Data-driven Tool Construction in Exploratory Programming Environments

by

Marcel Taeumel

A thesis submitted to the

Digital Engineering Faculty
University of Potsdam

in partial fulfillment of the requirements for the degree of

Doctor Rerum Naturalium
(Dr. rer. nat.)

Advisor

Prof. Dr. Robert Hirschfeld

Software Architecture Group
Hasso Plattner Institute
University of Potsdam, Germany

February 2, 2020

Abstract

This work presents a new design for programming environments that promote the exploration of domain-specific software artifacts and the construction of graphical tools for such program comprehension tasks. In complex software projects, tool building is essential because domain- or task-specific tools can support decision making by representing concerns concisely with low cognitive effort. In contrast, generic tools can only support anticipated scenarios, which usually align with programming language concepts or well-known project domains.

However, the creation and modification of interactive tools is expensive because the glue that connects data to graphics is hard to find, change, and test. Even if valuable data is available in a common format and even if promising visualizations could be populated, programmers have to invest many resources to make changes in the programming environment. Consequently, only ideas of predictably high value will be implemented. In the non-graphical, command-line world, the situation looks different and inspiring: programmers can easily build their own tools as shell scripts by configuring and combining filter programs to process data.

We propose a new perspective on graphical tools and provide a concept to build and modify such tools with a focus on high quality, low effort, and continuous adaptability. That is, (1) we propose an object-oriented, data-driven, declarative scripting language that reduces the amount of and governs the effects of glue code for view-model specifications, and (2) we propose a scalable UI-design language that promotes short feedback loops in an interactive, graphical environment such as Morphic known from Self or Squeak/Smalltalk systems.

We implemented our concept as a tool-building environment, which we call Vivide, on top of Squeak/Smalltalk and Morphic. We replaced existing code browsing and debugging tools to iterate within our solution more quickly. In several case studies with undergraduate and graduate students, we observed that Vivide can be applied to many domains such as live language development, source-code versioning, modular code browsing, and multi-language debugging. Then, we designed a controlled experiment to measure the effect on the time to build tools. Several pilot runs showed that training is crucial and, presumably, takes days or weeks, which implies a need for further research.

As a result, programmers as users can directly work with tangible representations of their software artifacts in the Vivide environment. Tool builders can write domain-specific scripts to populate views to approach comprehension tasks from different angles. Our novel perspective on graphical tools can inspire the creation of new trade-offs in modularity for both data providers and view designers.

Acknowledgments

Oh well ... it took me about seven years to conceive and present this work. Sometimes, it was not that easy to sort out the overwhelming amount of ideas, that is, to separate "Blah!" from "Aha!" to find added value. During that time, there were *many* people that supported my work, directly or indirectly, and thus influenced my decisions.

First of all, I am very grateful to my thesis advisor Robert Hirschfeld, who has been ensuring a *superb* work environment. You made all the books that I wanted to read available. You established numerous connections with interesting people so that I kept on talking about my ideas, giving demonstrations, and thus constantly and consciously reviewing my steps. You enabled many travel arrangements so that I could experience international cultures and research venues. As *sensei* Robert, you have been teaching me Bujinkan Budō Taijutsu, which noticeably improved my observation and movement skills. Thank you for all these opportunities. I know that many organizational aspects have been carried out with care by our chair's secretary, Sabine Wagner, to whom I am naturally very thankful, too.

Second, I owe my gratitude to Bastian Steinert, who inspired my behavior with his constant drive for *accuracy and diligence*. Your typically deep and critical discussions about your (and my own) topics have been changing my practices for the better. It has always been enlightening to observe your presentations and learn from your research artifacts in general.

Third, I am thankful to Patrick Rein, who can lend one an ear, even for the *craziest* of hypotheses. You are very open-minded and curious. You have always been *patiently* discussing my ideas, even if that effort did not contribute to your own workload. Especially during our travels to California, I enjoyed your challenging thoughts about my research.

[|]

I acknowledge the financial support I received from the HPI Research School and the Design Thinking Research Program. That support allowed me to carry out my research full-time with an unbiased focus on relevant issues in the problem domain of my choosing. Both programs fostered my progress through several workshops, where I could present my work and talk about contemporary issues with other researchers.

I am grateful to my external reviewers for exploring the (quite many) pages of this work. Thank you, Kim Mens, for welcoming me at your chair in Louvain-la-Neuve. Your critical comments on a previous version of this work helped me clarify important sections. And thank you, Michele Lanza, for having me at your chair in Lugano. The profound discussions with your group revealed relevant aspects about live programming and thus made me think.

[|]

I want to thank all (former and current) colleagues in our Software Architecture Group for constantly enriching the work atmosphere. You were always available for asking questions, trying out prototypes ... and breaking them. You can speak your mind and openly question my design decisions in a respectful manner. You always filled my talks with *loads of* feedback, even though the (effortfully prepared) slides quickly became superfluous. Thank you! Thank you for the *coffee* breaks. Thank you for the tea ceremonies. Thank you for the endless battles in Micro Machines or Mario Kart during (of course!) recreational lunch breaks. Thank you Malte "Red Car" Appeltauer, Michael "Blue Car" Perscheid, Jens "Lively" Lincke, Bastian "Undo" Steinert, Tobias "LaTeX" Pape, Tim "Constraint" Felgentreff, Toni "Concept" Mattis, Patrick "Flame Rider" Rein, Johannes "Frantic Items" Henning, Stefan "Rosalina" Ramson, Fabio "Mario" Niephaus, and Marcel "Hello Marcel!" Weiher.

I also want to send a big thanks to all my fellow researchers and "travel buddies". On our journeys abroad heading for conferences or workshops, I enjoyed each fruitful discussion or *joyful small talk* with you. We watched beautiful sunsets in California, observed eagles in Arizona, explored the streets of San Francisco, dared all kinds of food in Nanjing, stood on top of Table Mountain in Cape Town, or just had a beer in some local bar. Thank you Bastian Steinert, Frank Feinbube, Christian Neuhaus, Jan Klimke, Ibrahim Takuona, Sebastian Hahn, Tim Felgentreff, Anja Jentzsch, Jens Lincke, Lena Herscheid, Ekaterina Bazhenova, Sabine Wagner, Daniel Richter, Lauritz Thamsen, Patrick Rein, and Johannes Henning. We shared numerous invaluable experiences.

I was in the lucky position to have met and frequently visited *veterans and experts* in Lisp, Smalltalk, and the field of exploratory (or live) programming. Especially during my travels in California, I enjoyed many conversations with Richard "Dick" Gabriel, Daniel "Dan" Ingalls, Eliot Miranda, Gilad Bracha, and Bert Freudenberg. Thank you all for giving me advice and historical insights. You opened my eyes to many longly existing ideas and longly known challenges. I am very grateful for this healthy mix of frustration and enlightenment. This also addresses my friends from a younger generation of Smalltalkers, namely Robert Krahn

and Marko Röder, who flew out of Potsdam to seek new teachers in the Bay Area. Thank you for never being too busy for a quick chat and thanks for all the helpful comments on my research. ... I am still sad that Andreas Raab has passed away shortly after the beginning of my doctoral studies. My questions for you will forever remain unanswered. I have been and will always be grateful for your accomplishments in the Squeak/Smalltalk community. Your work is full of *mindfulness and care*. Thank you.

I am thankful for having colleagues and friends with whom I shared the *goals and agonies* of doing a PhD. We exchanged hopes and doubts, pictured (im-)possible impact of our research on humanity, or teased each other about the current page count. Thank you Sebastian Hahn, Sebastian Pasewaldt, Jan Klimke, Daniel Richter, Max Plauth, and Vera Grötz. I think we lifted some burdens from our shoulders by just talking about them.

Of course, I would like to thank all my friends for helping me manage that crucial *work-life balance*. We enjoyed so many activities for the body: (rock) concerts, (beach) volleyball, badminton, (ultimate) frisbee, cycling, skiing, hiking, bouldering, (kind of) randori-ing. We enjoyed so many activities for the mind: playing board games, watching (horror) movies, cooking delightful sushi or pasta, struggling with escape rooms. Thank you Jens Lincke, Jirka Müller, Robert Hirschfeld, and Daniel Richter. Thank you Stephanie "Stephi" Lehmann, Philipp "Tessi" Tessenow, Felicia "Feli" Flemming, Tim Felgentreff, Lan Jiang, and Fabio Niephaus. Thank you Anne Porath, Constantin Malycha, Nina Wagener, Jan Kersten, Vera Grötz, Sebastian Lindner, Jens Lindner, Michael Grünewald, Matthias Jacob, and Michael Grundmann.

There were many other people who supported and influenced my work. I am thankful to Stefan Hanenberg for giving me advice on how to design, conduct, and document controlled experiments. I am also thankful to the numerous undergraduate and graduate students who took the chance to work in the realm of my research. This includes students from lectures, project seminars, bachelor's projects, and master's theses. Thank you all for constantly calibrating my line of thought with your individual opinions and experiences.

[|]

Finally, I want to thank my parents, Christine and Peter, and my entire family for the continuous support during all these years. You do *firmly* believe in my success and happily celebrate all achievements along the way toward a promising future.

Contents

List of Figures

Part I

Domain-specific Tools for Program Comprehension

1 Introduction

Programmers have to understand the mechanics of complex *software systems*, so they can add new features, refine existing ones, or repair broken ones. The typical environment in which any software is created and maintained hosts many *interactive tools* that support producing and consuming all kinds of related information. There are *generic tools*, which support generic programming-language concepts, and *specific tools*, which support the vocabulary and information representation of certain domains. Now, the use of domain-specific tools can have a positive effect on *program comprehension*: programmers can make fewer mistakes and produce higher-quality software. Thus, it is beneficial to provide tools that accommodate the domain of the particular software project. Ultimately, there is perpetual interest in both academia and industry to research effective and efficient means to *create, integrate, and adapt* such tools in the development process.

The major goal of this work is to establish a novel perspective on the construction of graphical, interactive tools for (and in) object-oriented environments. In figure 1.1, we exemplify and idealize such an environment, comparing with Eclipse,[1] Visual Studio,[2] or IntelliJ IDEA.[3] The *glue* between data and graphics poses a costly barrier for tool builders to overcome—given that all structured information is represented as connected objects and appropriate designs for graphical representation are available. We propose to shrink this barrier with the use of declarative, data-driven, and functional languages that entail interactive, modular means of expression. As a result, more suitable graphical tools might be created for a particular project—maybe also some that would have never been discovered in the traditional way. Note that we observe, reason, and act in the course of *directness* [172] and *liveness* [197]. Thus, we encourage the way of *exploratory programming* [167], in which any tool user can become a tool builder. Yet, such a mindset shift could only happen if tool building becomes reasonable under all circumstances.

We begin with a description of this work's *context and motivation*, which peaks in our research question to answer. Then, we examine the *state of the art* in tool building, which includes object-oriented design, declarative languages, interaction paradigms, and existing systems that support related activities. After setting the ground, we summarize our approach and present the *thesis statement* that guides this work. Finally, we outline our *contributions*, which support both research question and thesis statement, and finish with this work's *organization* chapter by chapter.

[1]Eclipse, `https://www.eclipse.org/`, accessed 2018-11-24
[2]Microsoft Visual Studio, `https://visualstudio.microsoft.com/`, accessed 2018-11-24
[3]JetBrains IntelliJ IDEA, `https://www.jetbrains.com/idea/`, accessed 2018-11-24

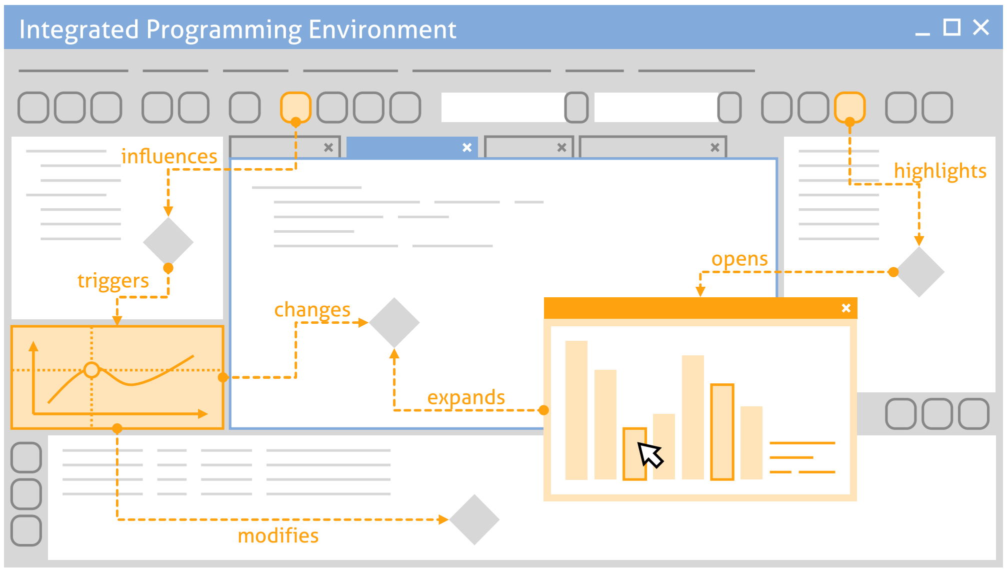


Figure 1.1: Traditional programming environments impede tool building. Any concrete idea (here: orange) poses challenges regarding process, control, data, and presentation integration. Only high-value changes are likely to be implemented. Uncertain ideas remain tacit knowledge, hardly to be evaluated.

1.1 Context and Motivation

Programming tools support programmers in basic programming tasks such as exploring, writing, running, and debugging source code [187, 55]. Besides such code-centric tasks, there are tools for bug tracking, team communication, architectural modeling, or version control—each producing or consuming different kinds of *software artifacts* such as tickets, e-mails, diagrams, and versions. In general, programmers benefit from tools that integrate and visualize all relevant information to better understand and maintain the complex software system at hand. By employing computational power as well as interactive, high-resolution input and output devices, complex or repetitive tasks can be accomplished faster and are less prone to human error [134, pp. 105–140]. Programmers can free up cognitive resources [120] and leverage their intellect for novel challenges. Consequently, they acquire skill by mastering programming languages *and* programming tools. That is, they learn how to *choose, configure, and apply* their tools in an effective and efficient manner.

Toward Domain-specific Programming Environments

Programmers work in *programming environments*, which provide (1) a selection of tools for recurrent tasks, (2) integration of tools to expedite context switches, and (3) support for externalizing thoughts about the task at hand. Considering *program comprehension* in general, programmers can fetch, examine, and retain information (or software artifacts) in an organized, integrated fashion [95]. The selection of tools aligns typically with the concepts of programming languages such as class browsers and object inspectors for class-based, object-oriented languages. The integration of tools covers usually a coherent structural and graphical representation of information such as everything-is-a-file [153] and everything-is-an-object [65] as well as list-based views with pop-up menus and clickable buttons [58]. The transcription of programmers' thoughts to the screen addresses graphical aspects such as window management [127] and the choice of visualization [10].

Programmers can benefit from tools that accommodate specifics of the project domain, current task, or own preferences. However, the possibilities of merely choosing and configuring a tool's interface are limited. For example, a supportive, external tool might lack a proper integration into the environment and personal working habits. Also, any tool's configuration space has to be anticipated in its design phase, which is likely to omit the yet unknown domains of future software projects. Consequently, *dedicated tool building* is an integral part in the development process. During program comprehension, programmers need specific tools to fetch, examine, and retain relevant information [95]. At best, user interactions make *software artifacts* feel almost *tangible* like real-world objects. *Fetching* addresses the integration of new artifacts, a shared awareness across tool boundaries, or the derivation of new information. *Examining* addresses the exposition (or filtering) of structural aspects that make artifacts become tangible, relatable, and hence understandable. *Retaining* addresses a larger time span where programmers want to persist and recall insights and keep a sense of familiarity of information. — The diverse nature of comprehension questions has already been investigated [177] and partially addressed with configuration during tool use [29, 30].

The Challenge of Tool Building

For tool builders, unfortunately, the *glue* involved to structurally query, visually map, and coherently present software artifacts can be hard to *find*, *change*, and *verify*. The timeliness and quality of *feedback* is unsatisfactory for the tool builder. On the one hand, scattered code (and other resources) with many levels of indirection makes it difficult to navigate from a tool's graphics to its mechanics. On the other hand, the effects of changes are not easy to verify (or test) because tools have to be restarted to achieve a consistent state. While there is often support for interactively designing a tool's *presentation language*, tool builders have to manually dive into the tool's mechanics to express *query and mapping languages*. They face rather generic framework glue, which they have to continuously separate from domain-specific, effective expressions. — Note that we focus on glue rather than making relevant information technically accessible in the given programming environment. Note that we also assume that there are already existing visual designs (or visualizations) to be populated with the right amount of information.

Thus, we address a common issue in program comprehension that occurs during the construction of graphical tools *for* program comprehension (and modification). The research question we approach reads as follows:

Research Question How can we support tool building in object-oriented, graphical environments where programmers mainly fetch, examine, and retain information in the form of tangible software artifacts through interactive views?

We are encouraged by the existence of *live programming systems* [197], which blend languages and tools into single environments to promote short feedback loops for general application development. For example, the Squeak/Smalltalk system [77] propagates code changes throughout the running environment so that all objects can exhibit the changed behavior directly afterwards. That system also features the Morphic framework [108], which provides inherent tangibility and direct manipulation [172] for all graphical elements. More recent approaches focus on Web development such as the LivelyKernel using JavaScript [105]. Overall in such systems, programmers can employ programming languages that feel like lightweight scripting languages [140] to create and maintain complex tools in short feedback cycles. The involving relief of cognitive capacities can lead to a more curious, exploratory programming style [167]. There, programmers can flexibly explore problem and solution space along the way. — *But how to handle the glue between data and graphics in tool building accordingly?*

1.2 State of the Art

It is common practice to build new technologies with the existing ones and eventually replace them. In the field of programming, such bootstrapping is important for implementing new languages, tools, and environments. For example, textual languages yield visual languages, keyboard input yields support for mouse or touch input, or text interfaces yield graphical interfaces. Consequently, *new* tools for programming should be built with *existing* tools for programming. — Unfortunately, we observed that users have to fall back to traditional tools frequently because new ones cannot match all requirements for the task at hand.

Programmers as Under-appreciated Users

By learning from past experiences and efforts, nowadays, we understand more about the cognitive capacities of programmers and humans in general [120, 122, 142]. Especially the recurrent, entangled challenge of program comprehension has been thoroughly documented regarding its navigation strategies [113], verbal explication [177], analysis techniques [35], and tool interaction [95, 55]. All these insights drive experimentation and innovation. There are, for example, *guidelines* to design domain-specific, multi-view interfaces [10], and software visualization tools [88]. There are also more general *heuristics* for designing non-frustrating user interfaces considering memory load, consistency, or feedback [131, pp. 115–163]. We think that the concept of *flow* [38] is driving factor in programming tool evaluation [125] and evolution. — Yet, given a historical perspective of programming tools [187], it is paradoxical that only well-understood problems have been solved with rather high efforts [156]. This is unfortunate for programmers, being tool users or builders, that face novel challenges in novel domains.

Tools that support well-known, specific aspects of program comprehension focus on programmers as *users* not *builders*. There are many interactive visualizations, which typically entail static or dynamic code analysis, embedded in the programming environment to support exploration. Even though there are often *means of configuration* to express domain- or task-specific intents, those are too limited for an elaborate integration of different artifacts or novel exploration paths. For static analysis, there are SourceMiner [27, 53], Software Terrain Maps [40], Voronoi Treemaps [70], or AspectMaps [51]. For dynamic analysis, there are also pre-defined views [157] such as Circular Bundle Views [34] to explore dynamic relationships between code artifacts. These existing approaches offer only a *fixed* set of views with *limited* means of configuration. — We observed that views with limited data-to-graphics mappings cannot sustain tool (or application) building in the long term. When facing unforeseen challenges, such views force programmers to constantly backtrack to lower-level representations such as text.

Borders to Overcome, Approaches to Unify

There are many ideas on how to improve the situation for programmers that are able to become tool builders. In the following, we take a look at the state-of-the-art in window management, scripting languages, data-driven (or dataflow) approaches, and the concept of "directness" in graphical interfaces. These explanations clarify the need for a programming environment that *unifies* many existing ideas in a tool-building context.

In terms of *window management*, there are several approaches that try to overcome the traditional, confined, "boxed" design known from Eclipse and Visual Studio, where a central code view dictates perspective. The Moldable Inspector [30] and the Patchworks Code Editor [73] propose the use of horizontally unlimited *tapes* to juxtapose software artifacts. Code Bubbles [21] and Code Canvas [42] call those containers *canvases*, which are both more flexible on the vertical screen axis. The Gaucho environment [139] uses artifact-specific *shapes* organized in rectangular containers called *pampas* to represent information, which compares to traditional overlapping windows. — Unfortunately, all these designs exist in isolation as an environment's baseline without proper means of combination or abstraction to construct more elaborate interfaces.

In terms of *scripting languages*, a common approach to blur the line between tool configuration and building is the integration of declarativity. We found that Prolog-based, logical languages are well-suited to query the complex structure of software artifacts such as in Jquery/TyRuBa [39] and CodeQuest/Datalog [71]. Also, imperative, high-level languages such as JavaScript and Smalltalk promote concise, declarative expressions for tool building. Examples include LivelyKernel [105], Glamour [24], and IntensiVE [116]. The latter combines both object-oriented Smalltalk and the logical SOUL language [117] in a flexible way. — We see such scripting languages as driver for low-effort, high-quality tool-building environments.

In terms of *data-driven approaches*, we found several projects that focus on data access and transformation. The Lucid language [209] is an early example of dataflow programming, which differs from procedural programming because programmers model data streams not data fields. The more recent KScript [136] brings the idea to graphical interfaces for event streams. Many such data-driven concepts resemble the functional programming paradigm. As a compromise between object-oriented and functional systems, Transmorphic [168] maps object structure on source code (functions) for graphical, direct manipulation environments. Since we target such environments for program comprehension, these projects offer interesting perspectives on tool design. Fabrik [79, 106] connects interactive views with transformation scripts written in Smalltalk (or JavaScript). Those scripts are directly accessible and hence foster a tool building experience with short feedback loops. We think that such an experience can be promoted by the separation of *tools* from the *materials* they work on under a given *aspect* [159]. — Yet, we could not find a comprehensive design that applies those ideas in a tool-building environment.

Finally, existing models for *immediacy* [206], *directness* [75], and *tangibility* [110] are somewhat descriptive for efficiency and effectiveness in user interaction. Users want to understand what they see on screen so that they can express their goals directly and evaluate the results immediately and repeat the process for complex tasks. In programming environments, this concept is based on a common data representation [63] to foster integrated, interactive tools. In Smalltalk systems [65], for example, everything is an object, and object graphs can represent any kind of structured information. Thus, tools can build on that object graph. From a tool design perspective, object-oriented user interfaces [31] apply this idea throughout the technology stack, which has been elaborated in Naked Objects [144] for business applications. — Yet, traditional, object-oriented, direct-manipulation systems cannot handle multiple software artifacts at once, which impedes complex programming tasks. Repetitive interactions can be prone to errors if done manually.

1.3 Thesis Statement

Programmers use graphical tools to explore and understand a complex information space, which consists of interconnected software artifacts such as source code and documentation. There are environments with well-integrated tools to support multiple, concurrent tasks. Still, the art and craft of tool building remains important because only adapted tools can access and present complex domain structures efficiently. Unfortunately, the feedback loop for creating and modifying tools is rather long and costly, which yields only tools with a high long-term value. Yet, we are interested in finding more efficient ways to build tools so that maybe even unique tasks might be supported, which would adopt the idea of exploratory programming in live programming systems.

In traditional environments, interactive visualizations are typically no "through streets" but *backtracking* is a necessary evil. Programmers, which includes tool builders, have to *retreat* to more primitive views to combine information and insights such as via the clipboard into text editors. Many promising interaction paradigms such as unlimited canvases are not combinable into complex tools, but they only form a *single* baseline for the user. Yet, there are high-level scripting languages that are fast, expressive, and promising in terms of generic software development, which includes tool building. There are many data-driven concepts, languages, and frameworks that favor data structure over application behavior, but they have not been explored in a *tool building context*. Considering program comprehension, we are looking for an environment that leverages interactivity and direct manipulation for numerous artifacts at once.

Data First, Tools Follow

We build on the idea of separating *model* and *view* like in Smalltalk's model-view-controller framework [98, pp. 7–12] or the more general model-view-presenter pattern [149]. The effect of such a separation on the actual program design can vary, which one can observe in the document-oriented Hopscotch [26] or the integration-focused StarBrowser [213]. In a comparable way, we, too, base our approach on a Squeak/Smalltalk environment [77] using the Morphic framework [108]. Thus, we assume a common data representation in a pure object-oriented system and inherent tangibility of graphical items, which renders our approach twofold:

1. Simplicity through live, data-driven scripting
2. Simplicity through live, interactive view composition

This work's main line of thought reads as follows:

Thesis Statement In a tool building framework that describes graphical tools as compositions of data-driven, script-based, interactive views, many common programming tools can be expressed in that design, and the results will be easy to modify directly during use to accommodate specific domains and tasks.

Note that we assume a *dedicated* tool building activity. Programmers are tool builders with a specification and time. We share a vision that goes beyond this work yet influences our design decisions: if we can reduce tool-building effort, programmers are more likely to improve their own tools. Such ad-hoc switching between user and builder would support exploratory programming [171, 60].

1.4 The Vivide Programming Environment

We designed and implemented a new framework and environment, which we call Vivide [194, 193], that builds on the Squeak/Smalltalk programming system. The source code and Smalltalk images are available via GitHub:

The Vivide Programming Environment
`https://github.com/hpi-swa/vivide/`

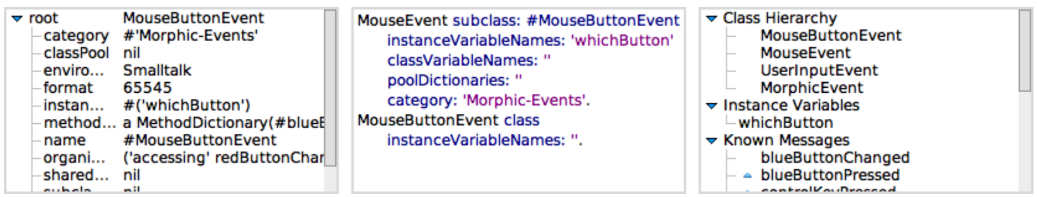
In this work, we explain the concepts and mechanics that represent that environment to answer our research question and to materialize our thesis statement.

Contributions

1. **Block-based Scripting Language** (section 4.1, section 5.1)
   We apply Squeak's anonymous functions (i.e. block closures) and collection protocol to define a subset of Smalltalk that *transforms* one set of objects into another set of objects with the goal to *generate* list (or tree) models with labeled properties at each node. Such models can populate interactive views or support data-processing pipelines. The following example squares number objects:

```
script := [:in :out | in do: [:num | out add: num * num]].
objects := #(1 2 3 4). "or data or (software) artifacts"
model := script interpretWith: objects. "or view model"
```

2. **Data-driven Strategy for Tool Design** (section 4.2, section 5.2)
   We propose a strategy to design tools that aim for *modular presentation* and *perpetual tangibility* of software artifacts (or objects). Our strategy distinguishes three rules: Rule of Distinctiveness for single artifacts, Rule of Similarity for sets of artifacts, and Rule of Context for artifact integration between views. Single-object tools form the basis for our strategy, exemplified for a class object in three characteristic views (i.e., generic tree, text definition, specific tree) as follows:

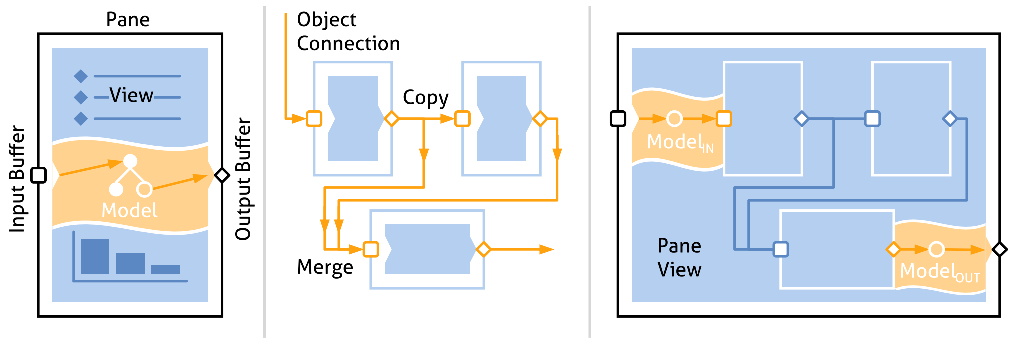


```
▾ root        MouseButtonEvent
   category   #'Morphic-Events'
   classPool  nil
   enviro...  Smalltalk
   format     65545
   instan...  #('whichButton')
   method...  a MethodDictionary(#blueE
   name       #MouseButtonEvent
   organi...  ('accessing' redButtonChar
   shared...  nil
```

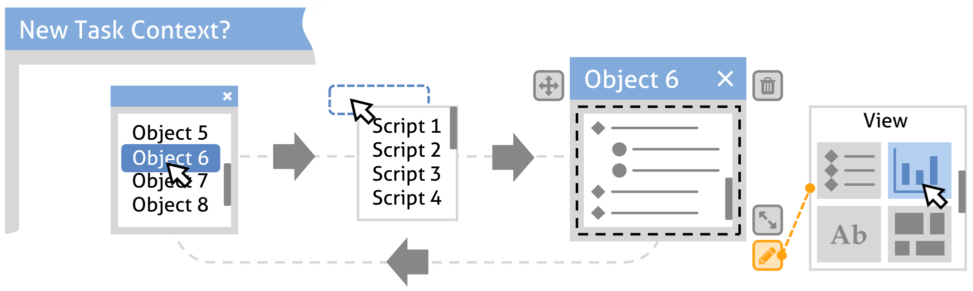
```
MouseEvent subclass: #MouseButtonEvent
    instanceVariableNames: 'whichButton'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Events'.
MouseButtonEvent class
    instanceVariableNames: ''.
```

```
▾ Class Hierarchy
      MouseButtonEvent
      MouseEvent
      UserInputEvent
      MorphicEvent
▾ Instance Variables
    whichButton
▾ Known Messages
      blueButtonChanged
      blueButtonPressed
```

3. **Morph-based UI-design Language** (section 4.3, section 5.3)
   We apply Squeak's Morphic and its halo concept to provide an interactive design experience for the tool builder. There are *panes* as basic tool elements, each holding objects, a script, a generated model, and the populated view (morph). Multiple panes can be combined via *connections* to exchange objects. Multiple sets of panes can be abstracted into *pane views* to be used in other scripts. This trichotomy of primitives, means of combination, and means of abstraction looks like this (from left to right):



4. **Data-driven Working Practice** (section 4.4, section 5.4)
   We enable a new working practice because programmers have to choose a set of artifacts first before examining structure in interactive views. This includes the use of text-based searching (or querying) facilities to get started. This is different from users having to "open a browser/explorer/perspective" to get started in traditional environments: artifacts first, tools follow. In our environment, users repeatedly choose objects, scripts, and views during program comprehension tasks:



The bottom line is that programmers being tool users can directly work with tangible representations of their software artifacts in the VIVIDE environment. Tool builders can easily write scripts that populate new views to approach program comprehension tasks from different angles. Multiple views can be combined into user interfaces that support more complex exploration tasks. Note that effective use of our new scripting language and interaction concepts requires training because the user's perspective on tools has to change. Yet, the main benefit of being able to shape any graphical representation of any kind of information is that the entire programming environment can be under the programmer's control. So, programmers can be *both* tool users and tool builders.

Goals

> "[...] This is the difference between 'education' and 'training.' Medical school is education, first aid is training. Education requires fundamental understanding, which can be used to grasp and respond to a nearly infinite variety of threats; training involves singular actions, which are useful only against anticipated challenges. Education is resilient, training is robust." — Team of Teams [115, p. 153]

Similarly, we are looking for a way to improve *resilience* in the tool building process, which complements the traditional *robustness* of generic but polished off-the-shelf tools. A tool building environment is likely to support a variety of unanticipated, domain-specific challenges if it implements a fundamental understanding of what programmers are doing in program comprehension tasks. This is were *education* plays an important role when designing such environments. In the past decades, the field of software engineering acquired much knowledge about how programmers think and work. Programming environments should leverage these insights to be flexible *at short notice* in the face of unforeseeable programming tasks of emerging complexity. Consequently, we have been pursuing the following goals:

Applicability There are many data sources and visual designs available in academia and industry, which we want to *re-use and integrate*. We are looking for an environment that joins these existing efforts. Given the profound structure of software artifacts, we want to provide tangibility even for information that has no inherent graphical representation.

Conciseness The expressions of query, mapping, and presentation languages should be concise and close to the domain or task so that tool builders can save time in code reading and writing. Thus, any programming or scripting language involved should be *declarative* as much as possible. As a baseline, we want to apply the dynamic aspects of the Smalltalk language.

Agility & Feedback We want tool builders to work with concrete artifacts throughout the tool design and construction process. We aim for short feedback loops so that they can iterate and learn quickly; and they might even solve an actual challenge with the given set of artifacts. Thus, we embrace an emergent and adaptive tool design by promoting ad-hoc customization and hence even throw-away prototypes.

Scalability When designing a tool building environment, we want to begin with a few simple concepts to connect data to graphics. Building on that, we want to add means of combination for more complex information exchange between tools.

Eventually, there should be means of abstraction to promote re-use and design elaborate tool interfaces.

MINDSET & PERSPECTIVE We are likely to re-design the perspective users have on graphical tools in programming environments. We want to shift the focus from tools that support *tasks* toward tools that provide tangible access to *artifacts*. When artifacts (or objects) come first, tools and task support will follow. The task of program comprehension is of primary interest in our research question. For example, application debugging and information exploration should become the same thing. We want to support programmers in self-organizing their work environment, which includes externalizing mental models and low-effort tool switching.

Methods

In order to achieve these goals, we require a working environment that supports exploration of ideas, experimentation with concrete tasks and artifacts, and the possibility of achieving production state to approach real-world challenges. Thus, we did apply the following methods:

- **Identify and form** pragmatic definitions for information, programmers' tasks, tangibility in graphical tools (or environments), and tool building. See chapter 2.
- **Transfer** these definitions to a technical level using the Squeak/Smalltalk system, which we consider purely object-oriented, and the Morphic graphics framework, which we consider inherently tangible. See section 2.5 and chapter 3.
- **Design and implement** a new framework in this environment so that standard programming tasks can be carried out. See section 3.3, section 3.4, chapter 4, and chapter 5.
- **Explore** tool modifications for daily programming tasks in that new framework without having to use the traditional tools, that is "eat your own dog food." See chapter 6.
- **Evaluate** applicability (and usability) in the form of student projects. Document misconceptions, bugs, and new ideas in an iterative fashion. See chapter 7.
- **Design and implement** a controlled, self-running experiment to evaluate effectiveness of the new framework and tool-building method. Regard proper training and pursue a within-subject design with two similar tasks to mitigate carry-over effects [82, 92]. See chapter 8.
- **Explore** that experimental setup in several pilot runs with undergraduate and graduate students. Document issues in training, application, and quality of the resulting tool modifications. See section 8.3.

1.5 Organization

We outline our work in three dimensions: parts, chapters, and field of research. Parts separate *problem*, *solution*, *evaluation*, and *outlook*. Chapters divide this work's content into self-contained chunks. The field of research splits our contribution into *programming philosophy* and *software engineering*, which we alternate in chapters as depicted in figure 1.2. We finish with notes on presentation to further guide the reader through this document.

Parts and Chapters

Part I — Domain-specific Tools for Program Comprehension

- Chapter 1, which is this one, gives an overview of this work's background, motivation, contributions, and related work. It also elaborates on our *goals and methods*, which influenced our perception of the problem domain and the design of our solution.
- Chapter 2 provides background information about software artifacts, program comprehension, exploratory programming, and tools. It elaborates on the programmer's responsibility to choose, configure, and build programming tools to efficiently approach the tasks of feature addition or bug fixing. We end the chapter by describing tools and tool building in the Squeak/Smalltalk environment where everything is an object and graphics are inherently tangible.
- Chapter 3 motivates our research question with an exemplary program comprehension task. A programmer has to explore a larger code base to understand a certain framework mechanism. She has an idea on how to improve debugger and code browser to better understand the artifacts at hand. We then present our working hypothesis with our solution concept on an abstract level. We end the chapter with a concrete tutorial on how to resolve the motivational challenge in Vivide.

Part II — The Vivide Programming and Tool-building Environment

- Chapter 4 describes our four main contributions. It begins with our new scripting language, followed by a new strategy to design tools. Then we propose our new UI-design language, which leads to a new data-driven working practice.
- Chapter 5 adds more details to our contributions and implementation guidelines. It begins with advanced usage of our scripting language. Then, it applies our tool-design strategy to construct script editors. After that, it expands on the interplay of scripting language and UI design, followed by further thoughts in our new working practice. We end the chapter by elaborating on some important implementation details of the Vivide environment.

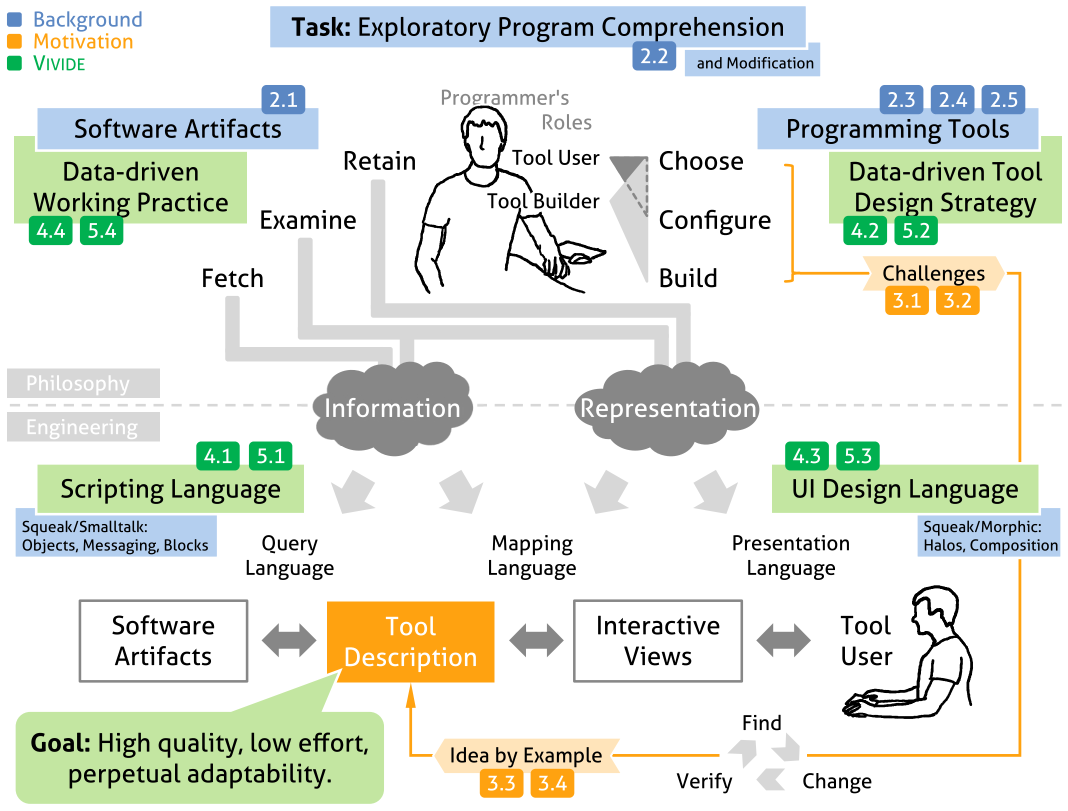


Figure 1.2: Our main line of thought and work organization. We separate two fields: (1) the upper half is more theoretical or philosophical and (2) the lower half is more practical or related to engineering. The color coding denotes background, motivation, and solution chapters. The numbers represent the respective sections in this work.

Part III — Evaluation and Discussion

- Chapter 6 contains the experiences we gathered so far by using Vivide in our daily programming work. This includes new programming tools, thoughts on code refactoring, and applicability of non-list views such as tree maps and polymetric views.
- Chapter 7 presents case studies, which were carried out by students in the course of project seminars or their master's thesis. This includes tool building for version control [195], language design [154], a new module system [184], and multi-level debugging [96].
- Chapter 8 illustrates a first attempt on evaluating the effect that Vivide has on tool building. We present a possible design for a controlled experiment and discuss insights from first pilot runs. We end the chapter with thoughts on a possible workshop to improve the experiments training phase.

Part IV — Future Directions for Programming Environments

- Chapter 9 shows related work in the fields of exploratory programming practices. This includes dynamic languages, means of visualization construction, and forms of tinker-able presentation.
- Chapter 10 concludes our thoughts by summarizing our main argument and its proposed impact. We end the chapter with a discussion about future directions for our contributions.

Philosophy and Engineering

We illustrate our main line of thought and organization in figure 1.2. There, we summarize all relevant argumentation models and simplifications of the problem domain, which we explain in the background (chapter 2) and apply in the motivation (chapter 3) as well as the solution (chapter 4, chapter 5). This includes details about programmers that fetch, examine, and retain information as tangible software artifacts in the context of program comprehension tasks. This also includes the exploratory characteristics of programmers that choose, configure, or even build graphical tools for programming in an agile setting.

Overall, we distinguish between *philosophical* and *engineering* content. First, we treat our perspective on software artifacts and programmers' habits as a possible, theoretical approach to manifest facts from the problem domain of our research question. We support our thoughts with existing research that often uses different terms for similar concepts. Second, we solve an issue on a technical level by handling the glue between data and graphics in interactive programming tools. Those contributions are tangible in the form of code artifacts and the Vivide environment itself.

In the end, we realized that the topic of program comprehension and tool building covers a vast area of concerns, which are typically discussed in isolation. In this work, we put effort in combining those concerns into a single comprehensive picture.

Notes on Presentation

Throughout this work, we complement the use of chapters, sections, paragraphs, and figures with the following "tools":

- *Gray boxes* hold additional thoughts that would interfere with the main argumentation. They are typically labeled as "Remark".
- *Source code* is presented inline between paragraphs but set in a `mono-spaced` font. Explanations will be directly before or after the respective code listing.
- *Captions of figures* try to be self-contained and hence are long enough to reveal all important aspects so that readers can browse figures and still learn about their contexts such as in figure 1.1 and figure 1.2.

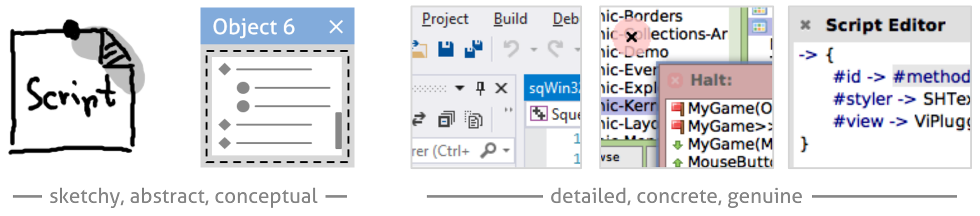


Figure 1.3: We combine sketches (left) with screenshots (right). While sketches show typically no meaningful data, screenshots represent existing systems.

- *Footnotes* are used to briefly explain additional thoughts, which do not require the attention of a gray box. This includes references to the appendix.

Throughout this work, we show and convey ideas and arguments in a graphical way. This includes *sketchy* models of user interfaces or user interactions but also *actual screenshots* from real tools and environments as shown in figure 1.3. That is, any labels or data shown in sketches do not have to be meaningful. For example, abstract charts or lists should rather illustrate themselves in the context of a possible interface. Readers can recognize sketches by the use of squiggly lines or plain gray/blue window decorations such as in figure 1.1.

Finally, there are *summaries* at the end of each (main) chapter and smaller *synopses* in gray boxes at the end of each section in that chapter.

[|]

In the next chapter, we begin with background information. We explain our notion of information as software artifacts, programming tasks, graphical tools for tangibility, tool configuration, and eventually tool building.

2 Background

The craft of programming entails challenges that go beyond reading and writing source code. In larger software projects, the mere communication with fellow programmers can yield hundreds of informational pieces to manage. For example, there are e-mails, mind maps, sketches, or chat logs. Programmers have to work with many kinds of such *software artifacts* because code is rarely (self-)documenting its current bugs, underlying theory, or prospective features.

When programmers write software, they make use of programming tools. First and foremost, there are tools for reading, writing, or debugging source code. Then, there are tools for other kinds of artifacts. If programmers want to remember bugs, they can use tools for creating bug tickets. If programmers want to schedule a meeting for discussing a refactoring, they can use tools for writing e-mails. Consequently, programming tools should be accessible, supportive, and integrated.

In this chapter, we describe our notion of software artifacts being "informational things" to be consumed and managed by programmers with tools throughout program comprehension tasks. We outline the virtues of graphical user interfaces having interactive means to exhibit underlying concepts in a tangible way while keeping cognitive effort low. We explain, why many programming tools and environments are generically aligned with programming languages but can be configured to accommodate specific domains, tasks, and personal preferences. Finally, we describe reasons for building new domain-specific tools and means to do so in the self-supporting, object-oriented environment Squeak/Smalltalk.

Remark This chapter explains relevant *vocabulary*, which we will use in the subsequent chapters. We form a *conceptual model* about the data-intensive programming profession as well as the idea of domain-specific, graphical tools in self-supporting environments. We will use this model to motivate this work's main challenge in the next chapter.

2.1 Software Artifacts

Programmers read, write, and run source code. In object-oriented programming languages, the impalpable term "source code" actually denotes *tangible* artifacts such as classes and methods. In fact, programmers create, modify, and observe many kinds of *software artifacts*, not just code-related ones [59, 177, 91]. For example, the software development process yields artifacts in the course of modeling and documentation. Architecture diagrams can be useful for program comprehension. E-mail conversations can be useful to document design decisions. When creating and maintaining complex systems, programmers handle a variety of these software artifacts. Consequently, there are many situations when *writing code* is not the primary activity.

Artifacts Under Observation: Concepts, Structure, and Transport

"Something created by humans usually for a practical purpose."
Definition of "artifact" from Merriam-Webster dictionary, March 2017.

Throughout this work, we talk about programming tools that support working with software artifacts. Any assessment of appropriate tool support requires a clear definition of the term "artifact". In the common meaning, artifacts are "things" created by humans with the intention of practical use. For software engineering, we also include things created by tools as tangible by-products. Since programming tools are operated by human programmers, it is programmers who create all software artifacts directly or indirectly.

Programmers *consume* artifacts through computers by means of *structure* revealed visually on screen. That is, programmers read labels and interpret graphics to recognize *concepts* and understand relationships. In general, programming tools provide many different interactive, graphical user interfaces to present artifacts visually. Meanwhile on a technical layer, tools manage files, objects, or tables to *transport* artifacts with the intent of persistence or integration.

So, we distinguish *three levels* when talking about software artifacts and tool support: (1) concept, (2) structure, and (3) transport. This resembles a definition for *design artifacts* known from *activity theory* [17]. There, artifacts capture knowledge from design activities, which results in the separation of conception, cooperation, and construction. We think that design artifacts and software artifacts share many properties because software *design* is a part of software engineering.

Conceptual level The artifact's underlying concept or representative of a concept matters. There is typically context information necessary to determine concepts. For example, the text "john.doe@ymail.org" can refer to a concrete person or the abstract concept of an e-mail account. Driven by the programmer's current goals, there is room for interpretation. Especially if information is ambiguous, programmers can adjust interpretations to fit the context. Tasks related to program comprehension often exhibit such properties of inconclusiveness. Tools do not form concepts themselves, but the programmers do with the help of a tool's visual output. That is, programmers have to understand many abstract rules, constraints, and assumptions that make up the software system.

Structural level The artifact's structure or "building blocks" matter. In programming, human-readable text is favorable due to its simple means to type-in

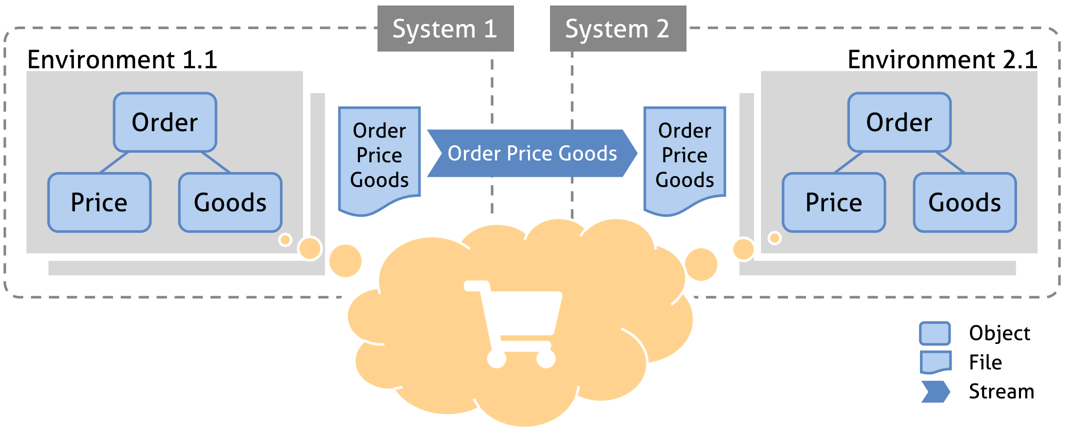


Figure 2.1: Synergy of artifact concept (here: an order), structure (here: order is price plus goods), and transport (here: objects, files, streams). The concept (here: in the bubble) is shared across all boundaries.

and print-out. More complex structures include sounds or pictures, which involve more sophisticated means to create and process. Artifacts can refer to other artifacts by name; namespaces can help resolve ambiguity. It is possible that two artifacts complement their structures by sharing a concept. For example, a bug (being the concept) can be documented in an e-mail (e-mail account) or in a ticket (bug tracker), each one revealing different details. Also, the structure of a single artifact does not necessarily identify a single concept. Especially in debugging, programmers encounter many complementary artifacts to understand a single bug.

Transport level The transport medium, which holds the artifact's structure, matters. In an object-oriented application, artifacts are represented as objects. In a relational database, artifacts are represented as tables. In a text editor, artifacts are represented as files. When treating an object, table, or file as an artifact of its own, transport media are used in combination. For example, the file can be an object, which is eventually stored in a table. A table can also be an object, which is eventually stored in a file. And so on. Consequently, tools can only exchange information via artifacts if they are able to process a common transport medium.

In figure 2.1, there is an example of a *digital shop* that processes orders on two computer systems, each having an object-oriented application environment. All domain-specific concepts are shared across all systems and environments: orders, prices, goods. When transporting artifacts between such environments (or systems), it can be necessary to serialize objects from the source environment into files, write bytes on a network stream, de-serialize contents again, and finally import the data as objects in the target environment. Such sharing across boundaries supports division of labor and modularization of large software systems.

Artifacts as Objects

We focus on object-oriented programming languages, tools, and environments. This implies that all software artifacts are represented as objects. An object encapsulates state and responds to messages. Creating artifacts means creating objects by defining structure and name to represent the underlying concept. Adding structure means referring to other, previously created, objects, which can represent other artifacts themselves.

We regard artifacts as largely state-defined, but behavioral descriptions can be used to derive new, maybe dynamically changing information. For example, a person's age is the span between today and the person's birthday. Objects can also generate more complex objects such as compilers that generate byte codes from source code. In this perspective, *everything is an object* and every object can be treated as a valuable software artifact. The compiler itself may be the artifact and not just a tool or application to use.

In addition to creation and generation, artifacts can be imported. The underlying operating system provides means for object-oriented programs to communicate with other programs. For example, this allows for accessing tickets in bug trackers that run on some servers in distant networks implemented in any programming languages running in any environments. At best, programmers can get the impression of not having boundaries at all.

Eventually, programmers have to understand, create, and modify *code* artifacts. For class-based languages [211], this means to describe classes and methods that correspond to the program's domain. Note that there is no inherent limitation in focusing on object-oriented languages only. We argue that the object-oriented programming paradigm is a good start to think about software artifacts and tools that handle those artifacts.

From Theory to Practice: Relevance and Availability

The variety of software artifacts can be overwhelming to programmers. Yet, there is no need to always consider *all* artifacts in well-modularized, large systems. Cohesive modules with low coupling to other parts can be treated in isolation, following the idea of abstraction [5, pp. 4–31] and information hiding [119, p. 25]. In addition to such *objective* measures, programmers differ in their needs for information, largely based on individual knowledge and experience. That is, they do assess artifact relevance in a *subjective* manner, too. This is especially important since multiple artifacts can share a concept while complementing structural information, which might be already familiar to the programmer. Programming tools can make both object and subject measures accessible.

The theory on *information foraging* has been applied to programming tasks [55]. In that case, constructs such as links, cues, attention, cost, and value evaluate the level of tool support. We simplify that theory and focus on a few data-oriented questions programmers typically have about software artifacts:

- Is the artifact required to run the program?
  Is it *source code*?
- Is the artifact expected to be maintained in the software development process?
  Is it *documentation*?
- Is it feasible to access the artifact in time?
  What are the *estimated costs*?
- Is it worth processing the artifact given the current knowledge?
  What is the *estimated value*?
- Is the artifact representing information about the system in its current form?
  Is it *up to date*?

We do not consider this list to be complete. Yet, we do emphasize the first question, namely the relevance of *code artifacts*. In a space of countless optional artifacts, source code is *crucial* and affects programming tasks directly. This includes the awareness of duplicate code artifacts and other *code smells* [57, pp. 75–88] worth investigating and refactoring. Overall, programmers use tools to assess artifact relevance by exploring artifact structure. We will put focus on the individual programmer that works on a concrete problem, is allowed to follow hunches, and makes informed decisions about what software artifacts to consider and which tools to use.

Remark We use the term "programming environment". The conventional abbreviation "IDE" means "integrated development environment". Since we refer to programmers and programming while not using developers and development, we will also refrain from talking about IDEs. Instead, we focus on "programming environments", which are always *integrated* to some degree.

Since resources are limited, it is likely that relevant software artifacts spread across several environments. For object-oriented environments, this means that programmers and tools have to integrate more *transport media* than just objects. While objects are primary, there are often files, streams, or tables, too, which depends on the artifact's location. For example, there can be source code in a file attached to an e-mail, which happens usually outside one programming environment. The programmer then has to *load that code file* to make its contents accessible in the preferred tools.

Integrated programming environments such as Eclipse provide a *shared data representation* to support the exchange of artifacts among tools *inside* the environment. Still, there are many tools involved outside the environment such as on-line bug trackers. Transferring artifacts from one environment into another involves switching the transport medium as illustrated in figure 2.1. Note that some object-oriented tools give programmers more control over files than others. In Eclipse, programmers have to actively manage files and file contents to store source code into. In Squeak,

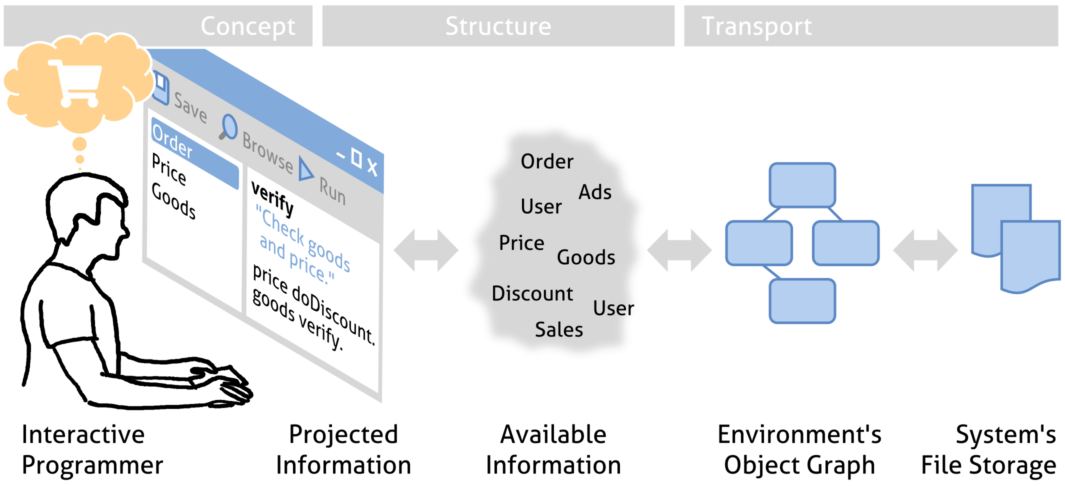


Figure 2.2: The interface of a programming tool supports programmers to recognize the concept and structure of software artifacts. There is usually more structural information of artifacts available than currently visible on the screen. In this example, the transport medium (object/file) remains hidden from the user's perception.

programmers just describe classes as objects and there is only a single file to store information into. So, when we talk about object-oriented programming environments and tools, it is obvious that there are boundaries to other environments. In daily work, the use of multiple transport media is inevitable.

Virtually Uniform Transport

Programming tools try to *hide* an artifact's transport medium. Primarily, tools show the *structure* of software artifacts on the screen as depicted in figure 2.2. In the context of program comprehension, programmers use *visual cues* such as the name of a class or the description of a bug to (1) to identify an artifact and (2) to gather more information about an artifact to accomplish the task at hand. Whenever a new class is created, the underlying object, file, or stream will usually be created automatically.

When fetching artifacts from outside the environment, tool interfaces request merely abstract location and identification information. For example, programmers have to provide only the address of the bug tracker whose tickets should be made available. The actual fetching with the help of transient files or data streams remains hidden. Having to deal with the transport medium in addition to structural information and underlying concepts requires additional effort. In case of an error, however, programmers may still have to deal with that. A tool's abstraction can break, for example, if the network connection is lost or if the hard disk is full. Then, a rather distant artifact that felt quite local will suddenly be exposed.

There are many programming environments that, intentionally, expose files to programmers. For historical reasons, these environments provide graphical interfaces to file-based tools and hence introduce a mixture of objects and files. As an advantage, programmers used to working with files can benefit from a shallow learning curve since there are only few additional concepts. Examples include Unix-like operating systems based on files, which started with text-based, command-line interfaces (such as the Unix shell) and got graphical interfaces only later in the form of DEC Fuse, HP SoftBench, or SUN SparcWorks.

Cross-platform compatibility can make it still necessary for programmers to actively manage files in addition to objects. In Microsoft's Windows operating system, there are objects in the first place and files seem to be a mere addition [165, pp. 21–22]. Unfortunately, such a dualism exposed to the programmer increases management efforts. Files have to be named and organized in folders, for example. Restrictions can help such as a one-to-one mapping from an artifact's object to its file. Java's official code conventions dictate a single public class per source file.[1] So that programmers can, to some extent, treat files and code artifacts interchangeably. Private classes, as one limitation, are not covered by such conventions. Hence, flexibility and ambiguity remain, which can be misused and impede program comprehension tasks when inconsistently supported in tools.

Synopsis We think that good programming tools support programmers in focusing on the structure and concepts of the software artifacts at hand. Goals and working strategies can be expressed in terms of acquiring more information and understanding about artifacts. Since the number of available artifacts can be overwhelming, programmers have to assess relevance according to several dimensions such as actuality and added value. Besides consumption, programmers also create, modify, or remove artifacts. When talking about tool architecture and implementation, the transport medium is very important. When talking about a tool's user interface, the (graphical) representation of artifact structure on screen matters because it reveals artifact concepts to the tool's user.

[1] A Java code convention reads: "Each Java source file contains a single public class or interface. When private classes and interfaces are associated with a public class, you can put them in the same source file as the public class. The public class should be the first class or interface in the file." (See `http://www.oracle.com/technetwork/java/codeconventions-141855.html`, accessed 2018-11-24)

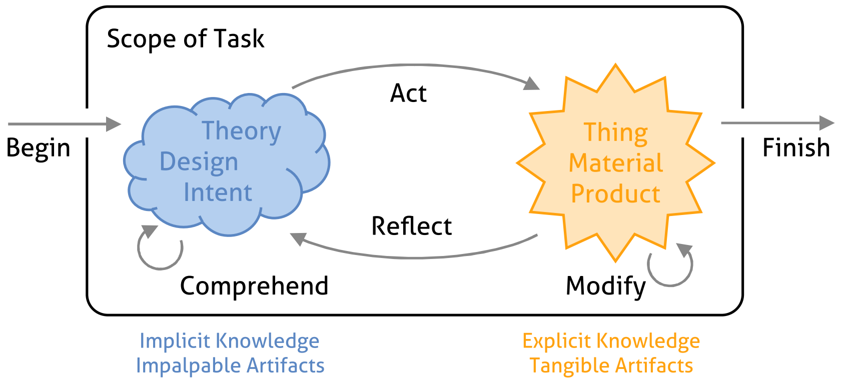


Figure 2.3: Program comprehension is an iterative, data-driven process. Programmers strive to understand a system's design and intents (left) before making changes. Analysis of the results (right) might reveal false assumptions about the actual role of some system module. Then, programmers go back, reflect, and revise their mental model. There are many different strategies to process software artifacts in this process.

2.2 Programmers' Tasks

Many programming tasks have the intent to fix known bugs or add new features [101, 100]. To fulfill this intent, programmers invest much time to navigate, explore, and eventually locate all relevant places in the code [178]. Then, they have to figure out which code to add, change, or delete. In this course of localizing and modifying, programmers build a *mental model* of the system's current design [133]. After some time, they will form a whole *theory* that supports making the right decisions in future tasks [130]. In a larger system, there is usually a *task context* and hence many parts to recall or learn about. Programming tools can facilitate stepwise learning and recalling in the whole software maintenance process.

Strategies for Program Comprehension

We know about many different program comprehension strategies [113, 95, 161], which suggest different kinds of tool support in combination. For example, programmers can start with the topmost modules to understand the overall architecture. Then, they dive down into the details of modules that might be relevant for the current task. Sometimes, it will be the other way around, especially if programmers can immediately recall a specific place to start with. Such ideas can be specific names to search for to directly navigate to code artifacts. Overall, program comprehension is an iterative process with backtracking involved. It includes learning new facts about the system and also recalling facts that were known from previous tasks as depicted in figure 2.3. Software systems are usually too big to memorize every single detail and bring back to mind all relevant facts when needed.

During program comprehension, there will be a gradual transition from code reading to code writing, sometimes even unnoticed. Once the problem is understood to some degree, programmers might have an idea on how to solve it. They write some code and begin to analyze the effects their changes have on the system. They might have to revise some assumptions and correct their mental model. If they are new to a system, they need to invest time and energy to build up an appropriate theory [130]. They might have to go back to figure out[2] the intent behind the design of relevant system modules. As an effect, programmers write code not only to fix the bug but also to better understand the system.

There are other artifacts that support program comprehension. It is not just about source code but also about comprehending associated artifacts such as libraries, documentation, and e-mails. Programmers use various programming tools to support this process of data-driven program comprehension.

Fetch, Examine, and Retain Information

It is not just computers but also programmers who constantly have to process information. As described above, information is manifested in numerous, partially wide-spread, software artifacts. There are many tools that support programmers to work with artifacts to build up a mental model and theory. Tools help foster informed decisions. Tool-agnostic program comprehension models [113], however, are not suited to describe and assess any tool's value in the process. For this, it is beneficial to pinpoint specific but recurrent steps such as "query a database" and "ask a question". Hence, such models cannot conclude the need for tools that help "browse source code". What does it even mean to "browse"? Programmers seem to browse information all the time in various ways.

Tilley et al. [202] distinguish tools for data gathering, knowledge organization, and information exploration. Ko et al. [95] apply an information foraging theory[3] and distinguish programmers that seek, relate, and collect information. Sillito et al. [177] categorize typical program comprehension questions into finding and expanding focus points or groups of focus points, which simplifies tool requirements into a different triad: asking questions, maintaining context, and piecing information together. We take these three perspectives into account and clarify our problem domain.

What do programmers do to understand systems, issues, or theories in the scope of certain domains or tasks? As illustrated in figure 2.4, we assume that programmers use tools to (1) fetch artifacts, (2) examine artifacts, and (3) retain artifacts. In each activity, programmers may focus on an artifact's concepts, structure, or transport

[2]If fellow programmers are available and able to explain important aspects of the theory, there might be no need for tools. However, we assume that it is possible for a programmer to build up some coherent model by herself using tools.

[3]Fleming et al. [55] elaborate on the relevance of information foraging theory for programming tools including explanatory background information on the theory itself.

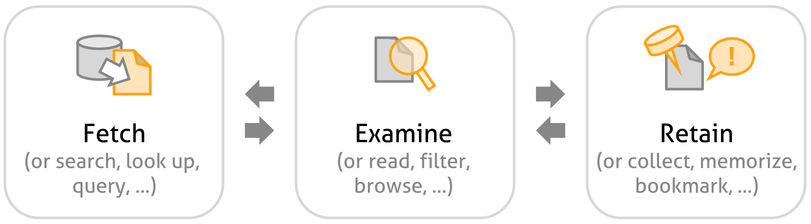

Fetch
(or search, look up, query, ...)
Examine
(or read, filter, browse, ...)
Retain
(or collect, memorize, bookmark, ...)

Figure 2.4: During program comprehension, programmers use tools to continuously fetch and examine software artifacts to learn and recall facts about the system. If considered relevant, artifacts are retained on screen or in mind. The whole process is iterative, flexible, and sometimes unanticipated when suddenly having an idea.

properties. Tools may have one major use case, but they typically support multiple activities. Especially the distinction between fetching and examining artifacts can be blurry. Nevertheless, such ambiguity is of minor concern because the concreteness of these steps in the program comprehension process will help us to describe and assess existing tools. We can pinpoint problems and show how to design new, versatile programming systems.

Fetch

"To go or come after and bring or take back."
Definition from Merriam-Webster dictionary, March 2017.

If programmers do not have enough information at hand, they can get more by fetching more artifacts. Tools use different vocabulary depending on the context. For example, programmers can *access* an external database, *jump to* the list of all symbol references, *seek* advice in a bug tracker, *search* a term across all project files, *navigate to* all code changes from a coworker, *query* a complex aggregation, *look up* the result of an expression evaluated in the command line, or simply *ask* a question. Embedded in the iterative, ongoing program comprehension process, programmers use tools to add more artifacts to the current working set. While this process may look like mere consumption via a tool's user interface, there can be new artifacts produced as a side effect of interactivity. For example, fetching the call tree of a code snippet or test case may actually run the code and produce loads of artifacts, that is, call nodes [146] If caching is used, a re-fetching might not have such a side effect over time but programmers may neither know nor notice.

Examine

"To inspect closely."
Definition from Merriam-Webster dictionary, March 2017.

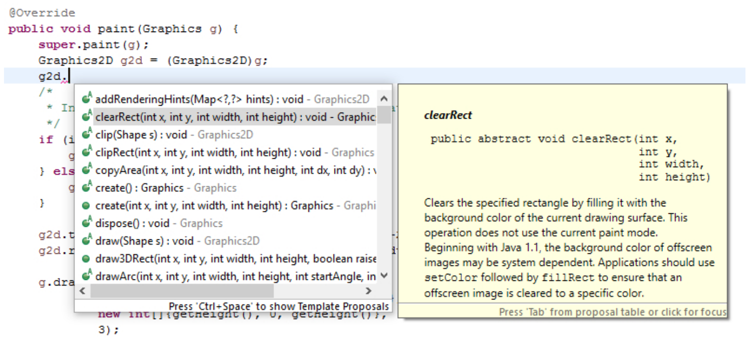
There is usually more information available than what fits on screen. For example, the result list of a search operation can be very extensive. To find out about the relevant pieces in such data sets, programmers use interactive widgets and visualizations. The visual information seeking mantra [174] outlines this activity as *overview-first*, *zoom-and-filter*, and *details-on-demand*. In general, tools use also different vocabulary depending on the context. Programmers can *read* the log output, *follow* references in code back-and-forth, *zoom into* a graph, *filter* a list by a keyword, *browse* files' contents one-by-one, or *jump to* the next match on a Website. To separate "fetch" from "examine", we think that programmers examining some artifacts do usually not intend to add more artifacts to the current working set. With "more artifacts", we mean artifacts on the same abstraction level. For example in Squeak, there will be objects created and removed all the time, even if you move the mouse cursor. While you can treat every object as a software artifact with important structural and conceptual properties, the distinction becomes obvious considering the programmer's current task and problem domain.

Retain

"To keep in possession or use; to keep in mind or memory."
Definition from Merriam-Webster dictionary, March 2017.

Programmers can always memorize information or take notes with a physical pen and a real sheet of paper. Considering programming tools, we focus on the means of a user interface to collect and retain a valuable representation of software artifacts that reduces the cognitive load. There are many tools for organizing textual notes with screenshots and other multimedia content. Examples include OneNote and Evernote.[4] As usual, tools use different vocabulary for retaining insights depending on the context. Programmers can *arrange* windows to compare things side-by-side, *check off* an item on a todo-list, *bookmark* a Website, or *collect* code snippets to document a cross-cutting concern [198]. All software artifacts collected to retain insights can be used as input to fetch new artifacts or re-examine the existing ones. Programmers may remember additional information while examining artifacts. Tools can support this process by actual side-by-side comparison, for example. In fact, programmers do usually have multiple tasks in the works [66]. One task can block another, for example, if more information is required before the programmer knows how to proceed [142, 91]. When switching between these tasks, programming tools should help recall the current progress in the form of open questions or confirmed assumptions.

[4]Both Microsoft OneNote (`https://onenote.com/`, accessed 2018-11-24) and Evernote (`https://evernote.com/`, accessed 2018-11-24) support versioning and sharing using proprietary server infrastructure. Programmers can hence access their notes from everywhere, which improves basic integration of these artifacts in the software engineering process.



```
@Override
public void paint(Graphics g) {
    super.paint(g);
    Graphics2D g2d = (Graphics2D)g;
    g2d.
    /*
     * In
     */
    if (i
        g
    } els
        g
    }

    g2d.t
    g2d.r

    g.dra
            new int[]{getHeight(), 0, getHeight()},
            3);
```

- addRenderingHints(Map<?,?> hints) : void - Graphics2D
- clearRect(int x, int y, int width, int height) : void - Graphic
- clip(Shape s) : void - Graphics2D
- clipRect(int x, int y, int width, int height) : void - Graphics
- copyArea(int x, int y, int width, int height, int dx, int dy) : v
- create() : Graphics - Graphics
- create(int x, int y, int width, int height) : Graphics - Graphi
- dispose() : void - Graphics
- draw(Shape s) : void - Graphics2D
- draw3DRect(int x, int y, int width, int height, boolean raise
- drawArc(int x, int y, int width, int height, int startAngle, in

Press 'Ctrl+Space' to show Template Proposals

clearRect

```
public abstract void clearRect(int x,
                               int y,
                               int width,
                               int height)
```

Clears the specified rectangle by filling it with the background color of the current drawing surface. This operation does not use the current paint mode. Beginning with Java 1.1, the background color of offscreen images may be system dependent. Applications should use `setColor` followed by `fillRect` to ensure that an offscreen image is cleared to a specific color.

Press 'Tab' from proposal table or click for focus

Figure 2.5: Even code *writing* activities are supported with tools that help *comprehend* the programming context. For example, syntax highlighting can give feedback about spelling mistakes. Auto-completion helps recall and learn about library functionality. (This screenshot was taken from Eclipse 4.5 “Mars”.)

How Programmers *Create* Information

In the course of program comprehension, software artifacts are not only consumed but can also be created, modified, or removed. That is, comprehension and change go hand in hand. Note that we do not mean to investigate elaborate ways to modify or transform program source code such as refactoring tools do when renaming a method, for example, in the entire code base. Instead, we want to address the means of user interfaces to support program comprehension:

- How do tools support the procedure of fetching, examining, and retaining information through artifacts throughout a task?
- How can programmers understand whether they found an answer to a question or some evidence for an assumption?

That is, the mere act of *a user typing a thought into a text field* is not the primary concern in this work. In many—if not most—programming tasks, programmers have to enter search terms, express new code in text, or save modified files. Even a simple search request results in new, maybe transient, artifacts. Thus, we acknowledge the fact that tools for program comprehension can involve aspects of artifact change intrinsically. In the field of software engineering, elaborate information creation or extraction techniques yield their own challenges. For example, the construction of efficient search algorithms [71] or run-time tracers [35] are out of scope for this work. We treat those as given, convenient means to fetch more and examine useful information through programming tools.

Since many programmers *modify* code to better understand it, we see a seamless combination of code reading and writing in the entire programming task. For example, programmers set breakpoints to test reachability or modify properties of a class, which aims at observing change effects at run-time. Eventually, some code artifacts have to be modified to fix the bug or add the feature. In object-oriented systems, programmers will create new classes, change methods, or add tests to achieve this. Tools support this *text-driven* code writing activity in various ways. For example in text editors, programmers can rely on highlighting for language syntax, auto-completion for identifiers, and code templates for common patterns. As illustrated in figure 2.5, such tools will integrate much information to help recall or learn facts about the used system parts. Albeit these tools are triggered during code writing, they support program comprehension activities as well. Thus, up to the last moment in a particular task, programmers use tools that *support comprehension* of the program and the process. Even when committing all changes to a shared code repository, tools can remind programmers about the artifacts changed, whether all tests pass, or to enter a descriptive message.

In addition to code-related artifacts, programmers leave more *informational traces*. Programmers might stumble upon new bugs or have new ideas on how to improve code readability or system usability. All these thoughts and insights usually get captured in additional notes or tickets in the project's bug tracker. In larger software projects, team communication yields many artifacts. Any such artifact can become relevant in future programming tasks: every question formed as an e-mail or instant message, every wiki page that persists meeting minutes can document a relevant insight. At some point, some of these artifacts might be *fetched* and *examined* to *retain* or recall some aspect that might turn out to be causal for making the final code change.

Information Overload, Tool Overload

Programmers use tools that support program comprehension because there is typically too much information to memorize and recall in time. There is often also new information to process and learn about because multiple programmers can work on the same project but independently. This includes all kinds of software artifacts such as code, e-mails, and project documentation. Not all information is important for every programming task, but some are. For example, coding guidelines have to be remembered and applied all the time. Any particular pattern in a specific system module may only be relevant if the programmer is working on that module.

Remark Our thoughts focus on a *single* programmer that is confronted with an *overwhelming* amount of information and using tools. In larger systems where multiple programmers are involved, changes can happen without being notified. For tool support, we see basically no difference in whether an artifact is unknown because of its novelty or former irrelevance.

We are aware of several *models* to better understand cognitive processes in the programmer's mind—and humans in general [122, 142]. Basically, these models try to clarify when there is a need for re-fetching or re-examining software artifacts to recall relevant information. On the one hand, an *externalized note* visible on screen that retains an information can remain there—unless being covered by another tool window. A *memorized note*, on the other hand, can be forgotten until recalled *by chance* or with the help of visual cues in the tool. This basic understanding is sufficient to distinguish programmers that can memorize much information from the ones that have to rely on tool support more intensively. Still, some cognitive limitations for processing information are shared among all human beings [120]. That is why tools do not only have to support specific domains or tasks, but they can also be tailored to individual programmers.

Tools are plentiful and programmers have to choose. Personal habits and the software development process can influence such choice. For example like all humans, programmers are also prone to the *active user paradox* [28]. It is basically the irrational behavior of users to not research a tool's best practices but instead just get started somehow and knowingly reject probable efficiency. Additionally, process limitations might restrict the free use of tools and forbid to integrate a new one into the controlled environment. In this work, we assume that programmers are free to embrace any chance to accomplish tasks more efficiently and more effectively.

Cognition and Concurrent Tasks

Whatever programmers can memorize and recall in sum, the human brain is capable of storing different kinds of memory with respect to *access time*. Miyata et al. [122] distinguish *long term*, *short term*, and *working memory*. Long term memory is able to store much information for a long time such as a software project's theory and domain concepts. Short term and working memory are quite limited and rather useful when thinking about the current programming task and its relevant software artifacts. Parnin et al. [142] talk about a more elaborate model and distinguish five different types. The *attentive memory* is used for focusing and filtering the information required for the current task. The *prospective memory* broadens the scope to what might be missing and worth fetching as next step. The *associative memory* keeps track of all the current steps in the task and might hence be useful for backtracking and sequential processing of assumptions or hypotheses. It is closely related to the *episodic memory*, which is used to recall past events. Finally, there is *conceptual memory* to grasp the whole picture, concepts, and theory of the system or system part.

Miyata et al. [122] also distinguish two kinds of control systems: conscious and subconscious. They explain why humans can only perform one conscious activity at a time. Other activities have to be learned, trained, and automated to be executed at the same time in the subconscious system. Examples include breathing or blind typing. Consequently, a writer that cannot type blindly on a keyboard will rather sketch her thoughts via pen-and-paper first, given that writing by hand does not pose a challenge itself. If so, recording a dictation could be an option while relying on speech and hearing. There are similar models such as Kahneman's "System 1" and "System 2" [83], which state the same limitation. Transferring this insight, programmers can only attend to one program comprehension task at a time. There are several reasons for not completing but *suspending* one task for another [66, 91]. Incoming notifications might draw the programmer's attention such as new e-mails and chat messages. An upcoming question or hypothesis might also require more information to be fetched. For example, a question like "Which type represents this domain concept or this UI element or action?" [177] can trigger a new debugging session that demands for the programmer's full attention. Such task switches involve suspension and resumption lags and thus yield several cognitive challenges [101, 8, 143]. If tools lack support for such interruptions, programmers will have to take additional notes to not forget about that suspended task and to speed up the resumption phase.

Exploratory Programming

Given that task concurrency is inevitable, we do encourage *exploratory programming* [171, 167, 204], where programmers are allowed to make unrelated changes along the way as they see fit. This can include refactorings and bug fixes that do not take much time. Especially when the project's domain and requirements are unclear, officially allowed flexibility can be valuable for programming efficiency. Such "side quests", however, also draw the programmer's attention and hence force the suspension of the current (main) task. There are tools that support logging [186] and untangling [195] of additional tasks to improve version control for such exploratory habits.

Programming tools support concurrent work on multiple tasks to some degree by offering flexible means to fetch, examine, and retain artifacts. Programmers can open the same tool, such as the code browser, multiple times and use it for different tasks. By keeping the tool windows open, tool switching can simply mean task switching. Typically, tool containers offer flexible layout strategies to group related information coherently on screen. As described above, there are many tools for taking notes to improve suspension and resumption lags. Such integrated tools can often be combined with any external tool through manual (or "off-line") integration. In the next sections, we describe ways to control the level of such support via *tool choosing* and *tool configuration*.

Synopsis Program comprehension is an integral part of a programmer's daily work. Especially in larger software systems, programmers use programming tools to comprehend the details relevant for task and domain. Such tools help cope with the fact that a programmer's mental capacity has limitations such as being restricted to perform only one conscious activity at a time. Tools have user interfaces that reduce cognitive effort. For program comprehension this means to conveniently *fetch* software artifacts, help *examine* the artifacts' details, and to *retain* insights somehow. We simplify program comprehension to these three steps to help explain challenges in existing tools and hence the need for tool configuration and tool building.

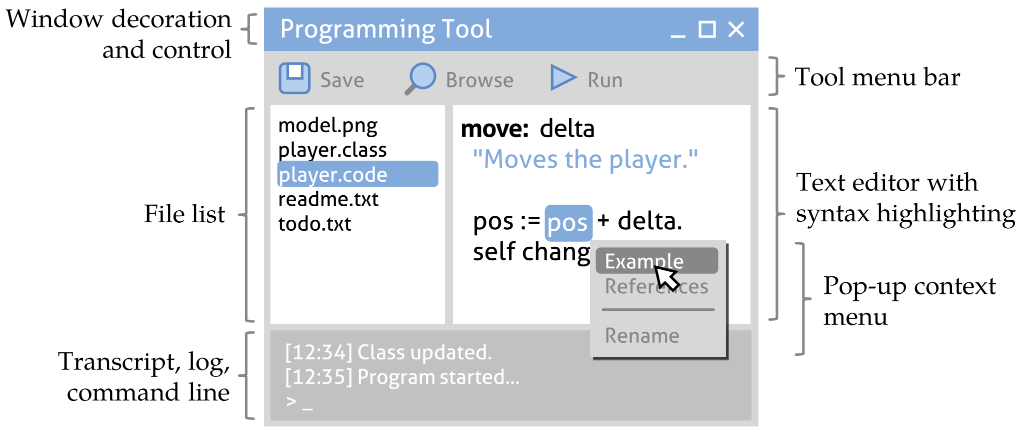
Tools have a main purpose but are not exclusive in many features. Some of them are good at fetching information, others focus on examining details of information that are already at hand. From user perspective, the distinction between *fetching* and *examining* can be difficult because revealing some details might "feel" like fetching more artifacts, even though it may technically not be.

The interplay between large amounts of software artifacts and existing cognitive limitations reinforces the necessity of having programming tools that accommodate specific domains, tasks, and individual traits. We acknowledge the fact that programmers might recall any relevant detail at some point in time without tool support. In the end, programmers do have to come up with a solution for that particular bug or the best way to add that particular feature to the system.

2.3 Choose Tools: Graphics for Tangibility

Modern computers support high-resolution input and output devices. Many research efforts from the past 40 years have been added into mainstream programming environments. Once limited to characters and text, modern programming tools can employ colorful shapes and elaborate widgets, placed on a screen having millions of pixels. Once started with mouse [50] and keyboard, programmers can employ gestures and shortcuts also via touch [203] or gaze [201]. While the field of *software visualization* [44, 99] explores different ways to present thousands of artifacts concisely on screen with colors, patterns, and shapes, *plain text* still plays a vital part in graphical user interfaces such as for button labels and menu entries—and for displaying the source code itself. While touch control and gaze input have their scenarios, mouse and keyboard are still quite reliable and precise input methods [166].

Programming tools combine all these means to help programmers (1) *view* structural properties of software artifacts on screen and (2) *interact* with software artifacts by modifying these properties. Figure 2.6 shows a simplified version of a source code editor. Figure 2.7 shows a simplified version of a programming tool that has *non-text* views. In combination, tools can support the many challenges posed by complex programming tasks. Yet, graphical tools for programming are plentiful and programmers have to choose. While there might be one favorite code editor, the right choice of tools for *fetching*, *examining*, and *retaining* software artifacts may change from task to task.



```
move: delta
  "Moves the player."

  pos := pos + delta.
  self chang
```

```
[12:34] Class updated.
[12:35] Program started...
> _
```

Figure 2.6: Sketch of a programming tool with graphical interface. Its single window has three buttons, a list of files, a text editor, and an interactive command line. The user just invoked a context menu on a text selection in the editor. Buttons and menus can open other tools.

Tangibility of Digital Information

"Capable of being precisely identified or realized by the mind."
Definition of "tangible" from Merriam-Webster dictionary, March 2017.

Programmers have to relate on-screen information with the structural properties of artifacts they know. Is it a class? A method? An e-mail? Or part of the latest meeting minutes? Then, programmers look at several artifacts in combination to examine their relationships. They assess the relevance for the current task. For this, they access the *conceptual level* of the respective artifacts. Does it belong to the cause of the bug X? Does it affect the implementation of the new feature Y? If such questions can be answered, software artifacts become *tangible*.

In programming, the notion of tangibility does not address primarily a *physical* representation but an aspect of *cognition*. Programming tools can make software artifacts be "capable of being precisely identified or realized by the mind" of the user. The idea of *tangibility* is influenced by the project domain, the current task, and personal preferences, experiences, or knowledge. Any tool's views can try to accommodate such requirements in different ways as depicted in figure 2.8. There is an extensive range of views including plain text boxes on the one end and elaborate tree maps [175] on the other end. While programmers might handle all details of *selected* artifacts, there are views that can concisely highlight task relevance even among *thousands* of them. Such views should help answer programmers' questions such as "How is domain knowledge represented in source code?" and "How are the rather abstract descriptions from the source code put into action during program execution?" Sillito et al. extracted a catalog of 44 questions from program comprehension activities [177], which can be used to create valuable graphical tools. The notion of tangibility can be used to separate good tools from bad ones (or

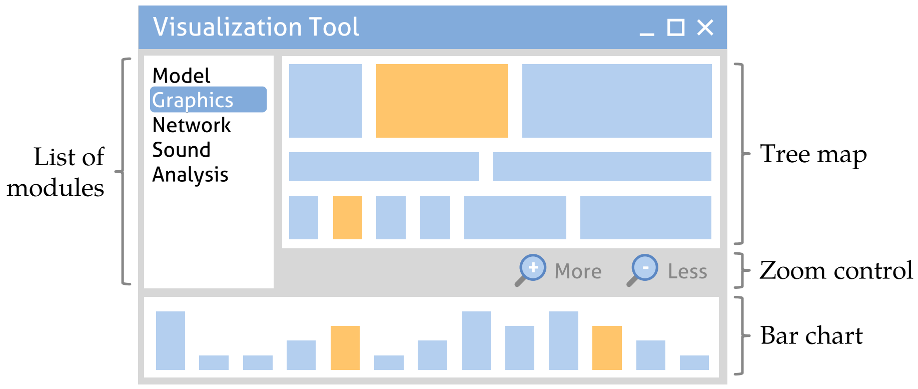


Figure 2.7: A programming tool that visualizes some modules of the software system. It provides some familiar controls compared to the code writing tool in figure 2.6. It also has a bar chart to provide a different perspective on the selected artifacts (here: "Graphics").

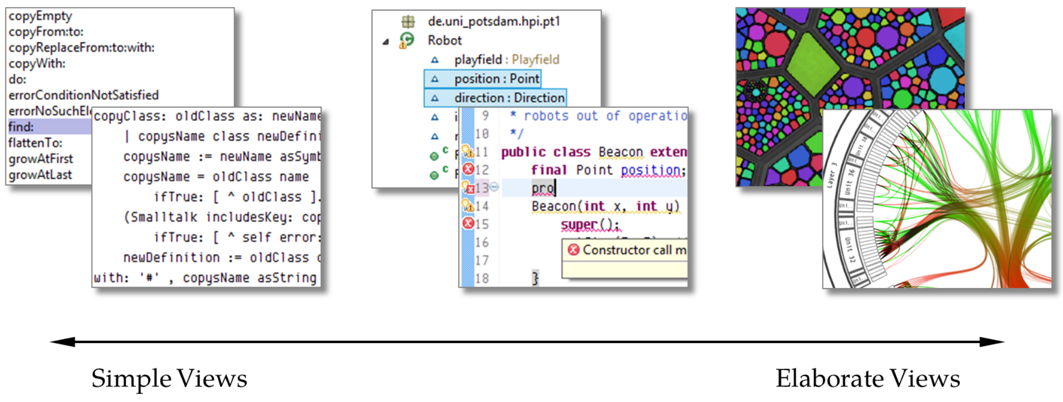
appropriate from inappropriate). That is, programmers can use these characteristics to pick the best tool, or view of a tool, for the job.

Graphical Tools can raise the chance of tangibility if they employ consistent, clear *representations* of the artifacts' characteristic *properties*. For example, a distinct pictogram for classes or methods can help reveal inheritance relationships at a glance. The artifact's unique name can also support programmers to locate its position on the screen. Besides such structural cues, user interaction can show *operations* in menus, which do typically pop up on a mouse click as depicted in figure 2.9. Seeing familiar operations can also help programmers recognize context and thus identify artifacts.

In fact, letting users point to objects first and only then perform an action *feels more direct* than vice versa. Jef Raskin, a usability expert and former member of Apple's Macintosh team, explains this distinction [152, pp. 59–62] using the example of a font change for a paragraph in a word processor as follows:

> "[...] The interface can allow you to sequence the operations in two ways. You choose either (1) the verb (change font) first and then select the noun (the paragraph) to which the verb should apply or (2) the noun first and then apply the verb. [...]"

Of course, this works especially well for a small set of artifacts. In contrast, pixel-based drawing applications require users to first select a brush and only then paint on pixels. It would be cumbersome for users to select the pixels first. Vector-based drawing applications, on the other hand, do employ both interactions styles: noun-verb and verb-noun. In Microsoft's PowerPoint, which is a slide-show design tool, there is a brush to copy the last formatting change over to other shapes. This interaction is verb-noun and complements the primary noun-verb style. Such *batch* operations can save interactions and thus time.



```
copyClass: oldClass as: newName
    | copysName class newDefini
    copysName := newName asSymb
    copysName = oldClass name
        ifTrue: [ ^ oldClass ].
    (Smalltalk includesKey: cop
        ifTrue: [ ^ self error:
    newDefinition := oldClass c
with: '#' , copysName asString
```

```
 * robots out of operatio
 */
public class Beacon exten
    final Point position;
    pro
    Beacon(int x, int y)
        super();

    }
```

Figure 2.8: There is a range of ways to show source code artifacts. Simple views (left) may just position text fragments in a meaningful way on the screen. The use of colors and pictographs (middle) can add semantics and provide various cues. Elaborate views (right) such as Voronoi Treemaps [70] and Bundle Views [34] can efficiently combine many properties for larger amounts of artifacts.

