Unterrichtsstoff passt

9 Bewegliche Zettel zur Visualisierung von Botschaften.
○ bisher noch nie
○ bisher insgesamt ein oder zwei Mal
○ gelegentlich
○ häufig (aber nicht immer, wenn es passen könnte)
○ praktisch immer, wenn es zum aktuellen Unterrichtsstoff passt

10 Haftzettel (Zettel mit Klebestreifen, Postits etc.), auf denen ein Name steht und die an ein Objekt geklebt werden, um es zu benennen.
○ bisher noch nie
○ bisher insgesamt ein oder zwei Mal
○ gelegentlich
○ häufig (aber nicht immer, wenn es passen könnte)
○ praktisch immer, wenn es zum aktuellen Unterrichtsstoff passt

Abb. 83: Fragebogen zur Verwendung von Visualisierungen (2)

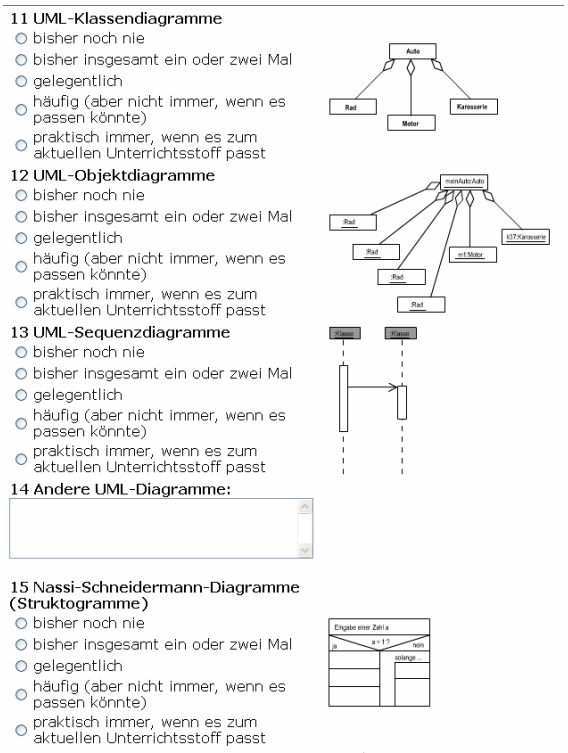


Abb. 84: Fragebogen zur Verwendung von Visualisierungen (3)

n = 20	nie	selten	gelegentlich	häufig	immer	Mittelwert
Behälter mit Etikett	17	0	3	2	3	1,20
Kästchen mit variablem Inhalt	4	2	6	3	5	2,15
Wertetabellen	1	2	3	5	9	2,95
Liste als Behälter mit Fächern	15	0	1	2	2	0,80
Zeiger	4	0	5	8	3	2,30
Box mit Ein- und Ausgang für Funktion	13	1	3	0	3	0,95
Aktive Entitäten	6	5	5	1	3	1,50
Bewegliche Zettel für Daten	12	2	4	0	2	0,90
Bewegliche Zettel für Botschaften	19	0	1	0	0	0,10
Haftzettel zur Benennung	13	3	3	1	0	0,60
UML-Klassendiagramme	5	1	4	3	7	2,30
UML-Objektdiagramme	8	0	3	3	6	1,95
UML-Interaktionsdiagramme	16	0	4	0	0	0,40
Nassi-Schneidermann-Diagramme	4	4	7	3	2	1,75
Flussdiagramme	2	6	11	0	1	1,60
Zustandsübergangsdiagramme	13	2	1	2	2	0,90

Tab. 26: Ergebnis der Umfrage zur Verwendung von Visualisierungen

2 Ergänzungen zur Verwendung intuitiver Modelle

2.1 Verstehen

2.1.1 Textformen in informatischer Fachliteratur

Die klassische Hermeneutik bezieht sich vor allem auf die Interpretation theologischer oder philosophischer Texte. Im Bereich der Informatik haben wir es vor allem mit Gebrauchstexten zu tun, die Konzepte technischer Systeme erklären. Ein typisches Beispiel sind Beschreibungen in einer Sprachreferenz, z.B. die Darstellung der Wirkungsweise einer Standardfunktion. Form und Abstraktionsgrad informatischer Texte können sehr unterschiedlich sein.

Nehmen wir als Beispiel verschiedene Darstellungen der Wirkungsweise der Python-Funktion `len()`, die wir hier der Einfachheit halber auf Listen einschränken.

Definition durch einen erklärenden Text

Die Funktion `len()` akzeptiert eine beliebige Liste als Argument und gibt deren Länge d.h. die Anzahl der enthaltenen Elemente zurück.

Definition durch Axiome

(4) len([]) == 0

(5) Sei a ein beliebiges Objekt. Dann ist `len([a]) == 1`.

(6) Wenn `s1` und `s2` beliebige Listen sind, dann ist `len(s1+s2) = len(s1) + len(s2)`. Dabei ist `s1+s2` die Konkatenation der beiden Listen `s1` und `s2`.

Definition durch einen Programmtext

```
def len(s):
    if s == []:
```

```
        return 0
    else:
        return len(s[1:])+1
```

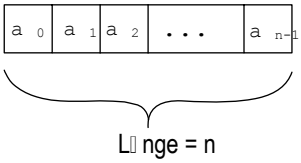
Definition durch Beispiele

Seien a1, a2, ... beliebige Objekte. Dann gilt

```
len([]) == 0
len([a1]) == 1
len ([a1, a2]) == 2
len ([a2, a2, a3]) == 3
```

usw.

Visuelle Darstellung



2.1.2 Experimente zur Beantwortung erkenntnisgewinnender Fragen

Verständnisfragen zu einem Text können häufig durch kleine Experimente am Computer beantwortet werden. Python unterstützt durch den interaktiven Modus diese Art der „Erkundung von Bedeutung“. Zur Illustration folgt eine denkbare Fragesequenz zum Thema Listen:

Textgrundlage: Die Funktion len() liefert die Länge einer Liste, d.h. die Anzahl der enthaltenen Elemente.

Frage 1: Ist bei verschachtelten Listen (Listen von Listen) die Länge die Anzahl der insgesamt vorkommenden Objekte?

Experiment 1.1: Wie groß ist z.B. die Länge der Liste `[[1, 2], [3, 4, 5]]`?

```
>>> len ([[1, 2], [3, 4, 5]])
```

2

Experiment 1.2: Wie groß ist z.B. die Länge der Liste `[[]]`? Sie enthält doch eigentlich nichts.

```
>>> len ([[]])
1
```

Frage 2: Vergleicht der Operator > die Längen zweier Listen?

Experiment 2:

Ist eine Liste mit drei Elementen immer „größer“ als eine Liste mit zwei Elementen?

```
>>> [1, 2, 3] > [4, 2]
False
```

Frage 3: Wie kann ich zwei Listen hinsichtlich ihrer Länge vergleichen?

Experiment 3: Man könnte die Längenfunktion verwenden.

```
>>> len([1, 2, 3]) > len ([4, 2])
True
```

Frage 4: Kann ich bei einem Aufruf der Längenfunktion auch einen Ausdruck mit der Konkatenation zweier Listen als Argument übergeben?

Experiment 4:

```
>>> len ([1, 2] + [3, 4])
4
```

Frage 5: Kann man die Längenfunktion auch zur Analyse von Texten mit regulären Ausdrücken verwenden?

Experiment 5: Vorkommenshäufigkeit als Länge einer Liste

```
>>> len (re.findall("[aeiou]", "Die Sonne scheint"))
6
```

Fragen heißt immer, die im Text mitgeteilten neuen Konzepte mit bereits gelernten Konzepten in Beziehung zu setzen. Die gefundenen Antworten auf Fragen an den Text können als Grundlage für intuitive Modelle dienen. Antworten auf selbst gestellte Fragen sind mit großer subjektiver Gewissheit verbunden, vor allem dann, wenn sie durch Experimente gewonnen wurden.

2.2 Das Bemühen um Verstehen bei der Vorbereitung auf einen Test

Es gibt viele Situationen, in denen selbstgesteuerte Verständnisgewinnung stattfindet. Eine Standardsituation im Schulalltag ist die Vorbereitung auf einen Test. In diesem Abschnitt werden die Ergebnisse einer Befragung von 29 Schülerinnen und Schülern eines Informatikkurses der Jahrgangsstufe 12 diskutiert.

Es geht um folgende Fragen:

- Welche Art von Beispielen wählen Schüler als intuitive Modelle, um sich ein abstraktes Konzept zu merken?
- Welche Rolle spielt das praktische Ausprobieren von Beispielen am Computer?
- Welche Merkmale haben Metaphern, die zur Veranschaulichung eines abstrakten Konzeptes von Schülern gewählt werden?

Die Schülerinnen und Schüler erhielten ein vierseitiges „Selbstlernmaterial“ zum Thema „reguläre Ausdrücke“ mit Texten, Animationen und Anregungen zum Ausprobieren am Computer. Für die Bearbeitung hatten sie 90 Minuten Zeit. Wie sie mit dem Material umgingen war ihnen freigestellt. Es wurde aber deutlich gemacht, dass der Sinn der Übung eine Vorbereitung auf den nächsten Test sein sollte. Zusammen mit dem Selbstlernmaterial bekamen die Schüler einen Fragebogen, in dem nachgefragt wurde, in welcher Weise sie die in dem Text vorkommenden Beispiele verwendet haben.

Das Lernmaterial zu regulären Ausdrücken enthält folgende Komponenten:

- Erklärende Texte wie in einer Sprachreferenz und eine Concept Map, in der wichtige Begriffe im Zusammenhang dargestellt werden. Sie beschreiben das Wissen, das im Test geprüft werden soll.
- Vollständige Beispiele, die in Gedanken nachvollzogen aber nicht am Computer ausprobiert werden können. Dazu gehören Beispiele für Sprachen regulärer Ausdrücke: `L("a") = {"a"}`
- `L(".aus") = {"Maus", "Laus", "Haus", ...}`
- Vollständige Beispiele, die in Gedanken nachvollzogen und am Computer ausprobiert werden können. Dazu gehören Aufrufe der Python-Funktion findall() mit verschiedenen Parametern. Derartige Beispiele beschreiben die Semantik sowohl der Funktion findall() als auch der verwendeten regulären Ausdrücken.
- Unvollständige Beispiele, die durch Nachdenken oder eigenes Ausprobieren vervollständigt werden können. Von dieser Art sind Aufrufe der Python-Funktion findall() mit verschiedenen Argumenten, wobei die zurückgegebenen Werte jedoch nicht verraten werden. Diese kann man sich selbst überlegen oder durch Ausprobieren im interaktiven Modus des Python-Interpreters herausfinden.
- Animationen, die das zentrale Konzept (z.B. reguläre Ausdrücke) durch unterschiedliche Metaphern veranschaulichen.

2.2.1 Auswahl repräsentativer Beispiele

Bezugnehmend auf den Abschnitt über Sprachen regulärer Ausdrücke im Lernmaterial wurde den Schülern folgende Frage gestellt:

„Welche der folgenden Beispiele des Arbeitsmaterials werden Sie sich besonders gut merken, um in Erinnerung zu behalten, was die Sprache eines regulären Ausdrucks ist?“

```
L1   L("a") = {"a"}
L2   L("Haus") = {"Haus"}
L3   L("a.") = {"aa", "ab", "ac", ...}
L4   L(".aus") = {"Haus", "Maus", "Laus", ...}
L5   L("a+") = {"a", "aa", "aaa", ...}
L6   L("Mu+h") = {"Muh", "Muuh", "Muuuh", ...}
L7   L("a*") = {"", "a", "aa", "aaa", ...}
L8   L("0*1") = {"1", "01", "001", "0001", ...}
L9   L("G[lr]as") = {"Glas", "Gras"}
L10  L("Glas|Gras") = {"Glas", "Gras"}
L11  L("In.*") = {"In", "Indianer", "Insel","Intuition", ... }
L12  L(".*\d.*") = {"0", "1000 EUR", "456", ... }
L13  L(".*[Bb]all") = {"ball", "Ball", "Handball", ... }
```

Analysieren wir zunächst die Beispiele. Sie unterscheiden sich in mehrfacher Hinsicht:

Konkretisierungsgrad. Unter Konkretisierungsgrad verstehen wir das Ausmaß, in dem Beispiele Objekte oder sensorisch erfahrbare Situationen die Realität wiedergeben. Manche Beispiele verwenden sowohl im regulären Ausdruck `re` als auch bei der Darstellung der Menge `L(re)`, die durch den regulären Ausdruck definiert wird, sinnvolle Texte. In den Beispielen `L2` und `L10` sind (abgesehen von Metazeichen) alle vorkommenden Zeichenketten sinnvolle Begriffe (Haus, Gras, Glas). Bei anderen Beispielen (`L4`, `L11`, `L13`) sind zumindest die in `L(re)` aufgeführten Zeichenketten Wörter der deutschen Sprache. All diese Beispiele sind also relativ konkret. In einigen Fällen kommt hinzu, dass das Beispiel so interpretiert werden kann, dass es eine Situation wiedergibt, die aus dem Alltag bekannt ist. Im Falle von regulären Ausdrücken handelt es sich um Situationen, in denen es um Sprache geht. So stellt `L5` eine Menge von Wörtern dar, die sich aufeinander reimen. Das „Thema“ von `L8` ist die korrekte Schreibweise der Zahl `1` mit beliebiger Anzahl führender Nullen.

Abstrakten Beispielen dagegen fehlt der Bezug zur Realwelt. In den Mengen von `L3`, `L5`, `L7` werden Zeichenketten aufgezählt, ohne dass ein Zusammenhang zu realen „Spracherlebnissen“ erkennbar ist.

Definitorischer Charakter. Einige Beispiele sind quasi Definitionen für Aspekte der Semantik regulärer Ausdrücke. `L5` und `L7` stellen die Wirkungsweise des Plus- und Sternoperators dar. Die Elemente von `L(re)` werden systematisch nach einem bestimmten Verfahren (Wiederholung des Zeichens vor dem Operator) aufgezählt. Die Punkte in den Mengenklammern repräsentierten eine „Fortsetzung“ und signalisieren, dass die gesamte (unendliche) Menge nach dem angedeuteten Verfahren konstruiert werden kann. Das Zeichen a kann auch als Metabezeichner für einen beliebigen regulären Ausdruck interpretiert werden.

In anderen Fällen ist der definitorische Charakter des Beispiels geringer ausgeprägt. In `L4` werden in der Beispielmenge für den Punkt des regulären Ausdrucks `".aus"` irgendwelche Buchstaben eingesetzt, so dass sich ein sinnvolles Wort ergibt (`"Maus"`, `"Laus"` usw.) Die Auswahl des Zeichens geschieht aber nicht systematisch sondern willkürlich. Die drei Punkte in den Mengenklammern symbolisieren keine „Fortsetzung“ sondern nur die Aussage, dass die Menge noch weitere Elemente enthält, die aber aus Platzgründen nicht aufgeführt werden.

Komplexität. Als Maß für die Komplexität regulärer Ausdrücke betrachten wir folgende Merkmale:

- Anzahl der vorkommenden Zeichen. `"Haus"` ist komplexer als `"a"`.
- Anzahl der vorkommenden Sonderzeichen (`*`, `+`, `.`, `|`, `[`, `]`, `\d`, `\w`). `"G[lr]as"` ist komplexer als `"Haus"`.
- Verschiedenartigkeit der vorkommenden Sonderzeichen. `".*a.+"` ist komplexer als `".*a.*"`

Offensichtlich unterscheiden sich die regulären Ausdrücke in den Beispielen hinsichtlich der Komplexität.

Ergebnisse

Im Mittel entschieden sich die Kursteilnehmer für 4.7 Beispiele. Am häufigsten (7 Teilnehmer) wurden vier Beispiele ausgewählt. Zwei Schüler/innen gaben an, sich alle 13 Beispiele merken zu wollen. Werfen wir einen Blick auf die Auswahl der „Merkbeispiele“ (Abb. 85).

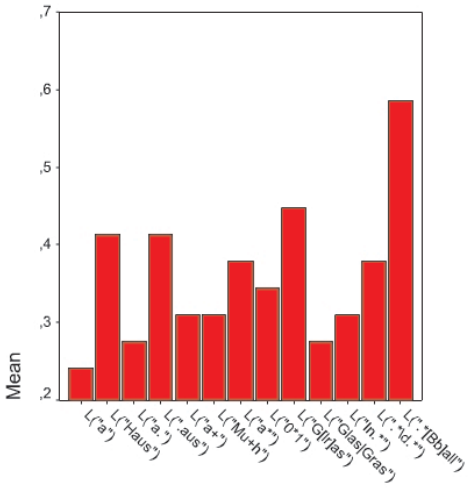


Abb. 85: Verteilung der gewählten Beispiele für reguläre Ausdrücke und ihre Sprachen (n=29)

Es zeigt sich, dass konkrete Beispiele gegenüber abstrakten Beispielen bevorzugt werden. Die vier am häufigsten gewählten Merkbeispiele kann man als konkret bezeichnen. Das konkrete Beispiel `L4` wurde fast doppelt so häufig gewählt wie das praktisch strukturgleiche abstrakte Beispiel `L3`. Dabei ist `L3` weniger komplex und hat einen stärkeren definitorischen Charakter. Beispiel `L6` weist allerdings die gleiche Häufigkeit auf wie sein abstraktes Pendant `L5`.

Die auffällige Dominanz des letzten Beispiels `L(".*[Bb]all")` kann eventuell damit zusammenhängen, dass die Animationen (s.u.) genau dieses Beispiel aufgegriffen haben.

2.2.2 Beispiele ausprobieren – Streben nach Gewissheit

Bei der Programmierung mit Python werden reguläre Ausdrücke als erstes Argument in einem Aufruf der Funktion `findall()` verwendet, um einen Text zu analysieren, der als zweites Argument übergeben wird. Der Aufruf `findall(re, text)` liefert eine Liste mit nicht überlappenden Teilstücken der Zeichenkette text, die auf den regulären Ausdruck `re` passen. Das Material zum Fragebogen enthielt vier Beispiele für Funktionsaufrufe zusammen mit dem Ergebnis, das die Funktion `findall()` liefert und 11 Beispiele ohne das Ergebnis. Außerdem gab es noch zwei kurze interaktive Programme, in denen eine Benutzereingabe mit Hilfe von `findall()` ausgewertet wurde.

2.2.3 Beispiele für die Verwendung regulärer Ausdrücke

Vollständige Beispiele für Funktionsaufrufe

`F1`

```
>>> findall("u", "Hallo")
[]
```

F2

```
>>> findall ("ab", "Aber abends habe ich Hunger")
['ab', 'ab']
```

F3

```
>>> findall ("G.as", "Ein Glas liegt im Gras.")
['Glas', 'Gras']
```

F4

```
>>> findall ("[Bb]\w*", "Bring mir bitte eine Birne.")
['Bring', 'bitte', 'Birne']
```

Unvollständige Beispiele für Funktionsaufrufe

W1

```
>>> findall("l", "Hallo")
```

W2

```
>>> findall("\d", "Notruf ist 110")
```

W3

```
>>> findall(".t", "Er hat Fett an den Fingern")
```

W4

```
>>> findall(".. ", "Wo ist mein Kuli?")
```

W5

```
>>> findall("", "Hallo")
```

W6

```
>>> findall(".+", "Meine Orange")
```

W7

```
>>> findall("B.*", "Die Birne ist durchgebrannt!")
```

W8

```
>>> findall("\d", "1, 2, 3 und 4")[0]
```

W9

```
>>> for i in findall("\d", "1, 2, 3 und 4"):
    print i
```

W10

```
>>> findall("te", "Er hatte Tee".lower())
```

W11

```
>>> len (findall("[l]", "hell"))
```

Skripte, in denen reguläre Ausdrücke verwendet werden

A1

```
eingabe = raw_input ("Noch Tee? ")
if findall("[jJ]", eingabe)!= []:
    print "Hier ist der Tee!"
```

A2

```
eingabe = raw_input (" :")
while findall ("cu|see you|bye", eingabe) == []:
    eingabe = raw_input (" :")
    if findall ("cu|see you|bye", eingabe) == []:
        print "Aha. Tell me more about it!"
```

```
else:
    print "Good bye"
```

2.2.4 Ergebnisse

Tab. 27 gibt einen Überblick, in wie vielen Fällen die Beispiele ausprobiert wurden.

Art des ausprobierten Beispiels	Mittelwert	Std.-Abweichung
Funktionsaufruf mit bekanntem Ergebnis (4 Beispiele)	1.4815	1.76222
Funktionsaufruf mit unbekanntem Ergebnis (11 Beispiele)	7.3704	3.97248
Interaktives Skript (2 Beispiele)	.8889	.80064
Selbst ausgedachtes Beispiele	1.0741	1.81714
Insgesamt ausprobiert	10.8148	4.99259

Tab. 27: Anzahlen der ausprobierten Beispiele (n=29)

Bemerkenswert sind folgende Beobachtungen:

- Etwa zwei Drittel der Funktionsaufrufe mit unbekanntem Ergebnis wurden am Computer ausprobiert (und so das Ergebnis gefunden). Dagegen wurden die interaktiven Skripte in erheblich geringerem Ausmaß getestet.
- Mehr als ein Drittel der vollständigen Beispiele (Funktionsaufruf mit bekanntem Ergebnis) wurden ausprobiert.

Offenbar spielte das praktische Ausprobieren eine große Rolle beim Bemühen um das sichere Verstehen regulärer Ausdrücke. Selbst Funktionsaufrufe, deren Ergebnisse bereits bekannt waren, wurden „nachgespielt". In diesem Fall war das Bild, das der Python-Interpreter beim Test lieferte, mit dem identisch, was im Text des Selbstlernmaterials stand. Es gab keinerlei zusätzliche Information.

Dieses Verhalten kann man als „Streben nach Gewissheit" interpretieren. Subjektive Gewissheit ist nach Fischbein ein zentrales Merkmal von Intuitionen. Der Test am Computer ist eine echte, Gewissheit spendende Erfahrung in der Wirklichkeit, vergleichbar mit einem naturwissenschaftlichen Experiment. Das Beispiel im Arbeitsblatt dagegen ist nur ein Medium, das ein bloßes Abbild der Realität liefert.

In der Untersuchung wurde nicht eruiert, wie die Schülerinnen und Schüler beim Ausprobieren der Beispiele mit unbekanntem Ergebnis (`W1` bis `W11`) vorgingen. Ein sinnvolles Verfahren ist sicherlich folgendes: Zuerst versucht man das Ergebnis eines Funktionsaufrufs in Gedanken zu ermitteln, und dann prüft man durch ein Experiment am Computer nach, ob man mit der Vermutung richtig liegt. Das Entwickeln einer Lösung in Gedanken kann als Beispiel einer Textinterpretation gesehen werden. Denn es geschieht vor dem Hintergrund erklärender Texte zu regulären Ausdrücken und hat das Ziel das eigene Textverständnis zu prüfen („Habe ich richtig verstanden, was der Sternoperator bedeutet?").

Derartiges „fragendes Experimentieren" ist etwas anderes als Problemlösen (auf das wir später zu sprechen kommen). Beim echten Problemlösen steht ein reales als relevant betrachtetes Problem im Vordergrund. Welchen Weg man beschreitet, um das Problem zu lösen (z.B. welche Programmierkonzepte man verwendet) ist zu Beginn des Problemlösungsprozesses völlig offen. Hier jedoch sind die Aufgabenstellungen der Beispiele von der inhaltlichen Thematik her irrelevant und austauschbar. Ziel ist, ein Konzept (reguläre Ausdrücke) zu verstehen. Die Zahlen in Tab. 27 belegen, dass es den Schülern mehr um das direkte und schnelle Ausprobieren einfacher Funktionsaufrufe (`W1` bis `W11`) mit regulären Ausdrücken ging (im Mittel wurden 7.37 von 11 Beispielen ausgeführt). Die beiden Skripte, die praxisnahe Anwendungen – also „echte Problemlösungen" – darstellten (`A1` und `A2`), wurden nur in erheblich geringerem Ausmaß getestet (0.89 von 2).

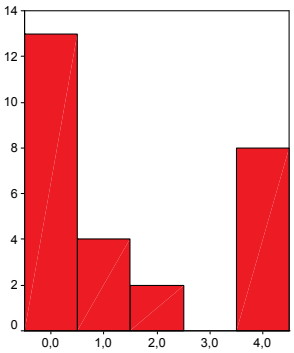


Abb. 86: Häufigkeitsverteilung der ausprobierten vollständigen Beispiele. (n=29)

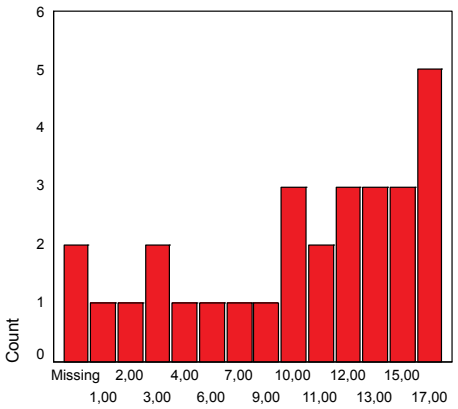


Abb. 87: Häufigkeitsverteilung der insgesamt ausprobierten Beispiele (n=29)

Die relativ großen Standardabweichungen und die Häufigkeitsverteilungen der ausprobierten Beispiele (Abb. 87 und Abb. 87) deuten an, dass man die Teilnehmer der Befragung grob in zwei Gruppen aufteilen kann: Personen, die sehr viele, und solche, die nur wenige Beispiele getestet haben. Besonders krass ist dieses Phänomen bei vollständigen Beispielen (`F1` bis `F4`). Dieses kann man folgendermaßen erklären. Wer wenig ausprobiert, besitzt bereits ein sicheres Verständnis der Funktionsweise regulärer Ausdrücke. Er oder sie kann die Beispiele in Gedanken nachvollziehen und ist sich sicher, dass das Ergebnis richtig ist. Nachprüfen am Computer wäre Zeitverschwendung. Nur wer sich unsicher ist, wem die Verständnis liefernde Intuition noch fehlt, sammelt Erfahrung durch Computerexperimente.

2.2.5 Verwendung von visuellen Modellen als Verstehenshilfe

Zum Selbstlernmaterial gehörten drei Animationen (Flash-Filme), die Metaphern für reguläre Ausdrücke darstellten. Im Begleittext wurden die Schüler aufgefordert, zu jeder Animation in einem Satz die dahinter steckende Idee zu beschreiben. Damit sollte eine Elaboration des Materials angeregt werden. Später sollten die Teilnehmer angeben, welche Animation ihre persönliche Vorstellung von einem regulären Ausdruck am besten wiedergibt. Die Abbildungen Abb. 88 bis Abb. 90 zeigen Screenshots der Flash-Filme.

Die erste Animation visualisiert einen regulären Ausdruck als Sieb, das aus herunterfallenden Karten mit Zeichenketten diejenigen herausfiltert, die auf den regulären Ausdruck passen, also zu `L(re)` gehören. Nicht passende Strings gehen durch das Sieb hindurch. Hier wird also allein der Aspekt des „Herausfilterns“, der Trennung von passenden und unpassenden Strings thematisiert. Wie diese Trennung funktioniert oder funktionieren könnte, wird nicht beschrieben.

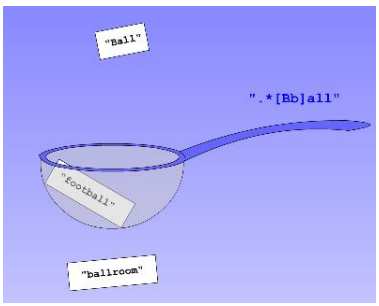


Abb. 88: Regulärer Ausdruck als Sieb, das bestimmte Zeichenketten „ausfiltert“

Der zweite Film veranschaulicht einen regulären Ausdruck als Maschine, die Zettel mit passenden Zeichenketten produziert. Hier wird also die formale Definition der Semantik eines regulären Ausdrucks re über seine Sprache `L(re)` visualisiert.

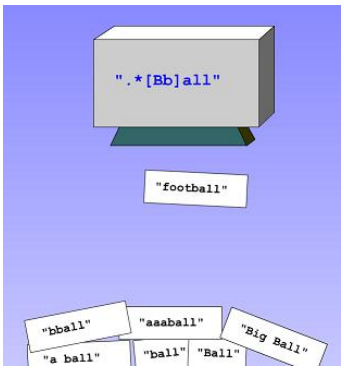


Abb. 89: Regulärer Ausdruck als „Produzent“ von Zeichenketten

Die dritte Animation verwendet die Metapher des Schloss-Schlüssel-Prinzips. Man sieht man „Karten“ mit aufgedruckten Zeichenketten, die über den Bildschirm wandern. Jede Karte besitzt an der Oberkante ein markantes Profil. Es ist der Form der Buchstaben der Zeichenkette nachgebildet (siehe Abb. 90). Der reguläre Ausdruck ist durch ein Gegenprofil visualisiert, das genau auf die Karten mit Strings aus `L(re)` passt. Ähnlich wie ein Magnetkran holt nun der reguläre Ausdruck alle passenden Strings aus dem vorbeiziehen Strom von Karten heraus. Wie bei der ersten Animation (Sieb) wird also das Konzept der Trennung passender und nicht passender Strings zusätzlich aber auch das Konzept des „Passens“ (Schloss-Schlüssel-Prinzip) quasi als Wirkungsmechanismus veranschaulicht.

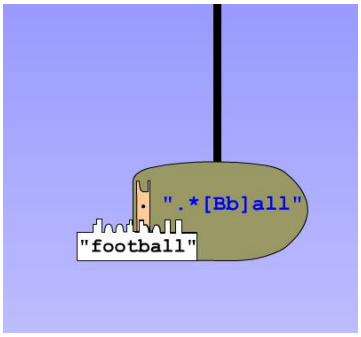


Abb. 90: Regulärer Ausdruck als Muster, das auf bestimmte Zeichenketten „passt“

Schüler der Jahrgangsstufe 12 wurden gefragt, welche der drei Animationen am ehesten ihre Vorstellung von der „Idee“ eines regulären Ausdrucks wiedergibt. Von den 27 Personen, die zu diesem Punkt Angaben machten, nannten jeweils 12 die erste (Sieb) und dritte Animation (Magnetkran). Nur drei Personen fanden, dass die zweite Animation (Wörter-produzierende Maschine) ihre Vorstellung am treffendsten wiedergab.

Vermutlich haben die Schüler vorzugsweise solche intuitiven Modelle gewählt, die in ihrer Vorstellungswelt bereits fest verankert sind. Das Prinzip der Trennung mit einem Sieb dürfte seit dem Kindergartenalter (Spielen im Sandkasten) vertraut sein (Sand von Kieselsteinen mit einem Sieb trennen). Auch das Schloss-Schlüssel-Prinzip gehört zum Alltag: Ein Schlüssel „erkennt" aufgrund seiner Form bestimmte Schlösser, zu denen er passt. Ein Legostein passt nur an bestimmte Stellen eines Lego-Bauwerks, ein Teller passt nur an bestimmte Stellen einer Spülmaschine etc.

Die Vorstellung, dass durch reguläre Ausdrücke eine (eventuell unendliche) Menge von Zeichenketten definiert wird (Mengenkonzept), erlaubt eine elegante Definition der Semantik regulärer Ausdrücke. Die zweite Animation ist eine Visualisierung dieses Modells. Die Produktion einer im Prinzip unendlichen Vielfalt von Dingen nach einem bestimmten Muster ist grundsätzlich ein Konzept, das auch im Alltag vorkommt (Ostereier färben, Mandelas ausmalen etc.). Dennoch wurde diese Intuition nur von wenigen Schülern auf reguläre Ausdrücke angewendet. Möglicherweise liegt es daran, dass das Mengenkonzept für Problemlösungen mit regulären Ausdrücken keine Rolle spielt. Bei praktischen Anwendungen regulärer Ausdrücke – etwa bei der Verarbeitung natürlichsprachlicher Eingaben oder der Analyse von Texten – geht es allein um das Erkennen von Zeichenketten und nicht um die Produktion.

2.3 Problemlösen

2.3.1 Fallstudie: Das Iterator-Pattern und seine Implementierung in Python

Das Iterator-Konzept gehört zu den klassischen 23 Design-Patterns, die in dem grundlegenden Werk von Gamma et al. (1995) beschrieben werden. Es fällt in die Kategorie Verhaltensmuster (behavioural pattern).

Der Begriff Iteration leitet sich von dem lateinischen Wort *iter* ab, was so viel wie Gang oder Marsch bedeutet. Dahinter steckt die Metapher, dass man von vorne nach hinten »durch eine Folge von Objekten marschiert« und z.B. auf jedes Objekt bestimmte Operationen anwendet.

Ein Iterator ist nun ein Objekt, das die in einem Container enthaltenen Objekte nach und nach hervorholt. Ein Iterator beherrscht allein die Methode `next()`, die das nächste Element einer Iteration über den Container liefert.

Duell (1997) beschreibt einige Alltagsbeispiele für Iteratoren:

- In einer Arztpraxis entscheidet die Sprechstundenhilfe der Rezeption, wer der nächste Patient ist, der vom Arzt behandelt wird. Aus der Art und Weise, wie die Patienten im Wartezimmer sitzen, geht die Behandlungsreihenfolge nicht hervor. Die Arzthelferin an der Rezeption ist also ein Iterator für den Container „Wartezimmer".
- Ein modernes Autoradio besitzt eine Taste, mit der man zum nächsten Sender springen kann. Der Benutzer braucht sich keine Gedanken zu machen, auf welchen Frequenzen die Radiostationen senden. Dieser Mechanismus ist ein Iterator für die Menge der Radiosender, die empfangen werden können.

Die Programmiersprache Python enthält seit der Version 2.2 (2002) eine Implementierung des Iterator-Patterns. Container-Klassen wie z.B. Sequenzen sind iterierbar. D.h. ihre Klassendefinition enthält eine Methode `__iter__()`, die zu einem Objekt dieser Klasse ein Iterator-Objekt liefert. Dieser Iterator ist dann an das Container-Objekt gekoppelt und besitzt als einzige Methode die Methode `next()`, die das »nächste Element« liefert, das nach einem bestimmten Mechanismus ausgewählt wird Der Iterator merkt sich das „aktuelle Element" und liefert beim nächsten `next()`-Aufruf den Nachfolger, sofern einer existiert. Falls der Container kein weiteres Element enthält, gibt es eine `StopIteration`-Ausnahme. Dem Iterator ist es nicht möglich, ein bereits geliefertes Element des Containers noch einmal zu referieren. Wenn die Iteration noch einmal von vorne beginnen soll, muss ein neues Iterator-Objekt generiert werden. Beispiel:

```
>>> liste = [1, 'Fisch']    # generiere Container (Liste)
>>> i = iter(liste)         # erzeuge Iterator
>>> i.next()                # naechstes Element der Liste
```

```
1
>>> i.next()
'Fisch'
>>> i.next()
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in ?
    i.next()
StopIteration
```

Es gibt keine explizit spezifizierte Klasse für Iteratoren. Vielmehr kann das Iterator-Pattern ein Merkmal einer (Container-)Klasse sein. Eine iterierbare Klasse enthält in ihrer Definition eine »magische« Methode `__iter__()` und eine Methode `next()`. Damit ist die Standardfunktion iter() befähigt, ein passendes Iteratorobjekt zu generieren (Überladen der Funktion iter()). In der Python-Dokumentation (van Rossum, Yee, 2001) spricht man von einem *Iterator-Protokoll*, dem alle iterierbaren Objekte folgen.

Iteratoren werden (in allen imperativen Programmiersprachen) implizit in Iterationen über Sequenzen (for-Schleifen) verwendet.

In C/ C++ oder Java-Programmen gibt es Anweisungen der Art:

```
for (i=0, i< 5, i++) {
   Anweisungsfolge
}
```

Hier wird durch den Ausdruck `(i=0, i< 5, i++)` eine Folge von Werten definiert, die die Laufvariable `i` nacheinander einnimmt. In der Python-Syntax wird eine solche Wiederholungsanweisung folgendermaßen formuliert.

```
for i in [0, 1, 2, 3, 4]:
    Anweisungsfolge
```

oder

```
for i in range(5):
    Anweisungsfolge
```

Dabei liefert die Funktion `range(n)` eine Liste mit ganzen Zahlen zwischen `0` und einschließlich `4`.

Das Konzept des Iterators ist elementarer und abstrakter als das Konzept der Iteration über eine Sequenz in einer Wiederholungsanweisung (for-Schleife). Denn bei einer Sequenz ist die Reihenfolge der Elemente sichtbar, beim Iterator nicht. Hier wird von der Reihenfolge abstrahiert, sie ist Sache des Iterators.

Nun stützt sich das for-Statement bei Python auf Iteratoren. Der Unterschied kommt dann zum Ausdruck, wenn man Iterationen über Container-Objekte durchführt, in denen die Reihenfolge der enthaltenen Elemente irrelevant und unsichtbar ist. Dazu gehören bei Python Abbildungen (mappings) wie z.B. Dictionaries. In einem Dictionary werden Schlüsseln Werte zugeordnet. Der Zugriff auf einen Wert erfolgt wie in folgendem Beispiel über den zuhörigen Schlüssel:

```
>>> dict = {'Sonne':'sun', 'Mond':'moon'}
>>> dict['Sonne']
'sun'
```

Die Reihenfolge der Schlüssel-Wert-Paare in einem Dictionary ist irrelevant. Angenommen wir wollen in einer Iteration sämtliche Werte eines Dictionaries ausgeben. Bei dieser Aufgabenstellung ist die Reihenfolge, mit der die Einträge des Dictionarys bearbeitet werden, völlig unerheblich. Hauptsache es werden tatsächlich *alle* Elemente erfasst. Nun haben wir zwei Möglichkeiten:

- Iteration über die Liste aller Schlüssel. Diese Liste wird von der Methode `keys()` geliefert. Technisch verwendet das Python-Laufzeitsystem dann den Iterator des *Listen*objektes, das die Schlüssel enthält.

- Iteration mit Hilfe des Iterators des Dictionary-Objektes. Hier wird also keine Liste (oder sonstige Sequenz) verwendet.

Im folgenden Skript-Beispiel werden die beiden Möglichkeiten realisiert:

```
dict = {'Sonne':'sun', 'Mond':'moon', 'Erde':'earth'}
for key in dict.keys():  # Iteration ueber Liste
    print dict[key]

for key in dict:         # Iteration ueber Dictionary
    print dict[key]
```

Das Iterator-Pattern hat folgende Vorteile (vgl. van Rossum, Yee, 2001):

- Kürzere und besser lesbare Programmtexte. Auf intermediäre Listen, die eigens für die Iteration erstellt werden (z.B. mit der `keys()`-Methode bei Dictionaries), kann verzichtet werden.
- Bessere Performanz. Durch die Verwendung des abstrakteren Iterators werden überflüssige Verarbeitungsprozesse vermieden. Es wird sozusagen nur das notwendigste getan. Wenn eine Iteration (wie im letzten Beispiel) von einer vorgegebenen Sequenz losgelöst wird, kann der Iterator selbst eine Reihenfolge unter dem Gesichtspunkt optimaler Rechenzeit wählen. Die Python-Liste ist ein sehr »kostenintensiver« Datentyp. Sie ist keine lineare Liste im üblichen Sinne sondern erlaubt direkten Zugriff auf einzelne Elemente über ihren Index. Diese Möglichkeit wird bei Iterationen nicht genutzt und muss aber dennoch (mit Performanzverlusten) »bezahlt« werden.

Fraglich ist, ob die Verwendung des Iterators an Stelle einer Liste mit einem Verlust an Anschulichkeit bzw. Intuitivität verbunden ist. Eine Python-Liste kann man sich z.B. als lange Kiste mit vielen Fächern vorstellen. Jedes Fach ist mit dem Index des dort enthaltenen Elementes beschriftet. Ein Container-Objekt mit Iterator (ohne offenkundige Reihenfolge der Elemente) kann man nicht so ohne weiteres visualisieren. Man müsste ja die enthaltenen Elemente in irgendeiner Weise darstellen, ohne dass die Darstellung eine bestimmte Reihenfolge suggeriert. Nun ist es natürlich legitim, als *Modell* eine listenähnliche Darstellung zu verwenden. Nur muss man sich dann im Klaren sein, dass die Reihenfolge der Elemente der Modell-Liste kein Merkmal des modellierten Original-Konzeptes ist.

2.4 Kontrolle

2.4.1 Intuitive Modelle und Testen

Inwiefern werden beim Testen eines Programms intuitive Modelle über dessen Arbeitsweise verwendet? Wir beschränken unsere Überlegungen auf das Testen kleiner Systemkomponenten, wie z.B. einzelne Funktionen. Als Beispiel betrachten wir das Testen der Quicksort-Funktion. Man testet eine Funktion, indem man sie mit verschiedenen Argumenten aufruft und das erwartete Ergebnis mit dem tatsächlich gelieferten vergleicht.

Im Software-Engineering unterscheidet man zwischen Black-Box-Testen und White-Box-Testen. Im ersten Fall ist der Programmtext unbekannt und man orientiert sich bei der Auswahl von Testdaten allein an den Anforderungen, die an das Programm gestellt werden (Sommerville 1997, S. 463 ff.).

Beim White-Box-Testen (oder strukturellem Testen) bezieht man Wissen über das Programm bei der Auswahl der Test-Werte mit ein. Das heißt verwendet spezifische Modelle zur Arbeitsweise des Programms. Dies sind Kontrollmodelle im engeren Sinne. Die Vereinfachung liegt darin, dass man die nur die Arbeitsweise bei bestimmten Eingabewerten betrachtet. Nehmen wir als Beispiel folgende Quicksort-Implementierung:

```
def qsort (liste):
    s = liste[:]             # s ist eine Kopie von liste
    if s == []:
        return s
    else:
        x = s[0]
        s.remove(x)          # entferne x aus der Liste s
```

```
    s1 = []
    s2 = []
    for i in s:
        if i <= x:
            s1.append(i)
        else:
            s2.append(i)
    return ergebnis
```

Dann treffen folgende Überlegungen zu:

1 Bei trivialen Eingabesequenzen (leere Liste) gibt die Quicksort-Funktion die Eingabesequenz als Ergebnis zurück.

2 Komplexere Eingabesequenzen werden in drei Stücke `s1`, `[x]`, `s2` zerlegt. Die mittlere Sequenz `[x]` enthält immer genau ein Element. Dabei kann man mehrere Fälle unterscheiden:

2.1 `s1` und `s2` sind leer

2.2 `s1` ist leer und `s2` enthält mindestens ein Element,

2.3 `s2` ist leer und `s1` enthält mindestens ein Element,

2.4 `s1` und `s2` enthalten mindestens ein Element.

Weitere Fälle gibt es nicht. Sofern man dieses Modell im Sinn hat, wird man bei der Wahl von Testsequenzen darauf achten, dass alle diese Situationen durchlaufen werden. Das ist die Idee des Pfadtestens (siehe folgender Abschnitt). Das folgende Set von Testaufrufen ist so konstruiert, dass *beim ersten Durchlauf* einer dieser fünf Fälle eintritt.

```
print qsort ([])              # 1
print qsort ([1])             # 2.1
print qsort ([1, 3])          # 2.2
print qsort ([3, 1])          # 2.3
print qsort ([2, 3, 1])       # 2.4
```

2.4.2 Paradigmatische Modelle beim Testen

Beim Testen eines Programms können Intuitionen eine Rolle spielen, die man zur Gruppe der paradigmatischen Modelle zählen kann. Denn sie beschreiben eigentlich eine allgemeine Strategie des Testens. Sie sind damit keine vereinfachten Modelle des zu testenden Programms, bieten aber ein Grundlage zur systematischen Konstruktion von Testszenarios (hier: Eingabesequenzen).

Modell der vollständigen Induktion

Nach dem Modell der vollständigen Induktion arbeitet die Funktion f unter folgenden Bedingungen korrekt.

- Sie liefert für die kleinsten denkbaren Eingabeobjekte (hier: leere und einelementige Sequenzen) korrekte Ergebnisse.
- Sei `f(s)` ein Funktionsaufruf mit Argument `s`. Unter der Voraussetzung, dass die in der Funktionsdefinition spezifizierten rekursiven Funktionsaufrufe `f(s1), ... , f(sn)` korrekte Ergebnisse liefern, liefert auch `f(s)` ein richtiges Ergebnis.

Wenn man sich an diesem Modell orientiert, wird man die Testargumente folgendermaßen wählen: Man testet die Trivialfälle, die zum sofortigen Abbruch der Rekursion führen. Dann testet man mit Argumenten, die zu verschiedenen rekursiven Aufrufen mit einfacheren Argumenten führen. Im Unterschied zum Modell des „Härtetests“ (s.u.) reicht es, möglichst einfache nicht triviale Testargumente zu verwenden. Man schließt dann induktiv, dass es auch mit komplexeren Werten klappt.

Pfadtesten

Beim so genannten Pfadtesten (path testing) achtet man bei den Testläufen darauf, dass sämtliche unabhängigen Ausführungspfade durchlaufen werden. Das bedeutet insbesondere, dass jede im Pro-

grammtext vorkommende Anweisung in der Summe der Testläufe mindestens einmal ausgeführt wird (Sommerville 1997, S.471 ff).

Ausgangspunkt für das Pfadtesten ist ein Programmflussgraph. Das ist ein gerichteter Graph, der ein auf vorkommende Kontrollstrukturen reduziertes Modell des zu testenden Programms darstellt. Darin werden if-else-Anweisungen als Verzweigungen und while- und for-Anweisungen als Schleifen dargestellt. Zwei unabhängige Ausführungspfade sind Pfade vom Start- zum Zielknoten, die sich zumindest in einer Kante unterscheiden. Abb. 91zeigt einen Programmflussgraphen für die Quicksort-Funktion.

