Hasso–Plattner–Institut für Softwaresystemtechnik
an der Universität Potsdam

Konzeptionelle Patterns und ihre Darstellung

Dissertation

zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.–Ing.)
in dem Fachgebiet Software–Engineering

eingereicht an der
Mathematisch–Naturwissenschaftlichen Fakultät
der Universität Potsdam

von
Dipl.–Ing. Bernhard Gröne

Potsdam, 3. August 2004

Vorwort

Bei der Betrachtung Software–intensiver Systeme fällt auf, dass bestimmte Ideen und Strukturen immer wieder auftauchen — so wie sich auch die Probleme ähneln, die zu lösen sind. Das Wissen um bewährte Lösungen ist Bestandteil der Erfahrungen von Entwicklern und Architekten und bringt diesen in ihrer Arbeit Vorteile, da sie durch die Wiederverwendung bewährter Lösungen schneller und mit geringerem Risiko die Probleme in ihrem aktuellen Projekt lösen können.

Die Arbeitsgruppe „Modellierung Software–intensiver Systeme“ am Hasso–Plattner–Institut (HPI) befasst sich mit der Darstellung und Wissensvermittlung komplexer technischer Sachverhalte und benutzt dafür die Fundamental Modeling Concepts (FMC). Als ich nach fast fünf Jahren Tätigkeit bei der SAP im Jahr 2000 der Arbeitsgruppe am HPI beitrat, hatte ich mir das Ziel gesetzt, einen Weg zu finden, um Erfahrungswissen von Entwicklern und Architekten festzuhalten und anderen zugänglich zu machen.

Patterns hatte ich über das Buch „Design Patterns“ von Gamma et al. kennengelernt, das eher auf Probleme mit der objektorientierten Programmierung zugeschnitten ist. Mir kam es aber auf die Probleme der Software–Architekten an, die in der Regel auf höherer Abstraktionsebene zu finden sind. Über die Architektur–Patterns lernte ich schließlich die Pattern–Community kennen und stellte fest, dass sie ein ähnliches Ziel verfolgt. Das führte schließlich zur Idee, Patterns mit der Darstellung komplexer Sachverhalte mittels FMC zu kombinieren. Die daraus resultierenden konzeptionellen Patterns stellen ein mächtiges Mittel zur Weitergabe von Erfahrungswissen von Software–Architekten dar.

An dieser Stelle möchte ich mich bei Professor Dr.–Ing. Siegfried Wendt bedanken, der meine Sicht auf informationsverarbeitende Systeme entscheidend geprägt hat und verantwortlich ist für die Fundamtental Modeling Concepts, die mir schon bei SAP eine große Hilfe bei der Arbeit waren. Meinen Kollegen danke ich sehr für die vielen interessanten Diskussionen und ihre Offenheit für neue Ideen, insbesondere Andreas Knöpfel, mit dem ich längere Zeit das Büro geteilt habe und der der wichtigste Diskussionspartner war. Offenheit gegenüber Neuem trifft auch auf die Teilnehmer der Pattern–Workshops der VikingPLoP zu. Hier möchte ich besonders Neil Harrison erwähnen, von dem ich viel über die Community gelernt habe.

Den Herren PD. Dr–Ing. habil. Martin Engelien und Prof. Dr.–Ing. habil. Otto Mayer danke ich sehr für die Übernahme des Koreferats.
Meiner Frau Ruth danke ich sehr für ihr Vertrauen und ihre Unterstützung.

Bernhard Gröne

Abstract

Planning large and complex software systems is an important task of a system architect. It includes communicating with the customer, planning the overall system structure as well as preparing the division of labor among software engineers. What's more, a system architect benefits from other professionals' experiences concerning system architecture.

Patterns provide a common form for the transfer of experiences. A pattern describes a widely used and proven solution to a problem that occurs in a certain context. Concerning software, categories of patterns are Idioms for programming–language–dependent solutions, Design Patterns for small parts of software and Architectural Patterns which have influence on larger parts or even on the whole progam. Pattern Systems group patterns for a specific problem domain. If they are supported by a guideline telling which sequence of patterns one should choose for which criteria, these systems are called Pattern Languages.