Direct Manipulation

Programming tools with interfaces that emphasize noun-verb interaction entail a *feeling of directness*. Software artifacts can become discoverable in any kind of elaborate view. Discoverability and also learnability can then positively affect program comprehension. Larry Tesler [199] coined the term "direct-drive user interface" in 1983, which later became "direct manipulation", when describing the potential impact of interactivity seen in contemporary Smalltalk systems. According to Hutchins et al. [75] and Shneiderman et al. [176, p. 214], the basic principles of direct-manipulation interfaces are:

> "1. Continuous representation of the objects and actions of interest with meaningful visual metaphors. 2. Physical actions or presses of labeled buttons, instead of complex syntax. 3. Rapid, incremental, reversible actions whose effects on the objects of interest are visible immediately."

Even though many programming tools are centered around text, they often expose the *source code as structured artifacts*. Such artifacts emerge from programming language syntax and semantics. There are, for example, lists of files, classes, or methods that belong to a software project written in a class-based, object-oriented language such as Java or Smalltalk. In combination with pop-up context menus, we clearly see characteristics of direct manipulation in graphical tools for programming.

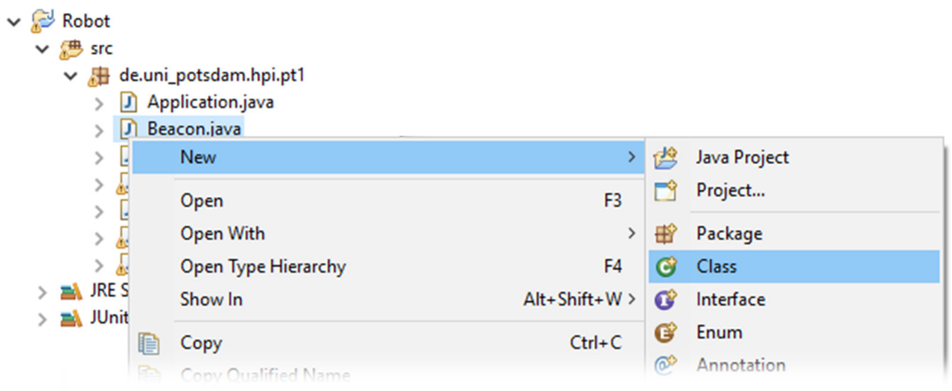


Figure 2.9: Programming tools employ concise means, such as distinctive pictograms and text, to expose software artifacts on screen. Context menus offer possible actions, which further supports discoverability and learnability. Such means of direct manipulation help programmers efficiently work with artifacts. (The screenshot shows Eclipse 4.5 “Mars”.)

In this work, we also draw lessons from *object-oriented user interfaces*. In his book “Designing Object-oriented User Interfaces” [31, pp. 89–91], Dave Collins describes a set of characteristics for such interfaces:

> “1. Users perceive and act on objects. [...] 2. Users can classify objects based on how they behave. [...] 3. In the context of what users are trying to accomplish, all the interface objects fit together into a coherent overall representation. [...]”

Many long-living principles stem from the urge to create user-friendly software for *everybody* such as word processors and e-mail clients. Domain artifacts in such applications do typically have real-world counterparts such as physical paper and a real postal service respectively. In contrast, software artifacts are often *purely virtual*. Still, programmers benefit from being able to recognize software artifacts and their relationships on screen. “Things” get manifested in mind as part of a mental model. These “things” bring new concepts to life and programmers have to work with them through programming tools. The absence of physical companions does not matter. For a further reading in this regard, we refer to the book “Software Development and Reality Construction” and especially the chapter “Software Tools in a Programming Workshop” [23], which adds more philosophical background information.

Tools and Tool Environments

There are numerous tools for writing code, managing bugs, communicating with fellow programmers, or finding a clue among thousands of software artifacts. Many tools are tailored to standard processes and artifact structures. For example, editors for object-oriented code are likely to provide lists for browsing classes and methods. Yet, there are more sophisticated tools that show strength when analyzing artifacts in-depth. For example, programmers might want to examine source code considering authorship, performance, contracts, and other properties. There comprehensive tools such as PathTools [145] that integrate many artifacts to create a guided debugging experience. There might be not only generic but also domain-specific tools available in a project to be used in domain-specific tasks. For example, the personal history of past tool activities can be available such as in command-line interfaces [69, pp. 65–83]. Also, fellow programmers might have created tools to solve recurrent problems in that domain [180].

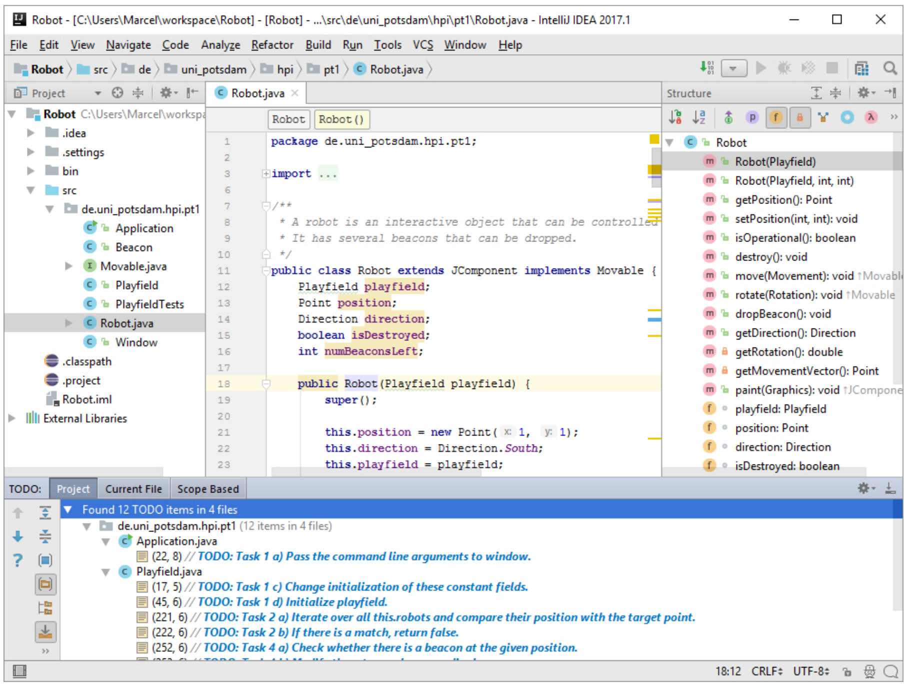
Programmers choose among such tools to *fetch*, *examine*, and *retain* software artifacts. They do not have to use the same tool for all purposes but can switch, for example, during artifact examination to get different impressions of the same information. Depending on the current task or domain, some tools might render artifacts in a more tangible way than others. Such a multi-tool approach works if tools use a shared transport medium such as files and objects with a common format. Considering file formats, there is the *JavaScript Object Notation*, JSON for short, and the *Extended Markup Language*, XML for short, to share structured information among tools [135]. Tools can derive a simple object representation from these file-based structures if needed.

When it comes to retaining and recalling information, programmers are likely to keep using tools with familiar views. They might end up using always the same set of tools and totally refrain from choosing more appropriate ones [28]. Having to learn new user interfaces takes time, even though shared concepts can ease this challenge. Such concepts include keyboard shortcuts for cut-copy-paste and further means of interaction such as pop-up menus when clicking the right mouse button. We see a strong relationship between interface *familiarity and tangibility* of the artifacts shown.

Not every programmer has to assemble a whole set of tools. Like for craftsmen or mechanics, there are existing *tool boxes* tailored to *standard scenarios*. Such boxes contain screwdrivers, wrenches, or hammers in many different sizes. The tools remain close at your working site, ready to be used if needed. Their mere presence suggests that there is a pretty good chance that you will need them some day. That is, other people—probably experts—have thoughtfully assembled that collection and novices can discover and apply it conveniently.

Considering the craft of programming, there is usually a basic set of tools for browsing, editing, and debugging source code. Novice programmers, or programmers new to a certain language, might not know which tools to use as best practice. We call such tool boxes *programming environments*. They are also referred to as integrated development environments, "IDEs" in short. As depicted in figure 2.10, examples include Eclipse, NetBeans, Visual Studio, and IntelliJ IDEA. Such environments contain not only graphical tools but also compilers, parsers, and other means to create and run programs written in one or multiple languages.

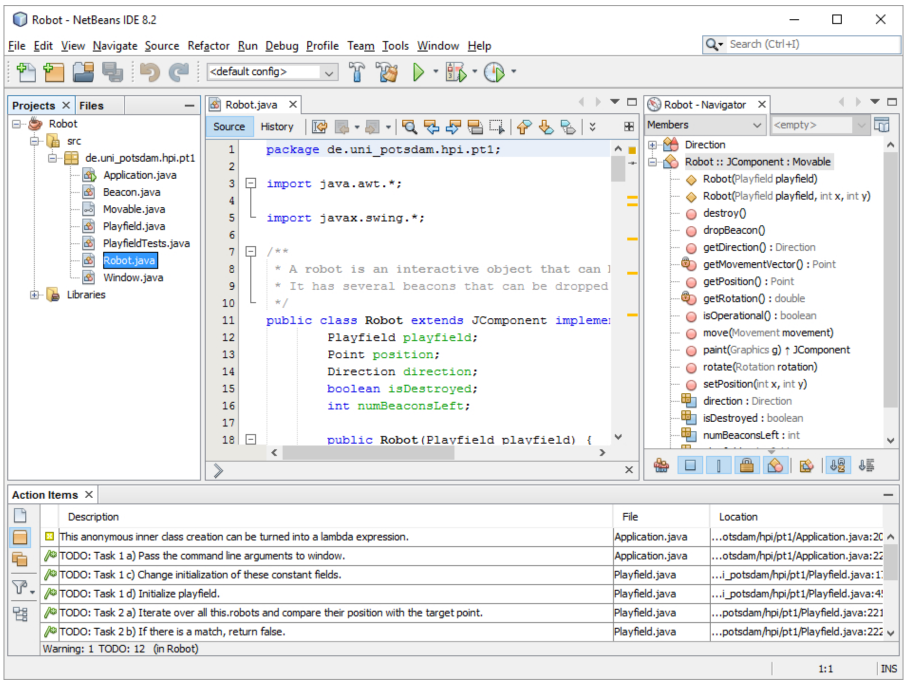
The integration of programming tools has a long history [210, 200]. Even today, we assess environments according to their presentation, data, control, and process


Robot - [C:\Users\Marcel\workspace\Robot] - [Robot] - ...\src\de\uni_potsdam\hpi\pt1\Robot.java - IntelliJ IDEA 2017.1
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
Robot src de uni_potsdam hpi pt1 Robot.java
Project
Robot C:\Users\Marcel\workspa
.idea
.settings
bin
src
de.uni_potsdam.hpi.pt1
Application
Beacon
Movable.java
Playfield
PlayfieldTests
Robot.java
Window
.classpath
.project
Robot.iml
External Libraries
Robot.java
Robot
Robot ()
Structure
Robot
Robot(Playfield)
Robot(Playfield, int, int)
getPosition(): Point
setPosition(int, int): void
isOperational(): boolean
destroy(): void
move(Movement): void ↑Movable
rotate(Rotation): void ↑Movable
dropBeacon(): void
getDirection(): Direction
getRotation(): double
getMovementVector(): Point
paint(Graphics): void ↑JCompone
playfield: Playfield
position: Point
direction: Direction
isDestroyed: boolean
TODO:
Project
Current File
Scope Based
18:12 CRLF UTF-8

```
package de.uni_potsdam.hpi.pt1;

import ...

/**
 * A robot is an interactive object that can be controlled
 * It has several beacons that can be dropped.
 */
public class Robot extends JComponent implements Movable {
    Playfield playfield;
    Point position;
    Direction direction;
    boolean isDestroyed;
    int numBeaconsLeft;

    public Robot(Playfield playfield) {
        super();

        this.position = new Point( x: 1,  y: 1);
        this.direction = Direction.South;
        this.playfield = playfield;
```

```
Found 12 TODO items in 4 files
  de.uni_potsdam.hpi.pt1 (12 items in 4 files)
    Application.java
      (22, 8) // TODO: Task 1 a) Pass the command line arguments to window.
    Playfield.java
      (17, 5) // TODO: Task 1 c) Change initialization of these constant fields.
      (45, 6) // TODO: Task 1 d) Initialize playfield.
      (221, 6) // TODO: Task 2 a) Iterate over all this.robots and compare their position with the target point.
      (222, 6) // TODO: Task 2 b) If there is a match, return false.
      (252, 6) // TODO: Task 4 a) Check whether there is a beacon at the given position.
```

(a) IntelliJ IDEA 2017.1

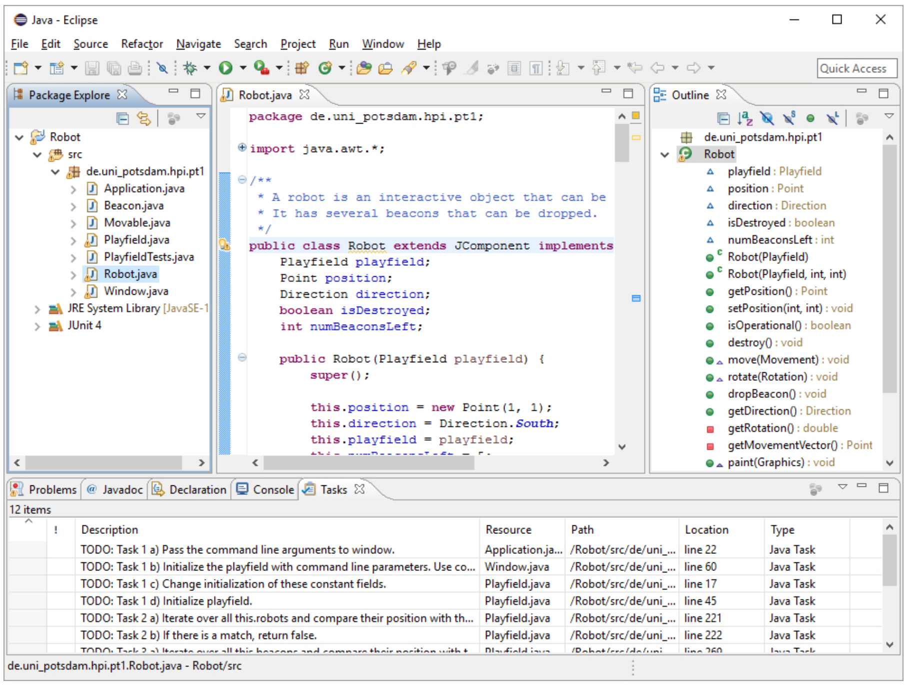


```
package de.uni_potsdam.hpi.pt1;

import java.awt.*;

import javax.swing.*;

/**
 * A robot is an interactive object that can l
 * It has several beacons that can be dropped
 */
public class Robot extends JComponent impleme
        Playfield playfield;
        Point position;
        Direction direction;
        boolean isDestroyed;
        int numBeaconsLeft;

        public Robot(Playfield playfield) {
```

	Description	File	Location
⚠	This anonymous inner class creation can be turned into a lambda expression.	Application.java	...otsdam/hpi/pt1/Application.java:20
✓	TODO: Task 1 a) Pass the command line arguments to window.	Application.java	...otsdam/hpi/pt1/Application.java:22
✓	TODO: Task 1 c) Change initialization of these constant fields.	Playfield.java	...i_potsdam/hpi/pt1/Playfield.java:1
✓	TODO: Task 1 d) Initialize playfield.	Playfield.java	...i_potsdam/hpi/pt1/Playfield.java:4
✓	TODO: Task 2 a) Iterate over all this.robots and compare their position with the target point.	Playfield.java	...potsdam/hpi/pt1/Playfield.java:221
✓	TODO: Task 2 b) If there is a match, return false.	Playfield.java	...potsdam/hpi/pt1/Playfield.java:222

Warning: 1 TODO: 12 (in Robot)

(b) NetBeans IDE 8.2



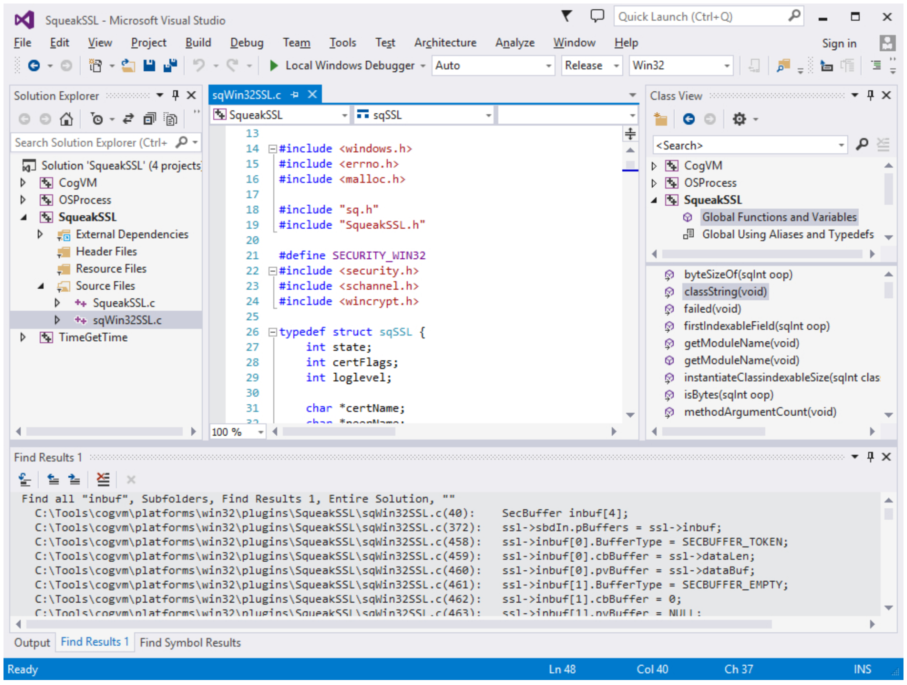
```
package de.uni_potsdam.hpi.pt1;

import java.awt.*;

/**
 * A robot is an interactive object that can be
 * It has several beacons that can be dropped.
 */
public class Robot extends JComponent implements
    Playfield playfield;
    Point position;
    Direction direction;
    boolean isDestroyed;
    int numBeaconsLeft;

    public Robot(Playfield playfield) {
        super();

        this.position = new Point(1, 1);
        this.direction = Direction.South;
        this.playfield = playfield;
```

12 items

!	Description	Resource	Path	Location	Type
	TODO: Task 1 a) Pass the command line arguments to window.	Application.ja...	/Robot/src/de/uni_...	line 22	Java Task
	TODO: Task 1 b) Initialize the playfield with command line parameters. Use co...	Window.java	/Robot/src/de/uni_...	line 60	Java Task
	TODO: Task 1 c) Change initialization of these constant fields.	Playfield.java	/Robot/src/de/uni_...	line 17	Java Task
	TODO: Task 1 d) Initialize playfield.	Playfield.java	/Robot/src/de/uni_...	line 45	Java Task
	TODO: Task 2 a) Iterate over all this.robots and compare their position with th...	Playfield.java	/Robot/src/de/uni_...	line 221	Java Task
	TODO: Task 2 b) If there is a match, return false.	Playfield.java	/Robot/src/de/uni_...	line 222	Java Task

de.uni_potsdam.hpi.pt1.Robot.java - Robot/src

(c) Eclipse Mars.1



```

#include <windows.h>
#include <errno.h>
#include <malloc.h>

#include "sq.h"
#include "SqueakSSL.h"

#define SECURITY_WIN32
#include <security.h>
#include <schannel.h>
#include <wincrypt.h>

typedef struct sqSSL {
    int state;
    int certFlags;
    int loglevel;

    char *certName;
```

```
Find all "inbuf", Subfolders, Find Results 1, Entire Solution, ""
  C:\Tools\cogvm\platforms\win32\plugins\SqueakSSL\sqWin32SSL.c(40):    SecBuffer inbuf[4];
  C:\Tools\cogvm\platforms\win32\plugins\SqueakSSL\sqWin32SSL.c(372):   ssl->sbdIn.pBuffers = ssl->inbuf;
  C:\Tools\cogvm\platforms\win32\plugins\SqueakSSL\sqWin32SSL.c(458):   ssl->inbuf[0].BufferType = SECBUFFER_TOKEN;
  C:\Tools\cogvm\platforms\win32\plugins\SqueakSSL\sqWin32SSL.c(459):   ssl->inbuf[0].cbBuffer = ssl->dataLen;
  C:\Tools\cogvm\platforms\win32\plugins\SqueakSSL\sqWin32SSL.c(460):   ssl->inbuf[0].pvBuffer = ssl->dataBuf;
  C:\Tools\cogvm\platforms\win32\plugins\SqueakSSL\sqWin32SSL.c(461):   ssl->inbuf[1].BufferType = SECBUFFER_EMPTY;
  C:\Tools\cogvm\platforms\win32\plugins\SqueakSSL\sqWin32SSL.c(462):   ssl->inbuf[1].cbBuffer = 0;
```

(d) Microsoft Visual Studio 2013

Figure 2.10: Traditional programming environments are tailored to the structure of the programming language constructs and the underlying transport medium such as files. There are means to configure layout, colors, keyboard shortcuts, or other properties to accommodate specific domains, tasks, and personal preferences.

integration. As explained above, a uniform look-and-feel in graphical tools supports usability and hence artifacts can become tangible. In addition to presentation integration, a whole environment can naturally control (and dictate) a single data format, that is, the transport medium for artifacts. If that format is open and documented like XML, external tools can easily be integrated as needed. Control integration addresses provision and use between tools or toward users. Process integration deals with details of the software development process such as bug tracking, deployment, and version control.

When discovering *deficiencies* in such convenient tool boxes, programmers can choose to add more tools to the environment. There are typically *plugins* and other means of extension that allow for trying out domain-specific visualizations and other, maybe unconventional, tools to accomplish tasks. For example, Code Bubbles [21] provides a different experience for organizing code or Software Terrain Maps [40] can reveal modularity issues in the system design. The creators of these plugins take care of plugging-in the required artifacts provided by other tools in the environment. Note that the term "programming environment" refers to any set of tools with some sort of integration. This includes Unix-based environments, whose text-based tools communicate via files and byte streams [85, pp. 163–217]. In this work, however, we focus on challenges that arise when building tools for *graphical* environments.

Synopsis Programmers can benefit from interactive, graphical *tools for programming*. While plain text does play a vital role in graphical interfaces, elaborate visualization can make software artifacts become more tangible. If programmers can quickly recognize artifacts and their relationships on screen, they are more likely to reason at the conceptual level of artifacts. Such reasoning is important to find the cause of bugs or the best way to implement new features.

In general, tangibility is important for programmers to find answers to program comprehension questions. When fetching, examining, and retaining artifacts, programmers can use many *tools in combination* to accommodate domain, task, or personal preferences.

Integrated programming environments can further improve the programming experience. Such environments provide a basic set of *coherent tools* by integrating presentation, data, process, and control. There are typically extension points to plug in additional tools that benefit from reusing the data format, presentation language, or other integrated means to accommodate specific programming challenges.

2.4 Configure Tools: Accommodate Domain, Task, Gusto

Many programming tools, especially whole programming environments, are tailored to *generic* scenarios to provide a high level of tool re-use and hence make the implementation worth the efforts. As depicted in figure 2.10, there is usually a larger area reserved for code editing and several lists for class browsing or file management. In general, it means that the graphical interface is aligned with the primary artifact structure. For code editors, this includes programming language constructs of the supported languages such as classes, methods, and instance variables. For e-mail clients, as non-programming example, this includes accounts, inboxes, and mails.

In this section, we explore *means of configuration*, which can make generic-looking tools exhibit domain-specific characteristics. We begin with describing the *virtues* of a common (or generic) look-and-feel for standard applications. We then explain existing ways to customize tool interfaces to accommodate project domain, current task, or personal gusto.

From Generic to Specific

Generic scenarios and best practices can help both tool *users* and *builders*. On the one side, users can learn new tools quickly drawing from shared (or common) experiences. On the other side, builders can clarify requirements early in the process and reduce the need for user testing. A common practice is *unification* [152, pp. 99–148]. For example, all list widgets should have scrollbars, copy operations should always be triggered via `[CTRL]+[C]`, and all windows should be resizable. This can be transferred easily to the programming domain. Symbols in a programming language, for example, need a browser tool to explore symbol references in the code.

However, any *dominant* representation of *generic* structure impedes working with domain-*specific* artifacts. Programmers might begin with generic terms in new software projects. Gradually, they introduce and use domain-specific terms and form custom programming interfaces. Such interfaces can evolve into a new domain-specific language [56]. As explained above, programmers have to understand a system's theory and concepts, which are encoded in terms of programming language constructs. That is, the conceptual level of a software artifact is important and should be exposed in the user interface to improve tangibility. Take an address book application, for example. If programmers would always be confronted with generic *classes*, instead of specific *persons* or *addresses*, their cognitive load would increase because of the continuous mapping activity. That mapping from the programmer's intents to tool actions and from the tools responses to programmer's expectations is called "gulf of execution" and "gulf of evaluation" [75]. Tools can improve the tangibility of information, and thus narrow down the gulfs.

There are many different *means of configuration* to accommodate domains, tasks, and personal preferences. Adapting tools can help raise the programmer's motivation and improve the programmer's efficiency. Consequently, not only *choosing* graphical tools (section 2.3) is important for efficient program comprehension but also the *configuration* of tool interfaces is.

Effective Scripting for Skillful Users

The *act of configuration* in a tool is usually simple and takes not much time. We hence assume that users should be able to reconfigure tools during regular work without noticeable interruption. There are typically graphical dialogs with checkboxes and buttons to change interface characteristics such as colors, fonts, and keyboard shortcuts.

Remark We consider programmers as being especially *skilled* users, which are capable of algorithmic thinking and advanced reasoning. They can employ, for example, mathematical skills and solve many "riddles" in their day job already. That is, the line between simple tool configuration and advanced tool building depends on experience and skills.

There are mechanisms in some tools' interfaces that *more difficult* but allow for expressing *more elaborate* modifications. We call these means *scripting languages*, which programmers can employ to customize tools for fetching and examining artifacts. Again, the vocabulary can vary: query languages, search expressions, or filter patterns. Such languages are usually declarative, high-level, and tailored to specific domains. For example, regular expressions [189] support finer grained search requests compared to wildcard or whole-word matches. While the mastering of regular expressions is actually quite challenging, many programmers do appreciate its ease-of-use for many common lookup tasks. Scripting languages provide many possibilities for tool builders that aim for *adaptable interface designs*. Experienced programmers might write smaller configuration scripts without even noticing an interruption from the current task. We do not primarily address tool adaptation toward personal preferences, which any regular user should be able to do. Programmers, which have programming skills, might also have ideas on how to accommodate the project's domain or work around some hitch in the current task.

In this work, we will separate *configuration* activities from *building* activities by the respective level of *difficulty*. Some means of configuration have become an integral part of programming tool usage and are not being noticed as something extra. For example, programmers are constantly moving, resizing, or closing windows to organize, examine, and retain information. *This is configuration of information layout on screen.* That is, to some extent, configuration can become a *subconscious activity* [122] embedded in any task. As mentioned above, there is typically a set of *default choices* such as for the default window layout. In figure 2.10, we can see a clear overlapping of defaults across several programming environments. Such defaults usually align with the primary programming language and best practices known from software engineering. Considering the vast landscape of available tools, again, meaningful defaults are *vital* to help programmers get started quickly.

There are different approaches to scripting languages, which can all be embedded in graphical interfaces:

- Simple keyword search, sometimes with wildcards
- Regular expressions often based on Perl [189]
- Logical queries based on the Prolog programming language [2]

So, we address any elaborate *search request*, which programmers can use to *find artifacts* that match textual properties in a certain *search space*. A typical space can be "all files that belong to the current project". For example, when looking for painter classes in a Java program, the regular expression "`class\s+Painter`" can describe some. Now, software artifacts can provide *rich structure*, which can be hard to describe this way. The search space of all accessible artifacts forms a graph with many nodes and edges, which represent the relationships in between. Conceptually, it can be easier to think of, for example, classes having methods instead of focusing of keywords and other constructs from the language or underlying transport medium. For exposing such structure, there are approaches that employ the *logical paradigm*, which refers to programming languages such as Prolog and Datalog. A prominent example for with paradigm with an implementation in Smalltalk is called SOUL [118]. The SOUL language can be used to describe *intentional views* [116] on code artifacts. These views support programmers to reveal broken invariants or other inconsistencies. Tools that use SOUL provide logical facts and rules to describe the corpus of all code-related artifacts. As a scripting language, programmers express logical queries like this one taken from [118]:

```
constructorMethod(?class, ?method) if
        metaClass(?metaClass,?class),
        methodInProtocol(?metaClass, [#'instance creation'], ?method),
        returnType(?method, ?class).
```

This query matches all methods that construct instances of classes, which are by convention in the protocol "instance creation". Considering *invariants*, the amount of matching tuples can encode validity. In general, any kind of *query-result view* can support code navigation. Double-click on a result with the mouse and the tool jumps to the right place in the source code. Further approaches that employ logic code queries are Jquery [39] or Codequest [71].

How Artifacts Present Themselves

Software artifacts specialize the appearance of graphical tools *inherently* by exposing structural information. Text plays an important role in graphical interfaces and hence an artifact's distinct name can support programmers to see a list of classes as *a list of domain concepts*. Well-chosen class names can help reduce the awareness of the underlying, generic language concepts. While iconic representation can, theoretically, also be used for this purpose, we think that many programmers would rather not create a pictogram for a concept since finding good names is hard enough. For complex structures, there are strategies to summarize artifacts automatically in text form [170].

Many tools provide programmatic hooks for domain models to improve examination of domain-specific artifacts. In many object-oriented systems, tools can ask all objects for a textual representation: the print string. At least the class name can be revealed and maybe an identity hash, too, like "`aPerson(1234)`". Programmers can customize such strings for any domain object with a custom implementation of the `toString()` method. This hook can take many forms of standardized, structural representation such as `toHtmlString()` or `toJsonString()`. However, programmers have to extract all the relevant information upfront.

In addition to such explicit serialization, tools are usually able to navigate and present an artifact's inherent structure. That is, programmers can navigate the instance of a class along with all referenced instances of other classes. Some environments even support the modification this visible structure. In Squeak/Smalltalk, there is a tool called "Inspector", which displays an object's low-level state in a generic fashion. To change this artifact (or object) presentation, programmers add certain methods to domain model classes, which compares to a custom print string. At this level, programmers can access any available means of aggregation or filtering. They can hide one aspect; they can emphasize on another one.

However, this approach is not very declarative and can thus be demanding on maintenance. We consider the *explicit* modification of the software's domain model toward tools the most *challenging and invasive* way of configuration. While print strings are well-known, structure modification on the code level might be feasible only in Smalltalk-like environments. In such environments, applications and tools reside in the very same information space. Additionally, the Smalltalk programming language supports concise yet readable expressions; it can be directly used as a scripting language for tools.

Window Management and View Layout

The richness of views in tools and tools in environments leads to the challenge of layout configuration. Programmers invest much time in moving, resizing, and closing windows or other content containers [163, 121]. Screen space is limited and hence programmers should efficiently layout tool views to support examining and recalling information [112, pp. 443–447]. The effective placement of views on artifacts can improve tool integration through the programmer's eyes and mind. At best, all artifacts relevant for a task can be placed side-by-side. There have been various

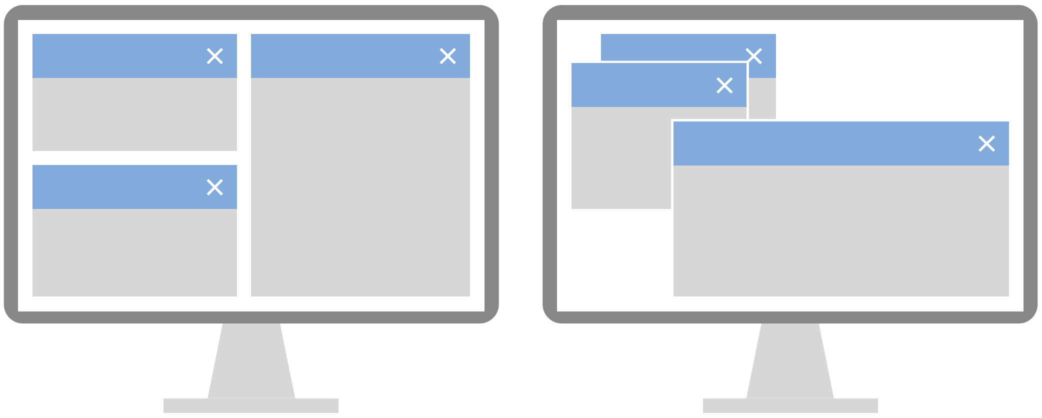


Figure 2.11: Tool views can be organized differently on screen. Sometimes the programmer has a choice. Sometimes it is dictated by the environment. Here, the left side shows a tiled window management and the right side shows an overlapping approach.

window management systems [127, 74] that form trade-offs between flexibility and convenience as depicted in figure 2.11. Nowadays, the two major approaches are *overlapping* and *tiled* windows, which go back to the Xerox Star user interface [179, 80] in 1981.[5] Overlapping windows provide a large degree of freedom and a fair layout stability while occasionally sacrificing visibility when new windows pop up. Tiled windows provide relative spacing and ensure all-or-nothing visibility for views, often implemented as *stacked or tabbed containers*. As programmers can notice in the Eclipse debugger perspective, tiled windows have a poor layout stability when switching tasks.

In programming environments, there is a primary layout strategy while integrated tools are free to employ their own. In Eclipse, we can see tiled windows. In Squeak, we can see overlapping windows but tiled views in each window. There have been many attempts to make these two strategies scalable, considering large amounts of information versus limited screen space. Code Bubbles, for example, expands the tiled strategy to an unlimited, two-dimensional canvas [21]. The Patchworks code editor [73] and the Moldable Inspector [30] employ and unlimited, one-dimensional tape, which is also tiled. The Gaucho environment [139] focuses on content composition via nested windows, which can be collapsed to save space.

We think that it is highly task-dependent whether one layout strategy is better than another. For new windows, a clever start position and size can save much efforts. The window manager in Smalltalk-80 systems gives the user control over the initial dimensions [64, p. 16], but this means always more interaction effort for the user. In the Moldable Inspector [30], in-depth examination of artifacts produces new views side-by-side on and endless tape. In the Debugger Canvas [41], which is the Code Bubbles concept implemented in Visual Studio, in-depth examination of

[5] Actually, Smalltalk-76 did introduce an overlapping window manager [43]. Yet, there were many trade-offs made such as inactive windows being lifeless hulls without updating contents.

method calls in a debugging session produces new bubbles without overlapping on the two-dimensional canvas. In addition to position and size, there have also been strategies developed to automatically "age" windows based on use [163].

Configuration Scope

Programming tools provide means to *scope* configurations. We distinguish global, project, and task scopes to encompass configurations that accommodate personal (global) preferences, project domains, and current tasks respectively. Programmers' favorite colors or fonts, for example, are usually set for the whole integrated environment including all tools and all tools' views. Many environments such as Visual Studio and Eclipse have also project management support. There, custom settings for compilers, runtimes, bug tracking, or version control are stored, which includes file paths, optimization switches, and login information.

Task-level scope, however, is rarely supported. Especially the explicit switching between several concurrent tasks is often left to programmers' creativity. For generic work modes such as debugging, there might be specialized configurations. Eclipse, for example, has the concept of *perspectives* to manage sets of views to emphasize on code writing, debugging, or versioning. Squeak uses a different meaning of *projects* to keep track of open windows, which can be used to organize tasks and switch in between [192]. Fortunately, many environments support at least one *set of open tools/views* to be restored on the next workday. Explicit task management can be beneficial as demonstrated via Mylyn [126]. There, a degree-of-interest model is used to trace artifacts that belong to the same task based on the user's exploration activities. Such traces can also be used to improve code navigation later when programmers have to work on similar features or bugs.

There is not only *external* configuration but also *invasive* modification of artifacts to accommodate tool characteristics. As mentioned above, this includes explicit interface modification of domain classes for textual representations such as `toString()` in Java and `#printOn:` in Squeak. In general, programmers have to put more effort into configuration if they miss or cannot even target the appropriate scope in tool environments.

Synopsis Programming tools are usually tailored to the underlying code artifacts to support best practices and standard workflows. There are means of configuration to accommodate domains, tasks, and personal preferences with low effort. While the plenitude of options can be overwhelming, there is typically a good set of defaults to help programmers get started quickly. Since programmers are rather skilled computer users, configuration does not only mean pushing buttons or dragging sliders. It does also employ elaborate scripting languages including regular expressions and logic queries. Additionally, efficient window management is paramount but also dependent on the particular program comprehension task. If programmers can manage to scope tool adaptation as intended—meaning global, project, or task level—they are more likely to do so and improve program comprehension activities, that is, fetching, examining, and retaining artifacts.

If programmers have ideas on how to improve a tool's view, they have to map their intent to a kind of configuration, come up with a value for it, and then observe the effects for assessment. They can iterate and try several values. If, however, a tool's implementation has to be changed to get the desired effect, *tool configuration becomes tool building*. The process of building a tool represents a task on its own, which often needs dedicated resources in the project schedule.

2.5 Build Tools: Beyond Configuration

When facing unforeseen programming challenges, tool configuration can work only so well. Any tool's high-level, user-friendly configuration interface does arguably broaden its field of applicability. Still, all dimensions of configuration have to be anticipated by the tool's designer, who makes them explicit. While this is, nowadays, straightforward when accommodating for individual gusto such as fonts and colors, customizing for domain-specific means of fetching or examining artifacts can be challenging this way, if even possible. That is, buttons, sliders, or scripting languages can hit a boundary that forces programmers look into the tool's source code. And programmers will find this more challenging even if the tool's internals employ a well-documented architecture, known idioms or patterns, and flexible extension points.

We emphasize the separation between tool building and tool configuration because there are environments such as Squeak where tools are written in a high-level language, which might resemble the simplicity of another environment's configuration language. *Tool configuration becomes tool building* whenever programmers have to leave the tool's user interface and are confronted with the tool's internals. The perceived difference in effort when switching perspectives is relative. For example, one can state that Eclipse plug-in development [61, pp. 33–90] is very difficult compared to writing new tools for Smalltalk environments,[6] so that any Smalltalker should favor writing domain-specific tools all the time. Even in Smalltalk environments, however, programmers can perceive a large difference in effort between tool configuration and tool building. That is, on the one side, tool configuration is safe and simple. You cannot break the tools in use, which you might do instead as an incautious, rushing tool builder—even if programmers can scope changes in such self-supporting

[6] Programming tools in Smalltalk-80 were written for the Model-View-Controller (MVC) framework [98, pp. 7–12]. Later for the Squeak/Smalltalk system, there was Morphic [108] introduced, which made debugging more interactive. Recent versions of Squeak also employ the Tool Builder framework [192] as a declarative abstraction for tool code.

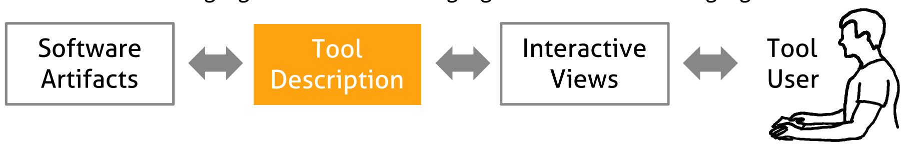


Figure 2.12: Programming tools are basically adapters between software artifacts and interactive views. Tools query artifacts, map artifact structure to visual properties, and present many views in composition. Tools mediate read *and* write operations initiated by user interactions.

environments [104]. If, instead, tool building is a *dedicated* activity, programmers can carefully assess the trade-offs and create safe configuration interfaces for tool users to tweak. Dedication means that tool building usually takes time and cannot be embedded in another programming task. For larger software projects, there are resources to invest in such tools [180].

So, we highlight certain aspects that become relevant when building and adapting graphical tools for programming to accommodate specific domains and recurrent tasks. Considering our focus on fetching, examining, and retaining software artifacts, this addresses new tools that integrate more information, provide better graphical views, or manage content more appropriately on screen.

Query, Map, and Present: Tools as Adapters

Graphical tools are basically *adapters* that let programmers work with software artifacts through interactive widgets as depicted in figure 2.12. The adapter pattern is a well-known, object-oriented design pattern that "lets classes work together that couldn't otherwise because of incompatible interfaces" [62, pp. 139–150]. Here, the problem of "incompatible interfaces" addresses the rich structure of artifacts, the physical limitations of digital input and output devices, as well as mainly *visual* human perception to improve learning and recalling, that is, program comprehension. For programmers, an appropriate connection between artifacts and widgets can yield a tangible and vivid representation of relevant concepts. In this work, we focus on this middle part and we take the set of valuable artifacts and elaborate widgets for granted. For example, if a tool should integrate traces of test runs [146] with code, we assume that call trees and code artifacts are basically accessible. That is, the tool builder does not have to design an efficient mechanism for tracing code execution. Like in figure 2.7, if a tool would benefit from a combination of buttons, lists, and bar charts, we assume that these widgets are on hand, too. Still, tool builders are able to express code that aggregates and filters artifacts and their structure to reveal new insights. Such new insights can be manifested as new kinds of *derived* artifacts. With reasonable effort, it is also possible to create and embed new graphical structures in existing ones. The result may just look like a new widget even if composed from primitive commons.

We follow common reference models for visualization construction [68] and apply them to tool building. In object-oriented environments, building tools means writing code that employs (1) query languages, (2) mapping languages, and (3) presentation languages as outlined in figure 2.12. Both software artifacts and interactive views are accessible and combinable.

Query language Tools fetch artifacts, apply filters, and aggregate missing information. Tools can integrate many information sources. Tool builders have to consider the tool's purpose to query and combine the relevant sources. As explained before, graphical interfaces can expose such languages as means of configuration. For example, relational databases can be queried via SQL, whose query expressions can be requested from the user. Think of a text widget where the user types expressions such as:

```
SELECT name FROM customers
```

This also applies to high-level programming languages such as Smalltalk, Python, and Visual Basic. The scripting language can be the same language the tool is written in. For example in Squeak, programmers can directly write Smalltalk to fetch code artifacts:

```
Smalltalk allClasses select: [:each | each name beginsWith: 'Abstract']
```

Here, the global variable `Smalltalk` is the source to fetch abstract classes that are denoted with the prefix "Abstract".

Mapping language While many software artifacts have a distinct textual property, only some do also provide an *inherent graphical representation*. If the artifact is a photograph, for example, tools can directly display that image on screen. If the artifact is an abstract syntax tree, tool views are likely to be text heavy. Even if artifact structure is primarily visual, the richness of structure may still require modification of the representation to get something the user can understand in the context of the current domain and task. So, tool builders have to express *rules* for mapping artifact structure to visual widget properties. This includes text, color, font, line thickness, and so on. Such a rule, for example, could be:

Map the *employee*'s salary to a *color* value that ranges from red to green.

Assuming that *employee* and *colors* are accessible through object-oriented code, the mapping can be written in Smalltalk like this:

```
employees collect: [:each |
  Color h: (each salary / 1000 min: 90) s: 1.0 v: 1.0]
```

This rule creates a list of colors where each color corresponds to each employee's salary where green stands for high values and red for low ones. Like query languages, mapping languages can be embedded in a tool's configuration interface if users are capable of "speaking" that language. For example, tools useful for statistical analysis provide many different means for mapping data to visual properties such as IBM SPSS and Microsoft Excel do. At the code level, tool builders can accommodate domain or task with appropriate mappings.

Presentation language For tool users, the presentation of artifacts in the graphical interface is paramount. They have to recognize concepts from visuals such as labels and pictures. They have to invoke operations on artifacts in an interactive fashion. A good presentation language emphasizes places of *possible input* while managing smooth updates of *explorable output*. As we focus on program comprehension tasks, the interactive fetching, examining, and retaining of relevant artifacts is of importance. Tools usually employ a combination of common widgets such as text fields, structured views, and buttons. The *means of composition* include windows and dialogs filled with rows and columns. A widget composition does not have to be recognized as such but maybe as a *coherent whole*. For example, there might be widgets representing whole music sheets [162] in music composition tools.

Remark So far, we have introduced many different kinds of languages: scripting, query, mapping, presentation, and programming languages.

We call elaborate means of configuration *scripting languages*. For the design and construction of graphical interfaces, we call the graphical, interactive means of tools *presentation language.* It is the language a tool "speaks" with its user. There is always a *query language* involved to fetch artifacts. When examining artifacts, tool users or tool builders can employ the same (or similar) query languages to filter artifacts or derive new artifacts. Since many artifacts do not have an inherent graphical representation, there is a *mapping language* for connecting artifact structure to widget visuals. Scripting languages can be employed to lift mapping and querying from the build level to the configuration level.

Finally, there is another kind of language involved to actually create tools: the *programming language.* In the remainder of this work, we employ Smalltalk as the object-oriented programming language. Except for the presentation language, Smalltalk can be used to *represent* the syntax and semantics for *all* of the languages above. Note that there are many other higher-level languages, such as Visual Basic and Python, that are referred to as scripting languages to distinguish from system programming languages [140], such as C and Java.

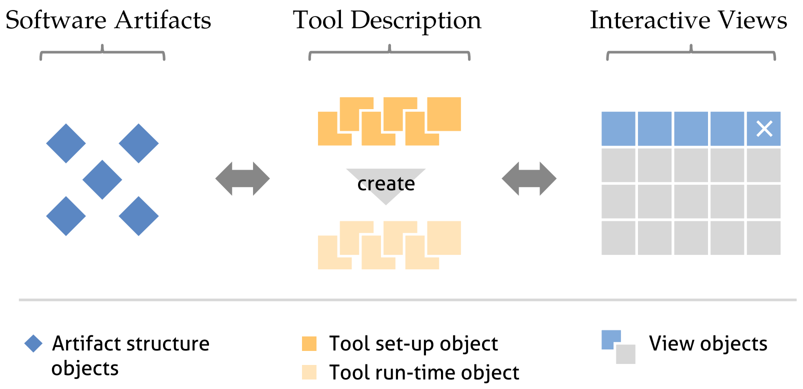


Figure 2.13: Graphical tools (here: middle) have to set up artifact access (here: left) and visual mapping (here: right. Tool objects mediate between artifact and view interfaces. Tool-building frameworks provide efficient means to design set-up and run-time objects.

Everything is an Object

In *pure* object-oriented applications, programmers design *objects for all concerns* in the system: artifacts, widgets, mapping rules, and so on.[7] We explained this issue before in section 2.1 with the three levels of software artifacts: (1) the conceptual level, (2) the structural level, and (3) the transport level. We are now approaching tool implementation details, namely the *transport level*, where "every thing" can be an object as depicted in figure 2.13.

Any object-oriented environment can be "situated" between other paradigms. While the object-oriented paradigm is rather *imperative*, other programming models can be created with the help of objects and, for example, provide *declarative* means to fetch artifacts or assemble widgets. The execution environment drives the object-orientation in programs. It can employ *different* programming paradigms like Squeak, for which there are virtual machines written in C or JavaScript.

Regarding the availability of artifacts, there are means that support converting from foreign transport media to objects such as object-relational mappers [12], which keep track of object representation in database tables. Regarding the availability of widgets, there are many libraries that provide the ones typical for graphical user interfaces: windows, buttons, text fields, or item views.

A tool's architecture makes trade-offs between programming *effort* and anticipated *added value* for future programming tasks. There are many frameworks and libraries that guide programmers in the tool design process. Still, programmers do not necessarily have to comply with all patterns or idioms that are provided. Like generic programming tools, the underlying tool frameworks do also support a wide range of applications. Depending on the particular cost-value ratio, a "quick-and-

[7] Even though widgets or rules could be treated as artifacts on their own, we distinguish them in the context of tool building. In object-oriented *systems* such as Squeak/Smalltalk, a similar distinction is necessary because everything can be treated as an object.

dirty" implementation can be good enough, which can intentionally ignore several conveniences of the framework. If, on the other hand, forthcoming tool maintenance is important, programmers are more likely to employ all hooks in the framework as intended to maximize the time saving. Sometimes, however, the framework's anticipated scenarios fall short and programmers have to *bypass abstractions* to design the desired tool behavior.

Glue for Set-up and Run-time Objects

As depicted in figure 2.13, we assess tool building frameworks by two kinds of objects that have to be designed for the tool (adapter) code: (1) set-up objects and (2) run-time objects. These objects mean packages, classes, methods, and other concepts provided by the used programming language. The source code that has to be written is often referred to as "glue code". *Set-up objects* govern widget composition, visual mapping, and usually also simple fetching rules. Programmers recognize such objects as initialization code or static user interface descriptions. *Run-time objects* govern all dynamic aspects of a tool that cannot be statically defined at set-up time. Programmers recognize such objects as callback methods to be implemented to adapt between widget interfaces and artifact interfaces. The general mismatch between domain-specific artifacts and general-purpose widgets renders it often mandatory for tools to mediate in between at run-time.

Tool frameworks can optimize the creation and maintenance of these two kinds of objects in different dimensions such as:

- **Granularity** of run-time objects, that is, artifact wrappers and tool-internal modularization
- **Declarative** or imperative means to construct set-up objects and run-time objects; support for breaking through framework abstractions robustly
- **Tool support** for defining set-up objects or run-time objects such as for widget composition and visual mapping, often with code generation
- **Integration** with and re-use of existing tools and other frameworks

In general, a tool framework benefits from employing many concepts that are provided by the underlying programming language and environment. For many class-based, object-oriented languages, this typically includes polymorphism and late binding. Consequently, there would be different abstract interfaces to implement in tool code so that the basic control flow works as intended by the framework design.

There are many different tool building frameworks and tool builders have to choose. Programming environments can support more than one framework while there is typically a primary one. We found the following examples supportive to give an overview of the aforementioned dimensions:

Granularity in Eclipse For programmers in Eclipse, tools should be implemented as *plugins* [61] to get access to other tools' artifacts and hence the internal transport medium, which is not files but objects. Besides data integration, plugins benefit from control and presentation integration [210, 200].

Granularity in Squeak Tools and any other application written in the Squeak environment do already benefit from the *shared object space*. Programmers only have to decide for the graphics framework, which dictates the control flow with respect to user interactions. Squeak's standard tools employ the *Tool Builder* framework [192], which entails compact and declarative set-up code. For historical reasons, code browser, object inspector, debugger, and other tools exhibit characteristics of a "fat model" design considering the run-time objects (i.e. the model classes) to manage widget state and callbacks. The main reason for the Squeak Tool Builder's design was the support of existing tools in two different frameworks: Model-View-Controller [98, pp. 7–12] and Morphic [108].

Declarative in Glamour Since the addition of new kinds of widgets and overall maintenance has been difficult in Squeak's Tool Builder, the *Glamour* framework [24] was created. It has also a compact and declarative way to set up tools. Originally, it did support two front-ends, namely the Seaside Web framework [147] and Morphic, but only Morphic remained. Glamour, as well, was meant to replace the standard programming tools first and foremost. This meant a rather static user experience with definitive places for lists and buttons and also a static description of expected artifacts. Nowadays, however, the "Glamorous Toolkit" supports dynamic and "moldable" tool behavior, especially when debugging [29] and examining [30] domain-specific software artifacts.

Integration in OmniBrowser Eclipse plugins, Tool Builder applications, and also the Glamour framework do not enforce or suggest a particular design for run-time objects. One approach to do so is called *OmniBrowser* [16] where each artifact is wrapped into its own run-time object to be shared between similar browsing or exploration tools. Task-specific tools can employ different kinds of wrappers. However, the maintenance of numerous wrappers for the same artifact can get challenging. We could only observe standard programming tools that wrapped artifacts from the Smalltalk language such as classes and methods. We are not aware of other tools written for the OmniBrowser framework.

Tool support in Qt Widget composition and visual mapping can benefit from tools that have direct manipulation interfaces. For example, the Qt framework for writing graphical, cross-platform applications [18] has an interactive tool, namely the Qt Designer, to compose the tool's interface and first interaction hooks. The produced artifact has a declarative format. For a long time, that artifact has been in XML, which has to be compiled to C++ code by another tool. With the rise of mobile applications, a more human-readable and human-writable format was created: QML, the "Qt Markup Language". QML is comparable with CSS, the "Cascading Style Sheets", or the Smalltalk code that programmers write when using Squeak's Tool Builder.

[|]

The integration of new tools with existing tools entails additional glue. Using Wasserman's model of tool integration [210], we *simplify* platform integration, data integration, and presentation integration. The *platform* will be specified by the programming environment you write the tools for. This is okay because many environments that augment operating systems do work in many operating systems. For example, Squeak and Eclipse run under Microsoft Windows, Apple macOS, and several Linux distributions. When working in an environment such as Squeak/Smalltalk, the *data* is already integrated when having an object representation for the artifacts of interest. Third, the interactive *presentation* of artifacts and operations is clear by employing common widgets such as lists and buttons.

What remains, is the challenge of *control integration*. For the scope of this work, we assume that tool provision and tool use [200] is triggered by user interactions. Since we focus on program comprehension tasks, programmers use tools to fetch, examine, and retain artifacts in an integrated fashion.

As explained before, supportive tools help programmers recognize relevant software artifacts on screen. That is, the *tangibility* of artifacts is paramount. When using one tool to examine some artifacts, there should be an appropriate connection to other tools that support continuing the task. Common interaction patterns include the use of pop-up context menus, the concept of selections, and also drag-and-drop operations. A tool can place a trigger into another tool's context menu. A tool can also react to drop operations. That is, in comparison, we do not primarily investigate the challenge of integrating auto-completion tools into a code editing tool. We focus on tools that are more high-level and that support fetching, examining, and retaining software artifacts by exhibiting structure and relationships in between.

Find, Change, and Verify: Iterative Building and Emergent Design

A programming tool is just another piece of software. So, the process of tool building is also iterative because programmers are constantly exploring problem and solution space. That is, creating a *new* tool quickly becomes modifying an *existing* tool, which is the same. We thus assume that programmers employ agile practices known from Scrum [169] or Extreme Programming [14] to be able to quickly respond to changed (or clarified) requirements. Especially if a certain tool is already in use, new requirements or deficiencies can arise quickly. So, there should be means to provide *short feedback loops* between changing the tool's source code and observing, if not understanding, the outcome. While agile processes do have a notion of well-defined tasks, we also encourage *exploratory programming* where programmers are free to follow hunches [171] to explore any upcoming idea. They may not even write tests in such a process [60] to not break the flow [125, 38]. Our focus on Smalltalk promotes such working habits because Smalltalk environments themselves support requirements exploration in many ways [167].

To better understand tool support for tool building, we model *a typical cycle* in the iterative building process as: (1) find observed behavior in the source code, (2) change that code as desired, and (3) verify changes via re-observation. Compared to the tasks we described in section 2.2, this process model for tool building looks similar. Yet, it is more detailed than just understanding the theory behind and the effects of the current changes. We will use this model to assess challenges in the contemporary tool building process that are not yet addressed in current tool building frameworks.

Find Triggered by a concrete visual impression in the tool's graphical interface, the tool builder has to find all places in the source code that are responsible for such observable behavior. This requires an understanding of the underlying architecture and theory as explained in section 2.2. Well-chosen identifiers for classes and methods can support this by revealing patterns behind it. For example in Smalltalk-80, which employs the model-view-controller pattern, programmers have to look for the right model and controller classes to spot relevant source code. It can take several find-change-verify cycles to actually find all relevant places in the code.

Change After finding relevant places in the code, the tool builder now has to make changes. For simple modifications, this may just mean a different use of the involved query, mapping, and presentation languages. For example, a set of artifacts can be filtered differently, the visual mapping be refined, or the widgets be recomposed. Larger modifications can require more time and thought such as integrating more data sources or refactoring glue code in the range of framework possibilities. It is in the interest of the programmer to express all intentions in a concise but readable fashion. We choose Smalltalk for this very reason because we think it has a lower baseline of verbosity compared with many other object-oriented programming languages. It can take several find-change-verify cycles to actually form such an agreeable kind of code quality.

Verify After making some changes to the tool's source code, a typical way to test for the intended effect is to *restart* the tool. If the changes affect set-up objects as described above, programmers have to ensure the re-execution of that changed initialization code. Especially if the role of some code remains unclear, *consistency* of tool state can only be assured if the programmers interacts with it after a fresh start. Smaller changes with a rather clear scope, however, might be verified while the tool is still running. For example, this applies to all resources that are dynamically loaded and refreshed such as configuration scripts that have a clear point of (re-)execution.

In Squeak, programmers can easily evaluate any piece of Smalltalk code and hence update the tools state—if they know what they are doing. An unsatisfying result after verifying the changes is typically the trigger to begin another cycle in this process.

Squeak's Shared Object Space

In Smalltalk environments, software artifacts being *accessible* means that there are *objects* representing them so that programming tools can use them. While our thoughts and explanations can easily be transferred to any object-oriented environment (or application), we choose Squeak/Smalltalk for its *image concept*. A Squeak image file captures a long-living, manageable object graph, which provides programmers with a clear and consistent realization of the idea that everything can be an object. Recapping our trichotomy from section 2.1, both artifact structure and concepts can be revealed using objects as the transport medium.

In the object-oriented programming paradigm, each object has *identity, state, and behavior* [119, pp. 218–228]. Since we distinguish between artifact *structure* and artifact *concepts*, we see value in separating objects that emphasize *managing state* from objects that emphasize *managing behavior*. Observable behavior can reveal concepts; state is typically of private matter and only supportive for expressing that behavior. Of course, when using programming tools that show artifact structure, object state becomes public and ready to be modified by tool users. There is a similar distinction in traditional catalogs for design patterns, which separate structural patterns from behavioral ones [62, pp. 9–11].

In figure 2.14, we show a simple model that connects our notion of software artifacts with concrete objects in Squeak's shared object space. Note that the figure is a more elaborate version of figure 2.13. If artifacts are managed externally, there have to be some objects that map from external transport media to objects. For example, e-mails or tickets can be created by tools that operate in other environments and hence use transport media different to objects—or at least different object spaces. Then, if objects should not be treated as mere *data holders*, there can be other objects (or messages) attached that enrich an object's interface to make it more tangible.

In Squeak, messages that directly read from or write to an object's state are called *accessors*. Methods that implement other messages known to that object typically re-use these accessors to define behavior and thus concepts. Such interfaces can be attached *from the outside* through tool code, too. However, if several tools share conceptual views on artifacts, it makes sense to move the respective code into the object representation of the artifact. For example, if several tools require the concatenation of a person's first name and last name, it makes sense to add `#fullName` to the object's interface that represents that person in the Squeak environment.

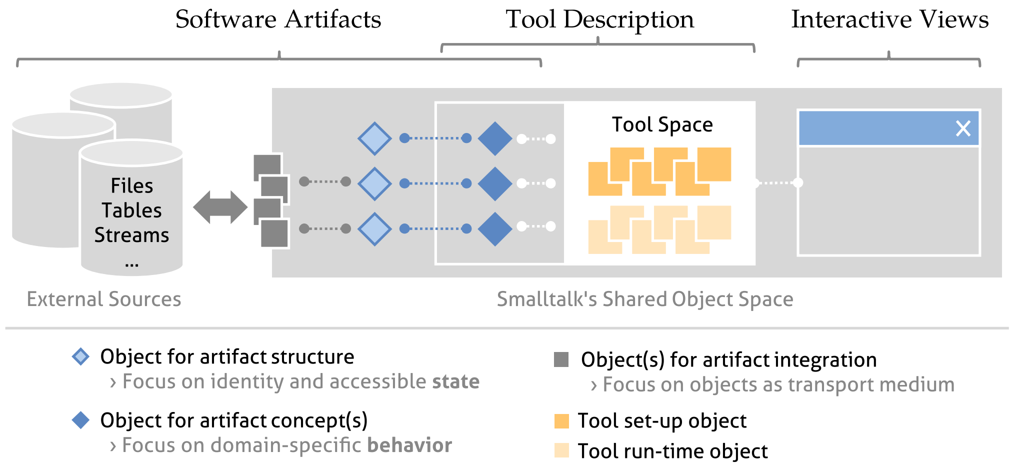

Software Artifacts
Tool Description
Interactive Views
Files
Tables
Streams
...
Tool Space
External Sources
Smalltalk's Shared Object Space
Object for artifact structure
› Focus on identity and accessible state
Object for artifact concept(s)
› Focus on domain-specific behavior
Object(s) for artifact integration
› Focus on objects as transport medium
Tool set-up object
Tool run-time object

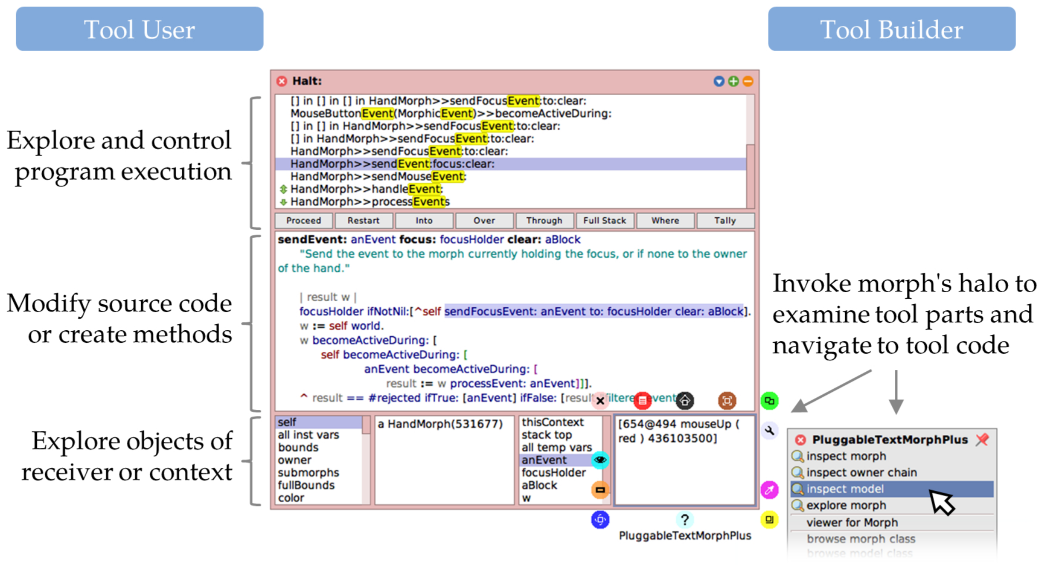
Figure 2.14: The concept of Smalltalk's shared object space requires external artifacts to be mapped onto objects as transport medium. In addition to object identity and state, object behavior becomes relevant if tools should reveal *artifact concepts*. Behavioral interfaces can be attached to objects directly or hidden in tool code.

Remark Squeak is a programming system that offers a class-based [211], object-oriented programming language: Smalltalk [65]. In Squeak, every object is the *instance* of a *class*. Classes are *blueprints* that describe instance variables and methods. Methods are organized in *protocols*, which capture different interfaces or perspectives. Typically, programmers model concepts by writing code for classes. Representatives of such concepts are then instances of these classes. Classes can extend a *super class*, which is called *inheritance*. Since Squeak/Smalltalk offers only single inheritance, there can be concerns/concepts that *crosscut* a given inheritance tree [198]. In that case, protocols can be used to attach additional concerns to objects. For example, the concept `Person` can be modeled as a class while the concept `Friendship` is just a protocol in the same class with messages such as `#befriendWith:` or `#unfriendFrom:`.

For the basic understanding of how to get an object representation of software artifacts into an object-oriented environment, it is not necessary to have a programming language that has a notion of classes. It is not even necessary to use a dynamically typed language such as Smalltalk. Our thoughts can easily be applied to environments such as the Eclipse [61], which is written in Java. Still, we see great value in the conciseness of programs written in the Smalltalk language.

Tool Building in Squeak

In Squeak, all graphical elements provide a feeling of directness and tangibility on their own. These elements are called *morphs* and the underlying framework is called *Morphic*. Tangibility becomes noticeable if programmers invoke a *meta* pop-up menu



```
[] in [] in [] in HandMorph>>sendFocusEvent:to:clear:
MouseButtonEvent(MorphicEvent)>>becomeActiveDuring:
[] in [] in HandMorph>>sendFocusEvent:to:clear:
[] in HandMorph>>sendFocusEvent:to:clear:
HandMorph>>sendFocusEvent:to:clear:
HandMorph>>sendEvent:focus:clear:
HandMorph>>sendMouseEvent:
HandMorph>>handleEvent:
HandMorph>>processEvents
```

```
sendEvent: anEvent focus: focusHolder clear: aBlock
	"Send the event to the morph currently holding the focus, or if none to the owner of the hand."

	| result w |
	focusHolder ifNotNil:[^self sendFocusEvent: anEvent to: focusHolder clear: aBlock].
	w := self world.
	w becomeActiveDuring: [
		self becomeActiveDuring: [
			anEvent becomeActiveDuring: [
				result := w processEvent: anEvent]]].
	^ result == #rejected ifTrue: [anEvent] ifFalse: [resul filtere vent
```

```
self            a HandMorph(531677)   thisContext      [654@494 mouseUp (
all inst vars                         stack top        red ) 436103500]
bounds                                all temp vars
owner                                 anEvent
submorphs                             focusHolder
fullBounds                            aBlock
color                                 w
```

Figure 2.15: The parts of Squeak's graphical tools are called *morphs* and can be selected or examined via each morph's *halo*. While tool users focus on representation of artifact structure and operations, tool builders can take a look *behind* that "facade" to find the objects responsible—even in for the debugger tool.

on a morph: the *halo*. A morph's halo looks like a collection of buttons ordered in a circle around the morph as depicted in figure 2.15. These buttons offer generic interaction such as selection, movement, and resizing. Programmers can access the entire graphical hierarchy this way.

The idea of having such a meta menu for graphical elements originates from Self [110] in 1995. Back then, this meta menu looked like a regular, "grayish" pop-up menu. In comparison, Squeak's Morphic [108] looks more colorful and playful because it was created in the context of a programming system suitable for children, namely Etoys [7]. Nowadays, it is easily possible to load other graphics frameworks into Squeak, but Morphic is still the most popular one [192].

Tool builders can easily examine the state of a running tool to find out about bugs and navigate to the underlying source code. See figure 2.15 for an example of examining a text widget in a debugger window. Consequently, if a programming tool is entirely composed of widgets and those widgets are composed of morphs, interactive examination becomes convenient. There is no need to mentally recall the class name of some visual object. Just click and explore. Still, programmers are not bound to employ morphs in graphical applications all the way down. In fact, the single letter in a text on screen is typically not a morph but just the result of some source code that frequently repaints an area on the screen. There is always some overhead involved when creating and managing morphs.

Smalltalk programmers apply tactics common to other communities for finding and changing code as well as verifying the changes. Yet, they do also benefit from specifics found only in such *self-supporting*, object-oriented programming systems. As described before, when programmers act as tool builders, they employ the *Morphic* framework for the graphics and usually the *Tool Builder* framework for the glue. There is also a *standard library*, which is an object-oriented implementation for elementary concepts such as numbers, strings, and collections. In the process, programmers constantly switch roles between tool user and tool builder.

Find in Squeak There are code browsers and object inspectors with symbolic search support for fetching and examining code artifacts. In Squeak's Morphic, programmers can click on each graphical element on screen to examine its state and navigate to its source code. Additionally, programmers can *always* use a special keyboard shortcut to pause the execution of an apparently unresponsive UI process, which then starts a symbolic debugger to investigate suspicious code.

Change in Squeak Programmers use code editors to modify classes or methods. The Smalltalk language supports very concise and readable expressions. Complexity is usually introduced through frameworks that add many concepts and glue. Besides code changes, programmers can *evaluate code* on resource objects to *modify state*, which can influence tool behavior. Such objects are in-image databases for pictures, sounds, or key bindings.

Verify in Squeak Programmers can restart tools, which means that certain references to tool objects have to be removed so that Squeak's garbage collector can clean up. Then, the tool's model class can be fed into the Tool Builder framework again to re-open the tool. In Squeak and other Smalltalk systems, the scope of a single change is very small. Typically, programmers change single methods, which are then re-compiled and integrated into Squeak's runtime in a matter of milliseconds [155]. As soon as tool objects send messages that trigger the changed methods, the tool can exhibit the changed behavior. Programmers either have to restore tool state manually after restart or they patch the tool's run-time objects directly. The latter requires proficient knowledge of tool internals.

Synopsis A tool's means of configuration are often limited and cannot accommodate *every* domain, task, or personal preference—which affects tangibility of visible information. When programmers create new tools or modify existing ones, they are free to integrate any kind of software artifact and choose any kind of interactive widget. From the perspective of tool users, integration means appropriate use of presentation, query, and mapping languages. Such a degree of freedom is typically non-existent in configuration interfaces because of the trade-offs made when designing such interfaces.

Programmers build tools in an iterative fashion. They undergo many cycles of (1) finding relevant code, (2) making changes, and (3) verifying the changes. Especially during verification, *tool builders become tool users* to some extent. They will interact with the tool and experience the new feature through the tool's graphical interface.

In Squeak/Smalltalk, programmers have objects for all kinds of information at hand. There are objects for software artifacts. There are objects for interactive views. There are objects that make up the tool code in between. The Morphic framework supports direct manipulation of all graphical elements via a morph's halo, which is an additional context menu to "look behind" the object's graphics. So, Squeak exhibits two important uniform principles:

1. Every "thing" is an object in a persistent, shared object space.
2. Every graphical object is tangible via a shared interaction paradigm.

In Squeak, programmers might not see a difference in effort between configuration and implementation because changing an object's state can simply happen by evaluating a piece of Smalltalk code in a tool's graphical interface.

Summary

Programmers have to work with all kinds of software artifacts. Source code is only one kind of artifact. To better understand the vices and virtues of tool support, we distinguish artifacts on a conceptual, structural, and transport level. Since we focus on object-oriented programming environments, artifacts are usually objects that expose structure through visual tool interfaces so that programmers can recognize concepts. There can be many different systems that hold the artifacts relevant for a given programming task. Programmers have to assess relevance, which can be driven not only by objective measures but also by individual knowledge and experience. Tools and environments typically hide an artifact's transport medium, yet they depend on it when sharing information among each other. Hence, programmers working in object-oriented environments may have to deal with other transport media such as tables, files, and streams from time to time.

Program comprehension takes up the majority of time in programming activities. Programmers have to learn and recall facts about the underlying theory and implications when making changes to the code as well as other artifacts. The process is iterative and consists of fetching and examining relevant software artifacts as well as retaining important insights. Since programming is also a creative and explorative activity, programmers have to perform many tasks concurrently. This is also due to elaborate and effortful sub-tasks for gathering more supportive artifacts. The act of switching between tasks takes time due to interruption and resumption lags.

In this work, we focus on *graphical* tools for programming. Such graphical interfaces do acknowledge the importance of text, but high-resolution input and output devices positively affect information tangibility. That is, programmers can get the impression of directly working with software artifacts as being tangible things on screen, which can be fetched or examined and also created or modified. Programmers usually work with a whole set of pre-selected tools: the integrated programming environment. In such environments, tools are integrated at the levels of presentation, data, process, and control.

Tools are usually generic to provide a high degree of re-use for any domain, task, or user. However, such generality impedes efficient application for specialized domains, tasks, or users. This affects, for example, efficient integration of custom artifact sources or interactive views. Means of configuration try to alleviate this problem. Tool users can easily modify aspects of the graphical interface such as colors, fonts, or keyboard shortcuts. Programmers, being more skilled users, can also employ scripting languages such as regular expressions to configure tool views. We also consider manual layouting of windows and other containers as configuration.

Finally, tool configuration can reach a limit where specific domains, tasks, or personal preferences cannot be accommodated sufficiently. Programmers can always act as tool builders and reshape programming tools to fulfill any requirement. Technically, tools are adapters to make software artifacts work with interactive, graphical widgets (or views) as desired. Therefore, tool builders employ presentation, query, and mapping languages. We focus in the object-oriented programming system Squeak/Smalltalk with Morphic as its graphical framework. There, everything is an object and every graphical object "feels" tangible through a special kind of user interaction. Building tools means describing run-time objects and set-up objects to connect artifacts to widgets. Considering source code, this means writing initialization code, which is executed once at set-up time, and callback code, which is executed frequently at run-time. Tool building is an iterative process with many cycles of finding code, changing code, and verifying the effects.

[|]

In the next chapter, we motivate the challenges of tool building with a story about program comprehension and inadequate tool support. We also show our solution, the Vivide environment, in a tutorial that tackles the story's challenge.

3 Motivation

The challenges of tool building are very clear in systems such as *Squeak/Smalltalk*, where relevant data and interactive graphics are *within reach*. Programmers have direct access to many software artifacts that help answer many program comprehension questions. Smalltalk images persist *long-living object graphs* with objects for all kinds of artifacts, not just code. Many *programming tools* that help manage this informational space are aligned to *generic* programming language concepts. Thus, programmers working on *specific* problems are likely to discover tool deficiencies during use. However, tool building in Squeak is still challenging because existing tools support traditional code-writing activities, not artifact exploration in general.

We think that tool-building frameworks in Squeak should *embrace* the characteristics of the interactive, self-supporting programming system it is embedded in. This includes the ability to find, change, and verify the rules for working with software artifacts including their tangible, graphical representations on screen. Consequently, *tools for tool building* should not only support traditional code-writing tasks. They should help manage *all kinds* of objects that make up the tools under construction.

In this chapter, we motivate our problem domain with an exemplary story in which a programmer wants to better understand a certain framework feature. The story illustrates several *triggers* that make the programmer want to modify the existing tools and eventually create a new one. We summarize the *barriers* according to our tool-building model, which consists of repeated cycles of find/change/verify. Finally, we explain our *idea* of a novel interactive, data-driven, script-based tool building environment, Vivide, which we use to solve the story's challenge to illustrate its main practices.

3.1 A Story of Code Exploration

In this section, we first clarify our notion of the programmer's mindset and the general approach to access artifacts relevant for a running application. Then, we present a scenario that highlights existing tool building challenges worth addressing. Note that we focus on *task-specific* tool adaptation because (1) tool building and tool using go hand in hand and (2) it may not be obvious whether future tasks can benefit from adapted (or new) tools. Also, any programmer is different, and we do not want to jump to conclusions about personal preferences. We argue that any issue with task-specific tool adaptation can also be mapped to domain-specific or global adaptations, were the expected benefits are high.

The Efficient Smalltalker

The mindset of efficient Smalltalk programmers is special in the sense that they try to *live* in their environment full of objects as long as possible. In Smalltalk images, all source code is accessible, even for dependent libraries and frameworks, which supports program comprehension tasks by making underlying patterns explorable. In Squeak 5.1, there are 95 packages with 2'260 classes having 52'338 methods, which approximates to 389'978 lines of code.[1] This covers everything from primitive types over networking to interactive graphics. A glance behind an object's interface into the method implementation might help discovering side effects or usage patterns without the need for browsing external documentation or on-line support forums. While there are some methods that are large and incomprehensible, many methods embrace Smalltalk's conciseness and are hence readable. While external documentation can be outdated, the actual source code is not. Consequently, one of the two most important tools is the *code browser*. The other one is called *object inspector*, which supports examining all kinds of artifacts—including code artifacts in a generic way that is inappropriate for changing code. Even though run-time is omnipresent in Smalltalk environments, there is a third tool, which is a combination of browser and inspector: *the debugger*. Debuggers show up whenever there is an unhandled exception to be examined by programmers. These three[2] tools form the basis for Smalltalk programming.

There is also "breakpoint debugging" in Squeak to explore the call stack of a suspended process. Programmers can inspect any object they find a reference to, which includes domain objects of the application and objects of the base system. While there are classes for `MessageSend` and `MethodContext`, the virtual machine (VM) will only reify respective instances on request because of performance reasons. Having this, a *breakpoint* is just a specific exception to be raised via `self halt` in application code. Since Morphic does only have a single UI process, a new UI process will be spawned if the breakpoint would render the system unresponsive. The debugger itself simulates the execution of Smalltalk code so that programmers can examine the control flow without knowing VM internals. Note that Smalltalk programmers have to manage system state in addition to code because side effects can be triggered over and over again when stepping through code in the debugger. Besides breakpoints, programmers can examine the state of any graphical application with the help of Morphic's halo concept as described before. If an object's state, meaning its instance variables, is readable, programmers can discover relevant code artifacts to modify or set breakpoints into. If, however, that state does not

[1]See the appendix A.2 for the code that was used to compute these numbers.

[2]Readers familiar with Smalltalk will miss the *workspace* as a place to try out smaller pieces of code. We omit this tool here because programmers can basically evaluate Smalltalk code in any other tool's text field with similar effects. We also ignore the *transcript* (or command-line output) because we focus on tools for interactive debugging and exploration sessions.

reveal the underlying classes that produced it, program comprehension can become challenging.

Remark For this motivational story, we simplify the availability of helpful information in the Squeak/Smalltalk environment. However, even the sense that the answer to a question lies "under one's very nose" can reveal many issues with generic, inconvenient programming tools.

The Task: Handling a Mouse Click

Imagine a programmer who works on a game. The game's graphical interface should have many different button-like elements where the player can click on to invoke in-game actions. The game is implemented in Squeak using Morphic as the graphics framework. Unfortunately, the programmer is unfamiliar with the framework, and so it happens that the first button-like element needs to be implemented. Since all graphical elements in Morphic are morphs, the button would be a morph, too. A colleague told the programmer that morphs that want to react to mouse input should implement the message `#mouseDown:`, which gets an event object as argument to distinguish mouse buttons and keyboard modifiers. After doing as told, the button seems to react to a click but not as intended: it is grabbed by the mouse cursor just like in a drag-and-drop operation. The custom callback is not even reached, but there seems to be a hidden default implementation the programmer was not aware of.

Since the Squeak system offers various ways of examining code artifacts, she tries to look for anything that provides clues about "event", "mouse", or even "mouse event" in combination. These terms form three different search queries, which all return confusing results with no clarification:

Search term	"event"	"mouse"	"mouse event"
Message name count	635	338	13
Code fragment count	2'220	1'454	26

The search for message names includes *class names*, and the search for code fragments includes *class comments*. Yet, even the most concise result sets for "mouse event" did not reveal artifacts that the programmer can relate to the initial hint about `#mouseDown:` in her code. Frustration arises because there seems to be no documentation about Morphic event handling accessible in the Squeak image, which is supposed to contain thousands of code artifacts and hence some readable clues about this event handling problem.

The programmer recalls that there are already buttons in the graphical interfaces of several tools. Those buttons (or button morphs) are expected to implement callbacks for mouse events somehow. After invoking a halo for the "versions" button in the code browser, the class `PluggableButtonMorphPlus` reveals itself in a text label (see figure 2.15). Unfortunately, the code hierarchy browser, which is a special code browser, also reveals that that class inherits from another button-related class called `PluggableButtonMorph`, which appears to document a rather complex implementation for a simple concept such as a button. In sum, both classes offer 90 methods. The programmer is still frustrated because the amount of code artifacts is overwhelming and was not anticipated.

Finally, the programmer undertakes an attempt to find *exemplary* implementations of `#mouseDown:`, hoping that some classes might offer context much simpler than the `PluggableButtonMorphPlus` code does. The implementors browser, which is another special code browser, reveals 88 different implementations in the image across many different packages. So, mouse-click handling seems to be very important and cannot be that difficult to implement correctly. Yet, exploring all 88 methods would be time-consuming. The list in the tool is *alphabetically* ordered, and the first implementation is in the class `AbstractResizerMorph`. The programmer is not expecting to find a simple answer to the problem and just browses this first result. This turns out to be a *lucky shot* because that class offers only 13 methods. There is a second callback, which is called `#handlesMouseDown:`, that needs to be implemented to return `true` and not (the default) `false`. This solves the problem and the programmer's implementation of `#mouseDown:` gets called on a mouse click. This solution suffices to accomplish the current task.

The programmer notices the message `#handlesMouseOver:` and wonders what other features the event handling in Morphic has to offer. A basic understanding might become beneficial in *future tasks* and help estimate the effort of upcoming feature requests for the game. As an unexpected side effect, the aforementioned *grabbing* of the morph disappeared, too. This is good because the programmer did already worry about how to get rid of this unintended behavior. Still, the basic mechanics of event handling seem rather obscure at this point. Since there is still some time left before closing the current task, the programmer begins to explore the internals of Morphic event handling.

The Task's Epilogue: Morphic Event Handling

The programmer now has a method at hand that executes event handling code, which makes *breakpoint debugging* feasible. The recent code search efforts did not yield satisfying results for understanding mouse click handlers anyway. After adding `self halt` to `#mouseDown:`, a debugger window appears, which shows a stack filled with 46 method activations. Each stack frame is basically a *message send* or a *code block* activation used for the body of conditionals and loops. At a first glance, the programmer notices several different classes such as `HandMorph` and `MouseButtonEvent`. In conclusion, the code artifacts for event handling seem to be spread across *many different classes* of the Morphic framework. Even though the stack seems rather long, the programmer gets the feeling of having access to many *relevant code artifacts* worth examining. The programming tools of *choice* are the code browser, the object inspector, and the symbolic debugger. Tool *configuration* is possible by means of list filters and window management.

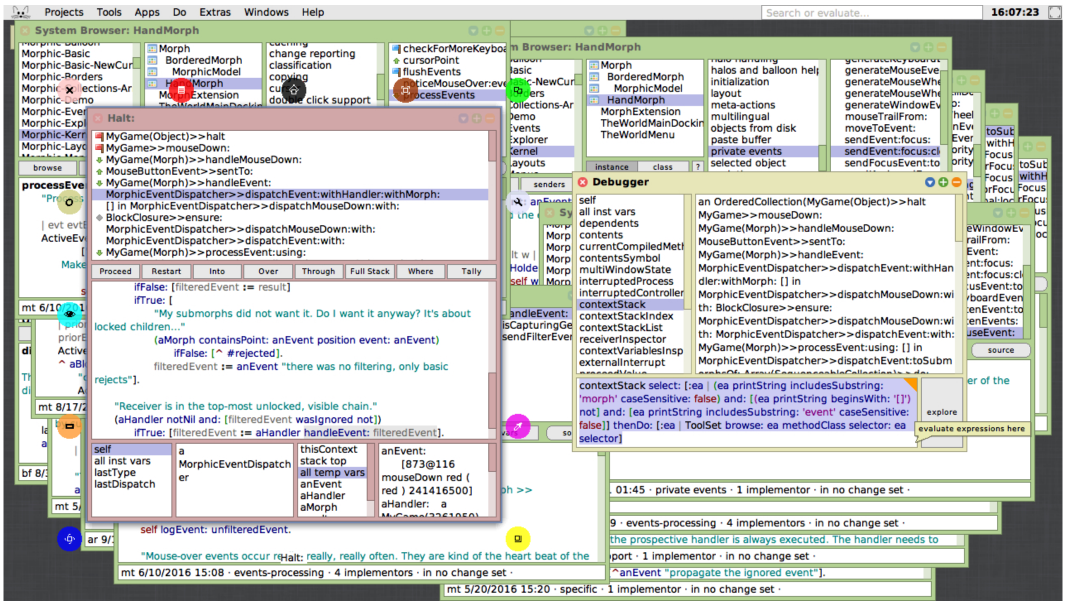
At first, the programmer tries to use the debugger's graphical interface to fetch and examine all relevant artifacts for the event handling concern. When browsing stack frames, reading code, or exploring other object state, she follows certain *rules*, which reflect her current understanding of the framework:

- I am looking for something about "morph" and "event".
- Such frameworks also know "keyboard" events besides "mouse" events.
- There might be events with information about "modifiers" or "time stamp".
- I do not want to see the block activations with "`[]`" but only methods.

These rules drive all browsing and filtering activities, which means clicking on buttons and bringing keywords into focus. Since the debugger shows only one method at a time, there is tedious back-and-forth navigation involved. The stack list can only be filtered for a single term such as "event" excluding wild cards or regular expressions. After a while, the programmer opens new code browsers to retain relevant code artifacts. Unfortunately, each browser window uses about 50% of its space for showing navigation context besides code. A certain level of frustration persists because back-and-forth navigation poses a high cognitive load, while additional tool windows contain much redundant information. These windows do also clutter the screen. In general, the programmer does not feel that the tool interaction leads toward the actual goal. Instead, the many mouse clicks seem to struggle over human forgetfulness. Side-by-side comparison of relevant information seems impossible because four code browsers can fill the entire screen.

The programmer recalls that, internally, the debugger has to have *access to objects* that represent the relevant code artifacts. In Squeak, everything is an object, which includes process context and stack frames. So, the programmer invokes the halo for the debugger tool window, and she directly finds the underlying `model` object. The model has an instance variable named `contextStack`, which reveals the desired artifact representation. From there, it is just a quick call to *access source code* for method activations. Since all object inspectors provide a text field for code evaluation, the programmer attempts to *configure* her programming experience with some Smalltalk scripts. A first expression should externalize her thoughts about relevancy and filter the stack list accordingly:

```
contextStack select: [:methodActivation |
  ((methodActivation printString includesSubstring: 'morph'
                                  caseSensitive: false)
      or: [methodActivation printString includesSubstring: 'event'
                                         caseSensitive: false]
      and: [(methodActivation printString beginsWith: '[]') not])].
```



Halt:

```
MyGame(Object)>>halt
MyGame>>mouseDown:
MyGame(Morph)>>handleMouseDown:
MouseButtonEvent>>sentTo:
MyGame(Morph)>>handleEvent:
MorphicEventDispatcher>>dispatchEvent:withHandler:withMorph:
[] in MorphicEventDispatcher>>dispatchMouseDown:with:
BlockClosure>>ensure:
MorphicEventDispatcher>>dispatchMouseDown:with:
MorphicEventDispatcher>>dispatchEvent:with:
MyGame(Morph)>>processEvent:using:
```

Proceed | Restart | Into | Over | Through | Full Stack | Where | Tally

```
		ifFalse: [filteredEvent := result]
		ifTrue: [
			"My submorphs did not want it. Do I want it anyway? It's about
locked children..."
			(aMorph containsPoint: anEvent position event: anEvent)
				ifFalse: [^ #rejected].
			filteredEvent := anEvent "there was no filtering, only basic
rejects"].

	"Receiver is in the top-most unlocked, visible chain."
	(aHandler notNil and: [filteredEvent wasIgnored not])
		ifTrue: [filteredEvent := aHandler handleEvent: filteredEvent].
```

```
self            a MorphicEventDispatcher    thisContext     anEvent:
all inst vars                               stack top           [873@116
lastType                                    all temp vars   mouseDown red (
lastDispatch                                anEvent         red ) 241416500]
                                            aHandler        aHandler:   a
                                            aMorph
```

Debugger

```
self                        an OrderedCollection(MyGame(Object)>>halt
all inst vars               MyGame>>mouseDown:
dependents                  MyGame(Morph)>>handleMouseDown:
contents                    MouseButtonEvent>>sentTo:
currentCompiledMeth         MyGame(Morph)>>handleEvent:
contentsSymbol              MorphicEventDispatcher>>dispatchEvent:withHan
multiWindowState            dler:withMorph: [] in
interruptedProcess          MorphicEventDispatcher>>dispatchMouseDown:wi
interruptedController       th: BlockClosure>>ensure:
contextStack                MorphicEventDispatcher>>dispatchMouseDown:wi
contextStackIndex           th: MorphicEventDispatcher>>dispatchEvent:with:
contextStackList            MyGame(Morph)>>processEvent:using: [] in
receiverInspector           MorphicEventDispatcher>>dispatchEvent:toSubm
contextVariablesInsp
externalInterrupt
```

```
contextStack select: [:ea | (ea printString includesSubstring:
'morph' caseSensitive: false) and: [(ea printString beginsWith: '[]')
not] and: [ea printString includesSubstring: 'event' caseSensitive:
false]] thenDo: [:ea | ToolSet browse: ea methodClass selector: ea
selector]
```

```
self logEvent: unfilteredEvent.

"Mouse-over events occur really, really often. They are kind of the heart beat of the
mt 6/10/2016 15:08 · events-processing · 4 implementors · in no change set ·
mt 5/20/2016 15:20 · specific · 1 implementor · in no change set ·
01:45 · private events · 1 implementor · in no change set ·
9 · events-processing · 4 implementors · in no change set ·
the prospective handler is always executed. The handler needs to
port · 1 implementor · in no change set ·
^anEvent "propagate the ignored event"].
```

Figure 3.1: After fetching and examining code artifacts through the debugger's run-time state, the programmer invokes code browsers for all several methods. The result is rather messy, yet it retains the relevant artifacts on screen. In the front, there is a debugger window (left) and an inspector window (right).

The use of the objects' textual representation via `#printString` avoids the need to learn the programming interface of `MethodContext` instances. After constructing and printing a simple string, the list of relevant methods goes down to 17, which are spread across only 5 classes: `HandMorph`, `MorphicEvent`, `MouseButtonEvent`, `Morph`, and `MorphicEventDispatcher`. Unfortunately, the use of strings results in the loss of *tangible representations* for the relevant software artifacts.