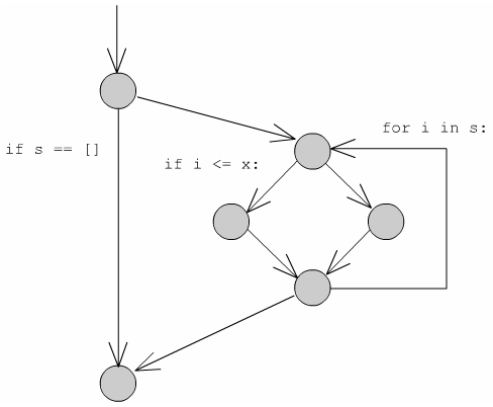


Abb. 91: Programmflussgraph für die Quicksort-Funktion

Belastungstest

Das intuitive Modell des Belastungstests basiert auf folgender Annahme: Wenn die Funktion „schwierige Argumente“ (z.B. lange Sequenzen mit großen Zufallszahlen) korrekt verarbeitet, dann wird sie erst recht für „einfache Argumente“ (kurze Sequenzen mit kleineren Zahlen) richtige Ergebnisse liefern.

Belastungstests zur Überprüfung korrekter Arbeitsweise sind aus dem Alltag bekannt:

- Beim Test eines Autos beobachtet man sein Fahrverhalten auf einer sehr holprigen Teststrecke („Marterstrecke“). Wenn es unter diesen Bedingungen verkehrstauglich ist – so schließt man -, dann ist es das erst recht auf einer normalen Straße.
- Ein Hersteller von Haushaltsgeräten ließ einen Teller mit einer kompletten Sahnetorte von einer Spülmaschine reinigen. Wenn die Spülmaschine diesen Teller in einem Spülgang säubern kann – so das Verkaufsargument -, dann erst recht normal verschmutztes Geschirr.

Zu beachten ist, dass wir hier über Korrektheitstests reden. Im Software-Engineering gibt es auch das Konzept des Stress-Tests (Sommerville 1997, S. 457 f). Stress-Tests werden nicht mit Einzelkomponenten sondern mit ganzen Systemen durchgeführt, die für eine bestimmte Belastung ausgelegt sind (Anzahl der Transaktionen pro Sekunde, Anzahl angeschlossener Rechner etc.). Beim Stress-Testen wird das System mit Absicht überlastet, um zu prüfen, wie das System mit der Situation fertig wird (Zusammenbruch oder kontrollierte Handhabung der Systemüberlastung).

Bewährung in Extremsituationen

Wenn die Funktion für „extreme“ Eingabewerte (Grenzfälle, Extremsituationen) das richtige Ergebnis liefert, wird sie auch bei normalen Argumenten, die man sich als „Mischung“ aus Extremsituationen vorstellen kann, korrekt arbeiten. Zum Beispiel bei einer Sortierfunktion könnten folgende Listen als Extreme verstanden werden:

`[]`	leere Liste
`[1]`	einelementige Liste

`[1, 2, 3]`	aufsteigend sortierte Liste
`[3, 2, 1]`	absteigend sortierte Liste
`[1, 2, 1, 2, 1, 2]`	Liste mit Duplikaten
`[1, 1, 1]`	Liste mit lauter gleichen Elementen

Im Unterschied zu den Belastungstests sind die hier gemeinten Extremsituationen nicht unbedingt „schwierig“. So ist eine leere Liste als Testargument für eine Sortierfunktion ein sehr einfacher Extremfall.

3 Materialien zu den empirischen Untersuchungen

3.1 Visualisierungsübungen

Im Juni 2006 habe ich mit Schülerinnen und Schülern der Holzkamp Gesamtschule Witten Visualisierungsübungen zu Java-Programmen durchgeführt. Dabei wurden die folgenden beiden Aufgabenblätter verwendet.

3.1.1 Aufgabenblatt 1

Informatik visuell "Ein Bild sagt mehr als tausend Worte"

```
public class Soccer
{
    private String[] team = {"Asamoah", "Ballack", "Klose", "Lahm",
"Schneider"};
    public Soccer()
    {
    }

    public void print (int n)
    // Druckt die ersten n Teammitglieder aus
    {
        for(int i=0; i<n; i++)
        {
            System.out.println(team[i]);
        }
    }

    public String get (String anfang)
    // Gibt vollständigen Namen zurück
    {
        String name = "unbekannt";
        String spieler;
        for(int i=0; i<team.length; i++)
        {
            spieler = team[i];
            if (spieler.startsWith(anfang))
            {
                name = spieler;
            }
        }
        return name;
    }
}
```

Übung 1

Zeichnen Sie ein Storyboard aus mehreren Bildern, das die Ausführung folgender Anweisungen veranschaulicht (Sie können zu jeder Anweisung mehrere Bilder zeichnen):

```
wmteam = new Soccer();              // 1
wmteam.print(3)                     // 2
spieler = wmteam.get("Ba")          // 3
```

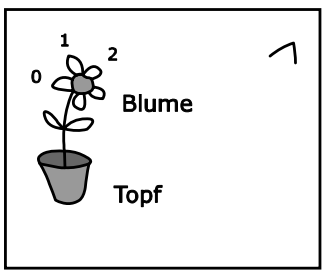
Übung 2

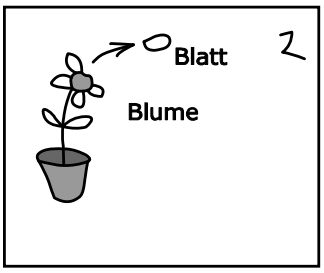
Die Klasse Soccer soll um eine Methode `getBin()` erweitert werden, die den gleichen Effekt hat wie die Methode `get()`. Allerdings soll der Spielername nach dem Verfahren „binäre Suche“ ermittelt werden. Zeichnen Sie ein Storyboard, das die Ausführung dieser neuen Methode bei folgendem Aufruf veranschaulicht:

```
spieler = wmteam.getBin("La")
```

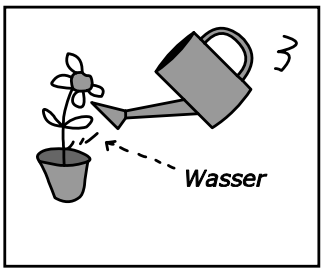
Storyboard

Ein Storyboard besteht aus mehreren Bildern. Sie können Symbole wie Pfeile, Linien, Rechtecke etc. oder auch Abbildungen von konkreten Dingen aus dem Alltag verwenden. Haben Sie Mut zur Fantasie! Jedes Bild ist mit einem kurzen Kommentar versehen, in dem erklärt wird, was passiert.





Entferne Blatt Nr. 2 von Blume



Gieße Blume

3.1.2 Aufgabenblatt 2

Informatik visuell (2)

Übung 3

Die Java-Klasse `Math` enthält statische Methoden, die aufgerufen werden können, ohne zuvor ein Objekt der Klasse zu instanzieren. Der Aufruf `Math.sqrt()` liefert die Quadratwurzel einer Zahl als Gleitpunktzahl vom Typ double. Beispiel:

`Math.sqrt(2)` liefert `1.41421356237`

Zeichnen Sie ein Storyboard aus mehreren Bildern, das die Ausführung folgender Anweisungen veranschaulicht:

```
double a;
a = Math.sqrt(2);
```

Übung 4

Objekte der Klasse String beherrschen die Methode `toUpperCase()`. Der Aufruf `a.toUpperCase()` liefert einen String aus Großbuchstaben.

Beispiel: Wenn `w` die Zeichenkette `"klein"` darstellt, dann gibt `w.toUpperCase()` die Zeichenkette `"KLEIN"` zurück. Das Objekt `w` selbst bleibt dabei unverändert.

Zeichnen Sie ein Storyboard aus mehreren Bildern, das die Ausführung folgender Anweisungen veranschaulicht:

```
String a;
String b;
a = "informatik";
b = a.toUpperCase();
```

3.2 Aufbau der Datenbank der PVS

3.2.1 Allgemeine Tabellen

person

Die Relation `person` enthält die Daten eines Spielers der Python Visual Sandbox.

Attribut	Datentyp	Erläuterung
id	varchar(20)	Primärschlüssel. Er enthält das Pseudonym (Nickname), das Personen bei der Registrierung angeben. Damit wird sie oder er während einer Session angeredet und es erscheint (falls gewünscht) in Highscore-Listen. Außerdem können Lehrer/innen Nicknames verwenden, wenn sie Gruppen definieren. (Die Passwörter dagegen bleiben geheim.)
age	smallint	Alter des Spielers bzw. der Spielerin
gender	char(1)	Geschlecht (f: weiblich, m: männlich)
country	varchar(10)	Kürzel für das Land, aus dem der Spieler kommt.
pass	varchar(20)	Passwort, das ein Spieler bei der Registrierung angegeben hat. Es muss vor dem Start eines Spiels eingegeben werden. Leider kann es sein, dass bei einer Registrierung ein Spieler per Zufall auf ein bereits verwendetes Passwort stößt. Er oder sie muss dann ein neues wählen, weiß dann aber das bereits verwendete Passwort. Dieses wird zu Gunsten einer möglichst einfachen Handhabung in Kauf genommen. (Wenn das Passwort kein Schlüssel wäre, müssten bei jedem Spielstart Name und Passwort eingegeben werden.)
date	int(20)	Datum der Registrierung als Anzahl der Sekunden seit Beginn der "Epoche" (1.1.1970).
profession	varchar(15)	Beruf des Spielers: "school": Schüler, "university": Student, "teacher": Lehrer, "professor": Professor, "professional": Software-Entwickler oder ähnliches, "other": anderer Beruf.
progtime	smallint	Stundenzahl pro Woche, die man programmiert.

Tab. 28: Tabelle `person`

person_group

Die Tabelle realisiert die n:m-Beziehung zwischen den Relationen person und group.

Attribut	Datentyp	Erläuterung
person_id	varchar(20)	Fremdschlüssel mit ID der Person
group_id	varchar(20)	Fremdschlüssel mit ID der Gruppe, in der sich die Person befindet . Das Feld bleibt leer, falls die Person zu keiner Gruppe gehört.

Tab. 29: Tabelle `person_group`

pvsgroup

Die Tabelle modelliert eine Gruppen (z.B. Informatikkurse) mit Gruppenleiter (Coach). Sie enthält gruppenbezogene Daten, mit denen die Validität von Daten der Tabelle `person` (in gewissen Grenzen) geprüft werden kann.

Attribut	Datentyp	Erläuterung
id	varchar(20)	ID der Gruppe
date	int(20)	Datum der Registrierung als Anzahl der Sekunden seit Beginn der "Epoche" (1.1.1970).
coach_id	varchar(20)	ID des Coaches
minage	smallint	Mindestalter

maxage	smallint	Höchstalter
experience	smallint	Tage Programmiererfahrung mit Python
lessons	smallint	Unterrichtsstunden pro Woche
female	smallint	Anzahl weiblicher Personen
country	varchar(10)	Kürzel für das Land
institution	varchar(10)	Art der Institution (university, school, other)
description	varchar(250)	Kurzbeschreibung der Gruppe
townsize	smallint	Größe der Stadt, in der die Gruppe unterrichtet wird. Einwohnerzahl bis 10 000, 50 000, 100 000, 500 000, 1000 000, mehr)

Tab. 30: Tabelle `pvsgroup`

Die Daten einer Gruppe werden in einem Online-Fragebogen erfasst, den der Gruppenleiter (coach) ausfüllen muss.

coach

Die Tabelle enthält die Daten von Gruppenleitern (z.B. Informatiklehrer/in). Ein Coach kann eine Gruppe definieren und die PVS-Sitzungen seiner Gruppen auswerten.

Attribut	**Datentyp**	**Erläuterung**
id	varchar(20)	ID des Coaches (Loginname)
pw	varchar(20)	Passwort
date	int(20)	Datum der Registrierung als Anzahl der Sekunden seit Beginn der "Epoche" (1.1.1970).
address	varchar(250)	Adresse der Institution, für die die Person arbeitet
email	varchar(100)	E-Mail-Adresse
institution	varchar(20)	Art der Institution (university, school, software company, other), an der die Person hauptsächlich beschäftigt ist

Tab. 31: Tabelle `coach`

model

Die Tabelle enthält Beschreibungen der verwendeten Modelle (Animationen).

Attribut	**Datentyp**	**Erläuterung**
id	varchar(20)	Primärschlüssel. Er enthält den Flash-Symbolnamen der Animation.
description	varchar(200)	Verbale Kurzbeschreibung des Modells
complexity	smallint	Anzahl der enthaltenen Elemente (grafische Elemente, Textelemente)
time	smallint	Dauer der Animation in Sekunden (0 bei statischen Modellen)
names	smallint	Prozent der Namen eines zugehörigen Programmtextes, die in diesem Modell zur Bezeichnung analoger Modellteile verwendet werden. Dies ist nur bei korrekten Animationen, die Programmtext illustrieren, relevant. Der Wert dieses Attributs ist ein Maß für die Abstraktheit eines Modells. 0 bedeutet: sehr abstrakt, 100 bedeutet sehr konkret.
a1	smallint	Zusätzliches Attribut für etwaige spätere Verwendung
a2	smallint	Zusätzliches Attribut für etwaige spätere Verwendung
a3	smallint	Zusätzliches Attribut für etwaige spätere Verwendung

Tab. 32: Tabelle `model`

Regeln für die Bestimmung der Komplexität (complexity)

Gezählt werden alle Elemente des Modells, die eine in sich zusammenhängende Entität darstellen. Meist ist es ein einzelner Gegenstand (z.B. Zettel oder Box), manchmal auch mehrere Gegenstände, die irgendwie zusammengehören und sich nicht separat bewegen können (z.B. alle Zettel in einer Kiste, wenn die Zettel sich nicht bewegen oder verändern, oder Blitze, wenn diese immer gleichzeitig auftreten). In der Regel wird ein solches Element in einer eigenen Ebene im Flash-Film dargestellt. Programmtexte und das animierte Highlighting von Programmzeilen werden nicht zum Modell gezählt, wenn sie sich außerhalb des Modellszenariums befinden.

Regeln für die Bestimmung der Abstraktheit (names)

Wir beschränken uns auf Namen. Literale, die im Programmtext vorkommen (Zahlen, Zeichenketten) werden nicht gezählt. (Sie sind oft nur exemplarisch). Zu den Namen zählen alle Namen für Objekte: Variablennamen, Namen von Funktionen, Operatoren, Klassennamen, alle vorkommenden Namen für Listenelemente (z.B. `s[i]`). Letztere werden separat gezählt. D.h. wenn die Namen `s` und `i` schon gezählt worden sind, wird `s[i]` als weiterer Name gewertet.

Nicht gezählt werden `def`, = (Zuweisung), `return`, `for`, `if`, `while`.

3.2.2 Tabellen für Python Visual

protocol_pv

Die Tabelle modelliert Sessions mit einer *Python Visual*-Applikation ohne das Befragungsergebnis. Jedes Tupel enthält zwischen drei und fünf Modellen, die bewertet werden können. Es können ein oder zwei Spieler teilnehmen.

Attribut	Datentyp	Erläuterung
id	varchar(20)	Primärschlüssel. Er enthält einen String folgenden Formats: <Nickname des ersten Spielers><Zeitstempel>
game	varchar(20)	Fremdschlüssel. Enthält id der Spielbeschreibung.
time	int(20)	Zeitstempel (Sekunden seit Beginn der Epoche).
time1	smallint	Betrachtungszeit erstes Modell
time2	smallint	Betrachtungszeit zweites Modell
time3	smallint	Betrachtungszeit drittes Modell (0, wenn Modell nicht existiert)
time4	smallint	Betrachtungszeit viertes Modell (0, wenn Modell nicht existiert)
time5	smallint	Betrachtungszeit fünftes Modell (0, wenn Modell nicht existiert)

Tab. 33: Tabelle `protocol_pv`

description_pv

Die Relation enthält Beschreibungen von „Python Visual“-Applikationen.

Attribut	Datentyp	Erläuterung
id	varchar(20)	Primärschlüssel. Er enthält den Namen des Spiels (Dateiname ohne Extension).
description	varchar(200)	Kurzbeschreibung
models	smallint	Anzahl der enthaltenen Modelle.
mod1	varchar(20)	Fremdschlüssel. Er enthält die id des 1. Modells
mod2	varchar(20)	Fremdschlüssel. Er enthält die id des 2. Modells
mod3	varchar(20)	Fremdschlüssel. Er enthält die id des 3. Modells
mod4	varchar(20)	Fremdschlüssel. Er enthält die id des 4. Modells, falls es existiert. Sonst leerer String.
mod5	varchar(20)	Fremdschlüssel. Er enthält die id des 1. Modells

question1	varchar(100)	Erste Frage (ev. Kurzform)
question2	varchar(100)	Zweite Frage (ev. Kurzform)
question3	varchar(100)	Dritte Frage (ev. Kurzform)
a1	varchar(10)	Weiteres Attribut (Reserve)

Tab. 34: Tabelle `description_pv`

protocol_pv_person

Die Relation verbindet die Relationen `protocol_pv` und `person`. Wenn ein Spieler eine Applikation vom Typ „Python Visual“ bearbeitet, wird hier ein Tupel eingetragen. Damit wird die Teilnahme einer Person an einer Sitzung – also eine Beziehung zwischen einer `person`-Entität und einer `protocol_pv`-Entität – modelliert.

Attribut	Datentyp	Erläuterung
id_person	varchar(20)	Fremdschlüssel mit id des Spielers.
id_protocol_pv	varchar(20)	Fremdschlüssel mit id der Session.
a1	smallint	Antwort Frage 1 (Modellnummer)
a2	smallint	Antwort Frage 2 (Modellnummer)
a3	smallint	Antwort Frage 3 (Modellnummer)

Tab. 35: Tabelle `protocol_pv_person`

3.2.3 Tabellen für Python Puzzle

description_pp

Die Relation enthält Beschreibungen von „Python Puzzle“-Applikationen.

Attribut	Datentyp	Erläuterung
id	varchar(20)	Primärschlüssel. Er enthält den Namen des Spiels (Dateiname ohne Extension).
description	varchar(200)	Kurzbeschreibung
time	smallint	Maximale Gesamtzeit des Spiels
a1	varchar(10)	Weiteres Attribut (Reserve)

Tab. 36: Tabelle `description_pp`

protocol_pp

Die Tabelle modelliert Sessions mit einer "Python Puzzle"-Applikation. An einer Session können ein oder zwei Spieler teilnehmen. Die Tabelle beschreibt allgemeine – nicht aufgabenbezogene Daten der Session.

Attribut	Datentyp	Erläuterung
id	varchar(20)	Primärschlüssel. Er enthält einen String folgenden Formats: <id des Spiels><Zufallszahl>
game	varchar(20)	Fremdschlüssel. Enthält id der Spielbeschreibung.
time	int(20)	Zeitstempel (Sekunden seit Beginn der Epoche) für Spielstart
gametime	smallint	tatsächliche Spieldauer in Sekunden
points	smallint	erreichte Punkte (-200, falls Session abgebrochen wurde)
players	smallint	Anzahl der Spieler (1 oder 2)
player1	varchar(20)	Fremdschlüssel. Enthält id des ersten Spielers.
player2	varchar(20)	Fremdschlüssel. Enthält id des zweiten Spielers (falls vorhanden)

Tab. 37: Tabelle `protocol_pp`

Zu Beginn einer Session wird ein Datensatz mit Punktzahl -200 angelegt. Am Ende der Session wird die Punktzahl aktualisiert.

protocol_pp_task

Jede Zeile der Tabelle beschreibt einen erfolgreichen Python Puzzle Testlauf eines selbst zusammengesetzten Programms.

Attribut	**Datentyp**	**Erläuterung**
id	varchar(20)	Primärschlüssel
task	smallint	Nummer der Aufgabe
session_id	varchar(20)	Fremdschlüssel. Enthält ID der Session (protocol_pp). Darin sind die z.B.Spieler-IDs enthalten.
runs	smallint	Anzahl der Testlaeufe
time	smallint	Zeitbedarf zur Lösung der Aufgabe
points	smallint	erreichte Punktzahl
c1 ... c10	varchar(60)	Programmzeile des korrekten Programmtextes
e1 ... e8	varchar(60)	Falsche Programmzeile, die bei einem Testlauf einmal verwendet worden ist

Tab. 38: Tabelle `protocol_pp_task`

Ein Tupel der Relation wird angelegt, sobald ein Testlauf erfolgreich war. Die Attribute c1, ..., c10 enthalten Programmzeilen, die zur korrekten Lösung gehören und zwar in der Reihenfolge, in der sie verwendet worden sind (c1 enthält die erste verwendete Programmzeile). Somit kann man später z.B. feststellen, an welchen Aspekt der Problemlösung die Teilnehmer zuerst oder zuletzt dachten.

Die Attribute e1, ..., e8 enthalten falsche verwendete Programmzeilen, also Chips, die irgendwann einmal während der Lösung der Aufgabe in das Editorfeld gezogen worden sind – wiederum in der Reihenfolge der Verwendung. Es kann sein, dass einen Spieler einen falschen Chip „anfasst“, ihm der Fehler aber sofort bewusst wurde und er den Chip wieder heraus zieht, so dass diese Anweisung niemals bei einem Testlauf zum Einsatz kam und deshalb auch nicht registriert wurde.

protocol_pp_model

Die Relation modelliert die Bewertung der betrachteten Modelle auf den Hinweiskarten. Sie realisiert eine Beziehung zwischen Modell (model) und Protokoll eines erfolgreichen Python-Puzzle-Testlaufs (protocol_pp_task).

Attribut	Datentyp	Erläuterung
model_id	varchar(20)	Fremdschlüssel. Er enthält einen String mit ID des Modells (model) auf der Cue Card
task_id	varchar(20)	Fremdschlüssel. Enthält ID des Protokolls zu einer Aufgabenlösung (protocol_pp)
players	smallint	Anzahl der Spieler, die hinter der Bewertung stehen
score	smallint	Bewertung des Modells
time	smallint	Betrachtungszeit des Modells

Tab. 39: Tabelle `protocol_pp_model`

3.2.4 Tabellen für Python Quiz

description_pq

Beschreibung einer Python Quiz Applikation.

Attribut	Datentyp	Erläuterung
id	varchar(20)	Primärschlüssel. Er enthält den Namen des Spiels (Dateiname ohne Extension).
description	varchar(200)	Kurzbeschreibung auf Englisch
a1	varchar(10)	Weiteres Attribut (Reserve)

Tab. 40: Tabelle `description_pq`

description_pq_task

Jede Zeile der Tabelle beschreibt eine Aufgabe eines Python Quiz. Eine Aufgabe besteht grundsätzlich darin, für verschiedene Modelle zu entscheiden ob sie zu einem Programmfragment passen. Die Korrektheit eines Modells (zum Programmtext passend, akzeptabel oder nicht passend) wird nicht modelliert.

Attribut	Datentyp	Erläuterung
id	varchar(20)	Primärschlüssel (wird zusammengesetzt aus ID des Spiels und Nummer der Aufgabe)
task	smallint	Nummer der Aufgabe
game	varchar(20)	Fremdschlüssel. Enthält ID des Spiels (description_pq), zu dem diese Aufgabe gehört.
program	varchar(250)	Programmtext oder verbale Beschreibung des Programmtextes, der durch Modelle interpretiert wird.

Tab. 41: Tabelle `description_pq`

protocol_pq

Die Tabelle modelliert eine Session mit einer "Python Quiz"-Applikation. Es können ein oder zwei Spieler teilnehmen. Die Tabelle beschreibt allgemeine – nicht aufgabenbezogene Daten der Session. Diese Tabelle wird automatisch von den Dienstprogrammen `pq_service.py` und `pq_startservice` aktualisiert.

Attribut	Datentyp	Erläuterung
id	varchar(20)	Primärschlüssel. Er enthält einen String folgenden Formats: <id des Spiels><Zufallszahl>
game	varchar(20)	Fremdschlüssel. Enthält id der Spielbeschreibung.
time	int(20)	Zeitstempel (Sekunden seit Beginn der Epoche) für Spielstart
gametime	smallint	tatsächliche Spieldauer in Sekunden
points	smallint	erreichte Punkte (-200, falls Session abgebrochen wurde)
players	smallint	Anzahl der Spieler (1 oder 2)
player1	varchar(20)	Fremdschlüssel. Enthält id des ersten Spielers.
player2	varchar(20)	Fremdschlüssel. Enthält id des zweiten Spielers (falls vorhanden)

Tab. 42: Tabelle `protocol_pq`

Zu Beginn einer Session wird ein Datensatz angelegt. Am Ende der Session wird die Punktzahl aktualisiert.

protocol_pq_model

Die Relation modelliert die Bewertung eines Modells in einem Quiz. Diese Tabelle wird automatisch vom Dienstprogramm `pq_service.py` aktualisiert.

Attribut	Datentyp	Erläuterung
protocol_id	varchar(20)	Fremdschlüssel. Enthält ID des Protokolls zu einer Quiz-Session (protocol_pq).
task_id	varchar(20)	Fremdschlüssel. Enthält ID der Beschreibung einer Aufgabe
players	smallint	Anzahl der Spieler, die hinter der Bewertung stehen
model	varchar(20)	ID des bewerteten Modells (in der Reihenfolge der Bewertung)
fits	smallint	Bewertung des Modells (0: unpassend, 1: passend)
time	smallint	Betrachtungszeit des Modells
points	smallint	gesetzte Punktzahl

Tab. 43: Tabelle `protocol_pq_model`

3.3 Gruppen einrichten

Lehrerinnen und Lehrern bietet die PVS die Möglichkeit, Schülergruppen zu definieren. Dazu geben sie auf einer interaktiven Webseite verschiedene Daten zu ihrer Gruppe an (Schule, minimales und maximales Alter der Gruppenmitglieder, Größe der Stadt, Programmiererfahrung ...) und zählen die Loginnamen (Pseudonyme) der Mitglieder ihrer Gruppe auf. Sie haben dann die Möglichkeit, statistische Daten zu ihrer Gruppe abzufragen. Zum Beispiel kann man die statistische Verteilung der Antworten zu einem Python Visual–Spiel als Aufhänger für ein Unterrichtsgespräch verwenden.

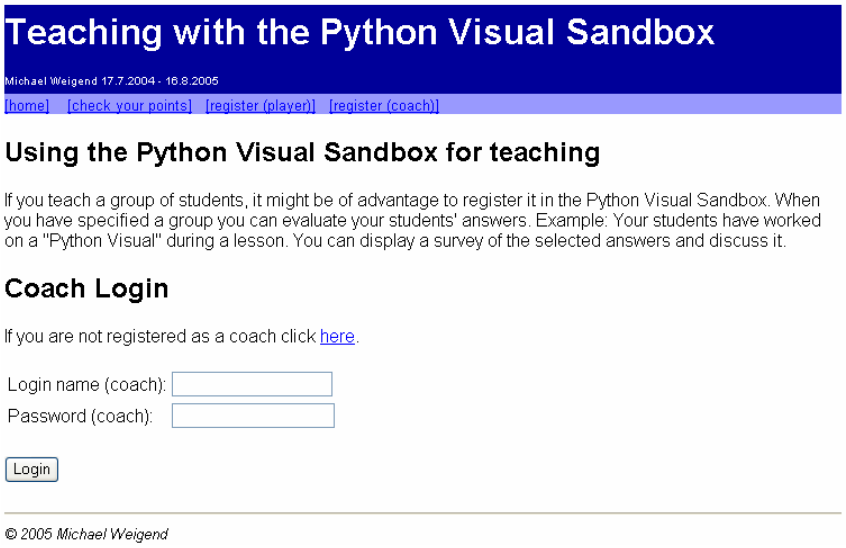


Abb. 92: Dialogseite für Coaches

3.4 Auswertung der Datenbank

Mit verschiedenen über das Internet zugänglichen Werkzeugen kann die Datenbank der PVS ausgewertet werden. Die Datenbank auch Daten von Testentitäten, die keine realen Personen repräsentieren und in keiner Auswertung berücksichtigt werden. Der Name einer Testentität beginnt mit der Zeichenkette `test_`. Testentitäten sind notwendig, um die Funktionalität der PVS zu prüfen und etwaige Fehler zu entdecken.

Die Auswertungswerkzeuge sind entweder öffentlich (Usage), involvierten Spielern zugänglich (z.B. Activity Reports, Highscore-Listen) oder komplett abgeschirmt (wissenschaftliche Auswertung).

3.4.1 Auswertungsmöglichkeiten für Spieler und Zuschauer

Völlig öffentlich ist der Usage-Bericht, der von der Startseite der PVS aus abgerufen werden kann. Er enthält globale Angaben über Anzahl der Sitzungen bei den verschiedenen Applikationstypen, Beruf und mittleres Alter der Teilnehmer und ähnliches.

Jeder Spieler der PVS kann (nach Eingabe seines Passwortes) einen Activity Report abrufen. Er enthält die bisher erreichte Gesamtpunktzahl und eine Auflistung aller Sitzungen.

Python Visual Sandbox

Michael Weigend 17. 7. 2004 - 30. 5. 2005

[home]

Test_hans 's Activity Report

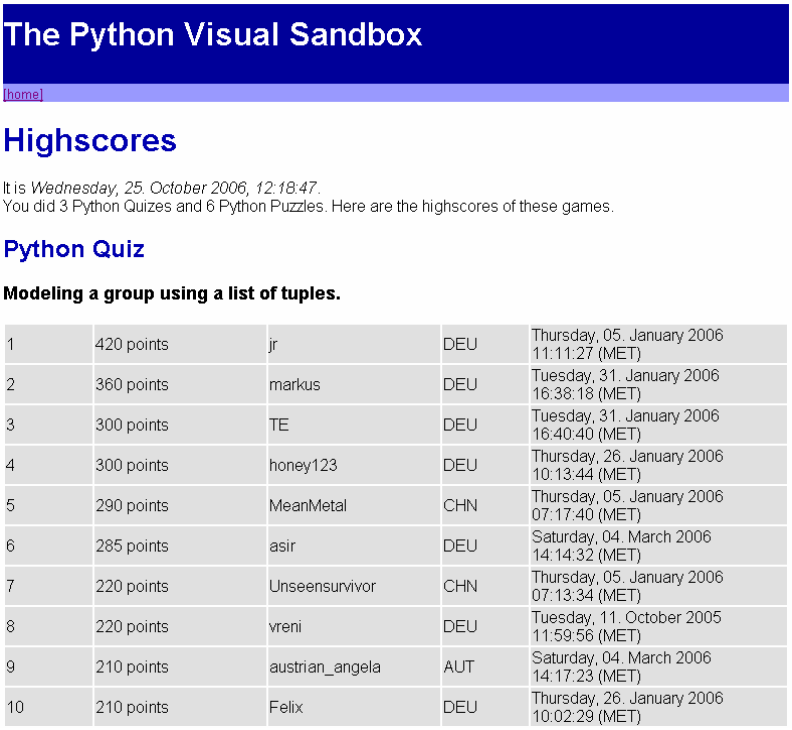
You have got **-12890** points.

Activities

Python Visual	**Time of first valid session**	**Points**
Recursive function that mirrors a string. *(1 valid session)*	Thu Aug 25 20:33:30 2005	20
Regular expressions *(2 valid sessions)*	Sat Aug 20 08:01:32 2005	20
What happens, when a function returns something? *(2 valid sessions)*	Tue Aug 2 08:18:25 2005	20
Recursive computation of factorial *(1 valid session)*	Wed Jan 4 14:48:54 2006	20
Analogies for iterations *(1 valid session)*	Mon Apr 18 23:07:21 2005	20
Multilists *(2 valid sessions)*	Sat Apr 23 10:11:59 2005	20
Python Puzzle	**Time of session**	**Points**
Modeling a group	Thu May 26 22:57:50 2005	-200
Modeling a group	Thu May 26 23:14:24 2005	60

Abb. 93: Actvity Report der Testentität `test_hans`

Jeder Spieler kann nach Eingabe seines Passwortes die Highscore-Listen aller Spiele abrufen, die sie oder er selbst durchgeführt hat. Jede Hisghscore-Liste zeigt Punktzahl, Nicknames der Spieler, Herkunftsland und Datum der besten zehn Sessions eines Spiels. Dies motiviert dazu, an vielen Spielen teilzunehmen und bei einem Spiel ein möglichst gutes Resultat zu erzielen.



The Python Visual Sandbox

[home]

Highscores

It is *Wednesday, 25. October 2006, 12:18:47.*
You did 3 Python Quizes and 6 Python Puzzles. Here are the highscores of these games.

Python Quiz

Modeling a group using a list of tuples.

1	420 points	jr	DEU	Thursday, 05. January 2006 11:11:27 (MET)
2	360 points	markus	DEU	Tuesday, 31. January 2006 16:38:18 (MET)
3	300 points	TE	DEU	Tuesday, 31. January 2006 16:40:40 (MET)
4	300 points	honey123	DEU	Thursday, 26. January 2006 10:13:44 (MET)
5	290 points	MeanMetal	CHN	Thursday, 05. January 2006 07:17:40 (MET)
6	285 points	asir	DEU	Saturday, 04. March 2006 14:14:32 (MET)
7	220 points	Unseensurvivor	CHN	Thursday, 05. January 2006 07:13:34 (MET)
8	220 points	vreni	DEU	Tuesday, 11. October 2005 11:59:56 (MET)
9	210 points	austrian_angela	AUT	Saturday, 04. March 2006 14:17:23 (MET)
10	210 points	Felix	DEU	Thursday, 26. January 2006 10:02:29 (MET)

Abb. 94: Auszug aus einem Highscore-Bericht

Für Python-Visual-Applikationen gibt es ein Auswertungswerkzeug, das die Antworten der Teilnehmer einer Gruppe wiedergibt (siehe Abb. 95). Es ist nur dem Coach einer Gruppe zugänglich. Der

Coach wählt eine Gruppe und eine Applikation aus und fordert einen Evaluationsbericht an. Dieser enthält zu jedem Modell der Aufgabe einen Screenshot, die zeitliche Länge und die durchschnittliche Betrachtungszeit. Aus letzterer kann man schließen, wie ernsthaft sich die Spieler mit dem jeweiligen Modell auseinandergesetzt haben. Außerdem werden die Fragen (in Kurzform) und zu jeder Frage der Anteil der Spieler, die das jeweilige Modell zu dieser Frage ausgewählt haben, dargestellt. In Workshops mit der PVS kann eine solche Evaluation als Anlass für eine Plenumsdiskussion über die dargestellten Modelle verwendet werden. Typischerweise werden in einer solchen Diskussion folgende Punkte angesprochen:

- Warum wird Modell X von den meisten abgelehnt? Inwiefern ist dieses Modell X ungeeignet oder gar falsch?
- Die meisten würden Modell X oder Modell Y verwenden, um die Arbeitsweise des Programms zu erklären. Vergleichen wir diese beiden Modelle. Was unterscheidet sie? Was spricht jeweils für das eine oder das andere Modell?
- Warum können sich die meisten an Modell X besonders gut erinnern? Warum kann man sich bestimmte Modelle besonders gut merken?

In einer solchen Diskussion reflektieren die Teilnehmer die Verwendung intuitiver Modell zur Erklärung und zum Verstehen von Programmtexten.

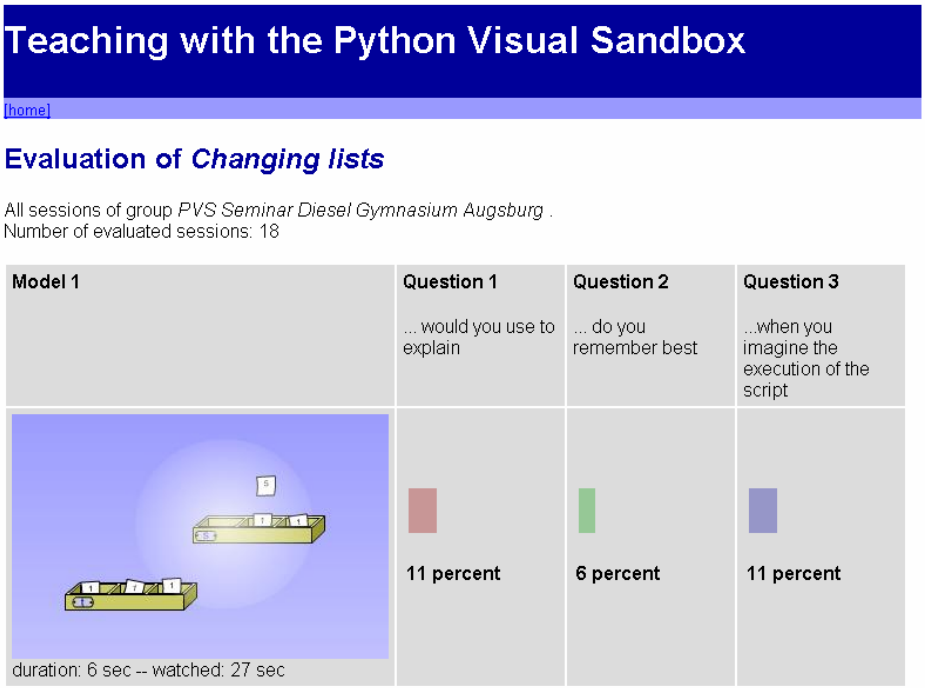


Abb. 95: Gruppenbezogene Auswertung eines Python-Visuals (Ausschnitt)

3.4.2 Wissenschaftliche Auswertung

Zur statistischen Auswertung enthält die PVS für jeden Typ (Python Visual, Python Puzzle und Python Quiz) ein Auswertungswerkzeug, das über das Internet erreichbar ist (Anklicken einer versteckten Stelle auf der PVS-Startseite und Eingabe eines Passworts). Diese Programme liefern jeweils eine Webseite mit statistischen Daten, die folgendermaßen aufgebaut ist.

Für jede der folgenden Personengruppen gibt es getrennt einen statistischen Überblick:

- Alle Teilnehmer.
- Alle Teilnehmer, die an einem Workshop mit der PVS teilgenommen haben.
- Schülerinnen und Schüler, die an einem Workshop teilgenommen haben.

Art und Darstellung der Daten zu der jeweiligen Personengruppe hängen von der PVS-Applikation ab. Bei Python Visuals werden Betrachtungszeiten für die Modelle und die Verteilung der Antworten auf die Fragen wiedergegeben. Besonders komplex ist die Auswertung der Python Puzzles. Hier wird die mittlere Reihenfolge, in der richtige und falsche Puzzlestücke verwendet worden sind, dargestellt. Außerdem enthält das Auswertungsdokument Angaben über die Benutzung und Bewertung der visuellen Modelle, die als Hilfe angeboten worden sind. Die Auswertung der Python-Quiz-Sitzungen konzentriert sich auf Entscheidungszeiten, Bewertung und Konfidenz der Bewertung von visuellen Modellen.

Jedes Spiel der PVS kann von einer oder zwei Personen gespielt werden. Damit wird die übliche Ausstattung von Rechnerräumen an Schulen berücksichtigt, die häufig nur einen Computer für zwei Personen vorsehen. In der statistischen Auswertung wird deshalb zwischen subjektiven und objektiven Sitzungen unterschieden. Wenn zwei Personen zusammen spielen, entstehen eine objektive und zwei subjektive Sitzungen. Allerdings wurden auf den Workshops mit der PVS in fast 90% der Fälle Einzelsitzungen beobachtet.

Nun kann ja eine Person eine PVS-Applikation beliebig oft spielen. In den Statistiken werden in der Regel nur die ersten (subjektiven) Sitzungen ausgewertet. Insbesondere Python-Quiz-Applikationen wurden jedoch von einigen Schülerinnen und Schülern gerne mehrfach gespielt. Deshalb gibt es im Auswertungsdokument für Python Quiz für diese „Vielspieler“ zusätzliche Statistiken, die die Beobachtungen der ersten drei subjektiven Sitzungen darstellen.

3.5 Python Visual

3.5.1 Dokumentation einer Session

Die Merkmale eines Python Visuals (IDs der Modelle, Fragen zu den Modellen etc.) sind in der Datenbank gespeichert (Relation description_pv). In der Relation protocol_pv werden für jede Session die Betrachtungszeiten der einzelnen Modelle festgehalten. In der Relation `protocol_pv_person` sind für jeden Spieler die Antworten niedergelegt.

Am Ende einer Session wird ein XML-Paket an das Service-Programm `pv_service.py` geschickt. Es ist folgendermaßen aufgebaut:

```
<answers>
<person a3="zahl3" a2="zahl2" a1="zahl1" id="Nickname des Spielers" />
…
<times z5="zeit5" z4=" zeit4" z3=" zeit3" z2="40" z1="28" z0="21" />
<game game="pv_mirror" />
</answers>
```

Beispiel:

```
<answers>
<person a3="1" a2="2" a1="2" id="test_juli" />
<times z5="0" z4="22" z3="34" z2="40" z1="28" z0="21" />
<game game="pv_mirror" />
</answers>
```

3.5.2 Auszug aus der automatisch erstellten Auswertung

Der folgende gekürzte Auszug aus dem Ergebnis der automatischen Auswertung liefert einen statistischen Überblick über die Antworten von Schülerinnen und Schülern, die an einem Workshop mit der PVS teilgenommen haben. Betrachtet werden nur die *ersten* Sitzungen mit dem der jeweiligen Python-Visual-Applikation. Es hat also keine Auswirkungen auf das Ergebnis, wenn Teilnehmer ein Spiel mehrfach durchführen.

Regular expressions

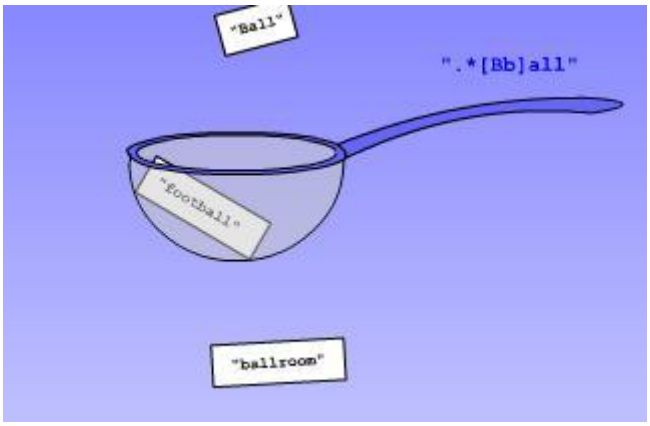
Some general information about the players of this category, who played the game at least *once*.

Professions	18 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	5 female and 13 male persons
Hours a week spent on programming (standard deviation)	1.67 hours (1.78)
Roughly estimated experience in Python programming(standard deviation)	201.17 days (166.35)
Age (standard deviation)	19.17 years (4.85)
Population of the town, where the workshop took place	Less than 100 000: 1, 100 000 to 500 000: 13, more than 500 000: 4
Country	Germany: 18 other country: 0

Model 1

Description: Visualizes the effect of a regular expression by a sieve.

Duration: 8 seconds
Concreteness: 0 percent
Average watchtime (standard deviation): 23.78 seconds (12.60)



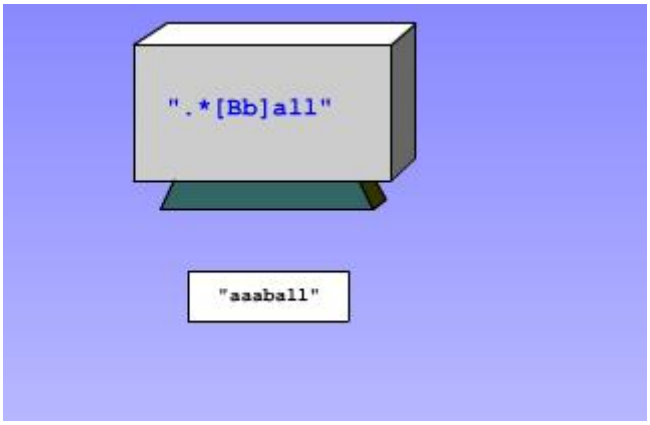
This model was evaluated in 18 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	8	44.44
... do you remember best	7	38.89
... when you imagine the use of a regular expression	9	50.00

Model 2

Description: Visualizes the effect of a regular expression by a machine which generates matching strings.

Duration: 12 seconds
Concreteness: 0 percent
Average watchtime (standard deviation): 22.61 seconds (14.65)



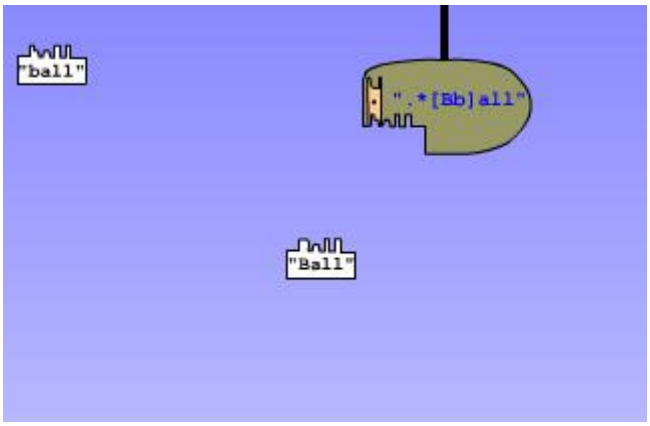
This model was evaluated in 18 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	1	5.56
... do you remember best	3	16.67
... when you imagine the use of a regular expression	2	11.11

Model 3

Description: Crane that selects maching string applying key-lock principle.

Duration: 21 seconds
Concreteness: 0 percent
Average watchtime (standard deviation): 23.39 seconds (9.20)



This model was evaluated in 18 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	9	50.00
... do you remember best	8	44.44
... when you imagine the use of a regular expression	7	38.89

Straight selection

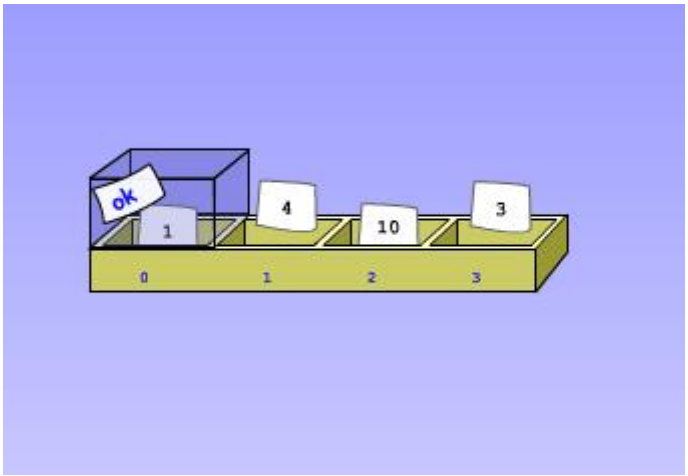
Some general information about the players of this category, who played the game at least *once*.