Most architectural patterns propose how to structure the code. From the system architect's point of view, this task usually comes *after* the definition of the functional components of the system. Furthermore, architectural patterns describe two different things: A basic structure for an application as well as the required infrastructure.

The graphical representations of architectural patterns usually focus on the software structures resulting from the solution in terms of classes and their relationships. This can be a problem if the solution doesn't imply one specific software structure but rather describes a concept which may be even independent from an implementation via software at all.

By separating system from software structures, one can now present the concepts of a pattern regarding its system structure without having to propose a software structure as the solution. For that reason, Conceptual Patterns are introduced. A pattern can be called conceptual if both problem and solution concern system structures. Here, the functional aspects and structures of the system are relevant while code structures or even the use of software for implementation are not. To support the focus on system structures, terminology and notation of conceptual patterns should use an adequate means such as provided by the Fundamental Modeling Concepts (FMC).

Many existing patterns provide concepts that fit into the conceptual pattern category. Transforming such a pattern into a conceptual pattern makes it necessary to adapt the pattern description to focus on the system structures. As an example for conceptual patterns, a pattern language for multi–tasking servers is presented which is one result of a deep analysis of different server systems.

Conceptual patterns are useful for system architects for finding and evaluating system architectures as well as for analysing existing systems, as they are a means to capture and transport experience concerning their system structure. Pattern languages of related conceptual patterns can transport experience of a specific problem domain and offer alternative solutions and criteria for their selection.

Zusammenfassung

Zur Beherrschung großer Systeme, insbesondere zur Weitergabe und Nutzung von Erfahrungswissen in der frühen Entwurfs– und Planungsphase, benötigt man Abstraktionen für deren Strukturen. Trennt man Software– von Systemstrukturen, kann man mit letzteren Systeme auf ausreichend hohem Abstraktionsgrad beschreiben. Diese Arbeit stellt die Kategorie der konzeptionellen Patterns vor, mit denen Erfahrungswissen bezüglich des Entwurfs und der Bewertung von Systemstrukturen weitergegeben werden können.

Software–Patterns dienen dazu, Erfahrungswissen bezüglich programmierter Systeme strukturiert weiterzugeben. Dabei wird unterschieden zwischen Idiomen, die sich auf Lösungen mit einer bestimmten Programmiersprache beziehen, Design–Patterns, die nur einen kleinen Teil des Programms betreffen und Architektur–Patterns, deren Einfluss über einen größeren Teil oder gar das komplette Programm reicht.

Mit Architektur–Patterns werden zwei Dinge beschrieben, nämlich eine Grundstruktur für die Anwendung sowie die dafür meist erforderliche Infrastruktur. Diese Zweiteilung kann problematisch sein; beispielsweise ist nicht immer klar, wofür der Pattern–Name eigentlich steht, zumal in der Literatur mit dem Namen des Patterns oft nur einer der beiden Teile in Verbindung gebracht wird. Es ist daher sinnvoller, diese Aspekte bei Architekturpatterns zu trennen und die resultierenden Patterns in einer Pattern Language, also einem System von Patterns, für bestimmte Einsatzgebiete zusammenzustellen.

Zudem erfüllt die grafische Darstellung von Architektur–Patterns in der Literatur häufig nicht den Anspruch, die Idee des Patterns schnell erfassbar und merkbar zu machen. Ein Grund dafür ist, dass häufig Darstellungsmittel aus dem objektorientierten Entwurf verwendet werden, die primär zur Darstellung von Software–Strukturen gedacht sind und sich nur bedingt für Systemstrukturen eignen.

Daher wird die Kategorie der konzeptionellen Patterns mit einer darauf abgestimmten grafischen Darstellungsform vorgeschlagen, bei denen Problem und Lösungsvorschlag im Bereich der Systemstrukturen liegen. Sie betreffen informationelle Systeme, sind aber nicht auf Lösungen mit Software beschränkt. Die Systemstrukturen werden grafisch dargestellt, wobei dafür die Fundamental Modeling Concepts (FMC) verwendet werden, die zur Darstellung von Systemstrukturen entwickelt wurden.