The first workaround that comes to mind is the *programmatic invocation* of code browsers. The programmer knows that the message `#browse:selector:` sent to the class `ToolSet` would do so. The effect, however, is rather unsatisfying as depicted in figure 3.1. The screen is cluttered again, and programmatic window management seems not straightforward. Frustration remains because the programmer has difficulties to separate *tool-support* objects from *task-relevant* objects. This is because the generic object inspector does only reveal how underlying frameworks make use of objects to construct interactive, graphical tools.

In retrospect, manual scripting against tool state seems to be inefficient to better understand Morphic event handling. Maybe a new tool can help by showing all or many relevant methods side-by-side. The programmer has still a little bit time left before having to attend to the project's next task.

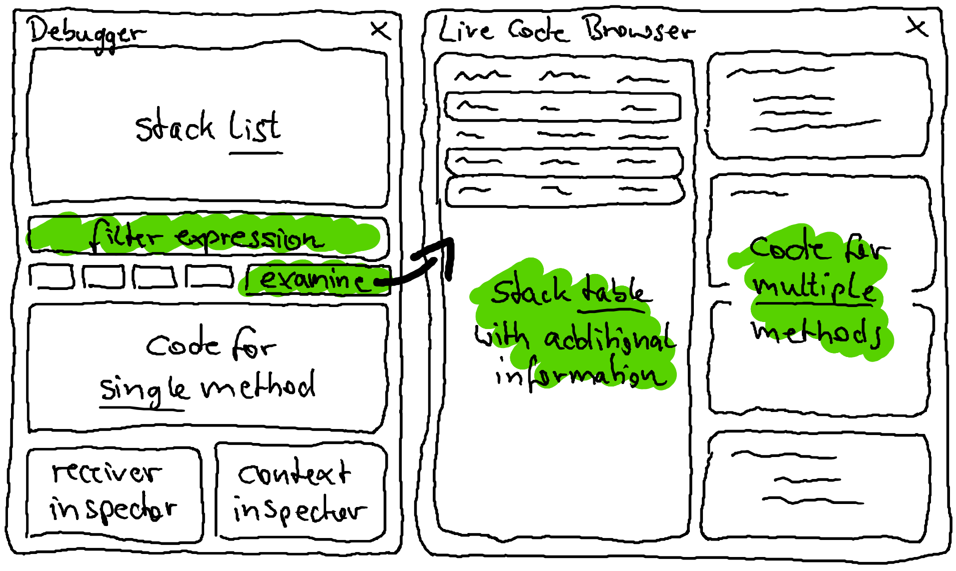


Figure 3.2: The programmer's idea for a better code reading experience is twofold: (1, left) stack filtering with elaborate expressions in the existing debugger and (2, right) a new tool that can retain and present relevant code artifacts more concisely.

Toward a New Debugging Tool

The programmer imagines a new debugging tool, which might also be useful for upcoming program comprehension issues. Using our understanding of programming tasks from section 2.2, the main issues with the current experience are as follows:

1. *Fetching* code artifacts for method activations or event artifacts for method arguments is tedious because you cannot easily open browsers or inspectors for many artifacts at once.
2. *Examining* the stack list is challenging because filter expressions are limited. Examining the source code works only one method at a time inside the debugger window. The same applies to event objects in the embedded inspectors.
3. *Retaining* relevant code artifacts or event artifacts in terms of tool windows is difficult because code browsers exhibit much redundancy and occlusion due to poorly designed layout. The same applies to object inspectors.

So, the programmer wants to be able to enter advanced filter expressions in the debugger as depicted in figure 3.2. She also wants to be able to spawn a new kind of code editor, which takes the currently visible stack frames as input, and which supports showing multiple code artifacts and other supportive information side-by-side without cluttering. There is no need to invent a domain-specific filter language, but *plain Smalltalk* should be used to access and select instances of `MethodContext` as shown in the inspector in figure 3.1. The new tool window should be labeled "Live

Code Browser" because the additional information in the left-hand stack table will be derived from actual code execution such as the print-string of method arguments.

As hinted before, the programmer has basic knowledge about the tool building framework that serves Squeak's programming tools. Looking at these tools, there is typically a single class that represents a "fat" model and hence contains the *initialization code* (or set-up objects) and all *callback methods* (or run-time objects) for connecting artifacts with graphics. The entry point for the graphical setup is the method `#buildWith:`, in which the programmer declaratively describes the widget layout and model callbacks. The model's state holds direct references to software artifacts, while it is decoupled from the actual widget morphs and only broadcasts signals for model changes. In the debugger code, the programmer sees a mixture of context stack management *and* widget state management. This might be challenging when inspecting the model state via the halo of the debugger window. From the recent code browsing experience, the programmer recalls that `contextStack` holds the relevant software artifacts for this tool building task.

Unfortunately, several challenges arise when the programmer tries to change the debugger. The model class itself has a rather long inheritance chain with numerous methods in each class: `Debugger` (128), `CodeHolder` (164), `StringHolder` (88), `Model` (33). Considering viable entry points, there are even 12 methods whose selectors match the pattern "build*with*". The programmer concludes that the debugger is a complex application of the tool building framework. Even her plan to add a new button to the button row turns out to be not straightforward because there are two build methods for button rows: `#buildControlButtonsWith:` and `#buildOptionalButtonsWith:`. When moving on from code to state, the programmer discovers a total of 17 instance variables. Three of these variables seem to manage the context stack in unison: `contextStack`, `contextStackList`, `contextStackIndex`.

After these rather confusing observations, the programmer is not so sure that she can extend the debugger without breaking it. Since a working debugger is paramount for future programming tasks, the programmer's plan includes only the new "Live Code Browser" for now. Such an extra tool can easily be opened by inspecting the debugger state again and extracting artifacts from `contextStack` as needed without interfering with debugger internals.

The programmer looks into the implementation of Squeak's Object Inspector, which has a smaller code base than the debugger. It turns out that the debugger actually *re-uses* the inspector model to reduce implementation effort for inspecting receiver and context objects as explained in figure 2.15. After some inspiration, the programmer manages to create an empty list in a window. The following code-writing activities reveal a major drawback in this framework: there is too much initialization code. Thus, changes in `#buildWith:` require tool restarts to make the changes effective. However, this feedback loop is essential since the code for widget layout is abstract, and mistakes are likely. It surprises the programmer that the Morphic halo cannot be used to describe the geometry of graphical applications in an interactive fashion. Having to use abstract code in the first place feels like a throwback to not self-supporting environments. After some practice, the new tool shows a table with information about method activations that are more explanatory than in the debugger. Then, a second issue arises for the planned list of source code for multiple methods. The attempt to re-use the `CodeHolder` model disappoints the programmer because it entails not only a code editor but also an unintended button bar. Another attempt to use regular text widgets reveals additional efforts to add syntax highlighting and the possibility to save source code changes. As the final challenge, the programmer cannot figure out how to put a set of code editors into a scrollable container to support browsing more than three or four methods at a time. Eventually, this tool-building attempt comes to an unfinished end.

Retrospective

In retrospect, the programmer did manage to complete the actual programming task about handling a mouse click in a button-like element for the game's graphical user interface. With mixed feelings, she returns to the debugger and reads more code from `HandMorph` and `MorphicEventDispatcher`. There are some methods with longer comments inside that document intentions behind event dispatching in its current form. Yet, the programmer does not understand why, in the beginning, the button morph was grabbed by the cursor automatically. Anyway, the time scheduled for this programming task is up. She decides to ask a colleague in the next sprint meeting. Though, the programmer learned more aspects of tool building in Squeak. Those insights can help estimate dedicated tool building tasks if the team agrees that such a new tool could be worthwhile.

Synopsis For program comprehension, efficient Smalltalk programmers embrace the means of self-supporting environments. External documentation is among the last resorts because the systematic, interactive exploration of run-time artifacts and thousands of code artifacts is likely to reveal insights.

Starting with vague hints, an imaginary programmer has to implement a mouse handler for a button-like element in a game. It takes several attempts of text-based code search and browsing exemplary implementations until the programmer discovers that `#handlesMouseDown:` is required in addition to the `#mouseDown:` handler. Eventually, the programmer wants to learn more about Morphic event handling.

The traditional means of breakpoint debugging and interactive exploration lead to the desire of (1) extending the debugger with better stack filtering and (2) creating a new tool that can show valuable information side-by-side. Unfortunately, the efforts to improve fetching, examining, and retaining of relevant code and event artifacts do not come to a satisfying end.

3.2 Triggers and Barriers for Tool Building

The story from the previous section illustrates a common problem that generic programming tools exhibit. When working with domain-specific artifacts, programmers might notice deficiencies and come up with ideas on how to better understand the information at hand. This is also true for the Squeak/Morphic environment where the tools can directly augment abstract code with concrete run-time information.

In this section, we provide more details about the triggers and barriers for tool building. We end with our *research question*, which is worth answering because it would open perspective on how to improve programming tools to fetch, examine, and retain task-relevant software artifacts to work more efficiently.

Remark We draw conclusions from a study about tool customization in 1991 [107], where many programmers in a Unix-based environment were involved. In that study, prominent triggers include spontaneity or curiosity to learn or fix something for managing the current task. A high level of difficulty and lack of time were among the usual barriers. A more recent study from 2012 [11] tries to empirically validate those findings, but it employs only simple graphical user interfaces and hence excludes a use of programming skills.

One Programmer, One Task, One Source of Inspiration

The design and implementation of valuable programming tools can be a challenging endeavor. Looking at existing tools, there were many decisions made to clarify the requirements and many hours invested to fix bugs and increase robustness. Consequently, there are many programmers and scenarios involved to achieve high-quality tools. Yet, we argue that such a long process is driven by many individual programmers that work on certain programming tasks and that discover deficiencies in tools on their own. Hence, we focus on a single programmer that realizes a poor selection of tools or a deficiency in a tool's configuration interface. This is where plans emerge to build new tools or modify existing ones.

When *fetching* artifacts that can help understand the task, we see the following triggers for tool building:

- Given a single tool, integrate more (or less) artifacts into the tool's views.
  In our story, the programmer wants to fetch multiple method objects at once while hiding irrelevant stack frames.
- Given two or more tools, create a shared awareness of relevant artifacts across all tools.
  In our story, the programmer wants to share the concept of method activations between debugger and new tool.

- Given a set of artifacts, derive new structure or new concepts from the existing information.
  In our story, the programmer wants to show additional information in a stack table.

When *examining* artifacts to assess relevance of information, we see the following triggers for tool building:

- Poor tangibility in views and hence issues to identify artifacts.
  In our story, the programmer wants direct access to methods looking at the stack list.
- Poor means of configuration considering the mapping language and rich artifact structure.
  In our story, the programmer wants to modify the labels in the stack list.
- Poor choice of views considering the presentation language.
  In our story, the programmer wants to change the stack list to a stack table.
- Poor selection of containers and interaction patterns and hence issues to examine multiple artifacts with little user interaction.
  In our story, the programmer wants to construct a list of method views.

When *retaining* artifacts that help recall relevant insights, we see the following triggers for tool building:

- Inadvertent redundancy when holding on to tool windows that show the relevant information.
  In our story, the programmer wants to omit the upper part (or 50%) in each code browser.
- High manual effort required for moving and resizing tool windows to find trade-offs in the limited screen real estate.
  In our story, the programmer wants to layout dozens of code-browser windows.
- Views that do not show artifact structure needed to recall insights later on.
  In our story, the programmer wants to show additional information in the stack table.

These are all *task-specific* triggers, which are also driven by the current programmer's knowledge and preferences. Still, it is likely to find domain-specific opportunities if tool building efforts turn out to be beneficial repeatedly in other tasks. In that case, there might be more resources available to make tools useful for many tasks and the whole team that is working on the project. Consequently, if tool building would be too expensive to be carried out as an additional activity in *any* task, we see two risk factors. First, single programmers can miss smaller productivity boosts. Second, some domain-specific tools might never be discovered at all. That is why we focus on tool building in the scope of the task at hand. Nevertheless, tool building is an iterative process where programmers keep on switching between *building* and *using* the tools under construction anyway.

The Barriers to Overcome

The circumstances for building new programming tools look promising. We have a programmer that has an *idea* on how to improve a certain tool that is currently used to accomplish a program comprehension task. The programming environment, in which the tool is running, is *self-supporting* and *reflective*, which means that the objects that produce the tool behavior are accessible and changeable while the tool is running. The programming language of choice is Smalltalk, which is capable of expressing thoughts and ideas quite *concisely* in a *readable* fashion. The Morphic graphics framework entails the *halo* mechanism, which is a meta menu to explore code and state of any graphical element on screen. *Still, why is tool building so challenging in Squeak/Smalltalk?* It turns out that the find-change-verify cycle for tool building is costly for various reasons.

Find Barrier First, it is difficult to *find* the artifacts to change because a halo exposes plain object state that is often derived from generic resources. If the inspected object does not inherently point to domain-specific code artifacts, programmers will have to fall back on text-based code search again. For example, any morph can be configured without custom classes to behave like a button via `aMorph on: #click send: #open to: anotherMorph`. A halo for such a graphical object might just point to the `Morph` class and not to the tool's set-up code. We do not argue that the flexibility of Smalltalk poses an inherent challenge. Instead, we think that the halo for any graphical tool object should be more specific to be useful for tool building tasks. In Squeak's standard tools, the `model` is rather discoverable for any graphical `window`. Actually, all programming tools subclass from the class `Model`. In any specialized model subclass, however, the `#buildWith:` message(s) can be overlooked if there is much other code. In the Moldable Inspector [30] framework, the graphical interface for inspector tools has an extra *meta* button for finding the tool's means to query, map, and present artifacts. Yet, for many other tool elements, including the meta button, there is no meta button. The generic halo does still exist but becomes practically useless because set-up code disappears behind numerous dynamically configured framework objects.

Change Barrier Second, it is difficult to *change* artifacts to express different tool behavior because domain-specific code is scattered and mixed with framework glue. Many object-oriented tool frameworks employ means of modularization and re-use such as inheritance and polymorphism to simplify tool building. Unfortunately, many framework designers anticipate programming environments that are centered around plain code editing. For expressing a presentation language in the graphical interface, interactive editors with code generators are state of the art as explained via "Interactive Graphical Tools" by Myers et al. [129] For querying artifacts and mapping visual properties, however, frameworks usually fall short of interactive tooling. On the one hand, we see great value in descriptive programming interfaces that are easy to change. On the other hand, even a simple description of a dominant tree structure in an object graph can involve many different classes and methods when following best practice patterns. Consequently, this entails effortful changing of glue code, too, even in Smalltalk frameworks. We argue that there is tool support missing that exposes not only a framework's presentation language but also its query and mapping languages. Considering task-specific tool adaptation, we want to avoid means that require changes in the actual domain code such as methods for `#printString` or `#printHtmlString`.

Verify Barrier Third, it is difficult to *verify* correctness of changed artifacts because tool frameworks do not support selected, coherent updates of running tools. A usual reaction to changed source code is a complete tool restart, which is rather coarse-grained and makes verification hard. The programmer might have to repeat much tool interaction to reconstruct tool state if even possible. A more guided, or restricted, code modification can be used to define selected tool updates as demonstrated in the Moldable Inspector [30]. However, we see issues with such tool frameworks not considering the Smalltalk environment as a self-supporting system, which has not only code objects. As illustrated in the story above, programmers can inspect and change *any* tool object to better understand the concepts and theory behind. Consequently, tool frameworks that want to support the creation of long-running tools should recognize object modification beyond code. We think that programmers can benefit from additional assistance that tool building frameworks should provide to scope tool object modification.

[|]

To our knowledge, existing tool frameworks exhibit at least one of the aforementioned barriers: (1) the halo not providing access to tool set-up objects, (2) no tooling for separating query/mapping/presentation languages from glue code, and (3) not considering object change besides code change in self-supporting environments. This leads to the research question of this work:

Research Question How can we support tool building in object-oriented, graphical environments where programmers mainly fetch, examine, and retain information in the form of tangible software artifacts through interactive views?

We think that there is consent among programmers that domain-specific tools are beneficial. However, we are not so sure whether each individual programmer in a team feels *encouraged* to try out ideas that *only might* become valuable tools for the whole project—or that might just waste time. We apprehend the occurrence of a psychological phenomenon called "learned helplessness" [134, p. 42], which we think is true for tool building. If tool building is too expensive, there has to be a dedicated tool building task scheduled. Since that dedicated task is out of context, any tacit knowledge from the “inspirational” task will not be available to the tool builder and hence only documented, recurrent scenarios will be addressed. If there are only domain-specific tools for recurrent scenarios, rare or unique scenarios will never get appropriate tool support. We see the risk of programmers being shaped by their tools and not thinking about alternatives. Consequently, our research question does not only address the need to save time in tool building tasks. We also want to increase the likelihood of *discovering and specifying* any supportive tools in the process.

Synopsis We think that programmers are able to detect deficiencies in the programming tools they use for fetching, examining, and retaining software artifacts in program comprehension tasks. A failure in understanding is often triggered by tools not being able to integrate and present all relevant artifacts in a tangible way.

Even in the Squeak/Smalltalk environment, tool building is difficult because the Morphic halo does not provide direct access to tool set-up objects, query/mapping/presentation languages mix up with glue code, and any means for dynamic tool updates do not align with typical Smalltalk object-inspection and debugging practices. By making the tool building process cheaper, the time savings might also be used to discover new tools for rare or unique challenges in a project.

3.3 Proposal: An Interactive, Data-driven Tool Building Environment

We see tool building activities as attempts to improve the perceived tangibility of accessible software artifacts. Programmers use graphical interfaces that show selected artifact structure in a way that helps understand (or recall) domain-specific (or task-specific) concepts. Consequently, the notion of tangibility addresses (1) the tool’s means to handle data and (2) the tool’s means to handle graphics. Both aspects should be under the programmer’s control and hence ready to be modified if necessary. Note that, for program comprehension, “data” relates more to fetching and examining artifacts, and “graphics” relates more to examining and retaining artifacts.

In this section, we explain our idea for a new tool building environment. We end with this work’s thesis statement that answers the aforementioned research question. In the next section, we then explore this hypothesis with a demonstration of the idea’s working implementation in Squeak.

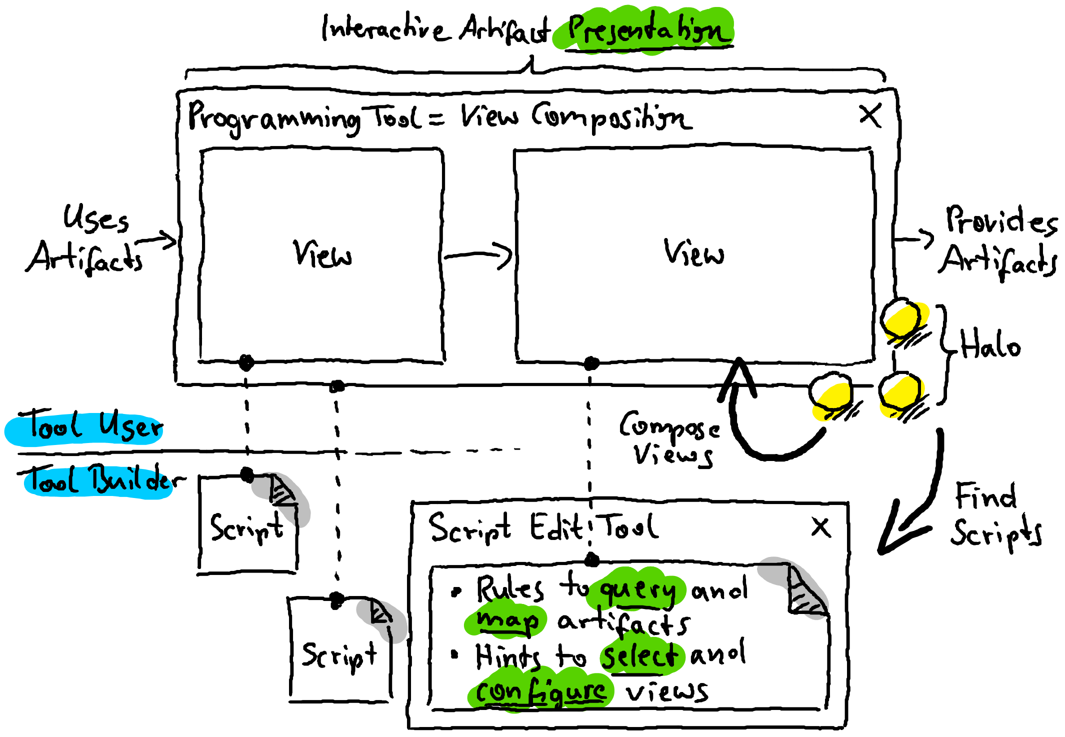

Interactive Artifact Presentation
Programming Tool = View Composition
Uses Artifacts
View
View
Provides Artifacts
Halo
Tool User
Tool Builder
Compose Views
Find Scripts
Script
Script
Script Edit Tool
• Rules to query and map artifacts
• Hints to select and configure views

Figure 3.3: Our idea for an interactive, data-driven tool building environment encompasses tool support for (1) composing the graphical interface and (2) authoring scripts that query artifacts and map visual properties for selected views. The Squeak/Morphic halo mechanism contributes to the overall usability.

Simplicity Through Data-driven Scripting

Since tool users benefit from tools for fetching and examining software artifacts, we want to help tool builders focus on the processing of these artifacts in tool code. For this, we want to encapsulate the *rules* for querying and mapping artifacts in *scripts* as depicted in figure 3.3. Also, programmers should benefit from a *script-edit tool* that embeds *concrete software artifacts* into *abstract script code*. Having this, tool builders could not only *access* artifact structure but also *derive* new information. Note that artifact processing entails many operations such as sorting, filtering, and merging multiple sources. In general, scripts should help hide much tool glue code in the underlying tool building framework.

By exposing only such scripts to tool builders, we reduce programming to the *functional portion* that is included in traditional object-oriented tool code. We see value in structuring a tool's set-up objects and run-time objects with the help of object-oriented idioms and patterns. However, the functional, artifact-centered perspective is usually not easily accessible because these idioms and patterns dominate the code at the expense of readability. We draw inspiration from the pipes-and-filters pattern [25, pp. 53–70], of which Unix shell scripting is a common application. In

Unix, filter programs [153, pp. 266–267] perform small processing steps with a high level of re-use by following the principle: "Be generous in what you accept, rigorous in what you emit." Filters can be combined via pipes to process a stream of bytes or characters. The Unix shell represents the run-time environment for redirecting the output of one filter as input to another one. A combination of filters can be encapsulated into a shell script, which becomes an executable file, and then serve as a filter on its own. The following example illustrates the simplicity of configuring and combining filters to fulfill many different data processing tasks:

```
curl http://en.wikipedia.org/wiki/Unix \
  | grep -o -P 'href="/wiki/.*?"' | sort | uniq \
  | sed -r 's/href="(.+)"/http:\/\/en.wikipedia.org\1/g' \
> urls.txt
```

These lines will execute in many shells on Unix-like systems such as in *bash* on *Linux*. As a script, it retrieves the HTML content of the Wikipedia article about Unix (`curl`), extracts relative URLs that point to other articles (`grep`), sorts them (`sort`), removes duplicates (`uniq`), transforms relative URLs into absolute ones (`sed`), and writes the output into a file (`urls.txt`).

By emphasizing the functional programming paradigm for data processing, we do not address "dataflow" but a "data-driven" perspective. We see the latter as the more general version of the former. From a technical perspective, the tool execution environment can dictate flow (or streaming) semantics like Unix filters do. From an application perspective, data may actually flow in terms of sensor data appearing at a certain frequency waiting to be processed or discarded. With our focus on graphical tools for program comprehension tasks, we see value in distinguishing automated processing from interactive, stepwise exploration. Yet, we build on research insights from pure *dataflow languages* such as Lucid [209] and LUSTRE [72]. Since there is an omnipresent notion of run-time information in Squeak, we also consider the domain of *scriptable debugging* [102, 111, 87]. There, domain-specific scripts are used to efficiently trace and fetch valuable portions of program run-time information. As explained in section 2.5, we assume that for the tool builder all relevant artifacts are accessible through an object representation. Revisit figure 2.12 and figure 2.14 for reference.

Simplicity Through Ad-hoc View Composition

Since tool users benefit from tools for examining and retaining software artifacts, we want to help tool builders focus on the graphical presentation of artifact structure. It is common to use interactive tools for designing graphical interfaces. Such tools usually generate code stubs for the underlying graphics framework, which have to be implemented after the design phase. In Squeak, there is much potential to modify running tools without having to close and restart them. The Morphic halo directly supports changing some layout properties such as position and size. We think that it should be possible to design the tool's graphical interface in its running and usable state showing actual artifacts. Such a short feedback cycle should include adding, removing, and composing views in any layout style such as overlapping or tiled. The Fabrik project [79, 106] did already demonstrate the feasibility of such an approach for the domain of simulation applications, which are also data-driven and naturally included some programming tools.

Since views (or widgets) should support a wide range of artifact structure, we prefer separating *model code* from *view code*. Such an architecture for graphical applications has a long history [36]. For example, the Model-View-Controller pattern [25, pp. 125–143] was created for Smalltalk-80. The more generic Model-View-Presenter pattern [149] was derived for later systems. Unfortunately, not even in Smalltalk systems it was possible to directly connect the data (model), which would be software artifacts, to views. Instead, a "view model" (or "virtual model" [98, pp. 7–12]) was required to adapt between the different programming interfaces. In our idea, we want to *generate* such supportive tool objects using our scripts mentioned above. Tool builders should focus on the *selection* and *configuration* of tool views, which could also be embedded in the aforementioned scripts. Most "glue" should be hidden in the tool's runtime. As depicted in figure 3.3, the direct relationship between views and scripts should simplify the tool building process. As explained in section 2.5, we assume that for the tool builder all kinds of elementary views are available such as text fields, lists, tables, buttons, and charts.

Even though our idea relates to several concepts from Unix shell scripting, we do not strive for adding a modular, graphical interface for existing filter programs in a file/stream-based environment like former graphical toolboxes did. Examples include DEC Fuse, HP SoftBench, and Sun SparcWorks. There is also a more recent project that tries to wrap Unix filters into ActiveX components to use the Java platform for interactive view composition [183]. Instead, we want to embrace many features of the object-oriented, self-supporting, interactive Squeak/Morphic system first and foremost. For example, the Morphic implementation illustrates that there is largely no need to further separate controller (or presenter) code but encapsulate it within the view code. In general, it should be possible to use any view (or widget) that has any object-oriented architecture implemented in Smalltalk for Squeak.

Opportunities for Find, Change, and Verify

Given some software artifacts, the programmer can add new views in a default configuration with the help of pop-up context menus. As shown in figure 3.3, views can be connected to each other to *use and provide* artifacts for automatic script processing or manual user interaction. Having this, further tool building activities should work as follows.

Find Opportunity Programmers can *find* the scripts, which store the rules to query and map artifacts, via each view's halo. Several mouse clicks can be necessary to access the halo of a nested view composition. Any halo should also support inspection of the view's *input* artifacts and *output* artifacts. It can be useful to provide a *list of scripts* to quickly exchange some part of the tool. For the current script, programmers can open a script-edit tool.

Change Opportunity Programmers can *change* scripts in a code editor that only shows code that represent the rules for querying and mapping artifacts. We think that the Smalltalk language, or a subset of it, would be well-suited to express these rules concisely. For querying, this means evaluating any Smalltalk code on the input set and produce any output set with the intent of filtering, sorting, or navigating the object graph. For visual mapping, this means extracting selected artifact structure as objects having specific roles such as "text", "icon", or "tool tip". Consequently, views should document their mandatory or optional display roles. The absence of a mandatory role, such as "text", should not render a view unusable but fall back on conventional means to display artifacts. Note that programmers can also select different views and make general view configurations, such as fonts and colors, by annotating scripts accordingly.

Verify Opportunity Programmers can *verify* immediately because the framework will be able to update the affected views if scripts change. The explicit connection between scripts and views is used to avoid a complete restart for tools that consist of multiple views, which is common for programming tools. Such controlled, consistent tool updates support short feedback loops and hence encourage explorative tool building. Multiple views have to be updated only if they depend on each other by sharing artifacts like browsing tools do. If the script input changes, the script output may change, too. These automatic tool updates suggest that scripts should not have side effects and hence follow the functional paradigm. The programmer cannot be aware of all script evaluations like she cannot anticipate all message sends in an object-oriented environment.

[|]

Given such a short, albeit idealized, feedback cycle for tool building, we form the following working hypothesis in this work:

Thesis Statement In a tool building framework that describes graphical tools as compositions of data-driven, script-based, interactive views, many common programming tools can be expressed in that design, and the results will be easy to modify directly during use to accommodate specific domains and tasks.

The feasibility and approval of our idea depends on efficient integration with existing artifacts, widgets, and tools in the environment. For Squeak, such integration addresses existing object-oriented libraries and existing tools that already handle relevant software artifacts. For example, whenever a script should sort a list of artifacts, sorting algorithms written in Smalltalk should be used. When an existing widget is yet incompatible with our framework, it should be a *one-time effort* to write an adapter to ensure compatibility. When there is no script-based tool to examine some artifacts, the programmer should be able to fall back to the conventional tools of the environment. While we think that a whole environment only filled with script-based tools would be most beneficial, we do not investigate the process that would produce such an outcome. While it should be *simple* to create and change *simple* program comprehension tools, more *complex* tools should still be *possible* to construct.

Synopsis Our idea for an interactive, data-driven tool building environment builds on the observation that programmers benefit from tangible representations of relevant software artifacts in tools. Having this, the tool architecture and the tools for tool building should support (1) expressing the rules to query and map artifacts and (2) composing views interactively to form the visual representation. Such a modular design can hide much glue (code) in the framework and hence positively affect the find-change-verify cycles in tool building. For feasibility and approval, our idea integrates with existing artifacts, widgets, and tools in the environment.

3.4 Vivide by Example

Our imaginary programmer from section 3.1 receives a tip from her colleague to install the Vivide framework into Squeak and try again. The installation process replaces the traditional code browser, object inspector, and debugger with script-based versions of these programming tools. As described above, the programmer still wants to learn more about Morphic event handling in context of her game project. Given the custom mouse-down handler in the domain-specific button class, she still has an accessible point in Morphic's control flow to set a breakpoint. Again, a debugger window shows up and reveals many relevant parts of Morphic's event dispatching and handling in the form of method activations (or stack frames).

As depicted in figure 3.2, our programmer has ideas about how to improve the means of fetching, examining, and retaining relevant code artifacts and event artifacts. In detail, it should be easier to work with multiple artifacts at once, filter the stack using elaborate patterns, reduce back-and-forth navigation in tools, control visual representation to avoid redundancy, and embrace side-by-side comparison of relevant artifact structure. Remember that it is not crucial whether her tool changes turn out to be beneficial for this comprehension task or not. Yet, she might even discover valuable tools for prospective tasks in her game project. That is the mindset for *exploratory tool building*.

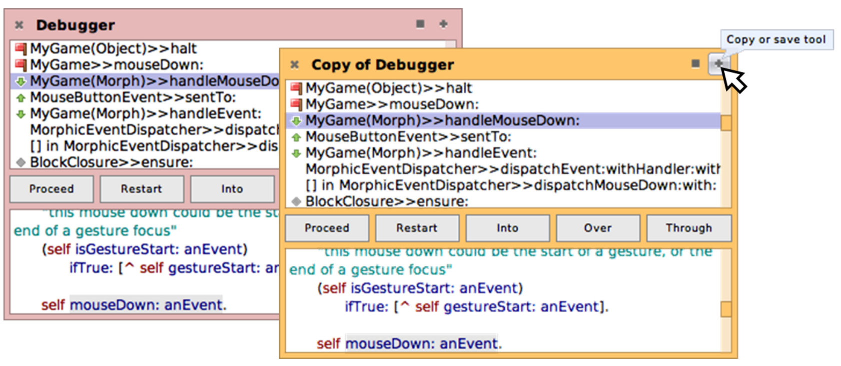
Remark This tutorial illustrates simple yet frequent interactions in Vivide. It's goal is to exemplify the programming experience from the perspectives of both tool user and tool builder. Since the actions taken in Vivide presume a new, or at least different, kind of problem-solving strategy, they are likely to require training, even if appearing straightforward to accomplish.

The following illustrations represent actual screenshots from a working Squeak 5.1 environment with Vivide installed. However, we idealize the visual impressions for reader focus with the help of resized windows, cropped images, added callouts, and reduced background noise. There are always real software artifacts involved, and hence the programmer shapes programming tools directly in use. See appendix B.1 for code listings.

Finally, the script-based debugger in this tutorial omits the two object inspectors in the bottom third compared with Squeak's standard debugger. This slightly cleaner appearance indicates a general observation we made in Vivide. Such script-based tools are less monolithic and hence embrace flexible integration among other tools in the environment. When debugging, it is therefore still possible for the programmer to examine the respective artifacts such as method arguments.

Tool Copy for Safe Trial-and-Error

The programmer makes the deliberate decision to modify an existing tool that is used in many programming tasks: the debugger. Therefore, she makes a copy of the debugger by clicking on a small button in the upper right window corner, which any script-based tool in Vivide exhibits. The tool copy opens directly besides the original but with a different window color and title prefix "copy of" for easier distinction. Otherwise, the tool views look identical because the underlying software artifacts remain shared. Since each script-based tool is typically represented as a single window, the programmer can directly estimate copy actions based on her particular intentions for tool building.



```
MyGame(Object)>>halt
MyGame>>mouseDown:
MyGame(Morph)>>handleMouseDo
MouseButtonEvent>>sentTo:
MyGame(Morph)>>handleEvent:
MorphicEventDispatcher>>dispatc
[] in MorphicEventDispatcher>>dis
BlockClosure>>ensure:
```

```
   "this mouse down could be the st
end of a gesture focus"
    (self isGestureStart: anEvent)
        ifTrue: [^ self gestureStart: ar

    self mouseDown: anEvent.
```

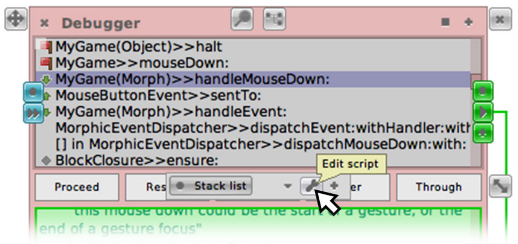
```
MyGame(Object)>>halt
MyGame>>mouseDown:
MyGame(Morph)>>handleMouseDown:
MouseButtonEvent>>sentTo:
MyGame(Morph)>>handleEvent:
MorphicEventDispatcher>>dispatchEvent:withHandler:witl
[] in MorphicEventDispatcher>>dispatchMouseDown:with:
BlockClosure>>ensure:
```

```
   "this mouse down could be the start of a gesture, or the
end of a gesture focus"
    (self isGestureStart: anEvent)
        ifTrue: [^ self gestureStart: anEvent].

    self mouseDown: anEvent.
```

We assume existing domain knowledge in this debugging scenario, although Vivide provides tools for examining the software artifacts involved in a tool's views. That is, there is a process object (`Process`) with a suspended context object (`MethodContext`) and access to the entire stack with method activations. Further, source code is accessible via method objects (`CompiledMethod`) and method arguments reveal other domain concepts such as event artifacts (`MorphicEvent`, `MorphicEventDispatcher`). The programmer keeps the original tool colors to better relate to her initial problem as depicted in figure 3.1.

Morphic Halo for Tool Building

Each view in the script-based debugger has a custom halo associated, which provides access to the underlying script and artifacts. While nested views require multiple mouse clicks to expand compositions, the programmer manages to quickly open the halo for the debugger's stack list view. In particular, any view halo provides buttons to change view geometry, choose another script, edit the current script, examine the incoming artifacts on the left, or examine the outgoing artifacts on the right.



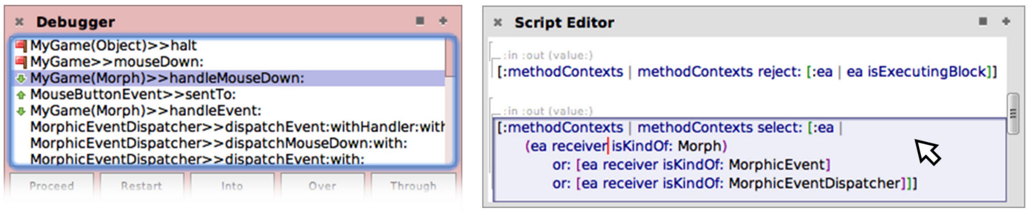
```
MyGame(Object)>>halt
MyGame>>mouseDown:
MyGame(Morph)>>handleMouseDown:
MouseButtonEvent>>sentTo:
MyGame(Morph)>>handleEvent:
MorphicEventDispatcher>>dispatchEvent:withHandler:with
[] in MorphicEventDispatcher>>dispatchMouseDown:with:
BlockClosure>>ensure:
```

```
this mouse down could be the start of a gesture, or the
end of a gesture focus"
```

One could see this open-halo action as a role transition from tool user to tool builder. By accessing such a *meta interface*, the programmer takes a first step to fulfill her tool building intentions. In this regard, the reason for copying a tool could also have been to retain useful artifact representations on screen. Note that the debugger keeps "running" all the time, even though its UI appears rather static by design and waits for user interaction. The programmer clicks on the "edit script" button, located on the view halo's bottom edge.

Smalltalk as Query Language

A script editor appears, and it reveals the current script as a series of data transformation steps. That is, a tool's query language in VIVIDE consists of regular Smalltalk blocks, which transform artifacts for task-specific fetching or examining. When the programmer interacts with the script editor, there are blue frames around all views in the environment that use the same script. Having this, the programmer can better anticipate the effects of a change. Now, she adds two filter rules to trim the stack list view. First, method activations that represent block executions, such as "`[] in Morph ...`", should be rejected. Second, only relevant domain code, as indicated by certain classes, should be selected.



```
MyGame(Object)>>halt
MyGame>>mouseDown:
MyGame(Morph)>>handleMouseDown:
MouseButtonEvent>>sentTo:
MyGame(Morph)>>handleEvent:
MorphicEventDispatcher>>dispatchEvent:withHandler:witł
MorphicEventDispatcher>>dispatchMouseDown:with:
MorphicEventDispatcher>>dispatchEvent:with:
```

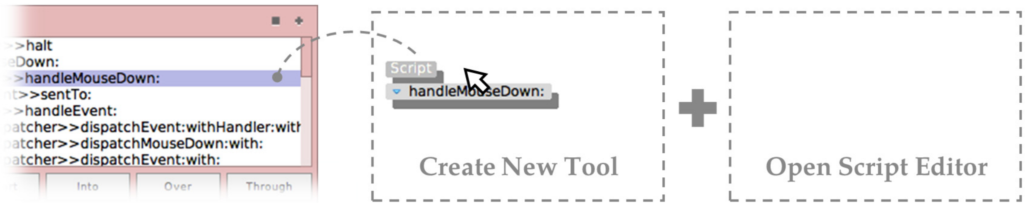
```
[:methodContexts | methodContexts reject: [:ea | ea isExecutingBlock]]
```

```
[:methodContexts | methodContexts select: [:ea |
    (ea receiver isKindOf: Morph)
        or: [ea receiver isKindOf: MorphicEvent]
        or: [ea receiver isKindOf: MorphicEventDispatcher]]]
```

After *finding* the script via the view halo, the programmer can now *change* the data-driven script code directly and *verify* the effect immediately. Each script change triggers an update in all associated views such as the debugger's stack list. Given the clear side-by-side presentation of tool and editor, the programmer can try out ideas, fix mistakes, and hence immerse in the tool building activity. Note that since these transformation steps consist of regular Smalltalk code, the programmer could also have employed the objects' `printString`-representation as depicted in figure 3.1. In general, the artifact's object representation inherently offers *messaging* as known from the underlying Squeak/Smalltalk environment. That is, scripts can navigate the entire object graph based on what the respective objects respond to the messages sent.

Tangible Artifacts Drive Tool Creation and Invocation

The programmer wants to create a new tool called *Live Code Browser*. For the necessary run-time information in such a browser, she plans to use artifacts from the debugger's stack list: the method activations. These artifacts exhibit a textual, yet tangible, representation that fosters conceptual distinction in the UI and hence direct manipulation. So, she points to a single item in that list view, drags it out of the debugger window, and drops it onto a free spot in the environment.

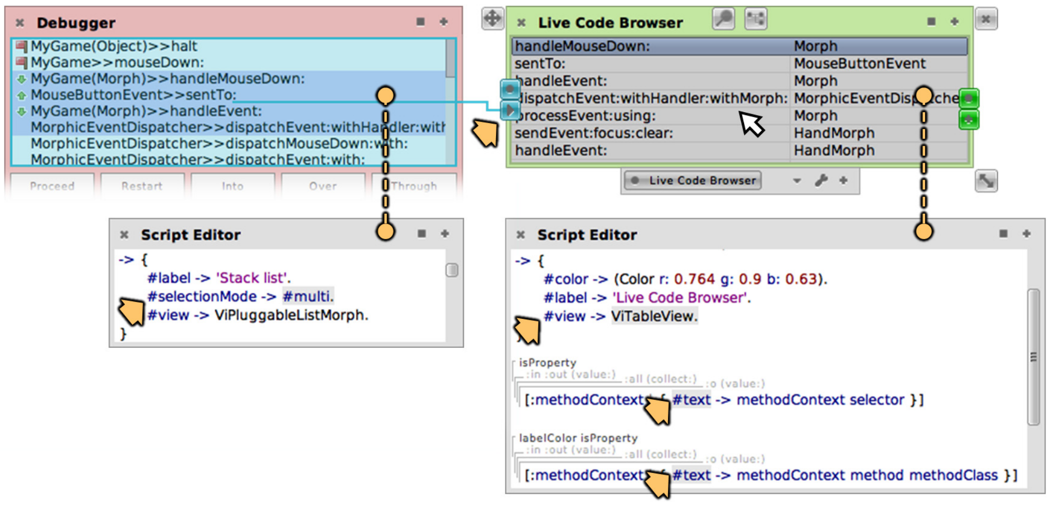


```
>>halt
eDown:
>handleMouseDown:
t>>sentTo:
>handleEvent:
atcher>>dispatchEvent:withHandler:witł
atcher>>dispatchMouseDown:with:
atcher>>dispatchEvent:with:
```

Usually, this action would open a script-based tool that supports this kind of artifact, which would be an object inspector in the most general case. In this situation, however, the programmer hits the modifier key `[S]` before dropping the artifact to indicate *scripting* and hence express her intention of creating a new tool. As expected, a new tool window and a script editor appear. She knows that, in addition to graphics, many views in Vivide employ *selection* and *drag-and-drop* interactions as generic means to support tangibility. While the visual representation of artifact structure remains configurable in scripts, she can usually rely on these recurrent, simple interactions.

Visual Mapping Within and Data Connection Between Views

The programmer wants to see a table view that should show selected artifact structure in its columns. For this, the script editor reveals *script properties* and *object properties* to specify for or extract from artifacts. In terms of script properties, the programmer can select and configure views. She uses these properties to select the table view (`#view`) and change the window color to green (`#color`). In the debugger's stack list, she enables multi selection so that she can examine multiple artifacts (`#selectionMode`). In terms of object properties, the programmer can complement the *visual mapping language*, which the view exhibits per artifact. She uses these properties to extract class name and message name from the method activations. This table view puts multiple text properties into separate columns. Like the query language, regular Smalltalk code and hence object messaging is supported in the mapping language.



```
MyGame(Object)>>halt
MyGame>>mouseDown:
MyGame(Morph)>>handleMouseDown:
MouseButtonEvent>>sentTo:
MyGame(Morph)>>handleEvent:
MorphicEventDispatcher>>dispatchEvent:withHandler:with
MorphicEventDispatcher>>dispatchMouseDown:with:
MorphicEventDispatcher>>dispatchEvent:with:
```

handleMouseDown:	Morph
sentTo:	MouseButtonEvent
handleEvent:	Morph
dispatchEvent:withHandler:withMorph:	MorphicEventDispatche
processEvent:using:	Morph
sendEvent:focus:clear:	HandMorph
handleEvent:	HandMorph

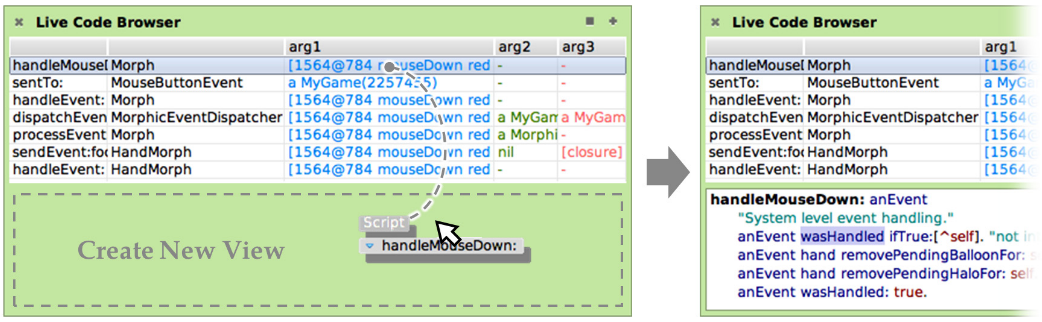
```
-> {
    #label -> 'Stack list'.
    #selectionMode -> #multi.
    #view -> ViPluggableListMorph.
}
```

```
-> {
    #color -> (Color r: 0.764 g: 0.9 b: 0.63).
    #label -> 'Live Code Browser'.
    #view -> ViTableView.

isProperty
  :in :out (value:)  :all (collect:)  :o (value:)
  [:methodContext #text -> methodContext selector }]

labelColor isProperty
  :in :out (value:)  :all (collect:)  :o (value:)
  [:methodContext #text -> methodContext method methodClass }]
```

As sketched in figure 3.3, views are associated with scripts, and data connection between views is explicit. The programmer establishes such a connection between the debugger's stack list and the browser's stack table by dragging the list halo's button on the right edge onto the table—that is, from output to input. Each time the selection in the debugger changes, the browser will now get the new set of artifacts to work with. In particular, there are *two* views that show structure differently based on the *same* object representation of the respective artifacts in the environment (compare with section 2.1). Luckily, the programmer sees no need for adding an extra "examine" button to the debugger (compare with figure 3.2) because of the direct manipulation facilities for tool invocation as described above. Now, she extracts information from relevant event artifacts to enable concise side-by-side examination in the table. She knows of object properties such as `#color`, `#icon`, and `#tooltip`. A textual representation of the first three method arguments would be sufficient for now.

Views on Artifacts Shape Presentation Language

Seeing the new tool displaying concrete information, the programmer realizes that her initial idea for the presentation language (or view layout) seems not appropriate anymore (see figure 3.2). A stack table on the left side and a code list on the right side would use up too much horizontal space in the environment. Thus, she drags a row from the table, which represents the code artifact, and drops it *below* the table in the same window. Again, she indicates a scripting activity by pressing the modifier key before dropping as described above. The tool window grows downwards, and a data connection is automatically established between the two views. The impression of working with actual software artifacts persists as another script editor appears, kind of "asking" for the transformations and extractions to perform. For inspiration, the programmer takes a look at script of the debugger's code view to learn about source code access and syntax highlighting.



		arg1	arg2	arg3
handleMouse(	Morph	[1564@784 mouseDown red	-	-
sentTo:	MouseButtonEvent	a MyGame(2257455)	-	-
handleEvent:	Morph	[1564@784 mouseDown red	-	-
dispatchEven	MorphicEventDispatcher	[1564@784 mouseDown red	a MyGam	a MyGam
processEvent	Morph	[1564@784 mouseDown red	a Morphi	-
sendEvent:fo(	HandMorph	[1564@784 mouseDown red	nil	[closure]
handleEvent:	HandMorph	[1564@784 mouseDown red	-	-

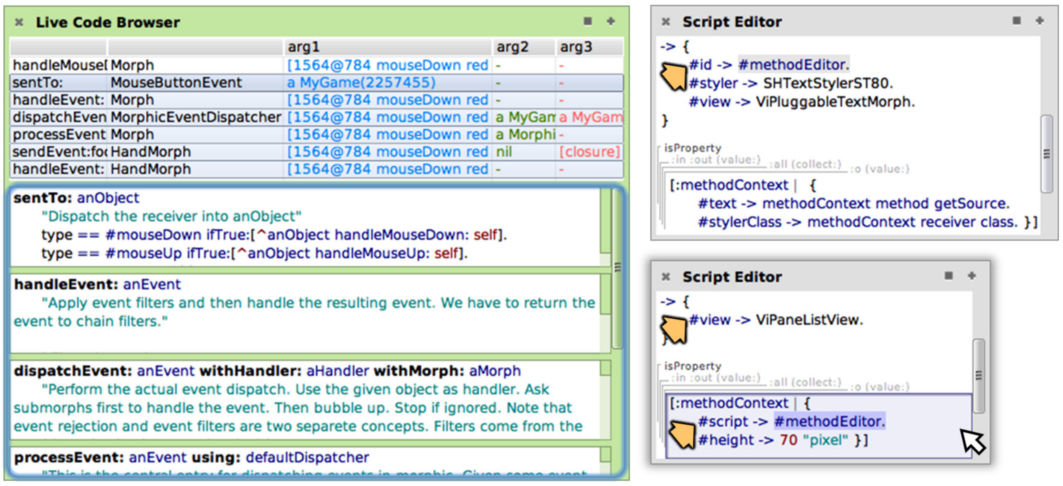
		arg1
handleMouse(	Morph	[1564
sentTo:	MouseButtonEvent	a MyGa
handleEvent:	Morph	[1564
dispatchEven	MorphicEventDispatcher	[1564
processEvent	Morph	[1564
sendEvent:fo(	HandMorph	[1564
handleEvent:	HandMorph	[1564

```
handleMouseDown: anEvent
    "System level event handling."
    anEvent wasHandled ifTrue:[^self]. "not i
    anEvent hand removePendingBalloonFor: s
    anEvent hand removePendingHaloFor: self
    anEvent wasHandled: true.
```

She could always re-arrange views using the halo and its move or resize buttons. It is also possible to cut and move views between tools (or windows). They would keep their current artifacts and scripts. The programmer knows that actual re-use of the debugger's scripts in the new browser does not work because each tool has a custom *script organization*, which represents separate namespace to look up scripts. Even though both scripts look similar, the programmer cannot confuse their application because of the blue frame around the affected views as explained before.

Advanced View Composition and Script Re-use

The programmer can now browse a single method at a time. Still, she wants to replace the single-method view with a multi-method view to better understand the control flow in Morphic's event dispatching and handling. Luckily unlike the debugger's stack list, the table view does already support the selection of multiple artifacts (or rows). VIVIDE provides a *composite view* that produces as much views as artifacts are present. The programmer defines a plausible identifier (`#id`) for the method script. She then creates a new script for the same area with the help of the view halo. The `ViPaneListView` serves as a composite view (`#view`), which provides a scrollable container and expects a script identifier to look for in the tool's script organization. The result looks promising.



		arg1	arg2	arg3
handleMouseI	Morph	[1564@784 mouseDown red	-	-
sentTo:	MouseButtonEvent	a MyGame(2257455)	-	-
handleEvent:	Morph	[1564@784 mouseDown red	-	-
dispatchEven	MorphicEventDispatcher	[1564@784 mouseDown red	a MyGan	a MyGam
processEvent	Morph	[1564@784 mouseDown red	a Morphi	-
sendEvent:fo	HandMorph	[1564@784 mouseDown red	nil	[closure]
handleEvent:	HandMorph	[1564@784 mouseDown red	-	-

```
sentTo: anObject
    "Dispatch the receiver into anObject"
    type == #mouseDown ifTrue:[^anObject handleMouseDown: self].
    type == #mouseUp ifTrue:[^anObject handleMouseUp: self].

handleEvent: anEvent
    "Apply event filters and then handle the resulting event. We have to return the
event to chain filters."

dispatchEvent: anEvent withHandler: aHandler withMorph: aMorph
    "Perform the actual event dispatch. Use the given object as handler. Ask
submorphs first to handle the event. Then bubble up. Stop if ignored. Note that
event rejection and event filters are two separete concepts. Filters come from the

processEvent: anEvent using: defaultDispatcher
```

```
-> {
  #id -> #methodEditor.
  #styler -> SHTextStylerST80.
  #view -> ViPluggableTextMorph.
}

[:methodContext |  {
    #text -> methodContext method getSource.
    #stylerClass -> methodContext receiver class. }]
```

```
-> {
  #view -> ViPaneListView.
}

[:methodContext | {
    #script -> #methodEditor.
    #height -> 70 "pixel" }]
```

This list of methods is the second example for composite views in the Live Code Browser. The first one was created implicitly, when the programmer dropped the code artifact to create the code view. It is called *tiled composite view*, and it has two degrees of freedom instead of one, compared with the *list composite view*. Vivide provides other compositions such as *stacked view* and *overlapping view*, which the programmer does not need at the moment. She recalls that both *script label* (`#label`) and *script identifier* (`#id`) foster script re-use and thus tool re-use. The human-readable label supports interactive tool invocation by manual script selection. The organization-specific identifier helps with view composition within tools.

Synopsis With Vivide, the programmer, who has an idea for improving a particular program comprehension task (i.e. fetch, examine, retain artifacts), is able to modify the debugger and build a new Live Code Browser in a few steps. These steps are likely to take minutes not hours. Given that she knows certain graphics concepts (i.e. list views, table views, text views) and domain (data) concepts (i.e. code artifacts, event artifacts), she can focus on expressing the task-specific rules for querying, mapping, and presenting those artifacts in interactive script editors. Verification is easy because all tools exhibit changes directly without restart.

For the programmer being a tool builder, Vivide scripts hide much glue code, and their automatic evaluation shortens the feedback loop compared to traditional tool building frameworks. For the programmer being a tool user, views in Vivide foster the tangibility and integration of software artifacts through selection, drag-and-drop, and connections.

Summary

Efficient Smalltalk programmers try to understand the concepts and theory of programs by examining code artifacts, which are present in the programming environment. For them, external documentation or on-line support forums are of lower priority. Additionally, Smalltalk programmers are used to evaluate code on objects in any tool's text view to reason about the current system state. Considering program comprehension, they can use such code evaluation to express rules to better fetch, examine, and also retain relevant artifacts.

However, the self-supporting, interactive characteristics of Squeak/Morphic do not contribute to tool building because its traditional tool architectures favor modular code structure over modular run-time structure. First, the primary means to explore state, namely the Morphic halo and the Object Inspector, do only expose generic object structure, which has rarely a direct connection to the underlying source code. Second, there is no support for expressing a tool's query, mapping, and presentation language concisely and interactively. Third, code (and other tool-object) changes typically entail full tool restarts, which impedes verification of the intended effects.

Our idea for a better tool building framework aims at improved modularization and object-modification support for running tools. On the one hand, we want to encapsulate the rules to query and map artifacts into *scripts*, which would directly correspond to the interactive views they define. A script edit tool should help focus on these rules while embedding run-time information into abstract code snippets. On the other hand, we want to apply traditional support of graphical design tools to *live, ad-hoc view composition* for running tools. With the integration of existing artifacts and widgets, we think that such a tool framework (and run-time) can *hide much object-oriented glue* from the tool builder. We think that such an approach can save time and even *encourage tool users to become tool builders* outside a dedicated tool building task.

By example, we illustrated how a programmer builds a new code browser using the artifacts a debugger window provides. She invoked the Morphic halo to *find* scripts and data connections that form the tool's interactive presentation language. Then, she wrote concise Smalltalk code to define queries and visual mappings, which *change* the tool's interface. The whole time, she experiences immediate updates after each modification and directly *verifies* her efforts. Such an iterative, exploratory working practice is likely to take minutes, not hours.

[|]

In the next chapter, we explain our Vivide programming environment in detail. This includes a block-based scripting language based on Squeak/Smalltalk, a data-driven strategy for tool design, a new UI-design language based on Morphic, and a new data-driven working practice that puts it all together.

Part II

The Vivide Programming and Tool-building Environment

4 Vivide: A Data-driven Tool-building Environment

We complement two *engineering* proposals with two rather *philosophical* ones. On the one hand, we present *new languages* to build tools, which contributes to engineering. On the other hand, we present *new strategies* to perceive and use tools, which contributes to philosophy. Overall, our solution considers the characteristics of *live environments*, the values of the *programmers* working in there, and the goals *domain-specific tasks* entail. While we remain compatible with traditions, we open a new data-driven perspective on programming tools, which embodies direct manipulation, tangible graphics, and noun-verb interaction. At the end of the day, Vivide can be an environment that encourages a mindset shift toward ad-hoc tool building in exploratory programming tasks.

In this chapter, we unpack the four contributions of this work. First, we present a new *scripting language* to describe view models only little glue. Then, we propose a new *strategy for tool design*, which follows our Rules of Distinctiveness, Similarity, and Context. Knowing what tools to build, we then describe a new *UI-design language* that uses our scripting language to express informational queries, visual mappings, and whole graphical presentation of software artifacts. Eventually, this entails a new *working practice* so that programmers can relate to the Vivide means of choosing, configuring, and building tools.

4.1 The Block-based Scripting Language

We designed a new scripting language to improve accessibility and maintainability of the *object graph* in object-oriented systems. Note that we build on the assumption that, for this work, objects transport structure of domain-specific software artifacts as explained in section 2.1. Guided by the concise appearance of Smalltalk code, our main goal is to support the description of task-related *hierarchical trees*, which are usually concealed in that object graph. That is, we employ Smalltalk syntax and retain its semantics, but we add new meaning for interpreting specific *code artifacts* in the context of *domain artifacts*. The resulting *model tree*, which consists also of objects, can then be used as a view model for graphical widgets or as an intermediate structure for non-graphical data-processing pipelines.

In the following, we will explain our scripting language using a trichotomy commonly applied when describing the elements of programming languages [5,

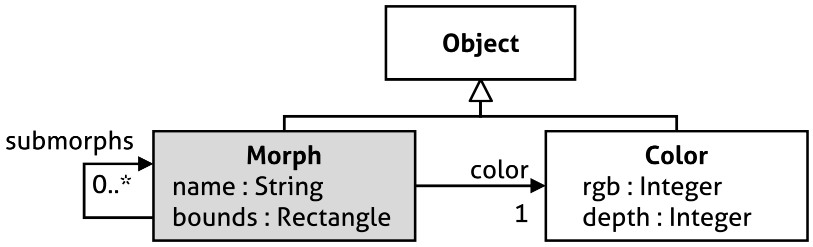


Figure 4.1: Morphs represent the graphical hierarchy in Squeak through submorphs. Each morph has a name, bounds, and a color for basic visual appearance. Note that we adapted this class diagram from UML [208, pp. 99–103] to fit the Smalltalk language.

pp. 4–31]: primitive expressions, means of combination, and means of abstraction. After that, we will elaborate on the specifics of script interpretation and the ad-hoc introduction of anonymous data structures. All script examples are valid Smalltalk code and use Morphic object relationships as shown in figure 4.1.

Primitives: Transform Objects and Extract Properties

All scripts represent the rules for (structurally) querying and (visually) mapping software artifacts. In object-oriented environments, we distinguish *two kinds of steps* in scripts: (1) transform objects and (2) extract properties from objects. That is, we postpone the graphical aspect for now, and we focus on selected artifact structure that is accessible via navigation in the environment's object graph. The first kind of step, called *transform step*, is semantically represented as a Squeak block closure, which is an objectified anonymous method (or function):

```
step := [:in :out | in do: [:morph | out addAll: morph submorphs ]].
```

In the programmer's mind, this example might manifest a rule like: "Transform all morphs into their submorphs." Such transform steps have an *input buffer* to read from and an *output buffer* to write into. Given some context, the programmer has to send appropriate messages to the incoming objects to query other objects and put them into the output. Since the original objects remain, we agree that the vocabulary is debatable: transform, access, explore, expand, navigate, query—to name a few. Consequently, script steps address the *structural level* of software artifacts and capture the *query language* in tools.

We make extensive use of Smalltalk's Collection interfaces [32], and the code complexity in a transform step is only limited by the Smalltalk environment. For example, the children in the Morphic hierarchy can be filtered by their color and extent as follows:

```
step := [:in :out | in do: [:morph | out addAll: (
  (morph submorphs
     reject:  [:each | each color = Color blue ])
     select:  [:each | each bounds extent > (50@50) ]) ]].
```

This example reads like: “Expand all morphs into their submorphs, but drop the ones that are blue or not larger than 50-times-50 pixels.” In each step, input and output buffers are collection objects. Typically, the resulting source code transforms objects without side effects, which resembles the *functional programming paradigm*.

The second kind of step in scripts is the *extract step*, whose intent is to emphasize relevant properties of objects (or artifact structure). In object-oriented environments, such properties are usually objects, too, but often more primitive such as numbers and strings. An object property extraction is actually a specialization of object transformation because output objects remain under the programmer’s control:

```
step := [:in :out | in do: [:morph | out add:
   { #object -> morph .
     "Extracted object (or artifact) properties."
     #name -> morph name .
     #color -> morph color  }]].
```

This example retains the incoming morphs as-is in `#object` and extracts two additional properties in `#name` and `#color`. Again, the code complexity is only limited by the Smalltalk language and environment. Compared with the transform step, the extract step’s output buffer holds not just a list of objects but a list of *model nodes*, which provide object-oriented access to both software artifacts and extracted properties. At his point, the extract step seems adequate to contain both the tool’s *query language* and *mapping language*. We think, however, that two kinds better reflect these two conceptual challenges programmers face when building tools as explained in section 2.5. We clarify the benefits from such a distinction in the language’s *means of combination* and *script-edit tool*.

Means of Combination: Define Tree Structures

Programmers can alternate transform steps and extract steps to specify an hierarchical tree with multiple levels as depicted in figure 4.2. During script interpretation with actual objects, a combination of *n* transform-extract pairs can produce model trees with *n* levels or less. That is, only if a transform step yields one or more objects, there will be another level in the tree. For each inner level, a step is interpreted as often as there are nodes (or objects) in the previous level. The following example shows how to describe a model tree that captures one or two levels of the graphical hierarchy using incoming morphs:

```
script := {
   "Level 1 – Transform and extract."
   [:in :out | out addAll: in]. "No change."
   [:in :out | in do: [:morph |
     out add: { #object -> morph. #name -> morph name }]].
   "Level 2 – Transform and extract."
   [:in :out | in do: [:morph | out addAll: morph submorphs ]].
   [:in :out | in do: [:morph |
     out add: { #object -> morph. #name -> morph name }]] } asScript .
```

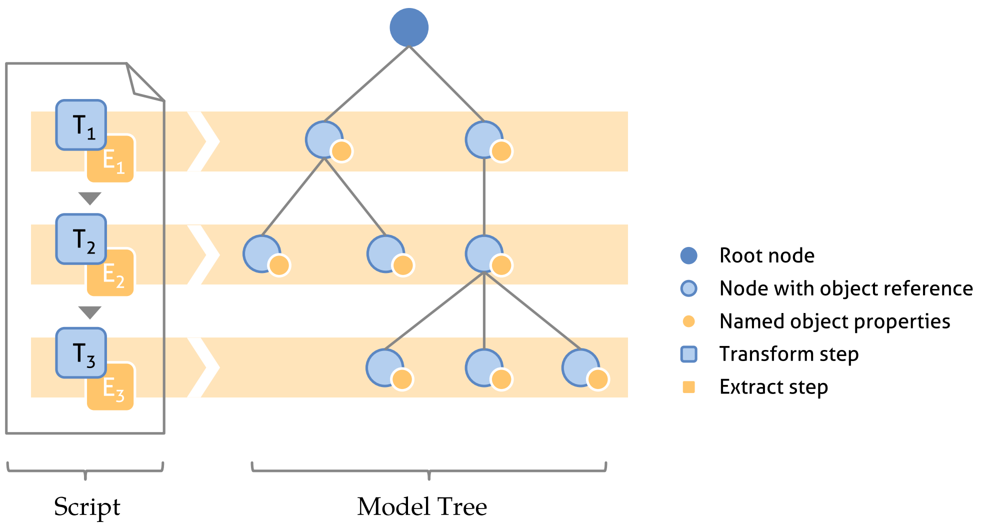


Figure 4.2: Scripts specify hierarchical tree structures level by level. Each extract step enriches model nodes with properties for a certain level. Nodes will only have child nodes if the particular transform step yields objects.

Basically, we wrap *Smalltalk blocks* in support objects so that we can store *additional information* for script interpretation. Here, a sequence of in-out blocks is put into an object array (i.e. `{...}`) and then converted into a script object via `#asScript`. All incoming morphs (or objects) will be represented as children of the root node on the first level. The nodes of morphs that have submorphs will have child nodes and thus a second level.

For the sake of code structuring and code readability, programmers can combine multiple transform steps or extract steps back-to-back. On the one hand, object transformation might benefit from clear separation of sorting, filtering, or graph navigation. On the other hand, property extraction might benefit from avoiding fill-in names for similar properties to be shown, for example, as `#text` in several table-view columns.

In general, the top-down sequence of steps in scripts dictates a primary reading direction and style. Such code-formatting style is not always straightforward[1] to achieve in plain Smalltalk and its collection interface:

```
((WebClient httpGet:
  'http://en.wikipedia.org/wiki/Unix') content lines
    gather: [:line | 'href="/wiki/[^"]*"' asRegex matchesIn: line])
    asSet asOrderedCollection "Remove duplicates."
    sorted "Sort strings lexically."
    collect: [:url |
      (url copyReplaceTokens: 'href="' with: 'http://en.wikipedia.org')
        allButLast "Remove trailing quote."].
```

[1]We formatted this code snippet to underline our argument of having a primary reading direction. This might not comply with best practices.

This example is a Smalltalk version of the one for Unix pipes-and-filters as shown in section 3.3. While it is possible to emphasize object transformations, such as `#sorted` and `#collect:`, it requires much discipline and care to add helpful line breaks or other whitespace. In our scripting language, in-out blocks propose a certain data-driven programming style, which inherently affects code readability.

The vertical reading direction becomes more clear if we hide the *repetitive syntax* for reading from the input buffer and writing into the output buffer. These are repeated message sends to the collection interface. An interactive script-edit tool could do such optional masking. Expressed in our scripting language *with masking*, the motivational example now looks like this:

```
script := {
   [:token | 'http://en.wikipedia.org/wiki/', token].
   [:url | WebClient httpGet: url].
   [:response | response content lines].
   [:line | 'href="/wiki/[^"]*"' asRegex matchesIn: line].
   [:tokens | tokens asSet asOrderedCollection]. "Remove duplicates."
   [:tokens | tokens sorted]. "Sort strings lexically."
   [:url | url copyReplaceTokens: 'href="' with: 'http://en.wikipedia.org'].
   [:url | url allButLast]. "Remove trailing quote."
} asScript openScriptWith: #('Unix').
```

The script semantics do not change considerably when concatenating steps of the same kind. For consecutive transform steps, there are just more input buffers and output buffers involved. For consecutive extract steps, there has to be a prefix or suffix to distinguish named properties in the model nodes like `name_1` and `name_2` for subsequent `#name` properties. To understand the amount of recurrent code that was omitted, the complete first step with `:token` looks like this:

```
[:in :out | out addAll: (in collect:
   [:token | ’http://en.wikipedia.org/wiki/’, token] )].
```

While our scripting language can concisely capture a tool's query language and mapping language, there is some *glue* remaining. That glue connects *domain rules* to *framework mechanics*: the collections representing input and output buffers.

Remark Naming is difficult. Especially for such alternating combinations, the terms we use for the two kinds of steps seem debatable. Having trees and levels in mind, the "extract" step could have been called "next-level" step. As we describe later in the script interpretation details, that step could have been also called "suspend" step because interpretation (and thus model generation) happen *on demand* when child nodes get accessed. From the programmer's perspective, however, we value the concept of data-driven transformation and extraction more than the concept of trees because of the focus on fetching, examining, and retaining software artifacts in a multi-view and multi-tool environment. Therefore, we favor the term "extract" over "level". Also, we think that *list models*, which are basically one-level trees, can be appropriate for many programming tasks.

Means of Abstraction: Identifiers, Organizations, and References

Named abstractions [5, pp. 7–9] enable programmers to manage their plentiful rules for querying and mapping artifacts on a conceptual level. For example, the script that "somehow exposes the graphical hierarchy for this morph" can become memorizable through the brief term "submorph tree". Such omission of details helps focus in complex tasks where programmers process many artifacts in various ways. To have such naming for scripts, we introduce *script identifiers* as distinguished state in script objects to look for in *script organizations*. Like message lookup in classes or class-name lookup in the environment, scripts become addressable by name in object-oriented code. Consequently besides the human aspect, we use this indirection to promote *dynamically scoped re-use* of scripts during interpretation.

Besides identifiers, programmers can store additional information into script objects through *script properties*. These properties have a Smalltalk representation similar to object properties in extract steps. Programmers can add more script properties as needed such as a human-readable description (`#label`), which we found beneficial for script browsing. We also denote extract steps via the `#isProperty` property to support the script interpreter. As an example, the following script extracts object properties from a list of morphs (without an extra transform step):

```
script := {
   [:in :out | in do: [:morph | out add:
      { #object -> morph.
         "Object properties."
         #name -> morph name.
         #color -> morph color }]]
   "Script (step) properties."
   -> { #id -> #morphNameColor . "Unique identifier."
         #label -> 'Morphs with Name and Color' . "Human−readable."
         #isProperty -> true "Indicate property extraction."}.
} asScript.
```

The format for *script properties* is the same one as for *object properties*. We see similarities to Squeak's method objects. In Squeak, methods are instances of the class `CompiledMethod`, which also support *additional method state*. Normally, programmers do not take advantage of that when writing object-oriented code. Still, there are *method pragmas*, which add such state to be accessed via Squeak's meta-object protocol in regular code.

Now we have the entire new *building block* our scripting language is made of: steps with properties. When constructing script objects from code, script properties are associated with the in-out block (`[...] -> {...}`). It becomes clear that each step in a script can represent a script on its own. We borrow the concept of *singly-linked lists*, which is an abstract data type [33, pp. 236–241]. Each step refers to either another step or nothing, which would be the last step then.

Script organizations separate scripts to embrace mistakes for the programmer's explorative attitude. There is a *global organization*, which keeps track of scripts that have value in recurrent tasks. Then, there are many *local (or temporary) organizations*, which capture scripting attempts that may not be worthwhile to keep around for longer. At any time, programmers can migrate scripts between organizations, for example, to preserve freshly constructed tools. Local organizations are created along with new scripts, and script identifiers are generated automatically by default. We want to free programmers from having to manage organizations or conceive abstractions at the beginning. Instead, given the particular program comprehension task, they should focus on explicating the rules for querying and mapping artifacts first and foremost. When creating scripts as mentioned above, the script's steps share an organization:

```
script := {
   [:in :out | in do: [:morph | out addAll: morph submorphs]]
        -> { #id -> #'61e787c9-fc8d-0241-bee4-463ec9915b5f'.
              #label -> 'Submorphs' }.
   [:in :out | in do: [:morph |
      out add: { #object -> morph. #name -> morph name }]]
        -> { #id -> #'01f88aa9-d4e8-004e-a001-1148a2c36e5f'.
              #isProperty -> true }.
} asScript.
```

Both `#'61e...'` and `#'01f...'` are in the same local organization. We think that programmers should not be bothered with such generated identifiers. Only manually chosen, memorizable identifiers should be displayed in tools that support script editing.

Script identifiers foster *fine-grained re-use* of specific script steps (and their following steps). Such *script references* are expressed as the script property `#next`. That property stores one or more script identifiers to be looked up in organizations. First, the local organization will be checked and, if not found, the global one. Recalling the previous examples, a script that transforms morphs into submorphs can then re-use the script `#morphNameColor` as follows:

```
script := {
   [:in :out | in do: [:morph | out addAll: morph submorphs]]
        -> { #label -> 'Submorphs'.
             #next -> #morphNameColor }.
} asScript.
```

Semantically, all referenced scripts (or script steps) are treated as *inlined* after the referencing step. Thus, programmers can add more steps *after* any step that has a `#next` property. Yet, referring to a step *before* the first one requires an *extra step* to be inserted, which may not do any transformation. The effects of such re-use include additional levels in the tree model or errors due to incompatible transformation rules. Consequently, tool support is required to manage complex scripts.

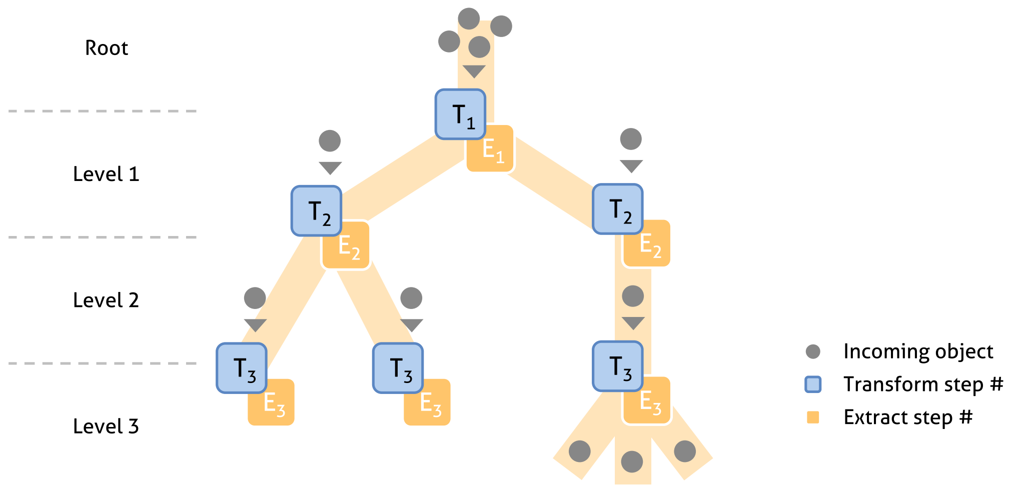


Figure 4.3: Only the first transform step is interpreted once with all incoming objects. Transform steps that describe inner levels are interpreted as often as there are objects produced by the previous level, each time with a single object. Extract steps receive any objects their preceding transform steps reveal. See figure 4.2 for comparison.

Script Interpretation for Model Construction

Script interpretation begins with the script's first step and a list of objects. Transform steps convert that list into another list of objects. Extract steps can convert again, but they primarily extract object properties and create nodes, which form the tree model. As depicted in figure 4.3, a model's root node holds many objects, and all other nodes hold a single object. When constructing the inner branches of a tree, script steps are interpreted several times, once for each object in the previous level. If steps refer to other script identifiers via the `#next` property, cycles can be defined, which then describe tree levels recursively. Depending on the object graph, such trees might have an infinite depth. Usually, script interpretation suspends after extract steps, that is, after a particular level in the tree was constructed. Views will trigger this node construction lazily by sending the message `#nodes` to the model (node). Interpretation will finish if transform steps yield no more objects.

Programmers can choose the kind of collections that will be used as input buffer or output buffer. By default, *index-based* access should be supported. As indicated in the script examples before, we think that programmers will usually address many objects at once. Given a concrete task with concrete artifacts, however, it can be possible that programmers just want to continue exploring the *n-th object* of some indexed input. In Squeak, such a collection would be an instance of `OrderedCollection`. Yet, there are others with different characteristics, such as sorting and duplicate removal, that can render script code more concise. Take the following script, which will calculate the square of each incoming number if that number is unique:

```
script := {
  [:in :out | in do: [:num |  (out includes: num) ifFalse:  [out add: num] ]].
  [:in :out | in do: [:num | out add:  num * num  ]].
} asScript.
```

Here, there are two transform steps. The first step discards duplicate numbers. The second step computes the square for each remaining number. For example, the list `#(1 2 2 3 3 3)` will be transformed to `#(1 4 9)`. With a different kind of collection, the first filter step can be made obsolete. There are two script properties to define the kind of input buffer (`#in`) and output buffer (`#out`):

```
script := {
  [:in :out | in do: [:num | out add:  num * num  ]]
    -> {  #in -> Set  "Choose input buffer with set semantics." }.
} asScript.
```

When using a *set* to arrange objects in the input buffer, the programmer can be sure that there are no duplicate numbers provided. Compromisingly, it is not possible to access objects by index without converting[2] it into a fitting collection first. Note that for views (or other clients) model nodes are always collections that support index-based access.

Tuples as Ad-hoc Data Structures

We assume that the object-oriented environment already provides objects that transport relevant structure to understand concepts of software artifacts. So, programmers can directly express the rules to query and map task-related information in scripts. It will just be *object transformation and property extraction*, which is backed by (Squeak's) object-oriented concepts *messaging* and *classification*. Since class objects are accessible in any piece of code, programmers can create, configure, and use new instances in a script's transform steps:

```
script := {
  [:in :out | in do: [:name | out add: ( Color fromString: name )]].
} asScript openScriptWith: #('banana' 'orange' '#FF0033').
```

However, there can be situations where existing classifications cannot reflect the programmer's mental model. In such case, it would be necessary to create new classes to bundle objects in a way that cannot be expressed with plain object-graph navigation. For example, a morph has a color, but colors are associated with many morphs and other objects. Even with Squeak's meta-programming facilities, it would not be possible to navigate from the color back to the original morph. That is when transforming objects in scripts, programmers might discard relationships that actually retain in mind. We think that programmers should refrain from defining

[2]In Smalltalk, collections can be converted between kinds by following a simple naming convention [13, pp. 28–30]: `#asClass`. For example, a set becomes ordered again via the message `#asOrderedCollection`.

new classes that might not be useful for other programming tasks. Instead, we conceptualize Smalltalk's *object arrays* to serve as *anonymous classifications*, which can be used ad-hoc in scripts: *tuples*.

As a compromise for missing named classifications, tuples combine objects to pass across steps in scripts and hence levels in the tree model. Note that we do not change the semantics of script interpretation, but we exploit the fact that any object can be stored in input buffers and output buffers—even collections. So, we introduce the use of object arrays as anonymous classifications (or data structures). In object-oriented code, tuples and arrays are indistinguishable:

```
tuple := { morph. morph color }.
```

Here, a morph and the morph's color are put into an array, which represents a tuple (or pair). Nevertheless, we require a convenient way to create tuples from lists of lists of objects because transform steps are especially useful for navigating one-to-many relationships in the object graph. So, we defined the "Cartesian product" on collections, which combines all elements of all collections in a collection:

```
tuples := #( (1 2) (a b c) ) asTuples. "Cartesian product"
tuples = #( (1 a) (1 b) (1 c) (2 a) (2 b) (2 c) ). "true"
```

While passing tuples between consecutive transform steps is straightforward, extract steps require additional semantics. Usually, one object in the tuple is of *primary importance* to the programmer. For example, if a step navigates to a morph's submorphs, the original morph (also known as *owner*) might be useful but secondary. Consequently, programmers can still denote such priority through the object property `#object`, while keeping the whole tuple in `#objects`. When constructing the next level, the responsible script step will get the tuple and not the priority object. The following script puts our tuple concept into an exemplary context:

```
script := {
   "Level 1"
   [:in :out | in do: [:morph |
      out addAll: {morph submorphs. morph} asTuples ]].
   [:in :out | in do: [:tuple | [ :morph :owner |
      out add: { #object -> morph. "Priority object."
                 #objects -> tuple . "Next-level input."
                 #color -> morph color }
      ] valueWithArguments: tuple ]] -> { #isProperty -> true }.
   "Level 2"
   [:in :out | in do: [ :tuple | "..." ]].
} asScript.
```

Here, we use part of Squeak's block-closure protocol to improve code readability: `#valueWithArguments:`. Programmers can use `tuple first` or `tuple second` if the tuple's collection supports index-based access, which arrays do. Such descriptive block arguments, additionally, avoid the need to add temporary variables for readability. Note that the use of tuples in combination with the `#objects` property does still conform with figure 4.3. Except for the model's root node, each inner node holds a single (priority) object reference. Inner transform steps are provided with tuples, but each of those tuples represents a single object to be unpacked to multiple objects only if needed.

Synopsis We designed a scripting language to improve accessibility and maintainability of the object graph in object-oriented environments. Our language uses Smalltalk semantics, which includes *messaging* in particular. Programmers compose *block closures*, each having an *input buffer* and an *output buffer*, to explicate the rules to query and map software artifacts represented as objects. They rely of Smalltalk's collection protocol to iterate over objects to sort them, filter them, or transform them into a set of relevant objects. We distinguish a script's *transform steps* and *extract steps*, which are managed in *script organizations*. When interpreting such scripts with concrete objects, tree models are generated, where each node in each level holds a single object reference and a list of extracted *object properties*. If the environment's existing classifications are insufficient, programmers can employ *tuples* as ad-hoc data structures, which form an extension to Smalltalk's object arrays and thus the collection protocol.

All software artifacts that this scripting language entails become part of the object-oriented environment. We think that additional tool support is required to hide recurrent syntax (or remaining glue) and foster readability of scripts in some cases. That is, interactive views on script artifacts have to complement this programming experience to support script-based tool building.

4.2 A Data-driven Strategy for Tool Design

We propose a different take on how to design interactive, graphical tools. Our goal is to support programmers to better distinguish the tool's "building blocks" during use. Then, tool-building programmers can be supported to find, change, and verify tools more easily. This idea aligns with general thoughts about a *modular tool design* to remain flexible as tool user and tool builder. Our strategy is twofold: (1) begin with tools for *single objects* and (2) use and combine such tools in (task) *contexts*. We formulate three rules, three guidelines, and three practices to support programmers in the tool-design phase.

First, we motivate and explain the *three rules* for our strategy. Then, we explain how to reduce redundancy and improve coherence of information in a tool's interface to represent *single, domain-specific objects*. We then describe the emergence and use of *composition context*, which can be anticipated as a by-product of tool (window) configuration.

Three Rules

Every object (or artifact) in the environment should be represented by an interactive morph (or tool) that supports highly recurrent tasks in a domain-specific way for a wide user group. When programmers approach unfamiliar source code, they begin with selected focus points [177]. Such "points" are single objects that form groups (and groups of groups) only later in the process. Consequently, there have to be compact tools that form *tangible* representations on screen so that programmers can focus in the *overwhelming* informational space. Recalling the tasks from section 2.2, programmers value tools that support to *examine* principal structure, *retain* relevant objects, and *fetch* more related objects. We formulate *three rules* to support this workflow based on an existing rule set for multi-view systems [10], which we optimize for object-oriented environments. By following these rules, tools are likely to favor *artifact structure over tool behavior*, which underlines our notion of "data-driven":

Rule of Distinctiveness There have to be tools that focus on *single objects* by showing domain-specific characteristics to support highly recurrent tasks for a wide user group. We call such tools *single-object tools*. For tool builders, such tools are the unit of composition. Highly recurrent tasks include exploration and modification of structure.

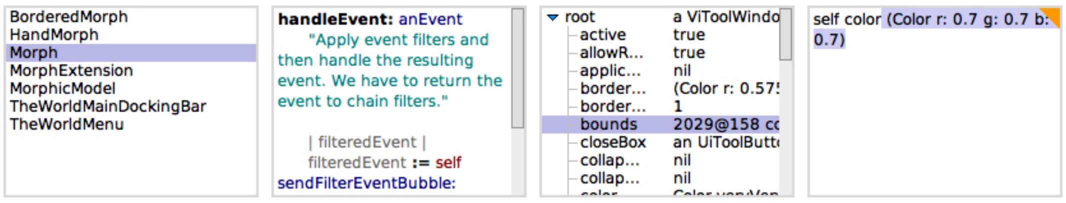
Rule of Similarity For each single-object tool, tool builders should explore the possibility of re-using the tool's interface for (1) multiple objects and (2) similar objects. For example, if a tool works for a single e-mail, it should work for multiple e-mails and other artifacts that exhibit properties like e-mails. Tools should embrace resemblance where its users do, too.

Rule of Context Programming tools should make use of the context they operate in. In graphical environments, this addresses a visual context such as windows of single-object tools arranged on screen. Tool builders should embrace and use these context objects to provide additional features.

Comparing with the existing rule set for multi-view systems [10], our rules share goals with the Rule of Diversity, Rule of Complementary, Rule of Decomposition, Rule of Self-Evidence, and Rule of Consistency. However, we do not distinguish "when" and "how" but focus on the construction of *complexity from simplicity*. In the following, we present *guidelines and practices*, which build upon the Rule of Parsimony, Rule of Space/Time Resource Optimization, and Rule of Attention Management from the same catalog [10].

Toward a Modular, Data-driven Presentation Language

There have been many systems that provide distinctive graphical representations of software artifacts in a tooling context. First of all, there are the tangible, iconic



```
BorderedMorph
HandMorph
Morph
MorphExtension
MorphicModel
TheWorldMainDockingBar
TheWorldMenu
```

```
handleEvent: anEvent
	"Apply event filters and
then handle the resulting
event. We have to return the
event to chain filters."

	| filteredEvent |
	filteredEvent := self
sendFilterEventBubble:
```

```
▼ root        a ViToolWindo
   active     true
   allowR...  true
   applic...  nil
   border...  (Color r: 0.57!
   border...  1
   bounds     2029@158 cc
   closeBox   an UiToolButt
   collap...  nil
   collap...  nil
```

```
self color (Color r: 0.7 g: 0.7 b:
0.7)
```

Figure 4.4: Four interactive views on single objects (from left to right): show the class category `Morphic-Kernel`, edit the method `handleEvent:` of `Morph`, browse an instance of `Morph`, interact with an instance of `Morph` in a workspace (or command line).

representations of artifacts embedded in list-based views, which were coined in the Xerox Star interface [80]. Even though not considered as separate tools, they do offer interaction in terms of double-click actions or pop-up menus. Then, there are other research projects that introduced rather tool-like views as supplements to achieve different goals in program creation and comprehension: the *components* in Fabrik [79] for data-driven application design, the *tiles* in Etoys [7] for child-friendly simulation design, the *bubbles* in Code Bubbles [21] for economic information layout, and the *shapes* in Gaucho [139] for direct object manipulation. We combine and generalize these examples and suggest the following guidelines for designing tool interfaces for domain-specific artifacts:

- Focus on relevant artifact structure and minimize information that would entail redundancy for similar artifacts — **less redundancy**
  For example, elaborate details about a method's class are not relevant if a tool focuses on the method's source code.
- Clearly distinguish (interactive) interface elements that represent artifact structure from elements that trigger side-effects — **more coherence**
  For example, a menu or tool bar should not mix actions that trigger side-effects on artifacts with invocations of tools for continued exploration.
- Provide hooks for tool integration primarily based on software artifacts and their relationships to other artifacts — **data-driven integration**
  For example, a tool should not offer to "browse versions" of a method but instead directly list all versions to choose from.

There are many ways to follow the **Rule of Distinctiveness** so that the environment provides one or more single-object tools that are tailored to one kind of software artifact. There can be a single view that provides a trade-off between interaction and display as depicted in figure 4.4. List views can afford selection and sometimes drag-and-drop. Text views typically also afford modification. We think that the level of accepted redundancy is debatable and hard to predict for arbitrary domain artifacts. One could measure such noise by opening many tools of the same kind but for different artifacts side by side and count the pixels that present the same information. For any kind of tool decoration, repeated patterns on screen can become

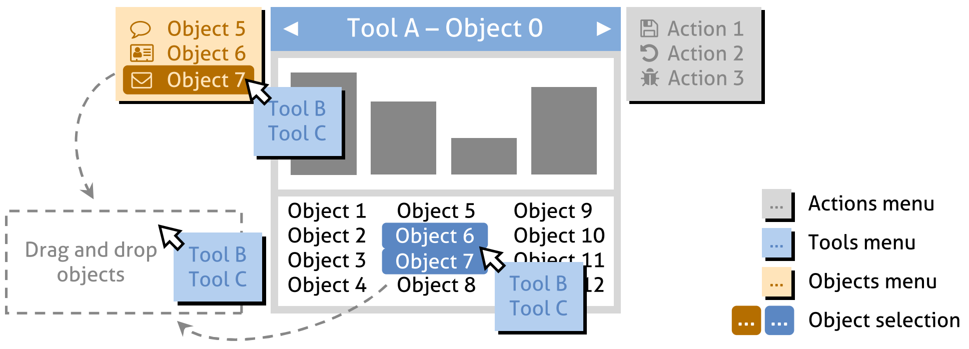


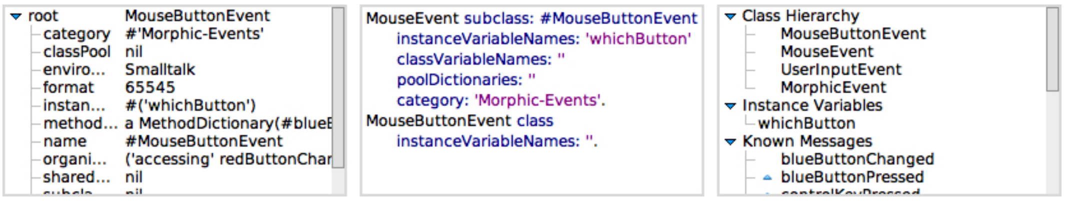
Figure 4.5: Single-object tools can provide any number of views to show distinct object structure. View-agnostic menus should separate related objects (left) and side-effect actions (right). If views expose objects, too, they should support selection. For selected objects, there should be a menu that offers available single-object tools to continue exploration. Drag-and-drop interaction can foster tangibility.

either annoying or easy to ignore. Tool builders have to experiment and test different designs.