Professions	15 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	3 female and 12 male persons
Hours a week spent on programming (standard deviation)	2.33 hours (2.44)
Roughly estimated experience in Python programming(standard deviation)	223.00 days (164.75)
Age (standard deviation)	18.20 years (1.21)
Population of the town, where the workshop took place	Less than 100 000: 4, 100 000 to 500 000: 8, more than 500 000: 3
Country	Germany: 15 other country: 0

Model 1

Description: Sorting by straight selection and exchanging places, actual elements of the list are marked by lifting.

Duration: 23 seconds
Concreteness: 9 percent
Average watchtime (standard deviation): 40.27 seconds (21.86)



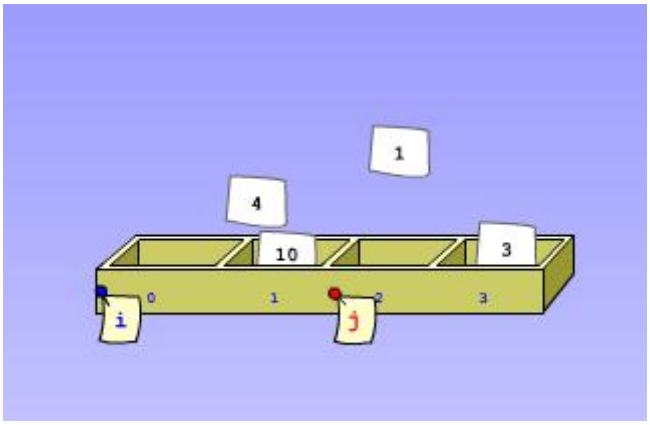
This model was evaluated in 15 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	9	60.00
... do you remember best	5	33.33
...when you imagine the execution of the script	7	46.67

Model 2

Description: Sorting by straight selection and exchanging places, actual elements of the list are marked by post-its i, j .

Duration: 21 seconds
Concreteness: 27 percent
Average watchtime (standard deviation): 31.53 seconds (43.67)



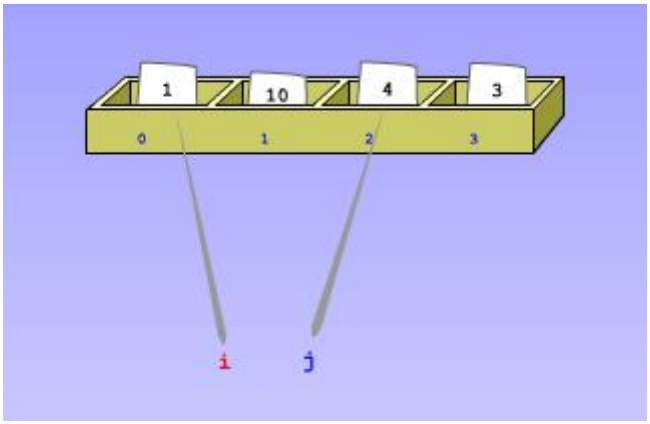
This model was evaluated in 15 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	4	26.67
... do you remember best	7	46.67
...when you imagine the execution of the script	4	26.67

Model 3

Description: Sorting by straight selection and exchanging places. The actual elements of the list are marked by arrows named i and j.

Duration: 21 seconds
Concreteness: 27 percent
Average watchtime (standard deviation): 18.40 seconds (15.44)



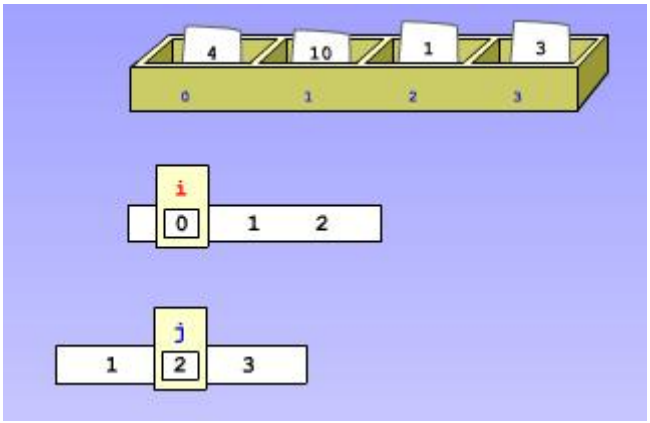
This model was evaluated in 15 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	0	0.00
... do you remember best	2	13.33
...when you imagine the execution of the script	3	20.00

Model 4

Description: Sorting by straight selection and exchanging places. Indices of the actual elements of the list are represented by frames on strips with numbers.

Duration: 21 seconds
Concreteness: 27 percent
Average watchtime (standard deviation): 43.47 seconds (110.10)



This model was evaluated in 15 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	2	13.33
... do you remember best	1	6.67
...when you imagine the execution of the script	1	6.67

Changing lists

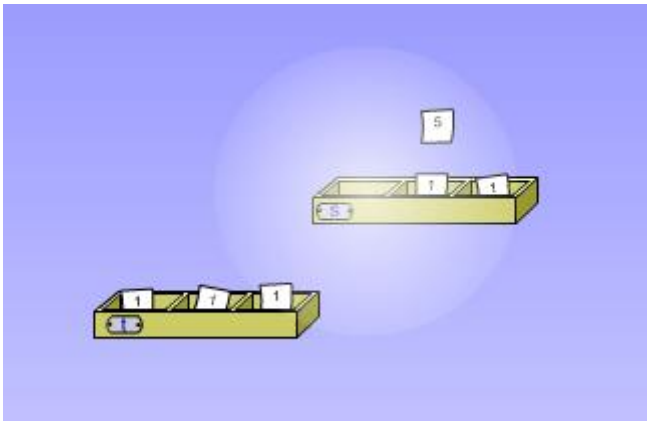
Some general information about the players of this category, who played the game at least *once*.

Professions	70 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	14 female and 56 male persons
Hours a week spent on programming (standard deviation)	3.94 hours (5.91)
Roughly estimated experience in Python programming(standard deviation)	184.67 days (160.58)
Age (standard deviation)	17.01 years (1.59)
Population of the town, where the workshop took place	Less than 100 000: 21, 100 000 to 500 000: 20, more than 500 000: 29
Country	Germany: 70 other country: 0

Model 1

Description: Wrong visualization of changing lists. Lists are represented by boxes with cards. Assign statement is interpreted as copy. Only one copy of the list is changed.

Duration: 6 seconds
Concreteness: 66 percent
Average watchtime (standard deviation): 24.47 seconds (14.09)



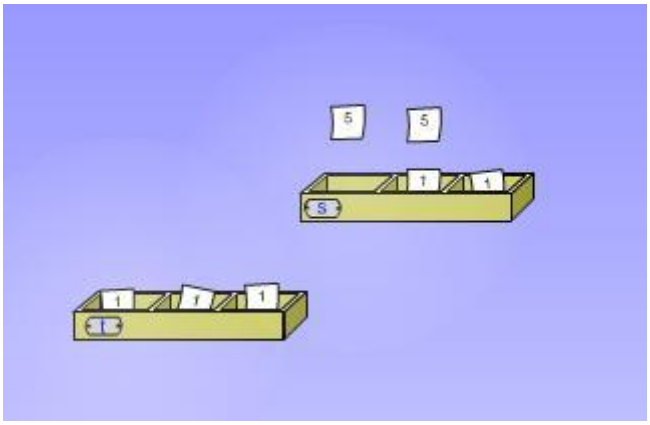
This model was evaluated in 70 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	15	21.43
... do you remember best	12	17.14
...when you imagine the execution of the script	10	14.29

Model 2

Description: Model of changing lists. Lists are represented by boxes with cards. Assign statement is interpreted as copy (ghost). Both copies of the list are changed.

Duration: 7 seconds
Concreteness: 66 percent
Average watchtime (standard deviation): 14.99 seconds (10.07)



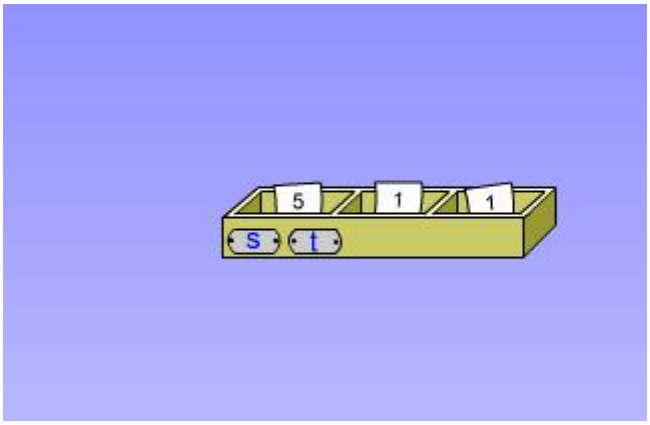
This model was evaluated in 70 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	40	57.14
... do you remember best	23	32.86
...when you imagine the execution of the script	34	48.57

Model 3

Description: Model of changing lists. Lists are represented by boxes with cards. Assign statement is interpreted as adding a second label to the box.

Duration: 5 seconds
Concreteness: 66 percent
Average watchtime (standard deviation): 14.46 seconds (15.19)



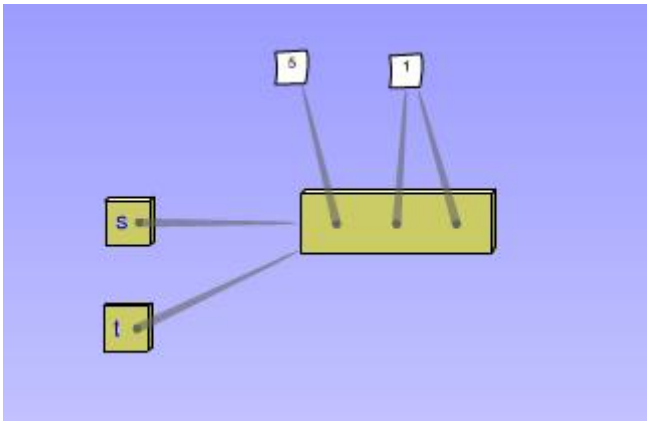
This model was evaluated in 70 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	11	15.71
... do you remember best	16	22.86
...when you imagine the execution of the script	17	24.29

Model 4

Description: Model of changing lists. Lists are represented by boards and arrows. Assign statement is interpreted as adding a new arrow.

Duration: 7 seconds
Concreteness: 66 percent
Average watchtime (standard deviation): 25.39 seconds (84.62)



This model was evaluated in 70 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	4	5.71
... do you remember best	19	27.14
...when you imagine the execution of the script	9	12.86

Recursive function that mirrors a string.

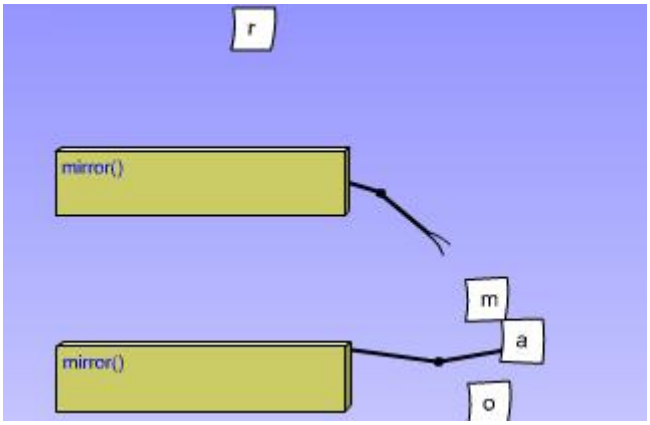
Some general information about the players of this category, who played the game at least *once*.

Professions	22 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	2 female and 20 male persons
Hours a week spent on programming (standard deviation)	4.23 hours (3.22)
Roughly estimated experience in Python programming(standard deviation)	83.59 days (110.33)
Age (standard deviation)	17.36 years (1.43)
Population of the town, where the workshop took place	Less than 100 000: 10, 100 000 to 500 000: 7, more than 500 000: 5
Country	Germany: 22 other country: 0

Model 1

Description: Function call is represented by an active element with grabs. Only one recursive call is visualized.

Duration: 22 seconds
Concreteness: 14 percent
Average watchtime (standard deviation): 31.86 seconds (16.29)



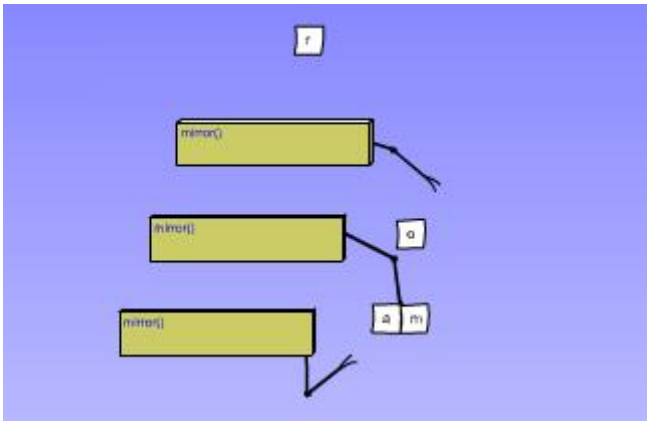
This model was evaluated in 22 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	3	13.64
... represents the idea of a recursive function	2	9.09
... most difficult to follow	8	36.36

Model 2

Description: Function call is represented by an active element with grabs. Full recursion depth.

Duration: 38 seconds
Concreteness: 14 percent
Average watchtime (standard deviation): 83.91 seconds (159.39)



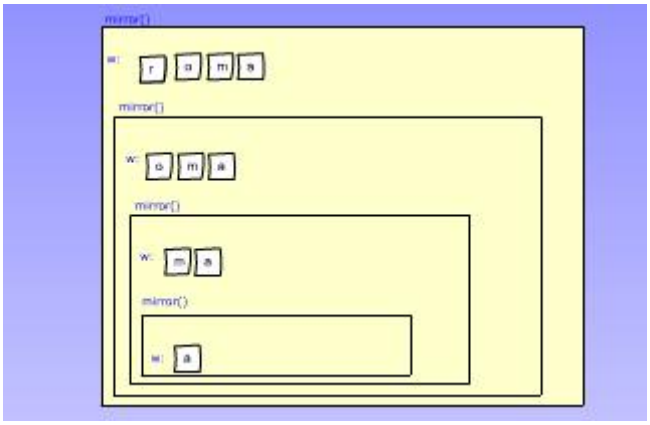
This model was evaluated in 22 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	7	31.82
... represents the idea of a recursive function	7	31.82
... most difficult to follow	4	18.18

Model 3

Description: Recursive function calls are represented by nested boxes. Full recursion depth.

Duration: 31 seconds
Concreteness: 28 percent
Average watchtime (standard deviation): 32.32 seconds (17.90)



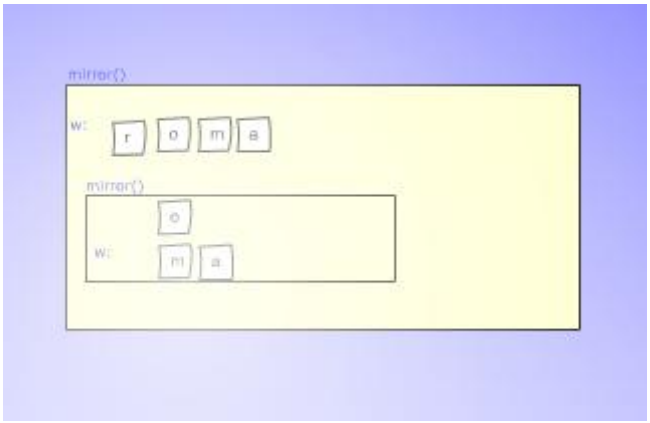
This model was evaluated in 22 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	8	36.36
... represents the idea of a recursive function	10	45.45
... most difficult to follow	4	18.18

Model 4

Description: Recursive function calls are represented by nested boxes. Only one recursive call is shown.

Duration: 17 seconds
Concreteness: 28 percent
Average watchtime (standard deviation): 76.73 seconds (152.29)



This model was evaluated in 22 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	4	18.18
... represents the idea of a recursive function	3	13.64
... most difficult to follow	6	27.27

Analogies for iterations

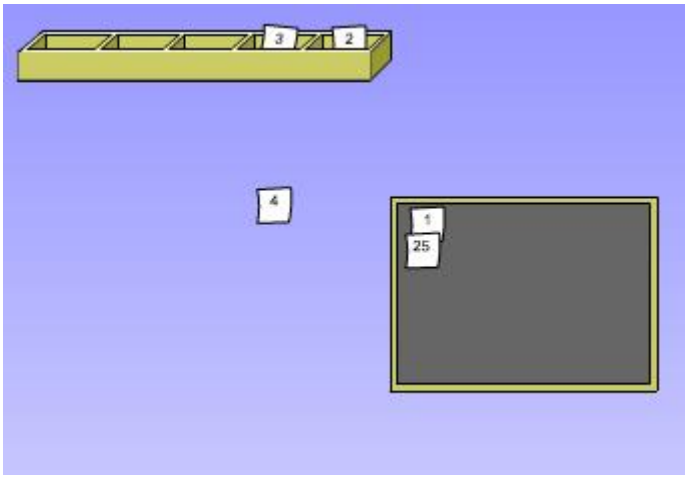
Some general information about the players of this category, who played the game at least *once*.

Professions	66 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	12 female and 54 male persons
Hours a week spent on programming (standard deviation)	5.09 hours (6.25)
Roughly estimated experience in Python programming(standard deviation)	130.23 days (158.32)
Age (standard deviation)	16.64 years (1.28)
Population of the town, where the workshop took place	Less than 100 000: 20, 100 000 to 500 000: 6, more than 500 000: 40
Country	Germany: 52 other country: 14

Model 1

Description: Iteration over a list visualized as box. Elements are taken from the box one by one

Duration: 16 seconds
Concreteness: 0 percent
Average watchtime (standard deviation): 29.32 seconds (11.54)



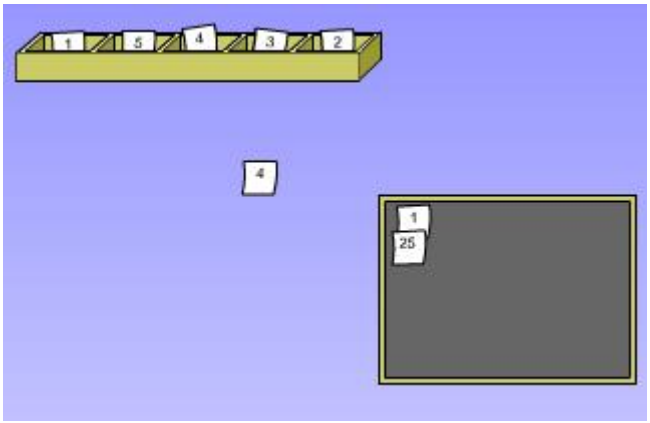
This model was evaluated in 66 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	6	9.09
... do you remember best	8	12.12
... when you imagine the execution of the script	9	13.64

Model 2

Description: Iteration over a list visualized as box. Elements are copied and the copies are processed.

Duration: 16 seconds
Concreteness: 0 percent
Average watchtime (standard deviation): 22.92 seconds (9.30)



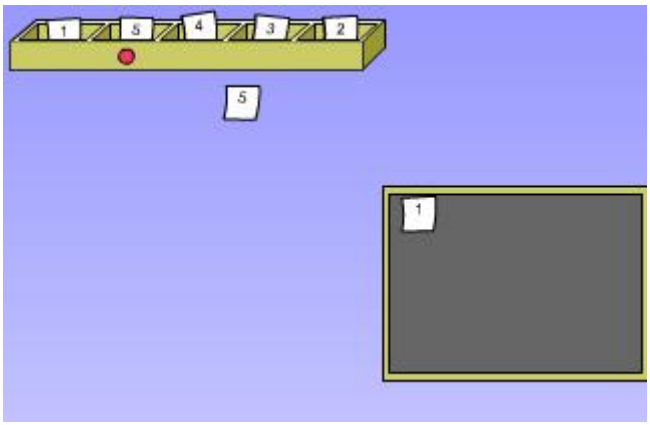
This model was evaluated in 66 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	13	19.70
... do you remember best	8	12.12
... when you imagine the execution of the script	7	10.61

Model 3

Description: Iteration over a list visualized as box. Elements are copied and the copies are processed. Additionally a red point marks the actual element.

Duration: 16 seconds
Concreteness: 0 percent
Average watchtime (standard deviation): 22.47 seconds (14.53)



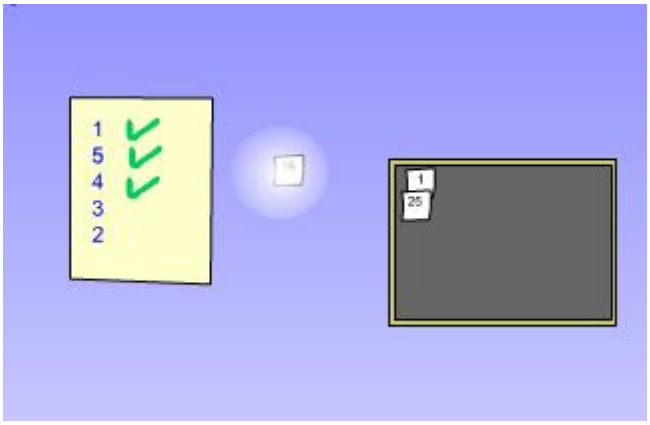
This model was evaluated in 66 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	30	45.45
... do you remember best	20	30.30
... when you imagine the execution of the script	30	45.45

Model 4

Description: Iteration over a list visualized as a column of numbers on a piece of paper. Numbers are copied and processed. Copied numbers are marked.

Duration: 10 seconds
Concreteness: 0 percent
Average watchtime (standard deviation): 17.55 seconds (13.92)



This model was evaluated in 66 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	17	25.76
... do you remember best	30	45.45
... when you imagine the execution of the script	20	30.30

Multilists

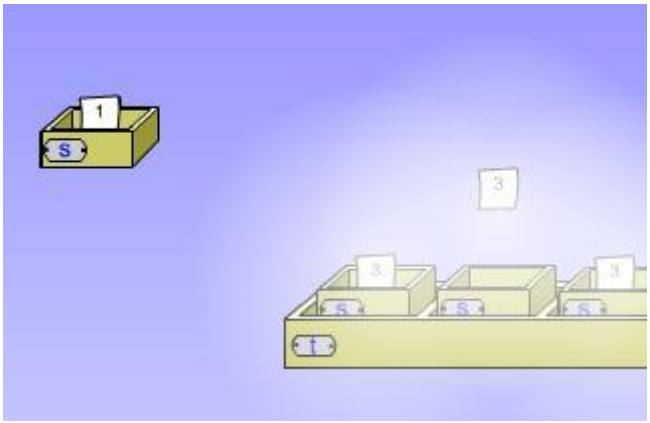
Some general information about the players of this category, who played the game at least *once*.

Professions	6 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	1 female and 5 male persons
Hours a week spent on programming (standard deviation)	3.00 hours (2.10)
Roughly estimated experience in Python programming(standard deviation)	240.50 days (121.79)
Age (standard deviation)	16.33 years (0.82)
Population of the town, where the workshop took place	Less than 100 000: 0, 100 000 to 500 000: 1, more than 500 000: 5
Country	Germany: 6 other country: 0

Model 1

Description: Iteration over a multilist. Ghost model with several ghosts of one original. Changes take place at the same time at all ghosts.

Duration: 11 seconds
Concreteness: 33 percent
Average watchtime (standard deviation): 24.83 seconds (10.72)



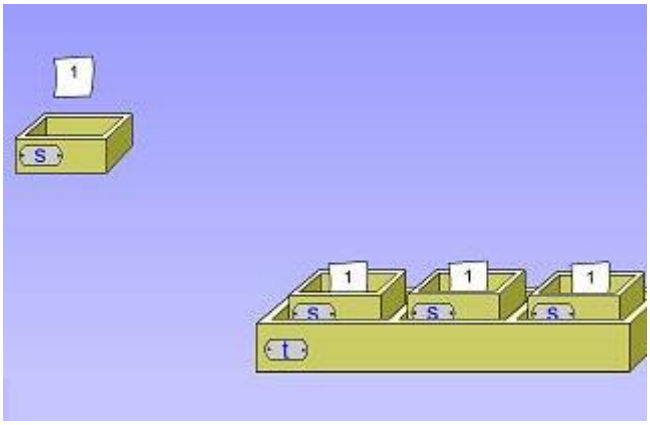
This model was evaluated in 6 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain?	2	33.33
... do you remember best?	1	16.67
... when you imagine the execution?	2	33.33

Model 2

Description: Iteration over a multilist. Ghost model with several ghosts of one original. Changes take place at the same time at all ghosts. Original is highlighted as the origin of all change.

Duration: 11 seconds
Concreteness: 33 percent
Average watchtime (standard deviation): 28.67 seconds (17.19)



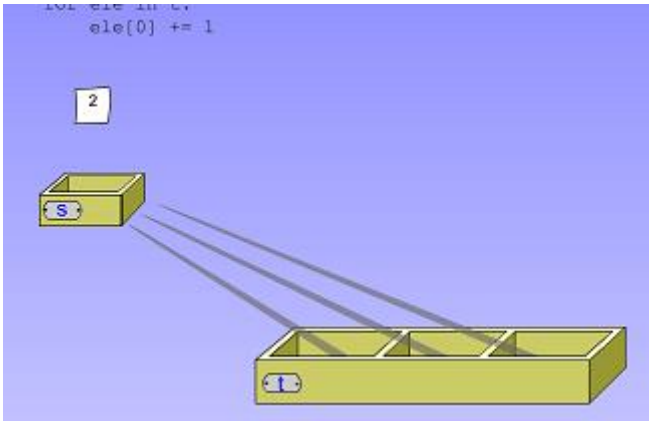
This model was evaluated in 6 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain?	0	0.00
... do you remember best?	2	33.33
... when you imagine the execution?	1	16.67

Model 3

Description: Iteration over a multilist. Mixture of container and pointer model. Top level ist is container with pointers to sublists.

Duration: 11 seconds
Concreteness: 33 percent
Average watchtime (standard deviation): 13.17 seconds (11.00)



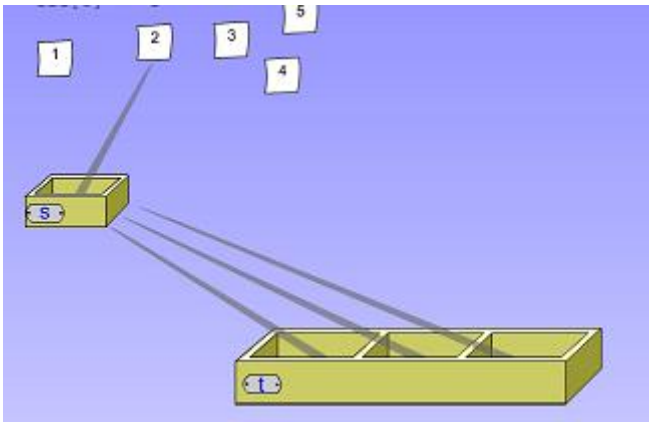
This model was evaluated in 6 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain?	2	33.33
... do you remember best?	1	16.67
... when you imagine the execution?	2	33.33

Model 4

Description: Iteration over a multilist. Consistent pointer model. Top level ist is container with pointers to sublists containing pointers to numbers.

Duration: 11 seconds
Concreteness: 33 percent
Average watchtime (standard deviation): 24.33 seconds (13.49)



This model was evaluated in 6 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain?	2	33.33
... do you remember best?	2	33.33
... when you imagine the execution?	1	16.67

Recursive computation of factorial

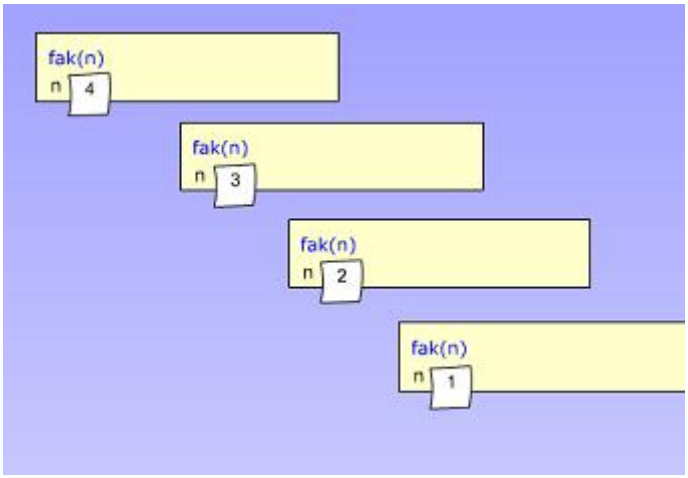
Some general information about the players of this category, who played the game at least *once*.

Professions	28 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	5 female and 23 male persons
Hours a week spent on programming (standard deviation)	2.25 hours (2.72)
Roughly estimated experience in Python programming(standard deviation)	79.29 days (122.83)
Age (standard deviation)	17.39 years (0.74)
Population of the town, where the workshop took place	Less than 100 000: 19, 100 000 to 500 000: 1, more than 500 000: 8
Country	Germany: 28 other country: 0

Model 1

Description: Berechnung Fakultaet, vollstaendige Rekursion (bis zum Abbruch), Execution Frames, white box

Duration: 28 seconds
Concreteness: 80 percent
Average watchtime (standard deviation): 93.25 seconds (42.36)



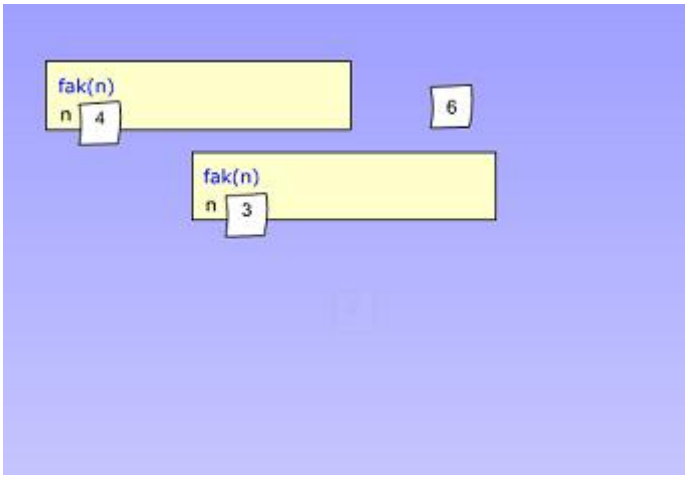
This model was evaluated in 28 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	11	39.29
... represents the idea of a recursive function best	9	32.14
... most difficult to follow	1	3.57

Model 2

Description: Recursive computation of factorial. Processes are represented by execution frames. Only one recursive call.

Duration: 9 seconds
Concreteness: 80 percent
Average watchtime (standard deviation): 38.61 seconds (25.62)



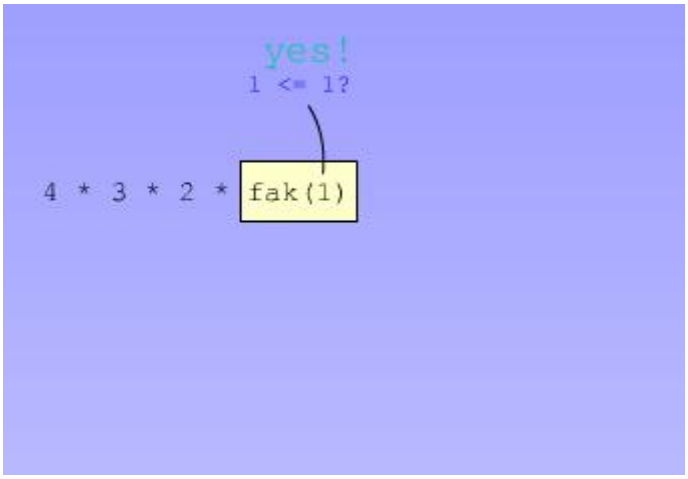
This model was evaluated in 28 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	5	17.86
... represents the idea of a recursive function best	7	25.00
... most difficult to follow	10	35.71

Model 3

Description: Recursive computation of factorial. Record of computation process, function calls are replaced by results. Full recursion depth.

Duration: 23 seconds
Concreteness: 80 percent
Average watchtime (standard deviation): 45.96 seconds (22.78)



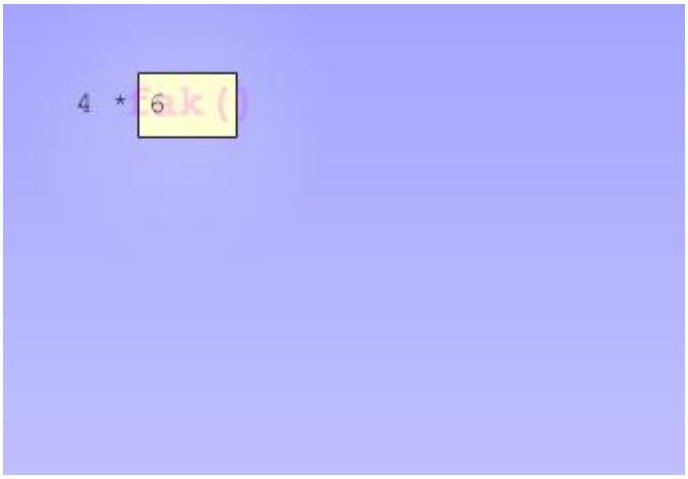
This model was evaluated in 28 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	11	39.29
... represents the idea of a recursive function best	10	35.71
... most difficult to follow	5	17.86

Model 4

Description: Recursive computation of factorial. Record of computation process, function calls are replaced by results. Only one recursive call is visualized.

Duration: 8 seconds
Concreteness: 80 percent
Average watchtime (standard deviation): 22.07 seconds (11.71)



This model was evaluated in 28 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	1	3.57
... represents the idea of a recursive function best	2	7.14
... most difficult to follow	12	42.86

Fibonacci numbers

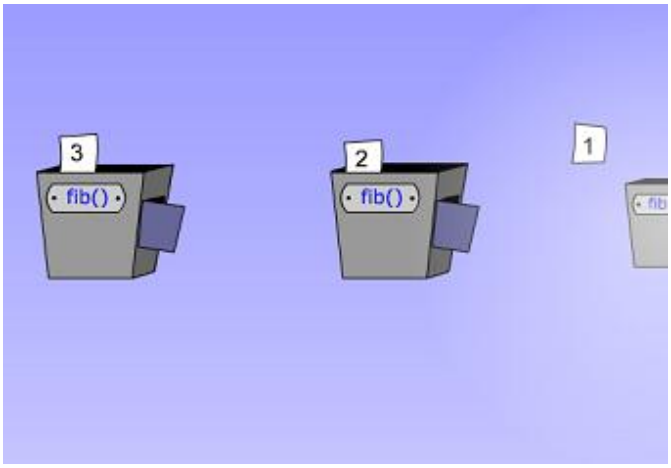
Some general information about the players of this category, who played the game at least *once*.

Professions	19 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	5 female and 14 male persons
Hours a week spent on programming (standard deviation)	1.58 hours (1.64)
Roughly estimated experience in Python programming(standard deviation)	59.21 days (111.83)
Age (standard deviation)	17.42 years (0.61)
Population of the town, where the workshop took place	Less than 100 000: 15, 100 000 to 500 000: 0, more than 500 000: 4
Country	Germany: 19 other country: 0

Model 1

Description: Recursive function that computes Fibonacci numbers. Process is represented by box with side exit. Full recursion depth. New boxes are generated serially.

Duration: 23 seconds
Concreteness: 33 percent
Average watchtime (standard deviation): 50.89 seconds (48.68)



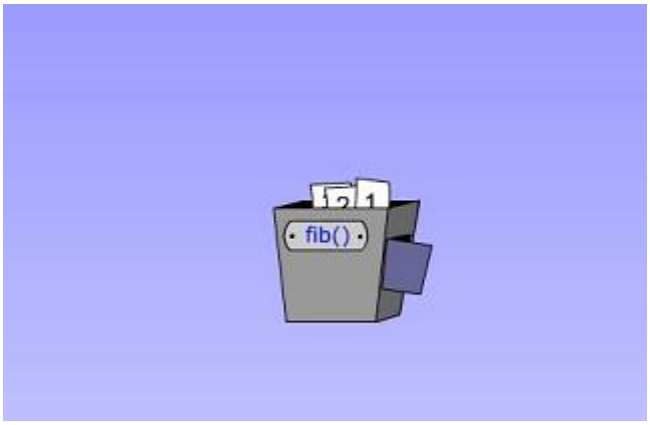
This model was evaluated in 19 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	4	21.05
... represents the idea of a recursive function best	8	42.11
... most difficult to follow	4	21.05

Model 2

Description: Recursive function that computes Fibonacci numbers. Process is represented by only one box with side exit. Full recursion depth. On top of the box (entrance) values are accumulated.

Duration: 21 seconds
Concreteness: 33 percent
Average watchtime (standard deviation): 28.84 seconds (14.42)



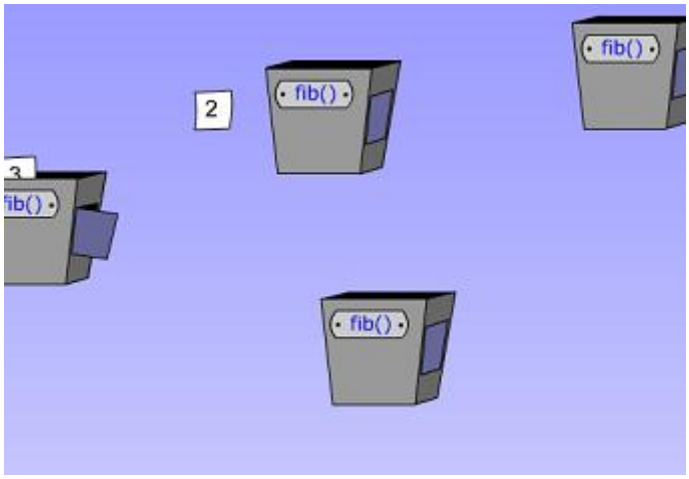
This model was evaluated in 19 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	3	15.79
... represents the idea of a recursive function best	2	10.53
... most difficult to follow	6	31.58

Model 3

Description: Recursive function that computes Fibonacci numbers. Processes are represented by four static boxes with side exit, which are already visible at the beginning. Full recursion depth.

Duration: 19 seconds
Concreteness: 33 percent
Average watchtime (standard deviation): 33.11 seconds (19.78)



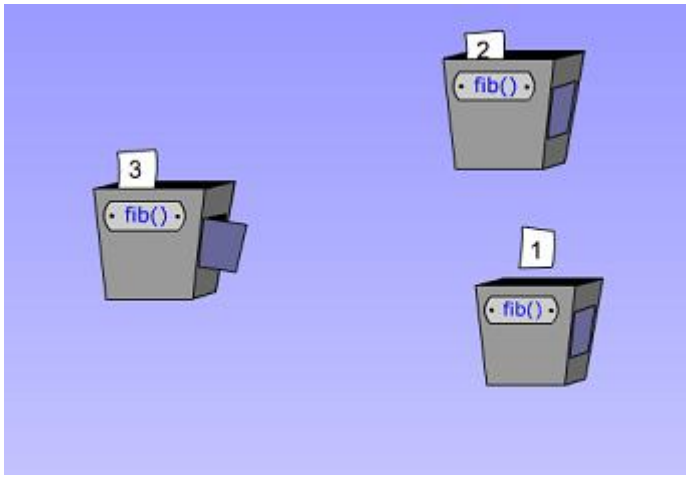
This model was evaluated in 19 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	3	15.79
... represents the idea of a recursive function best	3	15.79
... most difficult to follow	7	36.84

Model 4

Description: Recursive function that computes Fibonacci numbers. Processes are represented by five dynamic boxes with side exit. Parallel execution of function calls. Full recursion depth.

Duration: 15 seconds
Concreteness: 33 percent
Average watchtime (standard deviation): 21.79 seconds (19.68)



This model was evaluated in 19 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	9	47.37
... represents the idea of a recursive function best	6	31.58
... most difficult to follow	2	10.53

What happens, when a function returns something?

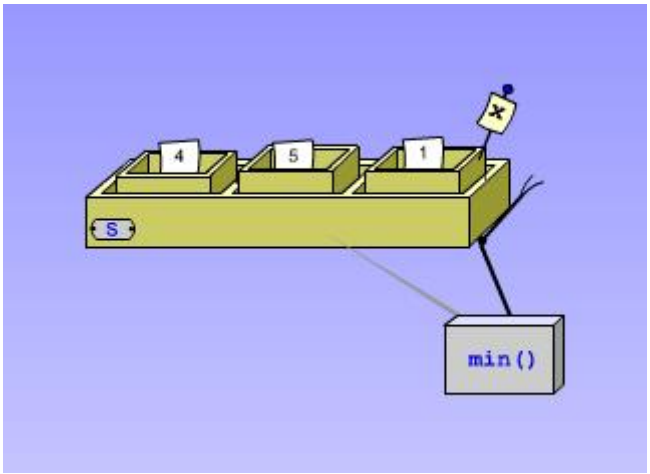
Some general information about the players of this category, who played the game at least *once*.

Professions	16 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	1 female and 15 male persons
Hours a week spent on programming (standard deviation)	6.50 hours (10.84)
Roughly estimated experience in Python programming(standard deviation)	92.06 days (124.06)
Age (standard deviation)	16.38 years (1.41)
Population of the town, where the workshop took place	Less than 100 000: 8, 100 000 to 500 000: 3, more than 500 000: 5
Country	Germany: 16 other country: 0

Model 1

Description: Function is represented by a box, searches for the minimal element of a list and puts a pin named with an x at this element

Duration: 8 seconds
Concreteness: 75 percent
Average watchtime (standard deviation): 36.00 seconds (23.34)



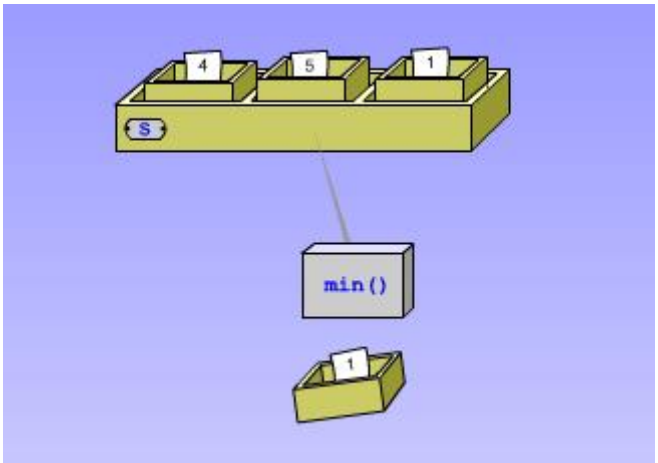
This model was evaluated in 16 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	3	18.75
... do you remember best	2	12.50
... explains worst	10	62.50

Model 2

Description: Function is represented by a box, searches for the minimal element of a list and returns a copy of this element (ghost). Assignment x=10 is done twice with x and the original in the list.

Duration: 10 seconds
Concreteness: 75 percent
Average watchtime (standard deviation): 24.44 seconds (12.21)



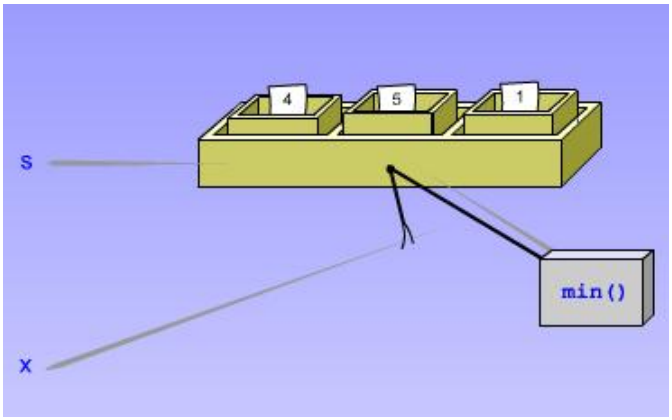
This model was evaluated in 16 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	4	25.00
... do you remember best	5	31.25
... explains worst	2	12.50

Model 3

Description: Function is represented by a box. Before the function appears, the name x and a pointer enters the scene. The funnction finds the minimal element of a list and moves the pointer x to it.

Duration: 12 seconds
Concreteness: 75 percent
Average watchtime (standard deviation): 25.06 seconds (12.01)



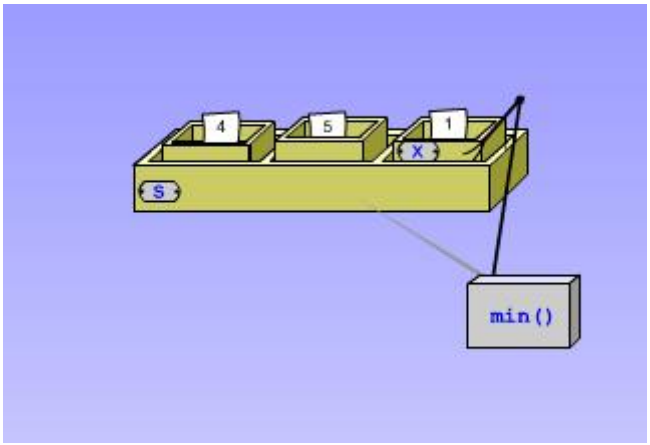
This model was evaluated in 16 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	6	37.50
... do you remember best	4	25.00
... explains worst	3	18.75

Model 4

Description: Function is represented by a box. Before the function appears, the name x (without object) enters the scene. The function finds the minimal element of a list and puts name x on it.

Duration: 11 seconds
Concreteness: 75 percent
Average watchtime (standard deviation): 19.56 seconds (4.38)



This model was evaluated in 16 first subjective sessions. The following table shows the results.