Als Beispiel wird ein System konzeptionelle Patterns für Multitasking Server vorgestellt, das anhand mehrerer Untersuchungen bestehender Systeme entstanden ist. Daneben wird eine Auswahl wichtiger Architektur– und Design–Patterns im Hinblick auf die dort beschriebenen Systemstrukturen dargestellt. Diese Beispiele zeigen, wie konzeptionelle Patterns aussehen und wie sie zur Weitergabe von Erfahrungswissen bezüglich des Entwurfs und der Bewertung von Systemstrukturen benutzt werden können.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Problembeschreibung

Ein häufig zu beobachtendes Problem in der technischen Entwicklung besteht darin, dass manche Lösungen zu Problemen immer wieder neu erfunden werden. Das liegt zum Teil auch daran, dass nur selten Erfahrungen aus früheren Entwicklungen an andere Menschen weitergegeben werden. Patterns[1] stellen eine Möglichkeit dar, derartige Erfahrungen in strukturierter Form festzuhalten und für andere verfügbar zu machen.

Ein Pattern ist die Beschreibung einer bewährten Lösung zu einem Problem, das in einem bestimmten Kontext auftritt. Ursprünglich in der Bau–Architektur verwendet, wurde die Idee der Patterns später auf Software übertragen. Während die ersten Design–Patterns die Strukturierung objektorientierter Programme behandelten, kamen bald allgemeinere Patterns dazu, die neben Software– auch Systemstrukturen betreffen.

Die Komplexität großer Systeme kann durch Abstraktionen ihrer Systemstrukturen beherrscht werden. Das gilt besonders für die frühe Planung der Struktur derartiger Systeme, die auch als Architektur bezeichnet wird. Als Ergebnis einer Untersuchung mehrerer Industrieprojekte stellt Keller in [Kel03] fest, dass „die angemessene Beschreibung der architekturellen Strukturen eines zu entwickelnden Systems von großer Bedeutung für die Planung, Durchführung und Kontrolle von Softwareprojekten ist.“

Aus diesem Grund sind Patterns, die Lösungen bezüglich der Systemstrukturen behandeln, ein Mittel zur Weitergabe von Erfahrungen für die Architektur–Phase und damit zur Beherrschung der Komplexität. Die so genannten Architektur–Patterns betreffen überwiegend Systemstrukturen. Sie beschreiben einerseits ein Konzept, nach dem ein System zu strukturieren ist, andererseits die dazu erforderliche Infrastruktur. Diese Zweiteilung der Lösung führt aber zu verschiedenen Problemen, beispielsweise zu der Frage, ob der Name eines Architektur–Patterns für beide oder nur für einen Teil steht. Aber nicht nur Architektur–Patterns, auch ein Teil der Design–Patterns beschreibt Lösungen im Bereich der Systemstrukturen, wodurch diese ebenfalls bereits in der Architektur–Phase einsetzbar sind.

Betrachtet man die Darstellung von Patterns, insbesondere die Grafiken, stellt man fest, dass hier Darstellungsmittel zur Beschreibung von Software–Strukturen eingesetzt werden. Diese sind aber in den meisten Fällen nicht angemessen, den konzeptionellen Teil des Patterns leicht verständlich wiederzugeben.

[1] In dieser Arbeit wird der englische Begriff „Pattern“ im Sinne der Beschreibung einer bewährten Lösung verwendet, während der deutsche Begriff „Muster“ seine ursprüngliche, weiter gefasste Bedeutung behält.

1.2 Lösungsansatz konzeptionelle Patterns

Als Lösung wird die Kategorie der konzeptionellen Patterns eingeführt. Es handelt sich dabei um Patterns, die Lösungen im Bereich der Systemstrukturen beschreiben. Zur grafischen Darstellung von Systemstrukturen haben sich FMC–Aufbaubilder bewährt. Daher bietet sich ihr Einsatz bei allen konzeptionellen Patterns an. Konzeptionelle Patterns eignen sich zur Verwendung in der Architekturphase.

In diese Kategorie lassen sich die meisten Architektur–Patterns und einige Design–Patterns einordnen. Bei den Architektur–Patterns sollte man neben einer Anpassung der Darstellung auf Systemstrukturen auch eine Auftrennung in mehrere Patterns innerhalb einer Pattern Language in Betracht ziehen, da Architektur–Patterns häufig überfrachtet sind. Dadurch ist es einfacher, verwandte Architektur–Patterns zu identifizieren, die beispielsweise die gleiche Infrastruktur verwenden.