Instead, we propose *three practices* for building a single-object tool to improve coherence and provide data-driven tool integration as depicted in figure 4.5:

- Any view's menus to continue fetching and examining relevant artifacts should *directly* show tangible, iconic representations of these artifacts, instead of explicit tool invocation. — **view-agnostic menus**
- Since artifact structure is usually transported as relationships to other objects in the object-oriented environment, any distinct object in the tool's views has to be selectable. — **object selection**
- Any selection of objects in the tool's views has to provide a menu to invoke suitable single-object tools to continue exploration. Drag-and-drop interactions into the environment can be convenient, too. — **selection menus**

Tool builders can design many single-object tools for the same object as depicted in figure 4.6 to highlight different aspects of complex objects. Such *complementary tools* can support a wide range of tasks. Yet, tool builders should follow the **Rule of Similarity** to improve usability. A common challenge in direct manipulation interfaces is the interaction with multiple objects at once [176, pp. 212–214]. If tools represent single objects only, programmers always have to interact with n tools to handle n objects. We propose to design interfaces around views that scale up: lists, tables, trees, text fields. The field of information visualization proposes similar ways [174] to display large amounts of data in a compact, interactive fashion on screen. Note that we *insist* on having single-object tools in object-oriented environments, and we *acknowledge* the need for tools that scale up only in the course of *complex* program comprehension tasks.



```
root          MouseButtonEvent
  category    #'Morphic-Events'
  classPool   nil
  enviro...   Smalltalk
  format      65545
  instan...   #('whichButton')
  method...   a MethodDictionary(#blueE
  name        #MouseButtonEvent
  organi...   ('accessing' redButtonChar
  shared...   nil
```

```
MouseEvent subclass: #MouseButtonEvent
    instanceVariableNames: 'whichButton'
    classVariableNames: ''
    poolDictionaries: ''
    category: 'Morphic-Events'.
MouseButtonEvent class
    instanceVariableNames: ''.
```

```
Class Hierarchy
    MouseButtonEvent
    MouseEvent
    UserInputEvent
    MorphicEvent
Instance Variables
  whichButton
Known Messages
    blueButtonChanged
    blueButtonPressed
```

Figure 4.6: Three different views on the class `MouseButtonEvent` (from left to right): a generic object inspector, a text-based class definition editor, and a custom inspector that shows class-specific structure such as superclasses and messages.

Staying with the Rule of Similarity, there is another aspect that fosters tool re-use: shared concepts. Many tools may look domain-specific in use, even though they process common structural information shared by many objects. For example, Squeak's object inspector shows an object's instance variables in a common textual format, which works for all kinds of objects in the environment. To raise the chance of re-using a tool for similar objects, tool builders should employ shared concepts. In traditional systems, there is a mere textual format expected from any kind of object such as `#printString` in Smalltalk and `toString()` in Java. We propose to broaden the list of such shared representations:

Label Comparable with `#printString`, provide a very short but distinct text representation of the object.

Icon Provide an iconic representation of the object. If the object has inherent graphics, such as for pictures and Squeak's morphs, a downscaled version can be sufficient.

Summary Provide a more elaborate summary of the object properties. For example, views can provide tooltips to show that information.

Color Provide a color as part of the environment's visual language for the object. Smalltalk systems have been using colors to distinguish kinds of tools. Code browsers are green, debuggers are red. However, colors for objects rather than tools are more likely to foster a data-driven perspective.

Origin Provide an object that serves as origin, or primary relationship, for that object. Tools can employ this to improve orientation in a crowded environment. For methods, the origin can be the respective class.

We suppose that this list is not complete. Tool builders should think about the overall presentation language of objects in the environment. While we focus on visuals in this work, other human senses could be addressed, too. For example, single-object tools could play a *distinct sound* if programmers hover the mouse cursor over. If an e-mail tool makes a chirping noise on the arrival of new mails, users can learn to automatically think of those specific artifacts after a while. Tool builders can exploit such effects when designing new tools.

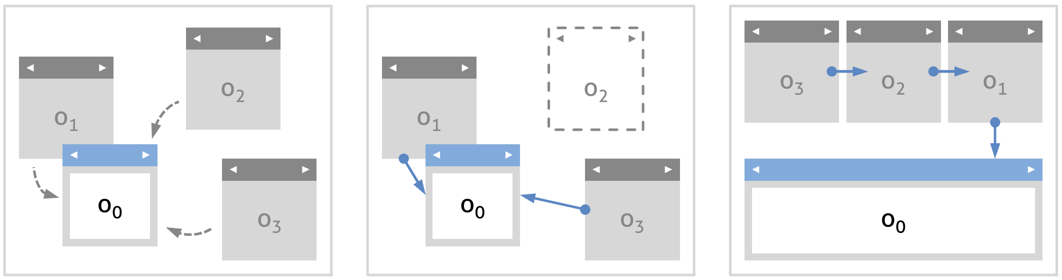


Figure 4.7: Tool windows form a composition context, and each window represents one or more domain objects. If tool builders want to exploit that context, the user's current tool should either use all objects implicitly (left) or a subset that is explicitly configurable (middle). The visual layout can support such configuration (right).

Emergence and Use of Composition Context

We propose the use of tool-composition context as additional dimension for a modular, data-driven design of tool features. That is, tool builders should anticipate a level of *user-controlled information integration*. There have been many research projects exploring different strategies for layouting tool windows with visual adornments to support the user's orientation. One popular strategy seems to be the arbitrary expansion of space along the horizontal axis with varying degrees of guided window positioning [21, 196, 73, 30]. Alternatives to overcome physical limitations include virtual [139][192] and zoomable [42][137, p. 46] screens. We agree that visual clustering of selected artifacts can help distinguish and document bugs, features, or any concern[3] in the system. Yet, if one tool can show *more* meaningful information with the help of another tool's object nearby, it should do so.

We think that tool builders can follow the **Rule of Context** by anticipating the programmers' habit of configuring the layout of graphical tool containers as explained in section 2.4. Usually, programmers move or resize windows to *examine* artifacts side by side and maybe *retain* them for later. (The possibility to *fetch* more artifacts is typically unrelated to this habit.) Consequently, tool builders should make use of the view compositions that emerge through such user interaction as depicted in figure 4.7. We distinguish implicit and explicit composition context:

IMPLICIT CONTEXT Since all tools in the environment represent one or more domain objects as described above, the composition context is defined as a set of all these objects to be processed by any tool in that composition. While the whole environment can serve as context, nested containers can further support domain-specific or task-specific grouping.

[3]In fact, flexible tool containers could be used to capture domain-specific concerns represented by usually widespread artifacts [198, 160]. In a self-supporting environment such as Squeak/Smalltalk, configurations of tools that directly represent artifacts can form new kinds of artifact themselves. The StarBrowser [213] provides such explicit classifications.

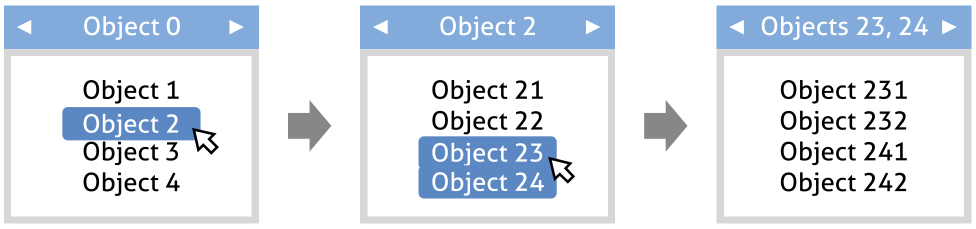


Figure 4.8: Explicit composition context can be defined in terms of the *explicit object selection* if tool views support it. Implicit context can still be defined in terms of single-object tools themselves as depicted in figure 4.7.

Explicit context If users can recognize connections between views visually, tools can specialize or clarify their informational needs. Tool builders can use such explicit data connections between views to design more complex tools that are data-driven and extensible.

That is, we propose to treat any *partial use of context* explicitly in the form of *discoverable* connectors between views, which is like Fabrik's connectors [79, 106]. Any particular window layout can imply provision or use of information. For example, Squeak's system browser has four views at the top, and navigation goes from left to right: class category, class, message category, message. Considering the data-driven perspective on tools in this work, one could see the browser's four views as consecutive data providers. So, a view's *object selection* (figure 4.5) can improve tangibility of its object representations, which complements the use of *single-object tools* as tangible entities. In figure 4.8, we illustrate a simple navigation task by *invoking tools* in terms of several consecutive selections. Consequently, tool builders have three options to define composition context:

- The object(s) a tool stands for as a single-object tool
- The object(s) a tool's view(s) show (in selectable form)
- The selected objects in a tool's view(s)

Tool builders should also support tool-controlled modification of that context, which can be compared to (semi-)automatic window management.

The distinction of *focus objects* and *context objects* helps design flexible, dynamic tools. Traditional programming tools are usually designed for a primary purpose with a *fixed composition context*. For example, Squeak's method objects occur as focus objects in code browsers and debuggers. In browsers, the context can be identified as a class category, a class, and a method category. In debuggers, the context can be identified as a receiver object, argument objects, and temporary objects. Tool builders have integrated those context objects so that tool users can examine them to better understand the focus object, which would be the method. However, the tool's nature would change if tool builders would re-design context, that is, change the kind of objects that provide additional information. We argue that code browser and debugger are quite different tools primarily because of their context objects. If programmers want to fetch the class of a method, they would have to leave the debugger and open a code browser. It is considered a different tool. In our tool-design strategy, tool boundaries become blurred because tool users are in control of the (single-object) tool's composition context, which now influences tool features. As an effect, the term "debugger" might become a role that users can attach to *any* set of tools.

Each composition context is a new, ad-hoc, custom relationship between the respective objects, which are represented by their views. Tools can use this relationship to bypass limitations in the underlying structure manifested as hardly accessible (or virtually non-existent) paths in the object graph. We think of concerns that crosscut the dominant decomposition of code artifacts and other artifacts that happen to be provided by some tool for actually different purposes. The graphical user interface can bring it all together, even if only for a single programming task. Tool users play an active part in establishing such relationships. Tool builders should make use of it.

Tool compositions can be treated as tools of their own accord, which can be composed with other tools and so on. To some extent, our suggested strategy compares with (i) primitives, (ii) means of combination, and (iii) means of abstraction—which is the trichotomy we used for explaining our scripting language in section 4.1. While a single view can be sufficient to represent one artifact, a combination of views might be necessary for another one. Tools that support complex tasks are usually filled with views of several kinds. The abstract *name* of one tool can be a viable handle to re-use and embed its functionality in another tool's interface. In any case, tool builders should provide a way to *access the underlying artifacts* as simple single-object tools so that tool users can retain information in a compact fashion on screen. For example, Squeak's object inspector can serve as a reliable way to retreat from inappropriate, maybe domain-specific, tools that fail to adequately support the programmer's current program comprehension task.

Synopsis We think that tools in interactive, self-supporting environments should be designed to directly represent the artifacts whose structure they show and offer to change. Thus, we formulate the Rule of Distinctiveness, the Rule of Similarity, and the Rule of Context to exploit and leverage the fact that everything is made of objects and that graphics are tangible in a shared interaction paradigm. That is, every object (or artifact) should have at least its own morph (or tool) that shows the main structure to foster tangibility and direct manipulation. Then, multiple such tools (or tool views) should form a *composition* that offers visual cues to its users. Plus, any composition should reveal new functionality based on each tool's *context objects*. Tool users manage such compositions like they manage application windows.

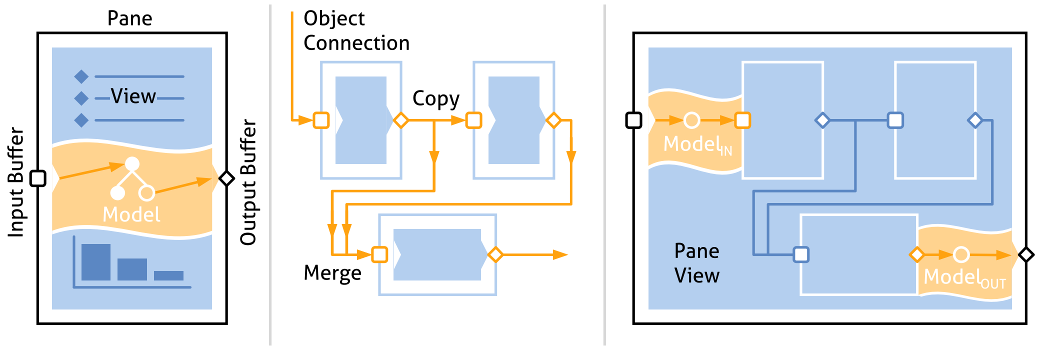


Figure 4.9: Vivide wraps each view in a pane (left), which governs objects and scripts for generating models. Panes can be combined via object connections (middle) to use a view's selected objects as input for other panes. For abstracting pane compositions into new views, there are pane views (right), which reify layout configuration.

4.3 The Morph-based UI-design Language

We designed a new kind of "brick" for constructing graphical tools in an interactive, data-driven way. We call that kind *panes*, in which we bundle the object-oriented paradigm, Smalltalk's direct code updates, and Morphic's tangibility. Like tools in general, panes exhibit views, artifacts, levels of composition, and visual decoration. We employ our scripts from section 4.1 to generate view models. Note that panes and views are morphs, which occupy screen space, offer visual composition, and react to user input. In sum, our morph-based UI-design language has one primitive concept, means of combination, and a strategy for abstracting nested compositions as illustrated in figure 4.9.

A *tool* can be seen as any pane in action, that is, all views have models to work with. To recap our scripting language, tool builders describe tool characteristics in terms of each script's transform steps and extract steps, which include *object properties* and *script properties*. We ensure the tangibility proposed in section 4.2 by distinguishing between *input artifacts* and *(selected) output artifacts* in each pane. For example, a code browser can be constructed with some panes showing list views, exchanging code artifacts, decorated by a window frame with the label "System Browser". In that browser, programmers can invoke the halo of any pane to access the (input) objects behind the respective view.

Remark We focus on tool *building* rather than tool *using*. Nevertheless, we think that some interactions are simple enough to be performed by any programmer, who may not notice the role swap between user and builder. Especially in Squeak/Smalltalk environments, there are all kinds of objects accessible, and "building" separates from "using" sometimes only through the perceived relevance of some object for the current task.

In this section, we first explain how a *single pane* uses a single script and some input objects to initialize a single view. Then, we elaborate on the use of *composition context* via object connections and the creation of *nested contexts* via pane views to construct complex tools. Finally, we employ Morphic halos as *pane decoration* to interactively change panes and their related objects. Overall, our UI-design language provides *consistent* updates for *running* tools after any change.

Configure: One Script, Some Objects, and One View

I have a collection of domain objects. How can I build a tool with a single view that shows some properties of these objects?

Basically, each pane needs a place on screen, some objects, and a script. A pane initiates script interpretation with the input objects to generate the view model. Then, it will create a default morph as view and provide the generated model so that the morph can show something. If the first script step has a `#view` property, the pane will consider that as the preferred kind of view (morph). In Morphic terms, each pane has a view as its primary submorph, and the view will have the same dimensions as the pane. This relationship allows panes to act as surrogates when replacing views so that a tool's layout remains stable. In the object-oriented environment, panes communicate with or otherwise manage the following objects:

P1 A *rectangle* holds spatial information for directing screen output and user input.
P2 A *script* holds the rules for generating the view model.
P3 A collection of *domain objects* represent the input for generating the view model, also called input buffer.
P4 The current *view model* can be re-used when switching views.
P5 The current *view morph* is the primary and often only submorph of a pane.
P6 A collection of *selected objects* is formed through user interaction with the view, also called output buffer.

Views have to provide a certain interface. In the tool building context, scripts represent a convenient way to express the query language and the mapping language. We argue that modifications and tweaks in these two languages are the essence of tool building. On the other hand, many views (or widgets or visualizations) represent re-usable entities that solve general challenges about (mostly) visual perception and aspects of cognition. It is usually a *one-time effort* to achieve compatibility with our notion of scripts and view models as explained in section 4.1. In Squeak, for example, a pane's view is just like any other morph with a specific interface to accept and provide support objects. That is, the view can be implemented in any framework as long as it meets the following requirements:

V1 Accept a *view model*, which contains relevant domain information in various roles to show such as text role or icon role.
V2 Accept a *script*, which contains meta information about the model structure such as the number of script steps.
V3 Provide an *object selection* so that panes can keep track of the objects the user is interested in.
V4 Be easily disposable, and accept *close requests*.
V5 Accept an *object selection* after view disposal/swap to help users regain focus.
V6 Provide a *rectangle* that represents the preferred dimensions on screen.

In general, there is data-driven communication between panes and views. Scripts help abstract this communication to be independent from domain-specific details in the transform/extract code. Recall that Squeak provides access to all objects in the environment, which is especially convenient for graphical items because of Morphic's halos. Consequently, programmers with access to a pane object can not only examine input and output objects but also control the current selection in the view. This control fosters direct manipulation of software artifacts, while remaining compatible with the programmer's explorative live programming style. We can use the pane's separation of responsibilities from views and scripts to *consistently update* tools after changes in view code or pane state, so that tool builders can verify the effects quickly:

E1 Generate a new view model, discard the old one, re-use the current view, and try to restore the object selection *if script code changed, a script notifier triggered, or the set of objects in the input buffer changed.*
E2 Create a new view instance, discard the old one, re-use the current model, and try to restore the object selection *if view code changed, script properties changed, or a script step was added/removed/reordered.*
E3 Store object selection in output buffer *if the view provides a new object selection.*

What is a programming tool? After this first part of our UI-design language, a tool is merely a script, some objects, and a morph that can serve as view. Tool builders use a pane to put it all together. We assume that they do not have to worry about how to bring objects into the environment and how to implement missing views. Script authoring is the major tool building activity.

Combine: Views in Context Exchange Artifacts

I have two tools side by side. How can I use the selection (of objects) from one tool for the other one as input?

Multiple panes form a context, where each pane represents a tool for one or more domain objects. We apply our Rule of Context from section 4.2 so that each pane can use *context objects* in an explicit or implicit way like shown in figure 4.7. In Morphic, a common owner morph could manifest such shared context, and the mediator pattern [62, pp. 273–282] would be applicable. However, programmers can treat any set of panes (or views) as a collective in the current task. We do not want to pose such restriction on the mental model. Even if two views appear side by side, transparent layout containers might obfuscate ownership easily. Thus, we propose the explication of context use (and provision) in terms of *directed object connections* as depicted in figure 4.9. In addition to a pane's basic relationships, especially the input buffer and output buffer, each pane does also know:

P7 An ordered collection of *incoming connections* represents the definition of context use. (cf. P3)
P8 A collection of *outgoing connections* represents the definition of context provision. (cf. P6)

We apply *tuples* as described in section 4.1 to combine multiple sources to fill the input buffer with a single collection of objects. For example, if one source pane provides the numbers `#(1 2)` and another source pane the symbols `#(a b)`, the target pane's script will work on the pairs `#((1 a)(1 b)(2 a)(2 b))`. That is, the script has to filter meaningful combinations first. The *order* of connections determines the structure of tuples to be filtered.

Object connections are an instance of the *observer pattern* [62, pp. 293–303], but each connection forms a one-to-one dependency. A one-to-many dependency can be represented as many object connections from one pane to many others. We see similarities to Cocoa Bindings,[4] where "a binding is an attribute of one object that may be bound to a property in another such that a change in either one is reflected in the other." Yet, we distinguish a pane's input and output, which makes our "bindings" uni-directional. Connections can pass *any* object that is part of a pane's configuration: domain (input) objects, selected (output) objects, script objects, model objects, view objects, and even the rectangle object. Anyway, we think that the objects for script input and the objects selected through view interaction are pivotal in our design language. To encode this purpose, each connection as a *provision mode* and a *use mode*, which are independent. For example, a connection can pass an object selection

[4]Cocoa Objective-C Bindings, `https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/CocoaBindings/Concepts/WhatAreBindings.html`, accessed 2017-10-28

but also use it as input for the next pane. Note that, unlike Fabrik's connectors [79], object connections do not transform objects but just transport them.

We see the semantics of *stepwise* object exchange via object connections as an implication of object-oriented environments whose interactive tools primarily support managing object state. In the course of exploratory programming, programmers continuously examine system state to understand application behavior. They want to retain selected artifacts, which can also represent control flow reified as state such as nodes in a call graph. Consequently, we think that tool builders benefit from a tool building framework that treats information not in "continuous flow" but rather as objects resting in several buffers. Objects "move" only for synchronization, which is triggered by user interaction, timers, or other well-defined events. This is conceptually different from Unix byte streams [153, pp. 266–267] or event streams in functional-reactive programming [136], where information remains in pipes unless being redirected into a persistent medium or the screen.

Having object connections, we extend our design language to ensure consistent tool updates with the following events:

E4 Remove a pane's incoming and outgoing connections *if that pane is removed from the context.*

E5 Remove a pane's incoming connections of a certain use mode *if that pane overrides that use mode.*

E6 Retain a pane's objects wrt. a certain use mode *if all incoming connections of that use mode get jointly removed.*

E7 Update a pane's objects wrt. a certain use mode *if the source panes' objects change, connections are added/removed, connection modes change—considering each connection's current use mode and provision mode.*

What is a programming tool? After this second part of our UI-design language, the term "tool" expands to multiple scripts, sets of objects, and views. Programmers can now fetch, examine, and retain software artifacts in an explorative, concurrent way. That is, they can follow multiple hunches at the same time. Tool builders establish connections between panes to integrate tools. We argue that a view's spatial dimension encodes its task affinity in a scenario with multiple tools.

Abstract: Pane Views Encapsulate Context

I have two sets of tools/views to examine two different concerns. How can I separate them and then use the results for a third tool as input?

We use *layout reification* for panes to capture tool context, which we materialize as special views: *pane views*. The idea of having objects dedicated to layout in support of direct manipulation has been successfully demonstrated through alignment morphs

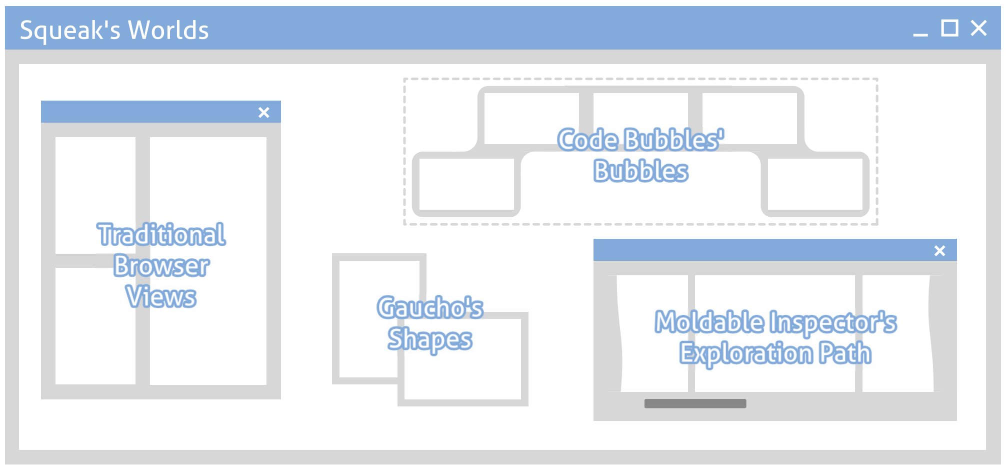

Squeak's Worlds
Traditional Browser Views
Code Bubbles' Bubbles
Gaucho's Shapes
Moldable Inspector's Exploration Path

Figure 4.10: Abstractions in the UI-design language foster task context organization. They can vary through layout strategies, browsing patterns, decorative overlays, or discoverable highlights. In addition to the tiled views in traditional browsers, we think of research projects such as Gaucho's shapes [139], Code Bubble's bubbles [21], Moldable Inspector's exploration path [30], and Squeak's project worlds [192]. They can all be embedded in each other or arranged side by side *ad-hoc* via panes, scripts, and pane views.

in Self's Morphic [110] and pampas' in Gaucho [139]. In our UI-design language, we follow the Rule of Context not only through object connections as described above. If pane context aligns with Morphic's graphical hierarchy, we can use our pane-view dualism as means of encapsulation and abstraction as shown in figure 4.9. That is, pane views can configure their contents the same way all views do: via scripts and models provided by a surrounding pane. Their contents, however, consist entirely of (decorated) panes. With such kind of abstraction, we support the construction of complex tools without having to leave our UI-design language.

With pane views, we broaden our tool building perspective from integrating existing widgets (or visualizations) to integrating existing *interaction patterns*. Panes are still our unit of tangibility for software artifacts. In combination, panes do not just pose the question of layout but also of interaction as depicted in figure 4.10. In the traditional *desktop pattern*, for example, users can arrange elements freely. Code Bubbles [21] makes the desktop virtually boundless, and it manages constraints that avoid occlusion and allow for ad-hoc grouping of adjacent elements. There are other patterns that vary through prescribed exploration paths, treatment of new elements, or decorative overlays in a way that is independent from artifacts and artifact-specific visuals. In our UI-design language, tool builders can embed many different layout/interaction styles in each other. We do not further discuss the utility of having, for example, a desktop in a column of an endless tape [30] in a bubble [21] on another desktop. Yet, tool builders can compensate for missing (compound) widgets by employing pane views to shape the intended user experience.

We propose a fairly generic and flexible pane view that applies scripts, script organizations, panes, and object connections to fulfill the requirements of regular views. Our goal is to stay within our UI-design language so that tool builders can further apply its elements recursively to construct more complex tools. There can be many kinds of pane views like there are many kinds of layout strategies and interaction patterns. Individual features aside, all custom pane views have to handle view models, scripts, and object selections as described before. Consequently, all pane views maintain the following *invariants*:

I1 In the graphical hierarchy, the *owner* of a pane view is a pane. (cf. P5)
I2 In the graphical hierarchy, all *children* of a pane view are panes. (cf. P5)
I3 All objects that a pane view distributes to its child-panes originate from the *view model*. (cf. P2)
I4 A pane view employs *object connections* to provide objects for selected panes in terms of input and selection. (cf. P3, P7)
I5 A pane view employs *object connections* to use objects from selected child-panes to form an object selection for its owner-pane. (cf. P6, P8)
I6 A pane view's configuration state, which includes child geometry, is always represented as *script properties*. (cf. P1)

Like all views in Vivide, pane views get populated through generated models. That is, scripts encode the configuration of child panes, including the distribution of (incoming) objects. The designer of such views can use object properties or script properties to store such configuration rules. Note that we assume that tool builders can choose from *existing* pane views like other visualizations. We consider the design and implementation of *new* pane views a different challenge.

Pane views define *object selection* in terms of object connections they set up between selected panes and themselves. Albeit pane views are not a special kind of panes, they support object connections and mimic the respective protocol. As illustrated in figure 4.11, tool builders can create complex tools with a clear semantics for object exchange between abstraction layers. We distinguish *horizontal object paths* and *vertical object paths*. The horizontal paths represent the means of combination of panes as described before. The vertical paths represent the means of abstraction. Such paths consist of transitions between panes and pane views, beginning from the root down to a leaf widget in the graphical hierarchy. Since panes are morphs, such structure can be embedded anywhere in the environment—maybe even replacing *all* tools in there. In sum, object connections simplify the object exchange for the one-to-many relationship between pane view and panes. In contrast, we think that the one-to-one relationship between a pane and its view cannot benefit from object connections because (vertical) object provision is already communicated through the view model.

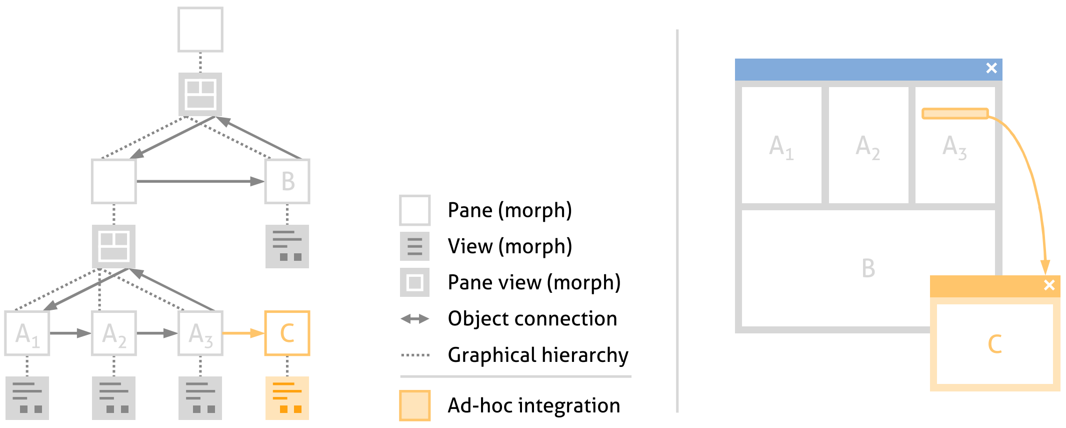


Figure 4.11: Pane views encapsulate sets of panes, configured via scripts, object connections, and a layout strategy. Other tools can access selected objects from any tool's pane using an extra connection (here: orange call-out), without impairing tool integrity.

The addition of pane views makes script organization more flexible. We can specialize global/local organizations from section 4.1 with the following *invariants*:

I7 A pane view has its own *script organization*.
I8 All *scripts* that a pane view's panes use are registered only once in some pane view's organization along the chain of graphical ownership or in the global organization.

Pane views look up script identifiers, which they find in object properties and script properties, to set up panes. Each pane view has its own organization so that tool builders can work with multiple sets of organized (script) context. That is, *local script organizations* align with the graphical hierarchy. The look-up starts in the innermost pane view and travels along the owner chain. Besides pane views configuring their panes, every script interpretation has to consider the graphical context. If a script step refers to another step via `#next`, the script interpreter has the following look-up priority: (1) organization of the current step, (2) all local organizations along the pane views,[5] and (3) the global organization. Note that the global one might not be necessary in environments that use pane views all the way up in the graphical hierarchy.

We extend the protocol between a view and its *owner pane*. Without the abstraction of pane views, we described regular views rather "passively" as object displays with interactive means of object selection. With pane views as context managers, we now have to account for user interaction that changes the amount of objects to work with. On the one hand, there are practices such as drag-and-drop, which empower users to compose context themselves. On the other hand, there are composition strategies such as multi-view windows, which can encapsulate common exploration paths. This extension yields two more requirements for views:

[5]Such look-up along the graphical hierarchy resembles the Boxer system [45].

V7 Provide *any set of objects* as object selection so that panes can keep track of the objects the user is interested in. (cf. V3)
V8 Provide a collection of *domain objects* to be used as input for the script to update the view model. (cf. P3, P7, I4)

That is, the pane's output buffer can contain objects that are not in the current view model, and the pane's input buffer can be updated from within the current view.

What is a programming tool? After this third part of our UI-design language, the term "tool" refers to a single pane with some objects and a script for a pane view. The script describes all configuration hints for the pane view as object properties and script properties. Tool builders should always consider the *outermost context* of a tool set to design that set as *a tool of its own*. Then, the terms "tool" and "environment" can become interchangeable.

Pane Decoration for Interactive Tool Building

There is a tool with many views. How can I see and change the artifacts that are exchanged in those views as well as the scripts that process those artifacts?

Tool building is an iterative activity that consists of many feedback loops to explore, understand, and assess the available means. So far, we have explained a tool's building blocks and how *changes* entail live updates so that programmers can directly *verify* the effects against their assumptions and expectations. Still, tool builders have to *find* and discover those building blocks in the first place—especially of complex tools or in unfamiliar situations. Since we have an object-oriented representation and inherently tangible graphics, it is only a question of *how* to make the tool's set-up and run-time objects (cf. section 2.5) accessible *in situ*.

We propose the use of *pane decoration* to let panes provide access to their configuration objects. In Morphic terms, panes can have *two* submorphs: view and decoration. The traditional decorator pattern [62, pp. 175–184] can be found in graphics design in the form of scrollable containers or application windows. Typically, decorations offer buttons and other visual adornments to both indicate and manipulate the component's current state. As depicted in figure 4.12, we follow this definition and further distinguish the *pluggability* of decorations. Morphic halos [108] demonstrate the feasibility of reserving an input gesture for entering a meta mode to *look behind* graphics into objects. Comparable with pop-up menus or keyboard modifiers, such *quasimodes* [152, pp. 55–59] make for discoverable and learnable means to consciously switch between tool using and tool building.

We designed a custom halo for panes as illustrated in figure 4.13. The halo buttons provide access to the pane geometry, the script (editor) and the object connections. Tool builders can click, drag, and drop buttons as handles to re-compose

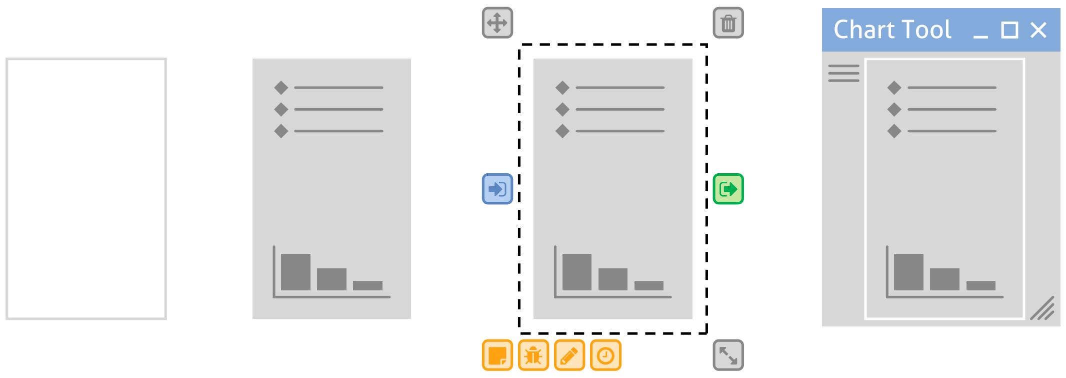


Figure 4.12: Panes are the building blocks of all tools in the environment. From left to right: panes can be empty, offer a view, offer a view and transient decoration, and offer a view and permanent decoration. The programmer stays in control of all options while the tools keep running.

panes, browse/debug scripts, or specify object exchange. In Morphic, programmers traverse nested compositions of morphs with repeated halo gestures. Hence, the current *pane's view* is accessible with its own halo. In contrast, we think that *pane views*, which manage a set of panes in context, do not need an own halo because their particular owner pane should offer interactive means of configuration. In result, such interaction through pane halos offers a tool building experience that is comparable with dedicated GUI-design tools. Yet, there is an important difference: our tools-under-design keep running. Consequently, new usage patterns can emerge such as ad-hoc integration of two tools as illustrated in figure 4.11. In that example, programmers can *fork object connections* to tap a tool's inner objects without impairing that tool's integrity.

Pane decorations can also guide the *creation of panes* besides their configuration. In direct-manipulation interfaces, users click, drag, and drop graphical items that represent their domain artifacts to accomplish tasks. Such items originate from various views and widgets that complement each other; a global search field can be the beginning of exploration. The dragged representations can range from plain text to graphical artworks, sometimes even animated. Given the program comprehension theme in this work, we explain *drop actions* with the user's desire to continue in-depth exploration of particular artifact structure. Then, *pane views* will typically accept those drop actions, and they can set up pane decorations to let the user choose from available scripts to spawn new tools. In our UI-design language, tool users drag only domain objects and windows, while tool builders drag also panes, scripts, and connections. If a tool builder drops one pane on another pane, for example, the pane view should employ a decoration to guide the creation of a composition and initial layout configuration for those two.

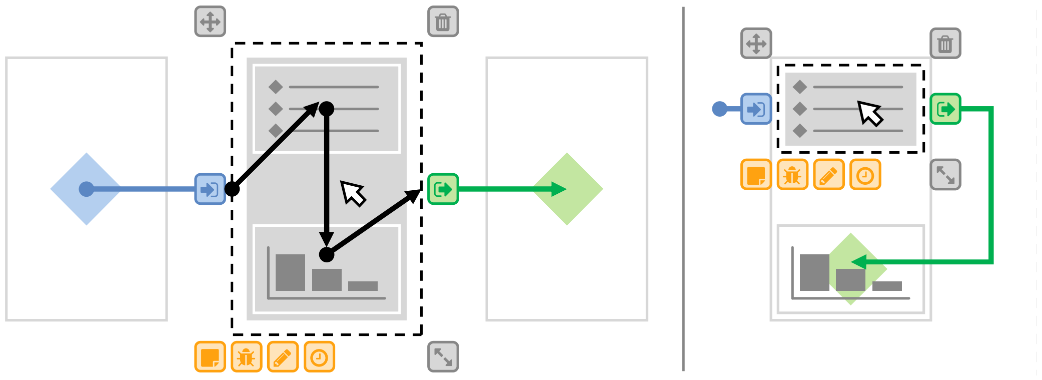


Figure 4.13: A pane's halo (left side) offers buttons and arrows. The buttons control geometry (gray), scripts (orange), input (blue), and output (green). The arrows indicate the pane's input/source pane(s) (blue), its output/target pane(s) (green), and the inner data connections (black) for pane views. The halo of an inner pane (right side) indicates input from or output to its owner pane (view) differently.

Synopsis We propose a UI-design language that is based on objects, messaging, and tangible graphics to form an interactive tool building environment. We employ our block-based scripting language to enable morphs to serve as flexible "building blocks" so that tool builders can follow our Rule of Distinctiveness, Rule of Similarity, and Rule of Context to support programming tasks.

In our design language, tools consist of one or more *panes*, which are configured with *scripts* to generate *view models* and govern interactive *views* respectively. Multiple panes can exchange objects via *object connections*, which allows for one pane's object selection to be used as another pane's script input. The context in which tool builders and tool users employ domain objects, scripts, panes, and connections can be encapsulated as *pane view* to serve as unit of re-use to compose complex tool interfaces.

We further support a direct, data-driven, interactive tool-building process with *pane decorations* such as common window adornments and Morphic's *halos*. Tool builders can directly access a pane's configuration state, modify it, and verify the effects because of inherently consistent tool updates. In our UI-design language, the terms "view", "tool", and "environment" can become interchangeable.

4.4 A Data-driven Working Practice

Vivide can encourage programmers to approach program comprehension tasks from different angles *if* they treat on-screen information differently. At a first glance, there are still tools to open and windows to move. Yet, programmers will omit expedient combinations of information if they miss the possibility of *ad-hoc connections* between arbitrary views.

We envision a working practice that is *artifacts-first* rather than *tools-first*. In the field of user interaction [152, pp. 59–62], this corresponds to noun-verb interaction rather than verb-noun interaction. In the field of direct manipulation [75], this entails shortening the semantic and articulatory distances to match the user's goals with the interface's actions and effects. It begins with a programmer being a mere beneficiary of the environment's amenities. It continues with that programmer actively defining the presentation and integration of all software artifacts on screen. When expressing appropriate intents, "open tool for" becomes "artifacts shown as". In Squeak/Smalltalk, for example, "open class browser" becomes "class shown as methods". Tools and tool boundaries can fade into the background. Software artifacts as tangible, visual representations remain.

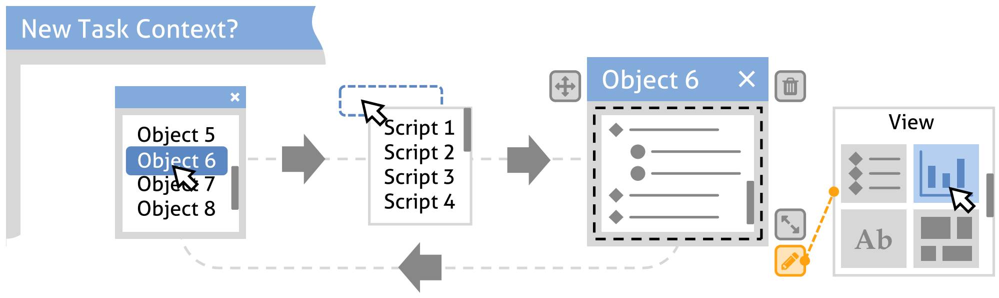
In this section, we propose how programmers should approach program comprehension tasks in the Vivide environment. We assume a transition of working practice from traditionally *tool-driven* to now *data-driven*. First, we relate aspects of our scripting language and UI-design language to the way programmers *choose*, *configure*, and *build* tools. Then, we describe possible mistakes programmers can make and how Vivide provides a "safety net" for continued exploration. Finally, we explain how programmers should *fetch*, *examine*, and *retain* artifacts for a single task. Vivide changes the entire *conversation* between programmers and tools.

Objects First, Tools Follow: Choose, Configure, Build

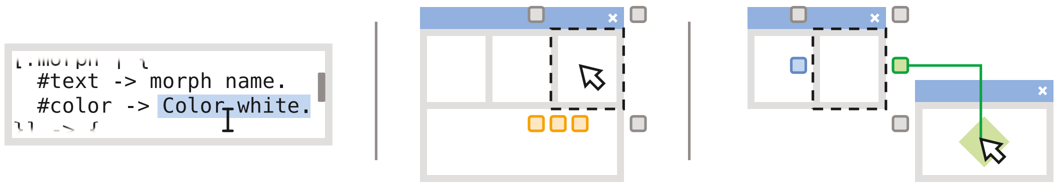
Scripts and panes can only be effective if domain objects are involved, which are usually provided from the environment. Any script code can refer to globals such as `Morph allInstances` and `WebClient httpGet: 'http://squeak.org'`. However, onward program exploration will trigger questions about *objects at hand* to be answered with different scripts evaluated with those objects. Consequently, the proposed working practice in Vivide begins with objects and treats scripts (or tools) secondarily.

Choose The act of choosing from existing means to approach a programming task is considered *very easy*, which hence concerns both tool users and tool builders. In traditional environments, programmers choose from existing graphical tools that offer different ways to fetch, examine, and retain software artifacts. There are (tool) button bars or pop-up menus that offer such choices. Noun-verb interaction will sometimes be available if such menus are attached to tangible artifact representations.

In Vivide, programmers choose artifacts from visible views to express interest in those artifacts' deeper relationships. Then, they choose from fitting scripts and views to make those relationships accessible. They can always choose to create a new task context for the prospective artifacts to explore. An exploration path emerges as the programmer continues choosing objects, scripts, and views in an iterative fashion. Thus, choosing in Vivide is an entirely visual cycle and does not require source code reading:



Configure The act of configuring a thing to accommodate a domain, task, or personal preference is considered *easy*, which hence also concerns both tool users and tool builders. In traditional environments, programmers face many defaults that are accepted in the particular programming (language) community such as file-based views for Eclipse/Java projects. The difficulty of changing such defaults depends on the anticipated configuration space. In Vivide, the situation is similar considering default scripts, views, and view compositions. Script authors can anticipate configuration by prominently placing *hard-coded values* to be found and changed by users. If well-documented, programmers can also add/change object properties such as `#icon` or `#color` with ease. This includes property renaming such as from `#value` to `#weight`. It also includes copy-and-paste from scripts with similar transform/extract steps. In pane (view) compositions, programmers can also influence pane geometry and choose different scripts or views. Finally, we consider the addition of object connections between panes a powerful way of tool integration that resides in the configuration space of any tool user. Hence, we argue that the novel means of configuration in Vivide are threefold:



```
  #text -> morph name.
  #color -> Color white.
```

Build The act of building a thing to accommodate a domain or task is always considered *possible*, but concerns programmers being tool builders only because it takes usually more time than configuration. In object-oriented environments, programmers create *adapter objects* to combine software artifacts with interactive widgets. Such adapters capture customizable query, mapping, and presentation languages and hence define a tool's utility and usability. In Vivide, programmers do not have to write generic, repetitive "glue" code but can focus on domain-specific transformation and extraction in an object-centered fashion. The composition of view context is mainly interactive and backed by automatically generated script properties. Since we see artifact provision and widget provision as extra activities, Vivide supports the following aspects of tool building: (1) authoring of scripts in general, (2) addition of object-oriented domain code to simplify script code, (3) definition of pane compositions and new kinds of pane views, (4) extension of views to support new kinds of script and object properties.

Mistakes and Error Handling

In the course of exploratory action, programmers can make mistakes that might lead to inadvertent results, waiting to be corrected. Encouragingly, Vivide provides an inherent "safety net" that keeps tools running. Even if a wrong script was chosen, a configuration turned out buggy, or building efforts reveal a conceptual fallacy: the worst implications will usually be (1) empty views or (2) otherwise lost artifact tangibility. While deceptive on-screen information can be challenging, too, we stress the need for a robust, forgiving environment that fosters exploratory working habits. Consequently, Vivide has a fit understanding of possible errors to complement our data-driven perspective:

- Scripts cannot be interpreted to generate the view model because objects fail to understand certain messages and raise (unhandled) exceptions.
- Script identifiers cannot be resolved because the current scope does not include script organizations that know about those.
- Views cannot accept a script or model because expected script/object properties are missing, yet no defaults are provided.
- Inadvertent user interaction, such as through changed views or changed object connections, discards or otherwise changes an object selection.

In most cases, panes can detect such errors and hence avert a disruptive user experience by providing empty/default models or selections. Programmers can then begin recovering from their mistakes either manually or tool-supported. We think that our UI-design language entails obvious hooks for tracing and undoing user actions. For example, script editors should provide traditional undo at the text-change level. Since pane views should always represent their configuration state as script properties and such properties should have a textual representation in script editors, accidental context changes, such as closing a window, would also be reversible.

On the Structure of Questions and Answers

The elements of our scripting language and our UI-design language can positively affect the conversation between programmer and environment. During program comprehension tasks, programmers pose questions and seek answers about the concepts behind and structure of software artifacts. Inadvertently and inevitably, the languages and tools in use carry a certain vocabulary that shape cognitive processes and hence this conversation. For example, experienced Smalltalk programmers would look for *implementors or senders of a symbol* rather than *definitions or references of a function*; they would search through *string properties in the object graph* and not *text snippets in the file system*; it would be about messages for *instance creation* and not *class constructors*. Yet, tool interfaces and applied languages can differ in their (domain-specific) vocabulary, which forces programmers to rephrase their intents and interpret observed effects. We argue that our Vivide environment can "bridge" many such articulatory and semantic distances so that users get the impression of a *direct manipulation interface* [75]. Such an impression fosters immersion and task focus through short, direct feedback loops to explore complex problems.

We refer to a specific catalog of 44 questions [177], which influenced our works due to its coherent overview of tool-supported program comprehension. There are four categories of questions, which reflect the common *divide-and-conquer* strategy for problem solving: (1) finding focus points, (2) expanding focus points, (3) understanding a subgraph, (4) questions over groups of subgraphs. The structure of each question in those categories maps directly to mechanisms and actions available in a self-supporting, object-oriented environment that provides our notion of a shared object graph, scripts, and panes. Consequently, missing tool support can be detected and specified in Vivide terms. Programmers can then either address the deficiency directly or schedule a dedicated tool building activity. In particular, the structure of questions maps as follows:

Finding focus points Programmers are looking for tangible, named things to get started. The question structure includes "UI element", "error message", "behavior", "exemplar", "entity named", or "unit". Thus, a list of objects can directly represent or scope/filter such focus points. In Squeak, the respective objects could be *morphs*, *strings*, or *classes*. The meta-object protocol can provide examples via `#someInstance` and harvest the object graph's text-based representations via `#printString`. That is, Squeak and Morphic provide the basic amenities themselves, while Vivide can help with scripts that exploit the environment's global state to make it more accessible to the programmer. We also think of scripts that require objects as "search terms"—presumably in the widest sense of the word.

Expanding focus points Programmers are looking for relationships between things to continue exploration. The question structure includes "parts", "part of", "siblings", "when [...] method called", or "arguments to this function". We argue that our scripting language has the primary purpose of traversing object relationships and hence expanding the programmer's focus points. Given a list of objects, the programmer can navigate through and choose from named scripts as explained above. At this point, the programmer can also write new scripts to accommodate the situation or follow a hunch. Note that we assume that the environment already provides the requested "expansions" like means to trace code execution or fetch external resources.

UNDERSTANDING A SUBGRAPH Programmers are looking into cross-cutting concerns, which involve multiple, mutually distant objects and hence entail navigational overhead. The question structure includes "how [...] concern [...] implemented", "execution path", or "data [...] look at runtime". In VIVIDE, programmers are in control of which artifacts are represented in which ways on screen. There can be multiple views for the same artifact structure. Ad-hoc object connections between panes can establish ad-hoc integration of otherwise self-contained tools. Considering the trade-off between screen size and information depth, we think that even bigger concerns can often be adequately represented side-by-side to reduce cognitive load.

QUESTIONS OVER GROUPS OF SUBGRAPHS Programmers are looking *beyond* individual concerns but address implementation strategies, development history, or impact analysis. Question structure includes "behavior vary over [...] cases", "UI types [versus] model types", or "direct impact of this change". Since Squeak's object graph represents the environment's state as-is, additional (reflective) information has to be expressed as specific kinds of objects. VIVIDE can provide self-updating scripts to inform programmers about change impact and task progress *if* any appropriate, computationally accessible metrics exist.

In VIVIDE, programmers will continuously answer the following questions to gather information:

1. What **output** (objects) transport supportive structure and concepts?
2. What **scripts** offer such objects accessible by exposing relevant relationships?
3. What **input** (objects) do these scripts need to populate views?

Programmers can answer these questions by getting familiar with tangible object representations, comprehending script labels/code, and configuring pane/script input buffers. Thus, VIVIDE terms can entail an increased awareness about artifacts, which makes up the majority of information needs [91] in programming.

Programming is still a text-heavy activity as programmers have to, eventually, read and write source code to fix bugs or add features. Thus, taking an object-oriented perspective on a programming task can be challenging if many lines of source code overshadow the environment's primary transport medium. In Squeak/Smalltalk, larger chunks of code are likely to spread across several methods, which frame noticeable boundaries and thus artifacts. Within each method, however, there is only text, which naturally makes programmers interpret and imagine. Consequently, (mental) references to *actual* objects fade, and programmers might start looking for tools that answer arbitrary high-level questions. Such a turning point impairs our data-driven perspective and thus Vivide mechanics. In Squeak, there is the notion of *bindings*, which are objects associated with symbols in code. If code editing tools can manage to always provide such bindings, then programmers can learn to treat text as a merely flexible UI and each word/symbol as a handle for an actual object. Then, text and text-heavy views can retain tangibility and object focus.

Synopsis In the Vivide environment, we have the tool-building mantra: objects first, tools follow. The combination of Squeak's objects and Vivide scripts/panes suggests a data-driven working practice. On screen, programmers look at comprehensive artifact structure, not decorated tool windows. Mistakes in use or configuration can be narrowed down to selected panes whose views might become empty, but the overall "tool" keeps running.

Vivide can positively affect the conversation between programmer and environment because the typical structure of questions and answers corresponds to its building blocks. Program comprehension puts a strong focus on artifacts and their relationships, which supports programmers to directly derive actions from their intents. (1) Which objects do I have? (2) Which objects do I want? (3) Which scripts make that happen?

Summary

We proposed a block-based scripting language, which uses regular Smalltalk code (i.e., block closures) to concisely express the semantics of *object transformation* and *property extraction*. In each script, multiple such transform steps and extract steps define multiple levels of a *tree structure* to frame significance in the environment's object graph. Scripts (and steps) have *identifiers* to express re-use. During lazy script interpretation, the amount of objects exchanged between steps determine completion. Finally, we proposed the use of *tuples* as anonymous data structures to reduce the need for writing new classifications as regular object-oriented code but focus on script authoring.

We proposed a new data-driven tool design strategy to form a more modular presentation language for all tools in the environment. Putting an emphasis on *single-object tools*, we declared the Rule of *Distinctiveness*, the Rule of *Similarity*, and the Rule of *Context* to enable the construction of complex tools whose building blocks are accessible and comprehensive during use. Any combination of tools that exchange information form a *composition context*, which can represent new relationships and tools of their own accord.

We proposed a new morph-based UI-design language to configure, combine, and abstract all elements of a tool's presentation language. We employed our scripting language to generate *view models*, and a generic morph called *pane* to support interactive re-design during use. A special view called *pane view* enables the arbitrary composition of different layout strategies (or window managers) to further accommodate tasks or personal preferences. We added *pane decorations* (e.g. pane halo) to provide access to all "meta" actions such as view resizing, tool integration, and script access.

We proposed a new data-driven working practice so that programmers can directly benefit from VIVIDE in program comprehension tasks. *Choosing* tools becomes choosing objects, scripts, and views in a continuous feedback loop. *Configuring* tools becomes changing simple expressions in scripts, modifying pane geometry, and re-wiring panes via object connections. *Building* tools encompasses all of the above and the design of object representations for new kinds of software artifacts or new kinds of views. We think that program comprehension questions can be simplified to (1) which objects provide relevant information, (2) which scripts provide such objects, and (3) which objects are needed by such scripts to populate views.

[|]

In the next chapter, we elaborate on more details of our solution. The strategies and languages we proposed in this chapter offer a range of possibilities and depth. We will explain our current thoughts on that topic.

5 Advanced Vivide Concepts and Implementation Guidelines

The Vivide environment offers ideas that affect tool construction and programmer behavior in the scope of exploratory program comprehension. Yet, concrete applications can go beyond "mere" graphical tools for programming. We think that the languages (section 4.1, section 4.3) and strategies (section 4.2, section 4.4) we proposed can serve as a baseline for many related ideas to improve the programming experience.

In this chapter, we take the next step and elaborate on advanced concepts and details we approached so far with Vivide. First, we expand our *scripting language* to cover recursion, concurrency, model updates, and object grouping. Then, we apply our data-driven strategy for tool building to design interactive *script editors*, which includes convenient code folding and templates. After such tools for script authoring, we expand our *UI-design language* to cover aspects of view documentation, complex object connections, and advanced script design. Finally, we broaden our *working practice* by explaining the support for interleaving tasks and (semi-)automated user interaction.

5.1 Advanced Concepts in the Scripting Language

Like other programming languages, our scripting language leaves room for improvement to accommodate best practices or patterns that are revealed during actual use. Our goal is to keep the language simple but expressive enough to support customizable exploration of, maybe changing and complex, artifact structure.

The Collection Protocol

A script step has code that captures the conversation of two buffer objects: input and output. Both buffers are *collections* [32], which understand at least the following messages to realize many semantics known from the functional programming paradigm. We explain the intent of these messages in the context of software artifacts and the task of program comprehension to better relate to our assumptions about the programmer's mindset:

`#select:` Pick all software artifacts that fulfill custom Boolean criteria, usually shaped by task relevance.

`#reject:` Discard all software artifacts that fulfill custom bBolean criteria, usually shaped by task relevance.
`#collect:` Construct a list of software artifacts by associating to each old artifact a new one, usually for navigating the information space.
`#detect:ifNone:` Find a single artifact that fulfills custom Boolean criteria in a list of artifacts, and provide a default artifact for unmatched criteria.
`#inject:into:` Aggregate all software artifacts into a single new artifact, usually constructed using math or text operations.

There are other messages in the collection protocol that support the programmer's explorative trial-and-error mode:

`#sorted:` Construct a list of software artifacts with a different order following custom sorting criteria.
`#reversed` Construct a list of software artifacts that will have all elements in reverse order.
`#first:` Construct a list of software artifacts with the first n elements of the old list.
`#last:` Construct a list of software artifacts with the last n elements of the old list.
`#gather:` Construct a list of software artifacts by associating to each old artifact a *a list of new ones*, usually for navigating the information space. Not part of the Smalltalk-80 implementation.
`#flattened` Construct a (flat) list of software artifacts from a (deeply) nested list. Only present in Squeak 5.0 or above.

In general, transform steps require means to iterate over lists of artifacts and to add artifacts into the output buffer:

`#do:` For each artifact in a list, evaluate some Smalltalk code.
`#add:` Put an artifact into the output buffer.
`#addAll:` Put a list of artifacts into the output buffer.

The bottom line is that there is *some* glue code that remains exposed to programmers. All messages that read from the input buffer and write to the output buffer are considered glue. In complex or poorly formatted scripts, programmers may still have difficulties to discover the domain rules. However, if script steps are small, we think that tools can help expose such domain rules through visual cues.

Script Cycles and Recursive Models

Programmers can use script identifiers to describe *recursive* tree (or model) structures. As described in section 4.1, model nodes will only have children if the transform step(s) write objects into the output buffer. Thus, we can safely re-apply object transformations to generate all levels of the model tree—until the step's outputs dry up. Programmers can represent such recursion as *cycles*, which are expressed as references to some preceding step via the script property #next. An example script that describes the Morphic hierarchy can look like this:

```
script := {
  [:in :out | in do: [:morph | out addAll: morph submorphs]]
      -> { #id -> #submorphs .
           #label -> 'Morph Hierarchy' }.
  [:in :out | in do: [:morph |
     out add: { #object -> morph. #name -> morph name }]]
      -> { #isProperty -> true.
           #next -> #submorphs  }.
} asScript.
```

However, programmers cannot always know in advance whether transform steps will dry up at all. It can be surprising that a certain path in the environment's object graph is cyclic and hence the resulting tree will have an *infinite depth*. We propose *two strategies* to handle this situation: (1) lazy evaluation and (2) interactive suspension. First if model nodes are *generated on demand* through user-view interaction, users will have a chance to recognize endless repetition on their own via the tool's presentation language: they can see it. Second if programmers work in an interactive environment such as Squeak, they will be able to interrupt code execution with a keyboard shortcut, even when the user interface becomes unresponsive [192]. We think that there are other strategies, such as *external monitors* [182] that become proactive. We focus on an interactive setting where programmers can intervene if necessary.

Note that if a script reference is used to describe a recursive tree, it will be *pointless to add more steps* after that reference. Such a cycle will only end if transform steps dry out, and any step after that will have no input. If, however, references just aim for reusing existing transformations or extractions to avoid code duplication, script interpretation will continue as usual after the identifier was resolved and the script interpreted. For example, scripts in the global organization can be good candidates for such re-use.

Concurrent Script Interpretation

As the choice of buffer format influences the script code, our scripting language can be applied in a way that supports background[1] interpretation to some extent. That is, our scripting language can remain flexible in terms of its interpretation semantics. Buffers with *thread-safe stream* (or shared queue) semantics can improve the programming experience without sacrificing much readability when querying and mapping artifacts. For example, fetching artifacts from a distant database over a *network* can take time and hence disturb the responsiveness of programming tools. Script properties can configure script interpretation as a background activity:

[1]In Squeak, there is a single (foreground) process that handles user-interaction and drawing requests. Most Smalltalk code runs in that process. There is often no need for process synchronization.

```
script := {
  [:in :out | [in atEnd] whileFalse: [in next in [:url |
    out nextPut: ( WebClient httpGet: url  "Takes time.") ]]
       -> {  #in -> ReadStream . "Preferably thread−safe."
             #out -> WriteStream . "Preferably thread−safe."
             #async -> true  }.
} asScript.
```

With such streams, programmers *must not* apply the collection protocol [32] to read and write input and output buffers. Instead, they have to use while-loops to fetch one object after another off the stream. One might think of even endless loops for potentially endless streams of objects. Having such concurrency involved, script interpretation becomes more difficult to understand because there can be many script interpreters (or script steps) in execution. Note that we will not further explore background script interpretation in this work.

Object Change and Model Update

Objects can change, and models have to be updated. Any transform step might produce different results when interpreted at different times on the same list of objects. Also, extracted object properties that represent (virtually) immutable objects such as strings and numbers can become obsolete soon after node creation. For example, a morph's position can change and so will its bounds, but any node based on that morph will *not* be updated by default. Such node will still refer to a rectangle object that represents the former bounds.

Scripts provide two means to manage changing objects: (1) block-based properties for node resilience and (2) object notifiers for node replacement. Block-based properties allow for *selected re-extraction* of object properties with the help of block closures. Object notifiers represent adapters [62, pp. 139–150] that integrate with existing observer patterns [62, pp. 293–303] or similar event sources for *selected re-construction* of model nodes, that is, levels in the tree. While block properties will only be effective if the client continuously asks for node properties, object notifiers provide more convenience through automation yet entail script re-interpretation.

The following example uses a *block-based property* for extracting a morph's color:

```
script := {
  [:in :out | in do: [:morph |
    out add: { #object -> morph.  #color -> [morph color]  }]]
       -> { #isProperty -> true }.
} asScript.
```

Note that `morph` will be retained in the object property `#color`. If clients ask for that property, the node will transparently evaluate the block closure and reveal the current color object. If the property is a block itself, there must be a second block wrapped around.

For the script that describes the graphical hierarchy, we can use an *object notifier*, which is defined as a script property, that triggers every second to re-construct the tree:

```
script := {
   [:in :out | in do: [:morph | out addAll: morph submorphs]]
    -> { #notifier -> [ViTimedNotifier every: 1000] "milliseconds" }.
} asScript.
```

Note that script properties do also support block closures. Programmers can configure notifier objects with regular object-oriented code and retain global objects such as classes. Conceptually, object notifiers should have access to the objects they observe. During script interpretation, notifiers can be created and configured accordingly. For example, Squeak provides a global "system-change notifier", which triggers, among others, on classes changes, but observers have to filter because subscription is generic. Since notifiers are associated with steps, each level in the model tree can have a different kind of notification.

[|]

Besides reading, extract steps can hold the *rules to modify* objects (or artifacts). We designed the scripting language primarily for program comprehension tasks. Artifact modification can be seen as a part of comprehension; bug fixing or feature adding are typically mixed with ongoing exploration. Programmers can express such rules for tools directly in the extract step:

```
script := {
   [:in :out | in do: [:morph | out add:
      { #object -> morph.
        #color -> morph color
               <- [:newColor | morph color: newColor] }]]
        -> { #isProperty -> true }.
} asScript.
```

In this example, the programmer anticipates the possibility of modifying a morph's color via clients. Views that try to *write* into the particular model node will *execute* this rule, which is stored as another block closure:

```
"Script code (written by tool builder)"
[:newColor | morph color: newColor].
"View code (written or derived by view designer)"
node at: #color put: Color yellow.
"Effective code (derived by VIVIDE)"
morph color: Color yellow.
```

When objects are changed this way, the extracted property in the node will update, too. Again, programmers can focus on their domain-specific artifacts and express reasonable rules to look at or modify artifact structure concisely at the same place. In this work, we will only briefly discuss artifact modification.

Tuples for Object Groups

Tuples can be used to form *groups*, which combine objects in terms of *key* objects as a group's representative:

```
group := #(1 ((1 a) (1 b) (1 c))).
group first = 1. "head or key object"
group second = #((1 a) (1 b) (1 c)). "tail"
```

Due to the broad definition of tuples being object arrays, *groups are tuples* in the form of pairs of key object (or head) and grouped tuples (or tail). Note that key objects are usually handled *by value* (or state) and not object identity because programmers work with artifact *structure*. So in scripts, such keys are compared via `#=` not `#==`.

The following example illustrates the *tuple format* that is required for grouping:

```
groups := #( (1 a) (1 b) (1 c) (2 a) (2 b) (2 c) ) groupByFirst.
groups = #( (1 ((1 a) (1 b) (1 c))) (2 ((2 a) (2 b) (2 c))) ). "true"
```

Here, a collection of pairs forms two groups because the first object in each pair takes only two different values: 1 or 2. If all source tuples have the same size, any *index* in the tuple could be used to pick the key objects: `#groupBy: n`.

In the following, we show how groups can help extract object properties that define levels in the tree model. For example, morphs could be grouped by color like this:

```
script := {
   "Level 1 – Group morphs by color."
   [:in :out | in do: [:morph | out add: { morph color . morph} ].
   [:in :out | out addAll: in  groupByFirst  ].
   [:in :out | in do: [:group | [:color :tuples | out add: {
      #object -> color . #name -> color name. #objects -> group }
      ] valueWithArguments: group ]] -> { #isProperty -> true }.
   "Level 2 – Discard group/color object, extract morph name."
   [:in :out | in do: [:group |
      [:color :tps | out addAll: tps] valueWithArguments:  group ]].
   [:in :out | in do: [:tuple |
      [:color :morph | out add: morph] valueWithArguments:  tuple ]].
   [:in :out | in do: [:morph |
      out add: { #object -> morph. #name -> morph name }]]
         -> { #isProperty -> true }.
} asScript.
```

In Level 2, the programmer has access to the group's key object `color`. We think that it helps simplify the semantics of script interpretation to *not discard* the key object automatically. However, the overhead for such simple grouping indicates, again, potential tool support such as by hiding repetitive syntax. We suppose that many scripts will not require programmers to be in full control of the input and output buffers all the time. Recalling our motivational example with Unix' pipes-and-filters from section 3.3, the same script can, if supported by tools, look like this:

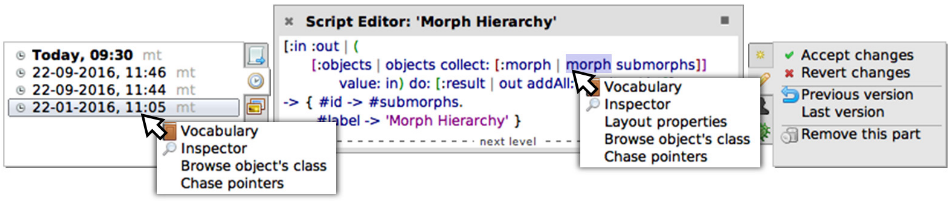
```
script := {
   "Level 1"
   [:morph | { morph color. morph } asTuples ].
   [:tuples | tuples groupByFirst ].
   [:color :tuples | { #name -> color name }].
   "Level 2"
   [:color :tuples | tuples].
   [:color :morph | morph].
   [:morph | { #name -> morph name } ].
} asScript.
```

In this simplified script format, programmers can still read domain-specific vocabulary to understand object transformation and property extraction. Of course, this cannot be plain text but an *interactive view* on objects that transport structure of the script artifacts. While we designed a scripting language to improve accessibility and maintainability of the object graph in object-oriented systems, the artifacts of that language are integrated in the *same* environment. Depending on the tool building intentions, our scripting language remains flexible and can take many forms if supported through *script-edit tools*.

Synopsis Our use of Squeak's collection protocol for scripts reduces *glue code* while exposing domain-specific transformation rules in a functional-programming style. That is, there is still generic glue left to read and write. — Model trees can be *recursive* and of *infinitive* depth if script references form cycles. Therefore, model nodes are constructed lazily level by level. Either programmers or view designers have to be careful to detect such infinity to avoid an *unresponsive* environment. — We think that the same scripting language could be extended to support *background interpretation* of scripts. Script properties are sufficient to configure synchronized buffers and additional hints for the script interpreter. — Model nodes can be invalidated with the help of *object notifiers*, which are stored as script properties. Using blocks, object properties can also hold the *rules to read and write* artifact structure. These rules are triggered through views after the (lazy) node construction. — Finally, programmers can extend tuples to form *groups*, which are also tuples, to align sets of objects (or object properties) with levels in the model tree. This would otherwise require the addition of custom classifications.

5.2 The Strategy for Tool Design by Example

In this section, we will employ our new strategy to design *script editors*, which can hide repetitive code sections, offer template-based code generation, and manage script properties. We then apply those single-object tools for scripts in a *composition context* to form a configurable script-editing experience that integrates run-time information for debugging with short feedback loops.



```
[:in :out | (
    [:objects | objects collect: [:morph | morph submorphs]]
        value: in) do: [:result | out addAll:
-> { #id -> #submorphs.
    #label -> 'Morph Hierarchy' }
```

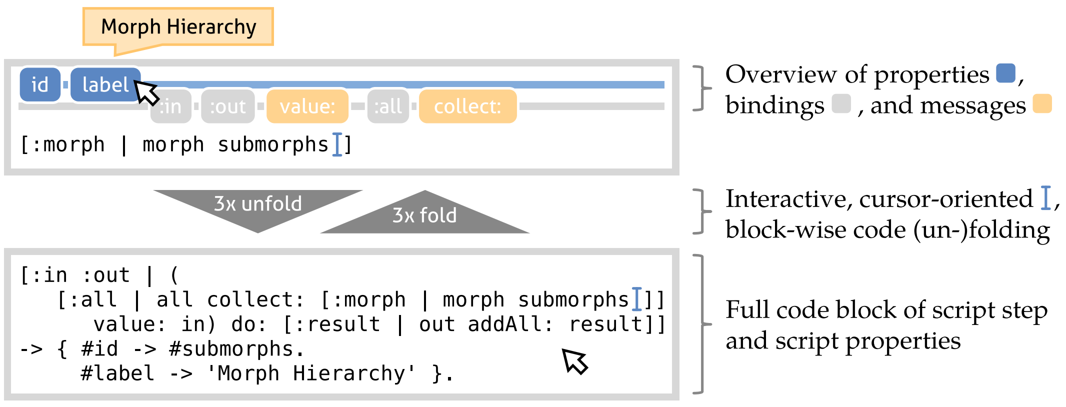
Figure 5.1: Our implementation of a single-object tool that represents a single script step. Related objects (left) include subsequent steps, versions, and script properties. Object selection within the view is text-based if the code editor has bindings for its symbols (middle: "morph"). The available actions with side-effects (right) resemble the ones expected from any text-based code editor.

The Script Editor

We decided to represent each step in a script as a separate object. That is, there is no extra object that encompasses all steps, which might wrongly be inferred from our use of object arrays for convenient script creation in section 4.1. To recap, each step knows its next step if any, the first step in each script *is* the script object, and hence each step can form a script (object) on its own. We propose a design for a single-object tool in figure 5.1, which applies our practices from figure 4.5. In that script editor, programmers can read and change script code as they would in a normal code editor. The code even looks like regular Smalltalk: an association between an in-out block and a list filled with script properties.

We follow the Rule of Distinctiveness, but we mostly ignore the Rule of Similarity by anticipating tool composition. First, script code looks different than class definitions or method sources because of the beginning "`[:`" and the overall shape "`[] -> {}`". Second, we applied our practices for coherence and integration by (1) separating related objects in a menu and (2) providing object selection for bound symbols in the code. We think that valuable relationships include subsequent script steps, code versions, and examples for script properties. We think that symbol bindings have to be established from a tool composition context, which we explain in the next section. Regarding the Rule of Similarity, we think that such text-based editors should not handle more than one object. The presentation of multiple steps in a single editor would be possible via code concatenation but inappropriate because it would impair each step's tangibility. Considering the idea of shared concepts, script properties should be used to show and modify a custom label, icon, summary, color, or origin. We think that the text form is sufficient for all kinds of properties in this view. Note that there can be other tools in the environment that show script steps, including their properties, differently.

The script editor offers two additional features to support code reading and code writing: (1) block-based code folding and (2) template-based code generation. These features exemplify the flexibility that single-object tools can provide while remaining data-driven and tangible.


Morph Hierarchy
id
label
:in
:out
value:
:all
collect:
3x unfold
3x fold
Overview of properties 🟦, bindings ⬜ , and messages 🟨
Interactive, cursor-oriented I, block-wise code (un-)folding
Full code block of script step and script properties

```
[:morph | morph submorphs]
```

```
[:in :out | (
   [:all | all collect: [:morph | morph submorphs]]
      value: in) do: [:result | out addAll: result]]
-> { #id -> #submorphs.
     #label -> 'Morph Hierarchy' }.
```

Figure 5.2: Our script editor resembles a source code editor with support for single script steps. Programmers can fold code at block boundaries to stay focused. The top shows a folded example with a compact overview of hidden information. The bottom shows all details as regular Smalltalk code.

Block-based Code Folding

Script editors improve code readability via *block-based code folding*. Due to the generic structure of in-out blocks, there are some (repetitive) elements that are likely to obfuscate the actual rule about object transformation or property extraction. For example, programmers do always have to `#add:` objects into the output buffer, and they do usually `#collect:` objects from the input buffer. Thus, we allow programmers to hide (or fold) sections before and after blocks to focus on what is inside the respective block.

Compared to code folding in sophisticated text editors, we do not sum up nested/inner elements but outer ones. As depicted in figure 5.2, programmers can "dive into" the block where the text cursor is currently located. When folded, there are adornments at the top border of the editor to show script properties, variable bindings, and messages. Without having to unfold, such adornments help programmers separate blocks for sorting from filtering, for example.

Additionally, we combine interactive, typically keyboard-driven, folding with a useful default folding state. For arbitrary Smalltalk code, it is difficult to automatically detect the best starting point. However, we explored a convenient heuristic to always fold to the *left-most, inner-most* block in the syntax tree. Since Smalltalk also employs blocks for conditions and loops, the editor should unfold automatically in those cases. Note that we anticipated such tool support for the design of our scripting language.

Script Wizard: Creation Templates

We add a template mechanism, which we call *script wizard*, to reduce the overhead for writing new scripts. That is, the wizard complements block-based code folding, which is used after creation and during onward modification. *Script templates* append and prepend generic boilerplate code (or glue) like this:

```
"Code to prepend automatically:"
[:in :out | ( [:all | all collect:

   "Code to write manually:"
   [:morph | morph submorphs]

"Code to append automatically:"
] value: in) do: [:result | out addAll: result asList]].
```

Again, we rely on simple heuristics. It is sufficient to look for *short patterns* in the syntax tree. Regarding our tool building context, to edit scripts means to change some tool's view in the environment. Hence, programmers can quickly detect false-positives and then debug. We use the following information from nodes in the syntax tree to form patterns:

- Names of message sends
- Number of arguments in blocks
- Names of arguments in blocks
- Statements that form associations

For associations, we are only interested in literal receivers such as "`#label -> 'Morphic Hierarchy'`" because those match the syntax of object/script properties.

We introduce a *new message* to `Object` to conveniently wrap single objects into arrays and leave collections as-is: `#asList`. Having this, we can reduce the number of different templates. Programmers should be supported to manually migrate from one template to another without too much effort. Hence, templates should look similar if possible.

[|]

In the following, we will describe the patterns in natural language and show the corresponding template as Smalltalk code. In each template, the gray box exemplifies the pattern as code, too. That box will be replaced with the programmer's code snippet if the respective pattern matches. We begin with the **general patterns**:

- If programmers describe a statement that forms a block associated with a literal array, integrate that array as script properties with any existing properties:

  ```
  [:in :out | ... ] -> { ... #label -> 'Foo' ... }
  ```

- If nothing else matches, assume that programmers want to perform a one-to-one or one-to-many mapping. The basic idea for tool-supported script authoring is that programmers should never have to write the `#collect:` call to the input buffer. Programmers can discard objects by mapping to `nil` because `#asList` will produce an empty array:

```
[:in :out | (
   [:all | all collect: [:each | ... ] ]
      value: in) do: [:result | out addAll: result asList]]
```

Programmers can choose to ignore the input, or they can choose to ignore the wizard. Hence, there are **extra patterns**:

- If programmers write a block with two arguments named `:in` and `:out`, assume that this code represents the entire script specification:

```
[:in :out | ... ]
```

- If programmers write a block with no arguments, which implies the use of literals or globals, ignore the input buffer and assume that either a single object or a list of objects is returned by that block:

```
[:in :out | out addAll: [ ... ] value asList]
```

In our scripting language, we employ the Smalltalk's collection protocol to a great extent. Consequently, there are **collection patterns** that consider the work with collections and thus accessing the entire input buffer:

- If programmers use the messages `#select:`, `#reject:`, or `#inject:into:` in their code, assume access to the entire input buffer, not single objects. Note that `#inject:into:` does not have to construct a collection due to `#asList`:

```
[:in :out | out addAll: ( [:all | ... ] value: in) asList]
```

- If programmers use other parts of the collection protocol for a many-to-many mappings, the same template will apply. We think of the messages `#sorted:`, `#reversed`, `#first:`, `#last:`, `#gather:`, and `#flattened`. This list should be extensible.
- If programmers look for a single object in the input buffer via `#detect:ifNone:`, the same template will apply due to `#asList`.
- While tuple construction via `#asTuples` does not need attention, the concept of grouping requires access to the entire input buffer, too. Hence, if programmers use the message `#groupByFirst`, the same template will apply.

Besides transform steps, programmers will describe extraction steps to enrich model nodes with supportive information and to express different levels in the tree. There are **extraction patterns** to cover that:

- If programmers describe one or more associations with literal receivers in a block, assume that an extract step with the required script property `#isProperty` should be constructed:

  ```
  [:in :out | out addAll: ([:all | all collect: [:each | (
     [:object | { #text -> object asString } ]
       value: each), { #object -> each }]] value: in)]
  -> { #isProperty -> true }
  ```

- If programmers do also control the object transformation here, avoid the duplication of `#object`:

  ```
  [:in :out | out addAll: ([:all | all collect: [:each | (
     [:object | { #object -> object } ]
       value: each)]] value: in)]
  -> { #isProperty -> true }
  ```

While one-argument blocks indicate regular object transformation or property extraction, blocks with more than one argument imply tuple usage. Except for fully qualified in-out blocks, there are **tuple patterns** to match:

- If programmers write a block with more than one argument, assume tuples to be unpacked and then mapped one-to-one or one-to-many. If programmers want to access the entire input buffer, such as for sorting, they have to handle their tuple format manually. Tuple unpacking also applies to tuples that form groups:

  ```
  [:in :out | (
     [:all | all collect: [:tuple |
        [:a :b :c | ... ] valueWithArguments: tuple]]
          value: in) do: [:result | out addAll: result asList]]
  ```

- If programmers describe associations with literal receivers *and* use more than one block argument, assume tuples to be unpacked and object properties to be extracted. Use the first object in each tuple as priority object to support grouping:

  ```
  [:in :out | out addAll: ([:all | all collect: [:tuple | (
     [:a :b :c | { #text -> b. #color -> c } ]
       valueWithArguments: tuple),
     { #object -> tuple first. #objects -> tuple }]] value: in)]
  -> { #isProperty -> true }
  ```

- If programmers do also control the (priority) object transformation here, avoid the duplication of `#object`:

  ```
  [:in :out | out addAll: ([:all | all collect: [:tuple | (
     [:a :b :c | { #object -> a. #text -> b. #color -> c } ]
       valueWithArguments: tuple),
     { #objects -> tuple }]] value: in)]
  -> { #isProperty -> true }
  ```

Exploratory programming is an important theme in this work, which includes exploratory tool building. Especially for collection patterns, the re-use of templates indicates a trade-off between exploration and planning. The use of `#asList` is not inevitable, yet it helps. We think that programmers can easily migrate manually between the templates for general patterns, extra patterns, and collection patterns if their requirements change. However, we expect additional effort when migrating from templates for extraction patterns and tuple patterns because these patterns involve more code re-writing. If possible, programmers should then consider a full re-write from scratch using the script wizard again.

Script Editor(s) in Composition

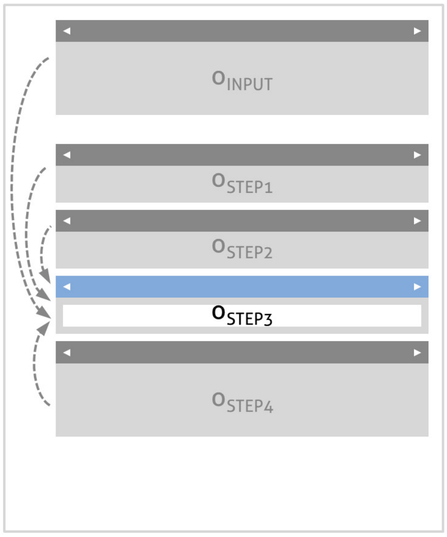
The environment should provide *composition tools* that focus on context management largely independent from particular domains or tasks—like a modular and re-usable variant of (global) window managers [127, 74]. We designed such a generic composition tool for script editing as depicted in figure 5.3. While users control the objects in the composition, the container (tool) is a generic mediator [62, pp. 273–282] that provides access to all context objects for all tools in the context without exposing the tools' interfaces. Again, the tool builders are encouraged to think in terms of objects not tools. Since script editors are basically code editors, we are in favor of a vertically tiled layout strategy with support for scrolling to overcome limited screen space. We argue that such a layout is beneficial for list or text widgets which often become space-efficient at a wide ratio. For example, typical Smalltalk methods show more characters per line than lines.[2] Composed vertically, programmers can see more code at a glance than they would with a horizontal strategy. For the same reason, programmers can read the source code of scripts with many steps efficiently.

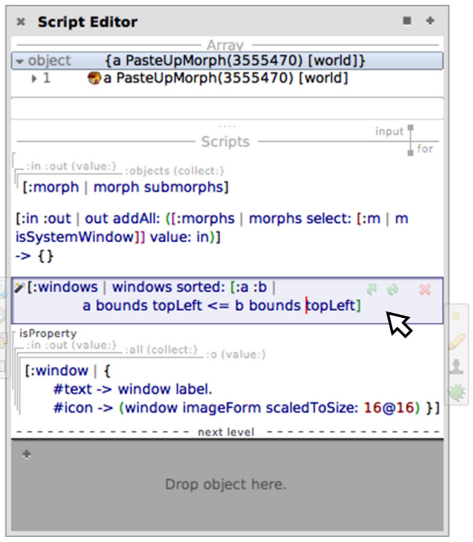
The actual "script editor" is a composition tool that is filled with single-object tools for the script steps and exemplary input objects. As described above, tool builders should design compositions of tools that support programmers, too, not just single-object tools. Instead of our script editor being able to handle multiple steps by itself, we designed such a composition to modify scripts.

Each script editor in the composition makes use of context objects to (1) bypass the fact that steps only know their next step and (2) to provide a short feedback loop with run-time information when editing code.

First, the editor's complete access to the entire script structure is used to add, remove, and re-order steps. If a tool *in* the composition asks for the addition of an object, the composition tool will open a new tool. If a tool *in* the composition asks for the removal of an object, the composition tool will close the respective tool. Note that this affects only the tool level and not the artifacts (or objects) that are represented. Programmers can drag-and-drop tools *in* the composition to change the order of

[2] In Squeak 5.1, methods with 10 lines or less have 23 characters per line on average. These are 79.6 percent of all methods. See appendix A.3 for details.





```
{a PasteUpMorph(3555470) [world]}
a PasteUpMorph(3555470) [world]

[:morph | morph submorphs]

[:in :out | out addAll: ([:morphs | morphs select: [:m | m
isSystemWindow]] value: in)]
-> {}

[:windows | windows sorted: [:a :b |
        a bounds topLeft <= b bounds topLeft]

[:window | {
    #text -> window label.
    #icon -> (window imageForm scaledToSize: 16@16) }]
```

Figure 5.3: Our script editors make use of composition context implicitly. Such context usually consists of the script's steps and other objects that serve as input for (partial) script evaluation and debugging. A vertical, scrollable layout reduces interaction overhead and reveals the step order. Users can change the context by adding input objects or re-ordering steps.

steps. For example, they can swap sorting and filtering to improve performance. As shown in figure 5.3, the vertical alignment suggests an order that is not only visible to the tool user but also to the tool builder and hence the tool itself.

Second, the composition context can include non-script objects, which are used to evaluate scripts to provide run-time information that should help programmers write or understand script code. To recap, scripts support the description of tree models to be used as view models or for non-visual, data-processing tasks. Hence, the model nodes are the unit of debugging in a setting where graphical views do not help. Partial script evaluation becomes directly accessible because the user can choose a certain editor and ask for the *results* of the respective step. This is a typical code comprehension task, where programmers fetch and examine artifacts and maybe retain them to understand effects.

In addition to any explicit evaluation, script editors use context to evaluate implicitly to (1) suggest names for the script wizard expecting a general pattern, (2) bind symbols in script code for object selections as depicted in figure 5.1, and (3) analyze code dynamically to warn about long-running scripts or infinite cycles. Recall that if programmers omit to alternate transform steps and extract steps in cycles but transform only, interpretation will not stop. Frequent use of script references may produce such infinite cycles by accident. Additional tool support can help recover from such accidents.

Synopsis We apply our tool design strategy through a *script editor* that looks like a regular code editor but supports reading and writing via block-based code folding and template-based code generation. Multiple such editors in *composition* represent the tool support for tool builders. Each editor makes use of its composition context to (1) bypass its own structural limitations and (2) integrate user-collected artifacts for debugging purposes.

5.3 Advanced Concepts in the UI-design Language

The interplay of scripting language and UI-design language leaves room for experimentation. In this section, we provide more details on how to use object properties and script properties to configure views in general and pane views in particular. We then explain advanced techniques for object connections and pane decorations.

Scripts Applied: Document and Store Properties

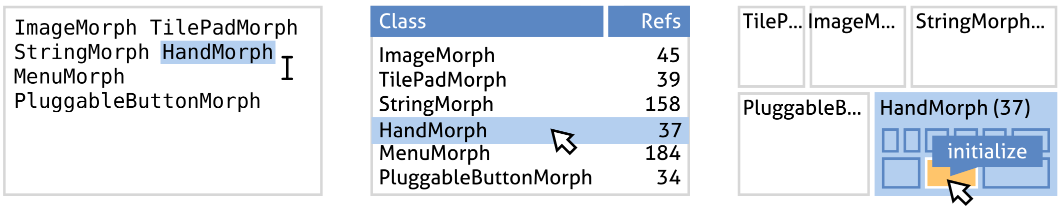
Compatible views should agree on *common properties* so that tool builders can try out different views while writing scripts. On the one hand, our scripting language helps evolve model structure from list-shaped over table-shaped up to tree-shaped. In figure 5.4, there is an example of how three different views can treat the same[3] view model:

- The text view ignores all but the first `#text` property in a node.
- The table view ignores all but the *first level* in the tree model.
- The tree view can also display *tooltips*.

On the other hand, there is our Rule of Similarity from section 4.2, which proposes the use of shared concepts to simplify tool building. We argue that object properties and script properties can reflect those concepts. Many views exhibit obvious similarities such as text labels and picture labels, which should be integrated by using common keys such as `#text` and `#icon`. This re-use supports script and view evolution. If tool builders figure out a way to represent an object as a `#color` but the current view does not (yet) support it, there is no need to discard the idea, but it should be preserved in script code. For example, current tree maps might need a `#weight` to calculate the filling area, but future tool building efforts could agree on a general `#number` property to represent domain artifacts. Then, other views could show useful information with the same script (or view model).

Besides the vocabulary of domain artifacts, tool builders have to learn about the concepts view providers expect. Object properties and script properties play an important role for configuring the mapping language and also forming the overall

[3]See appendix B.2 for script details.



```
ImageMorph TilePadMorph
StringMorph HandMorph
MenuMorph
PluggableButtonMorph
```

Class	Refs
ImageMorph	45
TilePadMorph	39
StringMorph	158
HandMorph	37
MenuMorph	184
PluggableButtonMorph	34

Figure 5.4: The same script can show different effects when applied to different kinds of views. Tool builders can use simpler views in the beginning and more complex ones as the script evolves or the task demands. From left to right: text view, table view, tree-map view.

presentation language of the tool. We think that this can entail an additional kind of *documentation*, which the designers of views should prepare. For example, the model-view framework in Qt formulates an extensive list of *display roles* its own views support.[4] Due to the dynamic nature of Squeak and Smalltalk, such an awareness is more important because the use of *reserved keys* can trigger debuggers deep in the Vivide framework's control flow:

- At the time of writing, reserved script properties are `#view`, `#id`, `#next`, `#notifier`, `#in`, `#out`, and `#async`.
- At the time of writing, reserved object properties are `#object` and `#objects`.

While tools such as the aforementioned script editor can help prevent such mistakes, the Squeak environment provides many different, flexible, often unprotected ways to change script objects. Nevertheless, the application of our block-based scripting language and our morph-based design language reflects a fair trade-off between flexibility and attentiveness. It complies with the programming habits practiced in interactive, self-supporting environments such as Self [207], Squeak/Smalltalk [192], and Lively Kernel [78].

Views should *store configuration state* in script properties. Since panes give views access to script objects, views can configure themselves independent from the view model (or the domain objects). We think of color schemes, table headings, or layout properties. In this regard, it makes sense to distinguish two sub-roles in the tool building context:

- **View creators**, which *declare* supported (or required) object and script properties.
- **Script authors**, which *define* (and use) object and script properties.

A *modular tool design* should consider interactive views that manage custom state in addition to the model's objects and extracted object properties. In Squeak, interactivity origins from morphs that react to user input. Consequently, each view morph provides custom controls and event handlers. While the view model should

[4]Qt cross-platform software development kit, Model/View programming: `http://doc.qt.io/qt-5/model-view-programming.html`, accessed 2018-09-19

always be the main source of information, additional configuration state is likely to be persisted across view instantiations. We see several options for view creators:

1. Read from and write into a dedicated database such as Squeak's `Preferences` and custom class variables.
2. Only write into such dedicated database but let script authors handle the read via script properties.
3. Use script properties to manage configuration state that is independent from domain objects.

We encourage programmers, especially view creators, to refrain from using global state, that is, dedicated databases that do not consider (local) tool context. Following the third option, view creators should make use of script properties because scripts represent such context without further ado. We suppose that conflicting views—the ones that interfere with each other's configuration keys—could be isolated with an additional *configuration namespace*. The database that would manage such namespaces, however, should consider tool context, too, to not become another global data structure.