Question	votes	percentage
... would you use to explain	3	18.75
... do you remember best	5	31.25
... explains worst	1	6.25

3.6 Python Puzzle

3.6.1 Screenshots aus einer Sitzung mit dem Python Puzzle „Modeling a group“

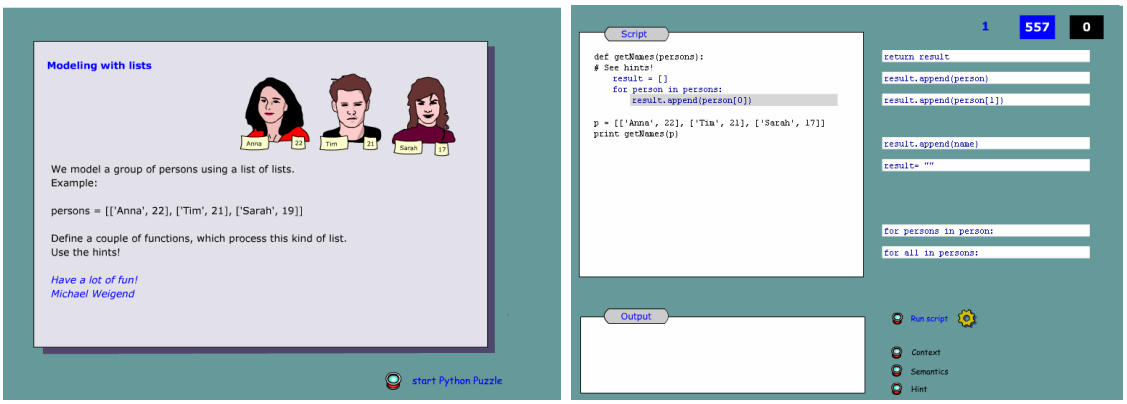


Abb. 96: Problemkontext (links) und Editorseite (rechts).

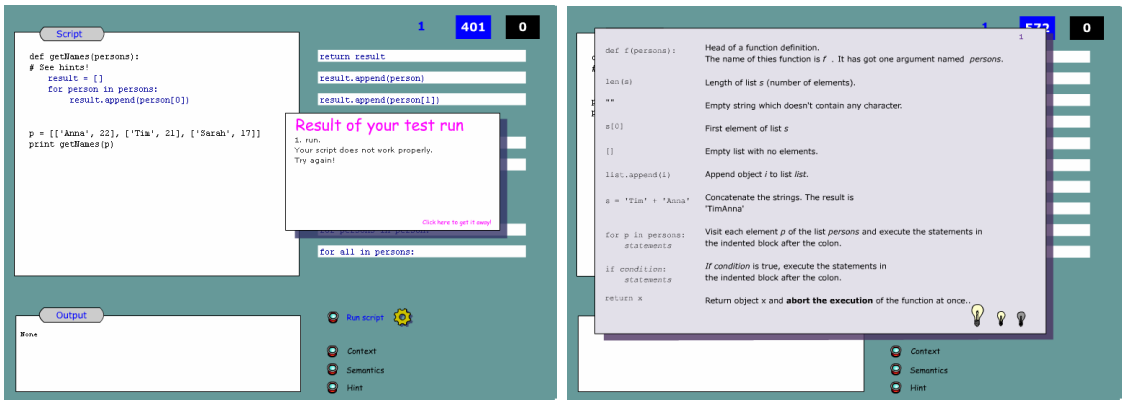


Abb. 97: Feedback zu einem Testlauf des Programms (links) und Kurzreferenz zur den vorkommenden Python-Sprachelementen (erscheint nach Klick auf „Semantics“).

3.6.2 Dokumentation einer Session mit XML

Zu Beginn einer Session wird ein XML-Paket an das Service-Programm pp_startservice.py geschickt. Es enthält u.a. die IDs (Passwörter) der Spieler und hat folgendes Format:

```
<log>
    <session game="gameId" id ="SessionID" />
    <person id="id1"  />
    <person id ="id2" />
</log>
```

Vom Serviceprogramm wird ein Datensatz (Tupel der Relation `protocol_pp`) für die Beschreibung der Session angelegt , als Spieldauer 0 Sekunden und als erreichte Punktzahl `-200` eingetragen.

Am Ende einer Session wird ein XML-Paket mit dem Sessionprotokoll an das Service-Programm `pp_service.py` geschickt. Es ist folgendermaßen aufgebaut:

```
<log>
   <task id = "Nummer" time="Bearbeitungsbeginn der Aufgabe">
       <model id="Primärschlüssel des Modells"
              time="Zeitpunkt der Betrachtung"
              watchtime="Dauer der Betrachtung"
              eval="Bewertung (1 bis 3)" />
```

```
        <correct time = "zeitpunkt"> Programmzeile </correct>
        …
        <error time = "zeitpunkt"> Programmzeile </error>
        …
        <time>Bearbeitungszeit</time>
        <points>erreichte Punktzahl</points>
        <runs>Anzahl der Testläufe</runs>
    </task>
    …
    <session id = "SessionID"
          game= "gameID"
          time="Spieldauer in Sekunden"
          points="erreichte Punktzahl"/>
</log>
```

Für jede gelöste Aufgabe gibt es einen task-Knoten (mit Nummer der Aufgabe und Bearbeitungsbeginn als Attribut). Er erhält Unterknoten mit Informationen zu folgenden Punkten:

- Eine Beschreibung der Modelle, die verwendet worden sind: Primärschlüssel (id), Zeitpunkt der Betrachtung, Dauer, Bewertung (1= "hat nicht geholfen", 3= "hat geholfen"). Jedes Mal, wenn die Hint-Schaltfläche angeklickt worden ist, wird ein solcher model-Knoten erzeugt.
- Eine Beschreibung der verwendeten Programmzeilen ("Bausteine"). Die correct-Knoten enthalten Beschreibungen der Programmzeilen des korrekten, getesteten Programms (Programmzeile und Zeitpunkt, wann der Baustein mit dieser Zeile zum ersten Mal bewegt worden ist). Die error-Knoten enthalten entsprechende Beschreibungen von fehlerhaften Zeilen. Eine Zeile wird als fehlerhaft gewertet, wenn sie nicht zum korrekten Programm gehört und dennoch irgendwann einmal während der Bearbeitung der Aufgabe in das Programmfeld bewegt worden ist. In diesem kann man davon ausgehen, dass der Spieler zu irgendeinem Zeitpunkt glaubte, die Zeile müsse in den Programmtext eingebaut werden. Es kann sein, dass ihm oder ihr der Fehler sofort aufgefallen ist oder vielleicht auch erst nach einem misslungenen Testlauf. Auf jeden Fall ist der Fehler später korrigiert worden.
- Bearbeitungszeit für diese Aufgabe in Sekunden.
- Erreichte Punktzahl (entspricht dem Schwierigkeitsgrad).
- Anzahl der Testläufe, bis das Programm fehlerfrei war.

Nach den task-Knotengibt es noch einen session-Knoten mit dem Primärschlüssel der Session in der entsprechenden Tabelle der Datenbank, der gesamten Spieldauer und der insgesamt erreichten Punktzahl. Beispiel für ein XML-Dokument:

```
<log>
<task id="1" time="50">
<model id="group1_task1" time="46" watchtime="1" eval="2" />
<model id="group1_task1" time="44" watchtime="1" eval="3" />
<correct time="28">return result</correct>
<correct time="37">result = []</correct>
<correct time="32">result.append(person[0])</correct>
<correct time="34">for person in persons:</correct>
<time>25</time>
<points>60</points>
<runs>1</runs>
</task>
<session id = "pp_group1508063" game="pp_group1" time="600"
         points="60" />
</log>
```

3.6.3 Beispiel für eine automatisch generierte statistische Auswertung

Kontext

Der nachfolgende Auszug aus dem Bericht des Auswertungsdienstes für Python Puzzles zeigt die Ergebnisse für Aufgabe 1 aus dem Puzzle *Modeling a group*. Der Aufgabenkontext lautet:

Wir modellieren eine Personengruppe durch eine Liste von Listen.
Beispiel:
`p = [['Anna', 22], ['Tim', 21], ['Sarah', 19]]`
*Definieren Sie verschiedene Funktionen, in denen eine solche Liste verarbeitet wir*d.

In Aufgabe 1 soll eine Funktion definiert werden, die eine Liste mit den Namen der Gruppenmitglieder liefert. Zu dieser Funktionsdefinition ist nur der Funktionskopf vorgegeben:

```
def getNames(persons):
```

Die Anweisungen des Funktionskörpers müssen eingefügt werden. Angeboten werden Puzzlestücke mit korrekten und falschen Anweisungen. Die korrekte Lösung lautet:

```
def getNames(persons):
   result = []
   for person in persons:
       result.append(person[0])
   return result
```

Zu dieser Aufgabe wurde nur eine einzige Animation angeboten, die auch als nonverbale Beschreibung der erwarteten Funktionalität gesehen werden kann.

Auszug aus der automatisch erstellten Auswertung

Modeling a group (pp_group1)

Maximal time: 600 seconds

Objective number of sessions	54 sessions
Subjective number of sessions (taking into accout that there are sometimes two players working together)	69 sessions
Average number of times (canceled and uncanceled sessions) a player played this game (standard deviation)	1.23 sessions (0.50)
Average number of players in all objective sessions (standard deviation)	1.28 (0.45) players
Objective number of uncancelled sessions (including sessions, in which no task was solved)	33 sessions
Subjective number of uncancelled sessions (including sessions, in which no task was solved)	40 sessions
Average number of points in all objective uncanceled sessions (standard deviation)	84.94 (94.68) points

Task 1

Successful solutions

Number of successful subjective solutions of task	24 solutions
Average number of test runs (standard deviation)	3.30 runs (3.21)
Average solution time (standard deviation)	319.91 seconds (191.84)

Some general information about the players of this category, who solved this task.

Professions	20 highschool students, 4 university students, 0 teachers, 0 professors and 0 others
Gender	4 female and 20 male persons
Hours a week spent on programming (standard deviation)	5.29 hours (5.77)
Age (standard deviation)	17.54 years (4.51)
Country	DEU: 22 USA: 2

Use of correct lines of code (pieces) in task 1. The rank of a piece indicates the time, when it was put into the program. We consider only successful solutions.

Correct line of code	Usage (percent)	Rank (standard deviation)
`result = []`	24 (100.00 percent)	1.38 (0.88)
`return result`	24 (100.00 percent)	2.96 (0.95)
`for person in persons:`	24 (100.00 percent)	2.67 (1.05)
`result.append(person[0])`	24 (100.00 percent)	3.00 (0.78)

Use of wrong lines of code (errors) in task 1. The rank of an error indicates the time, when a player tried to insert it into the program.

Error	Usage (percent)	Rank (standard deviation)
`for all in persons:`	8 (33.33 percent)	2.62 (1.69)
`for persons in person:`	11 (45.83 percent)	2.82 (1.66)
`result.append(name)`	7 (29.17 percent)	2.14 (1.35)
`result.append(person)`	14 (58.33 percent)	1.71 (1.14)
`result.append(person[1])`	6 (25.00 percent)	3.33 (1.03)
`result= ""`	5 (20.83 percent)	2.60 (1.82)

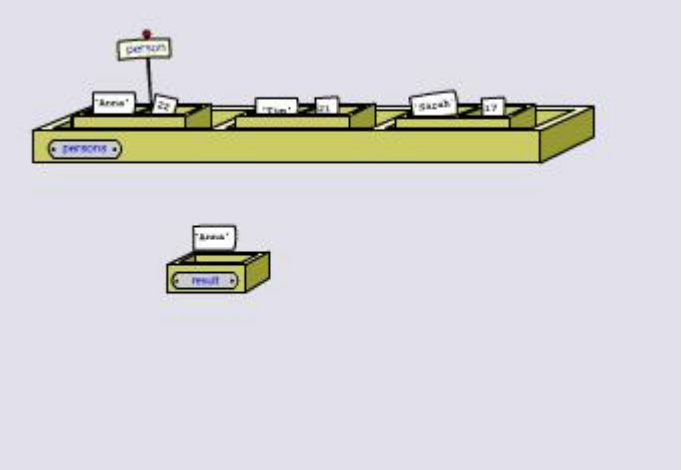
Use of visual models in successful solutions

The following tables show, how visual models are used while solving this task. We only consider subjective sessions, in which the task was actually solved.

Model group1_task1

Python Puzzle: Modeling a group
Task: 1
Description of model: Iteration over a list containing three lists, represented by a box containing three boxes. The actual element is marked by a pin.

Level of concreteness (percentage of names): 60 percent
Duration: 17 seconds



In how many subjective sessions this task was solved?	24 sessions
How often was this model watched altogether (successful task solutions only)?	26 times
In how many subjective sessions was this model used (successful task solutions only)?	16 sessions
We consider the players, who have actually used this model. How often did such a player watch this model in a session? (standard deviation)	1.62 times (0.96)
For how long did a player watch this model altogether during a session? (standard deviation)	40.44 seconds (21.56)
How useful was this model (average of all ratings in a session)? 3 = very useful, 2 = a bit useful, 1 = not useful (standard deviation)	2.27 (0.79)
How useful was this model (best of all ratings in a session)? 3 = very useful, 2 = a bit useful, 1 = not useful (standard deviation)	2.38 (0.81)

3.7 Python Quiz

3.7.1 Dokumentation einer Session

Zu Beginn einer Session wird ein XML-Paket an das Service-Programm `pq_startservice.py` geschickt. Es enthält unter anderem die Passwörter der Spieler und hat folgendes Format:

```
<log>
   <session game="gameId" id ="SessionID" />
   <person id = "id1"  />
   <person id = "id2" />
</log>
```

Vom Serviceprogramm wird ein Datensatz (Tupel der Relation `protocol_pq`) für die Beschreibung der Session angelegt, als Spieldauer `0` Sekunden und als erreichte Punktzahl `-200` eingetragen. Bei einem vorzeitigen Abbruch der Session, erhalten somit die Spieler Minuspunkte und der Sessionabbruch wird "bestraft". Am Ende einer Session wird ein XML-Paket mit dem Sessionprotokoll an das Service-Programm `pq_service.py` geschickt. Es ist folgendermaßen aufgebaut:

```
<log>
   <task id = "ID der Aufgabenbeschreibung">
       <model id="ID des Modells"
              watchtime="Dauer der Betrachtung"
              points="gesetzte Punkte"
              eval="Bewertung (0 oder 1)" />
       ...
   </task>
   ...
   <session id = "SessionID"
         time="Spieldauer in Sekunden"
         points="erreichte Punktzahl">
</log>
```

Für jede gelöste Aufgabe gibt es einen task-Knoten mit der ID der Aufgabenbeschreibung (Primärschlüsselattribut der Relation `description_pq_task`) als Attribut. Er erhält als Unterknoten Beschreibungen der Modelle und ihre Bewertung. Der `session`-Knoten enthält Daten des Sitzungsprotokolls, die nach dem Spiel in der Relation `protocol_pq` geändert werden müssen.

3.7.2 Auszug aus der automatisch erstellten Auswertung

Dieser Auszug aus der automatisch erstellten Auswertung beschränkt sich auf die ersten, nicht vorzeitig abgebrochenen, subjektiven Sitzungen von Schülerinnen und Schülern, die an einem Workshop teilgenommen haben. Wenn es Schüler gab, die ein Quiz drei Mal oder häufiger durchgeführt haben, wurde eine Übersicht über die Ergebnisse der ersten drei (nicht abgebrochenen) Sitzungen generiert.

Modeling a group using a list of tuples.

This game contains 5 tasks. I n each task the player has to judge several models, whether or not they are appropriate to explain the execution of some program statement.

Task 1	`result=[]`
Task 2	`for (n, a) in persons:`
Task 3	`a > age`
Task 4	`result.append(n)`
Task 5	`olderThan(group, 19)`

The following table provides some general information about the usage of this game (pq_list).

Number of objective sessions (*not* taking into account that there are sometimes two players working together)	106 sessions
Number of subjective sessions (taking into account that there are sometimes two players working together)	116 sessions
Number of subjective uncanceled sessions (taking into account that there are sometimes two players working together)	77 sessions
Number of subjective *first* uncanceled sessions (taking into account that there are sometimes two players working together and some people play this game several times)	68 sessions
Number of persons, who played three times or more	2 persons

Some general information about the players of this category, who played the game at least *once*.

Professions	68 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	12 female and 56 male persons
Hours a week spent on programming (standard deviation)	3.81 hours (5.96)
Roughly estimated experience in Python programming (standard deviation)	91.63 days (138.80)
Age (standard deviation)	17.15 years (2.12)
Population of the town, where the workshop took place	Less than 100 000: 13, 100 000 to 500 000: 9, more than 500 000: 46
Country	Germany: 41 other country: 27

Some general information about the players of this category, who played the game at least *three times*.

Professions	2 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	1 female and 1 male persons
Hours a week spent on programming (standard deviation)	1.50 hours (2.12)
Roughly estimated experience in Python programming(standard deviation)	136.50 days (188.80)
Age (standard deviation)	17.00 years (1.41)
Population of the town, where the workshop took place	Less than 100 000: 0, 100 000 to 500 000: 0, more than 500 000: 2
Country	Germany: 1 other country: 1

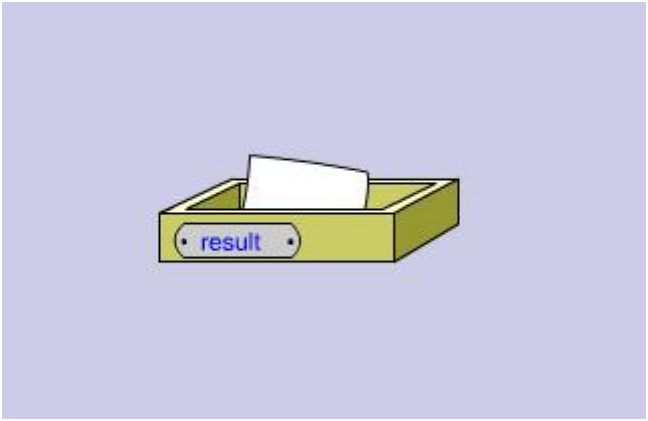
Task 1

Judging Model *pq_list_a1_1*

Associated program:
`result=[]`

Description: Visualization of an empty list by a box containing an empty card

Time: 0 seconds
Concreteness: 100 percent of names are used in the model



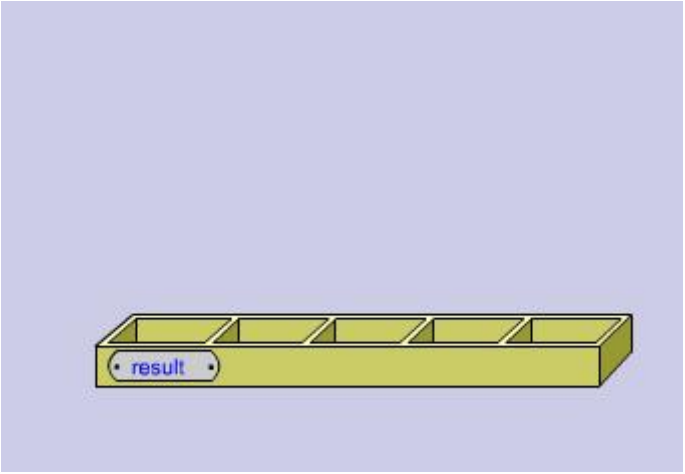
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.40 (4.04)
Average watch time (standard deviation)	17.25 seconds (21.64)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	41 players, 60.29 percent
Average confidence in "good"-judgement (standard deviation)	6.10 (3.95)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	27 players, 39.71 percent
Average confidence in "bad" judgement (standard deviation)	6.85 (4.19)

Judging Model *pq_list_a1_2*

Associated program:
`result=[]`

Description: Visualization of an empty list by a box with five empty compartments.

Time: 0 seconds
Concreteness: 100 percent of names are used in the model



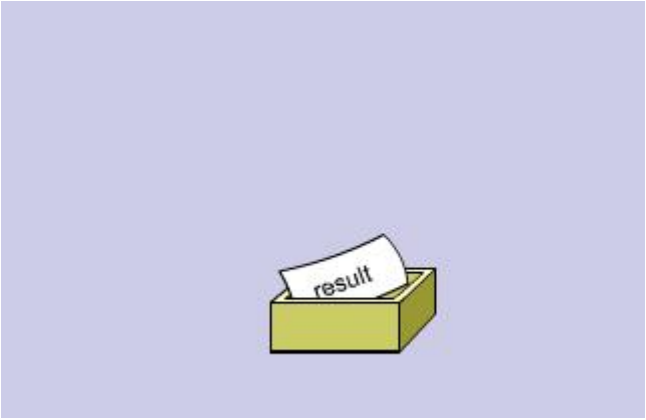
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	5.81 (4.46)
Average watch time (standard deviation)	25.12 seconds (74.37)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	47 players, 69.12 percent
Average confidence in "good"-judgement (standard deviation)	6.17 (4.45)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	21 players, 30.88 percent
Average confidence in "bad" judgement (standard deviation)	5.00 (4.47)

Judging Model *pq_list_a1_3*

Associated program:
```
result=[]
```

Description: Visualization of an empty list named result by a box containing a card with result written on it.

Time: 0 seconds
Concreteness: 100 percent of names are used in the model



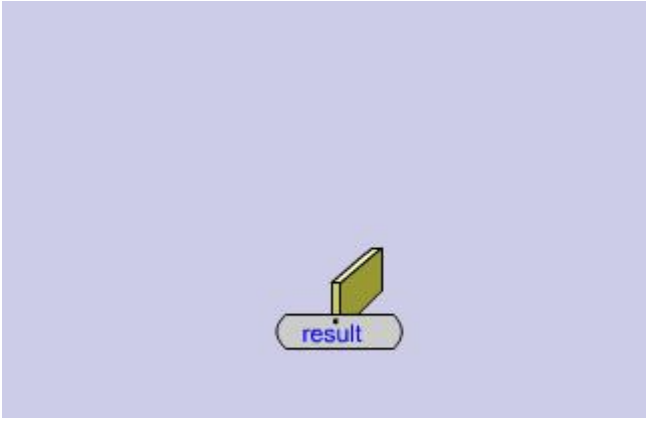
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.84 (4.04)
Average watch time (standard deviation)	18.82 seconds (27.59)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	34 players, 50.00 percent
Average confidence in "good"-judgement (standard deviation)	5.74 (4.11)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	34 players, 50.00 percent
Average confidence in "bad" judgement (standard deviation)	7.94 (3.72)

Judging Model *pq_list_a1_4*

Associated program:
```
result=[]
```

Description: Visualization of an empty list by a bord (box with zero compartments).

Time: 0 seconds
Concreteness: 100 percent of names are used in the model



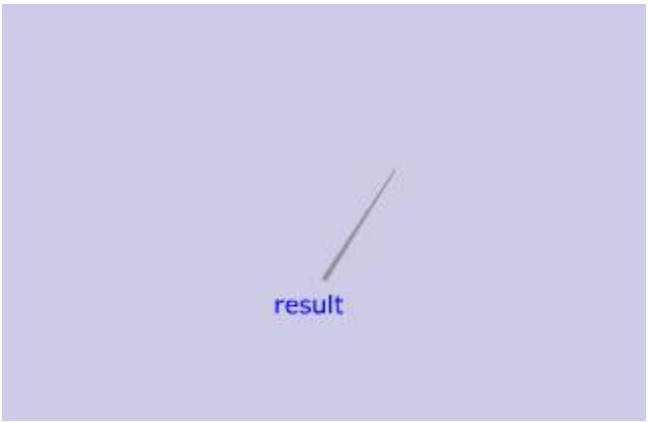
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	5.51 (4.50)
Average watch time (standard deviation)	17.40 seconds (17.32)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	41 players, 60.29 percent
Average confidence in "good"-judgement (standard deviation)	5.61 (4.50)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	27 players, 39.71 percent
Average confidence in "bad" judgement (standard deviation)	5.37 (4.58)

Judging Model *pq_list_a1_5*

Associated program:
```
result=[]
```

Description: Visualization of an empty list by a pointer pointing to nothing.

Time: 0 seconds
Concreteness: 100 percent of names are used in the model



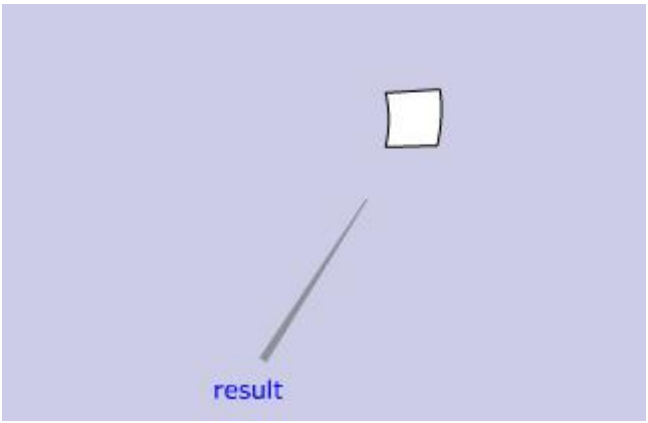
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	5.96 (4.43)
Average watch time (standard deviation)	19.99 seconds (27.56)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	44 players, 64.71 percent
Average confidence in "good"-judgement (standard deviation)	5.23 (4.57)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	24 players, 35.29 percent
Average confidence in "bad" judgement (standard deviation)	7.29 (3.90)

Judging Model *pq_list_a1_6*

Associated program:
```
result=[]
```

Description: Visualization of an empty list by a pointer pointing to an empty card.

Time: 0 seconds
Concreteness: 100 percent of names are used in the model



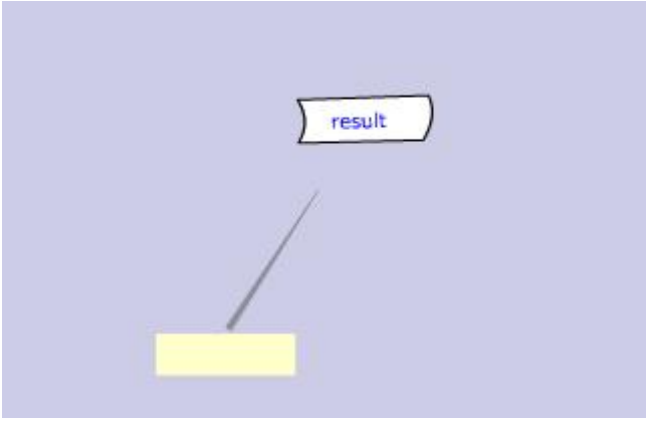
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.32 (4.11)
Average watch time (standard deviation)	14.01 seconds (17.25)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	44 players, 64.71 percent
Average confidence in "good"-judgement (standard deviation)	6.59 (4.14)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	24 players, 35.29 percent
Average confidence in "bad" judgement (standard deviation)	5.83 (4.08)

Judging Model *pq_list_a1_7*

Associated program:
```
result=[]
```

Description: Visualization of an empty list named result by a pointer pointing to the word result.

Time: 0 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.18 (4.15)
Average watch time (standard deviation)	16.50 seconds (21.98)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	29 players, 42.65 percent
Average confidence in "good"-judgement (standard deviation)	5.34 (4.21)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	39 players, 57.35 percent
Average confidence in "bad" judgement (standard deviation)	6.79 (4.05)

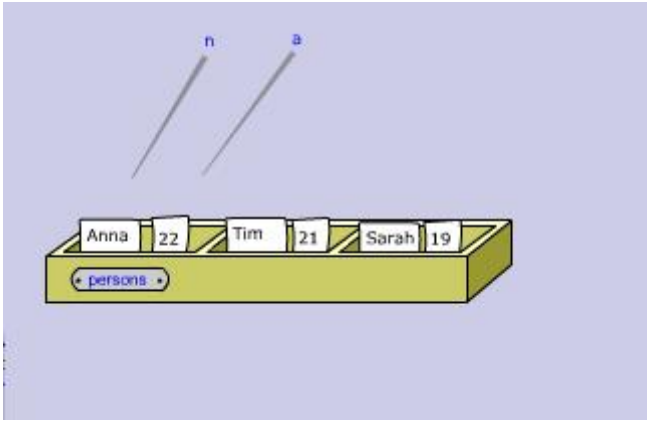
Task 2

Judging Model *pq_list_a2_8*

Associated program:
```
for (n, a) in persons:
```

Description: Visualization of an iteration using a cointainer to represent a list and pointers for iteration variables.

Time: 7 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.35 (3.71)
Average watch time (standard deviation)	10.28 seconds (11.09)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	54 players, 79.41 percent
Average confidence in "good"-judgement (standard deviation)	7.96 (3.30)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	14 players, 20.59 percent
Average confidence in "bad" judgement (standard deviation)	5.00 (4.39)

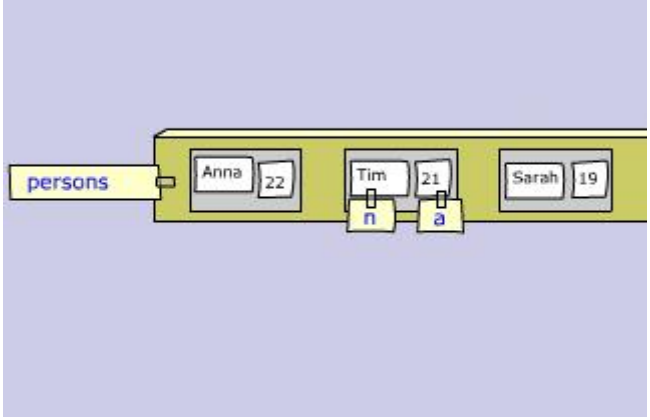
Judging Model *pq_list_a2_7*

Associated program:

```
for (n, a) in persons:
```

Description: Visualization of an iteration. Post-its with the names of the iteration variables are moving from item to item in the list

Time: 9 seconds
Concreteness: 100 percent of names are used in the model



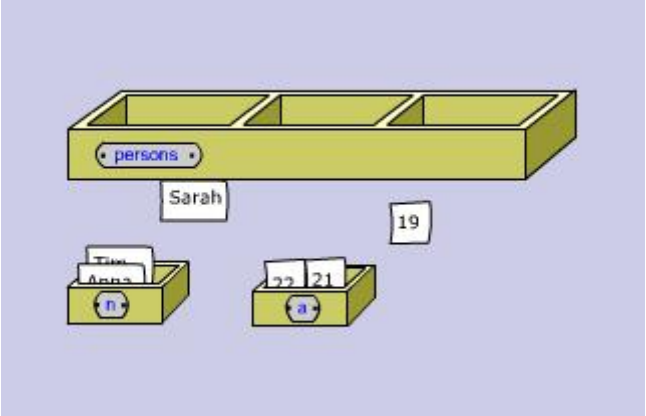
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.35 (3.71)
Average watch time (standard deviation)	12.40 seconds (17.74)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	54 players, 79.41 percent
Average confidence in "good"-judgement (standard deviation)	7.59 (3.60)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	14 players, 20.59 percent
Average confidence in "bad" judgement (standard deviation)	6.43 (4.13)

Judging Model *pq_list_a2_6*

Associated program:
```
for (n, a) in persons:
```

Description: Visualization of an iteration. Contains an unappropriate model of assignment. Cards (representing values) accumulate in a container.

Time: 10 seconds
Concreteness: 100 percent of names are used in the model



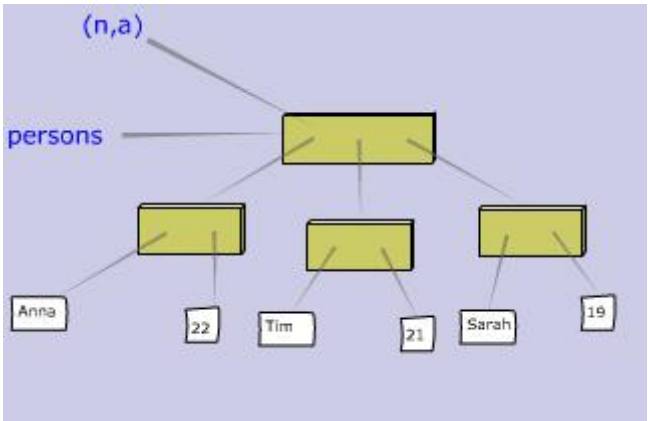
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.57 (3.81)
Average watch time (standard deviation)	9.78 seconds (5.97)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	35 players, 51.47 percent
Average confidence in "good"-judgement (standard deviation)	7.29 (3.90)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	33 players, 48.53 percent
Average confidence in "bad" judgement (standard deviation)	7.88 (3.76)

Judging Model *pq_list_a2_5*

Associated program:
```
for (n, a) in persons:
```

Description: Visualization of an iteration using a pointer model for the list and the iteration variable. The pointer of the iteration variable moves.

Time: 4 seconds
Concreteness: 100 percent of names are used in the model



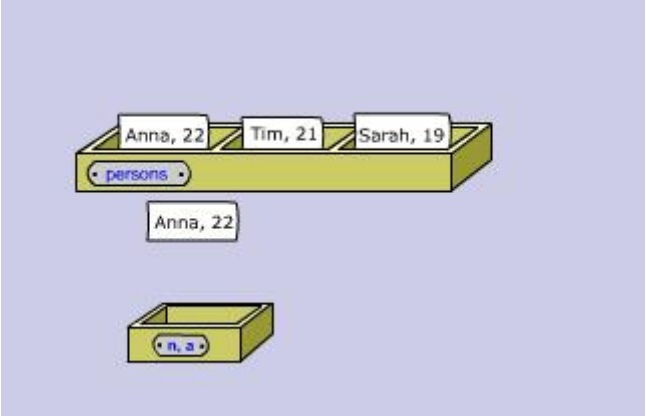
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.03 (4.19)
Average watch time (standard deviation)	27.22 seconds (93.78)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	42 players, 61.76 percent
Average confidence in "good"-judgement (standard deviation)	6.19 (4.25)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	26 players, 38.24 percent
Average confidence in "bad" judgement (standard deviation)	5.77 (4.17)

Judging Model *pq_list_a2_4*

Associated program:
```
for (n, a) in persons:
```

Description: Visualization of an iteration using a container model for the list and the iteration variable. Copies of cards with tuples move out of the list-container.

Time: 10 seconds
Concreteness: 100 percent of names are used in the model



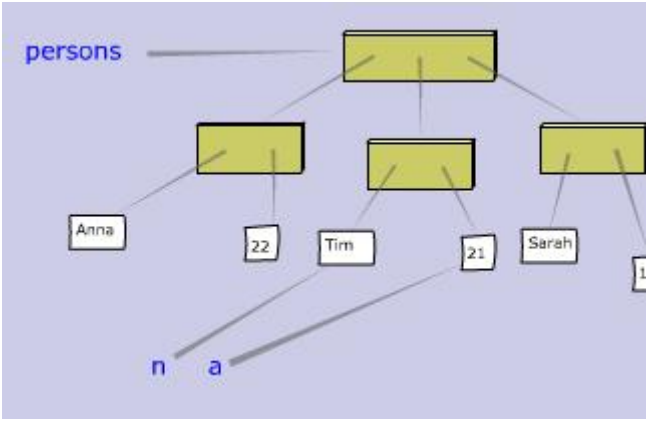
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.50 (3.72)
Average watch time (standard deviation)	9.06 seconds (7.43)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	43 players, 63.24 percent
Average confidence in "good"-judgement (standard deviation)	6.86 (4.09)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	25 players, 36.76 percent
Average confidence in "bad" judgement (standard deviation)	8.60 (2.71)

Judging Model *pq_list_a2_3*

Associated program:
```
for (n, a) in persons:
```

Description: Visualization of an iteration using a pointer model for the list and the iteration variable. The pointers of the iteration variable move.

Time: 4 seconds
Concreteness: 100 percent of names are used in the model



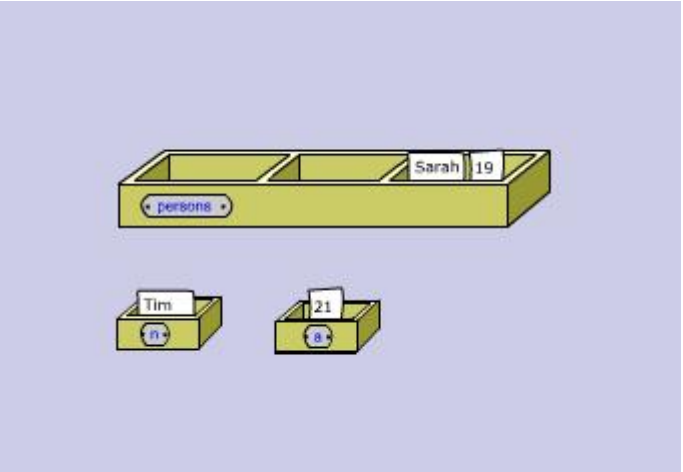
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.50 (3.91)
Average watch time (standard deviation)	19.40 seconds (65.88)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	57 players, 83.82 percent
Average confidence in "good"-judgement (standard deviation)	7.46 (3.91)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	11 players, 16.18 percent
Average confidence in "bad" judgement (standard deviation)	7.73 (4.10)

Judging Model *pq_list_a2_2*

Associated program:
```
for (n, a) in persons:
```

Description: Visualization of an iteration using a container model for the list and the iteration variable. The cards in the list-container are removed.

Time: 10 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.72 (3.81)
Average watch time (standard deviation)	8.81 seconds (6.22)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	40 players, 58.82 percent
Average confidence in "good"-judgement (standard deviation)	7.50 (3.92)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	28 players, 41.18 percent
Average confidence in "bad" judgement (standard deviation)	8.04 (3.69)

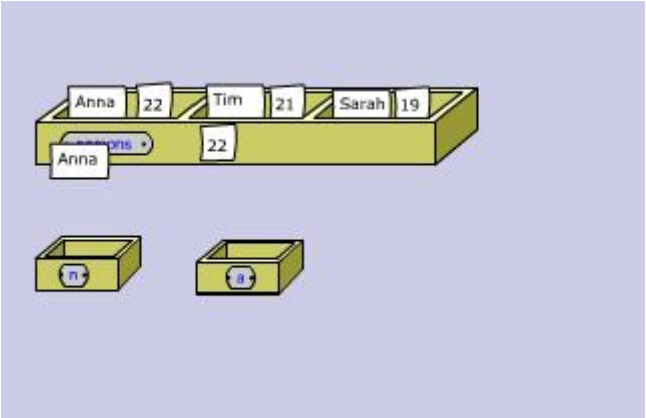
Judging Model *pq_list_a2_1*

Associated program:

```
for (n, a) in persons:
```

Description: Visualization of an iteration using a container model for the list and the iteration variable. Copies of the cards in the list-container move.

Time: 10 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.35 (3.91)
Average watch time (standard deviation)	11.18 seconds (10.50)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	49 players, 72.06 percent
Average confidence in "good"-judgement (standard deviation)	7.45 (3.97)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	19 players, 27.94 percent
Average confidence in "bad" judgement (standard deviation)	7.11 (3.84)

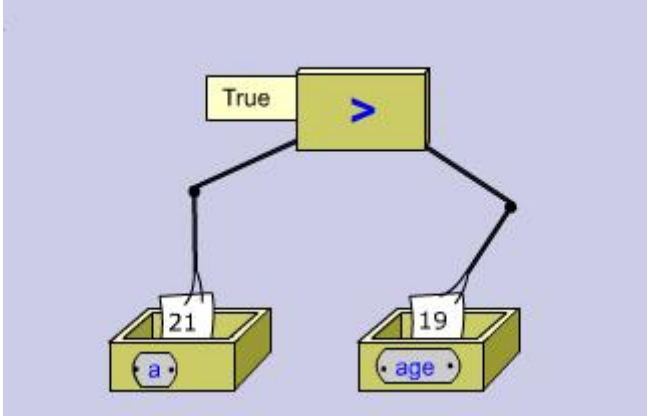
Task 3

Judging Model *pq_list_a3_4*

Associated program:
`a > age`

Description: Visualization of a greater-than-function by a box with sensors for the input values.

Time: 8 seconds
Concreteness: 100 percent of names are used in the model



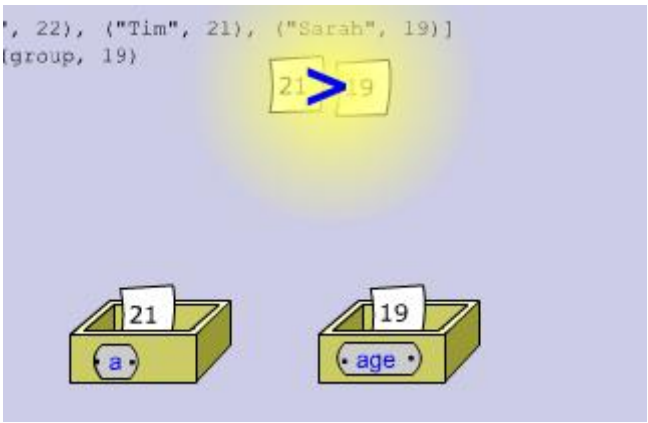
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.13 (3.70)
Average watch time (standard deviation)	10.65 seconds (7.18)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	58 players, 85.29 percent
Average confidence in "good"-judgement (standard deviation)	7.41 (3.66)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	10 players, 14.71 percent
Average confidence in "bad" judgement (standard deviation)	5.50 (3.69)

Judging Model *pq_list_a3_2*

Associated program:
`a > age`

Description: Visualization of the greater-than operation using a flash for the function call and a cointainer model for the compared variables.

Time: 7 seconds
Concreteness: 100 percent of names are used in the model



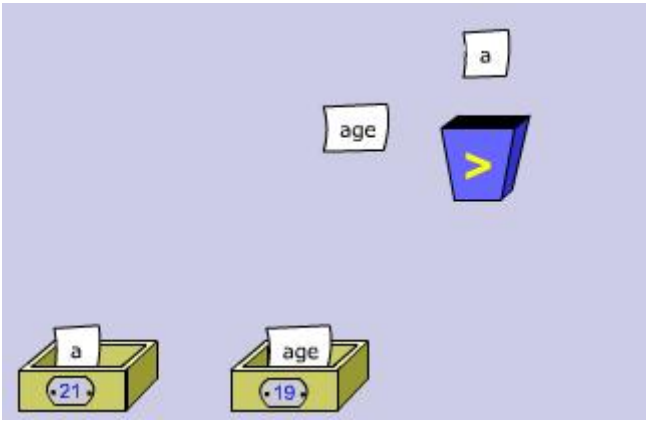
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.99 (4.06)
Average watch time (standard deviation)	7.88 seconds (5.58)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	58 players, 85.29 percent
Average confidence in "good"-judgement (standard deviation)	7.67 (3.77)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	10 players, 14.71 percent
Average confidence in "bad" judgement (standard deviation)	3.00 (3.50)

Judging Model *pq_list_a3_3*

Associated program:
`a > age`

Description: Shows an input-output model for the greater-than operator and a unappropriate container model for the compared variables (name and content mixed-up).

Time: 9 seconds
Concreteness: 100 percent of names are used in the model



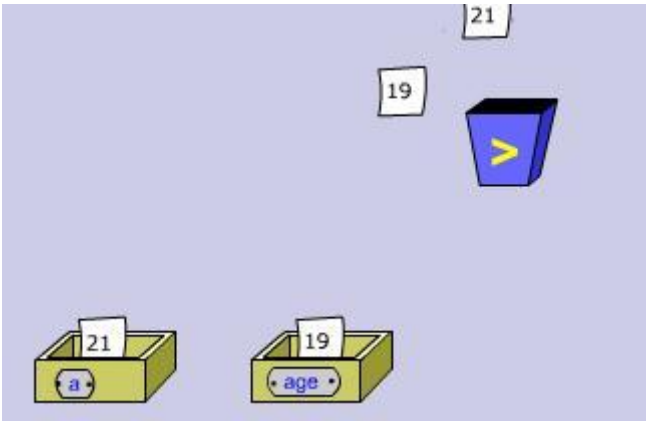
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.69 (4.29)
Average watch time (standard deviation)	27.18 seconds (95.74)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	43 players, 63.24 percent
Average confidence in "good"-judgement (standard deviation)	6.86 (4.37)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	25 players, 36.76 percent
Average confidence in "bad" judgement (standard deviation)	6.40 (4.21)

Judging Model *pq_list_a3_1*

Associated program:
`a > age`

Description: Visualization of the greater-than operation using an input-output model for the operator and a cointainer model for the compared variables.

Time: 9 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.69 (4.29)
Average watch time (standard deviation)	14.84 seconds (25.25)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	56 players, 82.35 percent
Average confidence in "good"-judgement (standard deviation)	7.23 (4.04)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	12 players, 17.65 percent
Average confidence in "bad" judgement (standard deviation)	4.17 (4.69)

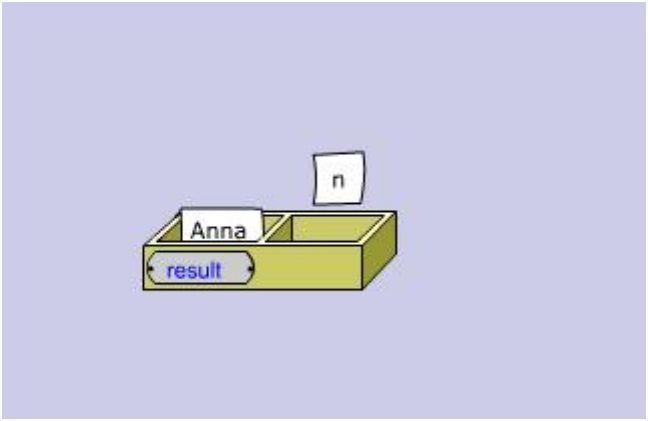
Task 4

Judging Model *pq_list_a4_5*

Associated program:
`result.append(n)`

Description: Unappropiate visualization of a list extension (append) that mixes-up name and value of variables. The list is represented by a container model.

Time: 5 seconds
Concreteness: 67 percent of names are used in the model



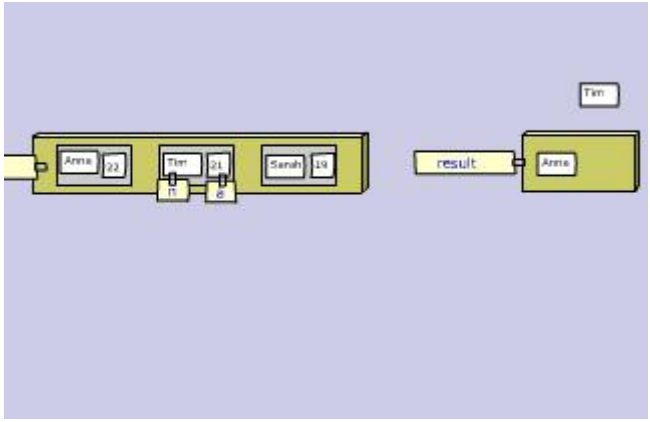
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.18 (4.24)
Average watch time (standard deviation)	9.24 seconds (6.09)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	43 players, 63.24 percent
Average confidence in "good"-judgement (standard deviation)	5.81 (4.22)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	25 players, 36.76 percent
Average confidence in "bad" judgement (standard deviation)	6.80 (4.30)

Judging Model *pq_list_a4_4*

Associated program:
`result.append(n)`

Description: Visualization of a list extension (append). Two lists are represented by boards with cards. A card moves from the 1st to the 2nd list which gets longer.