1.3 Gliederung dieser Arbeit

Die Begriffe System– und Software–Struktur sowie Architektur sind grundlegend für diese Arbeit und werden in Kapitel 2 vorgestellt. Bei der Betrachtung von Systemstrukturen spielt auch der Zusammenhang mit gängigen Sichtenkonzepten eine Rolle. Zur Modellierung von Systemstrukturen stellen die Fundamental Modeling Concepts (FMC) ein bewährtes Mittel dar, das daher auch in dieser Arbeit verwendet wird.

Patterns sind der Gegenstand von Kapitel 3. Hier wird deren Geschichte betrachtet, die Formen und Bestandteile eines Patterns werden erklärt und die Kategorien vorgestellt, in die Patterns üblicherweise eingeordnet werden.

In Kapitel 4 wird eine repräsentative Auswahl von Architektur– und Design–Patterns für Softwaresysteme aus der Literatur in kompakter Form vorgestellt, um an ihr Fragen zur thematischen Kategorisierung, dem Abstraktionsgrad und vor allem der Kategorisierung in Architektur– und Design–Patterns zu klären.

Im folgenden Kapitel 5 wird an Beispielen gezeigt, mit welchen grafischen Darstellungsmitteln Architektur–Patterns in der Literatur beschrieben werden.

Die Konsequenzen aus den Betrachtungen führen zu den *konzeptionellen Patterns*, die in Kapitel 6 eingeführt werden.

Als Anwendungsfall für konzeptionelle Patterns dient Kapitel 7, in dem ein Pattern–System für Netzwerk–Server und Anwendungsbeispiele dafür vorgestellt werden.

Den Abschluss der Arbeit bildet Kapitel 8 mit einem Fazit und Ausblick auf weitere Forschung.

Kapitel 2

Architektur und Systemstrukturen

Zur Beherrschung größerer Systeme müssen Abstraktionen geleistet werden, die die relevanten Strukturen dieser Systeme überschaubar machen. Weiterhin müssen eine Reihe von Entwurfsentscheidungen frühzeitig von einer oder wenigen Personen gefällt werden, bevor eine Arbeitsteilung sinnvoll beginnen kann. Für beide Sachverhalte wird der Begriff „Architektur“ verwendet. Davon abgegrenzt wird der Begriff „Design“ (Entwurf), mit dem entweder Strukturen niedrigerer Abstraktionshöhe oder die Entwurfsentscheidungen der Entwickler nach erfolgter Arbeitsteilung beschrieben werden.

Realisiert man ein informationsverarbeitendes System als programmiertes System, muss man zwei Arten von Strukturen in der Planung und in der Kommunikation mit den Beteiligten unterscheiden, nämlich die Struktur des gewollten (Gesamt–) Systems und die Struktur der dafür zu entwickelnden Software.

Neben der Abgrenzung von Architektur und Design gibt es Sichten auf ein System, mit denen je nach betrachtetem Aspekt verschiedene Strukturen eines Systems unterschieden werden können. Im Rahmen dieser Arbeit wird das „Siemens 4“ Sichtenkonzept von Soni, Nord und Hofmeister näher betrachtet, weil die dort eingeführte konzeptionelle Sicht für die weitere Arbeit interessant ist.

2.1 System– und Software–Strukturen

Ein Architekt oder Entwickler eines informationellen Systems[1] ist normalerweise am funktionierenden Gesamt–System als Ergebnis seiner Arbeit interessiert. Dieses besteht aus aktiven technischen Komponenten, die Verhalten zeigen, Orten, an denen Informationen und Werteverläufe beobachtet werden können, und in der Regel auch Menschen, die mit den aktiven technischen Komponenten interagieren.

Das Gesamt–System besteht damit aus einem üblicherweise technisch realisierten informationellen Subsystem und seiner Umgebung, mit dem dieses interagiert[2]. In dieser Arbeit wird der Begriff System abkürzend für das technisch realisierte informationelle Subsystem verwendet.