Scripts Applied: Properties for Pane Views

A pane view's model should be a *list* that reflects the script input. The underlying scripts should not filter or otherwise transform the available objects:

```
[:in :out | out addAll: in] -> { #view -> PaneView }.
```

The reason is simple: side effects through user interaction. Interactive views typically support the addition of elements via drag-and-drop gestures. Those views have to communicate such change to their owner-panes so that the next model and view update will not discard information. Note that a pane can discard its view at any time as the tool builder tries out different scripts and views. It would be surprising if an accidental view swap would change the set of software artifacts. Consequently, only scripts/models that allow views to infer the set of input objects can enable such kind of updates. While filters are okay because they would just get "materialized" after an update, elaborate object transformations can impede "input inference" and thus discard information.

The following script produces such a list model for some morphs by discriminating their visibility so that different scripts can be applied through the pane view's panes:

```
[:in :out | in do: [:morph | out add: {
   #object -> morph.
   #text -> morph name.
   #script -> (morph visible
                  ifTrue: [#'dfca5b31']
                  ifFalse: [#'4a66468d']) }]].
-> { "... script properties ..." }.
```

Here, the property `#script` discriminates the particular script (or tool or view) to show for an `#object` depending on its visibility.

Actually, pane views work with *lists of lists* of objects, and appropriate scripts have to account for that circumstance. Every pane accepts a list of objects as input, and pane views manage multiple such panes. Thus, a list of panes requires lists of lists of objects to be configured. First, scripts should expect each object in the input buffer to be a list of objects—like lists of *tuples*. Second, each model node should store a list of objects, not single ones, in `#object`. We can apply our `#asList` conversion from before to create robust scripts for pane views like this:

```
[:in :out | in do: [:oneOrManyMorphs | out add: {
   #object -> oneOrManyMorphs asList . "Special for pane views."
   #text -> oneOrManyMorphs printString. "Short compromise."
   #script -> (( oneOrManyMorphs asList allSatisfy: [:m | m visible] )
                 ifTrue: [#'dfca5b31']
                 ifFalse: [#'4a66468d']) }]].
-> { "... script properties ..." }.
```

Such scripts produce view models that might look confusing when applied to regular views. As explained before, the script editor's wizard will employ templates that *flatten* lists of lists of objects when performing one-to-many transformations such as "`morph submorphs`". Regular views are likely to not unpack such lists and treat those as objects of their own, which is valid in the object-oriented environment but not necessarily useful. A list of lists might also be falsely recognized as tuples if the inner lists do all have the same size. Consequently, the tool builder has to deliberately decide for pane views over regular views during script authoring.

Pane views manage configuration hints via script properties. Tool builders can add object-specific hints manually via object properties. If, however, a pane view finds a configuration hint *twice*, then it will use the one from the script properties. Note that views have to store modifications that were triggered through user interaction such as a new background color. We think that it is not feasible for views to rewrite a script's transformation (or extraction) code so that it keeps its semantics. As suggested before, all views should employ a script's *inherent task context* via script properties. Both tool builders and widget/view providers should avoid using *global* configuration state. We propose the following format:

```
[:in :out | "... object properties ..." ] -> {
   #isProperty -> true.
   "Define or override each pane's rectangle/geometry."
   #bounds -> #( (1 0 0 50 100) (2 50 0 50 100) ).
   "Distribute (ordered) model objects among panes with a 'use mode'."
   #input -> #( (1 input) (1 selection) (2 input) (3 input) ).
   "Define object selection as tuples by order and 'provision mode'."
   #output -> #( (2 selection) (3 selection) ).
   "Define or override each pane's script."
   #scripts -> #( (2 #'9ecff5f0') (3 #'e7e3b840') ).
   "Define connections among panes with 'provision mode' and 'use mode'."
   #connections -> #( (1 selection 2 input)(2 selection 3 input) )}.
```

Given such an example of configuration hints, we propose the following *implementation semantics* for pane views:

1. *Respect the order of the input objects.*— We specify object connections via `#input` to distribute incoming objects to panes. We also enumerate panes (`1,2,3,...`) as temporary identifiers to associate connections, scripts, and bounds.
2. *Avoid hints that are complex Smalltalk objects.*— We use only literals for numbers, symbols, strings, and arrays to encode state such as for geometry and object connections. We want to support text-based script authoring as explained in section 4.2. Complex objects would have hardly legible text representations.
3. *Remain flexible in terms of object count and object kind.*— We think that missing input objects should result in ignoring the remaining specs. If there are too much input objects, the pane view should use default values to create and fill panes. We assume robustness during script evaluation and object selection propagation. That is, if views cannot find the objects they should select according to their pane, then they should just ignore the request.

More on Object Connections

Recall that every object connection as *two modes* that specify the kind of objects to exchange: provision mode (for output) and use mode (for input).

A pane will *override* a connection's use mode if a pane's objects are changed for other reasons. Most notably, regular user interaction through views can change the *object selection*, which would discard all incoming connections that try to set the current object selection. In general, any object can send a message to a pane (object) to update its configuration. Programmers can use Squeak's Object Inspector to try different domain objects or scripts. In Squeak/Smalltalk, such direct inspection and manipulation of objects via custom code snippets is considered common practice. As an effect, the pane's respective incoming connections will be discarded. Note that any re-established connection will, again, override the override.

Object connections can create *circular dependencies* among panes. Given our previous thoughts on consistent tool updates, such cycles will render user interfaces unresponsive if not detected. For example, a Fahrenheit-Celsius converter would indefinitely exchange numbers, maybe fluctuating because of rounding issues. For another example, a list-based browser for a hierarchical structure would directly "go down" to the leaf if the first item would be on auto-select and the object selection used as input for *itself*. In general, scripts can hide elaborate transformations. Thus, a comparison via object identity is not sufficient, although same objects implicate similar view models for the same script in our program comprehension setting. (We ignore side effects here.) Instead, we detect update cycles by identifying a *start pane* to create an *update identifier*, which we pass between updating panes. Then, panes can refuse to update twice based on that identifier. Starting panes are either the ones whose current views just handled user interaction or the ones whose current script just triggered a model update via interactive script authoring or script notifiers.

Object connections are *ordered*. Thus, following our Rule of Distinctiveness, programmers can distinguish *focus objects* and context objects *by order*. In a common setting, the object selection of one pane serves as input for another pane like in figure 4.8. When using multiple panes in combination, the use of context implies the use of multiple incoming connections to fill the input buffer. Since there will be tuples created from multiple sources, we suggest treating the *first object* in a tuple as focus object and the rest as context:

```
connectionOne objects = #(1 2). "true"
connectionTwo objects = #(a b c). "true"
input := ( connectionOne objects, connectionTwo objects ) asTuples.
input = #( (1 a) (1 b) (1 c) (2 a) (2 b) (2 c) ). "true"
```

Here, numbers represent focus objects, and letters represent context objects. In practice, method objects can be enriched with run-time objects to describe elaborate transformations or extractions in a script to enrich views on source code.

Object connections can be the *means to integrate* VIVIDE to other GUI frameworks. That is, objects other than panes could make use of it. Systems such as Squeak/Smalltalk are dynamically typed, which means that any object can chose to respond in a meaningful way when spoken to in a certain protocol. In a way, we objectify certain Smalltalk accessors, which are called "getters" and "setters" in other environments, for the purpose of tool building. If tool builders want to *inject objects* from outside into an "ecosystem of panes", they could employ object connections. If they want to *extract objects* to put to a different use in the surrounding environment, they could employ object connections, too. We think that our UI-design language can be integrated in other design languages. In terms of Morphic, any morph can host a set of panes to interact with each other.

Pane Decoration: Menus for Relations and Actions

We apply pane decorations to further modularize the tool's presentation language as proposed in figure 4.5. That is, the presentation should make a clear distinction between *focus object(s)*, *related objects*, and the *available actions* so that users can reliably separate exploration from manipulation. Recall that we already handle focus objects in terms of pane input and view model. To realize a UI for related objects and actions, we propose to

1. support an arbitrary amount of pane decorations and
2. use panes and scripts to configure pane decorations.

Actually, we only need two more decorations to describe the left *menu for objects* (or relations) and the right *menu for actions*. Yet, the tool builder might want to model more interactions at this data-driven level. Pane-decoration panes should receive both the input and view of the to-be-decorated pane to extract appropriate labels and callbacks, which is expected from pop-up menus. A script for action menus as in figure 5.1 can look like this:

```
[:in :out | in do: [:tuple | [:view :objects | out add: {
     #object -> objects. "Merge multiple focus objects."
     #text -> ('Save changes for ', objects printString).
     #selected -> [[ view save ]]. "Trigger side effect." }
  ] value: tuple first value: tuple second ]
] -> {
  #isProperty -> true.
  #view -> ListView. "Menus are like lists."
  #color -> Color gray. "Common menu color." }.
```

Like for pane views, such an approach works best if the input objects basically represent the focus objects. If, for example, focus objects would be wrapped in other (input) objects, the code for unwrapping these would have to be duplicated in pane script and pane-decoration pane script. We think that the choice of pane decoration should be manifested in the script properties of the to-be-decorated pane's script:

```
[:in :out | "... transform and extract ..."]
  -> { #view -> TextView.
       #decoration -> #('2b2ca6c7' 'dabed281') "Script identifiers."}
```

The script identifier look-up is the same as for all referenced scripts in pane views or via `#next`, which we described before. Hints about pane geometry should be stored in the pane-decoration pane script such as whether the menu is on the left or on the right. Consequently, panes do not only have to invoke script interpretation to generate view models. They also have to scan script properties for decoration hints to further configure themselves by adding other panes besides the central view.

Given all these use cases, we propose to treat pane decorations as the *extension point* in our (Morph-based) UI-design language:

$$ui = panes + views + connections + decorations$$

If tool builders want to modify script management or object exchange, decorations can represent those variations in both *state* and *behavior*. With state variation, we mean the addition of *visual adornments* as decoration to change pane state. With behavioral variation, we mean that decorations can also *intercept the events* that trigger consistent tool update and hence influence the pane's behavior. We already experimented with the following additions:

- Animate data flow to support debugging of object connections.
- Switch scripts dynamically based on the kind of input objects.
- Log recently used scripts to manifest local variations in presentation.
- Refine object exchange to ignore empty sources during tuple creation.
- Create a default script automatically for empty organizations.

We argue that such combination of interactive graphics and behavioral variations shortens the feedback loop for tool building. One alternative would be to think in terms of *different kinds of panes* instead of added decorations. Yet, we will not elaborate any further in this regard. We focus on tool builders that use a pane's halo to access and modify pane compositions, scripts, and object connections.

Synopsis Our UI-design language describes tools as compositions of communicating *panes*. Primarily, tool builders use scripts to configure such panes (and their views) through *object properties* and *script properties*. Complementarily, tool builders use *object connections* and *pane decorations* to control panes interactively.

A view's supported object and script properties should be documented so that *script authors* can learn what *view designers* anticipated. We think that views should agree on *common properties* such as `#text` or `#icon` to support view switching. Overall, views should treat scripts (and script properties) as *task context* to *persist* generic configuration state.

Object connections define object exchange across panes and thus scripts. They are ordered and can thus be used to distinguish *focus objects* from *context objects*, which benefits *tangibility* of information. The framework will handle *cycles* to keep the system responsive.

Pane decorations add more configuration layers to single panes. So, decorations can change a pane's *state* such as the current script and additional menus. They can also influence a pane's *behavior* such as its reaction on object arrival.

5.4 Advanced Thoughts on the Working Practice

This section expands our data-driven working practice, which Vivide introduces with its new mechanics. First, we explain the possible dimensions of *re-using scripts* as a tool user or tool builder. Then, we describe ways to handle *interleaving tasks* and task switching, which includes focus recovery and the explication of thoughts on screen. Finally, we address a common deficiency that arises with *repetitive activities* in direct manipulation interfaces by applying existing user-monitoring strategies in a data-driven perspective.

Re-use: From Ad-hoc Prototypes to Accustomed Assistance

A script's transform and extract steps support fine-granular decomposition, which promotes re-use. However, script properties will compromise such re-use if programmers trigger task-specific modifications unintentionally during regular work. For example, pane views that offer a flexible desktop layout have to persist the position of all windows at any time to support re-evaluation of the outer script. We assume that programmers can easily *copy scripts* in exploratory tool-using or tool-building tasks. Yet, we also embrace re-use so that ideas that arise in allegedly unique scenarios have a chance to evolve into common tools that serve the entire project domain or even generic programming practice.

First, we will elaborate on script re-use for tools with *prescribed* presentation languages such as Squeak's browser, inspector, and debugger—which focuses on the source code for transformation and extraction. Then, we will elaborate on script re-use for tools with *flexible* presentation languages such as messy desktops and growing tapes—which focuses on script properties managed by (pane) views. Overall, the question of "Copy or re-use?" can be answered as follows:

	Transform & Extract	**Script Properties**
One Script (for a view)	Copy if sent messages are incompatible	Copy if view changes script for task scope
Many Scripts (in a pane view)	Copy all scripts if one script is incompatible	Copy if user starts new task (and scope)

For a **single script**, applicability depends on whether the domain objects can understand the messages sent during transformation and extraction. Smalltalk messaging positively affects re-use in this regard because actual "types" are dynamic and late-bound. Following our Rule of Similarity, tool builders and artifact providers should establish common representations such as text, icon, color, and sound. For example, if all objects could respond to `#asColor`, scripts that would employ such a message would inherently increase their re-usability. Both one-to-one and one-to-many transformations can benefit from this rule. Scripts that describe tree models can be used with several kinds of objects if they share navigation paths like `#entries` might do for file systems and `#elements` might do for graphics scenes. At this basic level in our UI-design language, tool builders would copy a script and make it fit, and tool users would have to look for another script/tool that fits.

For a **set of scripts**, applicability depends on message comprehension, too, but becomes more challenging for *selective adaptation*. Multiple panes in a (pane-view) context employ multiple scripts and multiple object connections to form the tool's exploration path. If a domain object fails to understand a single step in this path, the entire tool will not be applicable to that kind of domain object. For example, a suspended process cannot be fed into Squeak's code browser but into the debugger—even though both support methods and source code at some point. Tool builders will have to copy the set of scripts to make the adaptations and hence build an extra tool. However, Squeak's meta-object protocol can help write re-usable scripts if tool builders can anticipate variations upfront. For example, an object can be asked whether it understands `#submorphs`, `#nodes`, or `#items` to express a one-to-many transformation. Then, a textual representation can be derived via `#details`, `#name`, `#contents`, or `#printString`. Such accommodation for heterogeneous sets of objects, yet, will become challenging if their protocols overlap considerably.

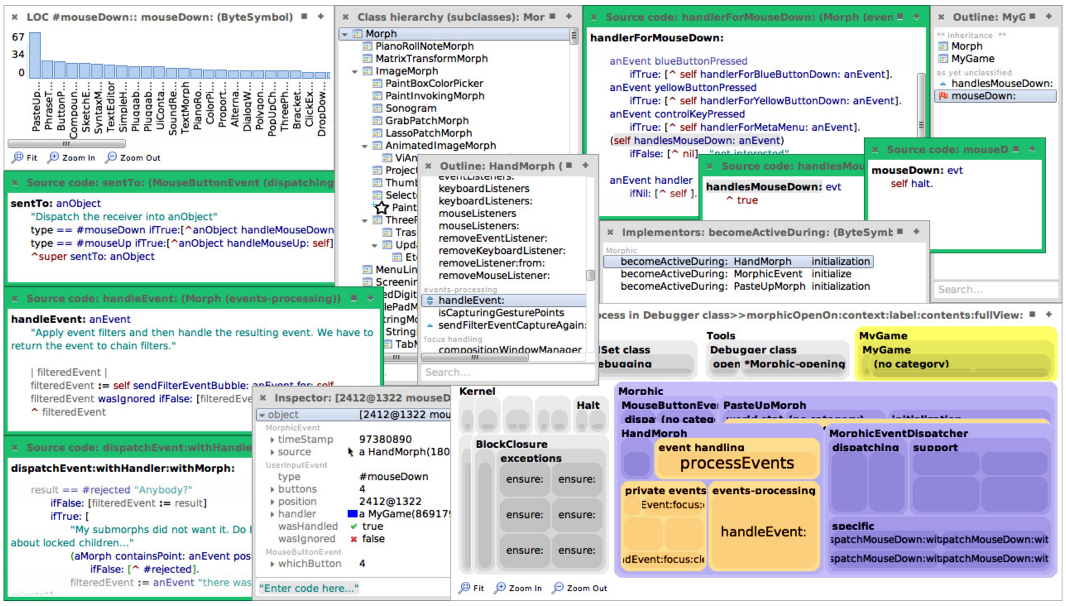
For a **single script**, re-use of *script properties* depends on the user's influence when using the scripted views. As proposed before, views should store model-independent configuration state as script properties because scripts inherently provide task/domain scope and avoid further global state. For a single non-composite view, we think that such configuration is comparable with user preferences for a tool. For example, Squeak's browser is green if windows should be colorful, which is independent from the actual source code being browsed. Consequently, there is no need to copy a script before applying it to some objects. We think that ad-hoc re-configuration of some script properties can be reasonable for both tool user and tool builder. Examples include `#view` for the kind of view and `#color` for the window color.

For a **set of scripts** in pane views, re-use of *script properties* depends on whether the user wants to continue a task or start a new one. Recall that the script of a pane view holds the configuration of all panes in the context. If the user can freely modify panes in that context, script properties will be modified as a side effect. Unlike workspaces in Eclipse or projects in Visual Studio, the user does not have to *save* the current state of the environment's presentation explicitly. Consequently, we think that there should be a notion of *prototypical* scripts for (composite) pane views that, for example, represent "desktops" or "endless tapes". *Using* a desktop then implies *copying* the respective script(s) and start re-configuration by adding panes. If users forget to copy, they will have to clean-up manually later. Squeak's projects [192] are similar in the sense that programmers who forget to open a new project (if they work on a new task) have to *close* tools and remove morphs *manually* when they finished; they cannot just *delete* the project.

How to Cope with Multiple Programming Tasks

Programmers, like all humans, can only perform one conscious activity at a time [122], and daily life usually involves multiple activities that are not back-to-back but interleaving. Still, they can take measures to reduce the overhead of switching, which consists of the two phases *suspension* and *resumption* [8, 143]. During program comprehension, tasks are often blocked and suspended due to information needs [91], which spawn sub-tasks to be completed first, which can again spawn sub-tasks and so on. Now, programmers can mitigate interruption lags by externalizing *task context* as informational *cues* [101, 8], preferably on screen. However, there is a trade-off between the costly *serialization* of thoughts and the limited *cue priming* via tools in the programming environment [143]. We argue that a programming environment can promote task switching if it gives programmers control over all information on screen. Eventually, serialization can be a by-product of cue priming, which we refer to as on-the-fly *tool building* in Vivide.

In our Vivide environment, programmers have precise control over the visual representations of software artifacts, which supports task management as depicted in figure 5.5. We distinguish *explicit* and *implicit* definition of task boundaries to



Source code: sentTo: (MouseButtonEvent (dispatching

```
sentTo: anObject
    "Dispatch the receiver into anObject"
    type == #mouseDown ifTrue:[^anObject handleMouseDown
    type == #mouseUp ifTrue:[^anObject handleMouseUp: self]
    ^super sentTo: anObject
```

Source code: handleEvent: (Morph (events-processing))

```
handleEvent: anEvent
    "Apply event filters and then handle the resulting event. We have to
return the event to chain filters."

    | filteredEvent |
    filteredEvent := self sendFilterEventBubble: anEvent for: self
    filteredEvent wasIgnored ifFalse: [filteredEve
    ^ filteredEvent
```

Source code: dispatchEvent:withHandle

```
dispatchEvent:withHandler:withMorph:

    result == #rejected "Anybody?"
        ifFalse: [filteredEvent := result]
        ifTrue: [
            "My submorphs did not want it. Do I
about locked children..."
            (aMorph containsPoint: anEvent pos
                ifFalse: [^ #rejected].
            filteredEvent := anEvent "there was
```

Source code: handlerForMouseDown: (Morph (even

```
handlerForMouseDown:

    anEvent blueButtonPressed
        ifTrue: [^ self handlerForBlueButtonDown: anEvent].
    anEvent yellowButtonPressed
        ifTrue: [^ self handlerForYellowButtonDown: anEvent].
    anEvent controlKeyPressed
        ifTrue: [^ self handlerForMetaMenu: anEvent].
    (self handlesMouseDown: anEvent)
        ifFalse: [^ nil]. "not interested"

    anEvent handler
        ifNil: [^ self ].
```

Source code: handlesMou

```
handlesMouseDown: evt
    ^ true
```

Source code: mouseD

```
mouseDown: evt
    self halt.
```

```
** inheritance **
Morph
MyGame
as yet unclassified
handlesMouseDown:
mouseDown:
```

```
keyboardListeners
keyboardListeners:
mouseListeners
mouseListeners:
removeEventListener:
removeKeyboardListener:
removeListener:from:
removeMouseListener:
events-processing
handleEvent:
isCapturingGesturePoints
sendFilterEventCaptureAgain:
focus handling
compositionWindowManager
```

Implementors: becomeActiveDuring: (ByteSymb

```
Morphic
becomeActiveDuring:  HandMorph     initialization
becomeActiveDuring:  MorphicEvent  initialize
becomeActiveDuring:  PasteUpMorph  initialization
```

Inspector: [2412@1322 mouseD

```
object          [2412@1322 mou
MorphicEvent
  timeStamp     97380890
  source        a HandMorph(180
UserInputEvent
  type          #mouseDown
  buttons       4
  position      2412@1322
  handler       a MyGame(86917
  wasHandled    ✓ true
  wasIgnored    ✗ false
MouseButtonEvent
  whichButton   4
"Enter code here..."
```

ocess in Debugger class>>morphicOpenOn:context:label:contents:fullView:

Figure 5.5: Programmers can control the visual representation of artifact structure to orchestrate multiple programming tasks. They can use distinctive views or colorful windows can help recognize concerns.

separate deliberate configuration/building efforts from (presumably) unanticipated side effects. First, programmers can make explicit use of geometry and layout to cluster artifacts. This largely means pane distribution across pane views, which are augmented with dependencies through object connections. Second, programmers can benefit from implicit effects in panes and pane views based on the artifacts they carry. For example, a pane-view's layout strategy might open new panes beside the requestor to document origin. For another example, pane decorations might automatically reveal object properties such as `#color` if the environment knows about it—which we proposed as the Rule of Similarity in section 4.2. Nevertheless, programmers can always *serialize* additional thoughts by creating helper objects such as photographs, texts, or voice memos in the environment.

Low-effort tool building that happens ad-hoc can positively affect task interruption. We argue that if programmers can find fitting, tangible representations for task-related artifacts, then task suspension can be basically "for free" without increasing the resumption lag. Any additional effort to reduce cognitive load and explicate the mental model should have been done in the course of regular program comprehension. If programmers favor distinctiveness and prominence over generality and richness, they implicitly spread discoverable cues that help remember task context and progress after the interruption.

In Vivide, task resumption is the phase where programmers figure out *what* artifacts are *where* and *why* on screen. That is for each view, they have to recall the *focus object* to then associate the task behind it. When interpreting screen contents, programmers do not just passively look at labels, colors, or shapes. They actively invoke pane halos to inspect input buffers and object selections. They also trace object connections to recover task context. Hence, the same mechanics can be used for tool building and task resumption without further ado. We think that this also affects task reminding [122], where noticeable signals and proper descriptions have to be designed.

Remark Any impulsive assessment aside, we consider the interplay of overlapping, graphical artifact views on screen as *intentional clutter*. Due to our inability to capture momentum in a picture such as figure 5.5, we cannot, in retrospect, fathom why any programmer chose to densely pack information side by side. There can be system and order in such a "creative mess". We think that this is each programmer's personal attitude to express. We want to underline the *flexibility* our environment provides. On closer inspection, there so no observable redundancy in figure 5.5, which means no duplicate class lists or other views. We argue that such a result is better than the average scenario that emerges in traditional windowing systems such as illustrated in our debugging scenario in section 3.1 (figure 3.1).

User Monitoring for Automated User Guidance

VIVIDE provides means to integrate *existing* data with *existing* views to form tangible representations that fit particular domains, tasks, and user preferences. In such an environment, there are likely to be many existing scripts, too, which were prepared for generic tasks or left over from previous projects. Now, a major challenge programmers will continually face is the one of *choosing*. Even though our scripting language and UI-design language consist of few concepts only, they can span a large decision tree when applied to artifacts from an actual problem domain:

- Which object(s) to choose to start or continue exploration?
- Which script(s) to choose to start or continue exploration?
- Which view to choose to best understand the available artifact structure?

Mistakes force the user to backtrack and retry, which takes time. There are also more advanced choices of whether to configure the environment or start building a new tool:

- When to consolidate or discard open panes (or windows) to handle limited screen space?
- When to author scripts to improve repetitive or complex activities, which is a common problem in such direct manipulation interfaces [75]?

We argue that a data-driven working practice should be complemented by an environment that constantly monitors its users to make suggestions in a data-driven way. That is, the environment can observe and record the programmer's past decisions to present them *in situ* through visual cues. For traditional programming environments, there have been efforts to study the effect of user monitoring to log (code) navigation histories that capture task-specific concerns [86, 141, 164, 121]. They share a focus on software artifacts and the goal to simplify access to wide-spread information for future tasks that affect similar parts in the system.

User monitoring in Vivide means tracing the essential steps from the programmer's exploration path. Since we see program comprehension as repeated fetching, examining, and retaining of software artifacts, such traces should contain respective information. *Given some artifacts' structure A, how does the user access artifacts' structure B?* We see a trade-off between level of detail, prospective utility, and resource limitations. There will be a compromise to store only "meta" information of the involved artifacts that is easily accessible and likely to support the user's cognition. In Squeak, for example, the object's class or package might indicate the applicability of a code snippet (or script) for future tasks. In Vivide, for example, there is also context in terms of pane views, which provide access to neighboring objects and spatial information. Thus, we argue that *effective tracing* of the programmer's exploration path is feasible because of our data-driven perspective.

Recorded exploration paths can help programmers choose from lists of objects, scripts, and views. In particular, they can drive and automate many of the aforementioned activities: selection, sorting, evaluation, composition, and removal/filter. For example, a programmer who is repeatedly interested in a morph's color can be automatically pointed to an existing script that extracts such a color object. If a programmer repeatedly navigates internal morph structure (such as `Morph` ⟶ `MorphExtension` ⟶ `LayoutPolicy`), a script can be derived that combines these repetitive steps. Also, any short-living view could be detected and closed, or put into background, automatically to tidy up screen space. In general, programmers (or tool builders) should have access to their monitoring data in the form of new artifacts to be used in scripts and hence fully integrated into the Vivide workflow.

Any two compatible scripts can be *merged* by concatenating their transform steps and extract steps. The resulting model tree will have more levels, which can be folded by re-writing the extract steps. Recall that extract steps are basically transform steps but describe additional properties for the tree node to be accessed by views. The same approach can be used to *split* any larger script so that the programmer can assign different views to each level in the tree. Object connections can then re-establish the tree structure across multiple panes.

If the monitoring includes Smalltalk's typical *do-it expressions*, new scripts can be derived directly. That is, messages that do not have side effects are safe to put into transform steps. For example, the following code snippet from a Smalltalk workspace fits our scripting language:

```
morph := Morph new. "Do it."
extension := morph extension. "Do it."
policy := extension layoutPolicy. "Do it."
policy hResizing. "Print it: #spaceFill"
```

We assume that the programmer evaluated those four expressions in order. Then, we can construct transform and extract steps from all traced do-its and print-its:

```
script := {
   "First, transform morphs into their layout policies."
   [:in :out | in do: [:morph |
      out addAll: morph extension ]].
   [:in :out | in do: [:extension |
      out addAll: extension layoutPolicy ]].
   "Second, extract the horizontal resizing behavior."
   [:in :out | in do: [:policy |
      out add: { #object -> policy.
                 #text -> policy hResizing }]]
     -> { #isProperty -> true}.
} asScript.
```

The idea is that expressions on *single objects* can be applied to *multiple objects* so that programmers can better understand larger information spaces. Note that this very code snippet can be embedded into a script that creates scripts on-the-fly, which might be derived from user monitoring data. In Squeak/Smalltalk, everything is an object. If the list of available scripts is created with a script itself, then programmers can integrate such additional information easily.

Synopsis Given such short feedback loops in Vivide, we encounter *inadvertent modifications* with advice on when to *copy scripts* and when to just *apply scripts* with objects to open tools.

Complicated information needs in program comprehension tasks are likely to spawn *interleaving* (sub-) tasks. If programmers can manage to put tangible representations of relevant artifact structure on screen, *task switching* will be supported because existing visual cues make *task suspension* cheap.

Finally, all Vivide interactions could be monitored and the traces again be provided in the course of tool configuration and tool building. Then, programmers could learn to optimize their interactions by actively reflecting on their data-driven working practice.

5.5 Implementation Guidelines

We only sketch the details of our Vivide implementation in Squeak 5.1 that could serve as guidelines when transferring our solution to other environments. Since we build on the concepts of existing interactive, self-supporting systems, there is more literature on this topic. To some extent, the foundation of Squeak is reasonably-well documented [77, 76], which includes the Morphic graphics framework [110, 108]. There are some script-based applications that relate to our implementation strategy such as Etoys [7] and Scratch [109]. We did also document the structure of *Squeak's code artifacts* in the scope of user-interface frameworks [192] to highlight possibilities in such an environment. Overall, keep in mind that everything is an object and every graphical element (or morph) is inherently tangible through its halo (meta) menu.

In such purely object-oriented system, we have to design (1) *structure* borne by objects and (2) *behavior* borne by messaging. In Squeak, we also use *classes* and inheritance to facilitate code re-use. Our goal is to design a tool framework whose applications, the tools, are changeable at run-time as proposed in the previous sections. That is in a tool-building scenario, tools should remain consistent and operational without a need to restart. Thus, framework behavior is defined by the rules that *return* all (tool) objects into a consistent, stable state *after* any user or builder[5] action. To achieve this goal, we refer to a common catalog of design patterns [62] and apply creational, structural, and behavioral patterns:

Composite [62, pp. 163–173] to define hierarchies for data (i.e. model nodes) and graphics (i.e. panes, views, and object connections)

Interpreter [62, pp. 243–255] and Builder [62, pp. 97–106] to construct the tree model from a script and some input artifacts

Observer [62, pp. 293–303] to watch for modifications in scripts, artifacts, model nodes, pane compositions, and view descriptions

Adapter [62, pp. 139–150] to make existing views (or visualizations) compatible with our framework

Adapter [62, pp. 139–150] and Bridge [62, pp. 151–161] and Facade [62, pp. 185–193] and Strategy [62, pp. 315–323] and Composite [62, pp. 163–173] to narrow down the flexible applications of scripts in the environment

We think that scripts in a dynamic system such as Squeak capture structural and behavioral patterns in a rather indistinguishable way. Their dynamic properties blend interface adaptation and algorithm encapsulation. Especially the automatically generated model, which we regard as a script's manifestation, shows characteristics of the bridge pattern from the view's perspective. We will not explore these subtleties any further.

[5]Note that we use the term "builder" (short for "tool builder") to address the programmer's role and "builder pattern" to refer to the design pattern [62] unless denoted else.

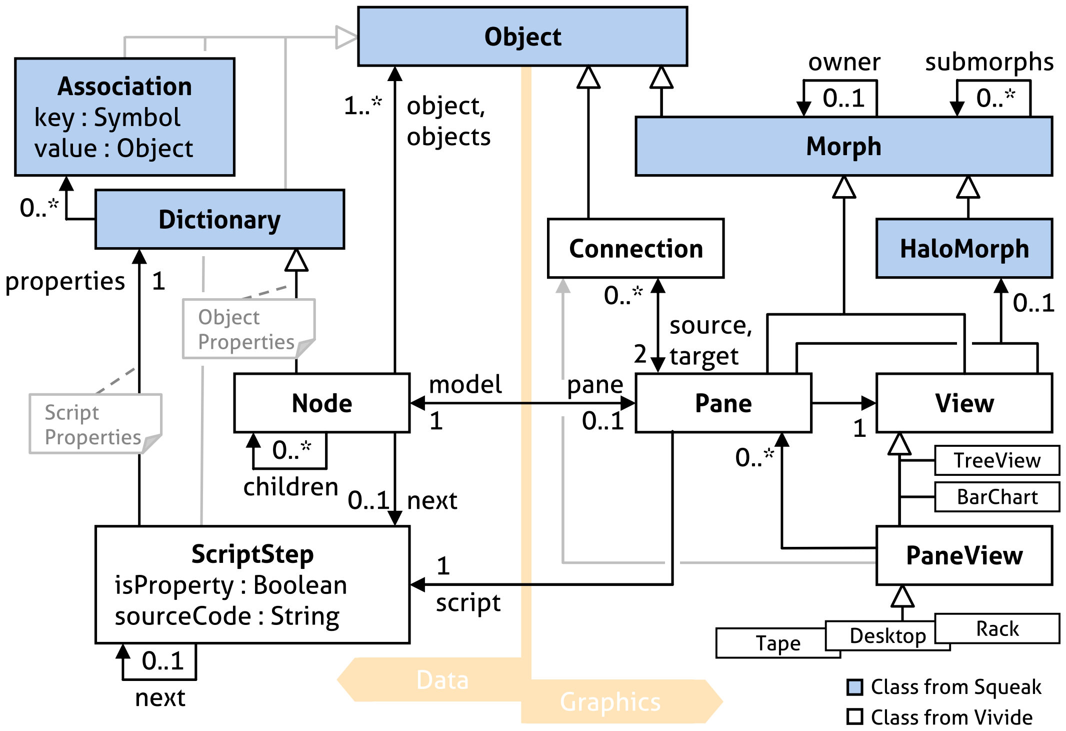


Figure 5.6: The major parts in our framework are implemented as dictionaries, morphs, or plain objects. The data side is represented as scripts and nodes. The graphics side is represented as panes, views, and connections. Note that model nodes *are* dictionaries and script steps *use* dictionaries.

We selected the aspects we think are most relevant for our framework. That is, we explain the class structure, object lifetime, script interpretation, and object exchange between panes. Then, we move from tool using to tool building, which covers (source) modifications of scripts, data, and views. Finally, we explain the integration of VIVIDE tools into the Squeak environment.

Class Structure and Object Lifetime

In figure 5.6, we show how VIVIDE uses composition and generalization to represent our scripting language (section 4.1) and our UI-design language (section 4.3). This Class Diagram [208, pp. 99–103] focus on Squeak-versus-VIVIDE on the vertical axis and Data-versus-Graphics on the horizontal axis, omitting many details from the actual implementation. There are many design decisions that we just present *as-is*, knowing that they might not be the best possible trade-offs.

Scripts are instances of the class `ScriptStep`. The text representation of in-out blocks (`[:in :out | ...]`) is stored in `sourceCode`. A compiled version is cached, but the text has more information such as indentation and comments. Scripts hold an `isProperty` flag to distinguish transform steps from extract steps during interpretation. Other properties are stored in `properties`, which is an instance of the class `Dictionary`. Steps are concatenated via `next` to describe model trees through a series of transformations and extractions. Note that script properties can also hold a `next` key to refer to another script (identifier) to facilitate script re-use and abstraction. Also note that in this linked list, every step can be a script on its own. There is no encompassing container for script steps.

Models are instances of the class `Node`. Each node is a `Dictionary` to store the extracted object properties directly. Yet, there are several reserved keys: `object`, `children`, `objects`, `next` (script), and `pane`. Except for the model's *root node*, each node is thus a wrapper for a single object. Each node is prepared to trigger the (lazy) computation of its child nodes if requested. The optional reference to `objects` can be used to carry over objects between tree levels such as when using *tuples* to form groups. For script interpretation with (`next`) references, nodes hold also the current *script stack* to keep track of data flow between tree levels.

The tool's graphical interface is composed of instances of the classes `Pane`, `View`, and `Connection`. Panes and views are morphs, which form the graphical hierarchy in Squeak via `submorphs` and `owner`. Specialized views such as `TreeView` use our generated model to create other morphs such as `StringMorph` and `ImageMorph` to form their visual appearance. Connections represent indirect, objectified references between panes to attach *use mode* and *provision mode* for configurable object exchange.

Layout reification happens through instances of the specialized class `PaneView` supported by `Pane` and `Connection`. Typically, we use a `Desktop` with overlapping windows at the topmost level. Inner containers such as `Tape` can provide different exploration strategies per window. Connections between a pane view and its (child) panes are necessary because panes should not treat object exchange to or from their outer container differently. Thus, pane views have to mimic the portion of the pane interface that handles connections.

[|]

The lifetime of VIVIDE objects, especially `ScriptStep`, depends on (1) organizations and (2) their visibility via pane views. Recall that organizations form namespaces for scripts to be looked up during script interpretation. Each pane view has its own organization, which implies a hierarchy of organizations parallel to the graphical one. (There is also one global organization.) Given that panes are decorated as windows, closing a window will discard (inner) scripts if that window held a pane view. Yet, the script *representing* the (closed) window (or pane) itself does still exist in the organization of the outer (pane-view or global) organization.

We did try to attach a new (temporary) organization to *each* window to foster experimentation. Successful experiments could then be registered in the global organization. Unfortunately, this made it impossible to share scripts between tool windows without installing them globally. Thus, we opted for pane views as visual representations of task context. We think that the graphical hierarchy is a reliable indicator for users to manage scripts. If they look for scripts, they can access the outer pane-view's organization or the global one. We log recently used script objects in a *history* to recover from accidental window closing.

Instances of `Node` and `View` have a shorter lifetime compared to scripts and (domain) artifacts. During exploration and modification, tool builders (and users) choose from existing artifacts, scripts, and view classes as explained in section 4.4. Automatically, the Vivide framework will create and dismiss nodes and views to yield a tangible representation of the respective artifacts on screen. The domain artifacts that serve as script input have been there before and will be there after script interpretation. Only the ones created afresh in scripts are coupled to the node's lifetime if not persisted elsewhere.

Script Interpretation for Model Construction

→ *This explanation complements the passage in section 4.1 with the same title.*

Our block-based scripting language is separate from but closely related to Smalltalk and objects. We use Squeak's parser and compiler to create helper objects, which we then use to build instances of `ScriptStep`. So, the *compact notation* we designed for script objects can be integrated in interactive tools such as our script editor. Recall the following example from before, where we described a hierarchy of morphs:

```
script := {
   [:in :out | in do: [:morph | out addAll: morph submorphs]]
        -> { #id -> #submorphs.
             #label -> 'Morph Hierarchy' }.
   [:in :out | in do: [:morph |
      out add: { #object -> morph. #name -> morph name }]]
        -> { #id -> #'01f88aa9-d4e8-004e-a001-1148a2c36e5f'.
             #isProperty -> true.
             #next -> #submorphs }.
} asScript.
```

Here, Squeak provides object arrays (`{}`), block closures (`[]`), and associations (`->`) as vehicle for scripts. Now, scripts become *effective* when they are used with objects to create model nodes. One could call this phase "interpretation" from a text-oriented perspective or "building" from a object-oriented one. Consequently, both interpreter [62, pp. 243–255] and builder [62, pp. 97–106] patterns can guide the implementation. The structure of Vivide scripts is generic, which makes it possible to use languages other than Smalltalk to process objects. The following listing in EBNF [3] highlights the Smalltalk-specific portion in our current implementation:

```
script = script step , { script step } ;
script step = ( transform step | extract step ) , script properties ;

transform step = smalltalk two argument block ;
extract step = smalltalk two argument block ;

script properties = [ script reference ] , script identifier ,
                    smalltalk dictionary ;
script reference = script identifier ;
script identifier = smalltalk symbol ;
```

Each script consists of one or more steps, each step is either a transformation block or an extraction block followed by script properties, interpretation suspends after extraction, script properties can include a reference, references are script identifiers, identifiers are Smalltalk symbols, properties are, basically, dictionaries.

Model nodes are self-sufficient to compute their children if requested from views. The (invisible) root node, called *model* in figure 5.6, has a list of objects, the first script step, and an empty script stack. Each inner node in the model has an object, a next script step, and a script stack of any size. If views request a node's children, that node instantiates a `ScriptInterpreter`, sets itself as context, and invokes interpretation, which happens as follows:

```
| stack step input output suspend |
stack := node stack copy. "... to support node reset"
step := node next.
suspend := false. "... after extraction"

input := node objects ifEmpty: [{node object}].
output := OrderedCollection new.

step ifNil: [^ #()]. "No next step, no child nodes."
input ifEmpty: [^ #()]. "No input objects, no child nodes."

[

"Actual interpretation of the scripting language."
step sourceCode compile value: input value: output.
suspend := step isProperty.

"Check and follow referenced script step."
(step properties includesKey: #next)
  ifTrue: [
    stack push: step.
    step := (step properties at: #next) asScript. "Lookup" ]
  ifFalse: [
    step := step next].

"Unwind stack if at script end."
[step isNil and: [stack isEmpty not]]
  whileTrue: [step := stack pop next].

"Suspend if at end or *after* property extraction."
] doWhileTrue: [step notNil and: [suspend not]].

"Build and return child nodes."
^ output collect: [:extractedProperties |
  (Node newFrom: extractedProperties) "cf. dictionary creation"
    stack: stack; "... to be self−sufficient"
    next: step; "... to be self−sufficient"
    yourself]
```

Given the Smalltalk language semantics and the Squeak system, this implementation is straightforward: evaluate blocks with input objects, follow script references, suspend after extractions, create model nodes. Note that this is a shorter version of our actual script interpreter because there is no support for a *default* extract step or *merging* of consecutive extract steps. Also, the script lookup is hidden behind `#asScript` sent to a symbol, which omits details about script-organization access. For such lookup, model nodes have access to their pane and thus also to organizations in (outer) pane views as depicted in figure 5.6.

Model nodes can store *rules to access* object properties instead of the extracted values. These rules are regular Smalltalk blocks. Recall that, in scripts, writable properties are defined with two arrows like in the following example:

```
#color -> [morph color]
        <- [:newColor | morph color: newColor]
```

Due to Smalltalk's precedence rules, this expression is, technically, an association from a block to an association from a symbol to a block: `[] -> (#c -> [])`. The different arrow notation (<-), the line break, and the indentation is just syntactic sugar. During script interpretation, this construct will be replaced with a single-block version to be stored in the node, which would be at the `#color` key in this example:

```
[:object :set | set
   ifFalse: [ [morph color] value]]
   ifTrue: [ [:newColor | morph color: newColor] value: object]
```

Whenever views read or write properties in such a node, the node will transparently derive the arguments `object` and `set` from the call `#at:ifAbsent:` (read) or `#at:put:` (write) to evaluate the block to fulfill the request.

In our implementation, programmers can configure the kind of `Collection` to be used as `:in` and `:out` in the transform block. The script properties `#in` and `#out` will be considered by the interpreter. Squeak earmarks object creation using another object via `#newFrom:`, which is implemented for all specializations of `Collection` such as `Set` and `SortedCollection`. So, we can directly use messaging and the class objects stored in those script properties to convert the results between steps. We send an extra `#value` to support user-defined configuration via blocks. The initialization of `input` and `output` from above now looks as follows:

```
input := (step properties at: #in ifAbsent: [OrderedCollection])
            value "for blocks" in: [:classOrObject |
              classOrObject isBehavior
                 ifTrue: [classOrObject newFrom: input]
                 ifFalse: [classOrObject addAll: input].
output := (step properties at: #out ifAbsent: [OrderedCollection])
            value "for blocks" in: [:classOrObject |
              classOrObject isBehavior
                 ifTrue: [classOrObject new ]
                 ifFalse: [classOrObject].
```

Note that we did not implement support for streams, yet. They typically entail a sense of concurrent script interpretation and piecewise computation of node children. A minor adaptation to the interpreter and nodes should be sufficient to hide such concurrency from view code. Panes could indicate progress automatically in the UI, overlaying views.

Views in Context Exchange Artifacts

→ *This explanation complements the passage in section 4.3 with the same title.*

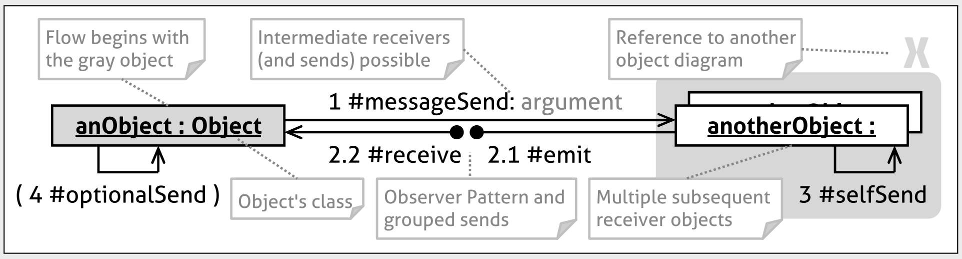
Considering the interactive, visual portion of our framework, we have to design for spontaneous, decoupled updates, which can be triggered through user interaction or other events. This raises the question of concurrency and synchronization.

Our Morph-based UI-design language builds on Morphic's inherent tangibility and extends it with explicit object (or artifact) exchange between views. Usually, Squeak runs the major portion of code in a single *UI process* [192], which avoids process synchronization. This includes *do-its* and other user interactions in the Smalltalk system. Whenever applications trigger sophisticated updates, such as paint requests, there are mechanisms to *avoid redundant work*. For example, the `DamageRecorder` logs and merges "dirty" rectangles, which are then used to schedule morphs to paint on screen in the next UI cycle. In a similar way, we use Morphic's *deferred UI messages* to handle redundancy. Such deferred "messages" are *block closures* to synchronize updates into the UI process. We apply the following pattern to trigger, try, and do updates:

```
triggerUpdate
   flag := true.
   Project current addDeferredUIMessage: [self tryUpdate].

tryUpdate
   flag ifFalse: [^ self].
   flag := false.
   self doUpdate.

doUpdate
   "Do the actual update."
```

This pattern provides flexibility for updating several nodes, panes, and views synchronously in the same UI process. Note that deferred UI messages are evaluated *in order*, which makes the `flag` effective for a particular sender. We can perform immediate updates with *do** and everything else with *trigger** messages. For longer chain reactions, it can take several UI cycles until the tools are in a stable state again. In the following, we explain how artifacts are exchanged between views when a user clicks on an item in a list view. (The same events happen when a programmer adds or removes object connections between panes during tool building.)

Remark on Communication Diagrams We show interaction between objects using an adapted version of UML Communication Diagrams [208, pp. 599–601] to accommodate the Smalltalk language and Squeak system. On a meta level, our diagrams look like this:


Flow begins with the gray object
Intermediate receivers (and sends) possible
Reference to another object diagram
X
1 #messageSend: argument
anObject : Object
anotherObject :
2.2 #receive
2.1 #emit
(4 #optionalSend)
Object's class
Observer Pattern and grouped sends
Multiple subsequent receiver objects
3 #selfSend

Boxes represent instances of classes, and *arrows* represent message sends. Class names are usually omitted because the instance name is descriptive. We use *annotations* to clarify further details. In addition to these common elements, we extend our Communication Diagrams as follows:

- The object represented as a *gray box* starts the communication flow, which is usually in the upper left corner.
- Each message send can be implemented as or followed by or preceded by *other (hidden) message sends*. The diagram shows the most relevant communication.
- If the sender broadcasts the message to (unknown) receivers, *two black circles* split the arrow visually. An example is the Observer Pattern [62, pp. 293–303].
- Messages can be grouped via *hierarchical numbers* to improve readability.
- If one communication can happen multiple times involving multiple senders or receivers, the respective object boxes appear *stacked*.
- *Gray background highlights*, accompanied by a gray, bold letter, represent references to other diagrams with more details.

In Squeak/Morphic, any user interaction begins in `theHandMorph`, which processes input events from the operating system delivered through the virtual machine. All mouse clicks or key strokes are converted into objects and then dispatched to the front-most morph at the mouse-cursor position. In figure 5.7, we depict an exemplary communication path for a mouse click in a (Vivide) list view. Our framework expects views to *yield* objects after such interaction. Considering the user's exploration task, this usually represents a *selection* of objects to explore further. The view's pane then talks to its connections, which apply their use mode and provision mode to deliver new objects to other panes. The arrival of new objects in a pane is *deferred* for one UI cycle because another connection might trigger for the same pane immediately afterwards. If the length of each incoming connection path varies, the subsequent UI cycles can still update the same panes repeatedly.

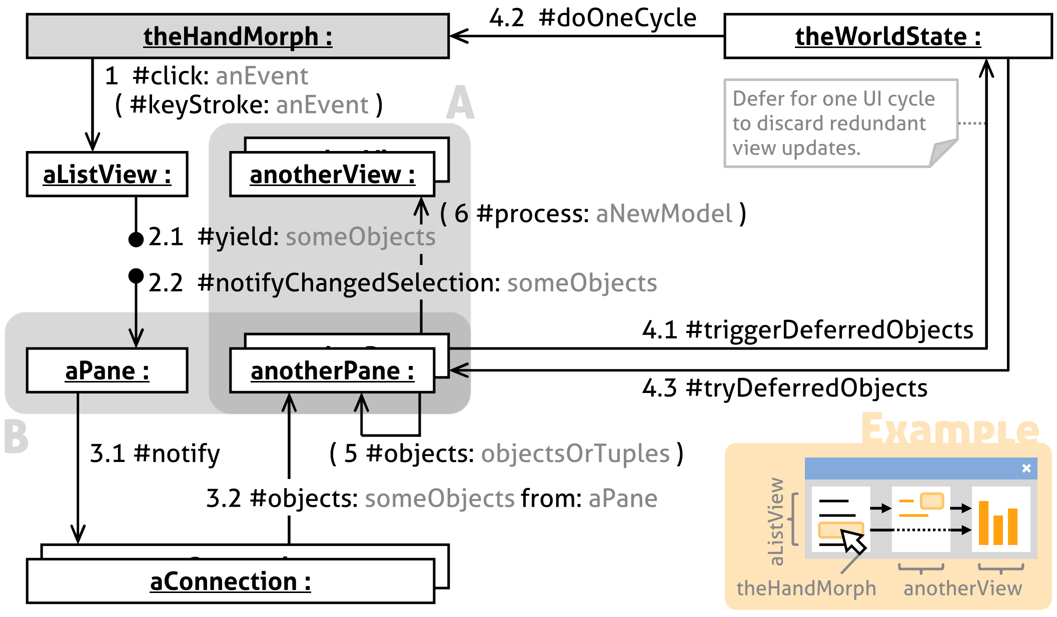


Figure 5.7: When a user interacts with a view, the view can yield objects, for example, to signal a selection change. All outgoing pane connections carry that notification to target panes, which then trigger an update request. The actual update is carried out in the next UI cycle. (See figure 5.8 for A and figure 5.9 for B.)

Panes *reset* their current model if new objects arrive as depicted in figure 5.8. The root node, which *is* the model for a pane, discards its children and receives a new list of objects to work with.[6] This underlines the self-sufficiency of model nodes, which are now prepared to compute new children; views do not notice this laziness. This also shows that Vivide does not re-use models across views at the same time; scripts are the unit of re-use. Now, views get notified about the new model, and they can begin accessing children to construct their submorphs. For example, they could access the `#icon` property in a node to get a `Form` instance to create an `ImageMorph` instance. Note that our architecture does not improve any `#hasChildren` request but performs child-node generation for such information.

There is no governing object that watches over pane/model updates. User-input events occur, panes react, objects get exchanged, other panes react, and so on. Such an update cycle might be *endless* because object connections can combine *any* two panes in the environment. In figure 5.9, we illustrate our approach to detect such cases. The idea is simple: define the begin of an update chain, generate a token, pass on that token, refuse to update if a certain token has been seen before. In Squeak, we can access the currently processed user-input event as a dynamically scoped variable from anywhere. This event is our token.[7] We rely on the event's high-resolution timestamp.

[6]Panes can use a different script if the arriving objects are not compatible with the current script.
[7]Model nodes can begin updates using any generated identifier as token.

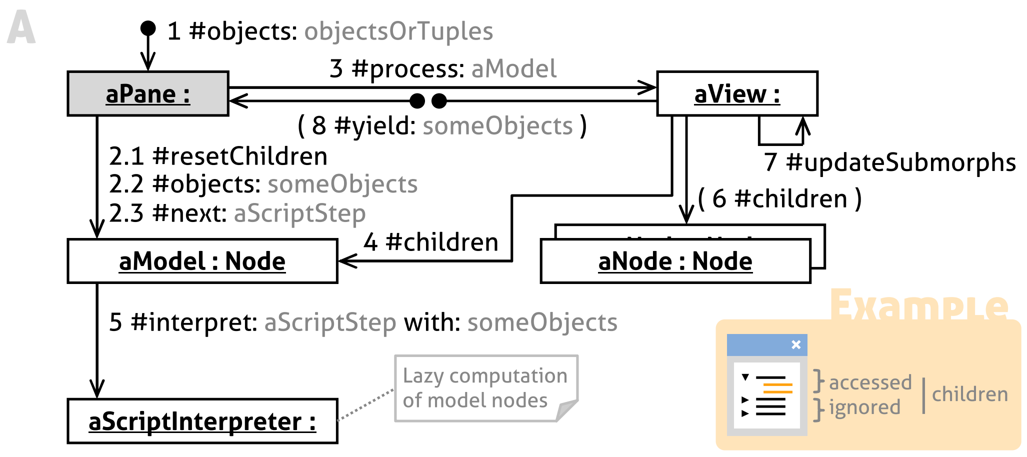


Figure 5.8: New incoming objects for a pane result in a complete reset of the model. Views will access (at least) top-level children again, which triggers script interpretation. Views can tell panes about restored selections.

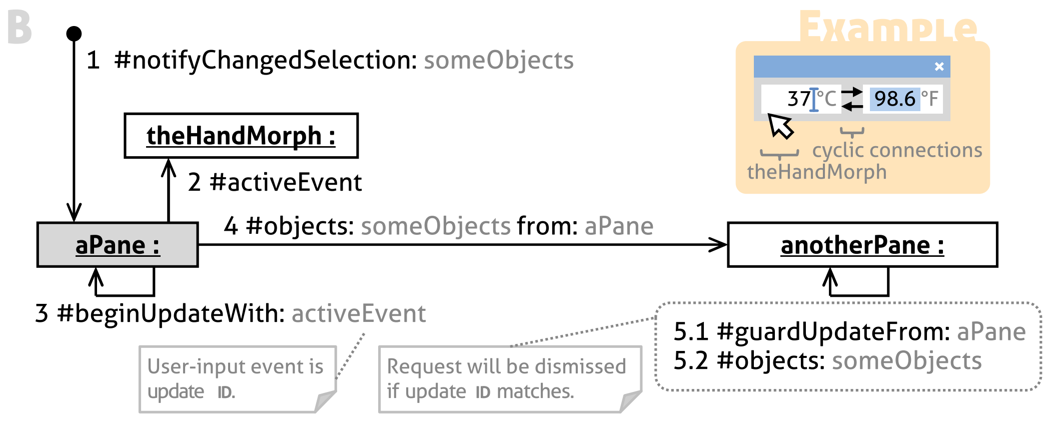

B
1 #notifyChangedSelection: someObjects
Example
37 °C
98.6 °F
cyclic connections
theHandMorph
theHandMorph :
2 #activeEvent
aPane :
4 #objects: someObjects from: aPane
anotherPane :
3 #beginUpdateWith: activeEvent
5.1 #guardUpdateFrom: aPane
5.2 #objects: someObjects
User-input event is update ID.
Request will be dismissed if update ID matches.

Figure 5.9: Detection and prevention of endless updates: we use the current user-input event as a token, which we pass on between panes. Panes accept any particular token only once.

Change and Update: Script Code, Artifact Data, View Code

In our tool-building scenario, programmers modify script code to form tangible representations of software artifacts. These artifacts can change while being observed. View code might also be affected in such try outs. In the following, we elaborate on the communication that such change events entail. Note that any object exchange between panes happens as described before.

Our interactive script editor (figure 5.1) can change script code and script properties. As shown in figure 5.10, the script step, which is the modified object, then informs the singular script-change notifier. That notifier reifies the request as an *event object* and then delivers it to all existing nodes[8] and panes. Consequently, applications other than script editors can modify steps, and our framework will coherently update all affected tools. In nodes and panes, the actual update will be deferred for one UI cycle to support multiple, synchronous changes to scripts. For example, several transform steps can map to the same set of child nodes, which requires only a single re-computation. Besides code editing, the following changes will also reset a node's children: (1) change a step's next[9] step, (2) change the `#next` property, (3) change the `#in` or `#out` properties. Changes in other script properties such as `#view` do typically affect views directly, which panes communicate through `#setUp:`.

If the observed artifacts change, scripts should be re-interpreted and model nodes be re-computed. For this purpose, our framework provides *notifiers*, which are observer-adapters to integrate existing event sources. We assume that artifacts can tell about their changes in a decoupled manner, which usually entails a publish-subscribe or observer implementation. For example, Squeak 5.1 has `#changed:`/`#update:` (since MVC), `#when:send:to:` (called *object events*), or code-specific `#notify:of...` in the `SystemChangeNotifier`. Our notifiers use the granularity of script steps and input objects to translate "artifact A changed" via "step B needs re-interpretation" to "node C might have different children." Consequently, we anticipate *many short steps* that each do one kind of transformation. The specification of a notifier as a script property looks like this:

```
#notifier -> [:object | ObjectNotifier for: object ].
```

Vivide evaluates the block behind `#notifier` repeatedly for each incoming object.[10] So, programmers can use any Smalltalk expression (here: gray box) to create and configure notifiers. Any object can be a notifier as long as it responds to the messages `#subscribe` and `#unsubscribe`. After the initial computation of a node's children, the node will create and subscribe all such notifiers. Further re-computations will unsubscribe existing notifiers first. Again, the node is *self-sufficient*. When a notifier

[8]For performance reasons, we only register the first *n* levels of the model tree to listen for changes.

[9]In figure 5.6, we chose to call the link between two script steps "next". This is different from the script property `#next`, which denotes a reference to another script.

[10]If the incoming object is a *tuple*, programmers freely choose, which object(s) in the tuple to consider. Tuples are just instances of `Array`.

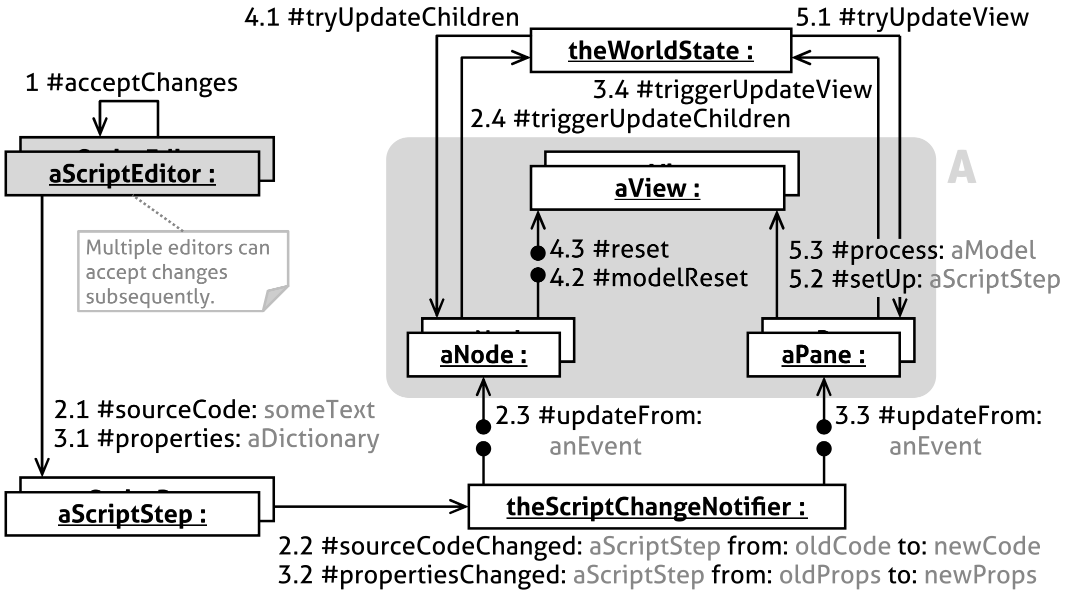

4.1 #tryUpdateChildren
5.1 #tryUpdateView
theWorldState :
1 #acceptChanges
3.4 #triggerUpdateView
2.4 #triggerUpdateChildren
aScriptEditor :
aView :
A
Multiple editors can accept changes subsequently.
4.3 #reset
4.2 #modelReset
5.3 #process: aModel
5.2 #setUp: aScriptStep
aNode :
aPane :
2.1 #sourceCode: someText
3.1 #properties: aDictionary
2.3 #updateFrom: anEvent
3.3 #updateFrom: anEvent
aScriptStep :
theScriptChangeNotifier :
2.2 #sourceCodeChanged: aScriptStep from: oldCode to: newCode
3.2 #propertiesChanged: aScriptStep from: oldProps to: newProps

Figure 5.10: Any change in a script's code or properties is propagated via the singular script-change notifier. Both models and panes get notified to re-configure their contents, which is deferred until the next UI cycle. (See figure 5.8 for A.)

signals `#notify`, our framework will call `#triggerUpdateChildren` in the respective node as illustrated in figure 5.10. We also provide *timed notifiers*, which signal repeatedly at a configurable rate without depending on an external event source. For tool builders, such polling can be the initial way to quickly get up-to-date nodes and thus tangible representations of artifacts.

Finally, view code can change, too, and all existing views should update by re-processing the current model and script. Although we assume the availability of useful view designs and implementations, tool builders can experiment with the adapter that makes such existing designs fit for our VIVIDE framework. That is, the mapping from script or object properties to what the view can do is subject to try-outs. We indicate in figure 5.8 and figure 5.10 that views (or view adapters) have to respond to `#setUp:`, `#process:`, and `#yield:`, to read script properties, to read object properties, and to tell about object selections respectively. We use Squeak's `SystemChangeNotifier` to let panes replace views if any code in that adapter class changes as illustrated via `#triggerUpdateView` in figure 5.10.

Integration into Squeak's Programming Tool Set

Squeak has several means to configure the user interface in a non-invasive way. There are *projects* to add new graphics frameworks [192], *user-interface themes* to add new color designs, or *application registries* to add new domain-specific tools. One such registry is `ToolSet`, which translates "open a code browser" or "debug this process" to an actual tool window. We created a new tool set that applies certain scripts, whose generated models are then displayed via panes and views. For example, the request `ToolSet inspect: myObject` becomes `Vivide openScript: #objectExplorer withObjects: myObject asList`. The script identifier `#objectExplorer` will be looked up in the global script organization. Note that programmers can still open Squeak's traditional tools, for example, through the workspace. Application registries only dispatch default actions such as for keyboard shortcuts or unhandled exceptions.

As proposed in section 4.2, we implemented our practice *view-agnostic menus* through special scripts that display available scripts for a set of objects. To retrieve such a list, we applied three new script properties: `#inputKind`, `#outputKind`, and `#priority`. Thus, for a given set of objects, we can narrow down the list of fitting scripts and then sort that list priority-wise.[11] We use the resulting interface as a replacement for context-menu invocation: `Pane chooseScriptFor: someObjects`. A list will appear, a mouse click will dismiss this list, and a new pane will appear representing the user's choice. Consequently, we integrate objects and scripts directly in the environment as explained in section 4.4. At the time of writing, we maintain `#inputKind` and `#outputKind` manually, but this can be complemented with automatic type harvesting or similar.

We added a global *search field* that yields a list of draggable results as a starting point for the programmer's tangible exploration path. We also added a custom *drop handler* to the world (or desktop) background, which automatically invokes the most appropriate script for the list of dropped objects. Programmers can also invoke our custom "context menu" on the search-result list as described above. Note that users drag custom *transfer (helper) objects*, which makes them incompatible with existing drop targets of Squeak's traditional tools. Yet, we added support for dropping regular transfer objects into VIVIDE tools to, for example, integrate Squeak's (traditional) code browser, which has already draggable classes and methods.

Synopsis We apply many common design patterns to implement structure and behavior of scripts, models, panes, and views. The *composite* pattern guides the representation of model nodes and pane-view compositions. Both *interpreter* and *builder* patterns can be found during script interpretation for model construction. *Observers* are paramount for modifying running tools through script editors or interactive pane re-composition. Being an *adapter* is characteristic for scripts themselves but also necessary to make existing views compatible with our framework.

The integration of VIVIDE into the Squeak environment is straightforward because there are many extension points to plug in new tools and actions. A custom *tool set* maps default requests such as "open browser" to script dispatch and interpretation. An extended *drag-drop handling* integrates existing means of direct manipulation with our framework.

[11]See appendix B.3 for a code listing of the "Scripts" script.

Summary

Our scripting language relies on Squeak's collection protocol to minimize, not prevent, *visible glue* in scripts. References to script steps (via `#next`) can express *cycles*, which result in recursive, maybe infinitely deep, model trees. Model nodes can be updated via *block properties* or *script notifiers*, which integrate external observers. We also expand our concept of anonymous classifications, called *tuples*, to construct *groups* to conveniently connect artifact structure to multiple levels in the model tree.

We apply our data-driven strategy for tool design to create interactive *script editors*. Such editors follow our Rules of Distinctiveness, Similarity, and Context. To illustrate flexibility of such single-object tools, we propose block-based code *folding* and pattern-based code *templates* to support script reading and writing. Overall, script editors in composition show how such tools can easily *bypass structural constraints* via graphical *composition context*.

In our UI-design language, the building blocks, called *panes*, are configured through a combination of (1) script code and (2) user interactions. Considering code, scripts and their object/script properties should be documented and unified by *view creators* so that *script authors* can iterate more efficiently. Considering interactivity, the use of *object connections* and *pane decorations* illustrate means of integration between Vivide and non-Vivide elements as well as pluggability of menus and other UI elements.

Vivide entails a data-driven working practice, which changes the notion of "tools" and "browsing". As a result, even users might have to *copy* scripts if they want to begin new programming tasks to not *overwrite* configurations stored as script properties. As an advantage, switching between *interleaving* programming tasks becomes easier because programmers control on-screen information, which entails *cue priming*. Such inadvertent, yet helpful, cues shorten *task suspension* without affecting task resumption. Finally, we think that *tracing* all these Vivide interactions can help reflect and create better scripts to improve future tools and support future tasks.

[|]

In the next chapter, we begin evaluating and discussing our solution. That is, we share our experiences with Vivide for daily programming tasks.

Part III

Evaluation and Discussion

6 Vivide In Use: Best Practices So Far

In this chapter, we explore some useful practices we observed so far during the design, implementation, and use of Vivide. Our tool-building framework provides generic mechanisms, which benefit from *training* and *experimentation* to discover powerful applications. At the time of writing, we have constructed several replacements for Squeak's standard browsing tools. We tried different ways to employ Smalltalk and Morphic to form an improved programming experience. In particular, we explored trade-offs between *object-oriented* Smalltalk code and *data-driven* Vivide scripts. We also explored trade-offs between *morph composition* from Morphic and *pane/view composition* from Vivide.

Recall that our notions of *information* and *representation* as explained in chapter 2 still guide the definition of "programming tool" and the programmer's goals in programming tasks. In a nutshell, the "what", the "why", and the "how" of tools are as follows:

What — Concept/Structure/Transport Our focus on object-oriented environments simplifies the *definition of information* as objects connected in a graph manifesting concepts of a certain domain as described in section 2.1.

Why — Fetch/Examine/Retain We simplified the information-foraging behavior of programmers in *program comprehension tasks* as activities around software artifacts to highlight issues with generic tools as described in section 2.2.

How — Choose/Configure/Build The programmer's ability to form *artifact representations* yields a range of activities of increasing difficulty. We see *tools as adapters*, which clarifies such tool-building activities as described in section 2.5: (1) tangibility depends on querying, mapping, and presentation languages; (2) application of such languages undergoes iterations of finding, changing, and verifying.

We proposed a new *data-driven working practice* (section 4.4), which alters the traditional habit from *tools-first* to *artifacts-first*. Yet, the level of tangibility we aim for depends on the particular application of scripts (section 4.1) and panes (section 4.3). So, we follow our tool-design strategy (section 4.2) to enact Vivide mechanisms in common scenarios. First, we present scripts (and views) that replace and improve *Squeak's code browsing tools*. Second, we explain when and how to *revise script and pane compositions* to improve modularity and readability. Third, we elaborate on the applicability of scripts for *non-list-based views* such as buttons and treemaps.

6.1 Adaptable Programming Tools

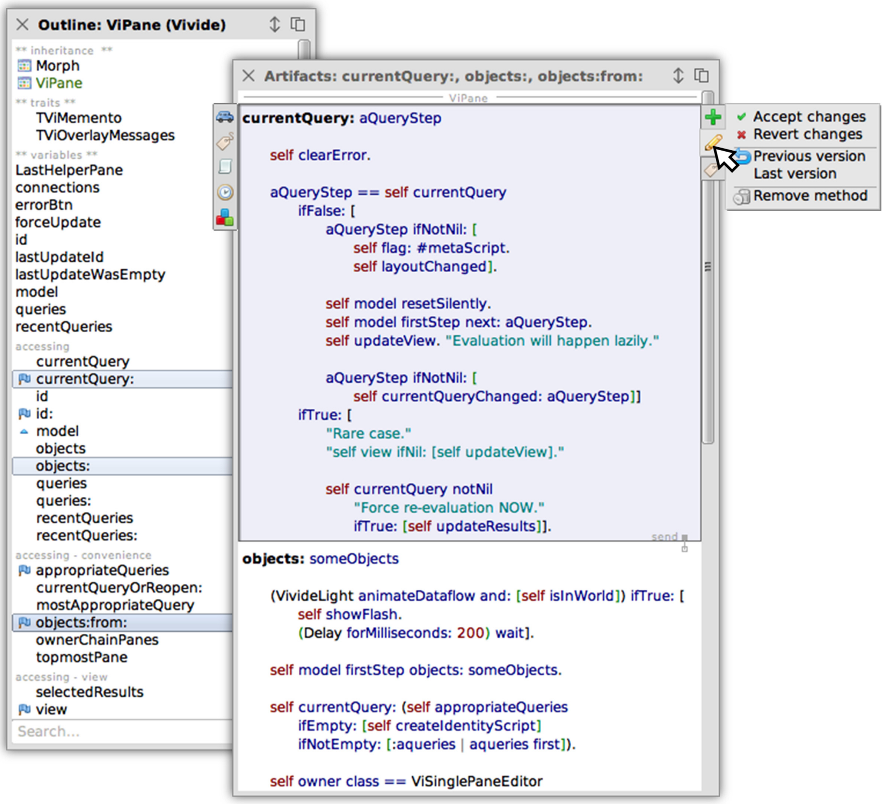
We think it is not reasonable to re-implement Squeak's programming tools *as is* because they impair *modularity in presentation* as proposed in our tool-design strategy in section 4.2. For example, the four-pane System Browser will force programmers to carry redundant context around even if they only want to keep a certain method visible on screen. Although it would be feasible to apply Vivide in that way, we follow a different path: compact, self-sufficient views as tangible representations of artifacts, exposing their distinct properties. That is, we *separate the parts* of the traditional browsers and make them independent views (or windows). Programmers can then compose relations via object connections. Note that the underlying layout strategy is a *desktop* with overlapping windows.

Remark If a Vivide tool is simple, it consists of a single pane and view surrounded by window decoration. For example, a tool that populates a tree view with a single script has no need for complex pane-view compositions. In that case, we can use the terms "tool", "window", "pane", and "view" interchangeably. Programmers can fetch, examine, and retain those entities on screen to arrange information comprehensively.

Class Outline and Method Editor

Two important parts we extracted from the traditional System Browser are (1) the class outline and (2) the method editor as shown in figure 6.1. We replaced the static view composition, many buttons, and pop-up menus with Vivide concepts. That is, we script *tangible code artifacts*, add *selection dragging*, and use *script choosing* to reduce visual overhead and foster onward exploration as proposed in section 4.4. First, the class outline shows the inheritance hierarchy, instance variables, and messages grouped by protocol. Second, the method editor provides syntax highlighting, access to related objects (e.g. senders or versions), and edit actions (e.g. save or revert). Both tools follow our Rule of Distinctiveness and Rule of Context comparing with our script editor in figure 5.1 from section 4.2. We designed all pop-up (or context) menus using our guidelines: the class outline shows a list of scripts for selected objects, the method editor offers two view-agnostic menus to reveal relationships and actions. Overall, we can freely populate the screen with class outlines and method editors to lay out related artifacts and reveal concepts.

The list of method editors is a practical application of *pane views*, which is part of our UI-design Language (section 4.3). We call that list *artifact list*, and it can hold other single-object tools such as class-definition editors or object inspectors. Such artifact lists appear when programmers drag methods from the class outline and drop them on the desktop. At any time, single-object tools can be dragged around



Outline: ViPane (Vivide)

```
** inheritance **
Morph
ViPane
** traits **
  TViMemento
  TViOverlayMessages
** variables **
LastHelperPane
connections
errorBtn
forceUpdate
id
lastUpdateId
lastUpdateWasEmpty
model
queries
recentQueries
accessing
  currentQuery
  currentQuery:
  id
  id:
  model
  objects
  objects:
  queries
  queries:
  recentQueries
  recentQueries:
accessing - convenience
  appropriateQueries
  currentQueryOrReopen:
  mostAppropriateQuery
  objects:from:
  ownerChainPanes
  topmostPane
accessing - view
  selectedResults
  view
Search...
```

Artifacts: currentQuery:, objects:, objects:from:

```
currentQuery: aQueryStep

    self clearError.

    aQueryStep == self currentQuery
        ifFalse: [
            aQueryStep ifNotNil: [
                self flag: #metaScript.
                self layoutChanged].

            self model resetSilently.
            self model firstStep next: aQueryStep.
            self updateView. "Evaluation will happen lazily."

            aQueryStep ifNotNil: [
                self currentQueryChanged: aQueryStep]]
        ifTrue: [
            "Rare case."
            "self view ifNil: [self updateView]."

            self currentQuery notNil
                "Force re-evaluation NOW."
                ifTrue: [self updateResults]].
```

```
objects: someObjects

    (VivideLight animateDataflow and: [self isInWorld]) ifTrue: [
        self showFlash.
        (Delay forMilliseconds: 200) wait].

    self model firstStep objects: someObjects.

    self currentQuery: (self appropriateQueries
        ifEmpty: [self createIdentityScript]
        ifNotEmpty: [:aqueries | aqueries first]).

    self owner class == ViSinglePaneEditor
```

Figure 6.1: Two important views: class outline (left) and list of method editors (right). We opted for not re-creating the traditional four-pane browser from Smalltalk. Instead, selected structural overviews and side-by-side details can make efficient use of screen real estate. In the list of editors, each single-object tool has view-agnostic menus as proposed in section 4.2.

to change artifact (list) compositions. Such compositions can serve as script input. That is, programmers can open the artifact-list halo, create a new script, and begin authoring the representation of the composition. For example, a set of methods can be scripted to reveal their joint usage context in the system. The additional information can support the program comprehension task.

We use Squeak's global `SystemChangeNotifier` to keep the content of class outlines and method editors updated. That notifier fires events for changes in code artifacts such as classes and methods. Each event distinguishes adding, removal, modification, and reorganization. We do not consider these change kinds but only filter according to the object they affect. That is, we want to discard update requests for classes or methods we do not browse in a particular pane. By integrating that notifier, programmers can still use Squeak's System Browser and VIVIDE tools will update, too.

View-agnostic menus, such as the ones for method editors, can be[1] expressed as scripts. For the left-hand relationships, this is straightforward because the object in a single-object tool *is* the input for such scripts. The resulting pane (or view) will represent the visual content in the menu. Take the method editor in figure 6.1 as an example. The underlying script has the object property `#relations`, which holds script identifiers to be looked up:

```
script := {
   [:in :out | in do: [:method | out add: {
     #object -> method.
     #relations -> { #superClasses. "car" #protocols. "tag"
                     #overrides. "page" #versions. "clock"
                     #bindings "bricks" }.
     #text -> [method sourceCode]
            <- [:newCode | method actualClass compile: newSource] }]]
   -> { #view -> TextView.
        #isProperty -> true. }
} asScript.
```

For the right-hand actions, however, scripts have to *modify* the processed object, which they usually do not do. In Vivide, script interpretation happens all the time, which is hardly controllable by the user. We experimented with the idea of having such side effects in scripts at the example of *script choosing* from section 4.4. There, programmers are presented with a list of scripts as pop-up menu to choose a tool to open. Just like other menus, when the user makes a choice, that list will disappear, and a new window will appear. Now, scripts that would just trigger a side effect *without filling the output buffer* could be used to construct such a menu (view). All menu entries would be constructed from the script's `#icon` and `#label` property. Again, take the method editor in figure 6.1 as an example. The underlying script has the object property `#actions`, which holds lists of identifiers:

```
script := {
   [:in :out | in do: [:method | out add: {
     #object -> method.
     #relations -> "... see above ..."
     #actions -> { "Categories of actions."
                   #create -> { #icon -> MenuIcons plusIcon.
                                #scripts -> { "..." } }.
                   #edit -> { #icon -> MenuIcons penIcon.
                              #scripts -> { #accept . #revert. "..." } }.
                   #tag -> { "... " } }
     #text -> "... see above ..." }]]
   -> { #view -> TextView.
        #isProperty -> true. }
} asScript.
```

Note that such menu configurations are *object* properties instead of *script* properties because we use them for single-object tools. That is, the input buffer in the script

[1] At the time of writing, we implemented artifact lists and editor menus in plain Morphic. The use of Vivide pane-views and modifier-scripts is considered future work.

contains a single object only. To go on in this example, the script for the menu entry "Accept changes" has the identifier `#accept` and looks like this:

```
script := {
   [:in :out | in do: [:tuple |
      [:node :editor | node at: #text put: editor theText]
        valueWithArguments: tuple]]
   -> { #id -> #accept .
         #label -> 'Accept changes'.
         #icon -> MenuIcons tickIcon.
         #shortcut -> #(cmd s). "Keyboard shortcut."
        #isModifier -> true . "Earmark side effects." }
} asScript.
```

We opted for *tuples as input* to provide editing view and the object being edited as model node. In the script above, `editor` is an instance of `TextView` and `node` holds the `#text` rule for editing source code. The additional script property `#isModifier` helps distinguish such scripts from regular ones for exploration tasks. The additional script property `#shortcut` denotes a keyboard shortcut, which is just another convenience known from pop-up menus. In this example, we underline the power of Vivide scripts to concisely express domain-specific rules for both exploration and modification.

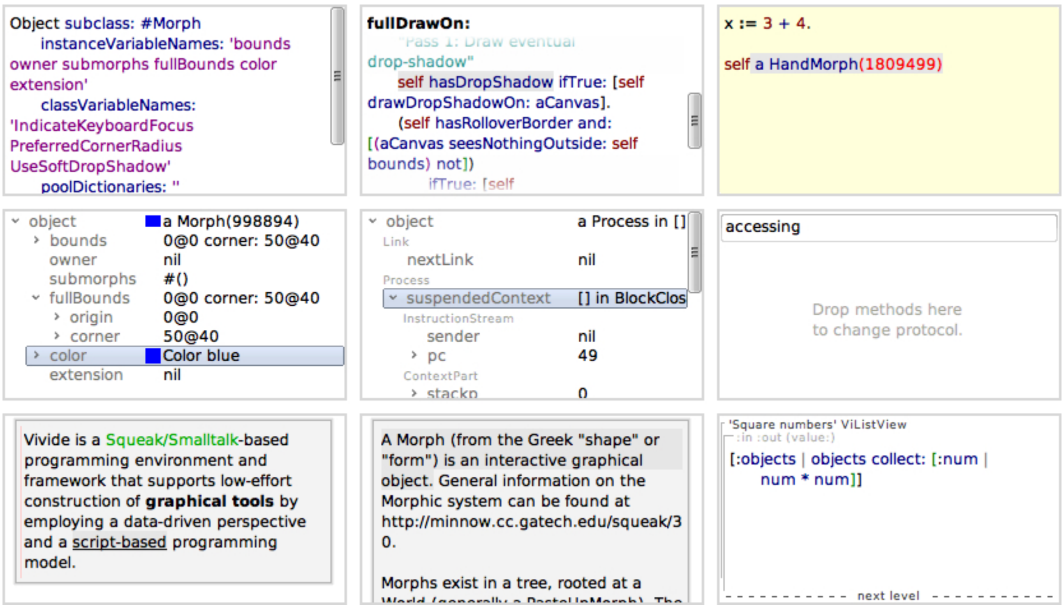
Basic Tangibility: Single-object Tools

We created more tools for single objects (figure 6.2) so that programmers can retain task-related information as compact views on screen. These tools are mainly for code and documentation artifacts.[2] Squeak's alternative, namely the *object inspector*, is too generic for this purpose. For example, programmers cannot comfortably modify methods in an inspector view on a `CompiledMethod` instance. Consequently, we are aiming for *middle ground* that lies between such simple, generic tools and complex, specialized ones such as debuggers and code browsers. In the following, we describe the single-object tools Vivide offers so far.

Class and method editors are the starting point for any code writing activity. We explained their usage in combination with class outlines before. Since these editors reside in artifact lists, they can *create objects* and *spawn editors* to be placed in their context. For example, programmers can create a new method from inside an existing method editor, which then spawns a new editor in the same artifact list.

Workspaces are text editors on objects. That is, the editor binds the pseudo variable `self` to that object. Programmers can then evaluate any Smalltalk expression like in the traditional Squeak workspace. In Vivide terms, this corresponds to the *input* read from the model node. Accordingly, workspaces create the *output* (or object selection) through the result of code evaluation (or *do-it*). Programmers can explore

[2]See appendix B.5 for details.



```
Object subclass: #Morph
    instanceVariableNames: 'bounds
owner submorphs fullBounds color
extension'
    classVariableNames:
'IndicateKeyboardFocus
PreferredCornerRadius
UseSoftDropShadow'
    poolDictionaries: ''
```

```
fullDrawOn:
    "Pass 1: Draw eventual
drop-shadow"
    self hasDropShadow ifTrue: [self
drawDropShadowOn: aCanvas].
    (self hasRolloverBorder and:
[(aCanvas seesNothingOutside: self
bounds) not])
        ifTrue: [self
```

```
x := 3 + 4.

self a HandMorph(1809499)
```

```
⌄ object          a Morph(998894)
  › bounds        0@0 corner: 50@40
    owner         nil
    submorphs     #()
  ⌄ fullBounds    0@0 corner: 50@40
    › origin      0@0
    › corner      50@40
  › color         Color blue
    extension     nil
```

```
⌄ object                 a Process in []
  Link
    nextLink             nil
  Process
  ⌄ suspendedContext     [] in BlockClos
    InstructionStream
      sender             nil
    › pc                 49
    ContextPart
    › stackp             0
```

```
accessing

Drop methods here
to change protocol.
```

Vivide is a Squeak/Smalltalk-based programming environment and framework that supports low-effort construction of **graphical tools** by employing a data-driven perspective and a script-based programming model.

A Morph (from the Greek "shape" or "form") is an interactive graphical object. General information on the Morphic system can be found at http://minnow.cc.gatech.edu/squeak/30.

Morphs exist in a tree, rooted at a World (generally a PasteUpMorph). The

```
'Square numbers' ViListView
:in :out (value:)
[:objects | objects collect: [:num |
    num * num]]
----------- next level -----------
```

Figure 6.2: We created several single-object tools for common Squeak and Vivide objects (left to right, top to bottom): text editor for `Class`, text editor for `MethodReference`, text editor for `Object` as workspace, property tree for `Object` as inspector, property tree for `Process` as debugger, editor and drop field for `ViProtocol`, text editor for `Text`, text editor for `Class`' comments, script editor for `ScriptPart`.

all *variable bindings* in the left-hand menu, which all such single-object tools offer to show related objects as draggable items.

Property trees form our version of a generic object inspector. They traverse all instance variables of an object recursively. If used as a view in Vivide, it does not rely on the (incoming) model nodes to provide these properties. Instead, it implements its own notion of Smalltalk object structure, which includes labels and icons. (Although, that notion *could* be implemented as scripts, too.)

If programmers combine *workspaces* and *property trees* via *object connections*, they get a flexible, data-driven exploration/scripting interface for free. That is, they can re-use workspaces that already have viable information in their bindings. We see this application as a form of *ad-hoc information integration*. To some extent, the result compares with Squeak's object inspector (or explorer), which already has a code-evaluation text field.

We have a specialized version of property trees for `Process` instances. In that version, we provide special *actions* in the right-hand menu such as "proceed", "step", and "terminate". If the inspected process is suspended, its stack frames become accessible. A more sophisticated debugger can then be constructed with Vivide scripts, panes, and views.

Protocol editors are novel in the sense that Squeak has no dedicated class for protocols (or message categories). We implemented `ViProtocol`, which combines the protocol name and its class context. The graphical interface consists of a *text editor* to rename the protocol and a *drop target* to add a method object to that protocol. Note that methods can only be in one protocol at a time. Thus, there is no need to ever remove methods from protocols explicitly.

Our rich-text editor is visually different from method editors and workspaces, which are also text-heavy. It supports instances of `Text`, which combine strings and formatting attributes. Programmers can take notes on the current progress; they can explicate thoughts about the artifacts nearby.

We have a specialized version of rich-text editors for `ViClassComment` instances. Like protocols, such instances combine textual content and class context. Programmers can open many class comments at once. For example, the documentation of the entire super-class hierarchy can be juxtaposed on screen. We display such objectified class comments in the left-hand menu of class and method editors.

Finally, there is our script editor, which we explained in section 4.2 in figure 5.1. Typically, we open all script steps as script editors in an *artifact list*. In contrast to methods in a list, changes in editor compositions have side effects in scripts. That is, closing a script editor means deleting a script step. Also, moving an editor means re-ordering script steps. Nevertheless, programmers can still dismiss *the script-editor tool* by closing the entire artifact-list window.

Getting Started: A Minimal Script Set

In Squeak, programming tools are invoked through *user actions* or *system events*. Such actions include keyboard shortcuts on code snippets and mouse clicks on buttons or menu items. Such events include unhandled exceptions during compilation and code execution. We selected a set of tools that support single programmers in code reading and writing in a non-distributed Squeak programming session. So, our goal is to provide VIVIDE scripts for tasks that align with programming-language artifacts. Comparing with the *class outline* from above, we focus on simple tasks and omit more complex ones, such as version control and team collaboration, for now. Based on our personal programming experiences with Squeak/Smalltalk, we present *a minimal script set* to bootstrap VIVIDE tools.

Remark We present those tools *not* as script code. Instead, we describe (1) the script's goal, (2) the kind of input objects, (3) the transformations applied, and (4) the properties extracted. Any resulting script can then be as simple or as complex as desired by the tool builder. We think that the kind of view is also a matter of personal choice. Typical choices include list-, table-, or tree-like views with support for drag-and-drop.

Our tool-building environment relies on a set of **kernel scripts** to support object collection, script selection, and pane debugging:

Artifacts Programmers should be enabled to collect objects in a container. This script supports all kinds of objects, does not apply any transformations, and extracts only generic properties such as `#text` via `#printString` and `#icon` via Morphic thumbnails. The view should support drag-and-drop to collect objects from other views. We decided to put single-object tools as representatives in the view as shown in figure 6.1.

Scripts Programmers should be enabled to select scripts for a set of objects. This script supports all kinds of objects, transforms them into a list of suitable scripts ordered by priority, and extracts descriptive properties for each script such as `#text`, and `#icon`. We think that a list of script organizations should be provided in the form of tuples. We decided to put the transformation from objects/organizations to scripts in the implementation of `Pane`; sort and extract is in script code. See appendix B.3 for example code.

Hierarchy Programmers should be able to explore and debug the graphical hierarchy made of scripts, panes, views, and pane views as illustrated in figure 4.11 in section 4.3. This script supports panes as input and transforms them into a *recursive* tree structure with three different levels: (a) panes, (b) script and view, (c) panes if view is pane view. Note that (b) produces always two objects while (a) and (c) can vary. We sort panes according to their position on screen. For scripts, we extract `#text` and `#icon` from their script properties. For panes, we indicate the use of pane views. For views, we extract a thumbnail, which all morphs provide. We decided to make this hierarchy script accessible through the pane halo.

There are **generic scripts**, which can be used as a starting point for a given list of objects:

Default Programmers should be enabled to examine a simple, tangible representation of any object list. Being similar to Artifacts, this script supports all kinds of objects, does not apply any transformations, and extracts only generic properties such as `#text` and `#icon`. Consequently, this script is the equivalent of Squeak's `#printString` for Vivide.