Time: 5 seconds
Concreteness: 67 percent of names are used in the model



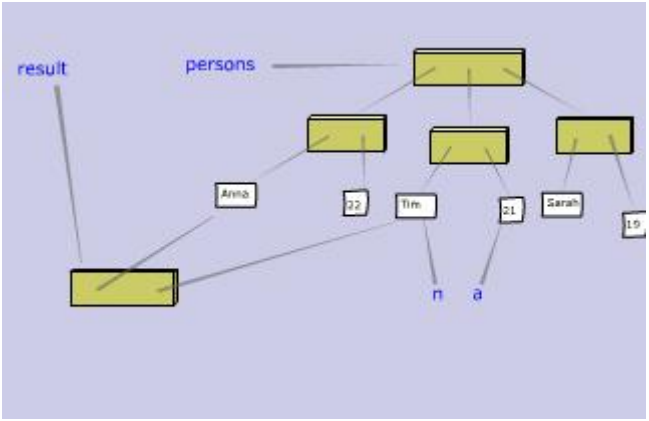
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.69 (4.02)
Average watch time (standard deviation)	11.44 seconds (12.78)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	54 players, 79.41 percent
Average confidence in "good"-judgement (standard deviation)	6.85 (4.04)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	14 players, 20.59 percent
Average confidence in "bad" judgement (standard deviation)	6.07 (4.01)

Judging Model *pq_list_a4_6*

Associated program:
`result.append(n)`

Description: Visualization of a list extension (append) using a consistent pointer model for the lists. Lists are represented by boards with pointers.

Time: 5 seconds
Concreteness: 67 percent of names are used in the model



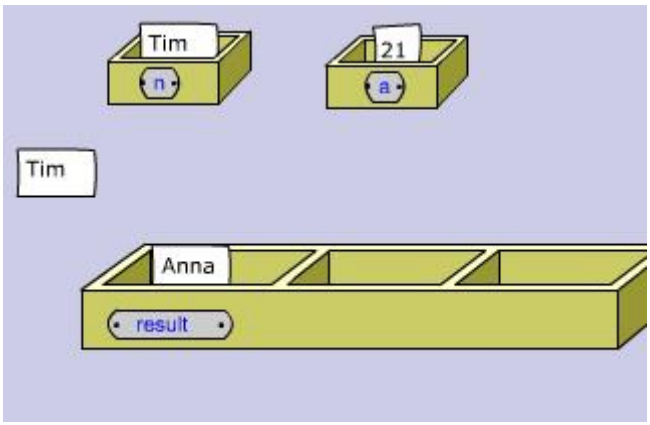
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	4.85 (4.57)
Average watch time (standard deviation)	9.62 seconds (7.51)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	40 players, 58.82 percent
Average confidence in "good"-judgement (standard deviation)	5.62 (4.56)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	28 players, 41.18 percent
Average confidence in "bad" judgement (standard deviation)	3.75 (4.44)

Judging Model *pq_list_a4_1*

Associated program:
`result.append(n)`

Description: Visualization of a list extension (call of append()-method). The list is represented by a box with a constant number of compartments.

Time: 5 seconds
Concreteness: 67 percent of names are used in the model



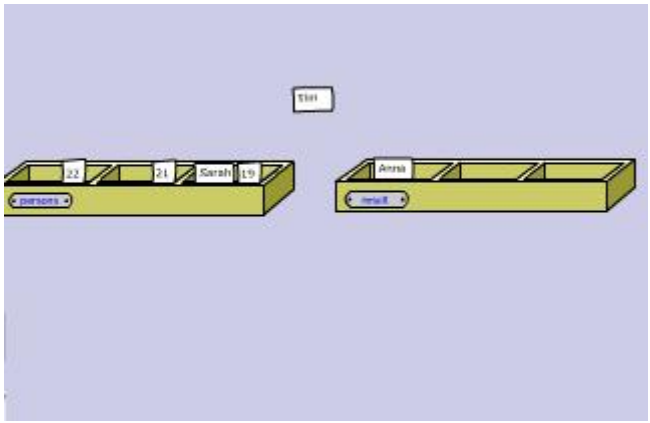
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.03 (4.28)
Average watch time (standard deviation)	9.66 seconds (7.28)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	48 players, 70.59 percent
Average confidence in "good"-judgement (standard deviation)	6.67 (4.17)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	20 players, 29.41 percent
Average confidence in "bad" judgement (standard deviation)	4.50 (4.26)

Judging Model *pq_list_a4_3*

Associated program:
`result.append(n)`

Description: Visualization of a list extension (call of append()-method). There are two list models, a card moves from one list to the other.

Time: 4 seconds
Concreteness: 33 percent of names are used in the model



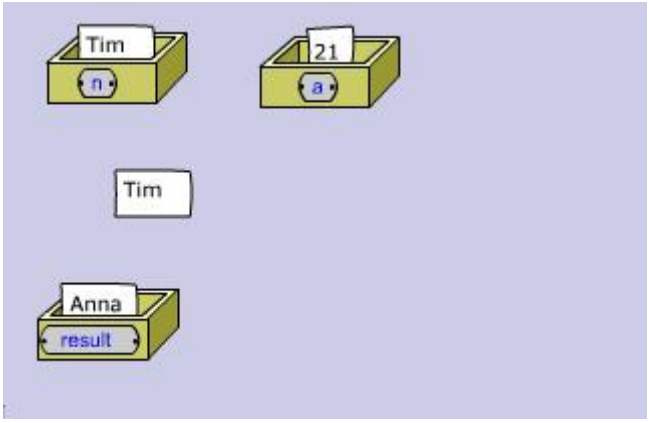
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.47 (3.96)
Average watch time (standard deviation)	10.01 seconds (10.24)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	39 players, 57.35 percent
Average confidence in "good"-judgement (standard deviation)	6.54 (4.16)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	29 players, 42.65 percent
Average confidence in "bad" judgement (standard deviation)	6.38 (3.76)

Judging Model *pq_list_a4_2*

Associated program:
`result.append(n)`

Description: Visualization of a list extension (call of append()-method). The list is represented by a box with increasing number of compartments.

Time: 5 seconds
Concreteness: 67 percent of names are used in the model



Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	5.59 (4.36)
Average watch time (standard deviation)	7.94 seconds (5.71)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	51 players, 75.00 percent
Average confidence in "good"-judgement (standard deviation)	5.78 (4.40)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	17 players, 25.00 percent
Average confidence in "bad" judgement (standard deviation)	5.00 (4.33)

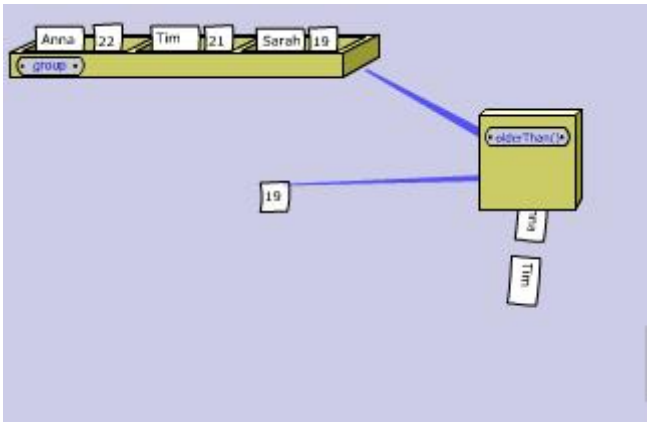
Task 5

Judging Model *pq_list_a5_4*

Associated program:
`olderThan(group, 19)`

Description: Visualization of a function call. The function is represented by a box with sensors.

Time: 9 seconds
Concreteness: 100 percent of names are used in the model



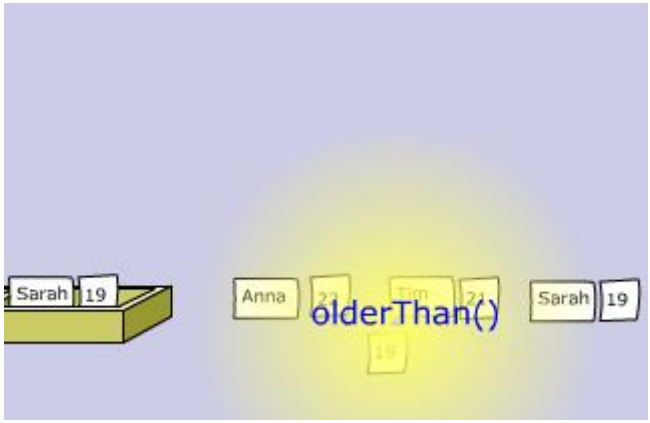
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.69 (3.92)
Average watch time (standard deviation)	7.75 seconds (6.09)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	48 players, 70.59 percent
Average confidence in "good"-judgement (standard deviation)	7.19 (3.85)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	20 players, 29.41 percent
Average confidence in "bad" judgement (standard deviation)	5.50 (3.94)

Judging Model *pq_list_a5_5*

Associated program:
`olderThan(group, 19)`

Description: Visualization of a function call. The function call is represented by a flash (event model).

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



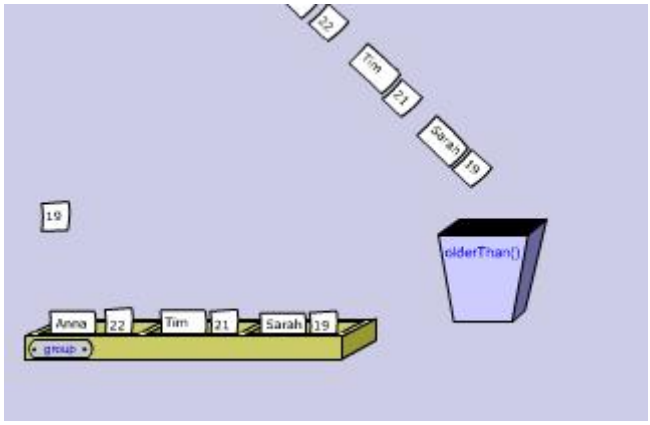
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.76 (4.03)
Average watch time (standard deviation)	8.09 seconds (6.32)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	49 players, 72.06 percent
Average confidence in "good"-judgement (standard deviation)	6.73 (4.02)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	19 players, 27.94 percent
Average confidence in "bad" judgement (standard deviation)	6.84 (4.15)

Judging Model *pq_list_a5_6*

Associated program:
`olderThan(group, 19)`

Description: Visualization of a function call. The function (input-output model) receives copies of all cards of a list (container) and a card with a number.

Time: 9 seconds
Concreteness: 100 percent of names are used in the model



First Judgements

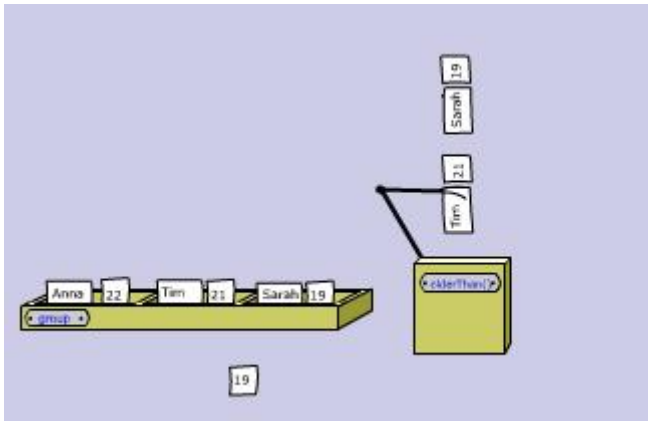
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.91 (4.05)
Average watch time (standard deviation)	10.37 seconds (9.02)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	46 players, 67.65 percent
Average confidence in "good"-judgement (standard deviation)	7.28 (4.04)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	22 players, 32.35 percent
Average confidence in "bad" judgement (standard deviation)	6.14 (4.06)

Judging Model *pq_list_a5_1*

Associated program:
`olderThan(group, 19)`

Description: Visualization of a function call. The Function is represented by a box with a manipulator arm, that graps input values.

Time: 14 seconds
Concreteness: 100 percent of names are used in the model



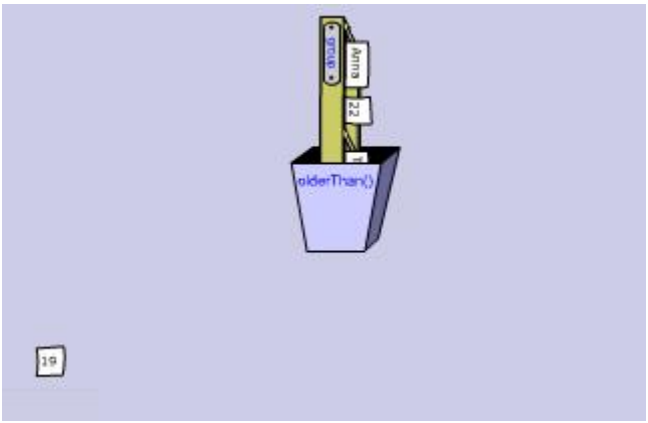
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.13 (3.99)
Average watch time (standard deviation)	12.75 seconds (12.56)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	55 players, 80.88 percent
Average confidence in "good"-judgement (standard deviation)	7.55 (3.71)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	13 players, 19.12 percent
Average confidence in "bad" judgement (standard deviation)	5.38 (4.77)

Judging Model *pq_list_a5_2*

Associated program:
`olderThan(group, 19)`

Description: Visualization of a function call. The function (input-output model) accepts a list model(container) and a card with a number.

Time: 9 seconds
Concreteness: 100 percent of names are used in the model



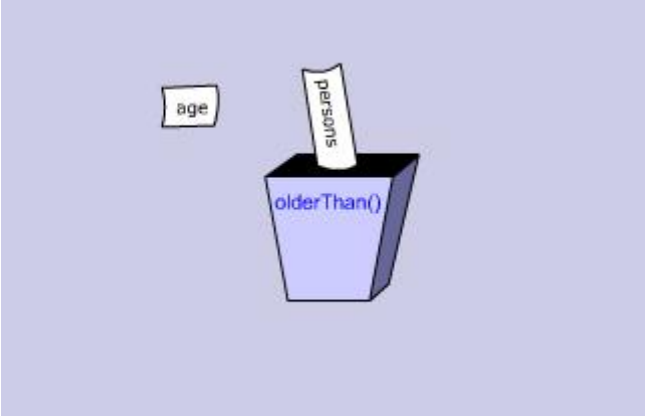
Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.62 (3.91)
Average watch time (standard deviation)	10.59 seconds (19.34)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	40 players, 58.82 percent
Average confidence in "good"-judgement (standard deviation)	6.25 (3.88)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	28 players, 41.18 percent
Average confidence in "bad" judgement (standard deviation)	7.14 (3.95)

Judging Model *pq_list_a5_3*

Associated program:
`olderThan(group, 19)`

Description: Unappropriate visualization of a function call. The function (input-output model) receives cards with names of formal parameters as input.

Time: 9 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	68
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.40 (4.13)
Average watch time (standard deviation)	27.81 seconds (86.31)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	35 players, 51.47 percent
Average confidence in "good"-judgement (standard deviation)	6.14 (4.22)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	33 players, 48.53 percent
Average confidence in "bad" judgement (standard deviation)	6.67 (4.08)

Modeling assignments.

This game contains 2 tasks. I n each task the player has to judge several models, whether or not they are appropriate to explain the execution of some program statement.

Task 1	`a = 3` `b = a`

Task 2	`today = "Monday"`
	`today = "Tuesday"`
	`today = "Wednesday"`

The following table provides some general information about the usage of this game (pq_assign).

Number of objective sessions (*not* taking into account that there are sometimes two players working together)	332 sessions
Number of subjective sessions (taking into account that there are sometimes two players working together)	388 sessions
Number of subjective uncanceled sessions (taking into account that there are sometimes two players working together)	313 sessions
Number of subjective *first* uncanceled sessions (taking into account that there are sometimes two players working together and some people play this game several times)	154 sessions
Number of persons, who played three times or more	41 persons

Some general information about the players of this category, who played the game at least *once*.

Professions	154 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	34 female and 120 male persons
Hours a week spent on programming (standard deviation)	3.23 hours (3.23)
Roughly estimated experience in Python programming(standard deviation)	94.95 days (130.39)
Age (standard deviation)	17.21 years (2.04)
Population of the town, where the workshop took place	Less than 100 000: 42, 100 000 to 500 000: 28, more than 500 000: 84
Country	Germany: 110 other country: 44

Some general information about the players of this category, who played the game at least *three times*.

Professions	41 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	13 female and 28 male persons
Hours a week spent on programming (standard deviation)	3.44 hours (2.27)
Roughly estimated experience in Python programming(standard deviation)	189.37 days (127.68)
Age (standard deviation)	16.80 years (1.23)
Population of the town, where the workshop took place	Less than 100 000: 4, 100 000 to 500 000: 13, more than 500 000: 24
Country	Germany: 39 other country: 2

Task 1

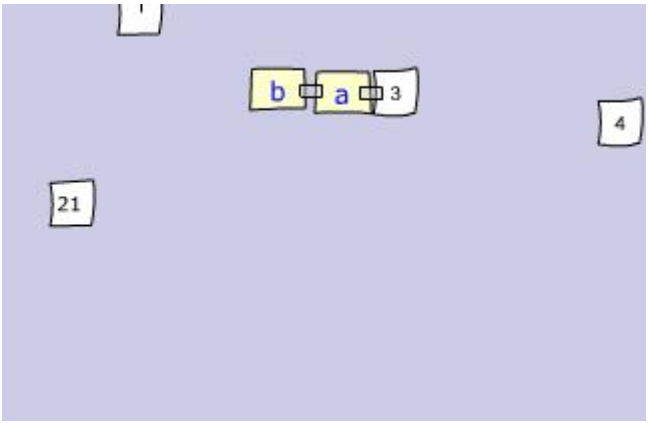
Judging Model *pq_assign_a1_9*

Associated program:

```
a = 3
b = a
```

Description: Names are represented by post-its. A post-it a is attached to a floating card with number 3. A second post-it b is attached to post-it a.

Time: 9 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.82 (3.07)
Average watch time (standard deviation)	13.75 seconds (11.58)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	78 players, 50.65 percent
Average confidence in "good"-judgement (standard deviation)	7.37 (3.09)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	76 players, 49.35 percent
Average confidence in "bad" judgement (standard deviation)	8.29 (3.01)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.56 (3.38)	9.39 (2.00)	8.90 (3.06)
Average watch time (standard deviation)	8.98 seconds (4.61)	7.20 seconds (4.59)	4.85 seconds (3.18)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	34.15 percent	17.07 percent	9.76 percent
Average confidence in "good"-judgement (standard deviation)	6.43 (3.06)	9.29 (1.89)	10.00 (0.00)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	65.85 percent	82.93 percent	90.24 percent
Average confidence in "bad" judgement (standard deviation)	8.15 (3.44)	9.41 (2.05)	8.78 (3.21)

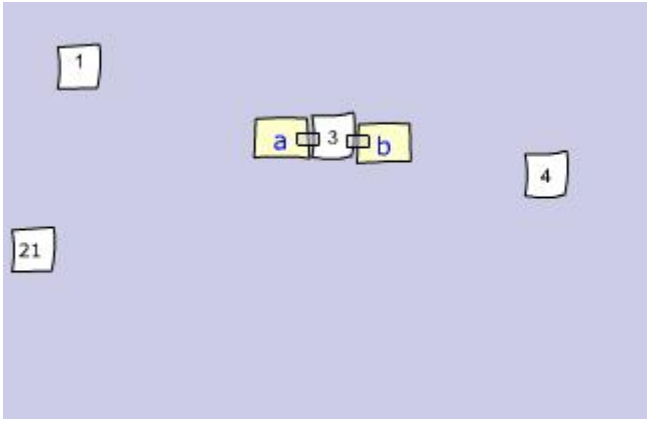
Judging Model *pq_assign_a1_8*

Associated program:

```
a = 3
b = a
```

Description: Names are represented by post-its. A post-it a is attached to a floating card with number 3. A second post-it b is attached to the same card.

Time: 9 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.21 (3.11)
Average watch time (standard deviation)	14.05 seconds (12.56)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	119 players, 77.27 percent
Average confidence in "good"-judgement (standard deviation)	8.49 (2.88)
Number and percentage of players, who declared that the model is *not* appropriate	35 players, 22.73 percent
Average confidence in "bad" judgement (standard deviation)	7.29 (3.71)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.05 (3.14)	8.90 (3.06)	9.15 (2.72)
Average watch time (standard deviation)	14.95 seconds (18.50)	7.22 seconds (3.93)	5.88 seconds (2.32)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	82.93 percent	90.24 percent	78.05 percent
Average confidence in "good"-judgement (standard deviation)	8.53 (2.62)	9.32 (2.40)	9.38 (2.46)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	17.07 percent	9.76 percent	21.95 percent
Average confidence in "bad" judgement (standard deviation)	5.71 (4.50)	5.00 (5.77)	8.33 (3.54)

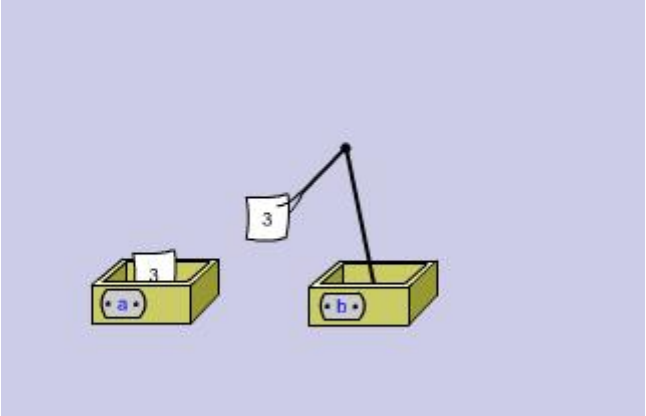
Judging Model *pq_assign_a1_7*

Associated program:

```
a = 3
b = a
```

Description: Variables are represented by containers containing cards. A manipulator arm of container b takes copy of the card in container a.

Time: 9 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.64 (2.44)
Average watch time (standard deviation)	12.68 seconds (11.56)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	125 players, 81.17 percent
Average confidence in "good"-judgement (standard deviation)	8.76 (2.26)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	29 players, 18.83 percent
Average confidence in "bad" judgement (standard deviation)	8.10 (3.11)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.05 (3.14)	8.66 (2.74)	8.66 (3.17)
Average watch time (standard deviation)	12.95 seconds (12.34)	7.71 seconds (5.28)	6.88 seconds (2.56)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	80.49 percent	92.68 percent	85.37 percent
Average confidence in "good"-judgement (standard deviation)	8.33 (2.70)	8.95 (2.37)	8.86 (2.99)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	19.51 percent	7.32 percent	14.63 percent
Average confidence in "bad" judgement (standard deviation)	6.88 (4.58)	5.00 (5.00)	7.50 (4.18)

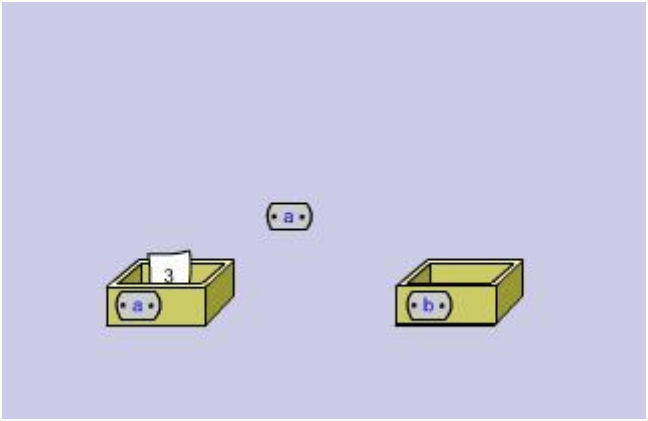
Judging Model *pq_assign_a1_6*

Associated program:

```
a = 3
b = a
```

Description: Variables are represented by containers containing cards. A copy of the name sign a goes into container b

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.44 (2.94)
Average watch time (standard deviation)	10.50 seconds (7.75)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	98 players, 63.64 percent
Average confidence in "good"-judgement (standard deviation)	8.72 (2.41)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	56 players, 36.36 percent
Average confidence in "bad" judgement (standard deviation)	7.95 (3.67)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.17 (3.31)	8.29 (3.28)	8.41 (3.25)
Average watch time (standard deviation)	10.59 seconds (7.03)	7.20 seconds (3.63)	6.37 seconds (2.75)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	51.22 percent	41.46 percent	24.39 percent
Average confidence in "good"-judgement (standard deviation)	8.81 (2.18)	7.94 (3.56)	9.00 (2.11)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	48.78 percent	58.54 percent	75.61 percent
Average confidence in "bad" judgement (standard deviation)	7.50 (4.14)	8.54 (3.12)	8.23 (3.55)

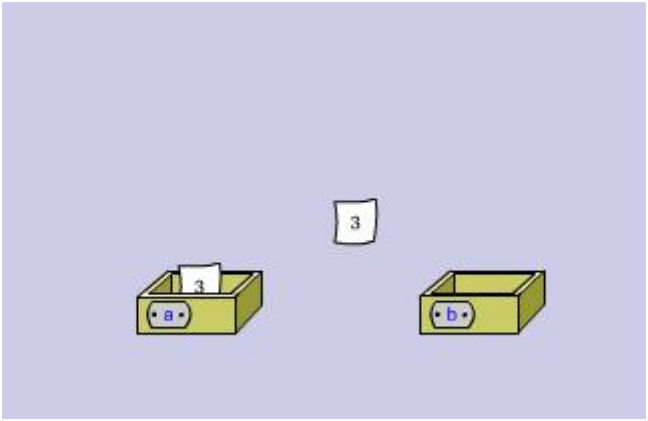
Judging Model *pq_assign_a1_5*

Associated program:

```
a = 3
b = a
```

Description: Variables are represented by containers containing cards. A copy of the card in a migrates to b.

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.77 (2.76)
Average watch time (standard deviation)	9.06 seconds (7.39)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	133 players, 86.36 percent
Average confidence in "good"-judgement (standard deviation)	9.02 (2.34)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	21 players, 13.64 percent
Average confidence in "bad" judgement (standard deviation)	7.14 (4.35)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.80 (3.88)	8.90 (2.62)	8.90 (3.06)
Average watch time (standard deviation)	7.71 seconds (6.10)	5.59 seconds (1.99)	5.05 seconds (2.10)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	80.49 percent	92.68 percent	95.12 percent
Average confidence in "good"-judgement (standard deviation)	8.64 (3.13)	9.21 (2.18)	9.36 (2.35)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	19.51 percent	7.32 percent	4.88 percent
Average confidence in "bad" judgement (standard deviation)	4.38 (4.96)	5.00 (5.00)	0.00 (0.00)

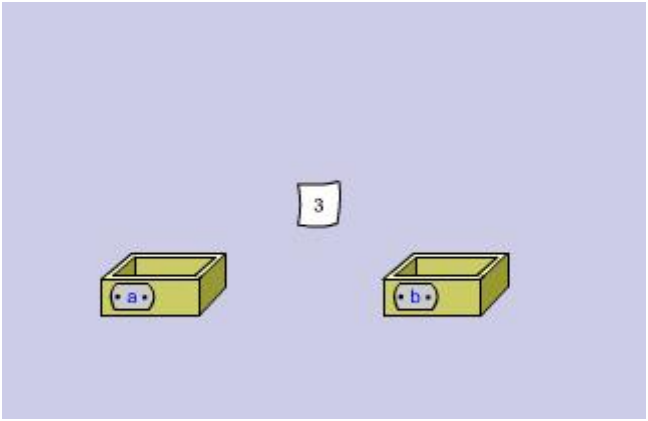
Judging Model *pq_assign_a1_4*

Associated program:

```
a = 3
b = a
```

Description: Variables are represented by containers containing cards. A card migrates from a to b.

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.86 (3.56)
Average watch time (standard deviation)	12.02 seconds (8.19)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	74 players, 48.05 percent
Average confidence in "good"-judgement (standard deviation)	8.18 (3.37)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	80 players, 51.95 percent
Average confidence in "bad" judgement (standard deviation)	7.56 (3.73)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.83 (4.29)	8.41 (3.25)	8.05 (3.69)
Average watch time (standard deviation)	11.80 seconds (7.66)	7.66 seconds (3.40)	7.12 seconds (3.78)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	43.90 percent	17.07 percent	12.20 percent
Average confidence in "good"-judgement (standard deviation)	7.50 (4.29)	6.43 (4.76)	5.00 (3.54)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	56.10 percent	82.93 percent	87.80 percent
Average confidence in "bad" judgement (standard deviation)	6.30 (4.32)	8.82 (2.77)	8.47 (3.55)

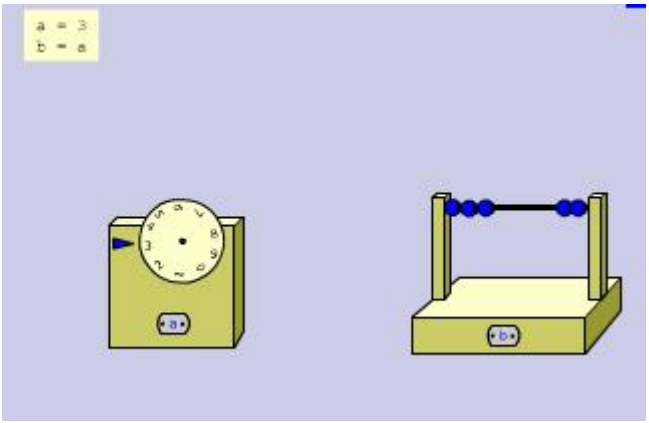
Judging Model *pq_assign_a1_3*

Associated program:

```
a = 3
b = a
```

Description: The value of a variable is represented by a rotating disk and secondly by an abacus. Assignment is visualized by adjusting the disk resp. abacus.

Time: 14 seconds
Concreteness: 100 percent of names are used in the model



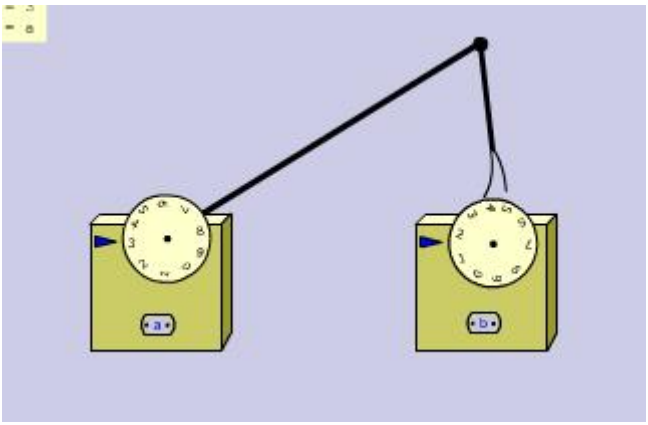
Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.15 (3.18)
Average watch time (standard deviation)	15.92 seconds (11.04)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	103 players, 66.88 percent
Average confidence in "good"-judgement (standard deviation)	8.83 (2.54)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	51 players, 33.12 percent
Average confidence in "bad" judgement (standard deviation)	6.76 (3.85)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.05 (3.51)	8.78 (3.12)	8.90 (3.06)
Average watch time (standard deviation)	16.37 seconds (12.91)	8.88 seconds (4.24)	6.90 seconds (3.49)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	70.73 percent	82.93 percent	80.49 percent
Average confidence in "good"-judgement (standard deviation)	8.62 (2.96)	9.12 (2.60)	9.85 (0.87)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	29.27 percent	17.07 percent	19.51 percent
Average confidence in "bad" judgement (standard deviation)	6.67 (4.44)	7.14 (4.88)	5.00 (5.35)

Judging Model *pq_assign_a1_2*

Associated program:

```
a = 3
b = a
```



Description: The value of a variable is represented by a rotating disk. Assignment is visualized by adjusting the disk. Variable b adjusts the disk of a.

Time: 14 seconds
Concreteness: 100 percent of names are used in the model

Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.38 (2.96)
Average watch time (standard deviation)	15.82 seconds (10.79)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	108 players, 70.13 percent
Average confidence in "good"-judgement (standard deviation)	8.80 (2.55)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	46 players, 29.87 percent
Average confidence in "bad" judgement (standard deviation)	7.39 (3.61)

First three judgements

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.32 (3.72)	8.54 (2.79)	8.90 (2.85)
Average watch time (standard deviation)	16.10 seconds (9.13)	22.59 seconds (62.44)	16.51 seconds (47.26)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	60.98 percent	70.73 percent	70.73 percent
Average confidence in "good"-judgement (standard deviation)	7.60 (3.57)	8.28 (3.07)	9.66 (1.86)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	39.02 percent	29.27 percent	29.27 percent
Average confidence in "bad" judgement (standard deviation)	6.88 (4.03)	9.17 (1.95)	7.08 (3.96)

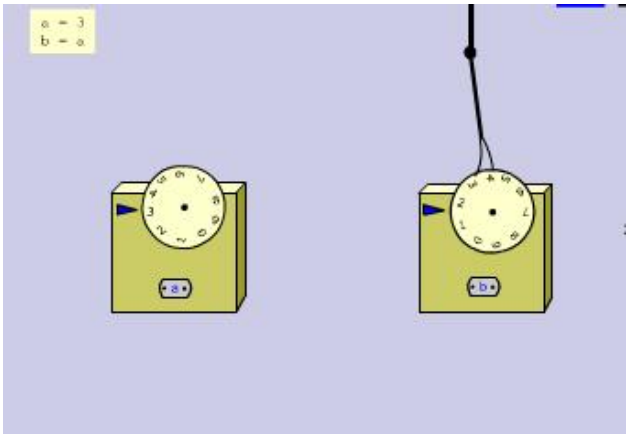
Judging Model *pq_assign_a1_1*

Associated program:

```
a = 3
b = a
```

Description: The value of a variable is represented by a rotating disk. Assignment is visualized by adjusting the disk.

Time: 14 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.82 (3.28)
Average watch time (standard deviation)	19.93 seconds (10.96)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	100 players, 64.94 percent
Average confidence in "good"-judgement (standard deviation)	8.30 (3.03)
Number and percentage of players, who declared that the model is not appropriate ("bad"-judgements)	54 players, 35.06 percent
Average confidence in "bad" judgement (standard deviation)	6.94 (3.56)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.83 (4.15)	9.15 (2.21)	9.15 (2.72)
Average watch time (standard deviation)	19.71 seconds (8.63)	12.41 seconds (6.88)	9.10 seconds (5.94)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	68.29 percent	92.68 percent	85.37 percent
Average confidence in "good"-judgement (standard deviation)	7.50 (3.97)	9.21 (2.18)	9.86 (0.85)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	31.71 percent	7.32 percent	14.63 percent
Average confidence in "bad" judgement (standard deviation)	5.38 (4.31)	8.33 (2.89)	5.00 (5.48)

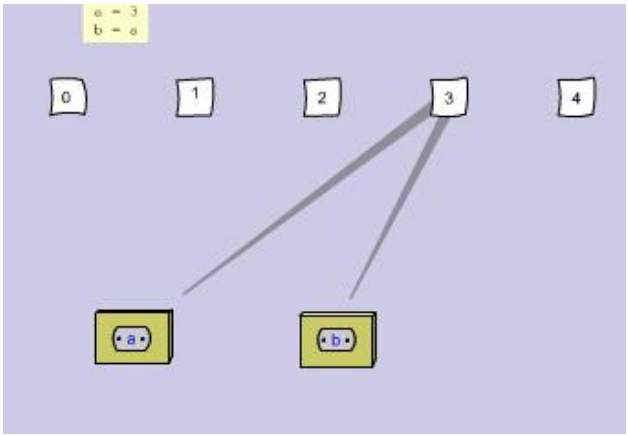
Judging Model *pq_assign_a1_13*

Associated program:

```
a = 3
b = a
```

Description: Shows a reversed pointer model for variables. Two pointers point from a card with number 3 to two names of the object 3 (a and b)

Time: 5 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.60 (3.58)
Average watch time (standard deviation)	13.35 seconds (13.77)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	70 players, 45.45 percent
Average confidence in "good"-judgement (standard deviation)	7.21 (3.77)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	84 players, 54.55 percent
Average confidence in "bad" judgement (standard deviation)	7.92 (3.40)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.34 (4.33)	8.05 (3.69)	8.41 (3.61)
Average watch time (standard deviation)	18.10 seconds (18.67)	6.56 seconds (3.29)	5.02 seconds (4.85)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	56.10 percent	53.66 percent	29.27 percent
Average confidence in "good"-judgement (standard deviation)	5.65 (4.60)	8.41 (3.58)	9.17 (2.89)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	43.90 percent	46.34 percent	70.73 percent
Average confidence in "bad" judgement (standard deviation)	7.22 (3.92)	7.63 (3.86)	8.10 (3.88)

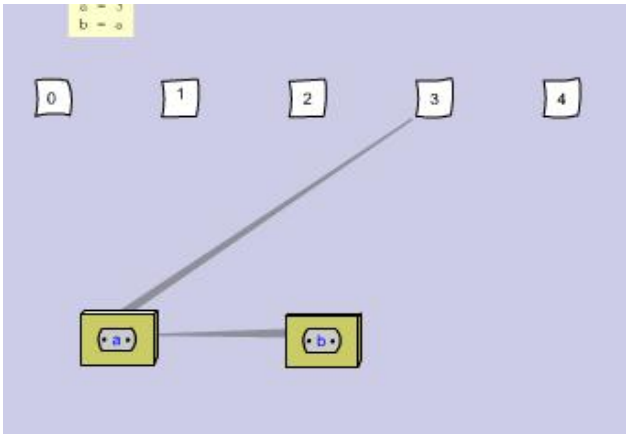
Judging Model *pq_assign_a1_12*

Associated program:

```
a = 3
b = a
```

Description: Variable a containing the value 3 is visualized by a pointer named a pointing to number 3. A second pointer b points to the origin of pointer a.

Time: 5 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.86 (2.46)
Average watch time (standard deviation)	12.07 seconds (13.72)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	87 players, 56.49 percent
Average confidence in "good"-judgement (standard deviation)	8.39 (2.80)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	67 players, 43.51 percent
Average confidence in "bad" judgement (standard deviation)	9.48 (1.77)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.90 (2.62)	8.66 (3.17)	8.54 (3.40)
Average watch time (standard deviation)	13.90 seconds (19.10)	5.00 seconds (2.58)	4.76 seconds (2.17)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	53.66 percent	14.63 percent	12.20 percent
Average confidence in "good"-judgement (standard deviation)	8.18 (3.29)	6.67 (4.08)	9.00 (2.24)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	46.34 percent	85.37 percent	87.80 percent
Average confidence in "bad" judgement (standard deviation)	9.74 (1.15)	9.00 (2.92)	8.47 (3.55)

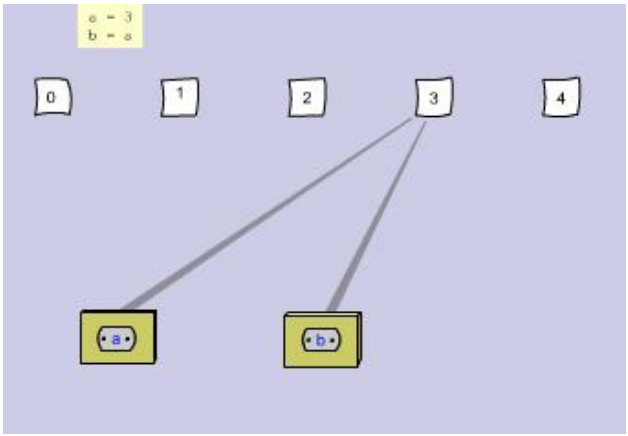
Judging Model *pq_assign_a1_11*

Associated program:

```
a = 3
b = a
```

Description: Variable a containing the value 3 is visualized by a pointer named a pointing to a card with number 3. A second pointer b points to the same card.

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.31 (2.98)
Average watch time (standard deviation)	13.36 seconds (11.38)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	119 players, 77.27 percent
Average confidence in "good"-judgement (standard deviation)	8.82 (2.50)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	35 players, 22.73 percent
Average confidence in "bad" judgement (standard deviation)	6.57 (3.79)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.80 (3.54)	9.27 (2.11)	8.41 (3.61)
Average watch time (standard deviation)	13.00 seconds (11.38)	6.17 seconds (2.61)	5.12 seconds (3.09)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	78.05 percent	87.80 percent	85.37 percent
Average confidence in "good"-judgement (standard deviation)	8.75 (2.54)	9.31 (2.12)	9.29 (2.47)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	21.95 percent	12.20 percent	14.63 percent
Average confidence in "bad" judgement (standard deviation)	4.44 (4.64)	9.00 (2.24)	3.33 (5.16)

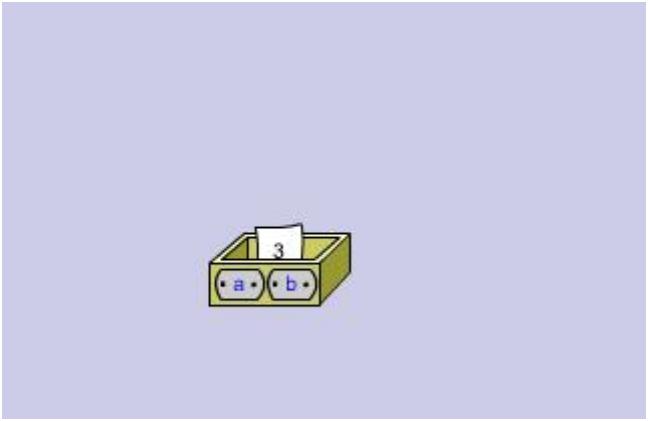
Judging Model *pq_assign_a1_10*

Associated program:

```
a = 3
b = a
```

Description: Variable a is visualized by a box with a name sign a containing a card with number 3. A second name sign b is attached to this box.

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.76 (3.13)
Average watch time (standard deviation)	17.31 seconds (18.40)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	103 players, 66.88 percent
Average confidence in "good"-judgement (standard deviation)	7.96 (3.09)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	51 players, 33.12 percent
Average confidence in "bad" judgement (standard deviation)	7.35 (3.22)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.29 (3.08)	8.90 (2.10)	8.78 (3.12)
Average watch time (standard deviation)	19.56 seconds (25.27)	8.07 seconds (3.45)	7.12 seconds (5.38)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	60.98 percent	68.29 percent	60.98 percent
Average confidence in "good"-judgement (standard deviation)	8.40 (2.78)	8.93 (2.09)	9.60 (1.38)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	39.02 percent	31.71 percent	39.02 percent
Average confidence in "bad" judgement (standard deviation)	8.12 (3.59)	8.85 (2.19)	7.50 (4.47)

Task 2

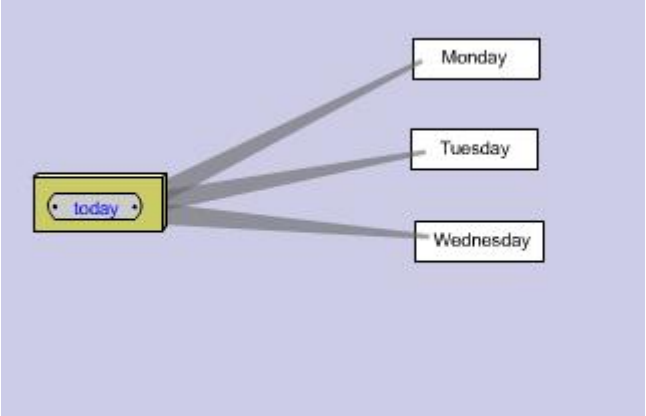
Judging Model *pq_assign_a2_8*

Associated program:

```
today = "Monday"
today = "Tuesday"
today = "Wednesday"
```

Description: A Variable is represented by a pointer. Successive assignments are visualized by accumulated pointers (fan) pointing to several data objects

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.47 (3.10)
Average watch time (standard deviation)	11.38 seconds (24.53)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	59 players, 38.31 percent
Average confidence in "good"-judgement (standard deviation)	8.81 (2.84)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	95 players, 61.69 percent
Average confidence in "bad" judgement (standard deviation)	8.26 (3.24)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.56 (4.05)	8.88 (2.65)	8.54 (3.40)
Average watch time (standard deviation)	9.63 seconds (5.78)	5.05 seconds (2.58)	3.73 seconds (1.99)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	43.90 percent	19.51 percent	7.32 percent
Average confidence in "good"-judgement (standard deviation)	7.78 (4.28)	7.50 (3.78)	6.67 (5.77)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	56.10 percent	80.49 percent	92.68 percent
Average confidence in "bad" judgement (standard deviation)	7.39 (3.95)	9.22 (2.24)	8.68 (3.22)

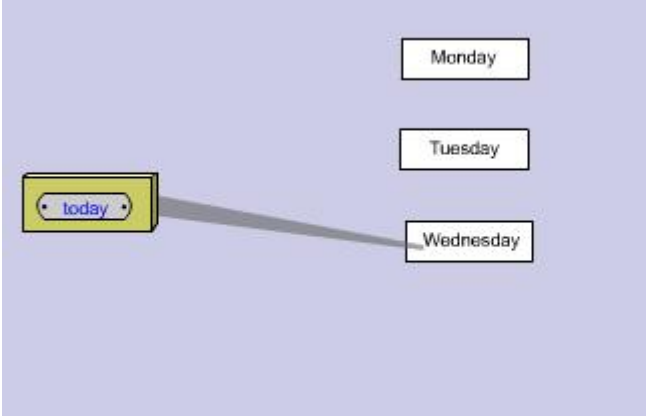
Judging Model *pq_assign_a2_9*

Associated program:

```
today = "Monday"
today = "Tuesday"
today = "Wednesday"
```

Description: A Variable is represented by a pointer. Successive assignments are visualized by one pointers pointing to appearing cards

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	9.09 (2.32)
Average watch time (standard deviation)	7.77 seconds (5.88)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	125 players, 81.17 percent
Average confidence in "good"-judgement (standard deviation)	9.36 (1.79)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	29 players, 18.83 percent
Average confidence in "bad" judgement (standard deviation)	7.93 (3.66)

First three judgements

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.29 (3.47)	9.00 (2.82)	8.66 (3.36)
Average watch time (standard deviation)	7.44 seconds (4.98)	5.80 seconds (3.87)	4.44 seconds (5.62)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	75.61 percent	90.24 percent	80.49 percent
Average confidence in "good"-judgement (standard deviation)	9.03 (2.39)	9.19 (2.50)	9.55 (1.92)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	24.39 percent	9.76 percent	19.51 percent
Average confidence in "bad" judgement (standard deviation)	6.00 (5.16)	6.67 (5.77)	5.00 (5.35)

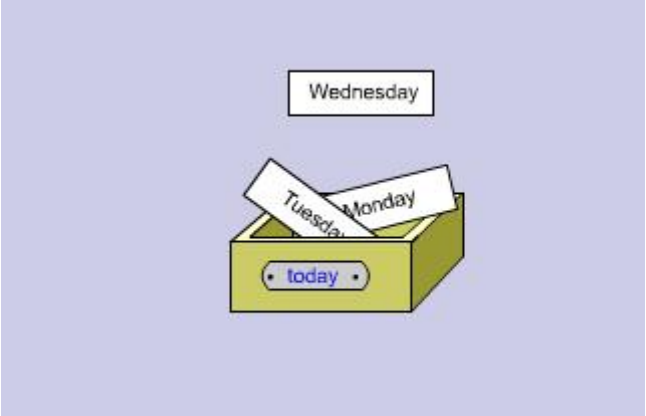
Judging Model *pq_assign_a2_1*

Associated program:

```
today = "Monday"
today = "Tuesday"
today = "Wednesday"
```

Description: A Variable is represented by a container. To visualize 3 successive assignments 3 cards move into the container without destruction of former content.