[1] Im Kontext dieser Arbeit werden *informationelle Systeme*, also dynamische Systeme betrachtet, bei denen nur die Informationsverarbeitung relevant ist. Information ist aber nur durch Interpretation der beobachtbaren Werte erfassbar. Ein *dynamisches System* ist ein „konkretes oder zumindest konkret vorstellbares Gebilde, welches ein beobachtbares Verhalten zeigt, wobei dieses Verhalten als Ergebnis des Zusammenwirkens der Systemteile angesehen werden kann.“ [Wen91, S. 136]

[2] Knöpfel benutzt die Begriffe *Interaktionssystem* für das Gesamtsystem sowie *Interaktives System* für das Subsystem [Kno04].

Bei programmierten und Software–intensiven Systemen werden die technischen Komponenten durch einen Universal–Interpreter (z.B. einen Mikroprozessor) mit Schnittstellen zu anderen Komponenten und Software für diesen Universal–Interpreter realisiert.

Damit muss man folgende Strukturen[3] innerhalb eines programmierten Systems unterscheiden:

- Systemstrukturen
  - Struktur des gewollten Systems
  - Struktur des Trägersystems
  - Strukturen zur Laufzeit (im Arbeitsspeicher des Trägersystems während der Ausführung der Software)
- Software–Strukturen
  - Struktur der Quellen
  - Struktur des Codes

2.1.1 Struktur des gewollten Systems

Mit dem gewollten System wird implementierungsunabhängig beschrieben, welchem Zweck das System dienen soll und welche Komponenten es dafür enthalten muss. Oftmals sind bestimmte Strukturen bereits vorgegeben, beispielsweise Teilsysteme, die integriert werden müssen.

Die Strukturierung erfolgt nach funktionalen Gesichtspunkten, wobei es unerheblich ist, ob eine Komponente über Programmierung, Hardware oder als Mensch, der in einer bestimmten Rolle agiert, realisiert wird.

Beispiel: Ein Geldautomat muss eine Benutzerschnittstelle zum Kunden anbieten und eine Verbindung zur Bank für die Transaktion haben. Weiterhin werden Komponenten zum Lesen einer Magnetkarte und zur Ausgabe von Bargeld benötigt.

Rollensystem

Wird das gewollte System mit Hilfe eines programmierten Systems als Trägersystem realisiert, spricht man auch vom Rollensystem [Wen91, S. 315]. Hierbei hat man die Vorstellung, dass man ein programmierbares System durch Einbringen und Starten eines Programms dazu bringt, die Rolle des gewollten Systems zu spielen, so wie auch ein Schauspieler im Theater beim Spielen seiner Rolle eine andere Person oder einen Charakter verkörpert. Zum Rollensystem gehört daher immer das programmierbare Trägersystem sowie das Programm selbst.

Begriff „Anwendung“

In dieser Arbeit wird der Begriff der *Anwendung* synonym zum Rollensystem verwendet.

[3] In der Mathematik versteht man unter Struktur ein „Gebilde aus Mengen und Relationen, welches mindestens eine nicht-leere Menge und eine Relation umfasst“ [Wen91]. Beispiele sind algebraische Strukturen wie Gruppe, Ring, Körper, Verband.

Verfeinerungen und Ebenenwechsel

Betrachtet man Systemmodelle, kann man diese nach drei Kriterien einordnen: Dem betrachteten Ausschnitt des Systems, der Auflösung (Detailgrad) und der Betrachtungsebene.[4]

Eine Erhöhung der Auflösung bedeutet eine hierarchischen Verfeinerung, bei der also äußere Strukturen erhalten bleiben und neue Strukturen innerhalb bislang nicht strukturierter Elemente auftauchen.

Ein Wechsel der Betrachtungsebene entsteht dagegen durch eine andere Interpretation der beobachtbaren Sachverhalte. Das ist bei informationellen Systemen der Fall, wenn die Implementierung betrachtet wird. Dabei können sich die Strukturen zweier Betrachtungsebenen deutlich unterscheiden. Im Gegensatz zur Erhöhung der Auflösung, bei der in der Regel nur weitere Informationen über das System hinzukommen, verschwindet beim Wechsel der Betrachtungsebene auch Information, die wegen der geänderten Interpretation hier nicht mehr relevant ist.