Groups Programmers should be enabled to browse groups after identifying a *group object*. This script supports a list of *tuples*, transforms them into groups using the first object in each tuple, and extracts generic properties for group and content objects. The transformation also includes lexical sorting using generic text representations for group and content objects. We decided to *inline* group objects as non-selectable separators. An alternative would be to form a two-level tree model with the first level being group objects and the second one being content objects.

Explorer Programmers should be enabled to examine the Smalltalk object structure for any object list. This script supports all kinds of objects as input and transforms them into a *recursive* tree structure. Squeak's meta-object protocol provides access to any object's (named and unnamed) instance variables, which then point to other objects. Comparing with Squeak's object inspector, this script's view can account for an input field to support code evaluation. For each object, we extract generic properties such as `#text` and `#icon`.

For standard programming activities, there are **code scripts**, which primarily replace Squeak's code browsers. These scripts are made for Squeak's *code artifacts*:

Classes (or class outline) Programmers should be enabled to browse the contents of classes. This script supports instances of `ClassDescription`, which includes `Class` and `MetaClass`. It transforms classes into super-classes, instances-variable objects, and methods sorted by name and grouped by protocol. It extracts labels and icons as illustrated in figure 6.1. (We decided to treat a script for class hierarchies or sub-classes as optional.)

Packages Programmers should be enabled to browse the contents of packages, which organize classes. This script supports instances of `PackageInfo`, transforms those package objects into classes sorted by name and grouped by category, and extracts `#text` and `#icon`. Besides their name, some classes such as `Morph`, `Magnitude`, and `Collection` have distinct icons to quickly recognize the inheritance relationship of subclasses.

Senders Programmers should be enabled to explore references to certain symbols in code artifacts such as methods. This script supports instances of `Symbol`, `MethodReference`, and `CompiledMethod`; it also supports instances of `ClassReference` and `ClassDescription`. First, this script transforms classes, methods, and references into symbols using their names. Then, it looks up code references through the global *SystemNavigation*, which yields method references. We suggest the use of *tuples* if no such global service is available.

Implementors Programmers should be enabled to explore the source code behind certain symbols. This script supports the same instances as Senders, performs similar transformations, but it can yield references to methods *and* classes. (Note that it is considered best practice to only let class names begin with an uppercase letter.)

Processes Programmers should be enabled to control suspended processes. As explained in [192], Squeak manages code execution through instances of `Process` and `MethodContext`. Thus, this script supports instances of `Process` and transforms them into context objects. It (and its view) should offer *actions* to perform the usual debugging activities such as "terminate", "proceed", and "step over". Note that each context object has references to code artifacts and variable bindings.

Remark The scripts Senders and Implementors are examples for Vivide tools that improve Squeak's standard tools. Scripts can process multiple objects and thus show, for example, senders of many symbols in combination. In Squeak, programmers have to open multiple Senders browsers to follow multiple symbols (or messages). This can be frustrating because Smalltalk maps *optional* message arguments to related messages such as `#with:`, `#with:with:`, and `#with:with:with:` in Squeak. While any traditional tool can be adapted to support such a feature, it is *inherently* supported in Vivide because of its underlying concepts.

Finally, there are **global scripts**, which do not depend on input objects. Instead, they expose Squeak's global variables to access objects. Also, text-based views can often derive (or look up) arbitrary objects from user input:

Search Programmers should be enabled to type any term into a text field. This script can support input of any kind. It does not transform that input, but it does extract textual properties to derive search terms. The view should yield all search results as object selections to the Vivide framework; it may directly display those results as selectable objects.

Workspace Programmers should be enabled to type any Smalltalk expression into a text field. This script can support input of any kind. It does not transform that input, but it derives the `#self` binding and other bindings for its textual view. Similar to the single-object tool in figure 6.2, the view yields the result of code evaluations as object selections to the Vivide framework. (Note that this script *can* be a debugging tool.)

Transcript Programmers should be enabled to watch the results of `Transcript show: anObject` and similar print-outs used for debugging purposes. This script can support instances of `TranscriptStream`. It then either passes on that stream or accesses the global variable `Transcript`, which is Squeak's default instance of this class. Since our current implementation does not support streams, the view as to watch for new contents to be displayed and updated frequently. We decided to add a helper object called *transcript history*, which logs all messages into a collection to be accessed in this script.

Environments Programmers should be enabled to browse *all available* code artifacts in the system, which is the manual version of the Search script. This script can support instances of `Environment`, `PackageOrganizer`, or `SystemOrganizer`. It then either passes on that organization or accesses the default environment. Then, it transforms environments into organizations into instances of `PackageInfo` sorted by name. As properties, it extracts `#text` from that name and maybe `#icon` for consistency. Note that we prefer packages over categories in Squeak because categories have no own class but are instances of `Symbol`.

Schedulers Programmers should be enabled to explore *all available* code paths in execution (or control flows). In Squeak, this script can support instances of

`ProcessorScheduler`. It then either passes on that scheduler or accesses the default one via `Processor`. Then, it transforms schedulers into instances of `Process` sorted by priority. As properties, it extracts `#text` from the process' name and also its priority. This script complements PROCESSES and TRANSCRIPT and is thus also used for debugging.

SINGLETONS Programmers should be enabled to browse *all available* software artifacts in the system. We refer to the eponymous design pattern [62, pp. 127–134], commonly applied in object-oriented systems. This script is more general than ENVIRONMENTS and SCHEDULERS. As input, it can accept *patterns* to look for through instances of `Symbol` or `MessageSend`. By default, this script looks for classes, which are globally accessible, that implement certain (class-side) messages: `#uniqueInstance`, `#instance`, `#default`, `#main`, `#current`. Here, transformation means sending those messages to class objects to retrieve the singleton objects. In addition, we look up all *global bindings* such as all class-name symbols, `Transcript`, and Morphic's dynamically scoped `ActiveHand`. (Note that we actually decided to omit class objects for the sake of overview and because of the scripts SEARCH and ENVIRONMENTS.)

[|]

Our framework generates **identity scripts** to support script modification. Recall that all views (or tools) in VIVIDE carry (or are) *tangible representations* of certain artifacts (or objects). However, some new artifacts are born only *after* user interaction. For example, a new method can only be compiled after the programmer has entered valid syntax including the method's name. After that, any view can "be" that method. To bypass this constraint, we *automatically* create the first version of a new artifact, which can then be modified or deleted interactively. For script editors, this means the creation of identity transformations to be inserted into existing scripts or used in existing panes. All single-object tools have this requirement to represent tangible objects. Thus, tool builders should design an "empty" version for any kind of object. From an architectural perspective, this compares to the Null-object pattern [84, pp. 301–309].

The creation of identity scripts is closely related to our template-based *script wizard* presented in section 5.2. We distinguish a normal version and a tuple version, which are both optimized for one-to-one or one-to-many transformations through `#asList`. Given a list of objects, we encode type information as block variables to help programmers read and modify the created scripts later on. For example, a list of *numbers* results in the following script:

```
[:in :out | (
  [:all | all collect: [:number | number] ]
    value: in) do: [:result | out addAll: result asList]]
```

Programmers can then edit the expression in the gray box or activate the script wizard to apply a different template. For tuples, we nest an additional block to expand the tuple to named block variables. For example, a list of tuples that associate *colors* and *morphs* results in the following script:

```
[:in :out | (
   [:all | all collect: [:tuple |
      [:color :morph | {color. morph} asTuples]
        valueWithArguments: tuple]]
           value: in) do: [:result | out addAll: result asList]]
```

We think that panes should automatically switch to an identity script if the current one cannot be interpreted. Programmers can then use the pane halo to access the faulty script. That is, we favor populated views over empty ones.

Scripts as Unit of Re-use

In Vivide, view models are *never* shared among views because they are generated from scripts and objects. Thus, scripts are the unit of re-use. As explained before, every script consists of one or more steps that form a singly-linked list. So, each step can be a script on its own to be interpreted along all linked steps. Scripts have identifiers, which can be referred to in the script property `#next`. The script interpreter will then follow that `#next` reference before turning to the next linked step. Now, programmers can apply both regular links and `#next` references to design scripts that re-use common object transformations and property extractions.

First, programmers can add the script property `#next` to any step. After interpreting the particular step, the interpreter will look up the identifier(s) in all accessible script organizations. It will *implicitly* amend the Default script if there is no property extraction specified at the end:

```
script := {
   [:in :out | in do: [:number | out add: number * number]]
      -> { #next -> #default  "Not required here."}
} asScript openScriptWith: #(1 2 3 4 5).
```

Note that Vivide inserts more standard properties if not specified such as `#id` and `#view`. Another common example for such script re-use is the Groups script. Programmers can just define a distinct object, construct tuples, and delegate further property extractions. With the help of our script wizard and templates, script code can be as short as this:

```
script := {
   [:morph | {morph color. morph} asTuples ]
      -> { #next -> #groups }
} asScript openScriptWith: someMorphs.
```

The Groups script can then decide whether to inline separators in the same level or allocate an additional one in the model tree. After that, again, there should be a reference to the Default script to get predictable (and configurable) results.

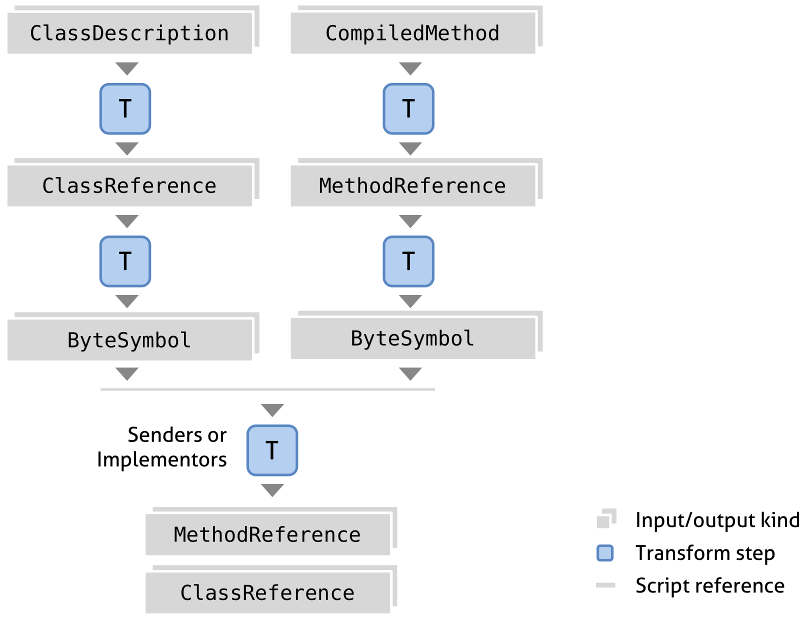


Figure 6.3: Script steps are unit of re-use. Every step is a script on its own. In Squeak/Smalltalk, senders and implementors of symbols can require different transformations for different kinds of objects. Note that there are no means to re-use such *preceding* steps across scripts, but duplication is necessary.

Second, programmers can link object transformations and exploit the fact that every step is a script on its own. In Squeak, there are many common exploration paths in the object graph. For example, there is a name for every class or a reference for every method. Depending on the kind of objects at hand, there are common transformations to be applied again and again during a programming session. In figure 6.3, we illustrate the script structure for SENDERS and IMPLEMENTORS. Given that programmers have tangible representations of classes or methods, Squeak's Senders and Implementors tools require several entry points. Eventually, there is a `ByteSymbol` to be looked up in all code artifacts. Yet, that information is stored somewhere in those given objects. Consequently, references should be transformed into symbols; classes or methods should be transformed into references. One could *directly* extract `ByteSymbol` from `ClassDescription`, for example, but that would aggravate modularity and possible re-use. There are more related object kinds, which can benefit from this practice, such as `Morph`/`Form` and `String`/`Text`.

We think that step(s) that follow after such preparing transformations should be referred to via the `#next` property because our script editor shows scripts as lists. A script step with more than one incoming link would be hardly discoverable. Thus, the preparation for SENDERS, including type information for each step, looks like this:

```
script {
  [:method | method methodReference]
    -> { #id -> #sendersMethods.
         #inputKind -> CompliedMethod }.
  [:reference | reference selector]
    -> { #id -> #sendersReferences.
         #inputKind -> MethodReference.
         #next -> #senders }.
} asScript openScriptWith: {
  Boolean >> #ifFalse:.
  Boolean >> #ifFalse:ifTrue:.
  Boolean >> #ifTrue:.
  Boolean >> #ifTrue:ifFalse: }.
```

This script invocation via `#openScriptWith:` results in a tool window that shows a list of all methods that use control-flow branching. In Squeak, programmers would have to open *four* Senders browsers instead.

Advanced Code Exploration

Vivide scripts can analyze code artifacts to reveal *abstract patterns* and meaningful insights for program comprehension. Analytical results can then point to the artifacts involved to support further reasoning. This practice complements (and integrates) the scripts from above, which support *direct* exploration and modification of such artifact structure. In object-oriented systems, there are common things to look out for such as *Code Smells* [57, pp. 75–88] and *Disharmonies* [99]. That is, the identity of, collaboration between, or classification of some objects might be questionable and worth improving. We think there is an own category of tools that analyze code and visualize the results such as *Linting Tools* [81] and *Intentional Views* [116]. Yet, there can also be *positive* reasons to extract higher-level concerns from a list of code artifacts for teaching or learning. In the following, we state our experiences with Vivide scripts to (1) check software metrics, (2) define extended message protocols, and (3) reveal navigation paths via code annotations.

Simple software metrics can be derived from Squeak's code artifacts. If programmers add thresholds, scripts can help watch out for metrics-related problems such as long methods or complex classes. Such thresholds are configurable by any tool user. We suggest that scripts should not transform artifacts *into* metrics but use them as filters or extract them as properties. Views can then display that information without impairing the artifact's tangibility such as via tooltips and extra table columns. In the following, we describe some exemplary metrics and how to implement them as transform or extract steps:

- If a method has more than 20 lines of code, make it red:

```
[:method | { #text -> method selector.
             #color -> ( method linesOfCode > 20
                          ifTrue: [Color red]
                          ifFalse: [Color transparent]) }]
```

- If a class has more than 10 instance variables in its hierarchy, add a small flame in front of its name:

```
[:class | { #text -> class name.
            #icon -> ( class allInstVarNames size > 10
                          ifTrue: [MenuIcons flameIcon]
                          ifFalse: [MenuIcons blankIcon]) }]
```

- In this package, show all classes that have no comments in their description:

```
[:in :out | in do: [:package | out addAll: (
   package classes select: [:class |
      class organization classComment isEmpty ] )]]
```

With scripts, extended message protocols can be derived from Squeak's code artifacts. Programmers can organize messages in categories, which are also known as protocols. Yet, higher-level rules such as the absence of certain messages or the use of Squeak's meta-object protocol have to be evaluated dynamically as the code changes. Scripts can capture these rules to filter code artifacts or extract descriptive properties for views to show. In the following, we describe exemplary rules and how to implement them as transform steps:

- In a package, show all classes that omit to implement `#hash` if they override `#=` and vice versa:

```
[:in :out | in do: [:package | out addAll: (
   package classes select: [:class |
       (class includesSelector: #hash)
         xor: [class includesSelector: #=] ] )]]
```

- If a method uses certain meta programming, make it red:

```
[:method | { #text -> method selector.
   #color -> (
      (#(respondsTo: perform: value:)
        anySatisfy: [:selector | method hasLiteral: selector])
            ifTrue: [Color red]
            ifFalse: [Color transparent]) }]
```

- In a package, show all classes that participate in Squeak's default observer pattern:

```
[:in :out | in do: [:package | out addAll: (
   package classes select: [:class | (
      #(update: update:with:)
        anySatisfy: [:symbol |
          "Implementors"
          class includesSelector: symbol] )
      or: [
      #(changed: changed:with:)
        anySatisfy: [:symbol |
          "Senders"
          (class whichSelectorsReferTo: symbol) notEmpty] ]] )]]
```

Scripts can expose comments and other annotations that document decisions without being relevant for program execution. Such annotations can be useful for explicating navigation paths [188] or communicating general concerns in the team [205]. Squeak reserves the non-functional `#flag:` message, which directly integrates with the Senders tool because it is just a message. Here is an example:

```
"..."
self flag: #refactor. "mt: Extract the computation to shorten this method and make the
code reusable elsewhere. See ticket 1234 for more discussions about this issue."
"..."
```

We analyzed the entire syntax tree to extract the symbolic argument as well as the comment. We created the class `ViFlagComment` to pack all information into an object to write scripts for. Flag comments point to the method objects, which is useful for integrating scripts to support onward exploration. Another example for such annotations are *method pragmas* like this:

```
ExampleClass >> exampleMethod
  <priority: 10>
```

Such pragmas (<. . .>) do usually influence an application's run-time behavior. Yet, they might as well denote landmarks in the source code, which would be similar to methods that just contain a large comment to document a part of the class. Scripts can access pragmas via Squeak's meta-object protocol to filter methods or extract properties, which is similar to the examples above.

Synopsis Programmers work with generic programming tools to explore code artifacts and write source code. We apply Vivide scripts to design traditional views on such artifacts, which render the resulting tools (or views) adaptable. On the one hand, there are *class outlines* and *lists of method editors* to freely and concisely juxtapose related code on screen. On the other hand, there are more *single-object tools*, which provide basic tangibility for other code artifacts such as comments and protocols. A *minimal set of scripts* provides the baseline for domain-specific or task-specific adaptations.

The generality of scripts offers many possibilities and demands for several choices to be made during tool building (or script authoring). Considering *scripts as unit of re-use*, programmers can package object transformations in scripts, whose steps either *link* to another directly or *refer* to it via the `#next` property. Considering *code analysis*, programmers can apply scripts to expose software metrics that either *filter* artifacts or *extract* additional properties, which views can display, for example, in tooltips or additional labels.

6.2 Revising Script Code and Pane Design

Tool building in Vivide remains an iterative process. Programmers revise script code and pane compositions to form (tool) views that carry tangible representations of artifacts. Guided by immediate feedback and experimentation, good and bad patterns can emerge. Thus, programmers should be aware of a *technical debt* [46], which can follow their design choices. Similar to programming in general, *refactoring* [57, 84] should remain an inherent part of tool building.

It is difficult to objectively assess the impact of design choices. So far, we have not yet discovered many bad applications of scripts and panes other than *long steps* or *complex connections*. Those are similar to common object-oriented smells such as Long Method [57, pp. 76–77]. In the following, we describe the new dimensions Vivide adds to tool building in Squeak. These dimensions imply trade-offs programmers have to make. For example, data-driven scripts and object-oriented methods can go hand in hand; script-based panes and plain morphs can both define the graphical user interface. Note that any resulting technical debt depends on the programmer's particular goals.

Fetch & Examine: Between Scripts and Messaging

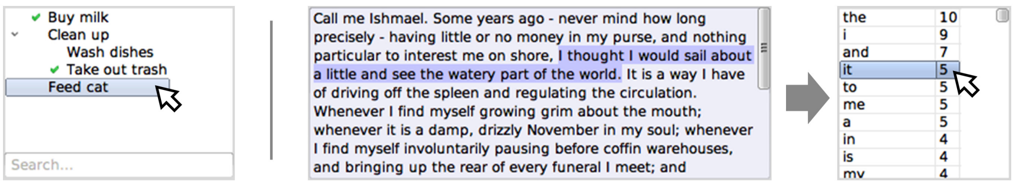
Can I express the rules to query and map my artifacts entirely as scripts or do I have to add new classes and methods?

Programmers can apply our scripting language *most concisely* if artifacts have *all* transformations and properties built in as *messages*. Yet, new ideas are likely to increase the script size due to extra Smalltalk code. To shrink verbose scripts, such code can be moved to classes or methods. While such refactorings can be effortful, they can also improve modularity for the entire object-oriented system. It remains a trade-off to be constantly evaluated in the process.

Smalltalk is a strongly-, dynamically-typed language. Everything *is* an object, but objects can take *any* form through relationships to other objects. For scripts, Squeak provides several *containers* that help programmers organize new object structure:

- `Array` can be used to carry ordered lists of objects between scripts. Programmers that use our *tuples* use instances of this class.
- `Association` can be used to relate a *key* object to a *value* object. For example, programmers can carry a name or some other context for any object with instances of this class.
- `Dictionary` can be used to construct complex object structures via lists of associations. For example, programmers can model a person with name and age without having to create a new `Person` class.

Such containers can be used to avoid the creation of new classes for, maybe unique, tool-building experiments. When artifacts do not provide the messages required for



```
✔ Buy milk
⌄ Clean up
    Wash dishes
  ✔ Take out trash
  Feed cat
Search...
```

Call me Ishmael. Some years ago - never mind how long
precisely - having little or no money in my purse, and nothing
particular to interest me on shore, I thought I would sail about
a little and see the watery part of the world. It is a way I have
of driving off the spleen and regulating the circulation.
Whenever I find myself growing grim about the mouth;
whenever it is a damp, drizzly November in my soul; whenever
I find myself involuntarily pausing before coffin warehouses,
and bringing up the rear of every funeral I meet; and

the	10
i	9
and	7
it	5
to	5
me	5
a	5
in	4
is	4
my	4

Figure 6.4: Squeak's generic object structures can store domain-specific contents without having to create new classes. The task list on the left uses `Association`, `String`, `Boolean`, and `Form`. The word counter on the right uses `Text`, `Bag`, `Association`, `String`, and `Number`. (The example analyzes the first paragraph of Herman Melville's "Moby Dick".)

transformation or property extraction, programmers can assess the situation using the following questions:

- Can I perform the transformation or extraction with the objects given in the local input buffer or global variable bindings?
- Can I capture the results in a container such as tuples to be passed on to the next script step?
- Do my additions affect the performance of script interpretation in a way that makes an external cache necessary?

In our experience, scripts will become more complex if they have to compensate for missing, but relevant, artifact structure. Specific domains and tasks are likely to yield more questions. For the general case, we found several *idioms* worth considering to cope with accidental complexity:

- Favor many, short, consecutive transformations over a few, long ones with many nested calls to the `Collection` protocol.
- Favor tuples and other containers over additional, task-specific data classes.
- Yet, favor additional domain classes over valuable transformations that spread across many scripts.

Eventually, script code should read like domain rules. While Vivide hides much glue in its framework, tool builders still have to express those rules concisely as Smalltalk code.

We present two productivity tools in figure 6.4: a task list and a word counter. Both examples configure Squeak's generic object structures as explained before to store domain-specific information. First, *tasks* are *lists of associations* between a label and a boolean or other tasks to support task composition:

```
tasks := { 'Buy milk' -> false.
           'Clean up' -> { 'Wash dishes' -> false.
                           'Take out trash' -> false }.
           'Feed cat' -> false }.
```

Our scripts can directly express this recursive structure and support interactive tick-off via mouse clicks. Our item views support the configuration of callbacks such as #doubleClicked and #selected, whose call-backs are Smalltalk block closures. Global script notifiers trigger the view update after ticking off a task:

```
script := {

   "Short version for property extraction. Uses script templates."
   [:task | {
     #text -> task key.
     #icon -> (task value == true ifFalse: ["..."] ifTrue: ["..."]).
     #doubleClicked -> [[
         task value isList ifFalse: [
             task value: task value not. "Tick off task."
             ViEventNotifier trigger: #tasksChanged. ] ]] }]
   -> { #id -> #taskList.
        #isProperty -> true.
        #notifier -> [ViEventNotifier named: #tasksChanged] }.

   "Recursion for composite tasks. Uses script templates."
   [:task | task value isList ifFalse: [#()] ifTrue: [task value]].
     -> { #next -> #taskList }

} asScript openScriptWith: tasks. "See above."
```

In the same spirit, the word counter can rely on Squeak's generic object structures. We connected two panes: the left one yields a text object entered by the user and the right one analyzes that text in a table. The word-count analysis makes use of Squeak's string manipulation and bag conversion protocols:

```
script := {
   "1) Transform and count words"
   [:text | text asString].
   [:string | string findTokens: ' .-,;' "Configurable separators"].
   [:token | token asLowercase].
   [:in :out | out addAll: in sortedCounts "Count words"]
     -> { #in -> Bag. "Removes duplicates; maintains quantity"
          #out -> OrderedCollection "The default collection"}.
   "2) Extract two text properties for two table columns"
  [:assoc | { #text -> assoc value } ]. "word"
  [:assoc | { #text -> assoc key } ]. "quantity"
} asScript openScriptWith: #('Call me Ishmael. Some years ago...').
```

In these two examples, the *persistence* of domain objects is volatile. In both task list and word counter, users should refrain from dismissing the panes' (tool) windows. This is a fairly common scenario in Squeak because users can save-and-close the entire environment, which compares to *hibernation* (or suspend-to-disk) in operating systems. Another example are workspaces that hold valuable code snippets only in their text buffer. Note that Squeak's windows can be made "unclosable", which removes the close button to avoid accidental dismissal. We argue that the inherent persistence of Squeak's object graph is worth exploiting to improve the user's and programmer's experience.

[|]

So far, we discussed the efficient use of messages in script code. But programmers can embed scripts in regular Smalltalk code, too. With some objects as input, they can interpret any script object to retrieve a model (node) object. Recall that everything is an object, and messaging yields behavior:

```
| input script node output |
input := #(1 2 3 4).
script := { [:num | num * num] } asScript.
node := script interpretScriptWith: input.
output := node children collect: [:ea | ea at: #object].
output := node objects. "Convenience message."
```

In such a scenario, scripts serve as *unit of extension* for any object-oriented module. Transparently, scripts can attach additional state (and behavior) to classes. This can be useful if programmers do not want to change the (foreign) module's sources directly. If expressed more concisely through script identifiers and compact model messages, the square of a number could read like this:

```
squareNumber := #square apply: aNumber.
squareNumbers := #square applyAll: someNumbers.
```

Here, `#square` is a script identifier to be looked up in a script organization. The messages `#apply:` and `#applyAll:` combine script interpretation and model unpacking for single objects or lists of objects respectively.

Examine & Retain: Between Panes and Morphs

Can I express the rules to map and present my artifacts entirely through (scripted) views or do I have to design new morphs?

Many elaborate presentation languages go beyond text. They combine shapes, colors, and geometry to visualize information and compare multiple data points side by side. While text remains important for selected details, other graphical elements are often more efficient to gain insights. For Vivide tools, this means that the visual mapping in scripts addresses not only `#text` but often `#color` and `#icon`. Programmers have to choose from typically generic views and populate them with specific structure. In Squeak, morphs play an important role for tool design because they define all interactive graphics in the environment. Every graphical element is a morph and Vivide panes hook into that hierarchy on screen.

First, every Vivide pane is a morph. This makes panes being a new "bridge" between information and representation. On the one hand, tool builders can make existing morphs compatible with panes by implementing the view protocol. On the other hand, tool builders can employ scripted views in any part of an existing Morphic application. Since panes provide their own halo, programmers can explore and debug the application's mixed graphical hierarchy in a common way. Considering games and other multimedia applications, programmers should assess the usefulness of panes beforehand:

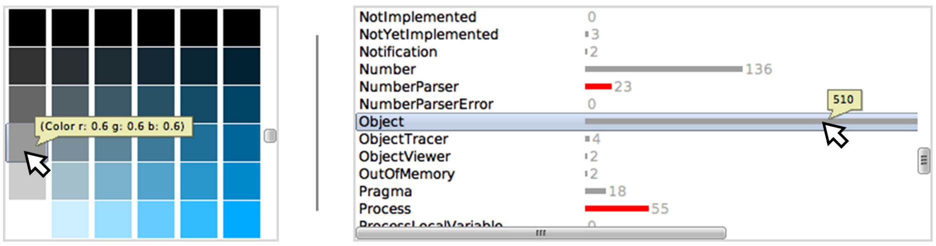


Figure 6.5: Morphs can be created from objects that have no inherent graphical representation such as numbers. On the left, color objects made of numbers for hue, saturation, and value can set a morph's basic color directly. On the right, the count of methods in classes can set a morph's geometry such as its width in pixels; the count of instance variables can set the color.

1. Does the morph show distinct properties of a certain kind of domain artifact?
2. Can the morph show multiple of such domain artifacts in combination?
3. Is the morph part of a larger context with other morphs influencing each other?

If so, VIVIDE panes can make a valuable addition to the user interface. For example, the *player* in a jump-and-run game should be a regular morph while the *highscore* table in that game could be a scripted view. We think that the opportunity of using VIVIDE views might influence the object-oriented design of the application. Such highscore objects, for example, might not just be associations of player name and number, but sophisticated objects that could also store replays of level runs.

Second, every morph is an object. Consequently, scripts can pass *morphs as object properties* to views, which can then integrate them directly into the graphical hierarchy. To an extreme end, views could rely on models to contain morphs that have all visual properties configured just waiting to be displayed. For example, a view can look like a *scatter plot* if its script maps objects to morphs, each having its own horizontal position, vertical position, color, and size. To a practical end, morphs could be interactive additions for views that are often just sophisticated layout containers. For example, a model node can hold a *clickable button* morph.

We experimented with morphs as vehicle for the `#icon` property, which our list, table, and tree views support. In Squeak, icons are instances of `Form`, which can be derived by rendering a morph via the message `#imageForm`. Then, any complex composition will be flattened into colorful pixels. In figure 6.5, we show two examples that use this technique: a color picker and a (vertical) bar chart. Color objects can be created from numbers:

```
colors := Array streamContents: [:stream |
   (0 to: 360 by: 20) do: [:hue |
      (0 to: 1.0 by: 0.2) do: [:sat |
         (0 to: 1.0 by: 0.2) do: [:val |
            stream nextPut: ( Color h: hue s: sat v: val )]]] ].
```

In a script's extract step, morphs can represent those colors directly:

```
script := {
   [:color | { #icon -> (Morph new extent: 32@32;
                                   color:  color ; "Direct mapping, color to color."
                                   imageForm) "Convert morph to icon." }].
} asScript openScriptWith: colors. "See above."
```

In the same spirit, the vertical bar chart creates bars from morphs. Here, we analyze classes and map the number of methods and instance variables to a bar's width and color respectively:

```
script := {
   [:class | { #text -> class name }].
   [:class | { #text -> class methodDict size.
               #icon -> (Morph new
                          height: 5;
                          "Direct mapping, number to number"
                          width:  class methodDict size ;
                          "Indirect mapping, number to color"
                          color: ( class instVarNames size > 5
                                      ifTrue: [Color red]
                                      ifFalse: [Color lightGray]);
                          imageForm) }].
} openScriptWith: (PackageInfo named: #Kernel) classes.
```

Besides rectangles, Squeak provides morphs for other shapes such as circles and polygons.

Levels of Interactivity: Between Script Composition and Pane Composition

What is the difference between one complex tree model in a single pane and a combination of connected panes each with simple list models?

The *dualism* of scripts and panes opens up a spectrum of user interfaces with varying levels of utility and usability. In scripts, tool builders can express the rules to query artifacts of interest. Recall that transform steps convert one set of objects into another; Squeak's object graph can be traversed and extended this way. Now, tool builders have to decide (1) when to make use of the model's *tree structure* and (2) when to choose *connected panes* (and views) for interactive exploration. On the one hand, one complex script in a single view can save screen space, but that view needs to be able to display the entire model tree. On the other hand, many simple scripts in many simple (list) views can be faster to create. So, there is a trade-off between scripts and panes to be made in a tool's presentation language.

Programmers can convert pane compositions into script compositions because both alternatives compose through the exchange of objects. For scripts, objects "move" from one step's output buffer to the next one's input buffer. For panes, objects "move"

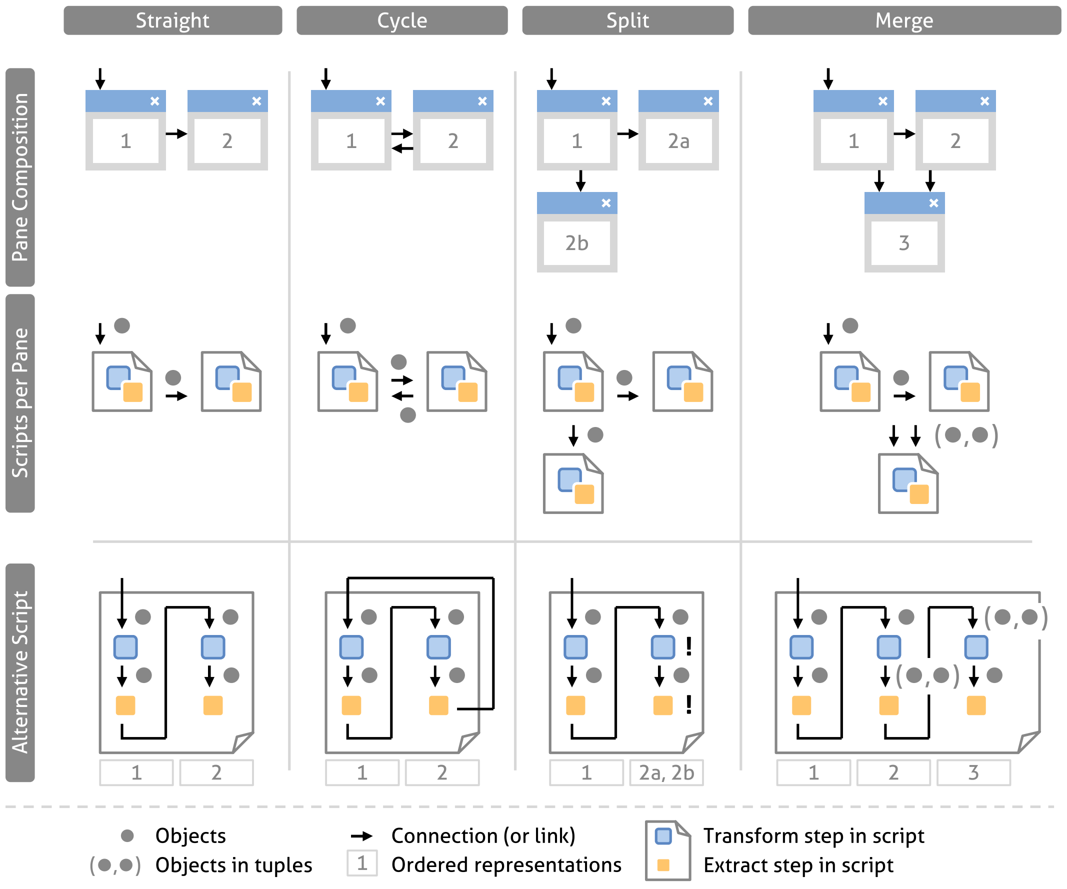
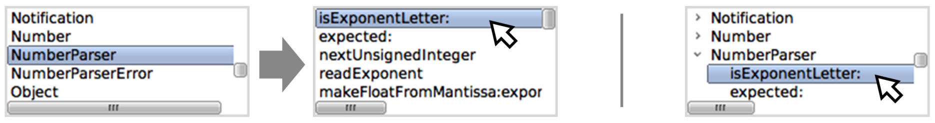


Figure 6.6: Script compositions and pane compositions can be converted into each other. Hierarchical view models require hierarchical views to display all artifacts. This forms a trade-off between spatial efficiency and overall usability.

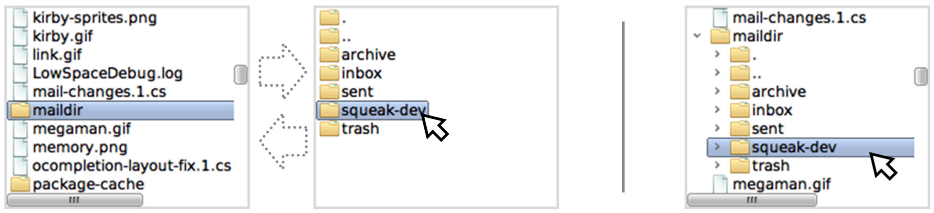
from one pane's view selection to another one's script input.[3] As illustrated in figure 6.6, we identified four of such composition patterns to resolve conflicts and guide tool building:

- **Straight** — Two or more panes form a connected, uni-directional line. To construct a single script, each pane's script can be appended one after another. The count of panes will match the count of levels in the model tree. For example, a browser for classes and methods with two list views becomes a two-level tree:

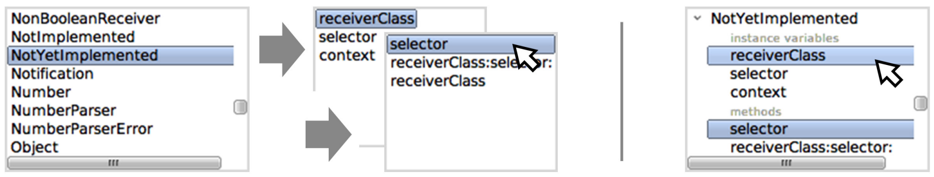


[3]As explained before, object connections are have a direction, a *use mode*, and a *provision mode*. By default, those modes are *view selection* and *script input* respectively.

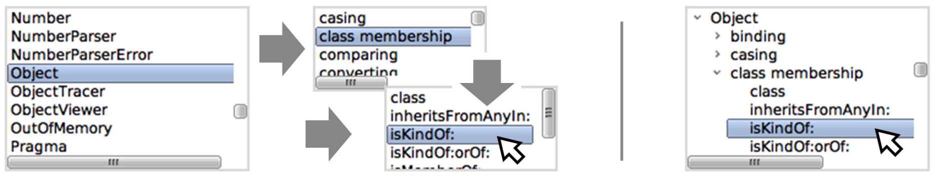
- **Cycle** — One or more panes form a connected, uni-directional cycle. To construct a single script, each pane's script can be appended one after another. Then, the last step refers to the first step to define recursion. For example, a file browser traverses a hierarchy of directories, which can be of arbitrary depth.[4] A single tree can show more context:



- **Split** — One (source) pane provides input for two or more (target) panes. To construct a single script, the target scripts have to be merged and the appended to the source script. Merging two scripts can require *meta programming* if they process two different kinds of objects. In the model tree, the object lists can be concatenated in the same level or separated in two different branches. For example, a class browser can show instance variables and methods in separate list views.[5] A single tree can show such branches more concisely:



- **Merge** — Two or more *ordered* (source) panes provide input for a single (target) pane. To construct a single script, the source scripts can be appended one after another to form levels in the tree. If they process unrelated objects, they have to be adapted to use *tuples*. The target script can be appended *as-is* because it already transforms tuples. For example, a class browser can show methods by category, but categories are symbols and thus unrelated to classes. A single tree can save screen space:



In complex tools, such patterns can occur in more elaborate ways. They can overlap and interleave. Thus, tool builders have to isolate them before they can revise them, which is a common practice for traditional refactorings, too.

[4]See appendix B.6 for the "File Browser" script code.
[5]See appendix B.4 for the similar "Class Outline" script code.

Synopsis Both Vivide and Squeak provide alternatives for certain design choices to increase flexibility in the tool-building process. We identified three kinds of such choices: (1) scripts or classes/methods, (2) panes/views or plain morphs, and (3) script composition or pane composition. Such choices can influence a tool's querying, mapping, and presentation languages. The expressiveness of such languages can then influence the tool's overall utility and usability.

Drawing from our experiences with Vivide and Squeak so far, we document three recurrent practices to guide those design choices. First, programmers should try using Squeak's generic containers such as arrays, associations, and dictionaries before adding new domain classes to the system. Second, programmers should try using morphs to configure generic views, and they should try embedding panes into Morphic applications. Third, programmers should try mixing simple scripts and simple views with complex scripts and complex views. At the end of the day, refactoring should remain an inherent part of (iterative) tool building like it is for object-oriented programming in general.

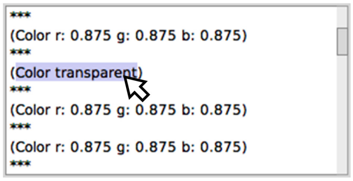
6.3 Beyond Lists: Different Views for Scripts

Typically, we use hierarchical or tabular list views because they directly fit the tree models that Vivide scripts describe. That is, child nodes form the list hierarchy, and similar node properties form the table columns. However, traditional tools can offer widgets that are different to lists such as buttons, text fields, and graphs. Consequently, we experimented with the applicability of scripts to configure such widgets, too. Tool builders (or view designers) should answer the following questions when adapting such different views for Vivide:

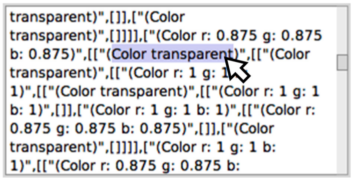
- How far can the view display the model's *node hierarchy*?
- How far can the view be configured through *object properties*?
- How far can the view be configured through *script properties*?

Considering our tool-design strategy (section 4.2), views might not have distinct *graphical shapes* for objects at all, which affects *object selection*, *context menus*, and thus tangibility.

In the following, we explain our experiences with integrating four different non-list views: a text field, a button bar, a polymetric tree view, and a treemap. Each view tackles different challenges: text lacks tangibility, buttons trigger effects on user input, the graphical hierarchy cannot be infinitely deep, and treemaps aggregate node properties. These cases illustrate that Vivide supports the integration of more elaborate visualizations.



```
***
(Color r: 0.875 g: 0.875 b: 0.875)
***
(Color transparent)
***
(Color r: 0.875 g: 0.875 b: 0.875)
***
(Color r: 0.875 g: 0.875 b: 0.875)
***
```



```
transparent)",[]],["(Color
transparent)",[]]]],["(Color r: 0.875 g: 0.875
b: 0.875)",[["(Color transparent)",[["(Color
transparent)",[["(Color r: 1 g: 1
1)",[["(Color transparent)",[["(Color r: 1 g: 1
b: 1)",[]],["(Color r: 1 g: 1 b: 1)",[["(Color r:
0.875 g: 0.875 b: 0.875)",[]],["(Color
transparent)",[]]]],["(Color r: 1 g: 1 b:
1)",[["(Color r: 0.875 g: 0.875 b:
```

Figure 6.7: Text fields can try to preserve artifact integrity via separators (left: "***") or other serialization formats (right: JSON). Yet, it is difficult to select objects with character-based selection.

Text Fields

Text is a powerful way to consume and modify artifact structure with low effort. Programmers can directly hit keys on the keyboard to process text at the level of characters; screens use fonts for display. Now, text fields in the Vivide environment face challenges similar to *structured (or projectional) editors* [90]. That is, users prefer to have a flexible document metaphor *and* sound text content at any time. Yet, the flat character layout and the structured model layout are difficult to synchronize without trade-offs. For example, the text selection can be surprising if it cuts structured elements.

We decided to make the model structure explicit by *serializing* nodes and node properties to get a flat, textual, representation. In particular, there is the object property `#text` in each node for details and the script property `#mode` for the overall serialization format. In figure 6.7, we illustrate an example with two formats: (1) dedicated separating characters and (2) JSON. While JSON can escape text by definition, tool builders have to configure any manual separator so that it does not overlap with the property. Otherwise, the view cannot compute the corresponding node after changes to write, for example, back the `#text` property. Also, lists of objects work well with separators, and hierarchies of objects benefit from JSON, XML, or similar representations.

The idea of serialization applies to all kinds of Smalltalk objects that represent valuable artifact structure. For the example in figure 6.7, we serialize instances of `Color` in a morph hierarchy. The script looks as follows:

```
script := {
   [:morph | morph submorphs]
   -> { #id -> #'1020f29a'.
        #view -> TextField.
        #mode -> #concatenate. "Or #json or #xml or ..."
        #separator -> '***' "Only for #concatenate."}.
   [:morph | {
      #text -> morph color storeString
            <- [:text | morph color: ( Color readFrom: text )] }]
   -> { #next -> #'1020f29a' }
} asScript openScriptWith: someMorphs.
```

Here, Squeak converts between objects and texts with `#storeString` and `#readFrom:`. If the view concatenates nodes with a separator, the first level of morph colors will be displayed. If the view uses JSON, the entire hierarchy of morph colors will be displayed.

Object selection can be challenging for users because text selection consists of characters, which may cut into labels by accident. Still, we think that all views should allow users to select some objects to continue exploration in another view through context menus or connected panes. Fortunately, Smalltalk programmers can associate spans of text with objects: the *bindings*. In Squeak's code editors, programmers can evaluate, print, or inspect expressions. Symbols are bound globally or locally in a debugging context, which makes source code appear concrete and running. Consequently, our text fields can support object selection in two ways: (1) via a node's `#text` property or (2) via a node's `#bindings` like this:

```
[:morph | {
  #text -> 'color'.
  #bindings -> { #color -> morph color } }]
```

Note that if the binding is not included in the text representation, then users first have to type the binding like they would in a Squeak Workspace. Since duplicate bindings are not supported, scripts should append a unique suffix such as an object's `#identityHash`. The evaluation of any Smalltalk expression, which uses bindings, then indicates the selection of objects for VIVIDE.

[|]

We think that all views should support lists of objects or hierarchies of objects. Yet, our *method editors* are text fields that each show only a single method. So, these editors are *single-object tools*, which is similar to traditional Smalltalk tools. We make a list of method editors using additional containers, which are *pane views* in VIVIDE terms. See section 6.1 for more details.

Button Bars

Buttons can trigger actions that are complementary to program comprehension. They can close windows, save changes, or toggle preferences. In VIVIDE, buttons (or button bars) are views, which operate on objects and are configured via scripts. Given that mouse clicks trigger actions, such a view can represent either *objects or actions* as buttons. In figure 6.8, there is a traditional bar with actions and one with objects. An action button will trigger the action for all objects behind it. An object button will trigger the action only for that object. Thus, objects-as-buttons can form *radio buttons* to provide user choices.

There are three dynamically bound variables to support meta programming in scripts: `thisScript`, `thisPane`, and `thisView`. These variables provide access to the current script, pane, and view objects respectively. Tool builders can use them for

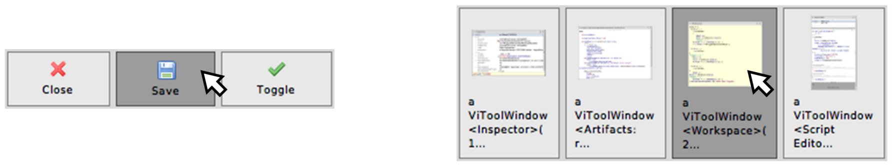

Close
Save
Toggle
a ViToolWindow <Inspector>(1...
a ViToolWindow <Artifacts: r...
a ViToolWindow <Workspace>(2...
a ViToolWindow <Script Edito...

Figure 6.8: In VIVIDE, button bars can either concentrate *m* actions for *n* objects (here: left bar) or offer *one* action triggered on a selected object (here: right bar). Scripts contain the source code for a button's callback.

debugging or to trigger side effects that concern the user interface. A button that closes the window in Squeak can be scripted as follows:

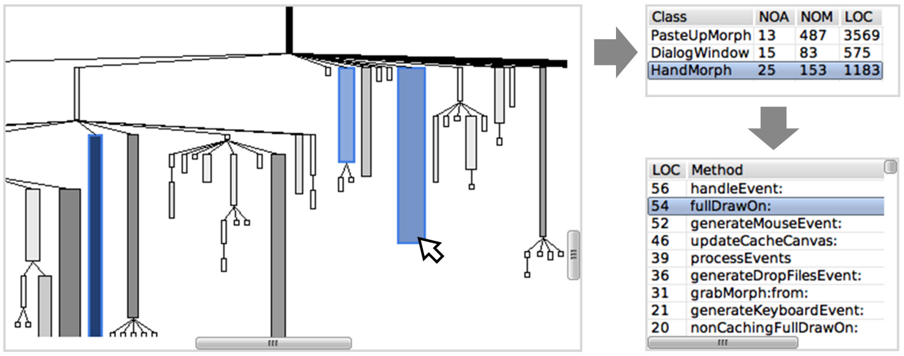
```
script := {
   [:object | | window |
      window := thisPane containingWindow.
      { #text -> 'Close'. "Button label"
         #clicked -> [[ window close ]] "Button callback"}]
   -> { #view -> ButtonBar }
} asScript openScriptWith: #(dummy).
```

Note that we have to bind the temporary variable `window` because `thisPane` will not be bound at the time when the user clicks the button. Plus, we have to provide a placeholder object so that the script interpreter generates a model node the for view to process.

Button-bar views can be used to *intercept* object exchange between other views. For example, Squeak's System Browser has two buttons, namely "instance" and "class", to toggle browsing methods for instances of the current class or for the class itself. From a tool perspective, there are objects for classes and objects for meta-classes, which can both access methods. Buttons can transform classes into meta-classes and *yield* the result via `thisView` as object selection to the view's pane:

```
script := {
   [:class | | view |
      view := thisView .
      { #text -> 'instance'.
         #clicked -> [[ view yield: class theNonMetaClass ]] }]
   -> { #view -> ButtonBar }.
  [:class | | view |
      view := thisView .
      { #text -> 'class'.
         #clicked -> [[ view yield: class theMetaClass ]] }]
} asScript openScriptWith: someClasses.
```

We think that button bars are useful to remain compatible with traditional user interfaces. Yet, such views are neither necessary nor appropriate to build tools in VIVIDE. Scripts transform objects, and users can choose scripts. Buttons as filters feel redundant at this point. Only in the form of *pane decorations* such as halo or window, buttons are convenient to manage the user-interface layout. These thoughts follow directly from our tool-design strategy as described in section 4.2.



Class	NOA	NOM	LOC
PasteUpMorph	13	487	3569
DialogWindow	15	83	575
HandMorph	25	153	1183

LOC	Method
56	handleEvent:
54	fullDrawOn:
52	generateMouseEvent:
46	updateCacheCanvas:
39	processEvents
36	generateDropFilesEvent:
31	grabMorph:from:
21	generateKeyboardEvent:
20	nonCachingFullDrawOn:

Figure 6.9: Our polymetric views always show a tree layout because of the node hierarchy in the model. Here, a system-complexity view [99, p. 35] about classes in the Morphic framework (left) is the input for table views (right) to show details.

Polymetric Views

Polymetric views [99, pp. 33–40] are a good fit for VIVIDE and Morphic. Such views consist of rectangular, connected, colored entities reflecting software metrics. Our scripts can describe tree models with properties added to each node. Morphic constructs graphics with composite morphs, which are rectangles by default. Thus, a *morph* can be a *node* and also an *object*, which is a peak in directness and tangibility for such views. Morphs can implement event handlers for mouse clicks or keyboard hits, which fulfills our criteria for object selection in VIVIDE views.

In figure 6.9, a polymetric view shows metrics for part of the Morphic class hierarchy; it is connected to table views for details. Each morph represents an entity in the polymetric view; submorphs are aligned horizontally with a top offset to leave space for edges. Besides the hierarchy, the script extracts metrics to configure morphs:

```
[:class | {
  #width -> class instVarNames size.
  #height -> class methodDict values size.
  #color -> (Color h: 0 "hue" s: 0 "saturation"
                "value; darker means more LOC"
                v: 1.0 - ((class linesOfCode / 3500) min: 1.0)) }]
```

Note that the maximum value for lines-of-code (LOC) is hard-coded with `3500` due to the exploratory setting. Programmers can change that number easily. Still, the actual maximum for any set of classes can be computed in scripts and passed between steps using *tuples* or other data structures. If the underlying domain model would provide that number directly, the script could be more concise.

Model trees of *infinite depth* require additional support in views. Our scripts are interpreted lazily as views navigate nodes in the model. So, polymetric views assume a finite depth like many visualizations, because they are made for *overview*. The *entire model* has to be visible in some form. This makes polymetric views incompatible for tools that navigate the object graph in cycles such as Squeak's Object Explorer. A

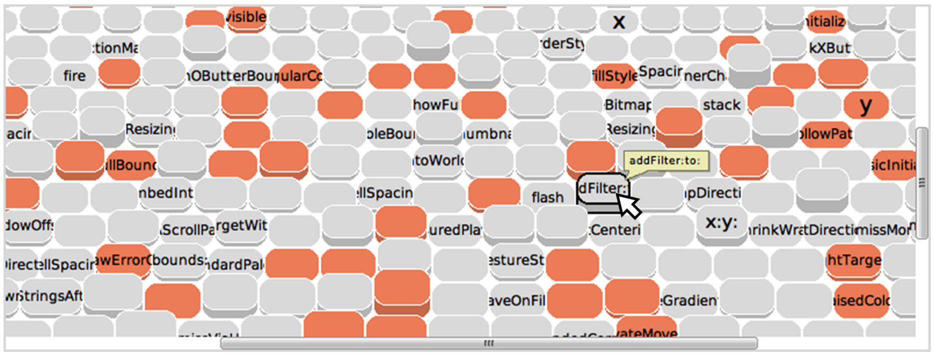


Figure 6.10: This (part of a) treemap view shows methods in classes. The area represents LOC, the color red means access to instance variables, the elevation translates to the number of arguments. Each shape in this hierarchy is an object that can be selected.

possible trade-off would be to delegate this issue to the user. Our list-based hierarchy views, for example, have entities that can be expanded and collapsed manually.

Views that expect many properties from models should account for many *default values*. For text-based views, this is easy because objects have a textual representation via `#printString`, which is at least the class name and an identity hash. This convenience can be found in non-Smalltalk environments, too. As explained in section 4.2, it is beneficial to have more *shared concepts* for objects such as an icon, a summary, or a color. If not, views have to make up their own default values. In polymetric views, missing `#width` or `#height` could be anything `> 0`. A neutral color could be white. In VIVIDE, views should be *forgiving* because scripts should work with any available view.

Treemaps

Treemaps [174] are space-filling views for hierarchical data. They nest rectangles or other polygons [70] with varying layout strategies that improve stability of relative areas and positions over time. Like polymetric views, they have to access the *entire* view model, which makes them incompatible with infinite trees. In figure 6.10, there is an example for a treemap that shows methods in classes, mapping LOC to the area.

Views can add new properties to model nodes such as for caching. Those properties will be discarded automatically when new objects arrive or the script changes because the entire model will be re-constructed. Our treemaps use this storage during layout computation. That is, treemaps use a *weight* for each node to calculate the size of the corresponding shape. Now, scripts do not have to provide that weight for *every* level in the tree but only for leaves. Then, treemaps iterate through the model to *aggregate* the `#weight` property for all inner nodes. This renders scripts for our treemaps more compact like in this example:

```
script := {
   "Level 1 – Packages"
   [:organzer | organizer packages] -> { #view -> Treemap }.
   [:package | { #text -> package name }].
   "Level 2 – Classes"
   [:package | package classes].
   [:class | { #text -> class name }].
   "Level 3 – Methods"
   [:class | class methodDict values].
   [:method | { #text -> method selector.
                #weight -> method linesOfCode }]
} asScript openScriptWith: { PackageOrganizer default }.
```

Only the last extract step has to consider the `#weight` property for this view, which methods provide directly via `#linesOfCode`. The LOC value for each class or package can be derived. There is no need to change the domain objects (here: packages or classes) to provide that value. At the script level, such aggregation would be challenging—if not infeasible—because script steps are only evaluated in one direction without access to future information.

Synopsis We claim that Vivide scripts can describe tree models for views to support exploratory program comprehension (and modification). We also claim that interactive views should be able to let users select objects. Comparing with traditional tools, such views are often *hierarchical or tabular lists*. However, there are many other widgets for visualizations that can help programmers explore complex problem domains.

To evaluate the applicability of scripts for non-list views, we exemplify the use of text fields, button bars, polymetric views, and treemaps. On the upside, view designers can create or adapt views for Vivide scripts and benefit from a flexible view model. They can anticipate *object properties* for visual mapping and *script properties* for general preferences. On the downside, infinitely deep model trees are often not supported. Buttons seem inappropriate for data-driven tool building. Text fields have difficulties in object selection. Treemaps have to aggregate data from leaf nodes up because scripts can only define models the other way around.

Summary

During the design and implementation of Vivide, we learned many lessons about tool building in a pure, object-oriented programming system such as Squeak/Smalltalk. At some point, we could "eat our own dog food." We wrote scripts that process Squeak's code artifacts to present them in list views and text fields to read and write Smalltalk. Step by step, our perspective on tool-supported programming changed from *tools-first* to *artifacts-first*. The script-based, data-driven approach offered through Vivide turned out to make many of Squeak's traditional programming tools obsolete. Yet, there are some tools whose design cannot be mapped directly to our notion of script and pane composition. On the one hand, such tools could benefit from re-design toward Vivide concepts. On the other hand, such tools could guide the refinement of Vivide itself in the future.

The design of *adaptable programming tools* with scripts and panes offers a more modular presentation language. The traditional System Browser becomes two kinds of tools: a *class outline* and a list of *method editors*. This way, programmers can collect and juxtapose related code artifacts on screen without accidental redundancy. A minimal set of *complementary scripts* then covers all relevant code-browsing activities such as senders/implementors of symbols and object exploration. There are also scripts to begin programming sessions such as via text search and code workspaces. Such a focus on *code artifacts* yields first thoughts about *script re-use* and the embedding of simple *software metrics* to support programmers.

There are many design choices to make when applying scripts and panes, which may need revisions. Comparable to *code refactoring*, we think that tool builders have to make trade-offs between Vivide scripts and Smalltalk messaging, Vivide panes and Morphic morphs, and script/pane compositions. We think that exploratory tool building should make use of Squeak's generic containers such as `Array`, `Association`, and `Dictionary` before starting to implement custom domain classes. Also, morphs are regular objects that can be stored in model nodes to simplify the implementation of views. And script steps exchange objects in a similar way panes do, which makes both concepts interchangeable.

Finally, there are ways other than hierarchical or tabular list views to form a tool's presentation language. For example, there are widgets such as button bars or visualizations such as polymetric views. We think that the view models described through Vivide scripts can supply information for such non-list views, too. However, there are limitations such as infinitely deep models, which are often not supported. Also, any aggregation of properties from leaf nodes up to a model's root has to be handled by views. Script interpretation only happens the other way around.

[|]

In the next chapter, we continue evaluating and discussing our solution. That is, we describe several case studies students carried out using Vivide in the course of project seminars or their master's theses.

7 Case Studies

We applied Vivide to improve the programming tools we use on a daily basis. As explained in chapter 6, we changed code browsers, added new visualizations, and found replacements for all kinds of software artifacts in the Squeak/Smalltalk environment. We also explored language extensions such as *context-oriented programming* [191] to evaluate the applicability of our tool-design strategy.

However, we complement our evaluation with projects that we did not carry out personally. There were several *case studies* where we observed and guided *graduate students* in various project seminars. Our goal was to better understand the thoughts and strategies of people new to Vivide. Naturally, we controlled their activities to build tools in that specific way, ignoring alternatives. Note that in all these projects, tool building was secondary. The challenging problem domains, each having new kinds of software artifacts, were of primary concern.

In the following, we present four case studies situated in four different programming domains: version control, live language development, module systems, and multi-language debugging. In each study, we present *figures* and *summaries* to explain project background, results, and efforts. Considering the evaluation of Vivide, we elaborate on *script usage* and *lessons learned*.

Remark For each case study, we present a *facts sheet* that summarizes topic, scope, duration, participants, and expected performance (or effort). Additionally, we approximate the *amount of information* available by naming the particular Squeak version and lines-of-code (LOC) involved, which we present with three numbers: (1) LOC changed by the participants, (2) LOC available for reading to understand problem/solution space, and (3) LOC of the provided system at that time. Those numbers should approximate the level of difficulty in each project.

7.1 Thresher: Grouping Micro Changes

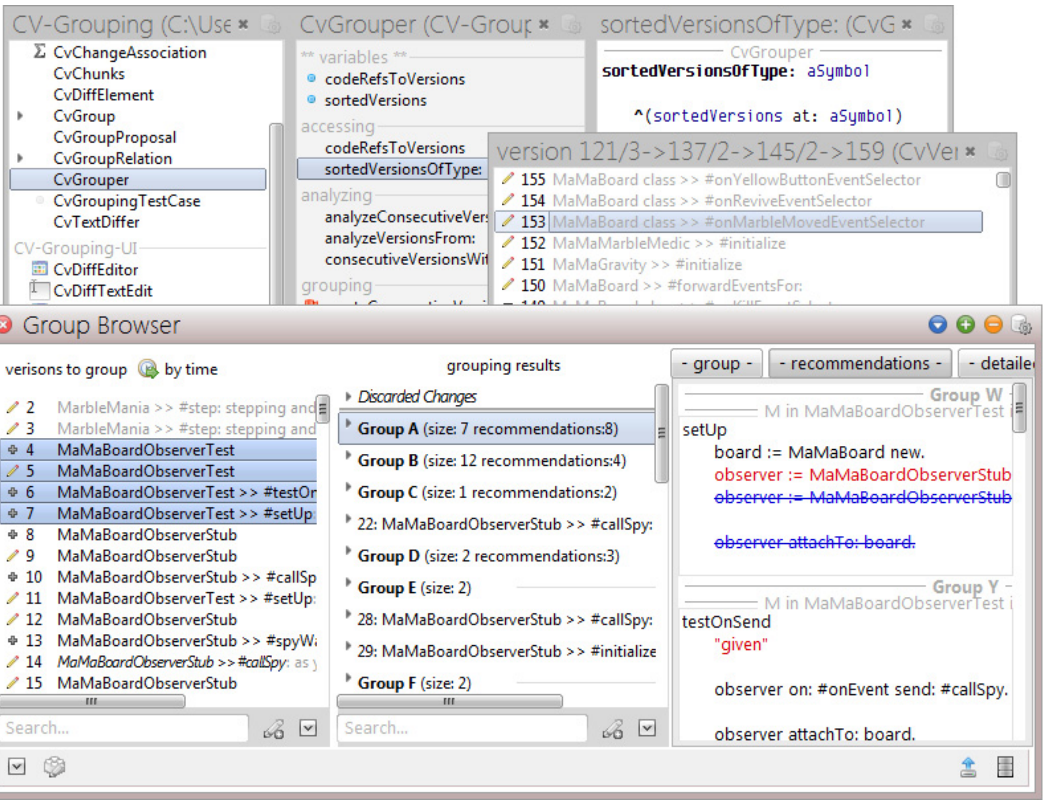
Topic	Design and implement tools to help programmers organize fine-granular code changes.
Scope	Software Architecture Group (Master's Thesis)
Duration	6 months (prepare) + 6 months (write up)
Participants	1 female graduate student, computer science
Performance	20+ hours per person per week
Squeak	4.4
# LOC	45'354 / 77'063 / 582'735
% LOC	7.8 / 12.2 / 100.0

Remark The master's thesis of a graduate student is usually related to an existing *research project* from our research group. At that point, the student has about 2 years of experience in Squeak/Smalltalk due to lectures and project seminars. In the thesis' scope, the student tries to solve a challenge in a way that combines methods from *engineering* and *science*. That is, the student applies programming skills but also investigates related work to better understand problem domain and solution space. The measure of success is usually support of a certain scenario, sometimes supplemented with a user study.

Project Summary

There are situations where programmers leave their current task context to do related work. Examples include quick bug fixes and code clean-up. Even if such an *exploratory style* is anticipated, best practices suggest to group and document such changes separately. Code repositories cluttered with incoherent version information impede program comprehension. There are approaches that try to *untangle* version histories later. Yet, it makes sense to include the programmer's tacit knowledge directly in the code-commit phase.

The results of this project are illustrated in figure 7.1. The student analyzed several traces of fine-granular source code changes, which were collected during refactoring sessions from another experiment. In particular, the student grouped changes at the level of (side) tasks *by hand* to infer patterns for *automated* analysis. Those patterns influenced the design and implementation of a new tool called "Group Browser", which should guide programmers during the usual commit phase. We documented and published the results in a paper [195].



```
CV-Grouping (C:\Use
Σ CvChangeAssociation
  CvChunks
  CvDiffElement
▸ CvGroup
  CvGroupProposal
▸ CvGroupRelation
  CvGrouper
  CvGroupingTestCase
  CvTextDiffer
CV-Grouping-UI
  CvDiffEditor
  CvDiffTextEdit

CvGrouper (CV-Grou
** variables **
  codeRefsToVersions
  sortedVersions
accessing
  codeRefsToVersions
  sortedVersionsOfType:
analyzing
  analyzeConsecutiveVer
  analyzeVersionsFrom:
  consecutiveVersionsWi
grouping

sortedVersionsOfType: (CvG
CvGrouper
sortedVersionsOfType: aSymbol

    ^(sortedVersions at: aSymbol)

version 121/3->137/2->145/2->159 (CvVe
155 MaMaBoard class >> #onYellowButtonEventSelector
154 MaMaBoard class >> #onReviveEventSelector
153 MaMaBoard class >> #onMarbleMovedEventSelector
152 MaMaMarbleMedic >> #initialize
151 MaMaGravity >> #initialize
150 MaMaBoard >> #forwardEventsFor:
```

```
Group Browser

verisons to group   by time
 2  MarbleMania >> #step: stepping and
 3  MarbleMania >> #step: stepping and
 4  MaMaBoardObserverTest
 5  MaMaBoardObserverTest
 6  MaMaBoardObserverTest >> #testOr
 7  MaMaBoardObserverTest >> #setUp:
 8  MaMaBoardObserverStub
 9  MaMaBoardObserverStub
10  MaMaBoardObserverStub >> #callSp
11  MaMaBoardObserverTest >> #setUp:
12  MaMaBoardObserverStub
13  MaMaBoardObserverStub >> #spyWi
14  MaMaBoardObserverStub >> #callSpy: as
15  MaMaBoardObserverStub
Search...

grouping results
▸ Discarded Changes
▸ Group A (size: 7 recommendations:8)
▸ Group B (size: 12 recommendations:4)
▸ Group C (size: 1 recommendations:2)
▸ 22: MaMaBoardObserverStub >> #callSpy:
▸ Group D (size: 2 recommendations:3)
▸ Group E (size: 2)
▸ 28: MaMaBoardObserverStub >> #callSpy:
▸ 29: MaMaBoardObserverStub >> #initialize
▸ Group F (size: 2)
Search...

- group -   - recommendations -   - detaile
Group W
M in MaMaBoardObserverTest
setUp
    board := MaMaBoard new.
    observer := MaMaBoardObserverStub
    ~~observer := MaMaBoardObserverStub~~

    ~~observer attachTo: board.~~
Group Y
M in MaMaBoardObserverTest
testOnSend
    "given"

    observer on: #onEvent send: #callSpy.

    observer attachTo: board.
```

Figure 7.1: This tool helps programmers select and group fine-granular changes to form coherent and complete sets for code repositories. Vivide scripts represent different grouping criteria (here: "by time"). Panes and object connections are hard-coded in a Morphic application (here: "Group Browser"). The windows in the back show Vivide scripts used to browse code artifacts and version artifacts.

Script Usage

At the end of the project, there were 18 domain-specific scripts with 140 steps in total and 8 steps per script on average. All scripts summed up to 900 lines of code, which averages to 50 lines spread across each script's steps. At that time, however, the kind of code re-use among scripts was different. Programmers did not read `#next -> #someScriptId` but the entire referenced code *inlined* in the same script. Considering this, there were 361 lines of code when counting only *unique* script steps. Most re-use happened for steps that *sort* by domain-specific properties.

The longest script was an *extract step* that constructed rich text labels and morphs with icons and buttons for the view. The student did write much code to extract the required information from the domain artifacts behind `group`, `groupRelation`, and `tmp`:

```
{ #text -> ( group version number asString, ': ',
             '<font color="#AAAAAA>',
             group version systemEvent cvItemReference asString,
             '</font>') asHtmlText.
  #frontMorph ->
            ((( tmp contents at: #subgroups ) includes: group)
               ifTrue: [ nil ]
               ifFalse: [ CvGroup
                  actionButtonsForRecommendation: groupRelation
                  in: ( tmp contents at: #currentGroup ) ]).
  #backMorph -> morph }
```

This excerpt from the actual script illustrates that model nodes can hold any kind of object to support views. The property `#frontMorph` holds button morphs, the property `#backMorph` holds icon morphs. Both will be embedded in the list view as shown in figure 7.1.

Many scripts in this project *modified* the objects they processed. Since users cannot control script evaluation directly, the student provided *copies of objects* to scope the side effects. A custom caching strategy for performance was the main reason for this design decision.

Lessons Learned

In the course of this project, we learned many things that influenced many design decisions. Since then, Vivide has changed in many aspects.

First and foremost, scripts needed to be simpler. Back then, there were more kinds of steps than *transform* and *extract*. The idea was to reduce the amount of glue code to write by having so many kinds. There were steps that *transform each*, *transform all*, *sort pairwise*, *group*, *group sort pairwise*, and *extract*. Any script could use an extra *context object*, which provided access to previous results. Eventually, we realized that the average Smalltalk programmer would have less issues with uniform `[:in :out | ]`-blocks.

Object properties can be specific to a kind of view. The extraction of `#frontMorph` and `#backMorph` targeted a custom version of list view. This is different to `#text` or `#icon`, which attach a re-usable *role* to the representation of information. We think that this could be an effect of storing morphs as properties in models. Morphs could be perceived as an implementation detail rather than a conceptual one.

The scripts we found were rather long and complex. Programmers can write code that actually belongs to the domain artifacts or the view artifacts. Refactoring of scripts is necessary, especially if they are used for multiple programming sessions or in an application.

In this project, panes were hard-wired in a custom application window as regular object-oriented source code. We learned how make the visual part more flexible in terms of combination and abstraction. Eventually, we added *pane views* and *object connections* to the Vivide mechanics.

Finally, we observed that our replacements for Squeak's code browsers work well. Method lists can hold related code fragments, which are otherwise spread across classes. Yet, overlapping, loosely-coupled windows are not always the best choice. Views similar to Smalltalk's four-pane browser can help explore many artifacts one after another.

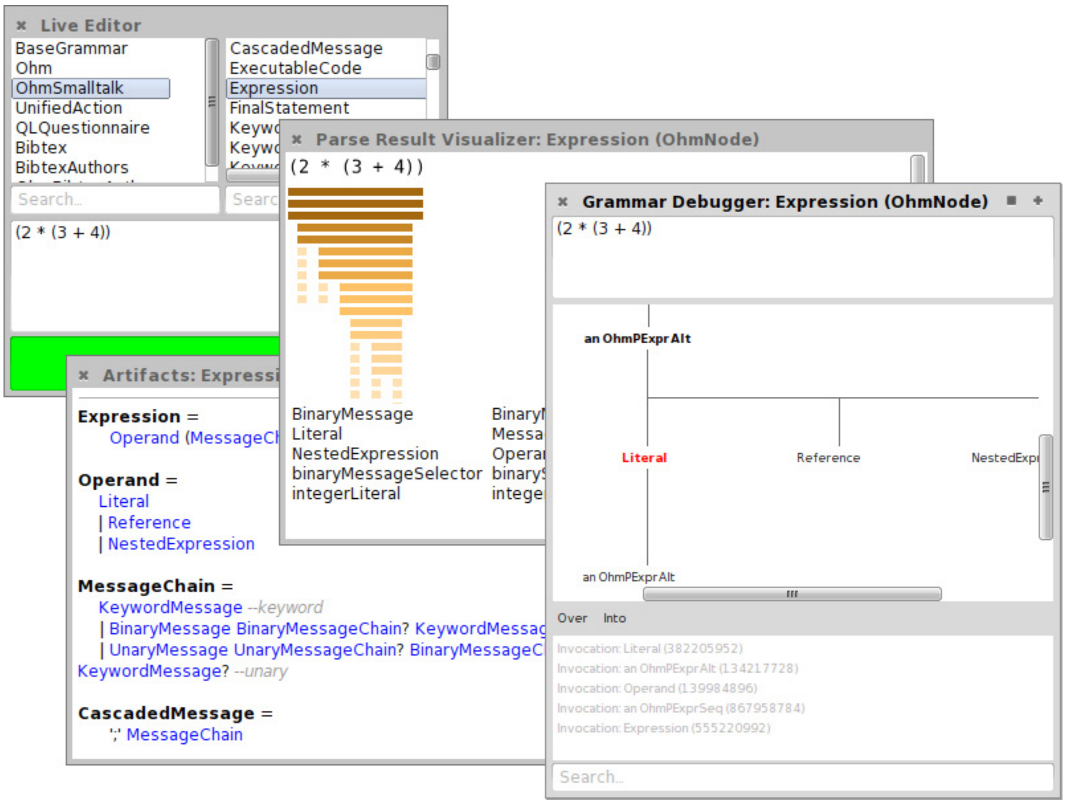
7.2 Gramada: Live Language Development

Topic	Design and implement tools to help programmers create domain-specific languages.
Scope	Software Architecture Group (Master's Thesis)
Duration	6 months (prepare) + 6 months (write up)
Participants	1 male graduate student, computer science
Performance	20+ hours per person per week
Squeak	4.6
# LOC	8'376 / 74'106 / 590'522
% LOC	1.4 / 12.5 / 100.0

Remark The master's thesis of a graduate student is usually related to an existing *research project* from our research group. At that point, the student has about 2 years of experience in Squeak/Smalltalk due to lectures and project seminars. In the thesis' scope, the student tries to solve a challenge in a way that combines methods from *engineering* and *science*. That is, the student applies programming skills but also investigates related work to better understand problem domain and solution space. The measure of success is usually support of a certain scenario, sometimes supplemented with a user study.

Project Summary

Domain-specific languages (DSLs) help programmers express application logic in an abstract, declarative way, which produces source code that is more concise and readable compared to general-purpose languages. However, the creation of such DSLs takes time because design and implementation of syntax and semantics often lacks tool support. That is, feedback loops are usually long, and debugging is difficult. Luckily, there is a (meta) language called *Ohm*, which simplifies the creation of *parsing expression grammars*. In the project, the liveness and directness



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

```
Expression =
   Operand (MessageCh

Operand =
   Literal
   | Reference
   | NestedExpression

MessageChain =
   KeywordMessage --keyword
   | BinaryMessage BinaryMessageChain? KeywordMessag
   | UnaryMessage UnaryMessageChain? BinaryMessageC
KeywordMessage? --unary

CascadedMessage =
   ';' MessageChain
```

```
Invocation: Literal (382205952)
Invocation: an OhmPExprAlt (134217728)
Invocation: Operand (139984896)
Invocation: an OhmPExprSeq (867958784)
Invocation: Expression (555220992)
```

Figure 7.2: Gramada is a tool suite that supports the development of domain-specific languages. The suite features a live editor, a visualizer for parse trees, a debugger for parse trees, and custom code editors for grammar rules. All tools are created with VIVIDE scripts and pane compositions, integrated via object connections.

of Squeak/Smalltalk should complement this experience with additional tools for such a language (framework).

The results of this project are illustrated in figure 7.2. First, the student implemented *Ohm/S*, which is Ohm for Squeak/Smalltalk, extended with strategies to reduce compilation time. Then, tools for creating, changing, and debugging grammar rules improved liveness and directness. There is a live editor, a parse-result visualizer, a custom debugger, and a custom code editor. Many exemplary grammars where created in the process such as one for Smalltalk and one for Bibtex. We documented and published the results in a paper [154].

Script Usage

At the end of the project, there were 20 domain-specific scripts with 54 steps in total and 3 steps per script on average. All scripts summed up to 372 lines of code, which averages to 18.6 lines spread across each script's steps. The student did only write new scripts for Ohm artifacts and the new tools. Smalltalk (code) artifacts were treated entirely with Squeak's traditional tools.

The longest script step is also a very interesting one because it disagrees with our known script practices so far. The button bar in the Grammar Debugger (figure 7.2) takes an instance of `OhmNode` to create an instance of `OhmDebugger`, which then flows to the upper views in that tool. Usually, we would implement a debugger (view) on an object that represents code-in-execution such as Squeak's `MethodContext` and `Process`. In Gramada, however, debugging means rather *trace exploration* because grammar parsing finished (or failed) at that point already. The actions 'step over' and 'step into' traverse this trace node by node. We would summarize the script as follows:

```
script := {
   [:ohmNode | OhmDebugger on: ohmNode]
      -> { #view -> ButtonBar }.
   [:debugger | { #text -> 'over'.
                  #clicked -> [[ debugger over ]] }].
   [:debugger | { #text -> 'into'.
                  #clicked -> [[ debugger into ]] }].
} asScript openScriptWith: { '(2 * (3 + 4))' parse }.
```

This means that the Grammar Debugger *resets* if the user re-interprets the script behind the button bar. From a tool-building perspective, we cannot assess whether this is a convenience or rather a nuisance of Vivide. This application shows that actions (or side effects) in Vivide are challenging. We think that button bars should be avoided if tools just support the exploration of artifact structure, which is accessible behind `ohmNode` in this case.

Lessons Learned

In the course of this project, we learned three things: (1) Squeak's tools are still preferred for code writing, (2) ad-hoc integration via object connections works, and (3) script identifiers and labels reveal the user's viewpoint.

First, the student kept on using Squeak's code browsers. Vivide was perceived as the new approach to build new tools, not to replace existing ones. We think that the different working practice in Vivide requires much training. Consequently, well-known activities such as "create a class" and "change a method" demand for well-known tools. Yet, the student created single-object tools for *grammar rules* as depicted in figure 7.2. Such rules are like methods, and Vivide arranges all single-object tools in *artifact lists*.

Second, the student integrated grammar tools via *object connections* in demo sessions. Starting with the Live Editor to select a certain grammar and rule, examples either passed (green) or not (red). That tool's output is the parse tree, which can be connected to the Parse Result Visualizer or Grammar Debugger as described above. We observed a strong focus on domain artifacts and not so much on the tools behind them.

Finally, we found a *prefix* in script identifiers: "ohm". The terms "rule", "debugger", or "match" have a well-understood meaning in this domain. Yet, they overlap, for example, with concepts that already exist in the Squeak/Smalltalk system. So, identifiers such as `#ohmRuleActionsNormal` and `ohmDebuggerMultiPane` clarify that meaning. In general, the student's naming approach is interesting in the sense that scripts are typically "viewers", "visualizers", "browsers", or "editors". This indicates a *tool-driven* rather than *data-driven* perspective.

7.3 Matriona: A Module System for Squeak

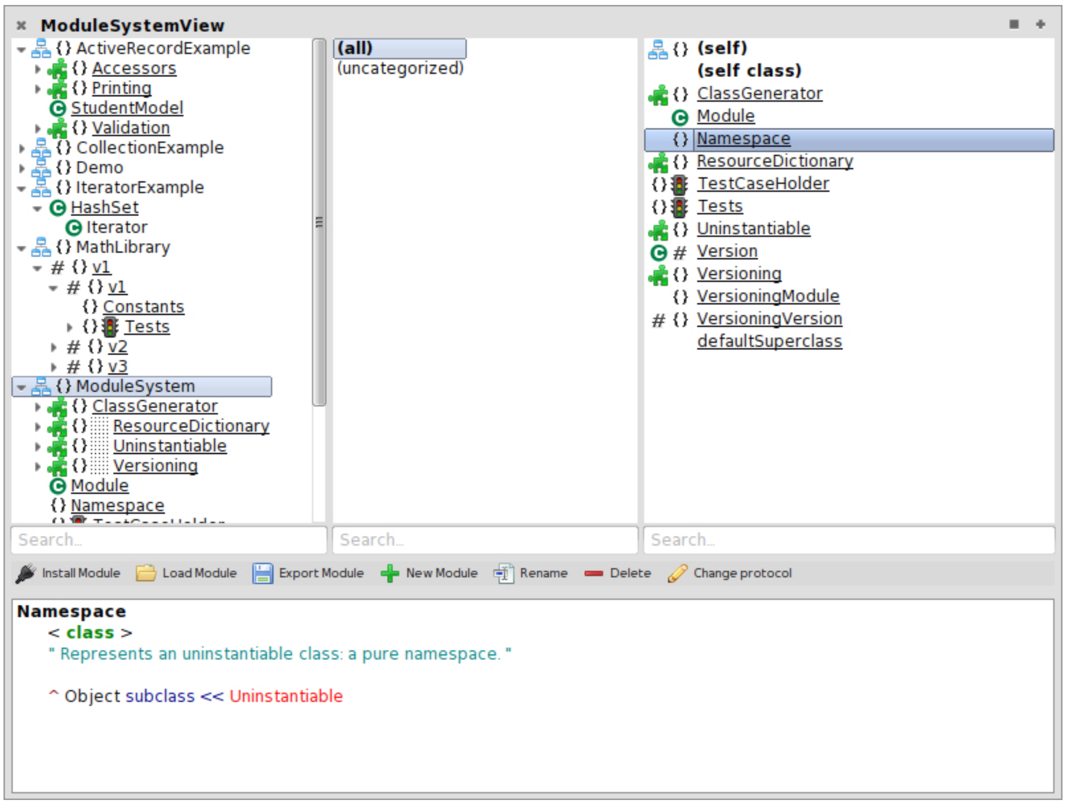
Topic	Design and implement class nesting with parametrization in Squeak/Smalltalk.
Scope	Module Systems (Project Seminar)
Duration	4 months
Participants	2 male graduate students, computer science
Performance	6+ hours per person per week
Squeak	4.5
# LOC	7'411 / 69'556 / 573'666
% LOC	1.3 / 12.1 / 100.0

Remark In project seminars, graduate students learn about an existing *research project* from our research group. At that point, they have about 1-2 years of experience in Squeak/Smalltalk. They try to solve a challenge in that project's domain in a way that combines methods from *engineering* and *science*. That is, they apply their programming skills, but they also investigate related work to better understand problem and solution space. The measure of success is usually the support of a specific scenario, sometimes also a small user study.

Project Summary

Squeak has a single namespace for all classes in the system. There are class-name prefixes to name generic concepts in specific libraries such as `FilePackage`, `MCPackage`, `SMPackage`, and `PackageInfo`. However, this approach impairs *modularity* because prefixes can still conflict if not organized globally. For Smalltalk-like languages, there are module systems such as Newspeak's Nested Classes [20], which add lexical scope during lookup.

The results of this project are illustrated in figure 7.3. The students implemented a new module system for Squeak/Smalltalk, called *Matriona*. They employ *classes as namespaces* to foster understanding through hierarchical decomposition, code re-use,



```
Namespace
    < class >
    " Represents an uninstantiable class: a pure namespace. "

    ^ Object subclass << Uninstantiable
```

Figure 7.3: Tool support for the module system *Matriona* is implemented with Vivide scripts and panes. This browser resembles Smalltalk's four-pane browser with navigation panes in the upper part and a code editor in the lower part. A button bar offers operations on modules such as "install module" and "export module".

and class versions. The balance between Squeak's (code) objects and Smalltalk's messaging suggests a compatible and integrated solution for existing projects and programmers. The students documented and published the results in a paper [184].

Script Usage

At the end of the project seminar, there were 6 domain-specific scripts with 30 steps in total and 5 steps per script on average. All scripts summed up to 162 lines of code, which averages to 27 lines spread across each script's steps. The students did only write new scripts for the new Module System View. Smalltalk (code) artifacts were treated either via existing Squeak tools or default Vivide scripts.

The longest script steps define the button bar as depicted in figure 7.3. In addition to the usual `#text` and `#icon` properties, the students implemented an interactive callback behind `#clicked`. An excerpt of that extract step looks as follows:

```
#clicked -> [[ | newName |
  newName := UIManager default request: 'Specify new name'.
  newName isEmpty ifFalse: [
    memberSpecification fullRename: newName.
    ViEventNotifier trigger: #ModuleSystemModuleView ]]
```

The default `UIManager` will present a dialog where the user can enter a new name, which then triggers an effect in the module system. Note that the named event `#ModuleSystemModuleView` is used to update the upper-left view in the browser. We suggested this kind of use for *event notifiers* because it keeps such side effects close to the tool implementation and not hidden in domain code. Alternatively, domain artifacts could provide their own signals to inform other objects about changes in their structure.

Lessons Learned

In the course of this project, we learned three things: (1) specifications for Vivide tools are different, (2) a few lines can form Smalltalk browsers, and (3) pane composition needs improvement.

First, we changed our perspective from "Browsers exchange objects anyway so make it explicit and adaptable." to "If we see browsers as [...], then scripts can help [...]." Our notion of *data-driven views* seems novel because the students did not think about programming tools that way. Instead, they always had to learn what information in the tool interface meant and how that information relates to their goals. We had to explain and tutor to make them see and think about the *artifacts first* when designing new tools.

Second, the students needed less than 100 lines of code to design a new code browser. Squeak's browser consists of more than 3000 lines, which renders adaptation challenging. In this project, script steps were often shorter than 5 lines, which underlines the modular design of our scripting language.

Finally, we discovered several issues with *pane compositions* and layouting. The interactive combination of panes and object connections demands for *robustness* known from other UI-design tools.

7.4 QoppaS: Multi-language Debugging

Topic	Design and implement a debugger for interpreter developers.
Scope	Programming Languages: Concepts, Tools, Environments (Project Seminar)
Duration	5 months
Participants	3 male graduate students, computer science
Performance	6+ hours per person per week
Squeak	5.0
# LOC	3'987 / 55'088 / 570'633
% LOC	0.7 / 9.7 / 100.0

Remark In project seminars, graduate students learn about an existing *research project* from our research group. At that point, they have about 1-2 years of experience in Squeak/Smalltalk. They try to solve a challenge in that project's domain in a way that combines methods from *engineering* and *science*. That is, they apply their programming skills, but they also investigate related work to better understand problem and solution space. The measure of success is usually the support of a specific scenario, sometimes also a small user study.

Project Summary

The design and implementation of *interpreters* for higher-level languages can be challenging because generic programming tools are typically unaware of domain-specific, high-level concepts. One category of such tools are *debuggers*, which do only operate at the level of *host languages* considering its artifacts and operations for control flow. Consequently, there is a need for tools that support new *guest languages*, being aware the interpreters' new abstraction levels. In particular, this entails a joint understanding of *stack frames*, *program counters*, and clarification of *step-into/over/out* operations.

The results of this project are illustrated in figure 7.4. The students implemented *QoppaS*, which is a Scheme/Lisp dialect for Squeak/Smalltalk. Then, they designed a new *multi-level debugger*, which separates (lower-level) Smalltalk messages from (higher-level) Qoppa lists. The main challenge was to find a useful trade-off for the traditional *stack list*. Eventually, that list view became a tree view. The overall problem can be transferred to the development of libraries, frameworks, and language extensions. The students documented and published the results in a paper [96].

Script Usage

At the end of the project seminar, there were 38 domain-specific scripts with 106 steps in total and 3 steps per script on average. All scripts summed up to 487 lines of code, which averages to 12.8 lines spread across each script's steps. The students did only write new scripts for QoppaS artifacts and the new debugger. Smalltalk (code) artifacts were treated either via existing Squeak tools or default VIVIDE scripts.

The longest script step looks as follows:

```
[:in :out |
  [:objects | | collectedNodes |
    collectedNodes := OrderedCollection new.
      objects do: [:node |
        node contextPart method == (QoppaSexprSemantic >> #eval:in:)
          ifTrue: [ | qoppaNode expression |
            qoppaNode := VirtualContextNode new.
            expression := node contextPart arguments first.
            qoppaNode summary:
               'Qoppa evaluation of ', expression qoppaPrintString.
            qoppaNode definitionContent:
               (Text fromString: expression qoppaPrintString).
            collectedNodes do: [:each | qoppaNode addChild: each].
            collectedNodes removeAll.
            qoppaNode addChild: node.
            out add: qoppaNode  ]
          ifFalse: [ collectedNodes add: node ]].
   out addAll: collectedNodes ] value: in]
```

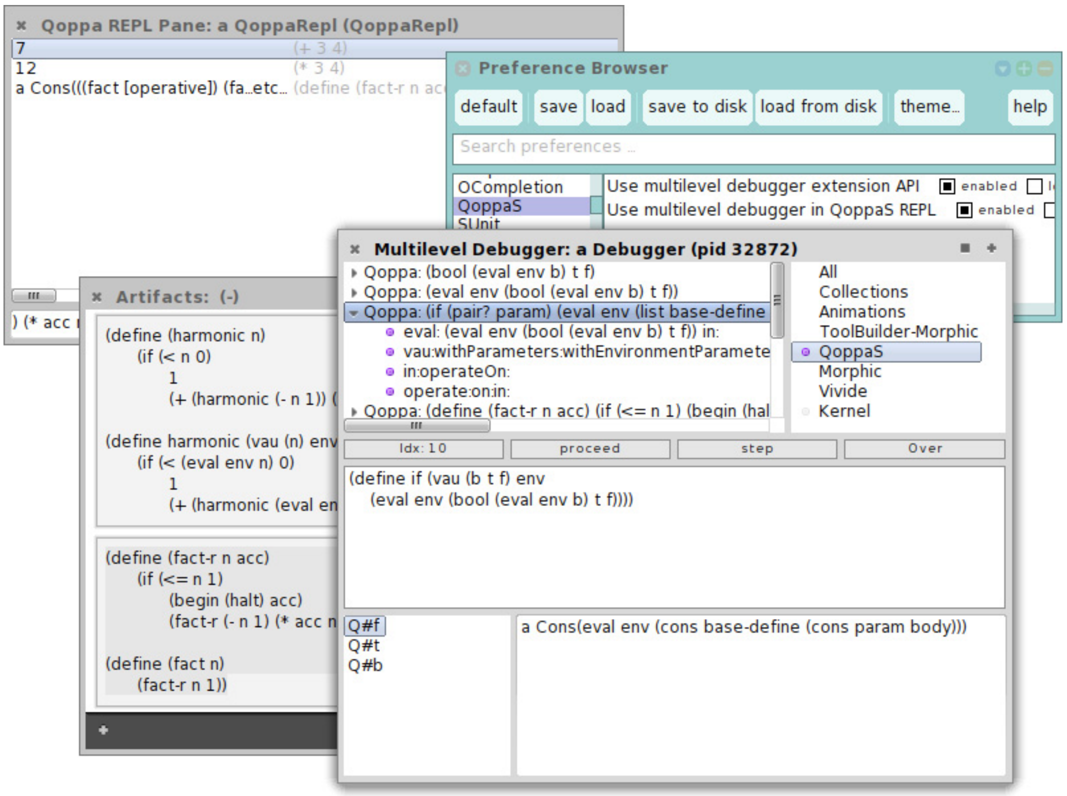
We think that such scripts indicate missing structure in domain artifacts. The relationship between `node`, `qoppaNode`, and `expression` could be encapsulated outside script code to improve readability and re-use.