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



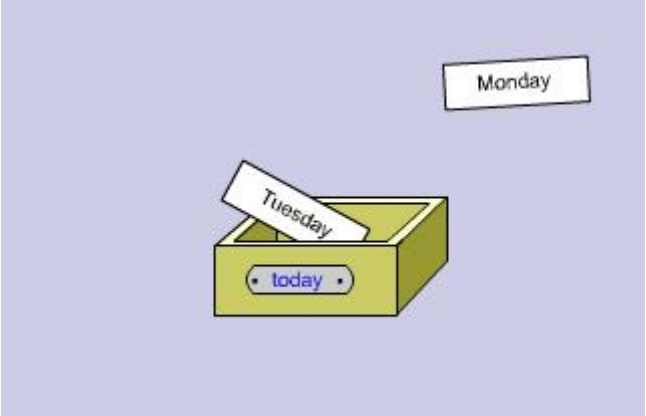
Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.86 (2.71)
Average watch time (standard deviation)	7.92 seconds (4.65)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	54 players, 35.06 percent
Average confidence in "good"-judgement (standard deviation)	8.70 (2.94)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	100 players, 64.94 percent
Average confidence in "bad" judgement (standard deviation)	8.95 (2.59)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.05 (3.51)	8.25 (3.50)	8.41 (3.61)
Average watch time (standard deviation)	8.15 seconds (5.14)	6.03 seconds (3.77)	4.07 seconds (2.07)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	36.59 percent	14.63 percent	7.32 percent
Average confidence in "good"-judgement (standard deviation)	8.67 (2.97)	4.17 (4.92)	6.67 (5.77)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	63.41 percent	85.37 percent	92.68 percent
Average confidence in "bad" judgement (standard deviation)	7.69 (3.80)	8.97 (2.69)	8.55 (3.47)

Judging Model *pq_assign_a2_2*

Associated program:

```
today = "Monday"
today = "Tuesday"
today = "Wednesday"
```



Description: A Variable is represented by a container. To visualize successive assignments cards move into the container after former content has moved out.

Time: 7 seconds
Concreteness: 100 percent of names are used in the model

Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.86 (2.83)
Average watch time (standard deviation)	9.45 seconds (12.45)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	120 players, 77.92 percent
Average confidence in "good"-judgement (standard deviation)	9.12 (2.40)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	34 players, 22.08 percent
Average confidence in "bad" judgement (standard deviation)	7.94 (3.92)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.29 (3.64)	9.25 (2.42)	8.66 (3.36)
Average watch time (standard deviation)	8.27 seconds (6.03)	5.53 seconds (3.26)	3.98 seconds (1.92)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	78.05 percent	90.24 percent	85.37 percent
Average confidence in "good"-judgement (standard deviation)	9.06 (2.68)	9.46 (1.97)	9.57 (1.87)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	21.95 percent	9.76 percent	14.63 percent
Average confidence in "bad" judgement (standard deviation)	5.56 (5.27)	6.67 (5.77)	3.33 (5.16)

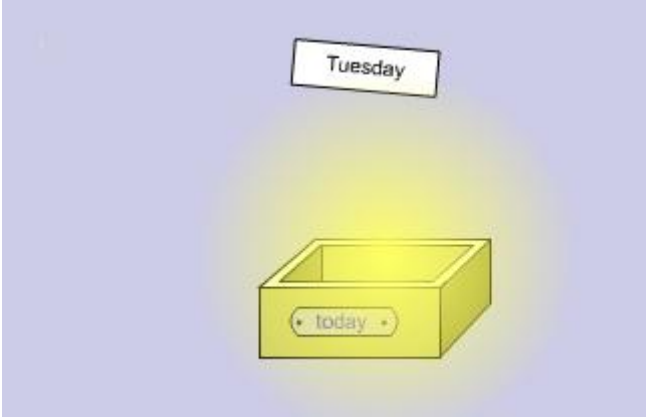
Judging Model *pq_assign_a2_3*

Associated program:

```
today = "Monday"
today = "Tuesday"
today = "Wednesday"
```

Description: A Variable is represented by a container. To visualize successive assignments cards move into it after former content has been destroyed.

Time: 7 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.44 (3.00)
Average watch time (standard deviation)	10.14 seconds (11.01)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	117 players, 75.97 percent
Average confidence in "good"-judgement (standard deviation)	8.93 (2.53)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	37 players, 24.03 percent
Average confidence in "bad" judgement (standard deviation)	6.89 (3.79)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.68 (3.55)	8.62 (2.99)	8.29 (3.64)
Average watch time (standard deviation)	10.00 seconds (10.61)	5.92 seconds (3.96)	4.10 seconds (2.34)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	65.85 percent	82.93 percent	75.61 percent
Average confidence in "good"-judgement (standard deviation)	9.26 (1.81)	8.68 (3.09)	9.19 (2.61)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	34.15 percent	17.07 percent	24.39 percent
Average confidence in "bad" judgement (standard deviation)	4.64 (4.14)	8.33 (2.58)	5.50 (4.97)

Judging Model *pq_assign_a2_4*

Associated program:

```
today = "Monday"
today = "Tuesday"
today = "Wednesday"
```

Description: A Variable is represented by a sheet with title (representing the name). Assignments are visualized through writing words on it.

Time: 8 seconds
Concreteness: 100 percent of names are used in the model

Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.70 (2.96)
Average watch time (standard deviation)	10.67 seconds (10.80)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	40 players, 25.97 percent
Average confidence in "good"-judgement (standard deviation)	8.38 (3.08)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	114 players, 74.03 percent
Average confidence in "bad" judgement (standard deviation)	8.82 (2.92)

First three judgements

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.68 (3.72)	8.62 (2.99)	8.90 (2.85)
Average watch time (standard deviation)	8.73 seconds (6.87)	5.45 seconds (2.15)	4.44 seconds (2.54)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	24.39 percent	24.39 percent	14.63 percent
Average confidence in "good"-judgement (standard deviation)	5.50 (4.38)	6.00 (4.59)	10.00 (0.00)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	75.61 percent	75.61 percent	85.37 percent
Average confidence in "bad" judgement (standard deviation)	8.39 (3.26)	9.50 (1.53)	8.71 (3.05)

Judging Model *pq_assign_a2_5*

Associated program:

```
today = "Monday"
today = "Tuesday"
today = "Wednesday"
```

Description: A Variable is represented by a sheet with title. Assignments are visualized through writing words on it after crossing out the former word.

Time: 8 seconds
Concreteness: 100 percent of names are used in the model

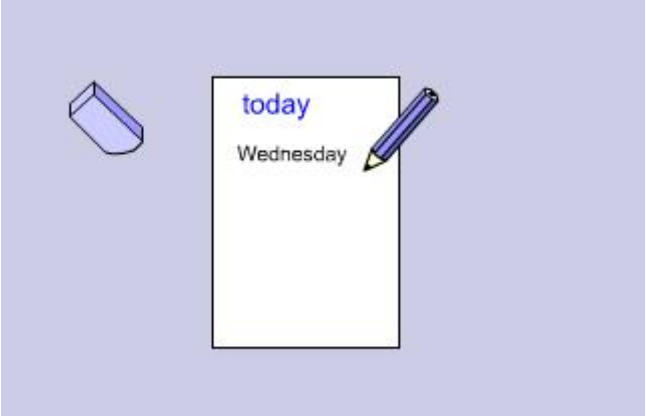
Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.93 (2.62)
Average watch time (standard deviation)	8.12 seconds (6.71)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	102 players, 66.23 percent
Average confidence in "good"-judgement (standard deviation)	9.17 (2.34)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	52 players, 33.77 percent
Average confidence in "bad" judgement (standard deviation)	8.46 (3.06)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.17 (3.31)	8.90 (2.37)	9.02 (3.00)
Average watch time (standard deviation)	8.90 seconds (8.45)	4.37 seconds (1.89)	3.73 seconds (2.37)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	75.61 percent	80.49 percent	75.61 percent
Average confidence in "good"-judgement (standard deviation)	8.71 (2.88)	9.09 (2.32)	10.00 (0.00)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	24.39 percent	19.51 percent	24.39 percent
Average confidence in "bad" judgement (standard deviation)	6.50 (4.12)	8.12 (2.59)	6.00 (5.16)

Judging Model *pq_assign_a2_6*

Associated program:

```
today = "Monday"
today = "Tuesday"
today = "Wednesday"
```



Description: A variable is represented by a sheet with title. Assignments are visualized through writing words on it after having erased the former word.

Time: 10 seconds
Concreteness: 100 percent of names are used in the model

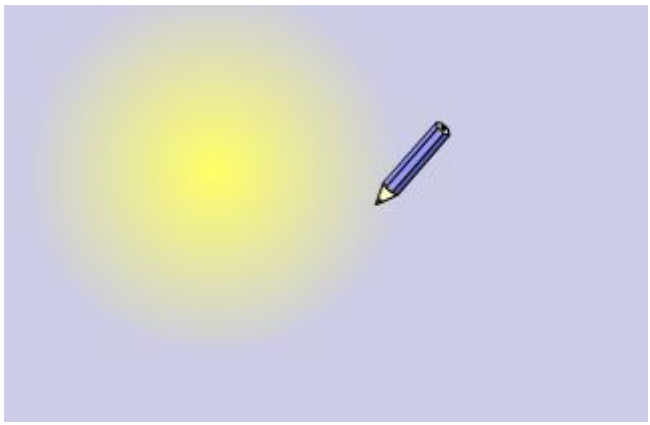
Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.77 (2.70)
Average watch time (standard deviation)	9.28 seconds (6.22)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	111 players, 72.08 percent
Average confidence in "good"-judgement (standard deviation)	9.46 (1.56)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	43 players, 27.92 percent
Average confidence in "bad" judgement (standard deviation)	6.98 (3.96)

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.05 (3.51)	8.62 (2.99)	8.66 (3.36)
Average watch time (standard deviation)	9.61 seconds (7.65)	5.83 seconds (3.31)	3.49 seconds (2.10)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	70.73 percent	85.37 percent	78.05 percent
Average confidence in "good"-judgement (standard deviation)	9.48 (1.55)	9.14 (2.26)	9.84 (0.88)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	29.27 percent	14.63 percent	21.95 percent
Average confidence in "bad" judgement (standard deviation)	4.58 (4.50)	5.00 (5.00)	4.44 (5.27)

Judging Model *pq_assign_a2_7*

Associated program:

```
today = "Monday"
today = "Tuesday"
today = "Wednesday"
```



Description: A Variable is represented by a sheet. An assignment is visualized through writing a word on it after complete destruction of the former sheet.

Time: 10 seconds
Concreteness: 100 percent of names are used in the model

Total number of first judgements	154
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.05 (3.15)
Average watch time (standard deviation)	11.79 seconds (10.68)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	96 players, 62.34 percent
Average confidence in "good"-judgement (standard deviation)	8.44 (2.93)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	58 players, 37.66 percent
Average confidence in "bad" judgement (standard deviation)	7.41 (3.41)

First three judgements

Total number of persons, who played this game at least three times	41		
	First session	Second session	Third session
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.07 (3.70)	8.75 (2.94)	8.05 (3.69)
Average watch time (standard deviation)	9.39 seconds (6.30)	6.80 seconds (4.91)	3.95 seconds (2.86)
Percentage of players, who declared that the model is appropriate ("good"-judgements).	60.98 percent	51.22 percent	29.27 percent
Average confidence in "good"-judgement (standard deviation)	8.20 (3.19)	7.86 (3.73)	7.92 (3.96)
Percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	39.02 percent	48.78 percent	70.73 percent
Average confidence in "bad" judgement (standard deviation)	5.31 (3.86)	9.74 (1.15)	8.10 (3.64)

Modeling messages in an object oriented program.

This game contains 5 tasks. I n each task the player has to judge several models, whether or not they are appropriate to explain the execution of some program statement.

Task 1	`bottle = Container (0.7)`
Task 2	`bottle.fill(0.4)`
Task 3	`bottle.empty()`
Task 4	`bottle.fill(0.4)`
Task 5	`vase.fill(bottle.empty())`

The following table provides some general information about the usage of this game (pq_objects).

Number of objective sessions (*not* taking into account that there are sometimes two players working together)	28 sessions
Number of subjective sessions (taking into account that there are sometimes two players working together)	36 sessions
Number of subjective uncanceled sessions (taking into account that there are sometimes two players working together)	22 sessions
Number of subjective *first* uncanceled sessions (taking into account that there are sometimes two players working together and some people play this game several times)	21 sessions
Number of persons, who played three times or more	0 persons

Some general information about the players of this category, who played the game at least *once*.

Professions	21 highschool students, 0 university students, 0 teachers, 0 professors and 0 others
Gender	4 female and 17 male persons
Hours a week spent on programming (standard deviation)	3.38 hours (2.69)
Roughly estimated experience in Python programming(standard deviation)	126.00 days (112.73)
Age (standard deviation)	17.24 years (0.77)
Population of the town, where the workshop took place	Less than 100 000: 1, 100 000 to 500 000: 12, more than 500 000: 8
Country	Germany: 20 other country: 1

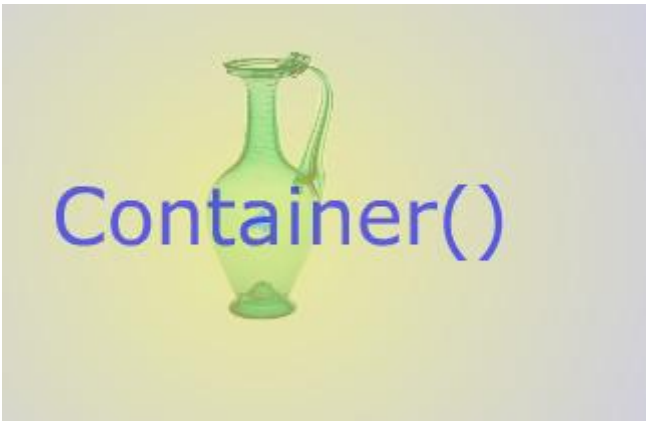
Task 1

Judging Model *pq_objects_a1_6*

Associated program:
```
bottle = Container (0.7)
```

Description: Visualization of instancing (a bottle). A number (size) is replaced by a bottle after a flash.

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



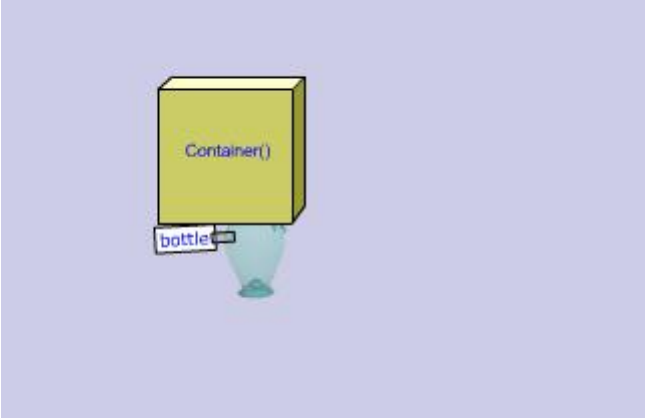
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.67 (3.98)
Average watch time (standard deviation)	16.62 seconds (18.08)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	16 players, 76.19 percent
Average confidence in "good"-judgement (standard deviation)	7.50 (3.65)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	5 players, 23.81 percent
Average confidence in "bad" judgement (standard deviation)	4.00 (4.18)

Judging Model *pq_objects_a1_5*

Associated program:
```
bottle = Container (0.7)
```

Description: Unappropriate visualization of instancing (a bottle).The class object receives a number (size) and a string (name) and returns a named object.

Time: 11 seconds
Concreteness: 100 percent of names are used in the model



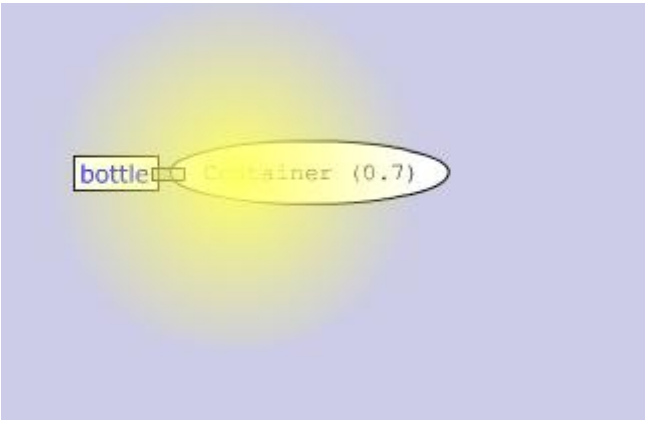
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.38 (3.75)
Average watch time (standard deviation)	20.52 seconds (25.00)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	12 players, 57.14 percent
Average confidence in "good"-judgement (standard deviation)	7.92 (3.34)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	9 players, 42.86 percent
Average confidence in "bad" judgement (standard deviation)	6.67 (4.33)

Judging Model *pq_objects_a1_4*

Associated program:
```
bottle = Container (0.7)
```

Description: Unappropriate visualization of the instantiation of an object (a bottle).

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.33 (2.42)
Average watch time (standard deviation)	11.76 seconds (12.33)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	16 players, 76.19 percent
Average confidence in "good"-judgement (standard deviation)	8.12 (2.50)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	5 players, 23.81 percent
Average confidence in "bad" judgement (standard deviation)	9.00 (2.24)

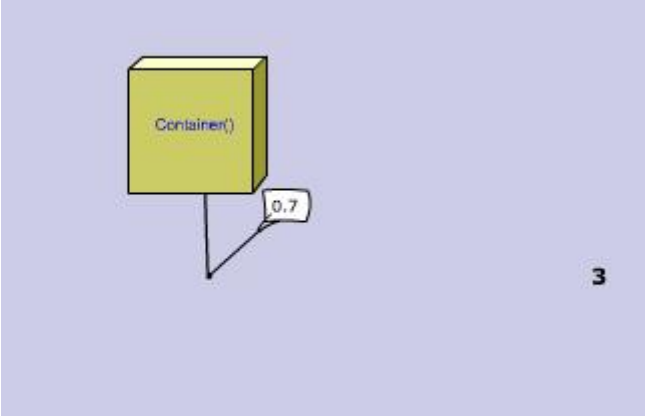
Judging Model *pq_objects_a1_3*

Associated program:

```
bottle = Container (0.7)
```

Description: Visualization of the instantiation of an object (a bottle). The class object is a box which takes a number (size) and returns a bottle.

Time: 12 seconds
Concreteness: 100 percent of names are used in the model



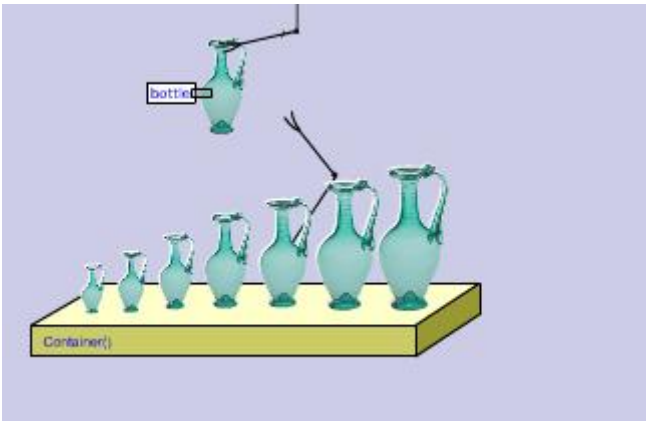
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.10 (3.35)
Average watch time (standard deviation)	14.29 seconds (10.22)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	17 players, 80.95 percent
Average confidence in "good"-judgement (standard deviation)	7.94 (3.56)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	4 players, 19.05 percent
Average confidence in "bad" judgement (standard deviation)	8.75 (2.50)

Judging Model *pq_objects_a1_2*

Associated program:
```
bottle = Container (0.7)
```

Description: Visualization of the instantiation of an object (a bottle). The class object has a reservoire of bottles of different sizes and chooses one.

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



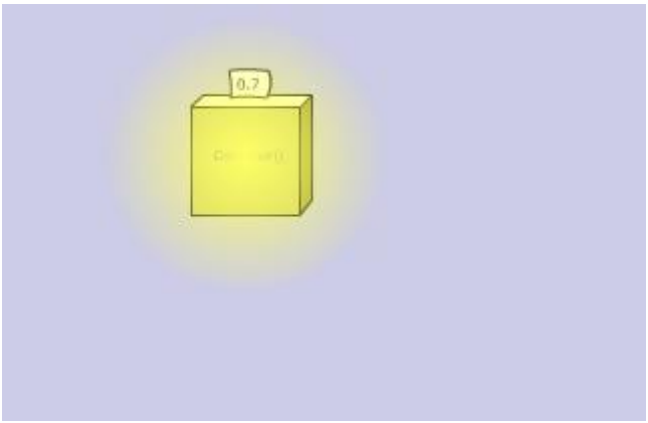
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.67 (3.98)
Average watch time (standard deviation)	26.05 seconds (29.00)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	16 players, 76.19 percent
Average confidence in "good"-judgement (standard deviation)	7.19 (4.07)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	5 players, 23.81 percent
Average confidence in "bad" judgement (standard deviation)	5.00 (3.54)

Judging Model *pq_objects_a1_1*

Associated program:
```
bottle = Container (0.7)
```

Description: Inappropriate visualization of the instantiation of an object (a bottle). During the instantiation the class object has been detroyed.

Time: 6 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.14 (2.99)
Average watch time (standard deviation)	21.67 seconds (35.51)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	17 players, 80.95 percent
Average confidence in "good"-judgement (standard deviation)	6.76 (3.03)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	4 players, 19.05 percent
Average confidence in "bad" judgement (standard deviation)	8.75 (2.50)

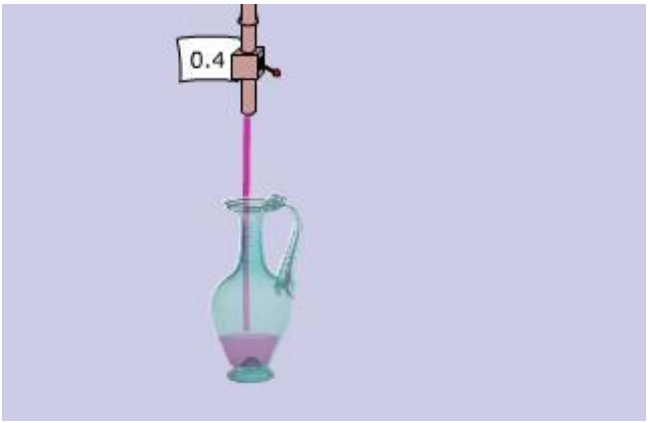
Task 2

Judging Model *pq_objects_a2_1*

Associated program:
`bottle.fill(0.4)`

Description: Visualization of the message bottle.fill(0.4) using two entities for the method and argument. The method is sent to the bottle and the argument to a filling device.

Time: 14 seconds
Concreteness: 50 percent of names are used in the model



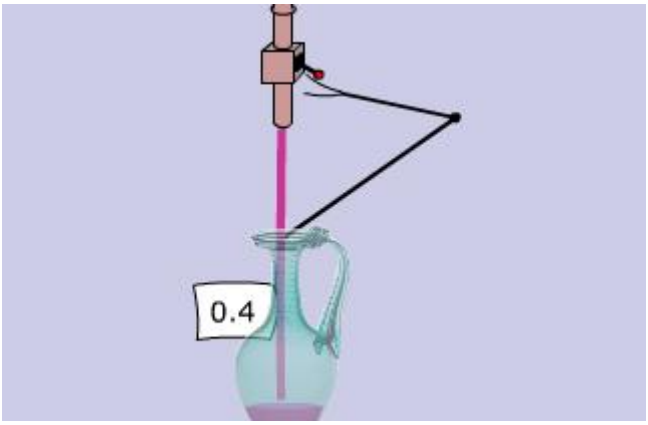
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.57 (2.80)
Average watch time (standard deviation)	11.38 seconds (7.77)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	14 players, 66.67 percent
Average confidence in "good"-judgement (standard deviation)	9.64 (1.34)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	7 players, 33.33 percent
Average confidence in "bad" judgement (standard deviation)	6.43 (3.78)

Judging Model *pq_objects_a2_2*

Associated program:
`bottle.fill(0.4)`

Description: Visualization of the message bottle.fill(0.4) using two entities for the method and argument. Both are sent to a bottle, which moves to a filling device.

Time: 13 seconds
Concreteness: 50 percent of names are used in the model



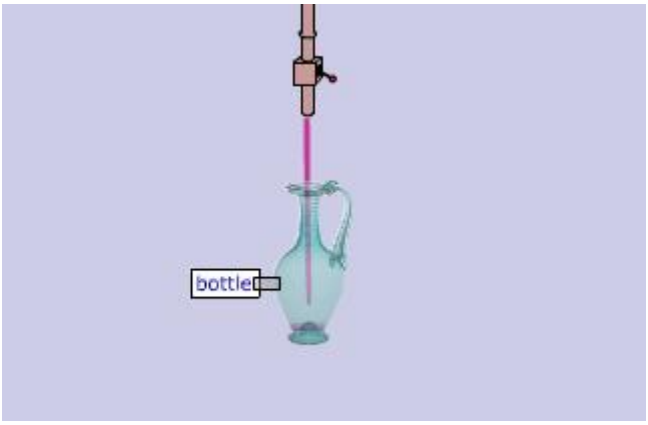
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.62 (3.75)
Average watch time (standard deviation)	12.10 seconds (8.99)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	18 players, 85.71 percent
Average confidence in "good"-judgement (standard deviation)	8.89 (2.14)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	3 players, 14.29 percent
Average confidence in "bad" judgement (standard deviation)	0.00 (0.00)

Judging Model *pq_objects_a2_3*

Associated program:
`bottle.fill(0.4)`

Description: Unappropriate visualization of the message bottle.fill(0.4) by an oval moving to a filling device which fills a bottle.

Time: 7 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.38 (3.75)
Average watch time (standard deviation)	9.43 seconds (6.60)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	15 players, 71.43 percent
Average confidence in "good"-judgement (standard deviation)	7.67 (3.20)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	6 players, 28.57 percent
Average confidence in "bad" judgement (standard deviation)	6.67 (5.16)

Judging Model *pq_objects_a2_4*

Associated program:
`bottle.fill(0.4)`

Description: Visualization of the message bottle.fill(0.4) by an oval moving to a bottle which is filled in a magical way.

Time: 6 seconds
Concreteness: 100 percent of names are used in the model

Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.38 (3.75)
Average watch time (standard deviation)	11.33 seconds (8.61)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	19 players, 90.48 percent
Average confidence in "good"-judgement (standard deviation)	7.89 (3.46)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	2 players, 9.52 percent
Average confidence in "bad" judgement (standard deviation)	2.50 (3.54)

Task 3

Judging Model *pq_objects_a3_1*

Associated program:
`bottle.empty()`

Description: Visualization of the message bottle.empty() by an oval moving to a bottle which pours out its content.

Time: 5 seconds
Concreteness: 100 percent of names are used in the model

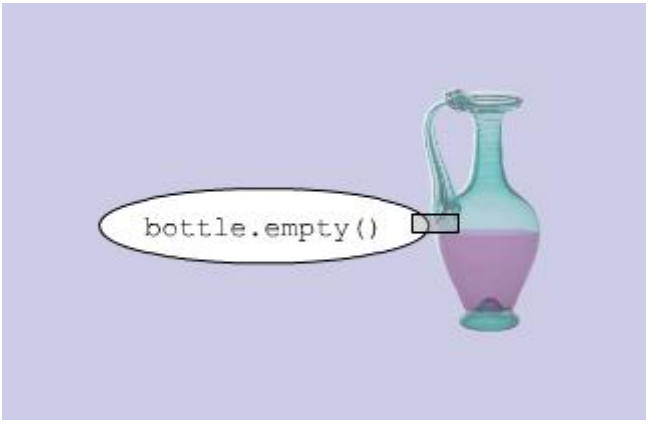
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.57 (2.80)
Average watch time (standard deviation)	5.67 seconds (3.97)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	17 players, 80.95 percent
Average confidence in "good"-judgement (standard deviation)	8.53 (2.94)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	4 players, 19.05 percent
Average confidence in "bad" judgement (standard deviation)	8.75 (2.50)

Judging Model *pq_objects_a3_3*

Associated program:
`bottle.empty()`

Description: Unappropriate visualization of the message bottle.empty() by an oval moving to a bottle. This oval is glued at the bottle.

Time: 4 seconds
Concreteness: 100 percent of names are used in the model



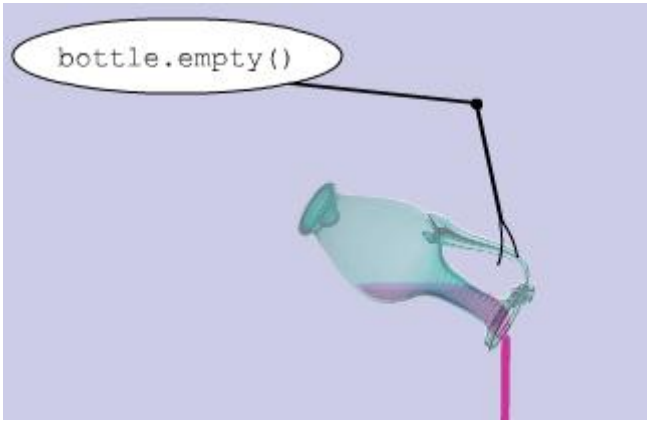
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	9.52 (1.50)
Average watch time (standard deviation)	9.29 seconds (7.36)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	5 players, 23.81 percent
Average confidence in "good"-judgement (standard deviation)	10.00 (0.00)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	16 players, 76.19 percent
Average confidence in "bad" judgement (standard deviation)	9.38 (1.71)

Judging Model *pq_objects_a3_2*

Associated program:
`bottle.empty()`

Description: Visualization of the message bottle.empty() by an oval moving to a bottle. This oval has a mnipulator arm which grabs the bottle and empties it.

Time: 5 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.14 (4.35)
Average watch time (standard deviation)	8.48 seconds (6.85)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	17 players, 80.95 percent
Average confidence in "good"-judgement (standard deviation)	8.53 (3.43)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	4 players, 19.05 percent
Average confidence in "bad" judgement (standard deviation)	1.25 (2.50)

Judging Model *pq_objects_a3_5*

Associated program:
`bottle.empty()`

Description: Visualization of the message bottle.empty() by an oval moving to a bottle. When it hits it, the bottle empties.

Time: 5 seconds
Concreteness: 100 percent of names are used in the model

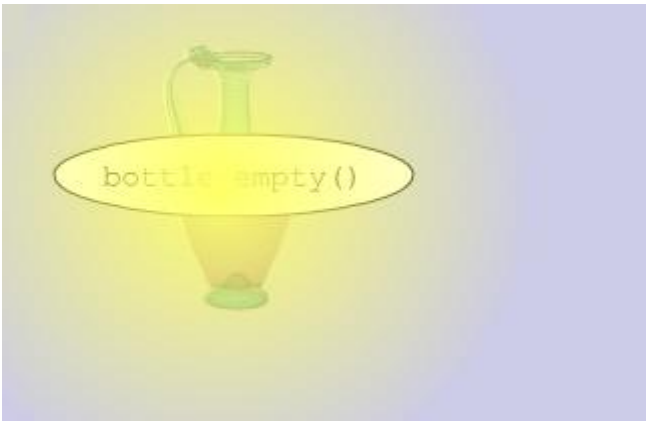
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	8.81 (3.12)
Average watch time (standard deviation)	6.10 seconds (4.45)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	17 players, 80.95 percent
Average confidence in "good"-judgement (standard deviation)	9.71 (1.21)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	4 players, 19.05 percent
Average confidence in "bad" judgement (standard deviation)	5.00 (5.77)

Judging Model *pq_objects_a3_4*

Associated program:
`bottle.empty()`

Description: Unappropriate visualization of the message bottle.empty() by an oval moving to a bottle. When it hits the bottle both vanish.

Time: 5 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	9.05 (2.56)
Average watch time (standard deviation)	8.95 seconds (7.63)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	4 players, 19.05 percent
Average confidence in "good"-judgement (standard deviation)	10.00 (0.00)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	17 players, 80.95 percent
Average confidence in "bad" judgement (standard deviation)	8.82 (2.81)

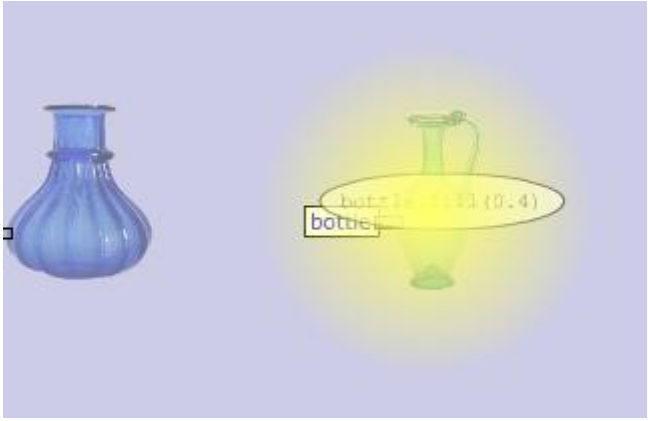
Task 4

Judging Model *pq_objects_a4_2*

Associated program:
`bottle.fill(0.4)`

Description: An oval representing the message bottle.fill(0.4) moves to a vase and then to a bottle, triggering a flash and the filling of the bottle with fluid.

Time: 8 seconds
Concreteness: 100 percent of names are used in the model



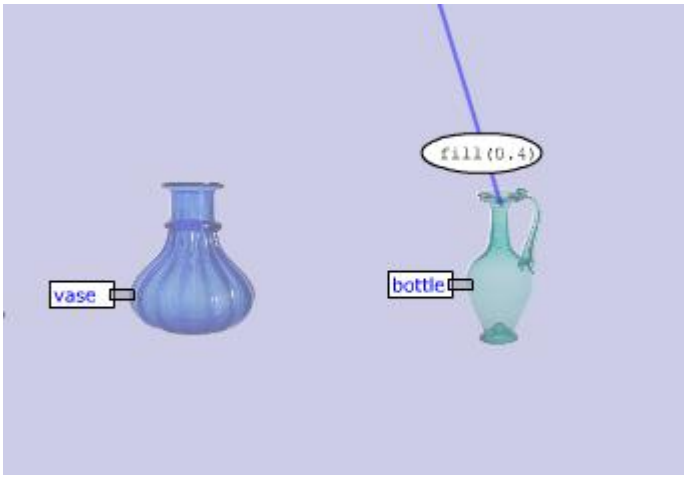
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.90 (4.32)
Average watch time (standard deviation)	12.05 seconds (12.84)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	15 players, 71.43 percent
Average confidence in "good"-judgement (standard deviation)	9.00 (2.80)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	6 players, 28.57 percent
Average confidence in "bad" judgement (standard deviation)	1.67 (2.58)

Judging Model *pq_objects_a4_3*

Associated program:
`bottle.fill(0.4)`

Description: The transmission of the message bottle.fill(0.4) is divied into two phases. First a beam (blue line) to the bottle appears, then the message glides along the beam.

Time: 7 seconds
Concreteness: 100 percent of names are used in the model



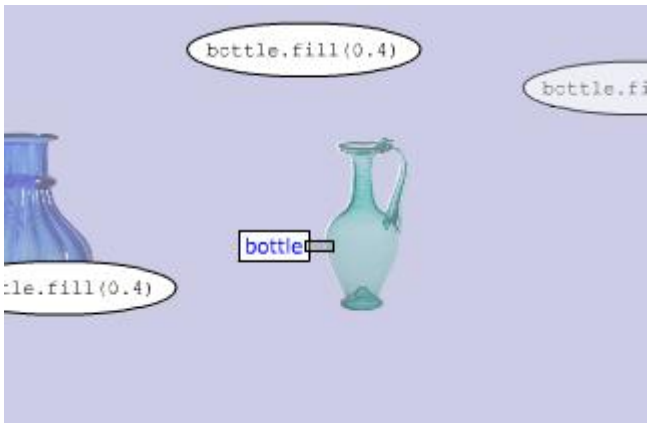
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.38 (3.75)
Average watch time (standard deviation)	11.62 seconds (13.39)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	13 players, 61.90 percent
Average confidence in "good"-judgement (standard deviation)	7.69 (3.30)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	8 players, 38.10 percent
Average confidence in "bad" judgement (standard deviation)	6.88 (4.58)

Judging Model *pq_objects_a4_1*

Associated program:
`bottle.fill(0.4)`

Description: Many ovals representing the message bottle.fill(0.4) are moving over the scene. One hits the bottle and the bottle is filled magically.

Time: 8 seconds
Concreteness: 100 percent of names are used in the model



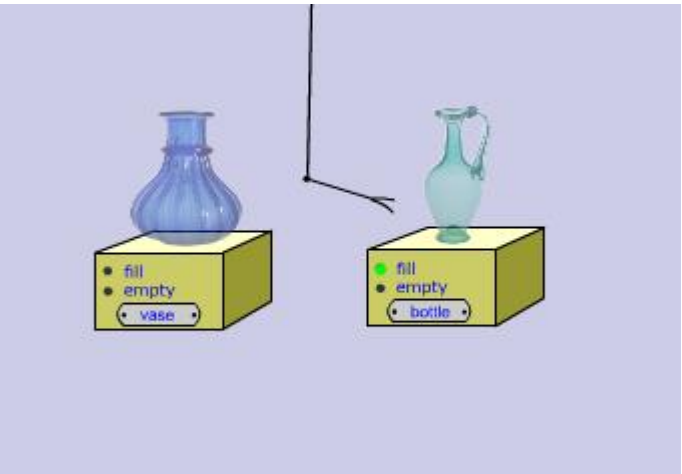
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.14 (3.73)
Average watch time (standard deviation)	9.43 seconds (7.13)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	7 players, 33.33 percent
Average confidence in "good"-judgement (standard deviation)	7.14 (3.93)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	14 players, 66.67 percent
Average confidence in "bad" judgement (standard deviation)	7.14 (3.78)

Judging Model *pq_objects_a4_4*

Associated program:
`bottle.fill(0.4)`

Description: The message bottle.fill(0.4) is not represented by an entity. Instead a manipulator arm presses a button named fill on the bottle-object and initiates the filling.

Time: 7 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.67 (3.65)
Average watch time (standard deviation)	13.05 seconds (11.19)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	17 players, 80.95 percent
Average confidence in "good"-judgement (standard deviation)	7.65 (2.57)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	4 players, 19.05 percent
Average confidence in "bad" judgement (standard deviation)	2.50 (5.00)

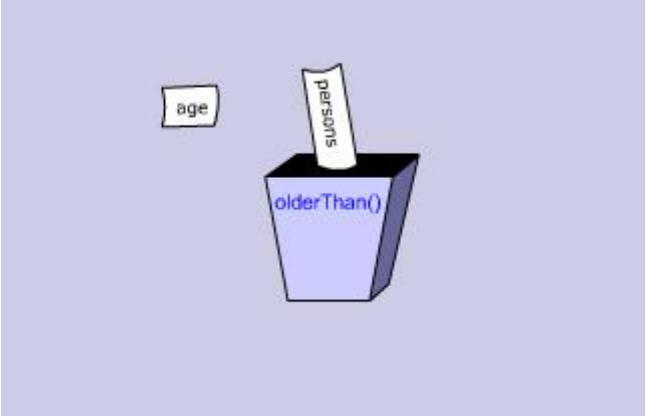
Task 5

Judging Model *pq_objects_a5_3*

Associated program:
`vase.fill(bottle.empty())`

Description: The message vase.fill(bottle.empty()) is represented by an oval going to the vase. A message bottle.empty() goes to the bottle which pours its content into the vase.

Time: 7 seconds
Concreteness: 100 percent of names are used in the model



Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.67 (4.28)
Average watch time (standard deviation)	11.29 seconds (10.27)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	11 players, 52.38 percent
Average confidence in "good"-judgement (standard deviation)	7.27 (3.44)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	10 players, 47.62 percent
Average confidence in "bad" judgement (standard deviation)	6.00 (5.16)

Judging Model *pq_objects_a5_2*

Associated program:
`vase.fill(bottle.empty())`

Description: The nested message vase.fill(bottle.empty()) is visualized by two messages going to the bottle and the vase. The bottle pours its content into the vase.

Time: 7 seconds
Concreteness: 100 percent of names are used in the model

Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.14 (4.35)
Average watch time (standard deviation)	8.00 seconds (6.96)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	9 players, 42.86 percent
Average confidence in "good"-judgement (standard deviation)	8.89 (2.20)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	12 players, 57.14 percent
Average confidence in "bad" judgement (standard deviation)	5.83 (5.15)

Judging Model *pq_objects_a5_1*

Associated program:
`vase.fill(bottle.empty())`

Description: Nested message vase.fill(bottle.empty()) is visualized by an empty()-message to bottle and a fill()-message from bottle to vase

Time: 7 seconds
Concreteness: 50 percent of names are used in the model

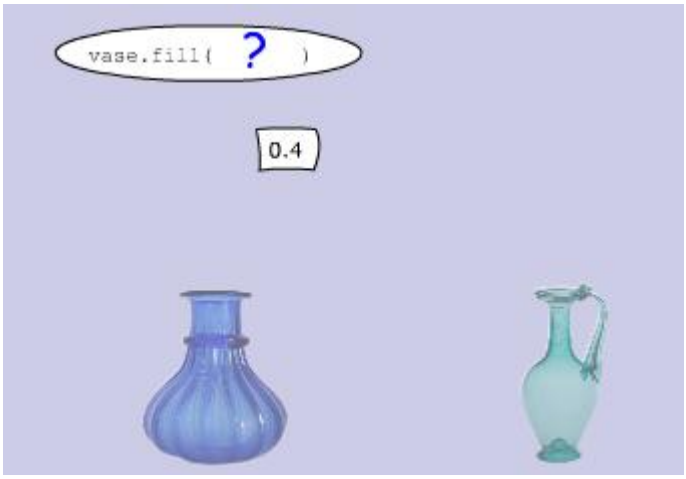
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.19 (4.72)
Average watch time (standard deviation)	9.86 seconds (9.25)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	14 players, 66.67 percent
Average confidence in "good"-judgement (standard deviation)	7.14 (4.26)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	7 players, 33.33 percent
Average confidence in "bad" judgement (standard deviation)	4.29 (5.35)

Judging Model *pq_objects_a5_7*

Associated program:
`vase.fill(bottle.empty())`

Description: The message vase.fill(bottle.empty()) is represented by a message that can send messages itself.

Time: 9 seconds
Concreteness: 50 percent of names are used in the model



First Judgements

Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.86 (3.73)
Average watch time (standard deviation)	10.10 seconds (13.73)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	15 players, 71.43 percent
Average confidence in "good"-judgement (standard deviation)	8.00 (3.68)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	6 players, 28.57 percent
Average confidence in "bad" judgement (standard deviation)	7.50 (4.18)

Judging Model *pq_objects_a5_6*

Associated program:
`vase.fill(bottle.empty())`

Description: The message vase.fill(bottle.empty()) is represented by two messages: empty() goes to the bottle, it returns 0.4, then message fill(0.4) goes to the vase.

Time: 9 seconds
Concreteness: 50 percent of names are used in the model

Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.62 (3.75)
Average watch time (standard deviation)	9.14 seconds (6.78)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	14 players, 66.67 percent
Average confidence in "good"-judgement (standard deviation)	8.57 (3.06)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	7 players, 33.33 percent
Average confidence in "bad" judgement (standard deviation)	5.71 (4.50)

Judging Model *pq_objects_a5_5*

Associated program:
`vase.fill(bottle.empty())`

Description: The message vase.fill(bottle.empty()) is not represented by an entity. A manipulator presses buttons on the objects and initiates the execution of methods.