Ein Wechsel der Betrachtungsebene, beispielsweise nach Anwendung von Entwurfsentscheidungen zur Implementierung, muss nicht zwangsläufig zur Betrachtung von Software–Strukturen führen; oft kann auch auf tieferer Ebene nach wie vor vom gewollten System gesprochen werden. Das folgende Beispiel verdeutlicht dies:

Beispiel Online–Store: Das gewollte System

Zur Verdeutlichung der Strukturen soll ein Online Store dienen, über den Kunden Waren bestellen können. Abbildung 2.1 zeigt die Struktur des gewollten Systems.

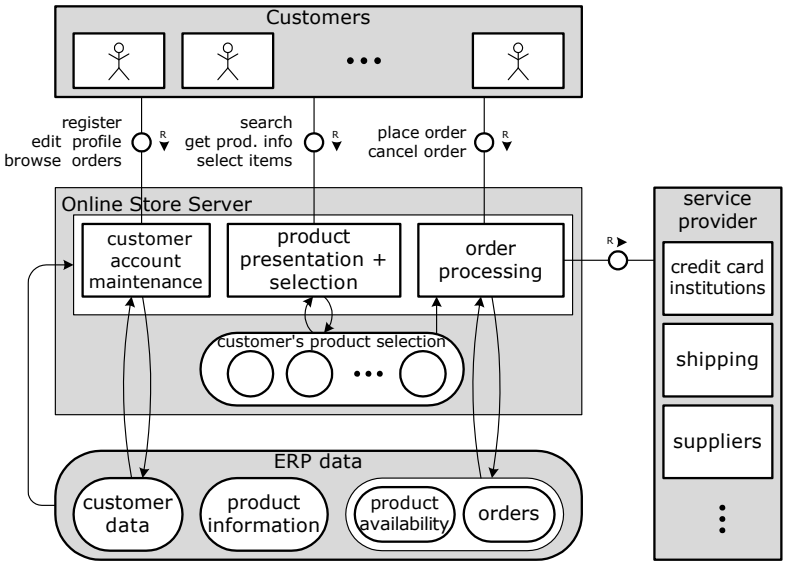


Abbildung 2.1: Online–Store: Struktur des gewollten Systems

[4]Bungert stellt in [Bun98, S. 35ff] den *Systemausschnitt* als horizontale Partitionierung vor. Bei der *Erhöhung der Auflösung* bleibt die äußere Struktur erhalten, es kommen mit den Details innere Strukturen hinzu. Die *Wahl der Betrachtungsebene* und die damit verbundene Unterteilung der zu beobachtenden Sachverhalte bezeichnet er als *vertikale Partitionierung*, die durch eine andere Interpretation entsteht. Primäre Sachverhalte betreffen die höheren Betrachtungsebene, beispielsweise das Bestellen eines Produkts, während sekundäre Sachverhalte die niedrigere Betrachtungsebene betreffen, beispielsweise die Benutzung eines Telefons. Dabei dienen die sekundären Sachverhalte zur Realisierung der primären.

Man erkennt oben eine offene Anzahl an Kunden, die über drei verschiedene Kanäle mit den Akteuren des Online Store Servers kommunizieren können, indem sie verschiedene Aufträge absetzen, nach Waren suchen oder Bestellungen stornieren. Der Online Store Server speichert den Warenkorb eines Kunden und hat Zugriff auf die betriebswirtschaftlichen Daten der Firma (ERP[5] data), beispielsweise Kundenadressen oder Produktdaten. Weiterhin nimmt der Online Store auch Dienste anderer Anbieter in Anspruch, von denen in Abbildung 2.1 nur drei beispielhaft gezeigt sind.