Lessons Learned

In the course of this project, we learned three things: (1) script identifiers are actually used to invoke VIVIDE tools from object-oriented code, (2) collections should not be used to represent domain artifacts, and (3) script serialization and sharing should be improved.

First, we found several script identifiers that either fostered re-use among scripts or tool invocation in object-oriented code. Those identifiers were `#ContextTree`, `#QoppaOperativeSource`, `#multilevelDebugger`, and `#qoppaReplPane`. They all capture relevant vocabulary from the project domain. They are always similar to the script's `#label` such as "Qoppa REPL Pane" for `#qoppaReplPane`.

Second, instances of `Array` seemed a good fit to directly represent Qoppa/Lisp expressions as Smalltalk objects. Unfortunately, VIVIDE treats collections as containers for actual domain artifacts. That is, scripts work on lists of objects, single objects are automatically wrapped via `#asList`, and tuples interfere with Lisp lists.



Qoppa REPL Pane: a QoppaRepl (QoppaRepl)

```
7                                          (+ 3 4)
12                                         (* 3 4)
a Cons(((fact [operative]) (fa..etc...     (define (fact-r n ac
) (* acc
```

Preference Browser

```
OCompletion   Use multilevel debugger extension API   ■ enabled
QoppaS        Use multilevel debugger in QoppaS REPL  ■ enabled
SUnit
```

Artifacts: (-)

```
(define (harmonic n)
    (if (< n 0)
        1
        (+ (harmonic (- n 1)) (

(define harmonic (vau (n) env
    (if (< (eval env n) 0)
        1
        (+ (harmonic (eval en

(define (fact-r n acc)
    (if (<= n 1)
        (begin (halt) acc)
        (fact-r (- n 1) (* acc n

(define (fact n)
    (fact-r n 1))
```

Multilevel Debugger: a Debugger (pid 32872)

```
▸ Qoppa: (bool (eval env b) t f)
▸ Qoppa: (eval env (bool (eval env b) t f))
▾ Qoppa: (if (pair? param) (eval env (list base-define
    eval: (eval env (bool (eval env b) t f)) in:
    vau:withParameters:withEnvironmentParamete
    in:operateOn:
    operate:on:in:
▸ Qoppa: (define (fact-r n acc) (if (<= n 1) (begin (hal
```

```
All
Collections
Animations
ToolBuilder-Morphic
QoppaS
Morphic
Vivide
Kernel
```

```
(define if (vau (b t f) env
  (eval env (bool (eval env b) t f))))
```

```
Q#f    a Cons(eval env (cons base-define (cons param body)))
Q#t
Q#b
```

Figure 7.4: This debugging tool is implemented with Vivide scripts. The user can organize stack frames to separate abstraction levels. *Qoppa* is an implementation of Scheme/Lisp for Smalltalk. The abstractions are reflected through packages such as "QoppaS" and "Morphic". A higher-level frame can correspond to multiple lower-level frames.

Consequently, Qoppa had to introduce custom domain classes, not subclassing from `Array` or similar.

Finally, version control and code sharing for scripts (or Vivide tools) got relevant, which revealed some issues. The students did not build programming tools to *support* their project. Instead, tool building *was* their project, which corresponds to regular application development. So, Vivide scripts were not considering throw-away prototypes but valuable, lasting project artifacts. At that point, scripts could be serialized into regular Squeak methods, which already supported version control. Additionally, we had to improve the script's serialization format to support proper comparisons at the text level, which also improved team communication.

Summary

The most useful insight we got repeatedly from every case study is related to the *understanding* and *application* of Vivide concepts. It took time until students could write scripts and set up pane compositions as we would do. To some extent, this is related to the underlying challenge of *live programming* in pure object-oriented systems such as Squeak/Smalltalk. Programmers have to learn to let go of the text-driven, file-centered way of programming, which one follows, for example, with Java and Eclipse. Such reflection and altering of the own programming knowledge and experience takes several projects to learn from.

However, we could also identify and correct issues that were inherently present in some Vivide concepts. At some point, the kinds of script steps were plentiful because we were looking for good trade-offs between user code and framework "magic". The case studies helped find redundant flexibility to clarify the elements of our scripting language (section 4.1) and UI-design language (section 4.3).

[|]

In the next chapter, we finish evaluating and discussing our solution. That is, we describe a controlled experiment, explain insights we collected from pilot runs, and sketch a possible workshop to tackle the challenge of training.

8 Toward Measuring Effectiveness

We claim that Vivide is better than traditional tool-building frameworks because programmers can iterate faster and save time. In our thesis statement (section 1.3), we use the wordings "...can be expressed in..." and "...will be easy to modify..." to frame the impact of our approach. On the one hand, programmers should be able to *build* generic tools. On the other hand, programmers should be supported to *quickly adapt* those tools to accommodate specific domains and tasks. To evaluate our ideas, we split up our statement into two parts:

Applicability - Yes or no? Programmers can apply Vivide concept **C** in software project **P**.

Effectiveness - How much? The time to modify tool **T** using Vivide is lower compared to using framework **F**.

Thus, the notion of being "better" translates to assessing *comparability* and making a (qualitative or quantitative) *comparison*. First, **yes**, we can apply Vivide, which we describe via many (internal) practices in chapter 6 and many (external) case studies in chapter 7. Both scripting language (section 4.1) and UI-design language (section 4.3) turned out to be useful. In this chapter, we approach the second part of the thesis statement: how much faster is tool building in Vivide?

At the time of writing, we have no conclusive answer to the question of effectiveness. Yet, we designed a *controlled experiment* and conducted first *pilot runs* to assess the feasibility of such an endeavor. In our experimental design, we followed *best practices* known from experimentation for software engineering in general [82] and for programming tools in particular [92]. Still, the biggest confounding factor is the human, which is hardly controllable (or predictable) in cognitively demanding activities such as programming and tool building.

In the following, we describe our learnings from our experimental design including the insightful pilot runs. Since we only have an implementation of Vivide for Squeak, potential participants are (under-)graduate students with $1 - 2$ years of Smalltalk experience from our lectures and seminars. Such a selection can speed up the training phase and reduce potential friction losses for the technology. That is, the students already now about Smalltalk's language concepts and Morphic's interaction concepts. So, we focus on the design of representative tasks (for external validity), intervention-free conduct (for internal validity), and efficient training (for practicability).

8.1 External Validity: Representative Tasks

We adhere to our notion of tools as described in section 2.1, section 2.2, and section 2.3: applications with interactive graphics to fetch, examine, and retain structural information of software artifacts. Further, we think that tools for Smalltalk's code artifacts represent a viable domain for such an experiment. In Squeak, this familiarity includes the System Browser, Object Inspector, Workspace, and Debugger. All these tools have simple list, text, or button views, which makes visual mapping straightforward: labels, pictures, colors.

AB-BA Design

We have a *within-subjects* approach because willing participants that know Smalltalk are scarce. So, one participant uses both Vivide, which is the *experimental group*, and a traditional framework, which is the *control group*. We expect to recruit about 10 to 20 students; only (much) higher numbers could justify the alternative between-subjects design. On the positive side, we can compensate for some *variation* in programming skills and other human traits. On the negative side, we have to address *carry-over* effects between rounds such as *learning* and *motivation*. For example, if a student begins with Vivide and then switches to a comparable framework, she might already know how to solve the task. This introduces a bias to the results. Consequently, we designed *two similar tasks* to be used in *two rounds* for each participant, which we balance in *two configurations*:

Configuration AB

Round I: Task 1, Vivide
Round II: Task 2, Other framework

Configuration BA

Round I: Task 1, Other framework
Round II: Task 2, Vivide

Balancing is a powerful mechanism to reduce the influence of confounding factors. Given enough resources, one could balance not only the order of tasks/treatments but also age, gender, and skill metrics such as years of experience. We make a trade-off that allows us to continue making measurements as participants arrive. That is, we can begin (or continue) analyzing the results when both configurations have the same number of participants.

In the control group, we use Squeak's Tool Builder [192] framework. It compares with traditional tool building because of (1) model-view separation [149] and (2) a declarative way of describing user-interface layouts. The only downside is that there is *no interactive way* of expressing such an interface; there is no "GUI builder" [128].

So, we have to consider that effort and separate it from other *glue code*. The layout code for an exemplary tool with a single text field looks like this:

```
buildWith: builder
  | window text |
  window := builder pluggableWindowSpec new
    model: self; "Target for widget callbacks."
    extent: 300@200; title: 'Example Tool'.
  text := builder pluggableTextSpec new
    model: self; "Target for widget callbacks"
    getText: #text; setText: #text:;
    frame: (0@0 corner: 1@1). "Fill the entire window."
  window children: { text }.
  ^ builder buildWith: window
```

From our experience with such tools, we think that the biggest effort lies in the implementation of the model (callbacks), not the `#buildWith:` method. Thus, Squeak's Tool Builder is comparable with frameworks for non-Smalltalk environments.

Small Effect Size

We expect an effect size of less than $10x$, which is considered challenging due to human variation and the signal-noise relationship in measurements. Yet, this is only a rule of thumb, and even factor $3 - 4$ would be good news for tool building.

We target up to *120 minutes* for the control group, which is the median for many such lab studies [92]. Thus, our task should be doable in about *30 minutes* using Vivide. Note that we use our framework as benchmark because we think that Squeak's Tool Builder can cost much more time if programmers get lost in glue.

We cannot exceed two hours per round because fatigue or lack of motivation would influence the second round. For example, a programmer who would take much over two hours with the traditional framework could be frustrated and hence bias the time measurement for Vivide.

Also, we cannot design tasks that are too simple because (1) they would not be representative and (2) the risk of shortcuts would be higher. This would impair external and internal validity. For example, if the participant should only add a button to the interface, any past experiences with Squeak's Tool Builder could distort the measurement at the expense of Vivide.

The bottom line is that the expected effect size dictates one of the biggest trade-offs in our experimental design. Even if the control group takes longer than two hours, which we can derive from observation, statistical analysis will have to assume "instant finish" for fairness. We suppose that the possible duration of four hours (for both rounds) might even exceed the maximal continuous span of attention (or motivation) programmers have. Any recreational freedom, such as longer breaks, would introduce more variables we cannot control in such an experiment.

Two Similar Tasks

As shown in figure 8.1, we designed two tasks to mitigate learning effects for our within-subjects strategy. Such a design choice, however, can impair comparability between rounds, which threaten the experiment's credibility. So, we followed two sets of *guidelines* during task design to ensure *similarity* and *novelty*. First, we think that the tools (or tasks) are "similar enough" because we use the same

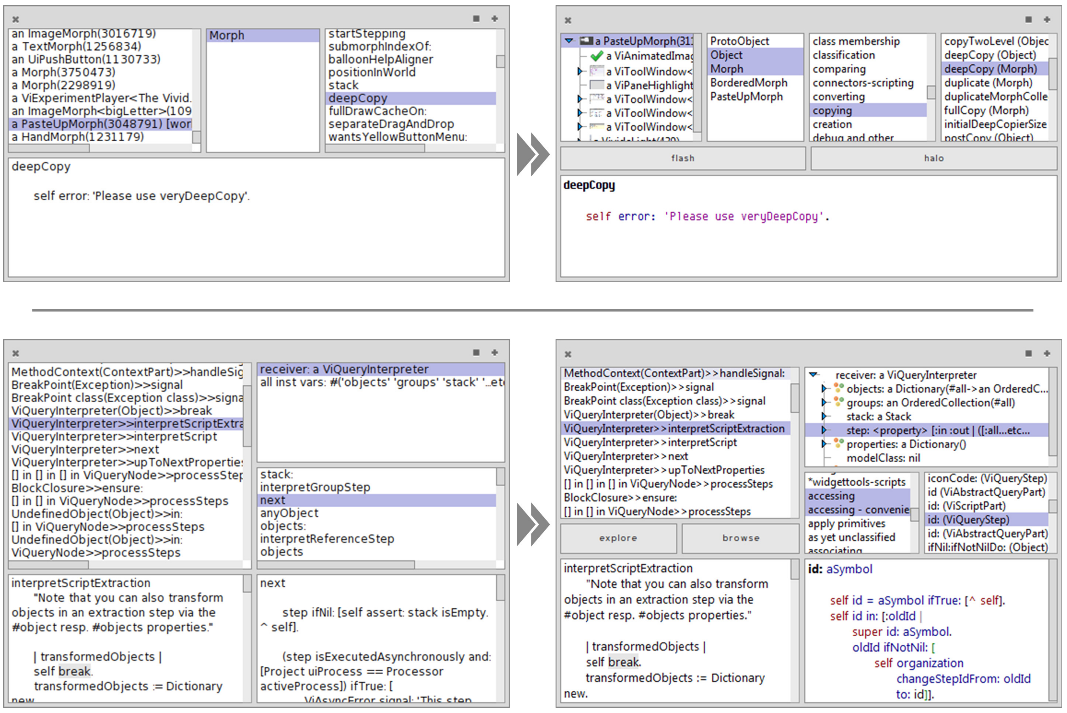
- kinds of widgets, namely buttons, (hierarchical) lists, and text fields,
- kinds of artifacts, namely code artifacts,
- kinds of sub-tasks, namely add/remove/change widgets with similar difficulties.

Second, we think that the tools (or tasks) are "novel enough", and thus not predictable, because we vary

- the layout of widgets, namely "browser style" versus "debugger style",
- the kind of complementary artifacts, namely graphics (i.e. `Morph`) or execution (i.e., `MethodContext`),
- the order of sub-tasks, which we show only one after another.

Both tasks focus on exploration, and participants should change the way tools present artifact structure. We partition each task into 11 sub-tasks to guide participants and reduce the risk of strolling. In the task about the *object browser*, the sub-tasks are (1) make list to tree, (2) show more objects, (3) add buttons, (4) add icons to tree, (5) configure text field, (6) sort objects, (7) add list view, (8) sort objects, (9) configure text field, (10) configure list selection, (11) change list labels. In the task about the *debugger browser*, the sub-tasks are (1) make list to tree, (2) add icons to tree, (3) add buttons, (4) configure text field, (5) sort objects, (6) add list view, (7) sort objects, (8) configure text field, (9) show more objects, (10) configure list selection, (11) change list labels. — We can complete each task in less than *20 minutes* using VIVIDE. This is the benchmark we set for a successful training phase for all participants.

Synopsis We have a *within-subjects* design with two similar tool-building tasks to mitigate *carry-over effects* between rounds. The tasks are *similar* because of shared kinds of widgets, artifacts, and sub-tasks. The tasks are *novel* because of different sub-task order, graphical layout, and complementary artifacts. The *experimental group* uses VIVIDE, and the *control group* uses Squeak's Tool Builder, which is a traditional model-view framework with declarative GUI definitions. We expect an effect size of $3x$ to $4x$ because of the complexity of representative tool building. The practical task size for the control group is capped at *120 minutes* because of motivation and fatigue. We can do our tasks with VIVIDE in under *20 minutes* each.



```
an ImageMorph(3016719)
a TextMorph(1256834)
an UiPushButton(1130733)
a Morph(3750473)
a Morph(2298919)
a ViExperimentPlayer<The Vivid.
an ImageMorph<bigLetter>(109
a PasteUpMorph(3048791) [wor
a HandMorph(1231179)
```

```
Morph
```

```
startStepping
submorphIndexOf:
balloonHelpAligner
positionInWorld
stack
deepCopy
fullDrawCacheOn:
separateDragAndDrop
wantsYellowButtonMenu:
```

```
deepCopy

    self error: 'Please use veryDeepCopy'.
```

```
a PasteUpMorph(31
  a ViAnimatedImag
  a ViToolWindow<
  a ViPaneHighlight
  a ViToolWindow<
  a ViToolWindow<
  a ViToolWindow<
```

```
ProtoObject
Object
Morph
BorderedMorph
PasteUpMorph
```

```
class membership
classification
comparing
connectors-scripting
converting
copying
creation
debug and other
```

```
copyTwoLevel (Objec
deepCopy (Object)
deepCopy (Morph)
duplicate (Morph)
duplicateMorphColle
fullCopy (Morph)
initialDeepCopierSize
postCopy (Object)
```

```
deepCopy

    self error: 'Please use veryDeepCopy'.
```

```
MethodContext(ContextPart)>>handleSig
BreakPoint(Exception)>>signal
BreakPoint class(Exception class)>>signa
ViQueryInterpreter(Object)>>break
ViQueryInterpreter>>interpretScriptExtra
ViQueryInterpreter>>interpretScript
ViQueryInterpreter>>next
ViQueryInterpreter>>upToNextPropertie
[] in [] in [] in ViQueryNode>>processStep
BlockClosure>>ensure:
[] in [] in ViQueryNode>>processSteps
UndefinedObject(Object)>>in:
[] in ViQueryNode>>processSteps
UndefinedObject(Object)>>in:
ViQueryNode>>processSteps
```

```
receiver: a ViQueryInterpreter
all inst vars: #('objects' 'groups' 'stack' '...et
```

```
stack:
interpretGroupStep
next
anyObject
objects:
interpretReferenceStep
objects
```

```
interpretScriptExtraction
    "Note that you can also transform
objects in an extraction step via the
#object resp. #objects properties."

    | transformedObjects |
    self break.
    transformedObjects := Dictionary
new.
```

```
next

    step ifNil: [self assert: stack isEmpty.
^ self].

    (step isExecutedAsynchronously and:
[Project uiProcess == Processor
activeProcess]) ifTrue: [
        ViAsyncError signal: 'This step
```

```
MethodContext(ContextPart)>>handleSignal:
BreakPoint(Exception)>>signal
BreakPoint class(Exception class)>>signal
ViQueryInterpreter(Object)>>break
ViQueryInterpreter>>interpretScriptExtraction
ViQueryInterpreter>>interpretScript
ViQueryInterpreter>>next
ViQueryInterpreter>>upToNextProperties
[] in [] in [] in ViQueryNode>>processSteps
BlockClosure>>ensure:
[] in [] in ViQueryNode>>processSteps
```

```
receiver: a ViQueryInterpreter
  objects: a Dictionary(#all->an OrderedC...
  groups: an OrderedCollection(#all)
  stack: a Stack
  step: <property> [:in :out | ([:all...etc...
  properties: a Dictionary()
  modelClass: nil
```

```
*widgettools-scripts
accessing
accessing - convenie
apply primitives
as yet unclassified
associating
```

```
iconCode: (ViQueryStep)
id (ViAbstractQueryPart)
id: (ViScriptPart)
id: (ViQueryStep)
id: (ViAbstractQueryPart)
ifNil:ifNotNilDo: (Object)
```

```
interpretScriptExtraction
    "Note that you can also transform
objects in an extraction step via the
#object resp. #objects properties."

    | transformedObjects |
    self break.
    transformedObjects := Dictionary
new.
```

```
id: aSymbol

    self id = aSymbol ifTrue: [^ self].
    self id in: [:oldId |
        super id: aSymbol.
        oldId ifNotNil: [
            self organization
                changeStepIdFrom: oldId
                to: id]].
```

Figure 8.1: Two comparable tool-building tasks. The upper one is about an *object browser* that connects the graphical hierarchy to source code artifacts. The lower one is about a *debugger browser* that connects code in execution to source code artifacts. In both tasks, participants should change views, add views, and add (functional) buttons.

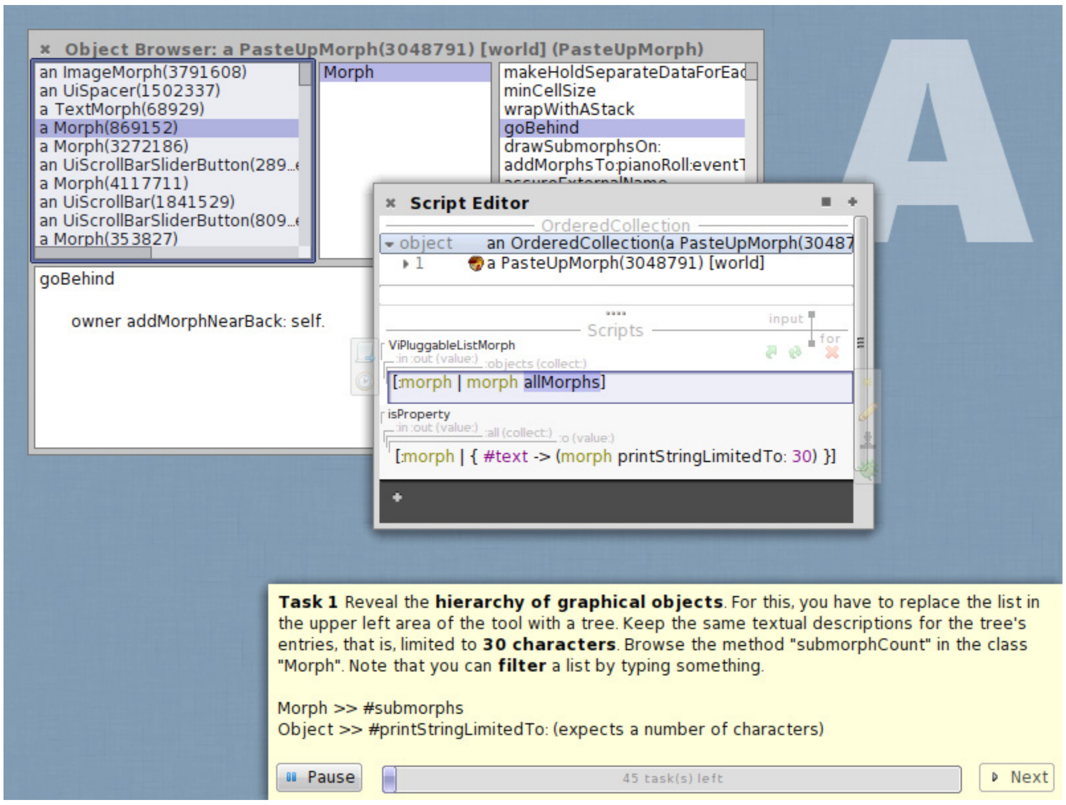
8.2 Internal Validity: Intervention-free Conduct

We modified the Squeak/Smalltalk environment to guide participants and collect data as illustrated in figure 8.2. At best, they can complete the entire experiment without intervention, which fosters *reproducibility* of our results. We were inspired by the Biscuit system [138], which supports experiments around tools for code exploration and organization.

We want to streamline two aspects in this experiment to allow for accurate time measurement and thus effect size. First, we want to *split up* the task instructions to hinder participants to plan ahead and take shortcuts. Second, we want to *automate* the tests for a sub-task's goal state to avoid cheating and strolling. Both aspects lead to thoughts on *breaks* and *idling*, which can also influence measurements.

Split Up Instructions

The instructions for each sub-task should be concise and comprehensive. As shown in figure 8.2, we tried to use simple English phrases and emphasis on selected terms. Participants can only see the current instructions and not go back or peek forth. We



```
goBehind

	owner addMorphNearBack: self.
```

```
object    an OrderedCollection(a PasteUpMorph(30487
  1       a PasteUpMorph(3048791) [world]
```

```
[:morph | morph allMorphs]
```

```
[:morph | { #text -> (morph printStringLimitedTo: 30) }]
```

Task 1 Reveal the **hierarchy of graphical objects**. For this, you have to replace the list in the upper left area of the tool with a tree. Keep the same textual descriptions for the tree's entries, that is, limited to **30 characters**. Browse the method "submorphCount" in the class "Morph". Note that you can **filter** a list by typing something.

Morph >> #submorphs
Object >> #printStringLimitedTo: (expects a number of characters)

Figure 8.2: The environment guides each participant through two rounds. The *control panel* at the bottom informs about the current sub-task (here: "Task 1"), domain code to use, and overall progress. On mouse over, the panel shows a screenshot of the expected visuals. The "Next" button will be activated when automated tests pass.

consider *four* aspects to describe sub-tasks: informal description, formal domain code, illustrative picture, and framework-specific hints.

The *task description* should meet the participants' common understanding of the domain, and *selected domain classes* (and methods) should connect that understanding to source code. We do not want to measure the time to find out about the specific domain code because our framework reduces *generic glue*. For the task about the *object browser*, we use 41 words on average per sub-task description and provide 20 domain methods in total. For the task about the *debugger browser*, we use 48 words on average and provide 29 methods in total. (Note that the latter number includes methods that re-occur because they can be needed repeatedly.) We think that such instructions are concise and comprehensive. Per sub-task, participants only learn about *one to three* domain methods, which are directly related to the informal description.

The *illustrative picture* should help participants understand the visual level of the current tool modification. Since there are sometimes browsing instructions, visual comparison can further clarify the textual description. For example in figure 8.1, the left screenshots could be the status quo and the right screenshots could be the goal state. In program comprehension, a tangible representation of artifact structure is key. We dictate this level of tangibility here because we want to *control* the experiment.

Finally, we provide *framework-specific hints*, which is basically additional source code. Given that participants are knowledgeable in the particular framework, occasional lookup in code or documentation is nothing to worry about. Again, we do not want to measure the time to do this lookup. Instead, we show selected hints directly in the control panel. For example, the first hints look like this:

> "It's Vivide! There is a `ViPluggableTreeMorph`, which shows a model's tree structure." — Experimental Group
>
> "It's Squeak's Tool Builder! There is a `PluggableTreeSpec` to specify tree widgets." — Control Group

For the task about the *object browser*, we use 10 and 7 words on average per hint in experimental group and control group respectively. For the task about the *debugger browser*, we use 12 and 7 words on average per hint in experimental group and control group respectively. We think that these additional hints preserve the conciseness and complement the comprehensiveness of our instructions.

Automated Tests

We want to automatically find out when each sub-task is done. Manual judgment takes time and can be inconsistent, which impairs reproducibility. As indicated in figure 8.2, the button to reveal the next instructions is disabled until the system "sees" the expected results on the screen. This guard does still not rush participants through the experiment. They have to click that button because the system should rather guide than force.

Test code can process Morphic's graphical hierarchy through the global variable `ActiveWorld`. So, we can perform periodic checks on that structure and look for patterns that satisfy the current sub-task. Both Vivide and Squeak's Tool Builder use the same kind of morphs to show information. From a quick look, one cannot tell which framework is in action. The only difference is the way those morphs are constructed and configured. That is, the means to *find* code, *change* code, and *verify* the results differ.

We designed an *embedded domain-specific language* to simplify the experiment's automated tests. We basically created a custom `TestCase` class that has messages for analyzing the graphical hierarchy. The correct widgets have to show the correct information. So, participants have to write code for the tools *and* use the tool to make the tests pass. For example, an instruction to browse the method `#submorphCount` in the class `Morph` can be tested like this:

```
self thereIsAnother: PluggableListMorph thatSatisfies: [:list |
  self is: list
      aListThatHasSelection: 'submorphCount'].
self thereIsAnother: PluggableTextMorph thatSatisfies: [:field |
  self is: field
      aTextThatShows: (Morph » #submorphCount) getSource].
```

There are messages to *query* a kind of widget an then run some checks:

- `#thereIsA: aMorphClass thatSatisfies: aBlock`
  Evaluates the block with any instance of `aMorphClass` that can be found on screen.
- `#thereIsAnother: aMorphClass thatSatisfies: aBlock`
  Evaluates the block with an instance of `aMorphClass` that has not been queried before in this test.
- `#thereIsNoOther: aMorphClass thatSatisfies: aBlock`
  Evaluates the block with all instances of `aMorphClass` that have not been queried before in this test.

There are messages to test the *visual mapping* for specific widgets:

- `#is: aListMorph aListThatShows: stringOrText`
  Navigates the internal structure of the given list to look for the given label.
- `#is: aTreeMorph aTreeThatShows: stringOrText`
  Navigates the internal structure of the given tree to look for the given label.
- `#is: aTextMorph aTextThatIncludes: stringOrText`
  Navigates the internal structure of the given text field to look for the given content.

There are also messages to test *control flow*, which we use for buttons. To be sure that a certain button works, we log the occurrence of a callback *plus* the object that triggered it. Thus, we can use common actions to implement with new buttons such as “explore” for an arbitrary object to invoke Squeak’s Object Explorer. Recall that we dictate certain domain messages in instructions as described above. In tests, such control-flow guards look like this:

```
| exploreWrapper |
"1) Create or retrieve the wrapper for the callback."
exploreWrapper := self
  ensureControlFlowCheck: [:context |
    "Here, we are in the middle of a button click."
    context receiver isKindOf: PluggableButtonMorph]
  in: StandardToolSet class >> #explore:.
"2) Find a button that matches."
self thereIsAnother: PluggableButtonMorph thatSatisfies: [:button |
  (button label asString includesSubstring: 'explore')
    and: [button == exploreWrapper matchingReceiver]].
```

Participants have to click on that button at least once and then wait for the periodic check to occur. If satisfied, the next instructions might be revealed. Note that we implement this mechanism with method wrappers [22]. For every new button, it takes one test cycle to install the wrapper, which might be noticed during the experiment. Yet, we test *every 2 seconds* and assume that participants would double or triple check the button if in doubt.

There can be *false positives* and *false negatives*. We noticed that participants can be skeptical if a sub-task is marked "done" too quickly. They would rather call the instructor instead of cheat. Since sub-tasks build on each other, any skipped modification could turn out to block a later one. If we learn about such mistakes, we can add additional lines to our tests because bug descriptions can often be articulated in terms of our test language. If not, the test languages can be extended.

Measuring Time: Idle and Break

We target measurements from 30 minutes up to 120 minutes. Due to our small effect size, we have to consider events that happen at the granularity of *seconds or minutes*. So, we combine *periodic checks* and *event logging* to support analysis and interpretation of the results. Logged timestamps should reveal task completion, breaks, and time-outs.

First, we distinguish the following *events*, whose timestamps might reveal outliers in later analyses:

- (Sub-)task started (or "Next" button clicked)
- (Sub-)task completed (or task test passes)
- Break started
- Break completed
- Auto-break detected
- Time-out detected

Second, we perform the following *periodic checks* to trigger the events mentioned above:

- Idle check every *10 milliseconds* for auto-break after *60 seconds* continuous idling
- Time-out check every *60 seconds* for time-out after *120 minutes* continuous work time (i.e. without breaks)
- Task check every *2 seconds* for "Next"-button unlock

We chose these sampling rates in all conscience. In common response-time guidelines, keyboard or mouse input is in the range of milliseconds [176, p. 445]. Breaks are likely to matter after one minute because we want to count minutes. Task checks may cause micro-lags, which does not affect the overall quality-of-service too much every several seconds.

We care for *breaks*, primarily, to handle *unforeseeable events* that do not relate to the experiment. For example, there are toilet breaks, phone calls, or lunch breaks. The latter is possible between rounds because we switch treatments (or frameworks). Participants can enter such a break at any time by clicking the “Pause” button as shown in figure 8.2. Then, the screen goes black to fully leave the task context. Nevertheless, we hope that longer breaks do not occur within rounds because *think time* is difficult to control then. The addition of *idle detection* tries to mitigate human error even further because phone calls, for example, can be urgent. Yet, participants can take off-line notes or think about the problem for more than 60 seconds. In that case, we assume that they will quit the break directly after noticing that the screen turned black.

Synopsis We split up task instructions into 11 sub-tasks. Each of those instructions consists of a concise, informal description of less than 50 words on average. It is complemented with domain code, a screenshot, and framework-specific hints. Participants should focus on user (or tool) interaction and not documentation lookup.

We expressed all task instructions in a custom, domain-specific language to check for task completion periodically. That language is embedded in Squeak/Smalltalk and extends its Unit-testing framework to analyze the Morphic graphical hierarchy.

The automatic time measurement considers task completion, breaks, and time-outs. Periodic checks ensure a granularity of seconds or minutes because we target 30- to 120-minutes sessions due to our small effect size.

8.3 Training is Key

Programmers will most likely benefit from Vivide if they adjust their habits to its data-driven perspective as described in section 4.4. Thus, participants in this experiment have to be trained to quickly articulate framework-specific actions based on abstract task instructions. For example, “add a list” (roughly) translates to “use `PluggableListSpec` in `#buildWith:`” for the control group and to “add pane with `ViListView` as `#view` in script” for the experimental group. The articulation of such goals *takes more time* for novices than for experts of the particular framework. Yet, our hypothesis assumes that programmers are fairly knowledgeable in this regard to make Vivide effective.

From our experience, we think it is not feasible to teach our scripting language, UI-design language, and data-driven working practice at the day of the measurement. Our participants are expected to know Squeak’s Tool Builder already, but that knowledge should be refreshed, too. So, the training phase represents another trade-off in addition to participant selection and task design. There is little documentation on how to proceed in this case. We are aware that this entire challenge “suggests that there is an inherent bias in controlled experiments toward evaluating tools

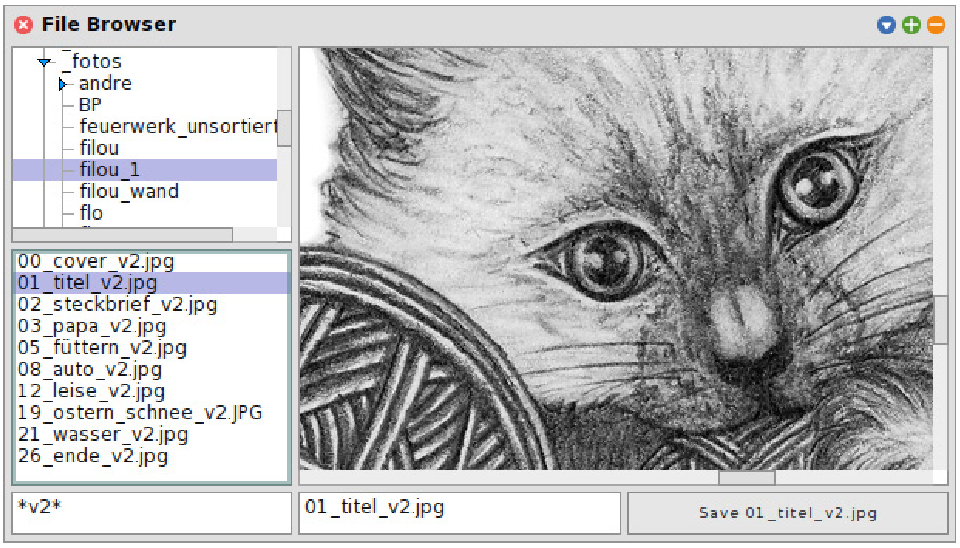


Figure 8.3: Participants will create this File Browser as training for Vivide and Squeak's Tool Builder. This tool processes files and directories. It can filter by name, rename files, and show picture or text files.

that can be quickly learned, and against tools that require significant practice." [92] Nevertheless, we are interested in gathering more insights, being careful not to jump to conclusions without proper data. Pilot runs can provide feedback on how to proceed.

Homework, Deliverable, Questionnaire

Participants should be able to invest the time they think they need for training. One week in advance, we provide the instructions for a *third task* and the Squeak environment, which includes Vivide, to do the task. As depicted in figure 8.3, that task is about files, directories, and pictures. During that week of training, we are available for additional support, personally or via e-mail. Then, at the day of the measurement, we want to *see* the tools created in the task to verify completion. A complementary *questionnaire* helps guide and track the learning progress for the two frameworks' features.

The training task is about creating a browsing tool for files and directories. Compared to our experiment tasks, we choose the *same* widgets but *different* domain artifacts. That is, programmers should know about their languages and tools, which are the frameworks and widgets, to approach an unknown problem, which are the artifacts. We prepared a *video tutorial* that accompanies participants and solves the task step by step. For Squeak's Tool Builder, there are *69 minutes* of dubbed footage divided into *10 files* for easy reference. For Vivide, there are *82 minutes* in *18 files*. Basically, participants watch and listen to the videos and replicate the interactions they see. Then, they save the Squeak image and bring along the image file for us to verify. We think that one week can be enough to process about 2.5 hours of video and learn from it—assuming that participants are willing and motivated.

Our goal is to teach the respective framework model so that participants can articulate actions using terms of the framework. Both homework task and video tutorial may transport this *by example*, but a generic understanding of terms requires more training. Instead of providing more tasks, which are difficult to design, we decided to create a *questionnaire* as a checklist. Participants should be able to answer all questions to themselves and to the instructor. Examples include "What is the relationship between model and view?" and "What are workspaces good for?" There are many questions to cover the many terms and concepts of Smalltalk, Squeak, Squeak's Tool Builder, and VIVIDE:

1. Framework design and interaction concepts
   - Squeak: model, view, builder, observer, specifications
   - VIVIDE: pane, script, view, connections
2. Programming language concepts
   - Squeak/Smalltalk: objects, messaging, enumeration, branching
   - VIVIDE: object transformation, property extraction, script references, tuples
3. Interactive tool support
   - Squeak: code browser, transcript, workspace, object inspector
   - VIVIDE: script editor, script wizard, code folding, script properties
4. Actual tool construction
   - Squeak: base classes, framework messages, layout properties
   - VIVIDE: add/remove panes or connections, typical views, typical script properties
5. Pitfalls and debugging
   - Squeak: breakpoint, process, debugger, manual interrupt
   - VIVIDE: pane halo, script editor

This summary of concepts illustrates the challenge we face: participants have to learn or refresh much information upfront. Recall that we target people that have 1 – 2 years of Squeak/Smalltalk experience, are knowledgeable in model-view separation, and have more than 2 years of programming experience. This covers procedural, object-oriented, and functional paradigms. Consequently, many concepts we need here can be seen as variations and applications of existing knowledge.

Lessons Learned from Pilot Runs

We recruited *three students* to test our experimental setup. For each session, we recorded the video (without audio) to discuss and debug issues that may occur. We learned that (1) the task instructions are comprehensive, (2) the experimental group progressed faster, (3) unsupervised homework fails, and (4) qualitative insights emerge.

First, **yes**, the instructions, domain code, picture, and framework hints were sufficient to guide participants through the experiment. There were no questions asked to the instructor about what to do to unlock the "Next" button. We conclude that our choice of artifacts and widgets is representative, which all participants confirmed to us. It actually feels like a tool-building task.

Second, **encouragingly**, the experimental group (or round) progressed faster than the control group (or round), looking at the sub-task completion rate. However, both groups timed out after two hours. Thus, we cannot *and should not* analyze the collected timestamps to draw conclusions because the experiment's integrity would be impaired. If we decide to reduce the number of sub-tasks, new pilot runs will be necessary.

Third, **no**, the kind of training we offered did not work. The seriousness of required practice was not clear enough. Participants did their homework very late in the week, even without completing it. The questionnaire turned out to be difficult to evaluate live before the measurement. In the recorded video, we noticed a sense of cluelessness from the way participants operated the user interface. Thus, the articulation of actions from instructions was flawed: the user's model differed strongly from the design (or system) model [134, pp. 12–17]. Consequently, we think about revising the training phase.

Finally, we think it is more likely to collect results rather *qualitatively* than quantitatively from this study setup. On the one hand, actual numbers support credibility when arguing the effect Vivide has in comparison to Squeak's Tool Builder. On the other hand, generalization for software tools is difficult because many ideas have only a few implementations. Different programmers are likely to implement ideas differently, influenced by their communities and daily challenges. Consequently, qualitative feedback can yield new perspectives on tools across programming languages, frameworks, and environments. That is, if Smalltalk-like environments can benefit from *artifacts-first* concepts, other data-oriented (e.g., files, tables, objects) environments might revise their *tools-first* strategy, too.

Toward a Tool-building Workshop

We want to maintain the participants' *motivation* throughout the lengthy training phase. From our case studies (chapter 7), we learned that tool building might best be a complementary activity. For example, the Thresher project (section 7.1) primarily looked for rules to automatically cluster fine-granular code changes; the Gramada project (section 7.2) had to implement Ohm for Squeak and several parsing-expression grammars to better understand the activity of language creation. In both cases, the setting entailed discussions with other students and advisors to collect feedback. We might be able to design such a setup for our experiment training, too. A group of participants could jointly experience and discuss Vivide and Squeaks' Tool Builder to understand each framework's terms and mechanics.

Training tasks should include more details about *live programming*. Typically, 1 – 2 years of occasional Smalltalk during lectures and seminars are not enough to fully understand and apply the benefits of immersive programming systems such as Squeak. For example, programmers are free to think only in terms of source code and how it gets written to the file system or code repository. They can set breakpoints, print statements to the console, and thus lengthen their feedback loop like in traditional, file-based environments. Yes, they can still create interesting applications that way. However, a thorough understanding of liveness may free many cognitive resources to create more complex solutions with smaller teams. We think that programmers can get more from Vivide if they are already proficient with Smalltalk's object graph and Morphic's tangibility.

We argue that a *multi-day workshop* is a viable next step to explore participant training in a supervised group setting. We would basically transform our homework task into an interactive, classroom-like version. The first days would be used to present the concepts of live programming and tool building. Several tasks would be performed individually and results discussed in groups. This might be a *filter for motivation* to let unwilling participants go before the measurement. On the last day, we would conduct the experiment and measure the time as described above.

[|]

We see systematic, rigorous evaluation of programming tools as an interesting, open field of research. While *quantitative* results may be hard to achieve, even *qualitative* insights should be collected in structured, reproducible ways.

Synopsis Training is necessary to form a baseline for all participants so that *varying measurements* are likely due to the experimental treatment and not "learning by doing." We designed an additional task to be done *off-site* and *one week* upfront. The results should be demonstrated to the instructor, and a *questionnaire* should check learning progress beyond that training task.

After three *pilot runs*, we conclude that such homework is likely to be ignored and that a supervised group workshop is a viable next step. There are *inconclusive* signs of Vivide outperforming Squeak's Tool Builder. However, the current form of training yields to *time-outs* (after 120 minutes) for both control groups and experimental groups.

Summary

In addition to our own experiences (chapter 6) and the student projects we advised (chapter 7), we are interested in *measuring the effectiveness* of VIVIDE compared to other tool-building frameworks. Such *quantitative evaluation* is usually done via controlled experiments (or lab studies). However, programming is a cognitively demanding activity and hard to control in terms of *dependent* and *independent* variables. Thus, only few practices for such rigorous experiments about (complex) programming tasks have been documented so far.

We designed a *within-subject* experiment with *two similar tasks* to mitigate variations because of the human factor. Participants complete *two rounds* and apply either VIVIDE or Squeak's Tool Builder to modify browsing tools. They change list labels, add buttons, or add new views. We cap each round at *120 minutes* to balance motivation, which is a typical limit for such lab studies.

We extended Squeak to guide all participants through the experiment without manual intervention. *Task instructions* are split up into 11 sub-tasks each. They include a natural-language description, the required domain code, framework-specific hints, and a picture of the resulting tool modification. We also created *automated tests* that use a *domain-specific language* to analyze the graphical hierarchy. So, participants have to use the tool in the expected way to make progress, not just write code. During time measurement, the system denotes *breaks* and *idling* to support later analysis.

Finally, we noticed that *training* is a crucial part in our experimental setup. At the time of writing, we learned from *pilot runs* that participants are not motivated to train at home. We think that a *supervised group setting* represents a viable next step to maintain motivation and check learning progress. In any case, we can and did already collect *qualitative* feedback from our setup; it revealed bugs and suggested features.

[|]

In the next chapter, we describe more related work, which broadens our perspective on querying, mapping, and presentation languages for interactive, graphical tools in exploratory programming environments.

Part IV

Future Directions for Programming Environments

9 Related Work

We designed our Vivide tool-building framework based on issues we observed in the realm of tools for domain-specific program comprehension. In section 1.2, we identified the state-of-the-art as many promising, yet isolated, ideas that lack conceptual integration as a flexible environment. Vivide can be such an environment because programmers can integrate software artifacts and interactive views to tailor the graphical representation of any information. They can combine different *window managers* to layout views. They can combine different *scripting languages* to process artifacts. Consequently, they can *easily shape here what is usually not malleable* in environments such as Eclipse and Visual Studio.

In this chapter, we describe more related work. There are many projects, tools, languages, libraries, frameworks, or environments that try to disenchant the challenges of programming [89]. A common vision is the creation of an accessible, powerful computational medium that augments the human intellect[1]—hopefully no professional training required. We share that vision. In relation to Vivide, we found (1) scripting languages for data processing, (2) means to describe graphical user interfaces, and (3) forms of presentation for different knowledge processing.

9.1 Related *Query* Languages

There are many scripting languages in which programmers can express rules to query information spaces with low effort. For our implementation of Vivide, Smalltalk was the obvious choice because its *messages* and Squeak's *blocks* foster conciseness in a declarative way. However, our idea of alternating *object transformation* and *property extraction* to describe hierarchical models is independent of Smalltalk.

In the following, we present high-level languages that can provide a concise and declarative syntax for processing data (or software artifacts). Vivide could incorporate them to further accommodate different data sources and programming paradigms.

Lisp-based Languages

Lisp [114] is a general-purpose programming language with a taste for the functional programming paradigm. Lisp applications can represent systems that are changeable at run-time and thus compare to Smalltalk systems. For example, the

[1]Douglas C. Engelbart wrote a report about a possible conceptual framework in this regard [49].

text-manipulation environment EMACS [185] is designed for and implemented in a dialect called EmacsLisp. Like Squeak/Smalltalk, such dialects accommodate certain domains with standard libraries. A popular dialect of Lisp is Scheme [1], which has been used in applications for painting [47] or charting [48] to add powerful extension points for expert users.

Inkwell [60] is a tool written in Lisp and using Lisp as configuration language to extensively *process data*. Scripts describe templates to transform pieces of writing according to metrics such as agreeableness and extraversion. Such writings can be poetry, narratives, or non-fictional reports. We think that this application of Lisp shows that the functional paradigm is suitable to help express domain-specific intents in *data-driven* tools.

In Vivide scripts, Lisp-based transform steps could be expressed like Squeak's Collection protocol. The Scheme dialect [1] has already functions to `map` and `filter` elements in lists, which is the same as `#collect:` and `#select:` in Squeak. Blocks correspond to lambda functions. For example, a script that computes the squares of numbers could look like this:

```
(define script (lambda (in)
  (map (lambda  (num) (* num num) ) in )))
(script '(1 2 3 4)) ; evaluates to '(1 4 9 16)
```

Such scripts could still be represented as objects in Squeak's object graph. There are implementations of Lisp for Smalltalk such as Qoppa, which we used in a case study (section 7.4). Since both Smalltalk and Lisp support the functional programming paradigm, we see Lisp as a good fit for programming-tool construction to reduce glue to focus on domain-specific transformations.

Prolog-based Languages

Prolog [2] is a general-purpose programming language that follows (or defines) the logic programming paradigm. Programs are made of rules and facts, which represent relations; program execution means *query evaluation* over these relations. This renders Prolog-based languages suitable as query languages to access as process software artifacts for programming tools. For example, Jquery/TyRuBa [39] can describe a three-level model hierarchy for views with a single line of Prolog code:

```
type(?T),re_name(?T,/Figure$/),method(?T,?M),returns(?M,?R)
```

For all matching tuples in the (ordered) form of `(?T,?M,?R)`, the first level would show classes whose names end with "Figure", the second level would show all methods in that class, and the third level would show the return type (or class) for each method. Note that the order is an additional property of such scripts, and the object properties (i.e. text and icons) cannot be configured in Jquery. In Vivide for Squeak/Smalltalk, such a script would look like this:

```
script := {
  [:classes | classes select: [:cls | cls name endsWith: 'Figure']].
  [:class | { #text -> class name }].
  [:class | class methodDict values].
  [:method | { #text -> method selector }].
  [:method | method returnType. "Not implemented."].
  [:class | { #text -> class name }].
} asScript.
script openScriptWith: SystemNavigation default allClasses.
```

With such logic rules, programmers can think differently about the structure of software artifacts. That is, they do not have to describe a hierarchy of artifacts level-by-level but their overall relationships. The hierarchy can be defined *afterwards* on the matching tuples, which can be treated as a different set of artifacts. On the one hand, Smalltalk messaging is suitable for navigating the object graph in small steps. On the other hand, Prolog query resolvers can take a broader look on *everything* and extract meaningful information. For example, common behavior between two classes can be described with this SOUL query [118]:

```
commonBehavior(?C1,?C2,?M1,?M2) if
  method(?C1,?M1), method(?C2,?M2),
  methodStatements(?M1,?Stats), methodStatements(?M2,?Stats).
```

In Vivide, such a query would become a larger Smalltalk expression in a single transform step, which would impair modularity and readability. We think that only the *results* of such Prolog queries should be processed in Vivide scripts. By using *tuples*, the script from above could change as follows and accept tuples of any size:

```
script := {
   [:tuples | tuples groupByFirst]
      -> { #id -> #'e9f654d1' }.
   [:object :tuples | { #text -> object printString }]
      -> { #isProperty -> true }.
   [:object :tuples | tuples collect: [:t | t allButFirst] ].
      -> { #next -> #'e9f654d1' }
} asScript.
script openScriptWith: (Prolog evaluate:
    'type(?T),re_name(?T,/Figure$/),method(?T,?M),returns(?M,?R)').
```

Note that this code assumes an implementation of `Prolog` behind `#evaluate:`, which should be integrated with the Squeak/Smalltalk object model. Also, the script's steps are so short because we assume that the *script wizard* will expand steps to the in-out form (`[:in :out | ...]`). A more specific *property extraction* would also bloat the generic, yet compact, "`object printString`" expression.

There are many other examples of Prolog-based scripting languages to query code artifacts for interactive programming tools. ASTLog [37] uses logic queries to examine abstract syntax trees. There are rules and facts to prepare code artifacts for the MOOSE visualization toolset [158]. We also think of SQL [4] and SPARQL [150] because their queries yield tuples (or tables), too. Compared to traditional Prolog, SQL statements in the form of SELECT/FROM/WHERE provide more control over the selection of sources to query artifacts from.

Dataflow-based Languages

There are programming languages (and systems) that focus on *streams of data* as opposed to singular data points stored in variables. Programs create, transform, and combine streams to filter or derive information. Streams can often be sampled and inspected or fed into other programs. The notion of *time* becomes tangible, and sometimes first-class, because programmers can model the history of data in tangible chunks. With this abstraction of time, interpreters for such languages can optimize resources via *parallelism* and *laziness*. In the object-oriented world, dataflow could be treated as an *elaborate observer pattern* [62, pp. 293–303], which could be reified through objects that follow the *stream protocol* (instead of the collection/list protocol). Such design is sometimes referred to as *reactive programming*.

We were inspired by UNIX filter programs [153, pp. 266–267]. Each filter performs a small task; many filters can be combined (or chained) to process streams of characters or bytes. Typical examples include *awk* [153, pp. 200–202] and *grep* [153, p. 266], which both offer textual pattern matching for filtering or other effects. UNIX shells execute filter programs in parallel because the parts in such data pipelines are not required to "finish". In Vivide, script interpretation has no such inherent parallelism at the time of writing. For tool views, *sorting* and *duplicate removal* are common transformations, which both require knowledge about the whole data set. In the future, a proper integration of *streams as input/output buffer* is likely to entail *asynchronous script interpretation* as sketched in section 4.1.

Pure dataflow languages such as Lucid [209] add several concepts to manage time. There is `fby` ("followed by"), `next` ("look ahead"), or `whenever` ("select if"). For example, a *stream of natural numbers* can be stored behind the variable *n* as follows:

```
n = 1 fby n + 1;
```

In this inductive definition, the first element is 1, which is then *followed by* increments $n+1$. In Vivide, such a stream (or generator) of natural numbers might be expressed like this:

```
script := {
  [:in :out | | n |
    n := 1. "First value"
    [ out nextPut: n.
      n := n + 1  "Increment"] repeat ]
  -> { #out -> SharedQueue. "Stream semantics. Thread−safe."
       #async -> true }
} asScript.
```

Here, the script step would constantly emit numbers into the output buffer. Another step can then consume that output. Note that there is future work to explore *consistent tool updates* for such concurrent script interpretation. Although, we expect little challenges because of our modular separation of scripts, panes, and object connections. There is an implementation of such dataflow for Squeak/Smalltalk in KScript [136], which uses *controlled ticking* to advance time and fill streams. For example, a *stream of points* in ticks of 100 milliseconds can be written as follows:

```
aPoint <- P(0, 0) fby P(timerE(100) / 10, 0)
```

To get such frequency for our natural-number script above, we could add "`(Delay forMilliseconds: 100)wait`" into the `repeat` loop. The underlying Squeak process would suspend for 100 milliseconds.

General-purpose programming languages can carry dataflow constructs as *embedded languages*. For example, Smalltalk's primary paradigm is object-orientation, but functional, logical, or dataflow programming can be expressed with objects and messaging. Squeak's collection protocol [32] supports a functional style with its messages `#collect:` (or "map"), `#select:` (or "filter"), and `#inject:into:` (or "reduce"). In Vivide, we use *Smalltalk's syntax* and object model to offer a *data-driven* way to describe view models. In the same spirit, there are *configurable program tracers* that use a subset of Python [87] or domain-specific Lisp [111] to control debugging tools. Such scripts usually *query* certain method activations or program slices of interest. The tracer can then collect information during program execution more efficiently. For example, the expression "`the_execution.all_calls()`" [87] does not trace any calls until needed in another expression. Vivide realizes such lazy evaluation, too, at the granularity of levels in the model tree. For this matter, the stream of natural numbers could be expressed as a *degenerated tree*:

```
script := {
   [:num | num + 1 ] -> { #id -> #'e0d4f8aa' }.
   [:num | { #object -> num } ] -> { #next -> #'e0d4f8aa' }.
} asScript.

tree := script interpretScriptWith: #(0).
tree children first object == 1.
tree children first children first object == 2. "... etc ..."
```

This tree is of infinite depth. It is degenerated because each level has a single child, which holds the number in the `#object` property. Programs can traverse this "tree-structured list" to consume numbers. If nodes have *no parent reference*, the system's garbage collector can clean up consumed portions of this "stream".

Synopsis Vivide scripts alternate steps of (a) object transformation and (b) property extraction to describe hierarchical view models. We use Squeak/Smalltalk because blocks and messaging can add little overhead to domain-specific rules. Yet, the use Smalltalk is not mandatory or exclusive in our design.

Learning from Lisp-based languages, we could adopt a similar functional style with a likewise *compact syntax*. Learning from Prolog-based languages, we could integrate a different perspective on artifact structure whose *logical queries* result in tables (or tuples) to be processed in Vivide scripts. Learning from dataflow-based languages, we might extend *script interpretation* to be parallel or at least concurrent for a better abstraction of time.

9.2 Related Means for Data-to-Graphics *Mapping*

In tool building, much of the effort can be accounted for *visual mapping* because the transition from domain code to view code entails most of the *glue*. Programmers have to prepare the queried artifact structure for views in the form of labels, colors, and other visual properties. They iterate and thus have to read and change any glue over and over again. To one end, *low-level event/drawing interfaces* provide a maximum of flexibility but little convenience. To another end, *high-level visual languages* and direct-manipulation interfaces [173, 75, 172] offer convenient view construction but limited means of configuration. Like Vivide, there are many approaches that build on model-view separation [149]. Typically, there is also some form of programming (or scripting) to tackle unforeseen challenges.

Generic Visualization Construction

The programming tools we address are interactive, graphical applications. Such tools make information about software artifacts accessible and comprehensive. So, we relate to the field of information visualization [174], which addresses data exploration and presentation. We argue that tool builders can adopt techniques from projects that improve the construction of interfaces that visualize information [68]. Vivide, Squeak, and Morphic offer practical means to help programmers apply best practices from that field:[2]

Visual Builder Designers have full control over visual elements like in presentation software or graphics editors. — Vivide offers such flexibility through a combination of scripts, connected panes, views and pane views. However, most of the flexibility in view design is attributed to Morphic's interactivity and the Smalltalk object model.

Shelf Configuration Designers can choose from a set of default charts to display data. Being less flexible than visual builders, they might still combine those charts in documents. — We created several views for Vivide that make *no* use of pane views but plain Morphic. There are lists, tables, trees, text fields, or treemaps to be configured through property extraction in scripts.

Template Editor Designers can choose from a set of default charts, *and* there is interactive guidance for chart configuration. For example, Microsoft Excel has interactive dialogs (or wizards) to support visual mapping. — Vivide reflects script changes *directly* in corresponding views. Thus, tool builders can quickly experiment with different configurations. Also, our script editor offers *templates* to shorten scripts.

[2]See [68] for a more elaborate explanation of the choice of categories and concrete example systems. An alternative catalog can be found in [31, pp. 436–442].

Visualization Spreadsheet Designers can layout many different chart configurations side-by-side to quickly explore different combinations of data sets. — Due to the self-supporting nature of Squeak/Smalltalk, tool builders can write a Vivide script that applies *multiple scripts* on the same data set to show *multiple views* populated in different ways. Since all views are morphs, such a side-by-side comparison is automatically integrated in the interactive environment.

Textual Programming Designers apply traditional programming languages to describe screen contents and interactivity. — In our current implementation of Vivide in Squeak/Morphic, programmers can always use Smalltalk code and Morphic objects in scripts. They can write custom views and pane views to express new layout containers or entire visualizations. Yet, we think that elaborate view design excels the duty of a programmer that wants to build programming tools.

Visual Dataflow Programming Designers can employ visual languages to express data transformations and visual mapping. There are often shapes that encapsulate operations and connectors between shapes that define dataflow. — Vivide offers two kinds of dataflow: (1) script interpretation and (2) object exchange between panes. When constructing the view model from script steps, data flows between input and output buffers. When interacting with views embedded in panes, data flows from one pane into other ones. Such object exchange triggers script interpretation to populate the next pane's view.

Vivide itself is a script-based, direct-manipulation tool-building environment that *needs* objects, scripts, and views to support programmers and programming tasks. Without objects, there is no information to show. Without scripts, there are no rules to populate views. Without views, processed information can only be exported into other frameworks or environments. Consequently, the amount of available objects, scripts, and views determines the level of tool-building support Vivide offers.

Objects-based Approaches

The notion of objects [211] influences the design and implementation of libraries and frameworks for creating (object-oriented) user interfaces [31]. For visual mapping, Vivide scripts wrap domain *objects* into model *objects* and enrich them with (named) property *objects*. Views can then access those helper *objects* to form tangible representations. So, our approach maintains objects throughout all abstraction layers using Smalltalk. There are many different takes on how to balance readability and extensibility for such glue code.

The design method called Naked Objects [144] argues to *derive* (or generate) the *single* presentation of an object from its distinguishing structure. There are no means of configuration for such presentations (or views). Consequently, different views on the same structure *require* different *domain* objects to be created. While we disagree with this viewpoint, we see value in *default presentations*. So, we created several scripts and views that make Squeak's object model accessible for common programming tasks. However, we argue that there is a need for *task-specific* views on objects because there is often more information available than space on screen.

The *Extended Markup Language*, XML for short, is often used as a declarative middle language to describe object-oriented user interfaces. XML nodes, elements, and attributes can directly be treated as *objects in hierarchies*. Examples include UIML [6] and UsiXML [103], which can describe mapping and presentation in a declarative way. Entire user-interface patterns can be captured this way [212]. Note that there is often glue code *generated* to connect to lower level languages. For example, the Qt framework [18] generates C++ from XML specified through an interactive GUI-design tool. Such code generation can target different languages and frameworks. In Vivide, programmers drag and drop *panes* to compose the user interfaces. We generate Smalltalk code from pane compositions to serialize and share tools. The serialization format could serve as intermediate for other graphics frameworks.

Structured source-code editors relate to our approach because they design *visual metaphors* based on implementation structures. For language artifacts, such structures are *abstract syntax trees*. For example, Barista [93] uses model-view separation and the Citrus framework [94] to enrich the visual representation of code. For another example, Envision [9] focuses on means of configuration for its visual metaphor. To some extent, many structured editors are like Naked Objects [144] for code artifacts. In Vivide, we build on the structured nature of Smalltalk tools, which show single class or method definitions at a time. One can further split up statements and messages using morphs as interactive vehicles as done in Etoys [7], Scratch [109], or other tile-based languages.

The Tools-and-Materials metaphor [159] describes software artifacts as materials to be processed by tools, which are configured under certain *aspects*. Such aspects define tasks or interactions such as "list-able", "browse-able", and "text-editable", which are thus closely related to the visual presentation. Following this metaphor, Vivide scripts could be treated as aspects because they describe how to populate certain interactive views. Using Morphic's halo to switch scripts in pane compositions, Vivide tools could process multiple aspects of a certain material (or software artifact).

Smalltalk-based Approaches

We found two projects that use Smalltalk as a scripting language to configure graphical views (or visualizations): (1) Glamour [24] and (2) Mondrian [15, pp. 214–260]. In both projects, there are *domain-specific objects* to store configuration data and *domain-specific messages* to change that configuration. Vivide has objects for scripts, panes, views, and connections. It adds *only a few* custom messages such as `#->` and `#<-` for property extraction. Mostly, programmers define the behavior of Vivide tools through interactive composition of object structure with the help of script editors, custom pane halos, and drag-and-drop.

A successful application of Glamour is the Moldable Inspector [30], which supports domain-specific exploration of objects. This meta tool uses Glamour's *presentations*, *transmissions*, and *ports* to integrate tool-building features into the user interface. Programmers can script the representation of software artifacts. For example, the graphical hierarchy can be derived from any given (root) morph as follows:

```
gtInspectorDisplaySubmorphsOn: aCanvas in: aContext
   <gtInspectorPresentationOrder: 80>
   <gtInspectorTag: #custom>
   ^ aCanvas tree
       title: ' Submorphs ' ;
       rootsExpanded;
       display: [:rootMorph | {rootMorph}];
       format: [:morph | morph printString];
       children: [:morph | morph submorphs];
       when: [:morph | morph submorphs notEmpty]
```

The messages `#format:` and `#title:` do object-*specific* visual mapping and object-*independent* view configuration respectively. Scripts are regular Smalltalk methods; pragmas (<. . .>) configure the integration in the tool environment. In Vivide, the same specification would be expressed as follows:

```
script := {
   [:morph | { #text -> morph printString }]
      -> { #id -> #'d6b8be61'.
           #view -> TreeView.
           #title -> ' Submorphs ' .
           #rootsExpanded -> true.
           #presentationOrder -> 80 .
           #tag -> #custom }.
   [:morph | morph submorphs]
      -> { #next -> #'d6b8be61' }
} asScript openScriptWith: {rootMorph}.
```

Note that we unify all object-independent properties such as `#title` and `#tag` as *script properties*, compared to the mix of pragmas and messages. In general, the amount and complexity of the resulting source code is not that different. Both approaches do the querying and mapping of software artifacts in a similar fashion. Yet, the recursive model definition is more concise in Vivide via `#id` and `#next` compared to `#display:`, `#children:`, and `#when:`.

Mondrian scripts describe views in terms of *shapes*, *edges* between shapes, and *layout* of shapes [15, pp. 214–260]. Programmers work with specific visual elements compared to Glamour's predefined specifications. That is, they let shapes look like a tree instead of configuring a tree's shapes. Like Vivide, Mondrian puts domain objects into (model) *nodes*, which can be enriched for visual mapping. For example, a polymetric view (like the one from figure 6.9 in section 6.3) can be created with the following Mondrian script:

```
view nodes: Collection withAllSubclasses .
view shape rectangle
             width: [:class | class instVarNames size * 3];
             height: #numberOfMethods;
             fillColor: [:class | class hasAbstractMethods
                                          ifTrue: [Color lightGray]
                                          ifFalse: [Color white]].
view edgesFrom: #superclass .
view treeLayout.
```

Here, the `Collection` class and all of its subclasses are shown in a system-complexity view [99, p. 35]. Edges show the superclass relationship, rectangle extent shows number of variables and methods, and the color shows the existence of abstract methods. In Vivide, the same specification would be expressed as follows:

```
script := {
  [:class | { #width -> (class instVarNames size * 3).
              #height -> class numberOfMethods.
              #fillColor -> (class hasAbstractMethods
                                     ifTrue: [Color lightGray]
                                     ifFalse: [Color white]) }]
    -> { #id -> #'341ccb84'.
         #view -> PolymetricView.
         #shape -> #rectangle. }.

  [:class | class subclasses ]
    -> { #next -> #'341ccb84' }
} asScript openScriptWith: {Collection} .
```

Note that we describe inheritance from top to bottom using the *one-to-many* relationship `#subclasses`. Mondrian scripts use the *one-to-one* relationship `#superclass`, which means that the *visual layout* has to reveal the hierarchy. Conceptually, that hierarchy is not *in* the Mondrian model. On the one hand, Vivide *tree* models and the `PolymetricView` cannot show graph structures because of limited means to define edges between nodes. On the other hand, Vivide *list* models and a (new) Mondrian-like graph view could yield a similar view specification, maybe like this:

```
script := {
  [:class | { #width -> (class instVarNames size * 3).
              #height -> class numberOfMethods.
              #fillColor -> (class hasAbstractMethods
                                     ifTrue: [Color lightGray]
                                     ifFalse: [Color white]).
              #hasEdge -> [:other | class superclass == other] }]
    -> { #view -> MondrianLikeGraphView .
         #shape -> #rectangle.
         #layout -> #tree }.
} asScript openScriptWith: Collection withAllSubclasses .
```

Overall, the amount and complexity of the resulting code is very similar[3] for both approaches. Yet, we think the separation of *query language* and *(visual) mapping language* is more clear, and maybe more expressive, in Vivide. If Mondrian scripts use list models, domain hierarchies (or trees) are expressed as view configuration via `#edgesFrom:`. If Mondrian scripts use tree models, shapes will be nested and visual mapping of sub-shapes will further interleave with domain queries at the level of Smalltalk messages. In contrast, Vivide offers clear separation at the level of script steps, which alternate between transformation and extraction.

Synopsis After querying relevant software artifacts, there are many ways to bring artifact structure to screen. Comparing with *traditional visualization construction*, Vivide can be applied in many ways ranging from visual builder over template editor to visual-dataflow editor. Looking into the domain of *object-oriented design*, there are many related approaches that describe visual interfaces in a declarative way, which reduces the amount of glue and fosters readability. Interactive design and code generation help integrate supportive languages, which Vivide approaches through interactive pane composition and tool serialization. Finally, there are other projects that use *Smalltalk as configuration language for visualizations*. Even though the specific design of objects and messages is subject to personal taste, the Vivide scripting language offers reasonable trade-offs for data-to-graphics mapping.

9.3 Related Forms of Tinker-able *Presentation*

Vivide offers several ways to express presentation languages of tools as described in section 4.3. Even though its building blocks are rectangular morphs (or panes), their visual appearance can have any shape. From our experience, it is straightforward to construct tools that do not follow traditional design choices as described in section 6.1. That is, our approach to organize source code in vertical containers feels different to traditional Smalltalk tools. Scripts and methods become more tangible, the notion of tools fades into the background.

There is more work that shares characteristics of Squeak/Morphic, which relates directly to our attempt to support tinkering with the environment and improving the own working habits. An arbitrary, but intentional, combination of data and graphics can emphasize domain- or task-specific details to foster efficient information consumption. Consequently, the environment's presentation language can influence knowledge access and growth. In the following, we sketch work that influenced

[3]The definition of edges between shapes is subject to the designer's taste. We prefer a boolean version via `#hasEdge` and a block that closures the source object. As an alternative, object properties could hold the concrete edge target like "`#edge -> class superclass`" or "`#edge -> #superclass`". This design choice influences the complexity of the view's implementation.

our thoughts in this regard: programmable environments, visual programming, the notebook metaphor, and Active Essays.

Programmable Design Environments

There is a perspective on software being *domain-oriented design environments* [54], which focuses on tool-supported content (or artifact) creation for specific domains. This perspective is extended through *programmable design environments* [48], which add programming (i.e., Scheme/Lisp) as flexible means of configuration and control. In such environments, for example, users can draw pictures, plan their kitchen, or populate charts with domain-specific widgets *and* scripts. The authors target *end users* and *learning* to set apart from professional programming. Yet, they also argue that programming facilities are necessary to meet unforeseeable challenges and requirements. We think that Scheme, like Smalltalk, offers little syntactic overhead, but users have to be *trained* to make use of scripting languages. It is rarely self-explanatory. In our experiment (chapter 8), we realized that a user's experience can *impede* the application of new problem-solving strategies. Maybe "unlearning" becomes necessary.

Programmable design environments combine elements from programming languages and direct-manipulation interfaces [47]. First, *languages* offer control structures, data composition, and abstraction through naming. Both our scripting language (section 4.1) and UI-design language (section 4.3) offer similar strategies to create complex tools. Second, *direct manipulation* offers domain-specific, visual metaphors. In Vivide, we use Morphic to offer tangible handles for graphical tool components and a pathway to the tool's source code. Tool builders can expose artifacts from any domain through our framework. We see great value in the interplay of high-level languages and interactive graphics in any similar system where "[language] interpreter meets [graphical] interface" [47].

One strength of Smalltalk systems is the persistent *object model* to represent information. In such systems, tools are just programs that make Smalltalk objects accessible so that programmers can understand and change that information. In the same spirit, the Rigi project [123, 124] extracts a *graph model* from source code and offers tools to (1) manage large system structures, (2) present information about, and (3) automate constraint checking against those structures. Similar to Vivide scripts, there are *Rigi objects* to *aggregate* and *generalize*, which can hence form additional views on the source code. There is an *electronic atlas* (tool), which uses such views for code exploration. Shimba [190] complements Rigi's *static methods* with *dynamic traces* to provide a more coherent reverse-engineering environment for Java programs. Eclipse or Visual Studio might be considered contemporary, spiritual successors of Rigi, yet they suffer a rather inflexible, text-centric perspective. Luckily, there are other recent environments such as Moose [132], which preserve the concept of objects and configurable views on objects.

Direct and Visual, Manipulation and Programming

Interactive, data-driven presentations go hand in hand with *direct manipulation*. The user interface of the Xerox Star [179], which was influenced by earlier Smalltalk systems [80], offers a visual metaphor for real-world objects to support productivity tasks. On screen, documents can be copied, organized, written on, or sent as mail. To overcome the limitations of direct manipulation, the Star supports *records processing* [151], which automates data transformation and visual mapping for such productivity tasks. Thus, Star's focus is different from our goals for Vivide. We value the consumption of many artifacts more than the creation and manipulation of selected ones. Yet, both approaches support both strategies.

In the course of programming and direct manipulation, the notion of *visual programming* emerges. Fabrik [79] is an environment that combines computational components with user-interface components to realize bi-directional dataflow. The idea has a more recent implementation [106] using contemporary technologies. The scope of Fabrik is broad: program music, animation, or any graphical structure that depends on domain-specific data. Vivide complies with the author's definition of a "visual programming kit" [79]:

1. "Specification of an effective visual and computational interface for each component" — Our *scripts* and scripting language represent the computational, *panes* and the UI-design language represent the visual component.
2. "Interactive access to an interesting [...] library of existing components" — Our script organizations represent libraries of example scripts to be applied and re-used. Yet, we rely on existing visualizations and widgets to provide interactivity.
3. "The ability to use and combine these components interactively to build new library components and finished applications" — Programmers can combine scripts to express complex view models. They can compose panes to express complex presentation languages. Both can be stored in organizations for scripts and view configurations.

Overall, Vivide approaches the challenge of visual programming for the domain of tool building for program comprehension (and modification). There are many systems that reify transformations as visual building blocks to be connected for dataflow (or object exchange). In these systems, the intended *simplicity* typically leads to *end-user* programming, which targets *non-professional* programmers. However, we think that our approach is far from *intuition* and thus requires *training*.

Documents in a Notebook

Paper has been inspiring for many visual metaphors so far. Sheets of paper can have different shapes and hold information in textual or graphical form. In the physical world, there are pages, documents, letters, cards, or posters organized in folders, scrapbooks, or notebooks. In the digital world, characteristics of such elements can be discovered quickly, and tasks can be associated intuitively. As mentioned above, the Star's user interface [179] is just one example for using *documents* as tangible artifacts on screen.

Paper-based metaphors can influence the *presentation* of information. HyperCard [67] uses *cards* and *links* to layout and connect information interactively. *Stacks* of cards represent shared properties among artifacts such as a set of contacts and a set of appointments. In Vivide, we explored a similar approach with *single-object tools* (section 4.2), which can be linked through *script selection* (section 4.4). Yet, our experiences cover mostly code artifacts, which relates to the Hopscotch [26] editor (and tool framework) for the Newspeak programming language [20]. Hopscotch implements a document-driven interface like in Web browsers. In each document, elements can be combined to reflect Newspeak's language concepts such as lexical scoping in the tool's presentation language.

Paper-based metaphors can guide the *creation* of information, which is comparable to note-taking. Mathematica/Wolfram Notebooks[4] support iterative, interactive data analysis. Notes become *tangible* entities on a persistent, shareable, digital medium. IPython Notebooks [148] apply the same metaphor for open-source communities. Basically, the term "notebook" refers to pieces of information that are positioned vertically as *connected, interactive views*. Such views are often *command-line interfaces* (or code editors) to access and process data sets. In Vivide, we apply such layout for code editors, object explorers, and other views. In practice, programmers can mix instances of `Text` with `CompiledMethod` to document their thoughts alongside task-relevant code artifacts.

Active Essays for Dynamic Systems

> "An 'Active Essay' is a new kind of literacy, combining a written essay, live simulations, and the programs that make them work in order to provide a deep explanation of a dynamic system. The reader works directly with multiple ways of representing the concepts under discussion. By 'playing with' the simulations and code, the reader gets some hands-on experience with the topic." — Alan Kay, cf. [214]

We argue that the challenges of program comprehension and tool support relate directly to the scope of Active Essays. Programmers have to understand the dynamic behaviors of complex software systems to fix bugs or add features. Active Essays can be seen as an innovation (or evolution) of (school) books using modern technology. In a similar way, interactive, integrated documentation of software systems could innovate—maybe even replace—traditional, external approaches, which quickly become outdated as systems evolve. Vivide can help explore possible forms of such documentation.

[4]Wolfram Mathematica: `https://www.wolfram.com/mathematica/`, accessed 2018-08-31

Boxer [45] is a "reconstructible computational medium" that combines a "spatial metaphor and naive realism." In this system, users define functionality through Lisp code that is nested in rectangular containers, called "boxes". The visual metaphor resembles Vivide pane compositions and script organizations that are attached to pane-views as means of abstraction (section 4.3). The designers of Boxer relate to "Interactive Books", which seems to be an earlier instance of Active Essays. The overall emphasis on *language* and *interaction* shares several aspects of Smalltalk systems and the Morphic framework. Still, the *accessibility* of such media depends on the *design* of applications they transport.

ThingLab [19] is a "graphics simulation laboratory" that implements a constraint-based programming language and a visual interface. For example, users can construct Physics experiments and interactively modify variables to experience their effects in the simulation. *Constraints* are an instance of *rules* that Vivide scripts can capture to configure interactive views. For example, scripts could trigger *constraint solvers* for debugging purposes. The visual interface of ThingLab tooling exhibits *data-driven* characteristics.

The Alternate Reality Kit (ARK) [181] "is a system for creating interactive animated simulations." The authors investigated the learnability of visual metaphors on a scale from "literal" to "magical". They considered buttons, menus, and other widgets. We face a similar challenge for Vivide concepts as discovered in our experimental setup (chapter 8): "magical" to its users even though "literal" to its designers. Note that ARK was implemented in Smalltalk-80 but precedes the Morphic graphics framework [110] known from Self and the halo [108] known from Squeak.

Synopsis There are many forms of interactive presentation in software. Means of programming (or scripting) are typically embedded to enable flexible configuration and tinkering with provided defaults. A recurrent and thus influential approach is the combination of (1) direct manipulation and (2) paper-based metaphors. Documents, cards, and notebooks mimic real-world artifacts to improve discoverability and acceptance of graphical user interfaces for knowledge processing.

Summary

The challenge of building *tools for program comprehension* relates to much existing work in the field of software engineering. In particular, Vivide addresses the integration of *scripting languages* and *interactive systems*. Thus, we focus on related means for *querying*, (visually) *mapping*, and *presenting* the structure of software artifacts. Yet, much related work targets users that are non-professional programmers.

There are many programming languages that can be applied as querying languages in a tool-building context. Lisp-based languages have a concise *syntax* and are suitable for *functional* programming, Prolog-based languages support *declarative* query expressions, and dataflow-based languages abstract *time* in a tangible way.

There are many related means for data-to-graphics mapping. Typically, *interactivity* and *programming* go hand in hand. *Object-orientation* fits the designer's perspective on artifact structure and visual properties. There are other *Smalltalk* frameworks for tool building, which underlines the practicability of our choice.

There are related forms of tinker-able presentation, which often combine *paper*-based metaphors and *direct manipulation*. Sheets of paper represent an accepted medium to *transport knowledge* in text or graphics. Programming tools and other graphical, interactive systems can build on that visual metaphor to improve accessibility.

[|]

In the next and final chapter, we summarize the main argument of this work, explain possible future work, and conclude our thoughts.

10 Conclusion

This chapter concludes our thoughts on the creation, integration, and modification of interactive, graphical tools for programming. We think that there is an overall agreement on the data-driven, domain-specific nature of contemporary programming challenges. Yet, the actual means to support program comprehension are often subject to personal experiences. Thus, environments that support exploration and experimentation in a generic fashion are likely to benefit all sorts of programmers. Active tool building during software development can help accommodate specific domains, tasks, and preferences.

In the final sections, we summarize the *argument* of this work and sketch a *vision* for next-generation programming environments. This includes possible *extensions* for VIVIDE and its mechanics as well as next steps toward thorough *tool evaluation*.

10.1 Argument of this Dissertation

The main argument of this work builds on our understanding of information as software artifacts, programming tasks as information foraging, and interactive tools as tangible medium (chapter 2). We motivate the challenge of tool building using Squeak/Smalltalk as research platform (section 3.2). On the *philosophical* side, we propose a different perspective on graphical tools in integrated environments (section 4.2), which implies a different working practice (section 4.4). On the *engineering* side, we propose a scripting language (section 4.1) and a UI-design language (section 4.3) as practical advice for better tool-building support.

Triggers and Barriers for Tool Building

We *simplify* the problem for programming environments by using a *purely object-oriented* system that has *inherently tangible* graphics: Squeak/Smalltalk [77] and Morphic [108]. That is, all relevant information can be accessed or derived through Squeak's object graph, and there are morphs that can serve as interactive views on objects. To get started, the environment already has *generic tools* that align with language constructs and support basic program comprehension and modification.

In such environments, the *triggers* for tool building are related to *managing the object graph*. That is, tools should provide *higher-level views* that crosscut generic structures to filter or derive new information. We distinguish programmers' needs to *fetch*, *examine*, and *retain* artifact structure because we want to emphasize a *data-driven* perspective on tools.

In such environments, the *barriers* for tool building are related to a lack of "tool" support for *separating domain code from framework glue*. That is, programmers want to populate views with data, which includes code for *querying*, *mapping*, and *presenting* software artifacts. Yet, that code is difficult to *find*, *change*, and *verify*, even with traditional Smalltalk and Morphic-halo tools.

A Data-driven Perspective on Tools

Our goal is to make programmers *see* artifacts as tangible representations on screen and *not see* tool windows with visual content to interpret. We formulate a supplementary *tool-building mantra*: "Artifacts first, tools follow." We argue that with such a data-driven perspective, we can improve the means to construct the interactive representations of software artifacts.

We propose a *tool-building strategy* where tools (1) focus on *distinctive* properties of single artifacts, (2) accept *similarities* for groups of artifacts, and (3) benefit from *context* artifacts in the environment. Even in complex tool interfaces, programmers should be able to identify the involved artifacts. For example, we follow this strategy to build tools for code artifacts so that we can read and write Smalltalk code, which we explained in (section 6.1).

As an implication, we propose a *data-driven working practice*, where programmers constantly choose artifacts from on-screen representations to choose further representations to eventually explore and understand task-related information. That is, they open *no tools but representations* to navigate the information space. By doing so, they interactively control the level of distinctiveness, similarity, and context to organize screen contents as desired.

A Combination of Scripting and Interaction

We follow two common practices that support the creation and maintenance of complex software systems: (1) high-level, declarative programming languages and (2) interactive, graphical programming tools. Both practices can yield *complexity from simplicity* by deriving compile-time and run-time structures, such as code and objects respectively, from abstract rules and user interactions.

First, we propose a new scripting language to describe *hierarchical view models*. The elements of that language alternate *object transformation* and *property extraction* to query and map software artifacts at each level in the model tree. In our implementation, we use Smalltalk and Squeak's `Collection` protocol to keep scripts concise and domain-specific.

Second, we propose a new UI-design language to interactively compose a tool's *interactive presentation*. The elements of that language are *panes*, *views*, *connections*, and *pane views*. Panes combine objects, scripts, (generated) models, and views; they can be connected to exchange objects. Pane views can encapsulate layouts and interaction paradigms to construct complex tools. In our implementation, we use Morphic to represent panes and views; a pane's halo provides access to its scripts, objects, and connections.

10.2 Future Directions for Vivide

Our implementation of Vivide in Squeak/Smalltalk is a fully working environment that supports common programming tasks. Conveniently, the Squeak system runs still underneath and represents a safety net for missing features. We picture several directions to extend Vivide mechanics and evaluate its usefulness.

Mechanics

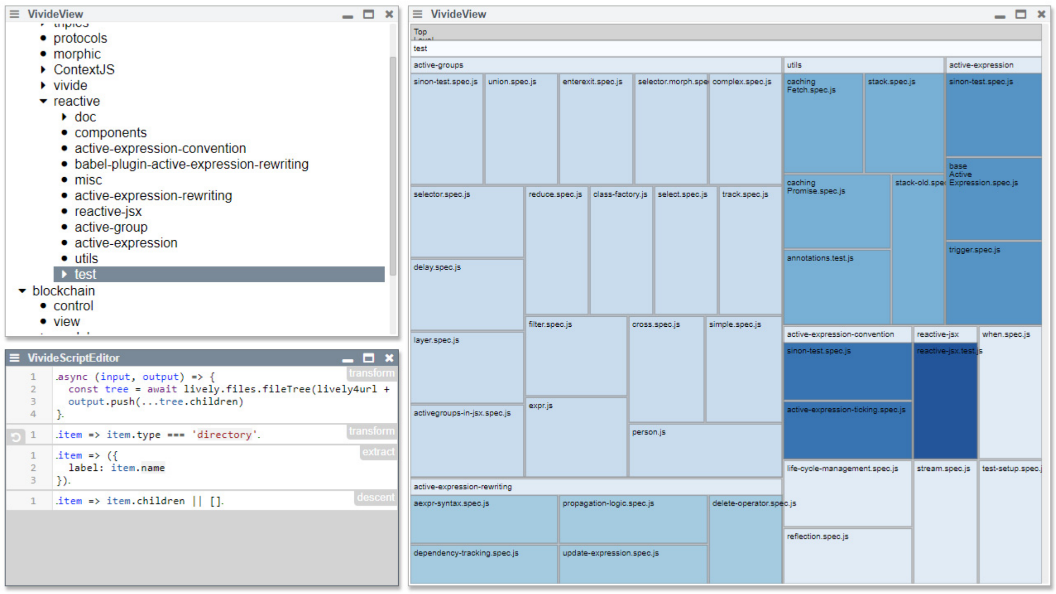
There is always a need for more software artifacts and more interactive views to explore the limits of our scripting language and UI-design language. We think that the following aspects need more attention:

- Applicability of our tree-based models for graph-based views, which entails trade-offs between script code and view code
- Asynchronous script interpretation and stream/queue semantics in each script's input/output buffers
- Coordination of visual properties across views, maybe via special connections between panes
- Better tool support for script authoring, that is, high-level undo and guided code refactorings

The composition of multiple views in complex, nested layouts can yield more challenges. We think that our approach can have an effect on the kinds of tools programmers want to build and share in the team. Thus, the following aspects need also more attention:

- Improve interactive tooling to create and manage a multi-layout presentation language with desktops, tiles, tapes, and other forms embedded in each other.
- Establish domain-specific tools (or tool building) as a form of integrated, up-to-date documentation, whose artifacts should be shared in the community.

Considering software engineering in general, the creation of data-driven applications for *non-programming* domains can reveal more challenges to investigate in the future.



```
.async (input, output) => {
  const tree = await lively.files.fileTree(lively4url +
  output.push(...tree.children)
}.
```

```
.item => item.type === 'directory'.
```

```
.item => ({
  label: item.name
}).
```

```
.item => item.children || [].
```

Figure 10.1: The languages and strategies we introduced with Vivide can be ported to other environments. There is a working prototype of Vivide for Lively4: VivideJS.

Evaluation

For our current study setup, we are going to reduce the task complexity to ensure the upper time limit as described in chapter 8. This includes a more effective training phase so that participants can express their actions in Vivide terms. After such rather traditional tool-building tasks, we want to investigate the effects of our framework in more detail. That is, we want to assess the *data-driven perspective* on tools using our scripting language and our UI-design language.

First, programmers should be able to express their information needs as scripts. They can write complex hierarchical models or simple list models. If the alternation of script steps for tree levels is too difficult, programmers will be likely to prefer lists or tables over trees. As explained in section 6.3, it is possible to convert multiple list scripts into single tree scripts. We want to investigate the need for such conversions.

Second, programmers should be able to manage context and view integration via panes and object connections. They can use a flat desktop style or choose to combine pane views for a mix of layout styles. If the transition between fundamentally different layouts is too difficult, programmers will be likely to prefer a single paradigm. That is, they might prefer the desktop style as in Squeak or the tiled layout as in Eclipse. We want to investigate the acceptance and benefits of mixed layouts to accommodate specific tasks and preferences.

Eventually, the languages and strategies we introduced with Vivide can be ported to other programming environments. Such ports can evaluate applicability for other programming communities. As illustrated in figure 10.1, there is already a working prototype of Vivide for Lively4,[1] which makes use of Web technologies such as HTML5 [52] and JavaScript.

Vivide as Programming Environment

Objects, Smalltalk, Morphic, and the Vivide-way of tool building: programmers can manage all kinds of software artifacts in such an environment. Then, the actual *programming languages* can become pluggable like in Eclipse or Visual Studio. For example, projects can use C++ or Java while Smalltalk will be the *tooling language*. Code artifacts will be represented as objects in the Smalltalk image; the file system can still hold (synchronized) code files as database for external tools. So, on the one hand, different implementations of Vivide can help find different tool-building challenges, which are likely due to different programming communities. On the other hand, extending the current Smalltalk-implementation of Vivide might lead to faster insights about future programming environments. Note that there has always been an influx of new languages and ideas, which have been requiring integration into existing technologies. Vivide might be that *integration platform* with its focus on tangible representations for all kinds of software artifacts.

Finally, we hope that *exploratory programming* can benefit from our proposed tool-building support. In our experience, *ad-hoc* tool building is difficult to trigger, trace, and evaluate. Yet, whenever a usually *complex building* activity feels like *simple configuration*, the underlying environment provides the right level of support. We think that future programming environments can benefit from our perspective on data, graphics, and tools.

[1]The Lively4 Environment: `https://github.com/LivelyKernel/lively4-core`, accessed 2018-10-28

Part V

Appendix

Appendix A

Smalltalk and Squeak

A.1 The Smalltalk Programming Language

In any Smalltalk system, such as Squeak, programmers have to read and write Smalltalk code. Any program consists of objects that exchange messages to collaborate, hopefully producing the intended behavior. Application domains cover multimedia, games, simulation, and many more. Here is an example of the Smalltalk syntax, based on "Smalltalk Syntax on a Postcard"[1]:

```
exampleWithNumber: x
  "A method that illustrates every part of Smalltalk method syntax
  except primitives. It has unary, binary, and keyboard messages,
  declares arguments and temporaries, accesses a global variable
  (but not an instance variable), uses literals (array, character,
  symbol, string, integer, float), uses the pseudo variables
  true, false, nil, self, and super, and has sequence, assignment,
  return and cascade. It has both zero argument and one argument
  blocks."
  | y |
  true & false not & (nil isNil) ifFalse: [self halt].
  y := self size + super size.
  #($a #a "a" 1 1.0) do: [:each |
    Transcript
      show: (each class name);
      show: ' '].
  ^ x < y
```

At the time of writing, this example still compiles and runs in recent Squeak releases. Our scripting language employs Smalltalk to form blocks that transform objects:

```
| block result |
block := [:in :out | out addAll: (in collect: [:ea | ea + 1])].
result := OrderedCollection new.
block value: #(1 2 3 4) value: result.
result includesAllOf: #(2 3 4 5). "true"
```

Traditional tools show Smalltalk code in a Workspace or one method at a time. Our scripting tools adopt such boundaries for one block at a time. Note that multiple tools can be used and displayed on screen.