Time: 10 seconds
Concreteness: 100 percent of names are used in the model

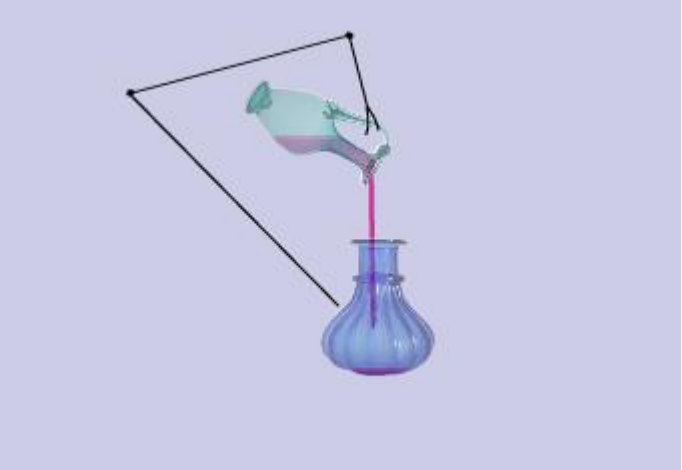
Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	7.38 (4.07)
Average watch time (standard deviation)	11.33 seconds (11.46)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	17 players, 80.95 percent
Average confidence in "good"-judgement (standard deviation)	8.53 (2.94)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	4 players, 19.05 percent
Average confidence in "bad" judgement (standard deviation)	2.50 (5.00)

Judging Model *pq_objects_a5_4*

Associated program:
`vase.fill(bottle.empty())`

Description: The message vase.fill(bottle.empty()) is sent to the vase. The vase grabs the bottle (using an manipulator) and pours its content into itself.

Time: 9 seconds
Concreteness: 100 percent of names are used in the model



First Judgements

Total number of first judgements	21
Average bet points (0 to 10) representing the player's confidence (standard deviation)	6.43 (4.23)
Average watch time (standard deviation)	35.57 seconds (114.62)
Number and percentage of players, who declared that the model is appropriate ("good"-judgements).	14 players, 66.67 percent
Average confidence in "good"-judgement (standard deviation)	7.86 (3.23)
Number and percentage of players, who declared that the model is *not* appropriate ("bad"-judgements)	7 players, 33.33 percent
Average confidence in "bad" judgement (standard deviation)	3.57 (4.76)

4 Ergänzungen zu Steuerungsmodellen

4.1 Verzweigungen in Datenfluss-Modellen

Datenfluss-orientierte visuelle Programmiersprachen müssen bedingte Anweisungen durch relativ aufwändige Konstrukte abbilden. Das System DRLP (Anjaneyulu & Anderson 1992) verwendet dazu gesteuerte „Ventile“. Das Diagramm in Abb. 98 modelliert eine Funktion, die ein Element an den Anfang einer Liste setzt, sofern es noch nicht in der Liste vorkommt. Es handelt sich um einen Graph mit Knoten (Rechtecke, Trapez, Dreiecke) und gerichteten Kanten (Richtung von oben nach unten). Die oberen Rechtecke sind Eingänge für Daten, die unteren Ausgänge. Im oberen Dreieck (MEMBER) wird geprüft, ob das Element (von links) in der Liste (von rechts) enthalten ist. Das Ergebnis ist ein boolescher Wert, aus dem in der trapezförmigen Einheit ein Steuersignal erzeugt wird, das entweder die rechte oder die linke Sperre (ENABLE) öffnet.

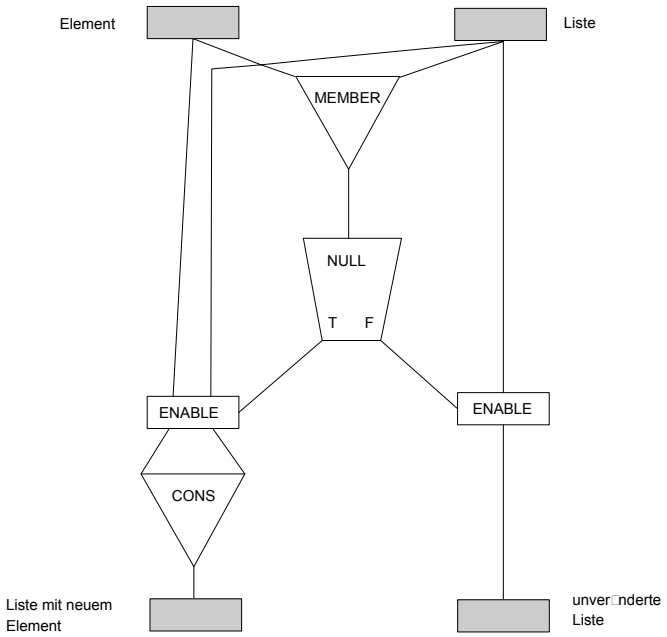


Abb. 98: Ein datenflussorientiertes visuelles Programm mit DRLP nach Anjaneyulu & Anderson 1992, S. 589

Das Modell unterstützt zwei intuitive Modelle zum Datentransport.

Zum einen kann man sich vorstellen, dass über die Datenleitungen Entitäten wandern, die Daten darstellen. In diesem Fall existieren aber von einem Datum mehrere Kopien (für jeden Weg eine), die von unterschiedlichen Einheiten verarbeitet werden oder ihr Ziel gar nicht erreichen, weil der Fluss durch eine Sperre unterbrochen ist.

Zum zweiten kann ein Zustandsmodell für Daten verwendet werden. Wie bei elektronischen Schaltkreisen befinden sich die Datenleitungen in einem Zustand, der durch den Knoten am Anfang der Kante determiniert ist und der die übertragenen Daten repräsentiert.

4.2 Der Fetch-Execute-Zyklus als Beispiel einer Schleife

Vom Benutzer wird die Eingabe eines Kommandos erwartet. Wenn dieses Kommando keine Beendigung des Dialogs signalisiert (quit), wird es ausgeführt, ein neues Kommando abgefragt usw.

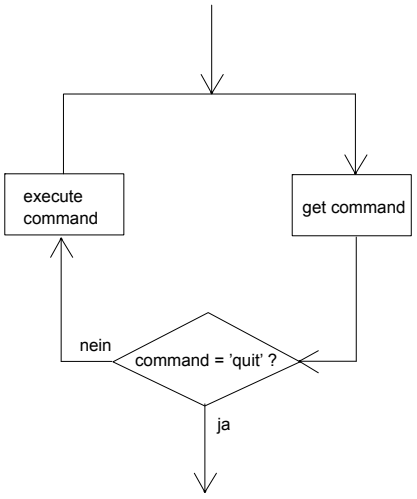


Abb. 99: Fetch-execute-Schleife eines interaktiven Systems. Nach Sommerville 1997, S. 287

Die Steuerungseinheit, die über die Wiederholung entscheidet, muss sich an einer bestimmten Stelle der Sequenz befinden, nämlich vor dem Block *execute command*. Sie ist nicht einer holistischen zu wiederholenden Aktion übergeordnet sondern „gleichrangiger" Teil der Sequenz.

Der fetch-execute-Zyklus kann mit Python auf folgende zwei Weisen implementiert werden. Erste Variante:

```
while True:
     c = raw_input()
     if c == "Quit":
         break
      else:
     execute(c)
```

Zweite Variante:

```
c = raw_input()
while c != "Quit":
        execute(c)
        c = raw_input()
```

Die erste Variante ist eine Implementierung des Schleifenmodells, das im Flussdiagramm aus Abb. 99 wiedergegeben wird. Mit `while True:` … wird im Prinzip eine Endlosschleife definiert, d.h. die (von der Programmiersprachensyntax geforderte) while-Bedingung ist trivial und sorgt nicht für eine Beendigung der Wiederholung. Der Ausstieg aus der Schleife erfolgt gegebenenfalls nach Prüfung der Eingabe und wird durch die Anweisung break ausgelöst. Die Kontrolle des Schleifenausstiegs ist also Element einer zyklischen Anweisungsfolge und nicht ihr übergeordnet.

In der zweiten Variante wird keine Programmverzweigung sondern eine while-Bedingung verwendet, die die Anzahl der Wiederholungen steuert. Unter dem Gesichtspunkt der Modellierung hat diese Implementierung jedoch einen Nachteil. Die zusammengehörige Folge von Eingabe und Auswertung der Eingabe wird auseinander gebrochen. Der while-Anweisung ist eine einmalige Eingabe-Anweisung vorgeschaltet. Die wiederholte Anweisungsfolge „Auswertung der Eingabe" und „neue Eingabe" ist dagegen ist keine sinnvolle Gestalt. Sie kann nicht für sich stehen (d.h. sie ist nicht ohne den Kontext der Wiederholung denkbar) da für die Auswertung im ersten Schritt eine Eingabe aus dem letzen Durchgang verwendet wird.

Das Beispiel zeigt, dass manche Wiederholungen besser mit dem Schleifenkonzept als mit dem Konzept der kontrollierten Wiederholung einer holistischen Aktivität modelliert werden.

4.3 Assoziierte Konzepte zur Rekursion

Levy & Lapidot (2000, 2001) beobachteten Schülerinnen und Schüler, die an einer Unterrichtssequenz zur Rekursion („classification and discussion learning activity", CDA) mit folgender Struktur teilnahmen:

1. Konfrontation mit rekursiven Phänomenen (z.B. Bilder)
2. Gruppenarbeit: Klassifizierung der rekursiven Phänomene
3. Präsentation der Klassifizierung, gefundenen Kategorien und Kriterien
4. Diskussion im Plenum mit dem Ziel, gemeinsame Merkmale von Rekursivität zu finden.

Auf Grund von Schüleräußerung identifizierten sie verschiedene gedankliche Konzepte – sie sprechen von Präkonzepten (pre-conceptions) – , die bei der Analyse rekursiver Phänomene ihrer Meinung nach eine Schlüsselrolle einnehmen (Levy & Lapidot 2001). Beispiele (in Anführungsstrichen beispielhafte Schüleräußerungen):

- Unendlich oder endlich. Hier geht es um den Abbruch oder die unendliche Fortsetzung eines rekursiven Vorgangs.
- Regelmäßigkeit
- Gradualität („vom Kleinen zum Großen" und umgekehrt)
- Periodizität
- Zurückkehren
- Sequentialität (z.B. „aufsteigende und absteigende Sequenzen")
- Abhängigkeit
- Selbstbezug („Dinge, die sich selbst bauen")
- Zirkularität
- Enthalten sein

Einige dieser Konzepte (unendlich/endlich, Zirkularität, enthalten sein) tauchen bereits zu Beginn der Auseinandersetzung mit Rekursivität auf. Manche Werke der bildenden Kunst thematisieren geradezu einen Aspekt von Rekursion. Ein Beispiel ist das Bild von Escher, das zwei Hände darstellt, die sich gegenseitig zeichnen (Drawing Hands, 1948). Sie zeigen das Paradox der wechselseitigen Abhängigkeit zweier Objekte. Keines kann ohne das andere existieren. In einer rekursiven Funktion verwende ich dieselbe Funktion, die ich gerade definiere und die also noch gar nicht existiert.

Es gibt konkrete Phänomene, die jeweils einige dieser Konzepte in einer geschlossenen Gestalt repräsentieren (Beispiele für „rekursive Phänomene" findet man in Levy und Lapidot, 2000). So sind in den russian dolls die Konzepte *Regelmäßigkeit*, *Gradualität* und *enthalten sein* verkörpert. Zwei gegenüber liegende parallele Spiegel eröffnen einen Blick in die Unendlichkeit, die durch rekursive Funktionen ohne Abbruchbedingung erzeugt wird.

Man kann diese erlebbaren Phänomene als Repräsentationen intuitiver Modelle betrachten, die mit Rekursion assoziiert werden, d.h. die bei der Interpretation oder Generierung rekursiver Funktionen verwendet werden. Diese Intuitionen sind jedoch – jede für sich – keine vollständigen Modelle. Sie sind fokussieren nur auf bestimmte Aspekte und können Teil eines Clusters von Intuitionen sein, das bei der intellektuellen Auseinandersetzung mit einer rekursiven Funktion verwendet wird.

Nach Wu, Dale & Bethel (1998) fördern „konkrete Modelle" der Rekursion (russian dolls, Prozess-Tracing, und Stack-Simulationen) unabhängig vom Lerntyp besser der Lernprozess als „abstrakte Modelle" (Strukturschablonen, mathematische Induktion).

4.4 Fehlerhafte Verwendung des Modells der Selbstaufforderung bei eingebetteter Rekursion

Dicheva und Close (1996) haben Schüler befragt, welche Bildschirmausgabe die Logo-Prozedur aus Tab. 44 mit einem eingebetteten rekursiven Aufruf liefert (initialer Aufruf: `pattern3 "LEGO`). Die Tabelle zeigt die korrekte und eine falsche Lösung.

Logo-Programm	Korrekte Lösung	Falsche Lösung (Beispiel)
`to pattern3 :w`	`LEGO`	`LEGO`
`  if empty? :w [stop]`	`EGO`	`EGO`
`  print :w`	`GO`	`GO`
`  pattern3 bf :w`	`O`	`O`
`  print :w`	`O`	`O`
`end`	`GO`	
	`EGO`	
	`LEGO`	

Tab. 44: Rekursive Logo-Prozedur mit einer korrekten und zwei falschen Bildschirmausgaben beim Aufruf von `pattern3 "LEGO`

Die richtige Lösung kann man nur finden, wenn man annimmt, dass beim rekursiven Aufruf ein neuer Prozess (Akteur) entsteht, der eine Kopie des Strings w ohne das letzte Zeichen einer (eigenen) internen Variablen w zuordnet und verarbeitet (Delegationsmodell, siehe nächster Abschnitt).

Auf die falsche Lösung (Tab. 44 rechts) kommt man, wenn man von der Annahme eines einzigen Akteurs ausgeht. Das Tracing kann z.B. folgendermaßen erklärt werden:

Zuerst wird die Zeichenkette LEGO auf den Bildschirm geschrieben. Beim rekursiven Aufruf wird die Ausführung der Aktionssequenz abgebrochen und von neuem begonnen (gleicher Akteur). Allerdings wird diesmal die Zeichenkette ohne den ersten Buchstaben (`bf :w`) verwendet. In der gleichen Weise wird fortgefahren bis der String nur aus einem Zeichen besteht. Dann wird der rekursive Aufruf `pattern3 bf :w` übersprungen (weil er nichts bewirkt) und die mit der letzten print-Anweisung die aktuelle Zeichenkette O nochmals ausgegeben.

Auch andere Fehlvorstellungen zur Rekursion, die Dicheva und Close identifiziert haben, basieren auf der Vorstellung eines einzigen Akteurs (Dicheva & Close, 1996; Close & Dicheva, 1997).

4.5 Anwendung des Delegationsmodells zur Visualisierung der Arbeitsweise rekursiver Funktionen

Das Delegationsmodell wird in unterschiedlichen Visualisierungsansätzen verwendet. Sie unterscheiden sich vor allem in der Durchlässigkeit der Systemgrenze.

Im Boxmodell wird der Akteur (Prozess) als Kasten dargestellt, in dessen Inneren bei Funktionsaufrufen weitere Kästen entstehen. In dieser Schachtelung klingt das von Levy, Lapidot & Paz (2001) beobachtete zur Rekursion assoziierte Konzept „enthalten sein“ an. Mit dem Boxmodell kann die Ausführung einer rekursiven Funktion durch ein System dargestellt werden, das nur über Ein- und Ausgänge Kontakt mit der Umgebung hat.

Eine zweite Variante des Delegationsmodells verwendet „Boxen mit Seitentüren“ und verzichtet auf Schachtelung. Beim rekursiven Aufruf werden Daten nicht durch den Ausgang (durch den nur das Ergebnis nach außen darf) sondern gewissermaßen durch eine „Seitentür“ an einen anderen Akteur übergeben. Dieser – und das ist das Entscheidende – befindet sich außerhalb des aufrufenden Akteurs.

Delegationsmodelle ohne Schachtelung werden – etwas versteckt – in verschiedenen Visualisierungstechniken zur Darstellung der Arbeitsweise rekursiver Funktionen verwendet:

- Im Stackmodell wandert die Repräsentation des wartenden Funktionsakteurs (der gerade eine Aufgabe delegiert hat) – der Execution Frame – auf einen Stack. Wenn ein Akteur (aktiver Prozess) seine Aufgabe abgeschlossen hat (der Prozess terminiert), übergibt er sein Ergebnis an den Execution Frame, der zuoberst auf dem Stack liegt. Eine Besonderheit dieses Modells liegt darin, dass immer nur ein Funktionsakteur aktiv sein kann und die anderen im Stack warten.
- Im Droid-Modell (Manis & Little, 1995) wird ein Funktionsaufruf durch eine grafische Repräsentation dargestellt, die u.a. den Namen der Funktion, die übergebenen Argumente und einen Pfeil zum aufrufenden Droid enthält. Auch hier gibt es keine Schachtelung.
- In der Veranschaulichung der Arbeitsweise einer rekursiven Funktion, die eine komplexe Datenstruktur bearbeitet (z.B. Sortieren einer Liste), können Akteure durch Bezeichnung ihres Tätigkeitsbereichs angedeutet werden. In dem Visualisierungssytem von Stern & Naish (2002) für den Quicksort-Algorithmus markieren Balken unterhalb einer Liste von Zahlen, welcher Prozess für welchen Teilbereich zuständig ist. Jeder Balken repräsentiert einen separaten Akteur.

Das Delegationsmodell (ohne Schachtelung mit statischen Akteuren) kann in Rollenspielen veranschaulicht werden, indem mehrere Personen gemeinsam einen rekursiven Algorithmus ausführen und jeweils Teilaufgaben an andere Personen delegieren. In dem folgenden Beispiel gehen wir davon aus, dass eine genügend große Anzahl von Personen in einer Reihe (zum Beispiel um einen Tisch herum) sitzt. Der Versuchsleiter gibt einer Person eine Tüte mit verschiedenfarbigen Gummibärchen in die Hand gegeben und fordert sie auf: „Iss rote Gummibärchen“. Was unter dieser Aktivität zu verstehen ist, wird in folgendem Algorithmus spezifiziert:

```
Algorithmus Iss rote Gummibärchen
     Nimm die Tüte vom rechten Nachbarn in Empfang
     Wenn die Tüte noch ein rotes Gummibärchen enthält:
           Entnimm der Tüte ein rotes Gummibärchen
           Gib die Tüte dem linken Nachbarn mit der Aufforderung
           „Iss rote Gummibärchen“
           Nimm die Tüte vom linken Nachbarn wieder in Empfang
      Gib die Tüte dem rechten Nachbarn zurück
```

4.6 Bevorzugung vollständiger Modelle zur Darstellung einer rekursiven Funktion

In verschiedenen Python Visuals wurden Teilnehmer mit verschiedenen Animationen zur Erklärung der Arbeitsweise rekursiver Funktionen konfrontiert (Berechnung der Fakultät und Spiegeln eines Wortes). Dabei gibt es zwei Varianten: In der längeren Version wird die Rekursion vollständig bis zum Abbruch bei Erreichen des Elementarfalls dargestellt. In der kürzeren Variante wird nur der erste rekursive Aufruf visualisiert. Abb. 100 zeigt Screenshots jeweils aus den längeren Varianten.

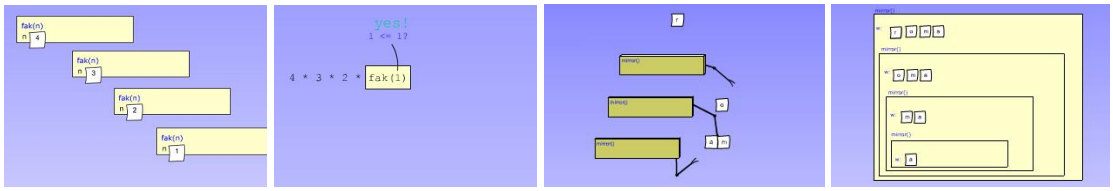


Abb. 100: Visualisierung der Arbeitsweise rekursiver Funktionen: Fakultät (Execution Frames und Termersetzungen) und Spiegeln eines Wortes (nicht geschachtelte Akteure, geschachtelte Execution Frames)

Die Spieler sollten u.a. angeben, welches Modell sie verwenden würden, wenn sie jemandem die Arbeitsweise der rekursiven Funktion erklären müssten. Tab. 45 zeigt die Verteilung der Wahlen. Die letzte Spalte gibt das Signifkanzniveau des χ^2-Tests bei Vergleich der beobachteten Wahlen mit einer Gleichverteilung wieder.

	ein Schritt	volle Rekursion	p
Berechnung der Fakultät, Execution Frames	5	13	0,05934645
Berechnung der Fakultät, Termersetzen	2	16	0,00096743
Wort spiegeln, nicht geschachtelt	3	12	0,02013675
Wort spiegeln, geschachtelt	4	11	0,07070116

Tab. 45: Ergebnisse von ersten Sessions zweier Python Visuals. Die Zahlen in Spalte 2 und 3 geben an, wie viele Personen das jeweilige Modell zur Erklärung der Arbeitsweise der Funktion verwenden würden. Erste und zweite Zeile: Rekursive Berechnung der Fakultät. 36 Spieler, darunter 29 Schüler und 5 Studenten. 31 der Spieler haben an Workshops mit der PVS teilgenommen. Dritte und vierte Zeile: Rekursives Spiegeln eines Wortes. 30 Spieler, darunter 22 Schüler und 4 Studenten. 24 der Spieler haben an Workshops mit der PVS teilgenommen.

Die meisten Teilnehmer der PVS bevorzugten für Erklärungen die längeren und komplexeren Animationen, die die Rekursion vollständig bis zum Elementarfall nachvollziehen.

5 Ergänzungen zu Verarbeitungsmodellen

5.1 Entstehungsprozesse im Alltag

Es gibt unterschiedliche Konzepte, um die *Entstehung* einer Sache zu beschreiben. Dazu gehören

1. die spontane Manifestation aus dem Nichts (z.B. Entstehung einer Wolke am Himmel) ohne expliziten Produzenten,
2. Erschaffung einer Sache durch einen Produzenten (z.B. Nebel aus einer Nebelmaschine),
3. das Zusammensetzen eines Objekte aus vorhandenen Teilen (z.B. Bau eines Autos) oder
4. die Auswahl eines Objektes aus einem Reservoir vorhandener Objekte (z.B. Würfeln einer Zufallszahl).

Spontane Entstehung aus dem Nichts kommt in der materiellen Realwelt – zumindest bei den Vorgängen, die wir in unserem Alltag erleben – nicht wirklich vor. Das verbieten die Erhaltungssätze der Naturwissenschaften (Energieerhaltungsgesetz, Massenerhaltungsgesetz). In der Gedankenwelt der Informatik ist dagegen spontane Entstehung ein zulässiges Konzept.

Bei einer Auswahl wird eine bereits existierende Sache in einen neuen Zusammenhang gestellt und mit neuen Qualitäten verbunden. Wer bei einer Multiple-Choice-Aufgabe eine Möglichkeit ankreuzt, hat ebenso eine Lösung hergestellt (obwohl der Antworttext vorher schon existierte), wie jemand, mit freiem Text antwortet. Ein Gegenstand, der in einem Geschäft ausgewählt wurde, kann durch diesen Akt zu einem Geburtstagsgeschenk (also einer neuen Entität) werden.

5.2 Vernichtungskonzepte im Alltag

Im Alltag gibt es eine Reihe von Vorgängen, bei denen Information vernichtet wird.

(1) *Als ungültig markieren.* In einem Text kann man durch Einklammern oder Durchstreichen Passagen vernichten. Ein Pass kann durch einen entsprechenden Stempelaufdruck für ungültig erklärt werden. Die Information bleibt aber immer noch lesbar.

(2) *Vollständige Zerstörung ohne Möglichkeit der Rekonstruktion.* Wenn man ein Schriftstück verbrennt, ist die gespeicherte Information endgültig verloren (sofern keine Kopien existieren).

(3) *Verlust.* Daten werden von ihrem angestammten Platz an einen unbekannten Ort verschoben (z.B. Notizzettel aus der Brieftasche herausnehmen und wegwerfen).

(4) *Lösen von Bezügen.* Dazu zählt die Zerlegung eines informativen Aggregats in seine Teile (z.B. Blätter eines Manuskriptes durcheinander bringen) als auch die Trennung eines Namens von der referierten Informationseinheit (z.B. Herausfallen eines Lesezeichens aus einem Buch).

(5) *Vergessen.* Vergessen ist die Vernichtung von Information ohne besondere – mit der jeweiligen Information zusammenhängende – Absicht. Information wird vergessen, wenn sie lange nicht mehr verwendet worden ist und der durch sie beanspruchte Speicherplatz für anderes benötigt wird.

(6) *Entwerten.* Durch einfaches Lesen kann ein Geheimnis vernichtet werden. Eine geheime PIN-Nummer ist nicht mehr geheim und verliert ihren Wert, wenn sie jemand anderes liest.

Was das menschliche Gedächtnis betrifft, sind die Vernichtungsarten 1 und 5 relevant, nicht aber 2, 3 und 4. Wir können gespeicherte Information als falsch oder ungültig anotieren oder unwichtige Dinge mit der Zeit vergessen. Es ist uns aber nicht möglich, einmal gespeicherte Inhalte gezielt zu zerstören oder zu entfernen.

Für die Interpretation von Programmtexten sind vor allem die Vernichtungskonzepte 1 bis 4 von Bedeutung. Das fünfte Konzept (Vergessen) kann zur Beschreibung einer Garbage Collection verwendet werden, wenn das Laufzeitsystem ein Objekt löscht (d.h. den für es reservierten Speicherplatz frei gibt), für das keine Referenz mehr existiert und das deshalb nicht mehr benötigt wird. Doch dieser Mechanismus wird (üblicherweise) nicht vom Programmtext kontrolliert. Das sechste Vernichtungskonzept (Entwerten) bezieht sich auf den pragmatischen Wert der Information, die durch Daten repräsentiert wird, aber nicht auf die Daten selbst.

5.3 Totale Vernichtung bei Zuweisungen

Abb. 101 zeigt Screenshots aus einer Animation (Python Quiz assign Aufgabe 2), die sukzessive Zuweisungen mit dem Notizzettelmodell folgendermaßen visualisiert: Vor einer erneuten Zuweisung wird nicht allein der vorige Inhalt der Variablen vernichtet sondern ebenfalls ihr Name. D.h. der gesamte Notizzettel verschwindet. Dann kommt ein neuer Zettel mit der Überschrift today (Variablenname) ins Bild, der dann mit dem neuen Wert beschrieben wird. Die meisten der in PVS-Workshops beobachteten Schülerinnen und Schüler (96 von 154) akzeptierten dieses Modell. Dieses Ergebnis zeigt, dass Anfänger häufig nicht deutlich zwischen dem Namen und dem Wert einer Variablen differenzieren, sondern beides als eine Einheit sehen.

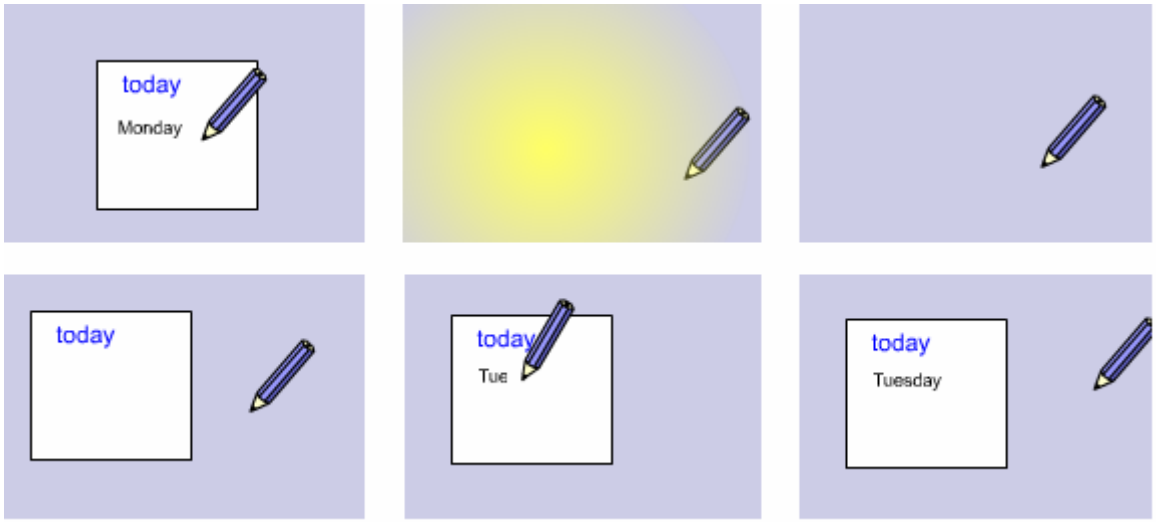


Abb. 101: Zuweisung mit totaler Vernichtung der Variablen

n = 154	**Dauer der Animation**	**Entscheidungszeit (Stdabw.)**	**Als passend beurteilt von**	**Konfidenz (Stdabw.)**
Notizzettel (pq_assign_a2_7)	10 s	11.79 s (10.68)	96 (62.3%)	84.4% (29.3)

Tab. 46: Beurteilung eines Zuweisungsmodells mit totaler Vernichtung der Variablen. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 154 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

Das Konzept der totalen Vernichtung ist sogar relativ hartnäckig, wie die Lernkurve und Antilernkurve in Abb. 102 zeigen. Obwohl es bei dem Python Quiz Minuspunkte gab, wenn man dieses Modell als passend wertete, beurteilten es 29 % der Vielspieler (41 Schülerinnen und Schüler, die mindes-

tens drei Mal gespielt haben) noch im dritten Durchgang positiv bei einer mittleren Konfidenz von 81%.

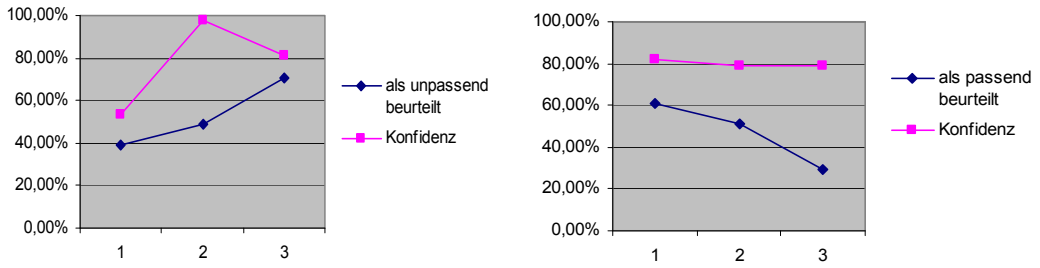


Abb. 102: Beurteilung eines Zuweisungsmodells mit totaler Vernichtung. Lernkurve (links) und Antilernkurve (rechts) bei Vielspielern (n = 41)

Dieses Ergebnis zeigt, dass Anfänger häufig nicht deutlich zwischen dem Namen und dem Wert einer Variablen differenzieren, sondern beides als eine Einheit sehen. Es könnte sich auch um eine spezifische Schwäche des Notizzettelmodells handeln. Im Alltag verwendet man Notizzettel häufig nur einmal und wirft sie weg, wenn die gespeicherte Information überholt ist. Leider wurde in der PVS nicht geprüft, ob die Vorstellung einer totalen Vernichtung beim Behältermodell oder Zeigermodell seltener auftritt.

Eine zweite Erklärung ist eine übertriebene Betonung des Zerstörungsaspektes bei sukzessiven Zuweisungen. Dies könnte man als Halo-Effekt bezeichnen, den man auch in anderen Zusammenhängen beobachten kann. Der Vernichtungsgedanke „strahlt aus“ auf andere eigentlich nicht betroffene Bereiche.

5.4 Beispiele für Datenumwandlungen

Abb. 103 zeigt eine Visualisierung der Anweisung (Python)

```
result = result + 4
```

Abb. 103: „Verschmelzen“ von zwei Datenentitäten. Drei Screenshots aus einer Animation der PVS (Python Puzzle Assert First Steps)

Hier bewegt sich ein Zettel mit der Zahl 4 (Datenentität) auf den mit *result* etikettierten Zettel zu. Bei der Berührung ereignet sich die Umwandlung (visualisiert durch einen Blitz, der mit einem Plus-Zeichen beschriftet ist). Die Entitäten vereinigen sich – ähnlich wie bei einer chemischen Reaktion – zu einer einzigen Entität mit der Summe. Sie leben gewissermaßen als Teil der Summe weiter. Offensichtlicher ist diese „Kontinuität der Existenz“ noch bei Visualisierungen, die die Konkatenation von Sequenzen (Listen, Zeichenketten) darstellen (Abb. 104).

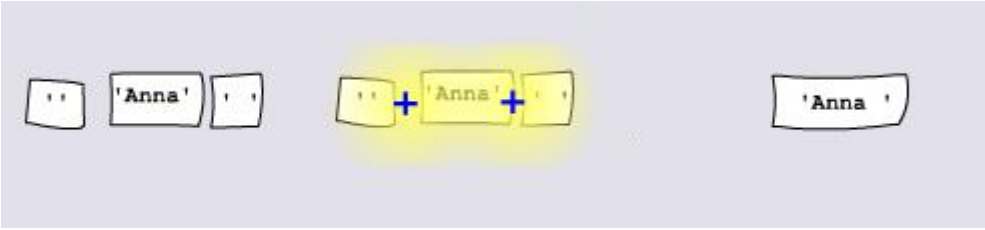


Abb. 104: Konkatenation von Listen. Drei Screenshots aus einer Animation der PVS (Python Puzzle Multilists)

Zerlegungen von Sequenzen (z.B. Listen oder Zeichenketten) stellt man sich häufig so vor, dass ein Materialstück – z.B. ein Papierstreifen – zerschnitten wird (siehe Abb. 105). Begriffe aus dem Fachjargon oder sogar Funktions- und Methodenbezeichner aus der Programmiersprachensyntax unterstützen diese Vorstellung (z.B. slicing für das Herauslösen eines Stücks aus einer Sequenz). Die Besonderheit gegenüber dem Entfernen von Elementen aus einer Sequenz ist, dass hier die Erhaltung der Identität aufgegeben wird. Aus einer Entität entstehen mehrere neue Entitäten mit eigenen Identitäten. Dennoch gibt es auch hier eine Kontinuität der Existenz, die man als „Erhaltung der Masse“ beschreiben könnte. Bei Zerlegungen geht nichts verloren. Die ursprüngliche Entität existiert in gewissem Sinne in ihren Teilen weiter. Somit kann man Zerlegungen guten Gewissens zu den Veränderungsprozessen zählen.

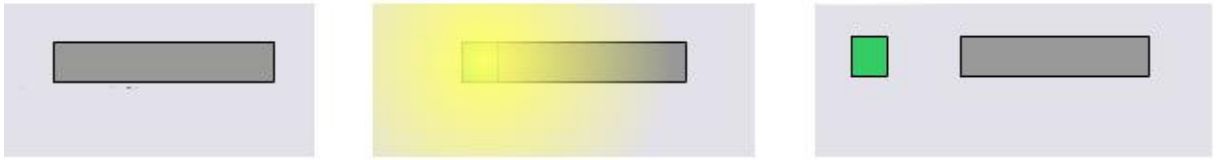


Abb. 105: Abstrakte Darstellung der Abspaltung eines Pivot-Elementes aus einer unsortierten Liste im Rahmen des Quicksort-Algorithmus. Drei Screenshots aus einer Animation der PVS (Python Puzzle „Assert Quicksort“)

Abb. 106 zeigt eine Visualisierung, in der die Berechnung der Wurzel durch die statische Java-Methode `Math.sqrt()` als Zerlegung dargestellt wird. Der Schüler erklärte, er stelle sich die Methode als Mechanik (symbolisiert durch die Zahnräder) vor, die die Zahl `2` auseinander nimmt und zwei gleiche Zahlen erzeugt, deren Produkt `2` ergibt.

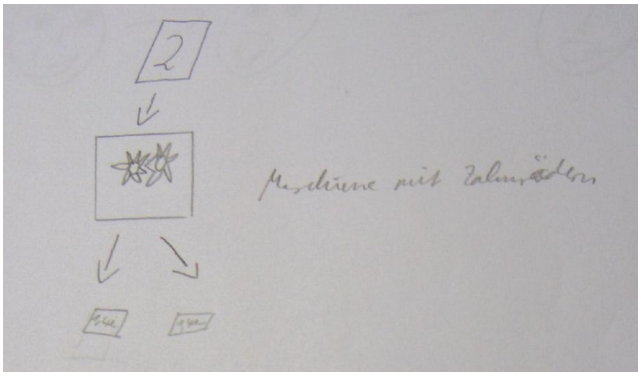


Abb. 106: Illustration des 17-jährigen Schülers M. zur Visualisierung der statischen Methode (Java) `Math.sqrt(2)`

Programmiersprachen (z.B. C, Java, Python) enthalten Konstrukte zur so genannten Typumwandlung (type conversion, cast). Bei Java und C z.B. wird die Typumwandlungsoperation folgendermaßen gebildet: Man schreibt den gewünschten Typ in Klammern – z.B. `(byte)` – und setzt dahinter das umzuwandelnde Objekt. Im folgenden Beispiel (Java) wird ein Objekt vom Typ `int` in ein Objekt vom Typ `float` „umgewandelt“:

```
int i = 1;
float f = (float) i;
```

Der Begriff Typumwandlung ist insofern missverständlich als nicht der Typ modifiziert wird. Ein Typ ist die Außenansicht einer Klasse (Balzert, 1999, S.23). Wörtlich genommen würde die Umwandlung eines Typs bedeuten, dass eine Klasse (etwa die Klasse der ganzen Zahlen) verändert wird. In der Regel werden in Lehrbüchern Typumwandlungen sinngemäß so wie in der Wikipedia erklärt:

„Typumwandlung (engl. type conversion oder cast) bezeichnet in der Informatik die Umwandlung eines Wertes eines Datentyps in einen Wert eines anderen Datentyps.“[27]

Es wird hier also das intuitive Modell einer Datenveränderung verwendet. Programmtechnisch findet aber bei einem Casting keine Veränderung statt, sondern es wird ein neues Objekt einer anderen Klasse generiert und das ursprüngliche Objekt bleibt erhalten.

Die Kontinuität der Existenz, die für Veränderungsprozesse charakteristisch ist, kommt dadurch zu Stande, dass das neue Objekt das gleiche Wissen repräsentiert wie das gegebene Objekt. Im obigen Beispiel repräsentiert das Objekt mit dem Namen `i` vom Typ `int` die Zahl `1`, also ein Stück mathematischen Wissens. Das Objekt mit dem Namen f vom Typ float repräsentiert ebenfalls die Zahl 1. Bei duck-typisierenden Sprachen wie Python wird der neue Wert auch durch ein anderes Literal dargestellt. Das float-Objekt für das mathematische Objekt `1` kann z.B. durch das Literal `1.0` beschrieben werden (an Stelle von `1` für das `int`-Objekt).

Das Motiv für ein explizites Casting ist pragmatischer Natur. Man möchte auf das Wissen-Objekt, das durch das Datum repräsentiert wird, Operationen anwenden, für die der ursprüngliche Datentyp nicht geeignet war (in unserem Beispiel etwa Divisionen oder Wurzelberechnungen).

5.5 Umbenennungen bei der der Ausführung von Funktionen

Den Prozess, der beim Aufruf einer Funktion entsteht, kann man als besonderen Bereich darstellen, der von der Außenwelt abgetrennt ist. Man kann sich vorstellen, dass eine Daten repräsentierende Entität, die einen solchen Bereich betritt, einen neuen Namen annimmt. Er ist in der formalen Parameterliste spezifiziert und bringt (bei gutem Programmierstil) die Rolle des Objektes im neuen Kontext zum Ausdruck.

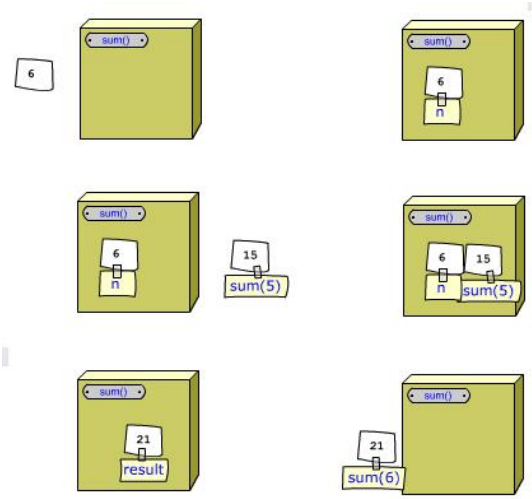


Abb. 107: Umbenennungen bei der Ausführung einer Funktion. Sechs Screenshots aus einer Animation der PVS (Python Puzzle Assert First Steps)

[27] Wikipedia, Stichwort „Typumwandlung“ www.wikipedia.de Zugriff am 1.9.2006

Abb. 107 zeigt einige Screenshots aus einer Animation, die die Ausführung der folgenden rekursiven Funktion veranschaulicht (Python) und Umbenennungen an den Grenzen des Execution Frames enthält :

```
def sum(n):
   if n = 0: return 0
   else:
      result = n + sum(n-1)
      return result
```

Aufgerufen wird `sum(6)`.

Der Ablauf ist folgender: Als erstes entsteht ein Brett mit der Aufschrift `sum()`, das einen Execution Frame der Funktion `sum()` repräsentiert. Ein Zettel mit der Aufschrift 6 (Datenentität ohne Namen) fliegt ins Bild (erstes Bild). Sobald er in den Bereich des Execution Frames eintritt (zweites Bild), erhält er das Label n (Name des formalen Parameters). Beim rekursiven Aufruf verlässt ein Zettel mit der Zahl 5 den Execution Frame und bewegt sich zu einem neuen Execution Frame der Funktion sum() (nicht im Bild). Der dritte Screenshot zeigt, wie ein Zettel mit dem Ergebnis des rekursiven Aufrufs – nämlich die Zahl 15 – in den Execution Frame zurückkehrt. Er trägt ein Etikett mit der Aufschrift sum(5), also der Bezeichnung des Funktionsaufrufs. Wie in Abschnitt 6.7.1 erläutert, kann man Funktionsaufrufe als Namen für Daten auffassen. Bei Berührung der beiden Datenentitäten (viertes Bild) werden die Werte addiert und es entsteht ein neuer Zettel mit der Zahl 21, der entsprechend dem Programmtext mit result etikettiert wird (fünftes Bild). Aus den beiden Namen `n` und `sum(5)` entsteht in dieser Animation also ein neuer Name. Dieses Objekt wird zurückgegeben und erhält beim Verlassen des Execution Frames (sechstes Bild) den neuen Namen `sum(6)`.

Der Name `result` macht nur innerhalb des Execution Frames der Funktion Sinn. Außerhalb ist er unbekannt.

5.6 Umbenennungen in Rechenprotokollen zur rekursiven Berechnung der Fakultät

Das folgende Python Skript definiert eine rekursive Funktion zur Berechnung der Fakultät:

```
def fak(n):
   if n <= 1: return 1
   else: return n * fak(n-1)
```

In der PVS-Applikation Python Visual Factorial wurden verschiedene Animationen, die die Arbeitsweise der Funktion `fak()` veranschaulichen sollten, einander gegenüber gestellt. Zwei dieser Animationen verwendeten das Modell der Execution Frames (siehe Abb. 108). In der ersten Variante wird die Ausführung vollständig und in der zweiten Variante unvollständig bis zum ersten rekursiven Aufruf dargestellt.

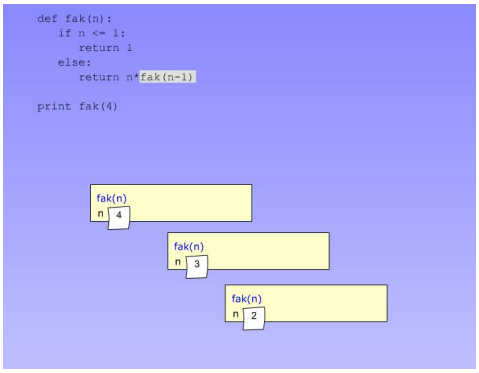


Abb. 108: Veranschaulichung der Arbeitsweise einer rekursiven Funktion zur Berechnung der Fakultät unter Verwendung von Execution Frames. Screenshot aus der PVS (Python Visual Factorial)

Zwei weitere Animationen verwenden ein Rechenprotokoll, bei dem Terme umgeformt werden. Wiederum gibt die eine Variante die Ausführung in voller Tiefe bis zum Erreichen des Elementarfalls wieder, während die andere nach dem ersten rekursiven Aufruf tiefer liegende Aufrufe überspringt. Abb. 109 zeigt drei Screenshots aus einer dieser Animationen. Der Teil des Terms, der im nächsten Schritt durch einen anderen ersetzt wird, ist durch einen hellen Kasten markiert.

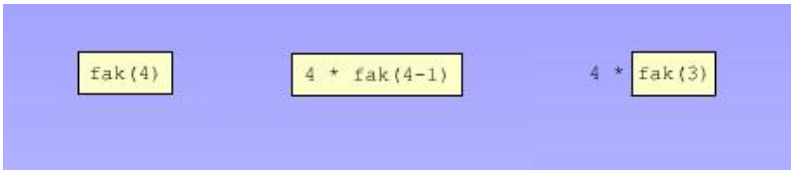


Abb. 109: Veranschaulichung der Arbeitsweise einer rekursiven Funktion durch sukzessive Umbenennung. Drei Screenshots aus einer Animation der PVS (Python Visual Factorial)

Die Spieler sollten am Ende einer Sitzung folgende Fragen beantworten:

- Welche Animation würden Sie verwenden, um jemandem zu erklären wie das Python-Skript funktioniert?
- Welche Animation gibt die Idee einer rekursiven Funktion am besten wieder?
- Welche Animation war am schwierigsten nachzuvollziehen?

Tab. 47 zeigt das Ergebnis der Sessions von 31 Teilnehmern verschiedener Workshops mit der PVS. Die Angaben zur ersten und letzten Frage unterscheiden sich signifikant von einer Gleichverteilung (Chi-Quadrat-Test, $p = 0.02$ bzw. $p = 0.04$). 12 der 31 Teilnehmer gaben an, dass sie die Animation mit einer vollständigen sukzessiven Termumformung verwenden würden, um jemandem die Arbeitsweise des Programms zu erklären. Nur sechs hielten dieses Modell am für am schwierigsten nachvollziehbar.

n = 31	**Execution frames (vollständig)**	**Execution frames (unvollständig)**	**Rechenprotokoll (vollständig)**	**Rechenprotokoll (unvollständig)**
Erklären	12	5	12	2
Idee der Rekursion	10	7	11	3
Schwierig nachvollziehbar	2	11	6	12

Tab. 47: Wahl verschiedener Modelle zur Veranschaulichung der Arbeitsweise einer rekursiven Funktion (Fakultät). Ergebnisse von 31 ersten Sessions in Workshops mit der PVS. Teilnehmer: 28 Schüler und 3 Studenten aus Deutschland, 6 weiblich und 25 männlich, sie beschäftigten sich im Schnitt 2.26 Stunden pro Woche mit Programmierung und hatten ein Durchschnittsalter von 17.65 Jahren.

5.7 Datenbewegung bei Iterationen

Die PVS-Applikation Python Quiz List enthält einige Animationen, in denen folgende Programmzeile (Python) aus einer Iteration visualisiert wird:

```
for (n, a) in persons:
```

Dabei ist `persons` eine Liste von Tupeln der Form `(Name, Alter)`, die eine Personengruppe modelliert. Abb. 110 zeigt Screenshots aus Animationen, die das Durchlaufen der Liste (dargestellt als Behälter mit drei Fächern) durch Zuweisungen modelliert. Die Modelle unterscheiden sich im Grad der Naivität, mit der Materialbewegungen auf die Darstellung von Zuweisungen angewendet werden:

(1) Im naiven Modell (pq_list_a2_6) werden keine Kopien angefertigt und beim Überschreiben eines Variableninhalts der vorige Inhalt nicht gelöscht. Aus dem Listenbehälter persons werden Karten entnommen und in Behälter, die mit n und a etikettiert sind, gesteckt. Das aktuelle Element befindet sich immer vorne und ist somit eindeutig erkennbar. Ein solcher Ablauf kann mit realen Gegenständen leicht nachgespielt werden.

(2) Im zweiten Modell (pq_list_a2_2) werden Elemente aus dem Listenbehälter herausgenommen und bewegen sich in die Behälter n und a, nachdem der vorige Inhalt mit einem Blitz verschwindet.

(3) Das dritte Modell (pq_list_a2_4) gibt Zuweisungen am realistischsten wieder, ist aber am weitesten von der physischen Situation entfernt. Hier werden Kopien der Items aus der Liste angefertigt und in die Behälter n und a bewegt, deren voriger Inhalt zerstört wird.

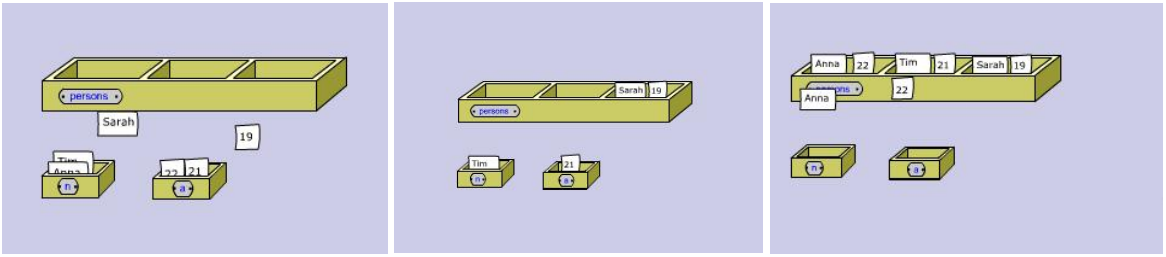


Abb. 110: Screenshots aus Animationen der PVS-Applikation Python Quiz List: (1) Naive Bewegung, (2) Bewegung ohne Kopien, (3) Bewegung von Kopien

Tab. 48 zeigt die Ergebnisse der Beurteilung von 68 Schülerinnen und Schülern (12 w, 56 m, Durchschnittsalter 17.15 Jahre). Die Mehrheit akzeptiert alle drei Modelle als geeignet, wobei das dritte Modell gegenüber den anderen signifikant bevorzugt wird.

n = 68	**Dauer der Animation**	**Entscheiungszeit (Stdabw.)**	**Als passend beurteilt von**	**Konfidenz (Stdabw.)**
Naive Bewegung (pq_list_a2_6)	10 s	9.78 s (5.97)	35 (51.7%)	75.7% (38.1%)
Bewegung ohne Kopien (pq_list_a2_2)	10 s	8.81 s (6.22)	40 (58.8%)	77.2% (38.1%)
Bewegung von Kopien (pq_list_a2_1)	10 s	11.18 s (10.50)	49 (72.06%)	73.5% (39.7%)

Tab. 48: Beurteilung von Modellen zur Veranschaulichung von Zuweisungen im Rahmen einer Iteration. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 68 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

Die beiden ersten Modelle können als Fehlvorstellung interpretiert werden, weil im zugehörigen Programm die Liste nicht verändert wird und in den Variablen a und n keine Daten gesammelt werden. Es kann aber auch sein, dass einem Befürworter dieser Modelle klar ist, welche Vereinfachungen bzw. Abweichungen in ihnen stecken und dass sie gewissermaßen größeren Abstand zum Programmtext haben. Die drei Modelle weisen in der oben dargestellten Reihenfolge wachsende strukturelle

Komplexität (und damit sinkenden Gestaltcharakter) auf. Das Modell der naiven Bewegung ist strukturell am einfachsten und insofern das intuitivste. Es kann denkökonomisch sein, ein solches Modell in Kombination mit einschränkenden Vorstellungen (z.B. „In Wirklichkeit verändert sich die Liste nicht“) zu verwenden.

5.8 Namenbewegung bei Iterationen

In einigen Animationen der PVS-Applikation Python Quiz List wird die Programmzeile

```
for (n, a) in persons:
```

durch Namenbewegungen veranschaulicht (siehe Abb. 111). Dabei werden Namen durch bewegliche Etiketten (z.B. Klebezettel) oder Zeiger repräsentiert.

(1) In der ersten Animation wird konsequent das Zeigermodell für Namen verwendet. Zeiger mit den Namen n und a markieren die beiden Items des aktuellen Tupels, das selbst ein Element der Liste persons ist. Die Zeiger bewegen sich von Tupel zu Tupel.

(2) Das zweite Modell stellt die Liste durch eine Variante eines Behältermodells dar (Brett mit grauen Bereichen für die Elemente). Die Namen n und a werden durch Klebezettel repräsentiert, die sich von Wert zu Wert bewegen.

(3) Im dritten Modell sind ein Behältermodell für die Liste und Zeigermodelle für die Laufvariablen n und a kombiniert.

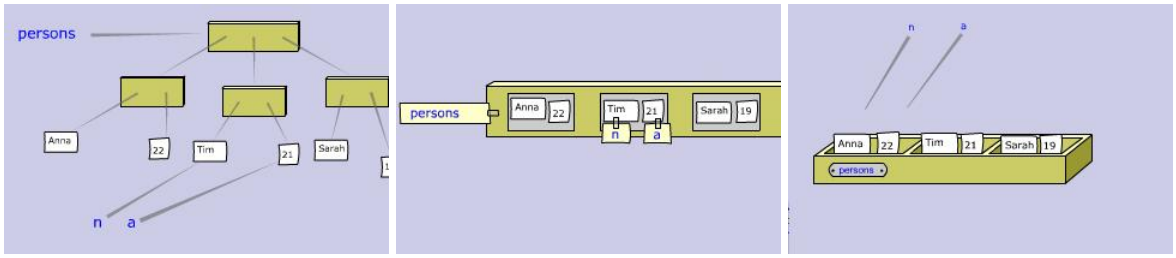


Abb. 111: Modelle mit Namenbewegung zur Visualisierung einer Iteration: Konsequentes Zeigermodell (pq_assign_a1_9) Transport der Kopie eines Namens (pq_assign_a1_6 und Zeiger über Behälter (pq_assign_a1_8)

Die Ergebnisse in Tab. 49 zeigen, dass alle drei Modelle etwa von vier Fünftel der Schüler als passend gewählt wurden bei etwa gleicher Konfidenz[28]. Es konnte keine Überlegenheit des einen oder anderen Modells festgestellt werden.

n = 68	**Dauer der Animation**	**Entscheidungs-zeit (Stdabw.)**	**Als passend beurteilt von**	**Konfidenz (Stdabw.)**
Zeiger (pq_list_a2_3)	4 s	19.4 s (65.9)	57 (83.8%)	75.0 % (39.1%)
Zettel und Behälter (pq_assign_a1_7)	9 s	12.40 s (17.74)	54 (79.4%)	73.5% (37.1%)
Zeiger und Behälter (pq_assign_a1_8)	7 s	10.20 s (3.71)	54 (79.4%)	73.5% (37.1%)

Tab. 49: Beurteilung von Modellen zur Veranschaulichung von Iterationen durch Namenbewegungen. Berücksichtigt wurden die Antworten aus ersten Spieldurchgängen von 68 Schülerinnen und Schülern, die an Workshops mit der PVS teilgenommen haben.

[28] Aufgrund der hohen Standardabweichungen, die auf einige besonders lange Entscheidungszeiten hinweisen, lassen sich die mittleren Entscheidungszeiten nicht vergleichen.

6 Ergänzungen zu intuitiven Modellen in der OOP

6.1 Klasse und Schema

Das intuitive Modell des Bauplans korrespondiert mit dem Begriff des Schemas in der Kognitionspsychologie. Nach Anderson (1996, S. 150) ist der psychologische Begriff des Schemas an das Konzept der Datenstrukturen angelehnt – stammt also eigentlich aus der Informatik. Wie eine Klasse beschreibt auch ein Schema einen Typ von Objekten durch ein Set von Attributen (Slots), denen typische Werte als Default zugeordnet sein können. Außerdem gibt es Attribute, die die Verbindung zu anderen Schemata herstellen (z.B. Oberbegriff). Schemata werden verwendet, um Objekte der Realwelt zu identifizieren (Assimilation, Piaget) oder zu konstruieren (Zeichnen, Bauen mit Bauklötzen etc.). Kinderzeichnungen aus verschiedenen Altersstufen illustrieren die zunehmende Verfeinerung der verwendeten Schemata. Die psychische Realität von Schemata konnte in Erinnerungsexperimenten von Brewer und Treyens (1981) nachgewiesen werden. Testpersonen sollten sich an Details der Einrichtung eines Büros erinnern. Es stellte sich heraus, dass sie sich besonders gut an Gegenstände erinnern konnten, die typischerweise in einem Büro zu finden sind, die also zum Schema eines Büros gehören.

6.2 Visualisierung von Klassen in Schülerzeichnungen

Abb. 112 zeigt links eine Schülerzeichnung, in der die Java-Klasse Tier (Skript in Anhang 3.1) durch ein Fabrikgebäude visualisiert wird. Eine Instanzierung eines Objektes wird durch einen Auftrag ausgelöst, der variable Attributwerte enthält. Die zweite Abbildung enthält die Fabrik-Metapher in abstrakterer Form, als Produzent beliebiger Objekte des Typs ohne expliziten Auftrag.

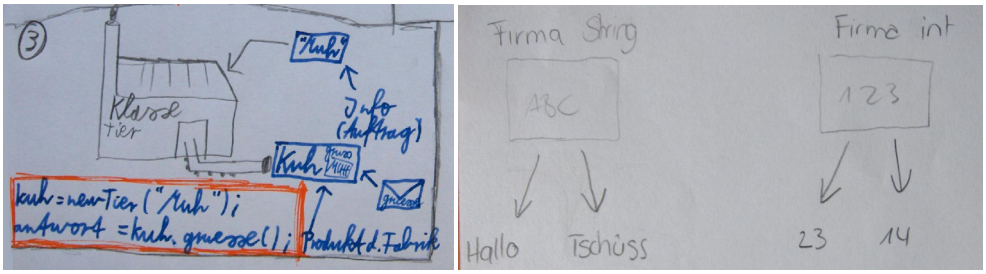


Abb. 112: Schüler-Visualisierungen mit Fabrik-Metaphern

Abb. 113 zeigt eine Schülerzeichnung, die den Begriff Klasse wörtlich nimmt und ihn als Klassenraum mit Tischen visualisiert. Jedes Element der Klasse ist ein Tisch in einem Klassenraum.

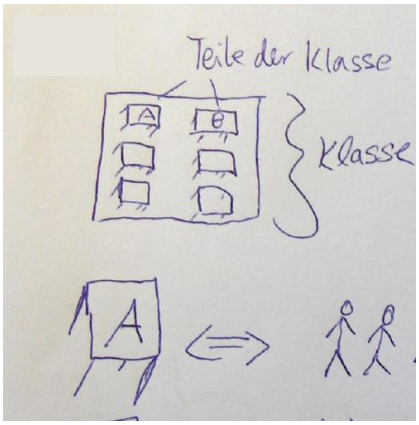


Abb. 113: Modell einer Klasse als Menge von Objekten

Abb. 114 zeigt die Visualisierung einer Klassendefinition, bei der die Prototyp-Metapher verwendet wurde. Die Klasse `Tier` wird durch ein konkretes Exemplar – eine Kuh – repräsentiert (die ersten drei Bilder in der Klammer). Die Definition der Methode `gruesse()` wird als eine Art Schulung des zunächst „dummen“ Prototypen dargestellt. Die Kuh lernt, auf die Botschaft mit einem Gruß zu ant-

worten. Der Text des Grußes ist zunächst unbestimmt und wird erst bei der Instanzierung eines Objektes (letztes Bild) festgelegt.

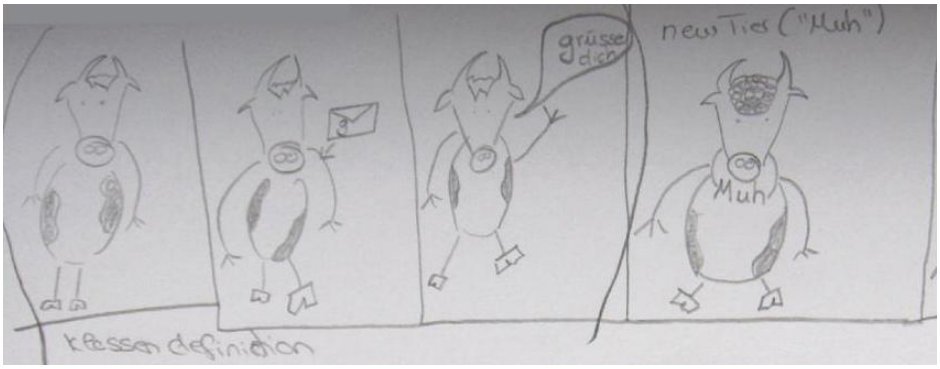


Abb. 114: Illustration einer Klassendefinition und Instanzierung eines Objektes (17-jährige Schülerin J.)

6.3 Klasse als Entität

Die Fabrikmetapher und die Toolbox-Metapher betrachten eine Klasse (wie ein Objekt) als eigenständige Entität in der Maschinerie eines laufenden Programms. Dieses Bild wird durch einige Features von Programmiersprachen unterstützt:

- Introspektive Klassenattribute enthalten Information über eine Klasse wie z.B. Basisklassen, von denen die Klasse abgeleitet ist (bei Python: `__bases__`), oder die verwendete method resolution order d.h. die Reihenfolge, nach der in Oberklassen nach Methoden gesucht wird (bei Python: `__mro__`). Hier wird nicht ein Mengenkonzept verwendet (Klassenattribute als gemeinsame Eigenschaften aller Objekte) sondern die Klasse selbst als Entität betrachtet.
- Eine statische Methode kann aufgerufen werden, indem man eine Botschaft an die Klasse und nicht an ein Objekt der Klasse sendet.

6.4 Prototyptheorien in der Kognitionspsychologie

In der kognitiven Psychologie unterscheidet man im Zusammenhang mit der gedanklichen Repräsentation von Begriffen „Exemplartheorien“ und „Abstraktionstheorien“ (Anderson 1996, S. 160f.). Abstraktionstheorien orientieren sich am aristotelischen Kategorienkonzept, nach dem Objekte auf der Basis gemeinsamer Merkmale zusammengefasst werden. Zum Beispiel gehören Lebewesen, die zwei Beine, Flügel und einen Schnabel besitzen zur Kategorie Vögel. Aristoteles ging von der Annahme einer einzigen korrekten Taxonomie der Dinge aus. Er glaubte, es existierten universelle Regeln, nach denen die Welt geordnet ist, und die der Philosoph nur noch zu entdecken braucht.

Dies wurde erstmals im 19. Jahrhundert von den britischen Philosophen Whewell und Jevons angezweifelt. Sie betonten, dass Klassifizierung kein mechanischer Prozess ist, der sich an universellen Regeln orientiert, sondern Kreativität erfordert (Taivalsaari 1997)

In seinen berühmten Philosophischen Untersuchungen (im Jahre 1953 erstmals veröffentlicht) nannte Ludwig Wittgenstein Beispiele für Phänomene, die man zwar durch einen Begriff aber nur schwierig oder gar nicht durch gemeinsame Eigenschaften charakterisieren kann. Ein viel zitiertes Beispiel ist der Begriff „Spiel“. Höchst unterschiedliche Aktivitäten (die teilweise nichts Gemeinsames haben) werden als Spiel bezeichnet. Wittgenstein definierte den Begriff Familienähnlichkeit. Danach wird die Zugehörigkeit zu einer Kategorie nicht durch gemeinsame Eigenschaften, sondern durch wechselseitige Ähnlichkeit bestimmt (Wittgenstein 1982).

Mit Eleonor Rosch (1973, 1975) hielt diese Sichtweise Einzug in die kognitive Psychologie. Sie ließ Personen die Typikalität verschiedener Exemplare einer Kategorie durch Werte zwischen 1 (sehr typisch) und 7 (sehr untypisch) einschätzen. Beispielsweise für die Kategorie Gemüse erhielt Möhre einen Wert von 1.1, wurde also als sehr typisch angesehen, während Petersilie nur mit 3,8 bewertet wurde (Rosch 1973). In den in der Folgezeit aufkommenden Prototyp- oder Exemplartheorien (z.B. Lakoff 1987) wird angenommen, dass Menschen sich zu einem abstrakten Schema ein typisches Ex-

emplar als Prototyp merken. Wenn beurteilt werden soll, ob ein Objekt zu dieser Kategorie gehört, wird es mit diesem Prototyp verglichen und bei genügend großer Ähnlichkeit der Kategorie zugeordnet.

6.5 Prototyporientierte Programmiersprachen

Im Hinblick auf die Eignung als Werkzeug zur Modellierung der Welt hat die OOP gewisse Schwächen (Taivalsaari 1997, Lakoff 1987):

- Manche Konzepte und Phänomene lassen sich nicht auf der Basis gemeinsamer Eigenschaften modellieren.
- Es gibt in der Begriffsbildung graduelle Zugehörigkeiten zu Klassen und unscharfe Grenzen zwischen Klassen.
- Kategorien sind nicht in einfachen Taxonomien aus Ober- und Unterbegriffen organisiert. Entsprechend gibt es keine optimalen Klassenhierarchien. Stattdessen ist in Programmierprojekten jede Klassenstruktur das Ergebnis eines sozialen Einigungsprozesses („consensus-driven design", Taivalsaari 1997). Zudem müssen bei konkreter Software neben logischen Aspekten immer auch technische Gesichtspunkte einbezogen werden. Eine konzeptionell besonders „natürlich" erscheinende Struktur wird manchmal abgelehnt, weil sie zu einem System mit geringer Laufzeit- und Speicherplatzeffizienz führt.

Vor dem Hintergrund dieser Schwächen der klassischen OOP und der Erkenntnisse der kognitiven Prototyptheorien sind prototyporientierte Programmiersprachen entstanden, z.B. Self (Smith & Ungar 1995) und Kevo (Taivalsaari 1992). Sie verwenden keine Klassen, von denen Instanzen gebildet werden (class-less programming). Stattdessen können neue Objekte durch Klonen aus vorhandenen Objekten gebildet werden. Bei Self können Eigenschaften eines Prototypen (beliebiges anderes Objekt) geerbt werden, indem man einem Objekt `O` eine Referenz zu einem Elternobjekt `P` zuordnet. Wenn in einer Botschaft an `O` z.B. eine Methode vorkommt, die in der Spezifikation von `O` nicht auftaucht, sucht das System in den Spezifikationen von `P` und gegebenenfalls bei dessen Eltern weiter.

Ein Feature prototyporientierter Programmierung ist das „Life Editing" von Objekten (Smith & Ungar 1995). Das heißt, bei einem Objekt können zu seiner Lebenszeit Attribute und Methodendefinitionen geändert oder hinzugefügt werden.

6.6 Implizite Verwendung des Prototypkonzepts bei der Entwicklung einer Klasse

Auch auf der Ebene des „Programmierens im Kleinen" kann die Entwicklung einer Klasse mit einer objektorientierten Programmiersprache nahezu völlig prototyporientiert sein. Anfänger, die in der Syntax und Semantik einer Programmiersprache noch unsicher sind, folgen häufig einer experimentellen Strategie, die dem Test Driven Development (TDD) des Extreme Programming (Beck 2003) ähnelt. Mit Python, das diese Vorgehensweise unterstützt, geht das folgendermaßen. Als Beispiel nehmen wir an, dass eine Schülerin namens Sandra die Klasse Geld aus dem vorigen Abschnitt entwickeln möchte.

Als erstes schreibt sie ein Skript mit einer kleinen überschaubaren Klassendefinition und einigen Programmzeilen zum Testen:

```
class Geld:
    wechselkurs={'USD':0.84998,
                 'GBP':1.39480,
                 'EUR':1.0,
                 'JPY':0.007168}

    def __init__(self, waehrung, betrag):
        self.waehrung=waehrung
        self.betrag=float(betrag)
```

```
# Testen
if __name__ == "__main__":          #1
    g = Geld("USD", 100)            #2
    print g
    print g.waehrung
```

Die Bedingung `__name__ == "__main__"` (#1) ist dann erfüllt, wenn die Datei direkt gestartet wird. Nur in diesem Fall werden die Anweisungen im eingerückten Block (ab #2) ausgeführt. Wenn die Klasse – die später Teil eines größeren Projektes sein kann – von einem anderen Modul aus importiert wird, ist die Bedingung nicht erfüllt, und die Testanweisungen werden nicht ausgeführt.

Nach dem Edieren des Programmtextes startet Sandra die Datei und kontrolliert die Ausgabe der Testanweisungen im Shell-Fenster. In diesem Fall sieht sie:

```
<__main__.Geld instance at 0x00D2C148>
USD
```

Falls die Ausgabe vom erwarteten Ergebnis abweicht oder Laufzeitfehler auftreten, enthält die Klassendefinition logische Fehler. Der Programmtext wird abgeändert und erneut getestet, solange bis das Programm das gewünschte Ergebnis liefert. Im obigen Beispiel ist das Programm korrekt. Die erste Zeile der Ausgabe enthält eine interne Repräsentation des Objektes `g`.

Sandra beschließt, als nächstes dafür zu sorgen, dass mit der print-Anweisung eine lesbare Objekt-Repräsentation ausgegeben wird. Dazu ergänzt sie eine „magische Methode“ `__str__()` , die in einer print-Anweisung vom Laufzeitsystem aufgerufen wird:

```
class Geld:
    ...
    def __str__(self):
        return self.waehrung+' '+str(self.betrag)

# Testen
if __name__ == "__main__":
    ...

Die Ausgabe im Shell-Fenster lautet nun wie erwartet:
USD 100.0
USD
```

Auf diese Weise fährt Sandra fort und erweitert schrittweise ihr Programm, bis es die gewünschte Funktionalität hat. Das Entscheidende ist folgendes: Obwohl Sandra formal eine Klassendefinition schreibt, modelliert sie doch in Gedanken einen Prototypen. Alle kognitiven Aktivitäten drehen sich um ein einzelnes Objekt, das die Klasse repräsentiert. Dass es sich programmtechnisch um eine Klassendefinition handelt, spielt nur ganz am Anfang eine Rolle, wenn sie die Klausel class Geld: notiert. Beim eigentlichen Entwicklungsprozess hat sie immer ein konkretes Objekt vor Augen.

6.7 Das Prototyp-Konzept bei der Nutzung von Grafik-Tools

Das Prototyp-Konzept wird auch bei der Arbeit mit vektororientierten Grafikwerkzeugen angewendet. Nehmen wir an, Tom will eine Abbildung mit vielen Gesichtern anfertigen. Dann zeichnet er zunächst ein Gesicht und fügt die Elementarobjekte (Flächen und Linien) zu einer Gruppe zusammen. Er hat damit einen Prototyp definiert. Von diesem macht er viele Kopien und wandelt jede Kopie in Details ab. Es entstehen viele Gesicht-Objekte, die zwar Familienähnlichkeit im Sinne Wittgensteins besitzen, aber nicht Exemplare einer Klasse im aristotelischen Sinne mit gemeinsamen Eigenschaften sind.

Macromedia-Flash, ein System zur Definition visueller Applikationen, ist zwar klassenorientiert unterstützt aber auch und vor allem das Prototyp-Konzept. Ein Flash-Film kann aus verschiedenen grafischen Objekten zusammengesetzt werden, indem man sie aus einer Symbolbibliothek mit der Maus auf die Arbeitsfläche holt. Technisch ist jedes Symbol eine Klasse und das „Auf-die-

Arbeitsfläche-holen“ kein Kopieren sondern die Instanzierung eines Objektes der Klasse. Jede Instanz auf der Arbeitsfläche besitzt einen Instanznamen, der z.B. von Bedeutung ist, wenn man an das Objekt eine Botschaft schicken will. Eine nachträgliche Veränderung des Symbols (Klasse) wirkt sich auf alle Instanzen aus. Bestimmte Merkmale einer Instanz (Größe, Helligkeit, Position auf der Arbeitsfläche) können variieren und werden durch Attribute mit variablen Werten repräsentiert. Wenn ein Flash-Entwickler – nennen wir sie Sandra – ein Symbol (Klasse) gestaltet, hat sie auf der Arbeitsfläche ein konkretes Objekt – einen Prototypen – vor Augen.

6.8 Modelle für die Herstellung von Objekten

Instanzierung kann als Herstellung eines neuen Objektes aufgefasst werden, das vorher noch nicht existiert hat. Je nachdem welches Modell einer Klasse man zu Grunde legt, gibt es unterschiedliche Modellvarianten für den Herstellungsprozess. Die Fabrikmetapher impliziert, dass die Klasse ein Akteur ist, der für den Herstellungsprozess zuständig ist. Beim Aufruf des Konstruktors erhält die Klassenentität einen Auftrag, in dem Details des neuen Objektes spezifiziert sind. Sie führt den Auftrag aus und generiert das neue Objekt. Stellt man sich eine Klasse als Bauplan vor, geht bei der Instanzierung die Aktivität von einer übergeordneten Entität aus (Laufzeitsystem), die das neue Objekt mit Hilfe der Information aus dem Bauplan und der Argumente des Konstruktoraufrufs generiert. Eine Alltagsanalogie für diesen Vorgang ist der Bau eines Hauses mit Hilfe eines Bauplans und unter Berücksichtigung bestimmter Sonderwünsche (Fassadenfarbe, Fenstertyp, Haustür etc.), die nicht im Bauplan verzeichnet sind. Eine ähnliche Intuition ergibt sich aus der Verwendung des Prototyp-Modells, bei dem eine Klasse als unfertiges oder abwandelbares Objekt gesehen wird. Hier wird die Instanzierung zu seinem Zusammenbau verschiedener Teile zu einem neuen und vollständigen Objekt. Die Klasse liefert dabei sozusagen das Grundgerüst und die Konstruktorargumente die Teile, die von Exemplar zu Exemplar unterschiedlich sein können. Wie bei der Realisierung eines Plans ist der Akteur eine übergeordnete Entität. In dieser Sichtweise haben die Argumente eines Konstruktoraufrufs eine etwas andere Bedeutung als die Argumente eines normalen Methoden- oder Funktionsaufrufs. Methoden oder Funktionen verbindet man in der Regel mit der Verarbeitung von Daten, die als Argument übergeben werden. Bei der Instanzierung eines Objektes stellen die Argumente, die meist in Klammern hinter dem Klassennamen angegeben werden, Teile (technisch: Attributwerte) dar, die in der Klassendefinition nicht spezifiziert worden sind.

6.9 Modellierung verschachtelter Botschaften

In der letzten Aufgabe von Python Quiz „Objects“ werden Visualisierungen der Anweisung

`vase.fill(bottle.empty())` angeboten. Es geht also um die Frage, wie eine verschachtelte Botschaft (Botschaft, die eine andere Botschaft enthält) aufgelöst wird. Hier gerät die Botschaftsmetapher an ihre Grenze. Wir unterscheiden drei Gruppen intuitiver Modelle:

(1) Die Anweisung `vase.fill(bottle.empty())` wird als eine einzige Botschaft interpretiert. Dann muss es einen Adressaten geben, an den sie gesendet wird. In diesem Fall ist das das Objekt vase, da der Adressat immer zu Beginn der Botschaft steht. Nun müsste also das Objekt vase das Objekt bottle beauftragen, die Operation `empty()` auszuführen. In zwei Modellen wird diese Sichtweise verwendet.

In der ersten Variante (pq_objects_a5_3) erhält die Vase die komplexe Botschaft `vase.fill(bottle.empty())`, dargestellt durch ein fliegendes Oval mit entsprechender Beschriftung. Es entsteht eine neue Botschaft `bottle.empty()`, die sich zur Karaffe bewegt. Die Karaffe empfängt die Botschaft, bewegt sich und schüttet ihren Inhalt in die Vase (Abb. 115). Hier wird konsequent das Konzept eigenaktiver Objekte, die Botschaften verarbeiten, angewendet – freilich auf unangemessene Weise, da gemäß der Semantik des Programms das Objekt `vase` keine Botschaften verschickt. Von 23 beobachteten Personen hielten 12 dieses Modell für passend (mittlere Konfidenz 75%).

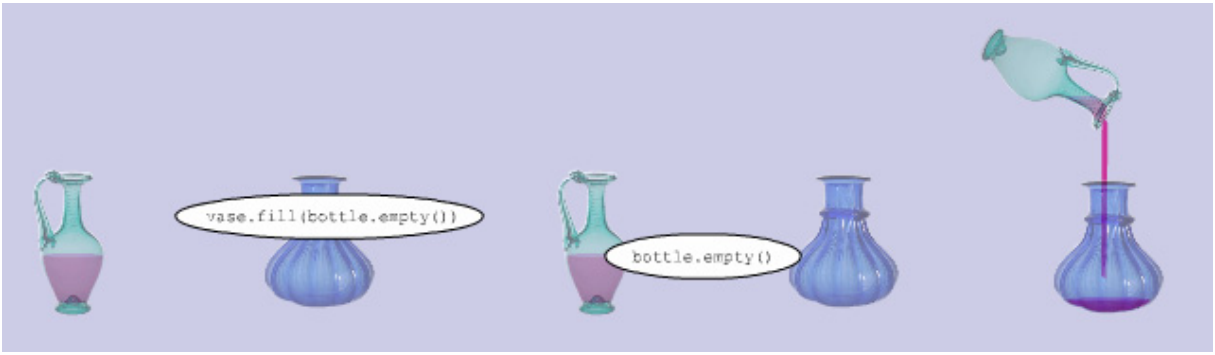


Abb. 115: Abspalten einer Botschaft aus einer verschachtelten Botschaft (pq_objects_a5_3)

In der zweiten Variante (pq_objects_a5_4) wird keine zweite Botschaft abgespalten sondern die Vase ergreift mit einem Manipulatorarm die Karaffe und schüttet deren Inhalt in sich hinein. Hier wird das Botschaftskonzept mit der Vorstellung passiver von außen manipulierter Objekte gemischt. Von den 23 beobachteten Spielern hielten 16 dieses Modell für passend (mittlere Konfidenz 81%).

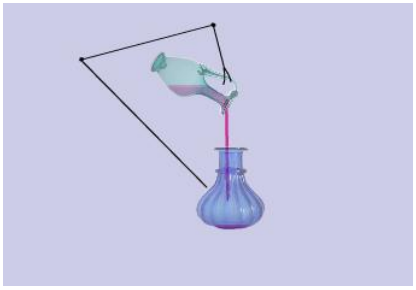


Abb. 116: Vermischen von passivem und aktivem Modell für Objekte (pq_objects_a5_3)

(2) In der zweiten Gruppe intuitiver Modelle wird die Programmzeile nicht als eine zusammenhängende Botschaft sondern von vornherein als eine Folge zweier getrennter Botschaften interpretiert. Die Auflösung der komplexen Botschaft erfolgt also durch den Akteur, von dem die Botschaft stammt (hier das Laufzeitsystem, der „oberste“ Akteur, der für die Ausführung des gesamten Programms zuständig ist). Einige Animationen der PVS greifen diese Vorstellung auf. Im Modell pq_objects_a5_6 zum Beispiel schickt das Laufzeitsystem zuerst die Botschaft `empty()` an die Karaffe, das Ergebnis (Karte mit Aufschrift 0.4) wird zurück gesendet und verschwindet aus dem Bild. Anschließend erscheint eine zweite Botschaft `fill(0.4)`, die an das Objekt `vase` geschickt wird und das Auffüllen des Behälters auslöst (Abb. 117). Von den 23 beobachteten Spielern hielten 15 dieses Modell für passend (mittlere Konfidenz 87%).

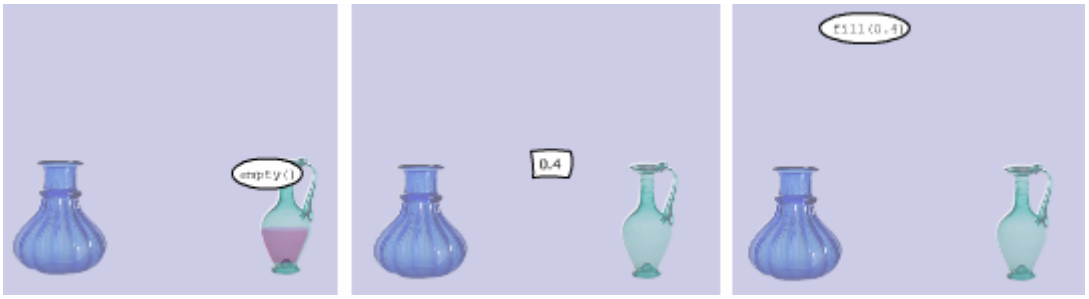


Abb. 117: Auflösung einer verschachtelten Botschaft durch das Laufzeitsystem (pq_objects_a5_6)

Die soeben beschriebene Abfolge kann als plausibel betrachtet werden. Die PVS enthält aber zu dieser Aufgabe auch eindeutig falsche Versionen mit zwei einfachen Botschaften. In der Animation (pq_objects_a5_1) wandert zunächst die Botschaft `empty()` zur Karaffe. Daraufhin entleert sich

diese und sendet dann die Botschaft `fill(0.4)` an die Vase. Diese füllt sich einem gewissen Volumen roter Flüssigkeit. Das Modell ist unpassend, weil das Objekt bottle definitiv keine Botschaft verschickt. Es wurde aber dennoch von 16 der 23 Spieler mit einer durchschnittlichen Konfidenz von 75% akzeptiert. Die 7 ablehnenden Spieler hatten dagegen ein erheblich geringeres Vertrauen in ihre Entscheidung (43%).

Abb. 118: Unplausibles Modell mit (pq_objects_a5_1)

(3) Eine dritte Möglichkeit schließlich besteht darin, die Anweisung

```
vase.fill(bottle.empty())
```

als eigenen Akteur zu begreifen, als Botschaft mit noch unbestimmtem Parameter. Dieser Akteur sendet zunächst die Botschaft `bottle.empty()` an die Karaffe, erhält einen Zahlenwert zurück und wird damit zu einer vollständigen Botschaft, die zum Objekt `vase` wandert (). Von den beobachteten 23 Spielern hielten 17 dieses Modell für akzeptabel (Konfidenz 82%) Abb. 119

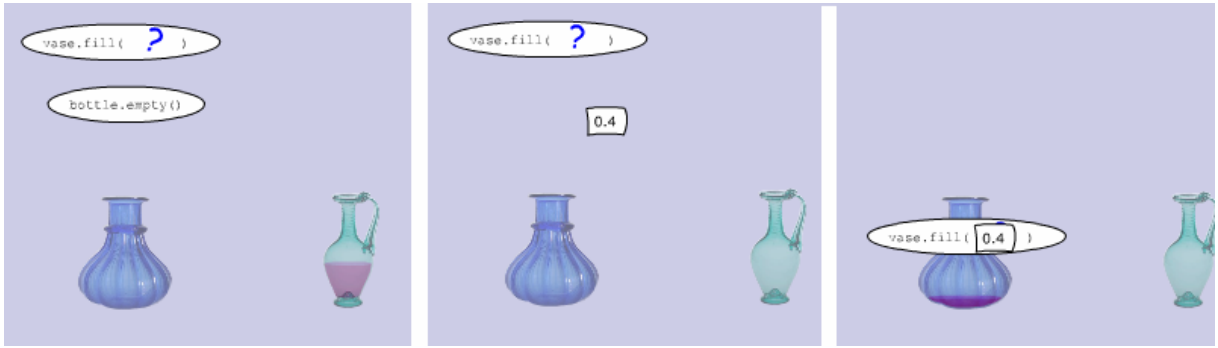


Abb. 119: Verschachtelte Botschaft als eigener Akteur, der Botschaften senden kann (pq_objects_a5_7)

6.9.1 Kontexte für die Verwendung von passiven Objektmodellen

Im Rahmen realer Programmierprojekte gibt es Zusammenhänge, in denen Objekte nicht als eigenaktive Entitäten gesehen werden, die Botschaften empfangen. So können in allen OO-Programmiersprachen öffentliche Attribute direkt gelesen oder verändert werden. Dabei bleibt das Objekt passiv und muss die Manipulation von außen erdulden. Häufig wird dies allerdings als schlechter Programmierstil gesehen. In Standardwerken der OOP (z.B. Balzert 1999) wird häufig empfohlen, den direkten Zugriff auf Attribute zu verbieten und stattdessen spezielle öffentliche Methoden für das Lesen und Schreiben zu definieren (set- und get-Methoden). Allerdings führt die Verwendung dieser Zugriffsmethoden zu komplexeren (und deshalb weniger intuitiven) mentalen Modellen. Mit Python ist es möglich, für ein Objekt (einer „New-Style-Klasse“) so genannte Properties zu spezifizieren (vgl. Weigend 2006a). Dabei werden für Attribute, die von außen erreichbar sein sollen, Zugriffsmethoden definiert, die bei einem scheinbar direkten Zugriff ausgeführt werden. Das heißt eine Zuweisung der Form

```
objekt.attribut = neuerWert
```

wird vom Laufzeitsystem als Methodenaufruf uminterpretiert und bekommt damit den Charakter einer Botschaft. Die Änderung selbst wird vom beauftragten Objekt – gewissermaßen „in Eigenregie“ – vollzogen.

Versetzen wir uns in die Situation eines Programmierers der eine solche Zuweisung im Rahmen einer Problemlösung formuliert. Er oder sie verwendet zwei intuitive Modelle gleichzeitig. Einerseits denkt er an eine Zuweisung, also die unmittelbare Veränderung des Zustandes eines passiven Objektes. Andererseits weiß er oder sie um den Mechanismus der Interpretation von Zugriffen auf Properties und verwendet dabei die Intuition eines aktiven Objektes, das auf Botschaften reagiert. Dies ist die typisches Beispiel für ein Modellcluster. Die gedankliche Vorstellung passiver Objekte spielt auch eine Rolle bei der Verwendung von Operatoren und Funktionen. So ist es vermutlich denkökonomischer, eine Anweisung wie

```
c = a + b
```

als Verarbeitung zweier (passiver) Objekte zu betrachten. Der Plusoperator ist in dieser Sichtweise ein Akteur, der die Objekte a und b als Eingabe verwendet und ein neues Objekt (die Summe) generiert. In einer Sichtweise, die dem objektorientierten Paradigma folgt, wird dagegen der Term `a + b` als Botschaft an das Objekt a interpretiert. Das Objekt a erhält den Auftrag mit Objekt b in Interaktion zu treten und ein neues Objekt zu generieren, das die Summe repräsentiert.

6.10 Indikatoren für die Validität der Ergebnisse

Das Python Quiz Objects enthält auch Animationen, die Vorgänge beschreiben, die objektiv im Widerspruch zur Semantik des Bezugsprogrammtextes stehen. Sie dienen in gewissem Maß als Kontrolle, inwieweit die PVS-Spieler überhaupt den Programmtext und die Bildersprache der visuellen Modelle verstehen oder das Spiel ernst nehmen, und sind damit ein Indikator für die Validität der erhobenen Daten. Abb. 120 zeigt einige Screenshots aus falschen Modellen für die Anweisung `bottle.empty()`. Im ersten Modell (pq_objects_a3_4) verschwindet die Karaffe (Objekt bottle), sobald die Botschaft (dargestellt als Oval) auf sie trifft. Es wurde nur von vier der 23 Spieler (17 %) akzeptiert. Im zweiten Modell (pq_objects_a3_3) wird das Oval, das die Botschaft repräsentieren soll, wie ein Etikett an die Karaffe geheftet. Dieses Modell hielten fünf der 23 beobachteten Personen (22 %) für passend.

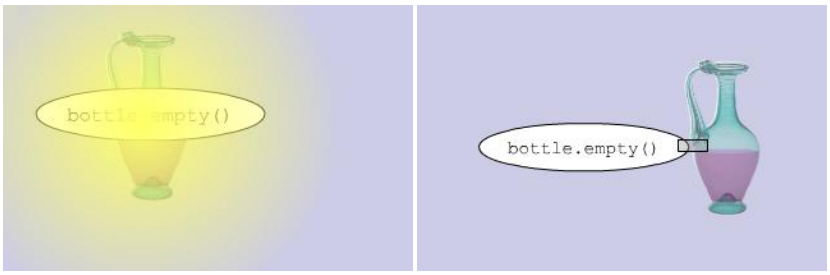


Abb. 120: Modelle, die der Semantik der Anweisung `bottle.empty()` widersprechen

7 Weitere Aspekte der intuitiven Modellierung

7.1 Intuitivität als messbare Größe

Der Begriffs „intuitiv“ wird in zwei Bedeutungsvarianten verwendet, die sich in einer Nuance unterscheiden. Häufig wird das Adjektiv „intuitiv“ in einem idealistischen, nicht steigerungsfähigen Sinn gebraucht. Eine Vorstellung ist intuitiv oder sie ist es nicht. Es gibt keinen Zwischenwert. Ich halte eine gedankliche Vorstellung für richtig oder ich habe Zweifel.

In der Praxis bewegen wir uns häufig an der Grenze zur Gewissheit. Insbesondere in kritischen Situationen (Entscheidungen und Handlungen, die schwerwiegende Konsequenzen haben), vergewissert man sich in Gedanken der Richtigkeit der gedanklichen Konzepte, die man z.B. für eine Problemlösung verwendet. Dieses sich Vergewissern kann fast unmerklich quasi zeitgleich mit dem Auftreten des Gedankens passieren, oder aber es erfordert größere Anstrengung.

Wenn wir von der Intuitivität eines Modells reden, dann gebrauchen wir den Begriff „intuitiv“ in einem steigerungsfähigen Sinn. Demnach kann ein Modell mehr oder weniger intuitiv sein. Der Hintergrund ist, dass Intuitivität ein subjektives Gefühl oder Erlebnis ist und von verschiedenen Begleitumständen des Erlebens abhängt. Man kennt die Erfahrung, dass etwa eine Beweisidee in einem bestimmten Kontext (z.B. während eines Vortrags) einleuchtet aber das Gefühl der Gewissheit später wieder verloren geht und erst durch eine gewisse Auseinandersetzung wieder hervorgerufen werden kann. Messbare Aspekte von Intuitivität sind:

Akzeptanzzeit: Die Zeit der Auseinandersetzung die benötigt wird um zu einem Gefühl der Gewissheit zu kommen. Bei der Rezeption eines mathematischen Beweises ist die Akzeptanzzeit die Zeit, die ein Rezipient benötigt, um einen Beweisschritt als richtig zu akzeptieren.

Evidenz: Idealerweise ist eine Intuition selbstevident. Das heißt man braucht keine weiteren Begründungen für ihre Richtigkeit. „Sich einer Intuition vergewissern“ heißt, dass man zur Sicherheit ihre Korrektheit mit Hilfe anderer Modelle (z.B. Kontrollmodelle) prüft. Das kann so schnell gehen, dass man sich dessen gar nicht mehr bewusst ist oder aber es wird zu einer bewussten subjektiv wahrnehmbaren – eventuell sogar (im Rahmen einer Erklärung) externalisierten – gedanklichen Operation.

Gewissheit: Wie schätze ich das Risiko ein, dass ich mit meiner Intuition falsch liege? Intuitionen ändern sich im Laufe einer Biographie. Jeder hat bereits erlebt, dass sich etwas als falsch herausgestellt hat, das man vorher als sicher richtig eingeschätzt hat. Auf Grund solcher Erlebnisse kann man sich eigentlich niemals vollständig sicher sein.

7.2 Überstülpen des EVA-Modells als Beispiel für Überstrukturierung

Betrachten wir folgendes Programm:

```
a = "up"
b = "and down"
c = a + b
print c
```

Jesse (16 Jahre, Virginia, USA) hat dieses Programm folgendermaßen durch eine Flash-Animation visualisiert: Das Programm wurde durch eine Box dargestellt. In diese Box wandern die beiden ersten Zeilen des obigen Programmtextes hinein und als Ausgabe kommt die konkatenierte Zeichenkette heraus. Jesse hat die ersten beiden Zuweisungen als Eingabe interpretiert, was sie formal nicht sind.

Man kann dies insofern als Überstrukturierung deuten, als hier dem Programm die Struktur „Eingabe-Verarbeitung-Ausgabe“ übergestülpt wurde. Das Verwechseln von Zuweisung und Eingabe wurde von Samurcay (1989) beobachtet.

Intuitive Modelle der Informatik sind gedankliche Vorstellungen über informatische Konzepte, die mit subjektiver Gewissheit verbunden sind. Menschen verwenden sie, wenn sie die Arbeitsweise von Computerprogrammen nachvollziehen oder anderen erklären, logische Korrektheit prüfen oder in einem kreativen Prozess selbst Software entwickeln. Diskutiert werden in diesem Buch intuitive Modelle für grundlegende Aspekte einer Programmausführung - etwa die Allokation von Aktivität, Benennung, Daten, Kontrollstrukturen und Verarbeitung. Mit Hilfe eines Systems von Online-Spielen, der Python Visual Sandbox, werden die psychische Realität verschiedener intuitiver Modelle bei Programmieranfängern nachgewiesen und fehlerhafte Anwendungen (Fehlvorstellungen) identifiziert.

ISBN 978-3-940793-08-9