[1] http://c2.com/cgi/wiki?SmalltalkSyntaxInaPostcard, accessed 2018-11-26

Many of our examples make use of a certain message that concatenates two collections such as strings:

```
'Hello World!' = ('Hello', ' ', 'World', '!'). "true."
```

This message `#,` is binary, and due to Smalltalk's precedence rules, we often have to put parentheses around such concatenations.

There are more details documented about the Smalltalk language and its tools in the *Blue Book* [65] and *Red Book* [64] respectively. There is much further literature on the tools and practices of Smalltalk-80 systems [97, 98].

Modern Smalltalk implementations, such as Squeak [77, 76], continue blurring the boundary between language and application even further. It is more like a continuously running multimedia *authoring system* than a simple programming language with a compiler and execution layer.

A.2 Squeak Code Exploration Efforts

In section 3.1, we used the following code snippet to count the packages, classes, methods, and lines of raw Smalltalk code:

```
"Number of packages to approximate number of libraries and applications."
numPackages := PackageOrganizer default packages size. "95"

"Number of classes to approximate kinds of run−time objects to explore."
numClasses := SystemNavigation default allClasses size. "2260"

"Number of methods to approximate code reading efforts."
numMethods := 0.
SystemNavigation default allSelectorsAndMethodsDo: [:b :s :m |
   numMethods := numMethods + 1]. "52338"

"Number of lines of code to approximate method sizes."
rawLinesOfCode := 0. "With comments but no empty lines."
SystemNavigation default allSelectorsAndMethodsDo: [:b :s :method |
  rawLinesOfCode := rawLinesOfCode + method linesOfCode]. "389978"
```

Note that we did not normalize the code style because the Squeak's code browser shows methods as they are stored. Usually, Smalltalk programmers have a personal taste regarding indentation and line breaks, which affects readability for the whole community. There is *pretty printing* to unify the displayed format, but it is not the default setting in code browsers, yet.

The system we used was Squeak 5.1, Build 16549. Note that programmers can significantly reduce the number of artifacts to explore if they are able to narrow down the domain of their tasks. For example, Squeak organizes its main parts in only a few packages:

KERNEL-* Code compilation, class (hierarchy) (re-)definition, basic exception handling, process scheduling and synchronization, user input events, primitive types such as numbers and Boolean values.

Chronology-* Everything related to time, time spans, durations, time stamps, etc.

Collections-* Everything related to working with multiple objects such as arrays, strings, streams, and dictionaries.

Graphics-* Everything related to graphical output, includes support for font rendering, drawing rectangles, and processing various image encodings such as PNG, JPG, and GIF.

Files-* Accessing the local file system using streams. Listing and navigating folders.

Network-* Using sockets to fetch network resources. Includes support for UDP, TCP, HTTP, and HTTPS.

Sound-* Managing audible output and reading various sound file formats. Includes support for FM, WAVE, and MIDI.

System-* Code change notification, object events, weak arrays and finalization, object serialization, projects.

We documented more details about Squeak's base system and its graphics frameworks in [192].

A.3 Visual Ratio of Smalltalk Methods

In section 4.2, we argue for vertical lists to show multiple Smalltalk methods at once because method source code is best shown at a wide ratio. We calculated the ratio by trimming trailing blanks in the method's source, replacing tabs with four spaces, and ignoring methods with more than ten lines:

```
ratio := OrderedCollection new.
lines := OrderedCollection new.

SystemNavigation default allSelectorsAndMethodsDo: [:b :s :method |
  | tmp |
  tmp := method getSource asString withBlanksTrimmed.
  tmp := tmp copyReplaceAll: String tab with: ' '. "four−char tabs"
  tmp := tmp lines collect: [:l | l size].
  tmp size <= 10 ifTrue: [
    lines add: tmp size.
    ratio add: tmp max / tmp size ]].

ratio average roundTo: 0.01. "23.01"
lines average roundTo: 0.01. "4.49"
lines median. "4"

"Number of methods w/ 10 lines or less."
lines size "41651 −> 79.6 percent"
```

The visual impression of a typical Smalltalk method is wider than tall and looks like this:

```
xxxxxx: xxx xxxxxxx: xx
  "xxxxxxxxxxxxxxxxxxx"
  | xxxx xxxxx xxxxxx |
  xxxx := xxx + xx. "x"
  ^ xxxxxxxxxxxxxxxxxxx
```

While the choice of font and line spacing influences this ratio, the fonts in recent Squeak releases do align with this example.

Appendix B

Listings for Vivide Scripts

B.1 The Live Code Browser

In our Vivide tutorial in section 3.4, we used multiple scripts to prepare the involved software artifacts. Here, we present only *compact scripts*, which make use of *templates* as described in section 4.2. The script-based debugger features a *stack list*, which can be constructed in the following way:

```
script := {
  "Access the method activation objects from the process object."
  [:process | process suspendedContext].
  [:methodContext | methodContext stack].
  [:methodContexts |
    methodContexts reject: [:ea | ea isExecutingBlock]].
  [:methodContexts |
    methodContexts select: [:ea |
      (ea receiver isKindOf: Morph)
        or: [ea receiver isKindOf: MorphicEvent]
        or: [ea receiver isKindOf: MorphicEventDispatcher]]].

  "Extract object properties."
  [:methodContext | {
    #text -> methodContext printString.
    #icon -> (ToolIcons iconNamed: (ToolIcons
              iconForClass: methodContext method methodClass
              selector: methodContext selector)) } ].
} asScript openScriptWith: {aSuspendedProcess}.
```

In the Live Code Browser, the stack table revealed colorful information about each context object in the stack. The check for each method's argument count renders the script a little bit more complicated. Mostly, each script step's properties configures the view with color and resize information:

```
script := {
  "Only extract object properties. No transformation required."
  [:methodContext | { #text -> methodContext selector }]
  -> { #labelColor -> Color black.
       #resizeMode -> #rigid}.
  [:methodContext | { #text -> methodContext method methodClass }]
  -> { #labelColor -> Color black.
       #resizeMode -> #shrinkWrap}.
  "For up to 3 arguments, we also show details."
  [:methodContext | { #text -> (
    methodContext selector numArgs > 0
    ifFalse: ['-'] ifTrue: [(methodContext at: 1) printString])}]
  -> { #headerText -> 'arg1'.
       #labelColor -> (Color r: 0.0 g: 0.502 b: 1).
       #resizeMode -> #rigid}.
  [:methodContext | { #text -> (
    methodContext selector numArgs > 1
    ifFalse: ['-'] ifTrue: [(methodContext at: 2) printString])}]
  -> { #headerText -> 'arg2'.
       #labelColor -> (Color r: 0.251 g: 0.502 b: 0.0).
       #resizeMode -> #rigid}.
  [:methodContext | { #text -> (
    methodContext selector numArgs > 2
    ifFalse: ['-'] ifTrue: [(methodContext at: 3) printString])}]
  -> { #headerText -> 'arg3'.
       #labelColor -> (Color r: 1 g: 0.4 b: 0.4).
       #resizeMode -> #rigid}.
} asScript openScriptWith: someMethodContexts.
```

The script for the code editor uses a *text field* and a *text styler* to show source code with syntax highlighting. The script property `#styler` configures the kind of styler, and the object property `#stylerClass` configures the styler so that class-specific information (e.g., instance variables) can be found:

```
script := {
  [:methodContext | {
   #text -> methodContext method getSource
         <- [:newSource |
              methodContext receiver class compile: newSource]
   #stylerClass -> methodContext receiver class . }]
   -> { #id -> #methodEditor.
        #view -> ViPluggableTextMorph.
        #styler -> SHTextStylerST80 }.
} asScript.
```

The browser's list of multiple code editors is an automatically generated *view composition*, which represent VIVIDE means of context abstraction in the UI-design language (section 4.3, figure 4.9). Here, the pane view `ViPaneListView` manages an individual pane for each incoming method context, and each such pane has the same script:

```
script := {
  [:methodContext | {
   #script -> #methodEditor .
   #height -> 70 "pixels" }]
  -> { #view -> ViPaneListView "i.e. a pane view" }
} asScript.
```

B.2 Single-object Tools' Scripts

In figure 4.4, there are four *panes* that show four different objects (from left to right):

```
ViPane new
  borderWidth: 2;
  borderColor: (Color gray: 215/255);
  objects: { 'Morphic-Kernel' };
  currentScript: {
    [:m | SystemOrganization listAtCategoryNamed: m]
      -> { #view -> ViPluggableListMorph. #borderWidth -> 0 }.
    [:nm | Smalltalk classNamed: nm] } asScript;
  openInHand.

ViPane new
  borderWidth: 2;
  borderColor: (Color gray: 215/255);
  objects: { Morph » #handleEvent: };
  currentScript: {
    [:m | #text -> m getSource. #stylerClass -> m methodClass]
       -> { #view -> ViPluggableTextMorph.
            #styler -> SHTextStylerST80.
            #borderWidth -> 0 }} asScript;
  openInHand.

ViPane new
  borderWidth: 2;
  borderColor: (Color gray: 215/255);
  objects: { ViToolWindow someInstance };
  currentScript: {
    [:m | #text -> 'root']
      -> { #view -> ViPluggableTreeMorph. #borderWidth -> 0 }.
    [:m | #text -> m printString].
    [:m | { m class withAllSuperclasses. m} asTuples].
    [:c :m | { c instVarNames . m } asTuples].
    [:tuples | tuples sorted: [:t1 :t2 | t1 first <= t2 first]].
    [:i :m | #text -> (i truncateWithElipsisTo: 9)].
    [:i :m | #text -> (m instVarNamed: i) printString] } asScript;
  openInHand.

ViPane new
  borderWidth: 2;
  borderColor: (Color gray: 215/255);
  objects: { Morph someInstance };
  currentScript: { [:m | #self -> m]
    -> { #view -> ViPluggableTextMorph. #borderWidth -> 0 }} asScript;
  openInHand.
```

In figure 4.6, there are three panes that show the same *mouse-button event class* in three different ways (from left to right):

```
ViPane new
  borderWidth: 2;
  borderColor: (Color gray: 215/255);
  objects: { MouseButtonEvent };
  currentScript: {
    [:m | #text -> 'root']
      -> { #view -> ViPluggableTreeMorph. #borderWidth -> 0 }.
    [:m | #text -> m printString].
    [:m | { m class withAllSuperclasses. m} asTuples].
    [:c :m | { c instVarNames . m } asTuples].
    [:tuples | tuples sorted: [:t1 :t2 | t1 first <= t2 first]].
    [:i :m | #text -> (i truncateWithElipsisTo: 9)].
    [:i :m | #text -> (m instVarNamed: i) printString] } asScript;
  openInHand.

ViPane new
  borderWidth: 2;
  borderColor: (Color gray: 215/255);
  objects: { MouseButtonEvent };
  currentScript: {
    [:m | #text -> (m definition, '.', String cr,
                    m class definition, '.' )]
      -> { #view -> ViPluggableTextMorph.
           #styler -> SHTextStylerST80.
           #borderWidth -> 0 }} asScript;
  openInHand.

ViPane new
  borderWidth: 2;
  borderColor: (Color gray: 215/255);
  objects: { MouseButtonEvent };
  currentScript: {
    [:cls | {
      #('Class Hierarchy' 'Instance Variables' 'Known Messages').
      cls } asTuples]
        -> { #view -> ViPluggableTreeMorph. #borderWidth -> 0 }.
    [:cat :cls | #text -> cat].
    [:cat :cls | (cat beginsWith: 'Ins')
      ifTrue: [ cls instVarNames ]
      ifFalse: [ (cat beginsWith: 'Cla')
        ifTrue: [ cls withAllSuperclasses ]
        ifFalse: [cls methodDict values ] ] ].
    [:obj |
      obj isCompiledMethod
        ifTrue: [ #text -> obj selector]
        ifFalse: [obj isBehavior
          ifTrue: [ #text -> obj name]
          ifFalse: [ #text -> obj]]] } asScript;
  openInHand.
```

Finally, we illustrated different levels of structure that can be revealed by different kinds of views in figure 5.4. There, we compared a *text view* with a *table view* and a *tree-map view*. Since the kind of view is stored as a script property, the same script can be re-used by changing `#view`:

```
script := {

   "First level."
   [:in :out | in do: [:class | out addAll: class subclasses] ]
     -> { #view -> ViTextView.
          "#view -> ViTableView."
```

```
        "#view –> ViTreeMapView." }.
   [:in :out | in do: [:class | out add:
      { #object -> class.
        #text -> class name }]]
      -> { #isProperty -> true.
           #headerText -> 'Class' }.
   [:in :out | in do: [:class | out add:
      { #object -> class.
       #text -> class allCallsOn size asString }]]
      -> { #isProperty -> true.
           #headerText -> 'Refs'.
           #hAlignment -> #right }.

   "Second level."
   [:in :out | in do: [:class |
      out addAll: class methodDict values]].
   [:in :out | in do: [:method | out add:
      { #object -> method.
        #weight -> method linesOfCode. "tree map only"
        #tooltip -> method selector }]]
      -> { #isProperty -> true }.

} asScript openScriptWith: {Morph}.
```

B.3 The “Scripts” Script

In section 5.5, we refer to a script that lists all available scripts for a set of objects. Given that script steps have type information in `#inputKind` and `#outputKind`, we compute acceptable types as follows:

```
| mostSpecific |
mostSpecific := Object.

script allStepsDo: [:step |
   | in out |
   "Use ProtoObject as fall–back to allow for Object type."
   in := step properties at: #inputKind ifAbsent: [ProtoObject].
   out := step properties at: #outputKind ifAbsent: [ProtoObject].

   "Narrow down the options if types are more specific."
   (in inheritsFrom: mostSpecific)
      ifTrue: [mostSpecific := in].

   "If a step transforms the type, stop analysis and return current result."
   (step isProperty not and: [in ~~ out])
      ifTrue: [^ mostSpecific]].

^ mostSpecific
```

So, any step in a script can specialize the input kind. The overall type for a set of objects would be the most generic one in the class hierarchy. The script that lists the remaining options sorted by `#priority` looks like this:

```
script := {
   "1) Sort by priority."
   [:in :out | out addAll: (in sorted: [:s1 :s2 |
      (s1 properties at: #priority ifAbsent: [9999])
```

```
          <= (s2 properties at: #priority ifAbsent: [9999]) ])]

        -> { #view -> ViTreeView.
             #yieldOn -> #activated "... clicked on like menus".
             #label -> 'Scripts' }.

     "2) Extract icon and label for each script."
     [:in :out | ([:all | all collect: [:o | (
        [:script | {
           #icon -> (script nextIcon ifNil: [ViIcons blankIcon]).
           #text -> (script nextLabel ifNil: [script sourceCode]) }]
              value: o), { #object -> o }]]
                 value: in) do: [:result | out add: result]]
        -> { #isProperty -> true }
} asScript openScriptWith: someScripts.
```

The view looks like in figure 5.1.

B.4 The “Class Outline” Script

In section 6.1 in figure 6.1, we presented a tool that shows a structural overview of classes. The script that extracts super classes, instance variables, and methods by protocol looks as follows:

```
script := {

   [:in :out | in do: [:cls | out addAll: (
     (cls withAllSuperclasses reversed
        collectWithIndex: [:c :i | i -> c]),
     (cls withAllSuperclasses gather: [:c | c traits]),
     (cls instVarNames collect: [:nm | cls -> nm]),
     (cls class instVarNames collect: [:nm | cls class -> nm]),
     (cls classVarNames collect: [:nm | cls class -> nm]),
     cls theNonMetaClass methodDict values,
     cls theMetaClass methodDict values)]]
   -> { #id -> #outline.
        #label -> 'Outline'.
        #notifier -> ViSystemChangeNotifier.
        #view -> ViTreeView }.

   [:in :out | in do: [:object | out addAll: (
      {
         object isVariableBinding
            ifTrue: [object value isBehavior
               ifTrue: ['** inheritance **']
               ifFalse: ['** variables **']]
            ifFalse: [object isBehavior
               ifTrue: ['** traits **']
               ifFalse: [object methodClass organization
                          categoryOfElement: object selector]].
         object
      } asTuples "#(category object)" )]]
   -> { #next -> #groups "script reference" }.

} asScript openScriptWith: {ViPane}.
```

Our generic `#groups` script expects tuples in the form `#(category object)`. A category can be any object because it will be converted into its text representation via `#asString`. The script looks like this:

```
script := {
    "Sort contents in the groups."
    [:in :out | out addAll: (in sorted: [:tuple1 :tuple2 |
        tuple1 second asString <= tuple2 second asString])]
    -> { #id -> #groups.
            #view -> ViTreeView }.

    "Create group tuples from content tuples."
    [:in :out | out addAll: in groupByFirst]
    -> { }.

    "Sort groups."
    [:in :out | out addAll: (in sorted: [:group1 :group2 |
        group1 first asString <= group2 first asString])]
    -> { }.

    "Insert separators."
    [:in :out | in do: [:group |
        out add:
            "Helper tuple for group separation."
            { #dummy. {{ '--', group first asString, '--' }} }.
        out add:
            "Group contents."
            group ]]
    -> { }.

    "Expand group and discard group element."
    [:in :out | in do: [:group | out addAll: group second]]
    -> { }.
    [:in :out | in do: [:group | out addAll: group allButFirst]]
    -> { }.

    "Simplified property extraction."
    [:in :out | in do: [:object | out add: {
        #object -> object.
        #text -> (object isString
                    ifTrue: [object]
                    ifFalse: [object printString]).
        "Separators are not selectable."
        #selectable -> (object isString
                            and: [object beginsWith: '--']) not }]]
    -> { #isProperty -> true }

} asScript openScriptWith: # ( (a 1)(a 2)(b 10)(b 20)(c 100) ).
```

Note that we inserted non-selectable group separators in the same tree level. The alternative would be to use an entire level in the model tree for group objects. In both cases, the user would see objects in groups.

B.5 More Single-object Tools

In section 6.1 in figure 6.2, we showed several single-object tools for common Squeak and Vivide objects. We created the view with the following code snippet:

```
| container |
container := Morph new
  color: Color white;
  layoutPolicy: TableLayout new;
  listDirection: #leftToRight;
  wrapDirection: #topToBottom;
  layoutInset: 25;
  cellInset: 10;
  yourself.

{
  ViClassDefinitionEditorView.
  ViMethodEditorView.
  ViWorkspaceView.

  ViExplorerView.
  ViEditorView. "Auto wrapper for ViProcessEditor"
  ViProtocolEditorView.

  ViEditorView. "Auto wrapper for ViMemoEditor"
  ViEditorView. "Auto wrapper for ViClassCommentEditor"
  ViScriptEditorView.

} with: {

  Morph.
  Morph >> #fullDrawOn:.
  ActiveHand. "Binds 'self' in workspace."

  Morph new.
  [ self halt ] newProcess.
  ViProtocol named: #accessing inClass: String.

  'Vivide is a Squeak/Smalltalk-based programming environment and framework
  that supports low-effort construction of graphical tools by employing a
  data-driven perspective and a script-based programming model.' asText.
  ViClassComment new theClass: Morph.
  [:num | num * num] asScript.

} do: [:view :object |
   container addMorphBack:
      (ViPane new
         borderWidth: 2;
         borderColor: (Color gray: 215/255);
         objects: { object };
         currentScript: { [:m | #object -> m]
            -> { #view -> view } } asScript ;
         extent: 270@150;
         yourself)].

container openInWorld.
```

Note that we created `ViProtocol` and `ViClassComment` because Squeak has no dedicated classes for these objects but uses `ByteSymbol` and `Text` respectively. Also note that ViEditorView is an adapter-wrapper around editors, which are automatically selected for the particular object. Editors are implemented in regular Morphic; views are compatible with panes and scripts.

B.6 The "File Browser" Script

In section 6.2, we showed a file browser that has a single list view. The script behind that view transforms an instance of DirectoryEntryDirectory, which is a proxy for a directory in the file system, into instances of DirectoryEntry, which are directories or files. It adds "." and ".." as handles for the directory itself and its containing directory. The only pane in this browser has an object connection to itself. The script looks like this:

```
script := {
   [:entry | entry isDirectory
                ifTrue: [entry asFileDirectory] ifFalse: []]
   -> { #id -> #'af533889'. "Only for tree view."
        #view -> ListView. "Or TreeView."
        #yieldOn -> #(doubleClicked returnPressed)}.

   [:in :out | in do: [:directory | out addAll: (
      { '.' -> directory directoryEntry.
        '..' -> directory containingDirectory directoryEntry },
        directory entries )]].

   [:entry | {
      #object -> entry value. "All objects know this message."
      #text -> ( (entry respondsTo: #key) "For '.' and '..'"
                 ifTrue: [entry key] ifFalse: [entry name]).
      #icon -> (entry value isDirectory
         ifTrue: [UiFugueIcons folderHorizontalIcon]
         ifFalse: [UiFugueIcons documentIcon]) } ]
   -> { #next -> #'af533889' "Only for tree view."}

} asScript openScriptWith: {FileDirectory default directoryEntry}.
```

Publications

All publications relate to this dissertation's theme, which includes aspects of tool building, exploratory programming, agile practices, and team collaboration.

Journal Publications

- Fabio Niephaus, Tim Felgentreff, Tobias Pape, Robert Hirschfeld, and **Marcel Taeumel**. "Live Multi-language Development and Runtime Environments". In: *Journal on The Art, Science, and Engineering of Programming* 2.3 (Mar. 2018), pp. 8:1–8:30. ISSN: 2473-7321. DOI: 10.22152/programming-journal.org/2018/2/8.
- **Marcel Taeumel**, Stephanie Platz, Bastian Steinert, and Robert Hirschfeld. "Unravel Programming Sessions with THRESHER: Identifying Coherent and Complete Sets of Fine-granular Source Code Changes". In: *Information and Media Technologies* 12.1 (Mar. 2017), pp. 24–39. ISSN: 1881-0896. DOI: 10.11185/imt.12.24.
- Michael Perscheid, Benjamin Siegmund, **Marcel Taeumel**, and Robert Hirschfeld. "Studying the Advancement in Debugging Practice of Professional Software Developers". In: *Software Quality Journal* 25.1 (Mar. 2017), pp. 1–28. ISSN: 1573-1367. DOI: 10.1007/s11219-015-9294-2.

Conference Publications

- Patrick Rein, Robert Hirschfeld, and **Marcel Taeumel**. "Gramada: Immediacy in Programming Language Development". In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* Amsterdam, Netherlands: ACM, Oct. 2016, pp. 165–179. DOI: 10.1145/2986012.2986022.
- **Marcel Taeumel**, Michael Perscheid, Bastian Steinert, Jens Lincke, and Robert Hirschfeld. "Interleaving of Modification and Use in Data-driven Tool Development". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* Portland, Oregon, USA: ACM, Oct. 2014, pp. 185–200. DOI: 10.1145/2661136.2661150.
- **Marcel Taeumel**, Bastian Steinert, and Robert Hirschfeld. "The VIVIDE Programming Environment: Connecting Run-time Information with Programmers' System Knowledge". In: *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software.* Tucson, AZ, USA: ACM, Oct. 2012, pp. 117–126. DOI: 10.1145/2384592.2384604.

- Bastian Steinert, **Marcel Taeumel**, Jens Lincke, Tobias Pape, and Robert Hirschfeld. "CodeTalk: Conversations About Code". In: *2010 Eighth International Conference on Creating, Connecting and Collaborating through Computing.* La Jolla, CA, USA: IEEE, Jan. 2010, pp. 11–18. DOI: `10.1109/C5.2010.11`.

Workshop Publications

- Robert Hirschfeld, Tobias Dürschmid, Patrick Rein, and **Marcel Taeumel**. "Crosscutting Commentary: Narratives for Multi-party Mechanisms and Concerns". In: *Proceedings of the 10th International Workshop on Context-Oriented Programming.* Amsterdam, Netherlands: ACM, July 2018, pp. 39–47. DOI: `10.1145/3242921.3242927`.
- Jens Lincke, Stefan Ramson, Patrick Rein, Robert Hirschfeld, **Marcel Taeumel**, and Tim Felgentreff. "Designing a Live Development Experience for Web Components". In: *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Experience.* Vancouver, BC, Canada: ACM, Oct. 2017, pp. 28–35. DOI: `10.1145/3167109`.
- Patrick Rein, **Marcel Taeumel**, Robert Hirschfeld, and Michael Perscheid. "Exploratory Development of Data-intensive Applications: Sampling and Streaming of Large Data Sets in Live Programming Environments". In: *Companion to the First International Conference on the Art, Science and Engineering of Programming.* Brussels, Belgium: ACM, Apr. 2017, pp. 25:1–25:11. DOI: `10.1145/3079368.3079399`.
- Fabio Niephaus, Dale Henrichs, **Marcel Taeumel**, Tobias Pape, Tim Felgentreff, and Robert Hirschfeld. "smalltalkCI: A Continuous Integration Framework for Smalltalk Projects". In: *Proceedings of the 11th Edition of the International Workshop on Smalltalk Technologies.* Prague, Czech Republic: ACM, Aug. 2016, pp.3:1–3:9. DOI: `10.1145/2991041.2991044`.
- **Marcel Taeumel** and Robert Hirschfeld. "Evolving User Interfaces From Within Self-sustaining Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs". In: *Proceedings of the Programming Experience 2016 (PX/16) Workshop.* Rome, Italy: ACM, July 2016, pp. 43–59. DOI: `10.1145/2984380.2984386`.
- Astrid Thomschke, Daniel Stolpe, **Marcel Taeumel**, and Robert Hirschfeld. "Towards Gaze Control in Programming Environments". In: *Proceedings of the Programming Experience 2016 (PX/16) Workshop.* Rome, Italy: ACM, July 2016, pp. 27–32. DOI: `10.1145/2984380.2984384`.
- Benjamin Siegmund, Michael Perscheid, **Marcel Taeumel**, and Robert Hirschfeld. "Studying the Advancement in Debugging Practice of Professional Software Developers". In: *2014 IEEE International Symposium on Software Reliability Engineering Workshops.* Naples, Italy: IEEE, Nov. 2014, pp. 269–274. DOI: `10.1109/ISSREW.2014.36`.

- **Marcel Taeumel**, Tim Felgentreff, and Robert Hirschfeld. "Applying Data-driven Tool Development to Context-oriented Languages". In: *Proceedings of the 6th International Workshop on Context-oriented Programming.* Uppsala, Sweden: ACM, July 2014, pp. 1:1–1:7. DOI: `10.1145/2637066.2637067`.

Book Chapters

- Patrick Rein, **Marcel Taeumel**, and Robert Hirschfeld. "Towards Empirical Evidence on the Comprehensibility of Natural Language Versus Programming Language". In: *Design Thinking Research: Investigating Design Team Performance.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Springer, Sept. 2019, pp. 111–131. DOI: `10.1007/978-3-030-28960-7_7`.
- Patrick Rein, **Marcel Taeumel**, and Robert Hirschfeld. "Towards Exploratory Software Design: Environments for the Multi-Disciplinary Team". In: *Design Thinking Research: Looking Further: Design Thinking Beyond Solution-Fixation.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Springer, Sept. 2018, pp. 229–247. DOI: `10.1007/978-3-319-97082-0_12`.
- Patrick Rein, **Marcel Taeumel**, and Robert Hirschfeld. "Making the Domain Tangible: Implicit Object Lookup for Source Code Readability". In: *Design Thinking Research: Making Distinctions: Collaboration versus Cooperation.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Springer, Oct. 2017, pp. 171–194. DOI: `10.1007/978-3-319-60967-6_9`.
- **Marcel Taeumel** and Robert Hirschfeld. "Making Examples Tangible: Tool Building for Program Comprehension". In: *Design Thinking Research: Taking Breakthrough Innovation Home.* Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Springer, Aug. 2016, pp. 161–182. DOI: `10.1007/978-3-319-40382-3_11`.
- Bastian Steinert, **Marcel Taeumel**, Damien Cassou, and Robert Hirschfeld. "Adopting Design Practices for Programming". In: *Design Thinking Research: Measuring Performance in Context.* Springer, Aug. 2012, pp. 247–262. DOI: `10.1007/978-3-642-31991-4_14`.

Technical Reports

- Tom Beckmann, Justus Hildebrand, Corinna Jaschek, Eva Krebs, Alexander Löser, **Marcel Taeumel**, Tobias Pape, Lasse Fister, and Robert Hirschfeld. *The Font Engineering Platform: Collaborative Font Creation in a Self-supporting Programming Environment.* Tech. rep. 128. Potsdam, Germany: Hasso Plattner Institute, 2019. ISBN: 978-3-86956-464-7.
- Jakob Reschke, **Marcel Taeumel**, Tobias Pape, Fabio Niephaus, and Robert Hirschfeld. *Towards Version Control in Object-based Systems.* Tech. rep. 121. Potsdam, Germany: Hasso Plattner Institute, 2018. ISBN: 978-3-86956-430-2.

- Philipp Otto, Jaqueline Pollak, Daniel Werner, Felix Wolff, Bastian Steinert, Lauritz Thamsen, **Marcel Taeumel**, Jens Lincke, Robert Krahn, Daniel H. H. Ingalls, and Robert Hirschfeld. *Exploratory Authoring of Interactive Content in a Live Environment.* Tech. rep. 101. Potsdam, Germany: Hasso Plattner Institute, 2016. ISBN: 978-3-86956-346-6.
- Eva Maria Herbst, Fabian Maschler, Fabio Niephaus, Max Reimann, Julia Steier, Tim Felgentreff, Jens Lincke, **Marcel Taeumel**, Robert Hirschfeld, and Carsten Witt. *ecoControl - Design and Implementation of a Prototype for Optimizing Heterogeneous Energy Systems in Multi-family Residential Buildings.* Tech. rep. 93. Potsdam, Germany: Hasso Plattner Institute, 2015. ISBN: 978-3-86956-318-3.
- **Marcel Taeumel**. "Leveraging Programmers' Skills: Interleaving of Modification and Use in Data-driven Tool Development". In: *Proceedings of the 8th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering.* Tech. rep. 95, pp. 191–202. Potsdam, Germany: Hasso Plattner Institute, 2015. ISBN: 978-3-86956-320-6.
- **Marcel Taeumel**. "No Tools But Objects: Towards Direct Manipulation Programming Environments". In: *Proceedings of the 7th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering.* Tech. rep. 83, pp. 161–172. Potsdam, Germany: Hasso Plattner Institute, 2014. ISBN: 978-3-86956-273-5.
- **Marcel Taeumel**. "Interleaving Programming Tasks: Challenges for Interruption Handling in Programming Environments". In: *Proceedings of the 6th Ph.D. Retreat of the HPI Research School on Service-oriented Systems Engineering.* Tech. rep. 76, pp. 179–190. Potsdam, Germany: Hasso Plattner Institute, 2013. ISBN: 978-3-86956-256-8.

Bibliography

[1] IEEE Std 1178-1990. *The Scheme Programming Language*. IEEE Computer Society, 2008.

[2] ISO/IEC 13211-1:1995. *Prolog*. International Organization for Standardization, 1995.

[3] ISO/IEC 14977:1996(E). *Extended BNF*. International Organization for Standardization, 1996.

[4] ISO/IEC 9075:2016. *Structured Query Language (SQL)*. International Organization for Standardization, 2016.

[5] Harold Abelson, Gerald J. Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. 2nd ed. The MIT Press, July 1996. ISBN: 978-0-262-51087-5.

[6] Marc Abrams, Constantinos Phanouriou, Alan L. Batongbacal, Stephen M. Williams, and Jonathan E. Shuster. "UIML: An Appliance-independent XML User Interface Language". In: *Computer networks* 31.11–16 (May 1999), pp. 1695–1708. ISSN: 1389-1286. DOI: `10.1016/S1389-1286(99)00044-4`.

[7] BJ Allen-Conn and Kimberly Rose. *Powerful Ideas in the Classroom*. Viewpoints Research Institute, Inc., 2003. ISBN: 978-0-9743131-0-8.

[8] Erik M. Altmann and J. Gregory Trafton. "Task Interruption: Resumption Lag and the Role of Cues". In: *Proceedings of the 26th Annual Conference of the Cognitive Science (CogSci)*. Chicago, IL, USA: Lawrence Erlbaum Associates, Inc., Aug. 2004, pp. 43–48. ISBN: 978-0-8058-5464-0.

[9] Dimitar Asenov and Peter Müller. "Envision: A Fast and Flexible Visual Code Editor with Fluid Interactions (Overview)". In: *2014 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. Melbourne, VIC, Australia: IEEE, Aug. 2014, pp. 9–12. DOI: `10.1109/VLHCC.2014.6883014`.

[10] Michelle Q. W. Baldonado, Allison Woodruff, and Allan Kuchinsky. "Guidelines for Using Multiple Views in Information Visualization". In: *Proceedings of the Working Conference on Advanced Visual Interfaces*. Palermo, Italy: ACM, May 2000, pp. 110–119. DOI: `10.1145/345513.345271`.

[11] Nikola Banovic, Fanny Chevalier, Tovi Grossman, and George Fitzmaurice. "Triggering Triggers and Burying Barriers to Customizing Software". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Austin, TX, USA: ACM, May 2012, pp. 2717–2726. DOI: 10.1145/2207676.2208666.

[12] Douglas Barry and Torsten Stanienda. "Solving the Java Object Storage Problem". In: *Computer* 31.11 (Nov. 1998), pp. 33–40. ISSN: 0018-9162. DOI: 10.1109/2.730734.

[13] Kent Beck. *Smalltalk Best Practice Patterns*. Ed. by Joe Czerwinski. Prentice Hall, 1997. ISBN: 978-0-13-476904-2.

[14] Kent Beck and Cynthia Andres. *Extreme Programming Explained: Embrace Change*. 2nd ed. Addison-Wesley, Nov. 2004. ISBN: 978-0-321-27865-4.

[15] Alexandre Bergel, Damien Cassou, Stéphane Ducasse, and Jannik Laval. *Deep Into Pharo*. Square Bracket Associates, 2013. ISBN: 978-3-9523341-6-4.

[16] Alexandre Bergel, Stéphane Ducasse, Colin Putney, and Roel Wuyts. "Creating Sophisticated Development Tools with OmniBrowser". In: *Computer Languages, Systems and Structures* 34.2 (July 2008), pp. 109–129. ISSN: 1477-8424. DOI: 10.1016/j.cl.2007.05.005.

[17] Olav W. Bertelsen. "Design Artefacts: Towards a Design-oriented Epistemology". In: *Scandinavian Journal of Information Systems - Special Issue on Information Technology in Human Activity* 12.1 (Jan. 2001). http://dl.acm.org/citation.cfm?id=372668.372678, accessed 2018-11-10, pp. 15–27. ISSN: 1901-0990.

[18] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 4*. 2nd ed. Prentice Hall, Feb. 2008. ISBN: 978-0-13-235416-5.

[19] Alan Borning. "The Programming Language Aspects of ThingLab, a Constraint-Oriented Simulation Laboratory". In: *ACM Transactions on Programming Languages and Systems (TOPLAS)* 3.4 (Oct. 1981), pp. 353–387. ISSN: 1558-4593. DOI: 10.1145/357146.357147.

[20] Gilad Bracha, Peter von der Ahé, Vassili Bykov, Yaron Kashai, William Maddox, and Eliot Miranda. "Modules as Objects in Newspeak". In: *ECOOP 2010 – Object-Oriented Programming*. Maribor, Slovenia: Springer, June 2010, pp. 405–428. DOI: 10.1007/978-3-642-14107-2_20.

[21] Andrew Bragdon, Robert Zeleznik, Steven P. Reiss, Suman Karumuri, William Cheung, Joshua Kaplan, Christopher Coleman, Ferdi Adeputra, and Joseph J. LaViola Jr. "Code Bubbles: A Working Set-based Interface for Code Understanding and Maintenance". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Atlanta, GA, USA: ACM, Apr. 2010, pp. 2503–2512. DOI: 10.1145/1753326.1753706.

[22] John Brant, Brian Foote, Ralph E. Johnson, and Donald Roberts. “Wrappers to the Rescue”. In: *ECOOP '98 – Object-Oriented Programming*. Brussels, Belgium: Springer, July 1998, pp. 396–417. DOI: 10.1007/BFb0054101.

[23] Reinhard Budde and Heinz Züllighoven. “Software Tools in a Programming Workshop”. In: *Software Development and Reality Construction*. Springer, 1992, pp. 252–268. ISBN: 3-642-76819-9. DOI: 10.1007/978-3-642-76817-0_20.

[24] Philipp Bunge. “Scripting Browsers with Glamour”. MA thesis. University of Bern, Apr. 2009.

[25] Frank Buschmann, Regine Meunier, Hand Rohnert, Peter Sommerlad, and Michael Stal. *Pattern-oriented Software Architecture: A System of Patterns*. Vol. 1. John Wiley & Sons Ltd., Aug. 1996. ISBN: 978-0-471-95869-7.

[26] Vassili Bykov. “Hopscotch: Towards User Interface Composition”. In: *1st International Workshop on Academic Software Development Tools and Techniques 2008 (WASDeTT)*. http://scg.unibe.ch/download/wasdett/wasdett2008-paper03.pdf, accessed 2018-11-09. Paphos, Cyprus: University of Berne, July 2008. DOI: 10.1007/978-3-642-02047-6_10.

[27] Glauco de F. Carneiro, Marcos Silva, Leandra Mara, Eduardo Figueiredo, Claudio Sant'Anna, Alessandro Garcia, and Manoel Mendonca. “Identifying Code Smells with Multiple Concern Views”. In: *2010 Brazilian Symposium on Software Engineering*. Salvador, Bahia, Brazil: IEEE, Sept. 2010, pp. 128–137. DOI: 10.1109/SBES.2010.21.

[28] John M. Carroll and Mary B. Rosson. “Paradox of the Active User”. In: *Interfacing Thought: Cognitive Aspects of Human-Computer Interaction*. The MIT Press, Mar. 1987. Chap. 5, pp. 80–111. ISBN: 978-0-262-53221-1.

[29] Andrei Chiş, Tudor Gîrba, and Oscar Nierstrasz. “The Moldable Debugger: A Framework for Developing Domain-specific Debuggers”. In: *Software Language Engineering*. Västerås, Sweden: Springer, Sept. 2014, pp. 102–121. DOI: 10.1007/978-3-319-11245-9_6.

[30] Andrei Chiş, Oscar Nierstrasz, Aliaksei Syrel, and Tudor Gîrba. “The Moldable Inspector”. In: *2015 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software (Onward!)* Pittsburgh, PA, USA: ACM, Oct. 2015, pp. 44–60. DOI: 10.1145/2814228.2814234.

[31] Dave Collins. *Designing Object-oriented User Interfaces*. The Benjamin/Cummings Publishing Company, Inc., 1995. ISBN: 978-0-8053-5350-1.

[32] William R. Cook. “Interfaces and Specifications for the Smalltalk-80 Collection Classes”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages, and Applications*. Vancouver, BC, Canada: ACM, Oct. 1992, pp. 1–15. DOI: 10.1145/141936.141938.

[33] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. 3rd ed. The MIT Press, July 2009. ISBN: 978-0-262-03384-8.

[34] Bas Cornelissen, Danny Holten, Andy Zaidman, Leon Moonen, Jarke J. van Wijk, and Arie van Deursen. "Understanding Execution Traces Using Massive Sequence and Circular Bundle Views". In: *15th IEEE International Conference on Program Comprehension (ICPC '07)*. Banff, AB, Canada: IEEE, June 2007, pp. 49–58. DOI: `10.1109/ICPC.2007.39`.

[35] Bas Cornelissen, Andy Zaidman, Arie van Deursen, Leon Moonen, and Rainer Koschke. "A Systematic Survey of Program Comprehension through Dynamic Analysis". In: *IEEE Transactions on Software Engineering* 35.5 (Apr. 2009), pp. 684–702. ISSN: 0098-5589. DOI: `10.1109/TSE.2009.28`.

[36] Joelle Coutaz. "Architecture Models for Interactive Software". In: *ECOOP 89: Proceedings of the Third European Conference on Object-oriented Programming*. Ed. by Stephen Cook. Nottingham, UK: Cambridge University Press, July 1989, pp. 383–399. ISBN: 978-0-521-38232-8.

[37] Roger F. Crew. "ASTLOG: A Language for Examining Abstract Syntax Trees". In: *Proceedings of the Conference on Domain-Specific Languages (DSL)*. `http://dl.acm.org/citation.cfm?id=1267950.1267968`, accessed 2018-11-10. Santa Barbara, CA, USA: USENIX, Oct. 1997, 18:1–18:14.

[38] Mihaly Csikszentmihalyi. *Flow: The Psychology of Optimal Experience*. Harper Perennial Modern Classics, July 2008. ISBN: 978-0-06-133920-2.

[39] Kris De Volder. "JQuery: A Generic Code Browser with a Declarative Configuration Language". In: *Practical Aspects of Declarative Languages*. Charleston, SC, USA: Springer, Jan. 2006, pp. 88–102. DOI: `10.1007/11603023_7`.

[40] Robert DeLine. "Staying Oriented with Software Terrain Maps". In: *Workshop on Visual Languages and Computation*. `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/vlc05-final.pdf`, accessed 2018-11-09. Microsoft Research, Jan. 2005, pp. 309–314.

[41] Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen, and Steven P. Reiss. "Debugger Canvas: Industrial Experience with the Code Bubbles Paradigm". In: *2012 34th International Conference on Software Engineering (ICSE)*. Zurich, Switzerland: IEEE, June 2012, pp. 1064–1073. DOI: `10.1109/ICSE.2012.6227113`.

[42] Robert DeLine and Kael Rowan. "Code Canvas: Zooming Towards Better Development Environments". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 2*. Cape Town, South Africa: ACM, May 2010, pp. 207–210. DOI: `10.1145/1810295.1810331`.

[43] L. Peter Deutsch. "The Past, Present and Future of Smalltalk". In: *ECOOP 89: Proceedings of the Third European Conference on Object-oriented Programming*. Ed. by Stephen Cook. Nottingham, UK: Cambridge University Press, July 1989, pp. 73–87. ISBN: 978-0-521-38232-8.

[44] Stephan Diehl. *Software Visualization: Visualizing the Structure, Behaviour, and Evolution of software*. Springer, 2007. ISBN: 978-3-540-46504-1.

[45] Andrea A. diSessa and Harold Abelson. "Boxer: A Reconstructible Computational Medium". In: *Communications of the ACM* 29.9 (Sept. 1986), pp. 859–868. ISSN: 1557-7317. DOI: 10.1145/6592.6595.

[46] Mark Doernhoefer. "Surfing the Net for Software Engineering Notes". In: *ACM SIGSOFT Software Engineering Notes* 37.3 (May 2012). "Technical Debt", pp. 10–17. ISSN: 0163-5948. DOI: 10.1145/2180921.2180928.

[47] Michael Eisenberg. "Programmable Applications: Interpreter Meets Interface". In: *ACM SIGCHI Bulletin* 27.2 (Apr. 1995), pp. 68–93. ISSN: 0736-6906. DOI: 10.1145/202511.202528.

[48] Michael Eisenberg and Gerhard Fischer. "Programmable Design Environments: Integrating End-user Programming with Domain-oriented Assistance". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Boston, MA, USA: ACM, Apr. 1994, pp. 431–437. DOI: 10.1145/191666.191813.

[49] Douglas C. Engelbart. *Augmenting Human Intellect: A Conceptual Framework*. Tech. rep. Menlo Park, CA, USA: Stanford Research Institute, Oct. 1963.

[50] William K. English, Douglas C. Engelbart, and Melvyn L. Berman. "Display-Selection Techniques for Text Manipulation". In: *IEEE Transactions on Human Factors in Electronics* 8.1 (Mar. 1967), pp. 5–15. ISSN: 2168-2852. DOI: 10.1109/THFE.1967.232994.

[51] Johan Fabry, Andy Kellens, and Stéphane Ducasse. "AspectMaps: A Scalable Visualization of Join Point Shadows". In: *2011 IEEE 19th International Conference on Program Comprehension*. Kingston, ON, Canada: IEEE, June 2011, pp. 121–130. DOI: 10.1109/ICPC.2011.11.

[52] Steve Faulkner, Arron Eicholz, Travis Leithead, Alex Danilo, and Sangwhan Moon. *Hypertext Markup Language (HTML) 5.2*. World Wide Web Consortium (W3C), 2017.

[53] Glauco de Figueiredo Carneiro and Manoel Gomes de Mendonça Neto. "SourceMiner: Towards an Extensible Multi-perspective Software Visualization Environment". In: *Enterprise Information Systems*. Angers, France: Springer, July 2013, pp. 242–263. DOI: 10.1007/978-3-319-09492-2_15.

[54] Gerhard Fischer. "Domain-oriented Design Environments". In: *Automated Software Engineering* 1.2 (June 1994), pp. 177–203. ISSN: 1573-7535. DOI: 10.1007/BF00872289.

[55] Scott D. Fleming, Chris Scaffidi, David Piorkowski, Margaret Burnett, Rachel Bellamy, Joseph Lawrance, and Irwin Kwan. "An Information Foraging Theory Perspective on Tools for Debugging, Refactoring, and Reuse Tasks". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22.2 (Mar. 2013), 14:1–14:41. ISSN: 1557-7392. DOI: 10.1145/2430545.2430551.

[56] Martin Fowler. *Domain-Specific Languages*. Addison-Wesley, Sept. 2010. ISBN: 978-0-321-71294-3.

[57] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Ed. by J. Carter Shanklin. Addison-Wesley, June 1999. ISBN: 978-0-201-48567-7.

[58] Susan Fowler. *GUI Design Handbook*. Ed. by Scott Grillo and Pamela A. Pelton. McGraw-Hill, 1998. ISBN: 978-0-07-059274-2.

[59] Thomas Fritz and Gail C. Murphy. "Using Information Fragments to Answer the Questions Developers Ask". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. Cape Town, South Africa: ACM, May 2010, pp. 175–184. DOI: 10.1145/1806799.1806828.

[60] Richard P. Gabriel. "I Throw Itching Powder at Tulips". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Portland, OR, USA: ACM, Oct. 2014, pp. 301–319. DOI: 10.1145/2661136.2661155.

[61] Erich Gamma and Kent Beck. *Contributing to Eclipse: Principles, Patterns, and Plug-ins*. Addison-Wesley, 2004. ISBN: 978-0-321-20575-9.

[62] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Abstraction of Reusable Object-oriented Software*. Addison-Wesley, 1995. ISBN: 978-0-201-63361-0.

[63] David Garlan. "Views for Tools in Integrated Environments". In: *Advanced Programming Environments*. Trondheim, Norway: Springer, June 1986, pp. 314–343. DOI: 10.1007/3-540-17189-4_105.

[64] Adele Goldberg. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, Dec. 1983. ISBN: 978-0-201-11372-3.

[65] Adele Goldberg and David Robson. *Smalltalk-80: The Language and its Implementation*. Addison-Wesley, Dec. 1983. ISBN: 978-0-201-11371-6.

[66] Victor M. González and Gloria Mark. "Constant, Constant, Multi-Tasking Craziness: Managing Multiple Working Spheres". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Vienna, Austria: ACM, Apr. 2004, pp. 113–120. DOI: 10.1145/985692.985707.

[67] Danny Goodman. *The Complete HyperCard 2.2 Handbook*. 4th ed. Vol. 1. iUniverse.com, Inc., Dec. 1998. ISBN: 978-0-9665514-2-6.

[68] Lars Grammel, Chris Bennett, Melanie Tory, and Margaret-Anne Storey. "A Survey of Visualization Construction User Interfaces". In: *EuroVis - Short Papers*. Ed. by Mario Hlawitschka and Tino Weinkauf. Leipzig, Germany: Eurographics, June 2013, pp. 19–23. DOI: `10.2312/PE.EuroVisShort.EuroVisShort2013.019-023`.

[69] Saul Greenberg. *The Computer User as Toolsmith: The Use, Reuse and Organization of Computer-based Tools*. Ed. by Nicholas J. Long. Cambridge University Press, Jan. 1993. ISBN: 978-0-521-40430-3.

[70] Sebastian Hahn, Jonas Trümper, Dominik Moritz, and Jürgen Döllner. "Visualization of Varying Hierarchies by Stable Layout of Voronoi Treemaps". In: *Proceedings of the 5th International Conference on Information Visualization Theory and Applications*. Lisbon, Portugal: SciTePress, Jan. 2014, pp. 50–58. DOI: `10.5220/0004686200500058`.

[71] Elnar Hajiyev, Mathieu Verbaere, and Oege de Moor. "codeQuest: Scalable Source Code Queries with Datalog". In: *ECOOP 2006 – Object-Oriented Programming*. Nantes, France: Springer, July 2006, pp. 2–27. DOI: `10.1007/11785477_2`.

[72] Nicholas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. "The Synchronous Data Flow Programming Language LUSTRE". In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320. ISSN: 0018-9219. DOI: `10.1109/5.97300`.

[73] Austin Z. Henley and Scott D. Fleming. "The Patchworks Code Editor: Toward Faster Navigation with Less Code Arranging and Fewer Navigation Mistakes". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Toronto, ON, Canada: ACM, Apr. 2014, pp. 2511–2520. DOI: `10.1145/2556288.2557073`.

[74] Dugald R. Hutchings and John Stasko. "Revisiting display space management: understanding current practice to inform next-generation design". In: *Proceedings of Graphics Interface 2004*. Waterloo, ON, Canada: Canadian Human-Computer Communications Society, May 2004, pp. 127–134. ISBN: 978-1-56881-227-4.

[75] Edwin L. Hutchins, James D. Hollan, and Donald A. Norman. "Direct Manipulation Interfaces". In: *Human-Computer Interaction* 1.4 (Dec. 1985), pp. 311–338. ISSN: 1532-7051. DOI: `10.1207/s15327051hci0104_2`.

[76] Daniel H. H. Ingalls. "Back to the Future Once More". Unpublished draft. Sept. 2000.

[77] Daniel H. H. Ingalls, Ted Kaehler, John H. Maloney, Scott Wallace, and Alan C. Kay. "Back to the Future: The Story of Squeak, a Practical Smalltalk Written in Itself". In: *Proceedings of the 12th ACM SIGPLAN Conference on*

Object-oriented Programming, Systems, Languages, and Applications. Atlanta, GA, USA: ACM, Oct. 1997, pp. 318–326. DOI: 10.1145/263700.263754.

[78] Daniel H. H. Ingalls, Krzysztof Palacz, Stephen Uhler, Antero Taivalsaari, and Tommi Mikkonen. “The Lively Kernel A Self-supporting System on a Web Page”. In: *Self-Sustaining Systems*. Potsdam, Germany: Springer, May 2008, pp. 31–50. DOI: 10.1007/978-3-540-89275-5_2.

[79] Daniel H. H. Ingalls, Scott Wallace, Yu-Ying Chow, Frank Ludolph, and Ken Doyle. “Fabrik: A Visual Programming Environment”. In: *Conference Proceedings on Object-oriented Programming Systems, Languages and Applications*. San Diego, CA, USA: ACM, Sept. 1988, pp. 176–190. DOI: 10.1145/62083.62100.

[80] Jeff Johnson, Teresa L. Roberts, William Verplank, David C. Smith, Charles H. Irby, Marian Beard, and Kevin Mackey. “The Xerox Star: A Retrospective”. In: *Computer* 22.9 (Sept. 1989), pp. 11–26. ISSN: 0018-9162. DOI: 10.1109/2.35211.

[81] Stephen C. Johnson. *Lint, a C Program Checker*. Tech. rep. Murray Hill, NJ, USA: Bell Laboratories, July 1978.

[82] Natalia Juristo and Ana M. Moreno. *Basics of Software Engineering Experimentation*. Kluwer Academic Publishers, Feb. 2010. ISBN: 978-1-4419-5011-6.

[83] Daniel Kahneman. *Thinking, Fast and Slow*. Pengiun Books, May 2012. ISBN: 978-0-14-103357-0.

[84] Joshua Kerievsky. *Refactoring to Patterns*. Ed. by Joshua Kerievsky. Addison-Wesley, 2005. ISBN: 978-0-321-21335-8.

[85] Brian W. Kernighan and Phillip J. Plauger. *Software Tools*. Addison-Wesley, June 1976. ISBN: 978-0-201-03669-5.

[86] Mik Kersten and Gail C. Murphy. “Mylar: A Degree-of-Interest Model for IDEs”. In: *Proceedings of the 4th International Conference on Aspect-oriented Software Development*. Chicago, IL, USA: ACM, Mar. 2005, pp. 159–168. DOI: 10.1145/1052898.1052912.

[87] Yit Phang Khoo, Jeffrey S. Foster, and Michael Hicks. “Expositor: Scriptable Time-Travel Debugging with First-Class Traces”. In: *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 352–361. DOI: 10.1109/ICSE.2013.6606581.

[88] Holger M. Kienle and Hausi A. Müller. “Requirements of Software Visualization Tools: A Literature Survey”. In: *2007 4th IEEE International Workshop on Visualizing Software for Understanding and Analysis*. IEEE, June 2007, pp. 2–9. DOI: 10.1109/VISSOF.2007.4290693.

[89] Andrew J. Ko, Robin Abraham, Laura Beckwith, Alan Blackwell, Margaret M. Burnett, Martin Erwig, Chris Scaffidi, Joseph Lawrance, Henry Lieberman, Brad A. Myers, Mary B. Rosson, Gregg Rothermel, Mary Shaw, and Susan Wiedenbeck. "The State of the Art in End-user Software Engineering". In: *ACM Computing Surveys (CSUR)* 43.3 (Apr. 2011), 21:1–21:44. ISSN: 1557-7341. DOI: 10.1145/1922649.1922658.

[90] Andrew J. Ko, Htet Htet Aung, and Brad A. Myers. "Design Requirements for more Flexible Structured Editors from a Study of Programmers' Text Editing". In: *CHI '05 Extended Abstracts on Human Factors in Computing Systems*. Portland, OR, USA: ACM, Apr. 2005, pp. 1557–1560. DOI: 10.1145/1056808.1056965.

[91] Andrew J. Ko, Robert DeLine, and Gina Venolia. "Information Needs in Collocated Software Development Teams". In: *29th International Conference on Software Engineering (ICSE'07)*. Minneapolis, MN, USA: IEEE, May 2007, pp. 344–353. DOI: 10.1109/ICSE.2007.45.

[92] Andrew J. Ko, Thomas D. LaToza, and Margaret M. Burnett. "A Practical Guide to Controlled Experiments of Software Engineering Tools with Human Participants". In: *Empirical Software Engineering* 20.1 (Feb. 2015), pp. 110–141. ISSN: 1573-7616. DOI: 10.1007/s10664-013-9279-3.

[93] Andrew J. Ko and Brad A. Myers. "Barista: An Implementation Framework for Enabling New Tools, Interaction Techniques and Views in Code Editors". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. Montréal, QC, Canada: ACM, Apr. 2006, pp. 387–396. DOI: 10.1145/1124772.1124831.

[94] Andrew J. Ko and Brad A. Myers. "Citrus: A Language and Toolkit for Simplifying the Creation of Structured Editors for Code and Data". In: *Proceedings of the 18th Annual ACM Symposium on User Interface Software and Technology*. Seattle, WA, USA: ACM, Oct. 2005, pp. 3–12. DOI: 10.1145/1095034.1095037.

[95] Andrew J. Ko, Brad A. Myers, Michael J. Coblenz, and Htet Htet Aung. "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks". In: *IEEE Transactions on Software Engineering* 32.12 (Nov. 2006), pp. 971–987. ISSN: 0098-5589. DOI: 10.1109/TSE.2006.116.

[96] Bastian Kruck, Stefan Lehmann, Christoph Keßler, Jakob Reschke, Tim Felgentreff, Jens Lincke, and Robert Hirschfeld. "Multi-level Debugging for Interpreter Developers". In: *Companion Proceedings of the 15th International Conference on Modularity*. Málaga, Spain: ACM, Mar. 2016, pp. 91–93. DOI: 10.1145/2892664.2892679.

[97] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk*. Vol. 1. Prentice Hall, May 1990. ISBN: 978-0-13-468414-7.

[98] Wilf R. LaLonde and John R. Pugh. *Inside Smalltalk*. Vol. 2. Prentice Hall, Jan. 1991. ISBN: 978-0-13-465964-0.

[99] Michele Lanza and Radu Marinescu. *Object-Oriented Metrics in Practice: Using Software Metrics to Characterize, Evaluate, and Improve the Design of Object-Oriented Systems*. Springer, 2007. ISBN: 978-3-540-24429-5. DOI: 10.1007/3-540-39538-5.

[100] Thomas D. LaToza and Brad A. Myers. "Developers Ask Reachability Questions". In: *Proceedings of the 32nd International Conference on Software Engineering - Volume 1*. Cape Town, South Africa: ACM, May 2010, pp. 185–194. DOI: 10.1145/1806799.1806829.

[101] Thomas D. LaToza, Gina Venolia, and Robert DeLine. "Maintaining Mental Models: A Study of Developer Work Habits". In: *Proceedings of the 28th International Conference on Software Engineering*. Shanhai, China: ACM, May 2006, pp. 492–501. DOI: 10.1145/1134285.1134355.

[102] Raimondas Lencevicius, Urs Hölzle, and Ambuj K. Singh. "Dynamic Query-based Debugging of Object-oriented Programs". In: *Automated Software Engineering* 10.1 (Jan. 2003), pp. 39–74. ISSN: 1573-7535. DOI: 10.1023/A:1021816917888.

[103] Quentin Limbourg, Jean Vanderdonckt, Benjamin Michotte, Laurent Bouillon, and Víctor López-Jaquero. "USIXML: A Language Supporting Multi-path Development of User Interfaces". In: *Engineering Human Computer Interaction and Interactive Systems*. Hamburg, Germany: Springer, July 2004, pp. 200–220. DOI: 10.1007/11431879_12.

[104] Jens Lincke and Robert Hirschfeld. "Scoping Changes in Self-supporting Development Environments Using Context-oriented Programming". In: *Proceedings of the International Workshop on Context-Oriented Programming*. Beijing, China: ACM, June 2012. DOI: 10.1145/2307436.2307438.

[105] Jens Lincke and Robert Hirschfeld. "User-evolvable Tools in the Web". In: *Proceedings of the 9th International Symposium on Open Collaboration*. Hong Kong, China: ACM, Aug. 2013, 19:1–19:8. DOI: 10.1145/2491055.2491074.

[106] Jens Lincke, Robert Krahn, Daniel H. H. Ingalls, and Robert Hirschfeld. "Lively Fabrik - A Web-based End-user Programming Environment". In: *2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing*. Kyoto, Japan: IEEE, Jan. 2009, pp. 11–19. DOI: 10.1109/C5.2009.8.

[107] Wendy E. Mackay. "Triggers and Barriers to Customizing Software". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New Orleans, LA, USA: ACM, Apr. 1991, pp. 153–160. DOI: 10.1145/108844.108867.

[108] John H. Maloney. “An Introduction to Morphic: The Squeak User Interface Framework”. In: *Squeak: Open Personal Computing and Multimedia*. Ed. by Mark Guzdial and Kim Rose. Prentice Hall, 2002. Chap. 2, pp. 39–67. ISBN: 978-0-13-028091-6.

[109] John H. Maloney, Mitchel Resnick, Natalie Rusk, Brian Silverman, and Evelyn Eastmond. “The Scratch Programming Language and Environment”. In: *ACM Transactions on Computing Education (TOCE)* 10.4 (Nov. 2010), 16:1–16:15. ISSN: 1946-6226. DOI: 10.1145/1868358.1868363.

[110] John H. Maloney and Randall B. Smith. “Directness and Liveness in the Morphic User Interface Construction Environment”. In: *Proceedings of the 8th Annual ACM Symposium on User Interface and Software Technology*. Pittsburgh, PA, USA: ACM, Nov. 1995, pp. 21–28. DOI: 10.1145/215585.215636.

[111] Guillaume Marceau, Gregory H Cooper, Jonathan P. Spiro, Shriram Krishnamurthi, and Steven P. Reiss. “The Design and Implementation of a Dataflow Language for Scriptable Debugging”. In: *Automated Software Engineering* 14.1 (Mar. 2007), pp. 59–86. ISSN: 1573-7535. DOI: 10.1007/s10515-006-0003-z.

[112] Deborah J. Mayhew. *Principles and Guidelines in Software User Interface Design*. Ed. by Harriet Tellem. Prentice Hall, 1992. ISBN: 978-0-13-721929-2.

[113] Anneliese von Mayrhauser and A. Marie Vans. “Program Comprehension during Software Maintenance and Evolution”. In: *Computer* 28.8 (Aug. 1995), pp. 44–55. ISSN: 0018-9162. DOI: 10.1109/2.402076.

[114] John McCarthy, Paul W. Abrahams, Daniel J. Edwards, Timothy P. Hart, and Michael I. Levin. *LISP 1.5 Programmer's Manual*. 2nd ed. The MIT Press, 1965. ISBN: 978-0-262-13011-0.

[115] Stanley A. McChrystal, Tantum Collins, David Silverman, and Chris Fussell. *Team of Teams: New Rules of Engagement for a Complex World*. Penguin, Nov. 2015. ISBN: 978-0-241-25083-9.

[116] Kim Mens, Andy Kellens, Frédéric Pluquet, and Roel Wuyts. “Co-evolving Code and Design with Intensional Views: A Case Study”. In: *Computer Languages, Systems and Structures* 32.2 (July 2006), pp. 140–156. ISSN: 1477-8424. DOI: 10.1016/j.cl.2005.09.002.

[117] Kim Mens, Tom Mens, and Michel Wermelinger. “Maintaining Software Through Intentional Cource-code Views”. In: *Proceedings of the 14th International Conference on Software Engineering and Knowledge Engineering*. Ischia, Italy: ACM, July 2002, pp. 289–296. DOI: 10.1145/568760.568812.

[118] Kim Mens, Isabel Michiels, and Roel Wuyts. “Supporting Software Development Through Declaratively Codified Programming Patterns”. In: *Expert Systems with Applications* 23.4 (Nov. 2002), pp. 405–413. ISSN: 0957-4174. DOI: 10.1016/S0957-4174(02)00076-3.

[119] Bertrand Meyer. *Object-oriented Software Construction*. 2nd ed. Prentice Hall, Nov. 1998. ISBN: 978-0-13-629155-8.

[120] George A. Miller. "The Magical Number Seven, Plus or Minus Two: Some Limits on our Capacity for Processing Information." In: *Psychological Review* 63.2 (Mar. 1956), pp. 81–97. ISSN: 0033-295X. DOI: `10.1037/h0043158`.

[121] Roberto Minelli, Andrea Mocci, Romain Robbes, and Michele Lanza. "Taming the IDE with Fine-grained Interaction Data". In: *2016 IEEE 24th International Conference on Program Comprehension (ICPC)*. A: IEEE, May 2016, 11:1–11:10. DOI: `10.1109/ICPC.2016.7503714`.

[122] Yoshiro Miyata and Donald A. Norman. "Psychological Issues in Support of Multiple Activities". In: *User Centered System Design: New Perspectives on Human-Computer Interaction*. Ed. by Donald A. Norman and Stephen W. Draper. Lawrence Erlbaum Associates, Inc., 1986, pp. 265–284. ISBN: 978-0-89859-872-8.

[123] Hausi A. Müller and Karl Klashinsky. "Rigi: A System for Programming-in-the-large". In: *Proceedings of the 10th International Conference on Software Engineering*. Singapore: IEEE, Apr. 1988, pp. 80–86. DOI: `10.1109/ICSE.1988.93690`.

[124] Hausi A. Müller, Scott R. Tilley, and Kenny Wong. "Understanding Software Systems using Reverse Engineering Technology Perspectives from the Rigi Project". In: *Proceedings of the 1993 Conference of the Centre for Advanced Studies on Collaborative Research: Software Engineering - Volume 1*. Toronto, ON, Cananda: IBM Press, Oct. 1993, pp. 217–226. DOI: `10.1145/962309`.

[125] Gail C. Murphy. "Getting to Flow in Software Development". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Portland, OR, USA: ACM, Oct. 2014, pp. 269–281. DOI: `10.1145/2661136.2661158`.

[126] Gail C. Murphy, Mik Kersten, Martin P. Robillard, and Davor Čubranić. "The Emergent Structure of Development Tasks". In: *ECOOP 2005 – Object-Oriented Programming*. Glasgow, UK: Springer, July 2005, pp. 33–48. DOI: `10.1007/11531142_2`.

[127] Brad A. Myers. "A Taxonomy of Window Manager User Interfaces". In: *IEEE Computer Graphics and Applications* 8.5 (Sept. 1988), pp. 65–84. ISSN: 0272-1716. DOI: `10.1109/38.7762`.

[128] Brad A. Myers. "User Interface Software Tools". In: *ACM Transactions on Computer-Human Interaction (TOCHI)* 2.1 (Mar. 1995), pp. 64–103. ISSN: 1557-7325. DOI: `10.1145/200968.200971`.

[129] Brad A. Myers, Scott E. Hudson, and Randy Pausch. "Past, Present, and Future of User Interface Software Tools". In: *ACM Transactions on Computer-Human Interaction (TOCHI) - Special Issue on Human-Computer Interaction in the New Millennium, Part 1* 7.1 (Mar. 2000), pp. 3–28. ISSN: 1557-7325. DOI: `10.1145/344949.344959`.

[130] Peter Naur. "Programming as Theory Building". In: *Microprocessing and Microprogramming* 15.5 (May 1985), pp. 253–261. ISSN: 0165-6074. DOI: `10.1016/0165-6074(85)90032-8`.

[131] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, Nov. 1994. ISBN: 978-0-12-518406-9.

[132] Oscar Nierstrasz, Stéphane Ducasse, and Tudor Gîrba. "The Story of Moose: An Agile Reengineering Environment". In: *Proceedings of the 10th European Software Engineering Conference held jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. Lisbon, Portugal: ACM, Sept. 2005, pp. 1–10. DOI: `10.1145/1081706.1081707`.

[133] Donald A. Norman. "Cognitive Engineering". In: *User Centered System Design: New Perspectives on Human-Computer Interaction*. Ed. by Donald A. Norman and Stephen W. Draper. Lawrence Erlbaum Associates, Inc., 1986, pp. 31–61. ISBN: 978-0-89859-872-8.

[134] Donald A. Norman. *The Design of Everyday Things*. Basic Books, 2002. ISBN: 978-0-465-06710-7.

[135] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. "Comparison of JSON and XML Data Interchange Formats: A Case Study". In: *22nd International Conference on Computer Applications in Industry and Engineering (CAINE-2009)*. Ed. by Dunren Che. San Francisco, CA, USA: ISCA, Nov. 2009, pp. 157–162. ISBN: 978-1-880843-73-4.

[136] Yoshiki Ohshima, Aran Lunzer, Bert Freudenberg, and Ted Kaehler. "KScript and KSWorld: A Time-aware and Mostly Declarative Language and Interactive GUI Framework". In: *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Indianapolis, IN, USA: ACM, Oct. 2013, pp. 117–134. DOI: `10.1145/2509578.2509590`.

[137] Fernando Olivero. "Object-focused Environments Revisited". PhD thesis. Università della Svizzera Italiana, Faculty of Informatics, Apr. 2013.

[138] Fernando Olivero, Michele Lanza, Marco D'ambros, and Romain Robbes. "Tracking Human-centric Controlled Experiments with Biscuit". In: *Proceedings of the ACM 4th Annual Workshop on Evaluation and Usability of Programming Languages and Tools*. Tucson, AZ, USA: ACM, Oct. 2012, pp. 1–6. DOI: `10.1145/2414721.2414723`.

[139] Fernando Olivero, Michele Lanza, and Marco D'ambros. "Object-focused Environments Revisited". In: *Science of Computer Programming* 98.3 (Feb. 2015), pp. 394–407. ISSN: 0167-6423. DOI: 10.1016/j.scico.2013.07.011.

[140] John K. Ousterhout. "Scripting: Higher Level Programming for the 21st Century". In: *Computer* 31.3 (Mar. 1998), pp. 23–30. ISSN: 0018-9162. DOI: 10.1109/2.660187.

[141] Chris Parnin and Carsten Gorg. "Building Usage Contexts During Program Comprehension". In: *14th IEEE International Conference on Program Comprehension (ICPC'06)*. Athens, Greece: IEEE, June 2006, pp. 13–22. DOI: 10.1109/ICPC.2006.14.

[142] Chris Parnin and Spencer Rugaber. "Programmer Information Needs After Memory Failure". In: *2012 20th IEEE International Conference on Program Comprehension (ICPC)*. Passau, Germany: IEEE, June 2012, pp. 123–132. DOI: 10.1109/ICPC.2012.6240479.

[143] Chris Parnin and Spencer Rugaber. "Resumption Strategies for Interrupted Programming Tasks". In: *Software Quality Journal* 19.1 (Mar. 2011), pp. 5–34. ISSN: 1573-1367. DOI: 10.1007/s11219-010-9104-9.

[144] Richard Pawson and Robert Matthews. *Naked Objects*. John Wiley & Sons, Ltd., Dec. 2002. ISBN: 978-0-470-84420-5.

[145] Michael Perscheid, Michael Haupt, Robert Hirschfeld, and Hidehiko Masuhara. "Test-driven Fault Navigation for Debugging Reproducible Failures". In: *Information and Media Technologies* 7.4 (Dec. 2012), pp. 1377–1400. ISSN: 1881-0896. DOI: 10.11185/imt.7.1377.

[146] Michael Perscheid, Bastian Steinert, Robert Hirschfeld, Felix Geller, and Michael Haupt. "Immediacy through Interactivity: Online Analysis of Run-time Behavior". In: *2010 17th Working Conference on Reverse Engineering*. Beverly, MA, USA: IEEE, Oct. 2010, pp. 77–86. DOI: 10.1109/WCRE.2010.17.

[147] Michael Perscheid, David Tibbe, Martin Beck, Stefan Berger, Peter Osburg, Jeff Eastman, Michael Haupt, and Robert Hirschfeld. *An Introduction to Seaside*. Software Architecture Group, Hasso Plattner Institute, Apr. 2008. ISBN: 978-3-00-023645-7.

[148] João F. N. Pimentel, Vanessa Braganholo, Leonardo Murta, and Juliana Freire. "Collecting and Analyzing Provenance on Interactive Notebooks: When IPython meets noWorkflow". In: *Proceedings of the 7th USENIX Conference on Theory and Practice of Provenance*. http://dl.acm.org/citation.cfm?id=2814579.2814589, accessed 2018-11-10. Edinburgh, Scotland: USENIX, July 2015, pp. 155–167.

[149] Mike Potel. *MVP: Model-View-Presenter - The Taligent Programming Model for C++ and Java*. Tech. rep. Taligent, Inc., 1996.

[150] Eric Prud'hommeaux and Andy Seaborn. *SPARQL Query Language for RDF*. World Wide Web Consortium (W3C), 2008.

[151] Robert Purvy, Jerry Farrell, and Paul Klose. "The Design of Star's Records Processing: Data Processing for the Noncomputer Professional". In: *ACM Transactions on Information Systems (TOIS)* 1.1 (Jan. 1983), pp. 3–24. ISSN: 1558-2868. DOI: `10.1145/357423.357425`.

[152] Jef Raskin. *The Humane Interface: New Directions for Designing Interactive Systems*. Addison-Wesley, Mar. 2000. ISBN: 978-0-201-37937-2.

[153] Eric S. Raymond. *The Art of UNIX Programming*. Addison-Wesley, Jan. 2004. ISBN: 978-0-13-142901-7.

[154] Patrick Rein, Robert Hirschfeld, and Marcel Taeumel. "Gramada: Immediacy in Programming Language Development". In: *Proceedings of the 2016 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Amsterdam, Netherlands: ACM, Oct. 2016, pp. 165–179. DOI: `10.1145/2986012.2986022`.

[155] Patrick Rein, Stefan Lehmann, Toni Mattis, and Robert Hirschfeld. "How Live are Live Programming Systems?: Benchmarking the Response Times of Live Programming Environments". In: *Proceedings of the Programming Experience 2016 (PX/16) Workshop*. Rome, Italy: ACM, July 2016, pp. 1–8. DOI: `10.1145/2984380.2984381`.

[156] Steven P. Reiss. "The Paradox of Software Visualization". In: *3rd IEEE International Workshop on Visualizing Software for Understanding and Analysis*. Budapest, Hungary: IEEE, Sept. 2005, pp. 59–63. DOI: `10.1109/VISSOF.2005.1684306`.

[157] Steven P. Reiss. "Visual Representations of Executing Programs". In: *Journal of Visual Languages and Computing* 18.2 (Apr. 2007), pp. 126–148. ISSN: 1045-926X. DOI: `10.1016/j.jvlc.2007.01.003`.

[158] Tamar Richner and Stéphane Ducasse. "Recovering High-level Views of Object-oriented Applications from Static and Dynamic Information". In: *Proceedings IEEE International Conference on Software Maintenance*. Oxford, England, UK: IEEE, Aug. 1999, pp. 13–22. DOI: `10.1109/ICSM.1999.792487`.

[159] Dirk Riehle and Heinz Züllighoven. "A Pattern Language for Tool Construction and Integration based on the Tools and Materials Metaphor". In: *Pattern Languages of Program Design*. Ed. by James O. Coplien and Douglas Schmidt. Addison-Wesley, May 1995, pp. 9–42. ISBN: 978-0-201-60734-5.

[160] Martin P. Robillard and Gail C. Murphy. "Representing Concerns in Source Code". In: *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16.1 (Feb. 2007), 3:1–3:38. ISSN: 1557-7392. DOI: `10.1145/1189748.1189751`.

[161] Tobias Roehm, Rebecca Tiarks, Rainer Koschke, and Walid Maalej. "How Do Professional Developers Comprehend Software". In: *2012 34th International Conference on Software Engineering (ICSE)*. Zurich, Switzerland: IEEE, June 2012, pp. 255–265. DOI: 10.1109/ICSE.2012.6227188.

[162] Florence Rossant. "A Global Method for Music Symbol Recognition in Typeset Music Sheets". In: *Pattern Recognition Letters* 23.10 (2002), pp. 1129–1141. ISSN: 0167-8655. DOI: 10.1016/S0167-8655(02)00036-3.

[163] David Röthlisberger, Oscar Nierstrasz, and Stéphane Ducasse. "Autumn Leaves: Curing the Window Plague in IDEs". In: *2009 16th Working Conference on Reverse Engineering*. Lille, France: IEEE, Oct. 2009, pp. 237–246. DOI: 10.1109/WCRE.2009.18.

[164] David Röthlisberger, Oscar Nierstrasz, and Stéphane Ducasse. "SmartGroups: Focusing on Task-Relevant Source Artifacts in IDEs". In: *2011 IEEE 19th International Conference on Program Comprehension*. Kingston, ON, Canada: IEEE, June 2011, pp. 61–70. DOI: 10.1109/ICPC.2011.20.

[165] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals - Part One*. 6th ed. Microsoft Press, 2012. ISBN: 978-0-7356-4873-9.

[166] Lawrence Sambrooks and Brett Wilkinson. "Comparison of Gestural, Touch, and Mouse Interaction with Fitts' Law". In: *Proceedings of the 25th Australian Computer-Human Interaction Conference: Augmentation, Application, Innovation, Collaboration*. Adelaide, Australia: ACM, Nov. 2013, pp. 119–122. DOI: 10.1145/2541016.2541066.

[167] David W. Sandberg. "Smalltalk and Exploratory Programming". In: *ACM SIGPLAN Notices* 23.10 (Oct. 1988), pp. 85–92. ISSN: 1558-1160. DOI: 10.1145/51607.51614.

[168] Robin Schreiber, Robert Krahn, Daniel H. H. Ingalls, and Robert Hirschfeld. *Transmorphic: Mapping Direct Manipulation to Source Code Transformations*. Tech. rep. 110. ISBN: 978-3-86956-387-9. Potsdam, Germany: Hasso Plattner Institute, 2016.

[169] Ken Schwaber. *Agile Project Management with Scrum*. Microsoft Press, 2004. ISBN: 978-0-7356-1993-7.

[170] Niko Schwarz. "DoodleDebug, Objects Should Sketch Themselves for Code Understanding". In: *5th Workshop on Dynamic Languages and Applications (DYLA)*. http://scg.unibe.ch/archive/papers/Schw11bDoodleDebug.pdf, accessed 2018-11-08. Zurich, Switzerland: University of Berne, July 2011.

[171] Beau Sheil. "Datamation®: Power Tools for Programmers". In: *Readings in Artificial Intelligence and Software Engineering*. Ed. by Charles Rich and Richard C. Waters. Morgan Kaufmann, Inc., July 1998. Chap. 33, pp. 573–580. ISBN: 978-0-934613-12-5. DOI: 10.1016/B978-0-934613-12-5.50048-3.

[172] Ben Shneiderman. "Direct Manipulation for Comprehensible, Predictable and Controllable User Interfaces". In: *Proceedings of the 2nd International Conference on Intelligent User Interfaces*. Orlando, FL, USA: ACM, Jan. 1997, pp. 33–39. DOI: 10.1145/238218.238281.

[173] Ben Shneiderman. "Direct Manipulation: A Step Beyond Programming Languages". In: *Computer* 16.8 (Aug. 1983), pp. 57–69. ISSN: 0018-9162. DOI: 10.1109/MC.1983.1654471.

[174] Ben Shneiderman. "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations". In: *Proceedings 1996 IEEE Symposium on Visual Languages*. Boulder, CO, USA: IEEE, Sept. 1996, pp. 336–343. DOI: 10.1109/VL.1996.545307.

[175] Ben Shneiderman. "Tree Visualization with Tree-Maps: 2-D Space-filling Approach". In: *ACM Transactions on Graphics (TOG)* 11.1 (Jan. 1992), pp. 92–99. ISSN: 1557-7368. DOI: 10.1145/102377.115768.

[176] Ben Shneiderman and Catherine Plaisant. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. 5th ed. Addison-Wesley, 2010. ISBN: 978-0-321-60148-3.

[177] Jonathan Sillito, Gail C. Murphy, and Kris De Volder. "Asking and Answering Questions During a Programming Change Task". In: *IEEE Transactions on Software Engineering* 34.4 (Apr. 2008), pp. 434–451. ISSN: 0098-5589. DOI: 10.1109/TSE.2008.26.

[178] Janice Singer, Timothy Lethbridge, Norman Vinson, and Nicolas Anquetil. "An Examination of Software Engineering Work Practices". In: *Proceedings of the 1997 Conference of the Centre for Advanced Studies on Collaborative Research*. Toronto, ON, Canada: IBM Press, Nov. 1997, pp. 174–188. DOI: 10.1145/1925805.1925815.

[179] David C. Smith, Charles Irby, Ralph Kimball, Bill Verplank, and Erik Harslem. "Designing the Star User Interface". In: *BYTE - The Small Systems Journal* 7.4 (Apr. 1982), pp. 242–282. ISSN: 0360-5280.

[180] Edward K. Smith, Christian Bird, and Thomas Zimmermann. "Build it Yourself! Homegrown Tools in a Large Software Company". In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 369–379. DOI: 10.1109/ICSE.2015.56.

[181] Randall B. Smith. "Experiences with the Alternate Reality Kit: An Example of the Tension between Literalism and Magic". In: *IEEE Computer Graphics and Applications* 7.9 (Sept. 1987), pp. 42–50. ISSN: 0272-1716. DOI: 10.1109/MCG.1987.277078.

[182] Randall B. Smith, Mario Wolczko, and David Ungar. "From Kansas to Oz: Collaborative Debugging When a Shared World Breaks". In: *Communications of the ACM* 40.4 (Apr. 1997), pp. 72–78. ISSN: 1557-7317. DOI: 10.1145/248448.248461.

[183] Diomidis Spinellis. "UNIX Tools as Visual Programming Components in a GUI-builder Environment". In: *Wiley Software: Practice and Experience* 32.1 (Jan. 2002), pp. 57–71. ISSN: 1097-024X. DOI: 10.1002/spe.428.

[184] Matthias Springer, Fabio Niephaus, Robert Hirschfeld, and Hidehiko Masuhara. "Matriona: Class Nesting with Parameterization in Squeak/Smalltalk". In: *Proceedings of the 15th International Conference on Modularity*. Málaga, Spain: ACM, Mar. 2016, pp. 118–129. DOI: 10.1145/2889443.2889457.

[185] Richard M. Stallman. "EMACS: The Extensible, Customizable, Self-documenting Display Editor". In: *Proceedings of the ACM SIGPLAN SIGOA Symposium on Text Manipulation*. Portland, OR, USA: ACM, June 1981, pp. 147–156. DOI: 10.1145/872730.806466.

[186] Bastian Steinert, Damien Cassou, and Robert Hirschfeld. "CoExist: Overcoming Aversion to Change". In: *Proceedings of the 8th Symposium on Dynamic Languages*. Tuscon, AZ, USA: ACM, Oct. 2012, pp. 107–118. DOI: 10.1145/2480360.2384591.

[187] Margaret-Anne Storey. "Theories, Methods and Tools in Program Comprehension: Past, Present and Future". In: *13th International Workshop on Program Comprehension (IWPC'05)*. St. Louis, MO, USA: IEEE, May 2005, pp. 181–191. DOI: 10.1109/WPC.2005.38.

[188] Margaret-Anne Storey, Li-Te Cheng, R. Ian Bull, and Peter Rigby. "Shared Waypoints and Social Tagging to Support Collaboration in Software Development". In: *Proceedings of the 2006 20th Anniversary Conference on Computer Supported Cooperative Work*. Banff, AB, Canada: ACM, Nov. 2006, pp. 195–198. DOI: 10.1145/1180875.1180906.

[189] Tony Stubblebine. "Pearl 5.8". In: *Regular Expression - Pocket Reference*. 2nd ed. O'Reilly, July 2007. Chap. 3, pp. 16–25. ISBN: 978-0-596-51427-3.

[190] Tarja Systä, Kai Koskimies, and Hausi A. Müller. "Shimba—An Environment for Reverse Engineering Java Software Systems". In: *Software: Practice and Experience* 31.4 (Feb. 2001), pp. 371–394. ISSN: 1097-024X. DOI: 10.1002/spe.386.

[191] Marcel Taeumel, Tim Felgentreff, and Robert Hirschfeld. "Applying Data-driven Tool Development to Context-oriented Languages". In: *Proceedings of the 6th International Workshop on Context-Oriented Programming*. Uppsala, Sweden: ACM, Aug. 2014, 1:1–1:7. DOI: 10.1145/2637066.2637067.

[192] Marcel Taeumel and Robert Hirschfeld. "Evolving User Interfaces From Within Self-supporting Programming Environments: Exploring the Project Concept of Squeak/Smalltalk to Bootstrap UIs". In: *Proceedings of the Programming Experience 2016 (PX/16) Workshop*. Rome, Italy: ACM, June 2016, pp. 43–59. DOI: 10.1145/2984380.2984386.

[193] Marcel Taeumel and Robert Hirschfeld. "Making Examples Tangible: Tool Building for Program Comprehension". In: *Design Thinking Research: Taking Breakthrough Innovation Home*. Ed. by Hasso Plattner, Christoph Meinel, and Larry Leifer. Springer, Aug. 2016, pp. 161–182. DOI: 10.1007/978-3-319-40382-3_11.

[194] Marcel Taeumel, Michael Perscheid, Bastian Steinert, Jens Lincke, and Robert Hirschfeld. "Interleaving of Modification and Use in Data-driven Tool Development". In: *Proceedings of the 2014 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Portland, OR, USA: ACM, Oct. 2014, pp. 185–200. DOI: 10.1145/2661136.2661150.

[195] Marcel Taeumel, Stephanie Platz, Bastian Steinert, Robert Hirschfeld, and Hidehiko Masuhara. "Unravel Programming Sessions with THRESHER: Identifying Coherent and Complete Sets of Fine-granular Source Code Changes". In: *Information and Media Technologies* 12.1 (Mar. 2017), pp. 24–39. ISSN: 1881-0896. DOI: 10.11185/imt.12.24.

[196] Marcel Taeumel, Bastian Steinert, and Robert Hirschfeld. "The VIVIDE Programming Environment: Connecting Run-time Information With Programmers' System Knowledge". In: *Proceedings of the ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Tucson, AZ, USA: ACM, Oct. 2012, pp. 117–126. DOI: 10.1145/2384592.2384604.

[197] Steven L. Tanimoto. "A Perspective on the Evolution of Live Programming". In: *2013 1st International Workshop on Live Programming (LIVE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 31–34. DOI: 10.1109/LIVE.2013.6617346.

[198] Peri Tarr, Harold Ossher, William Harrison, and Stanley M. Sutton Jr. "N Degrees of Separation: Multi-dimensional Separation of Concerns". In: *Proceedings of the 21st International Conference on Software Engineering*. Los Angeles, CA, USA: ACM, May 1999, pp. 107–119. DOI: 10.1145/302405.302457.

[199] Larry Tesler. "Object-oriented User Interfaces and Object-oriented Languages (Keynote Address)". In: *Proceedings of the 1983 ACM SIGSMALL Symposium on Personal and Small Computers*. San Diego, CA, USA: ACM, Dec. 1983, pp. 3–5. DOI: 10.1145/800219.806644.

[200] Ian Thomas and Brian A. Nejmeh. "Definitions of Tool Integration for Environments". In: *IEEE Software* 9.2 (Mar. 1992), pp. 29–35. ISSN: 0740-7459. DOI: 10.1109/52.120599.

[201] Astrid Thomschke, Daniel Stolpe, Marcel Taeumel, and Robert Hirschfeld. "Towards Gaze Control in Programming Environments". In: *Proceedings of the Programming Experience 2016 (PX/16) Workshop*. Rome, Italy: ACM, July 2016, pp. 27–32. DOI: `10.1145/2984380.2984384`.

[202] Scott R. Tilley and Dennis B. Smith. *Coming Attractions in Program Understanding.* Tech. rep. Carnegie Mellon University, Software Engineering Institute, Dec. 1996.

[203] Nikolai Tillmann, Michal Moskal, Jonathan de Halleux, and Manuel Fahndrich. "TouchDevelop: Programming Cloud-connected Mobile Devices via Touchscreen". In: *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*. Portland, OR, USA: ACM, Oct. 2011, pp. 49–60. DOI: `10.1145/2048237.2048245`.

[204] Jason Trenouth. "A Survey of Exploratory Software Development". In: *The Computer Journal* 34.2 (Jan. 1991), pp. 153–163. ISSN: 1460-2067. DOI: `10.1093/comjnl/34.2.153`.

[205] Christoph Treude and Margaret-Anne Storey. "Work Item Tagging: Communicating Concerns in Collaborative Software Development". In: *IEEE Transactions on Software Engineering* 38.1 (Jan. 2012), pp. 19–34. ISSN: 0098-5589. DOI: `10.1109/TSE.2010.91`.

[206] David Ungar, Henry Lieberman, and Christopher Fry. "Debugging and the Experience of Immediacy". In: *Communications of the ACM* 40.4 (Apr. 1997), pp. 38–43. ISSN: 0167-8655. DOI: `10.1145/248448.248457`.

[207] David Ungar and Randall B. Smith. "Self". In: *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*. San Diego, CA, USA: ACM, June 2007, 9:1–9:50. DOI: `10.1145/1238844.1238853`.

[208] *Unified Modeling Language (UML) 2.5.1*. `https://www.omg.org/spec/UML/2.5.1/`, accessed 2018-11-10. Object Management Group, Dec. 2017.

[209] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Langugage*. Vol. 9. APIC Studies in Data Processing. Academic Press, Inc., 1985. ISBN: 978-0-12-729651-7.

[210] Anthony I. Wasserman. "Tool Integration in Software Engineering Environments". In: *Software Engineering Environments*. Ed. by Fred Long. Chinon, France: Springer, Sept. 1989, pp. 137–149. DOI: `10.1007/3-540-53452-0_38`.

[211] Peter Wegner. "Dimensions of Object-based Language Design". In: *Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*. Orlando, FL, USA: ACM, Oct. 1987, pp. 168–182. DOI: `10.1145/38807.38823`.

[212] Stefan Wendler, Danny Ammon, Teodora Kikova, and Ilka Philippow. "Development of Graphical User Interfaces based on User Interface Patterns". In: *PATTERNS 2012, The Fourth International Conferences on Pervasive Patterns and Applications*. Nice, France: IARIA, July 2012, pp. 57–66. ISBN: 978-1-61208-221-9.

[213] Roel Wuyts and Stéphane Ducasse. "Unanticipated Integration of Development Tools using the Classification Model". In: *Computer Languages, Systems and Structures* 30.1 (Apr. 2004), pp. 63–77. ISSN: 1477-8424. DOI: `10.1016/j.cl.2003.08.003`.

[214] Takashi Yamamiya, Alessandro Warth, and Ted Kaehler. "Active Essays on the Web". In: *2009 Seventh International Conference on Creating, Connecting and Collaborating through Computing (C5 2009)*. Kyoto, Japan: IEEE, Jan. 2009, pp. 3–10. DOI: `10.1109/C5.2009.10`.