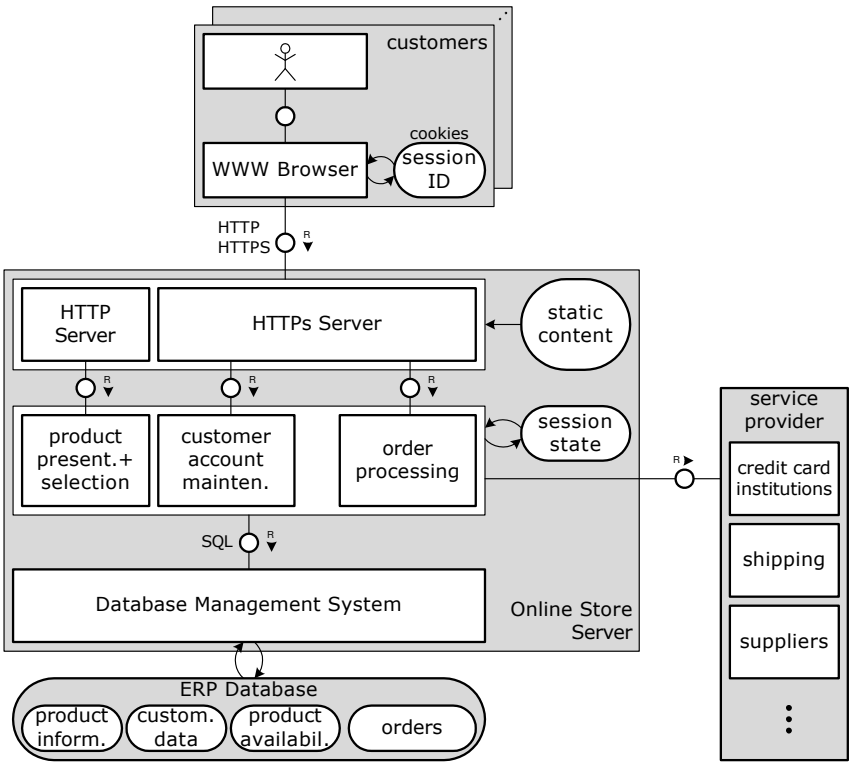


Abbildung 2.2: Online–Store: Verfeinerung des Online Store Servers

Verfeinerung Zur Realisierung dieses Online–Store–Systems muss man nun gewisse technische Entscheidungen treffen, die aber immer noch ausschließlich Systemstrukturen betreffen. In Abbildung 2.2 sieht man jetzt, dass die Kunden mit Hilfe eines WWW Browsers mit dem Online Store kommunizieren. Statt der verschiedenen Auftragssorten sieht man jetzt nur noch einen Kanal, auf dem das Protokoll HTTP bzw. dessen verschlüsselte Variante HTTPS „gesprochen“ wird. Es tauchen neue Akteure auf, nämlich die HTTP– und HTTPS–Server sowie ein Database Management System (DBMS), das über die Sprache SQL angesprochen wird.

Die Verfeinerung betrifft die Kanäle zwischen Benutzer und Server (Neu: WWW Browser und HTTP Server) sowie die Zugriffe auf die ERP–Daten über ein SQL–Datenbanksystem. Die Information auf diesen Kanälen wird auf einer niedrigeren Betrachtungsebene interpretiert, nämlich als Protokolle wie z.B. HTTP.

[5] Die Abkürzung ERP steht für Enterprise Resource Planning und wird gewöhnlich im Zusammenhang mit Systemen verwendet, die zur Verwaltung und Steuerung betriebswirtschaftlicher Prozesse von Unternehmen dienen (Warenwirtschaftssysteme).

2.1.2 Struktur des Trägersystems

Im Falle von programmierten Systemen benötigt man ein System, das in der Lage ist, Programme auszuführen und das über Schnittstellen verfügt, an denen das gewünschte Verhalten gezeigt wird. Im Beispiel des Geldautomaten wäre das Trägersystem also ein Rechner mit Prozessor, Speicher, Tasten und Bildschirm–Schnittstellen, Netzwerkanschluss sowie ein Kartenleser, Krypto– und ein Geldausgabemodul.

Das Rollensystem (siehe Abschnitt 2.1.1) entsteht also dadurch, dass man dem Trägersystem das Programm übergibt und es anweist, dieses Programm auszuführen. Ein Trägersystem muss daher immer über einen Speicherort für das Programm verfügen.

Manche Trägersysteme sind standardisiert oder so typisch, dass sie nicht mehr explizit betrachtet werden, z.B. das Trägersystem „PC mit Betriebssystem Windows“. In Bereichen wie den so genannten eingebetteten Systemen (Embedded Systems, beispielsweise eine Steuerung für einen Ottomotor oder eine Waschmaschine) gilt das nicht, denn hier gibt es eine Vielzahl von Prozessoren, Betriebssysteme (falls überhaupt vorhanden) und I/O–Komponenten, und damit eine Vielfalt unterschiedlicher Trägersysteme. In diesen Bereichen müssen die Fähigkeiten des Trägersystems daher immer berücksichtigt werden.

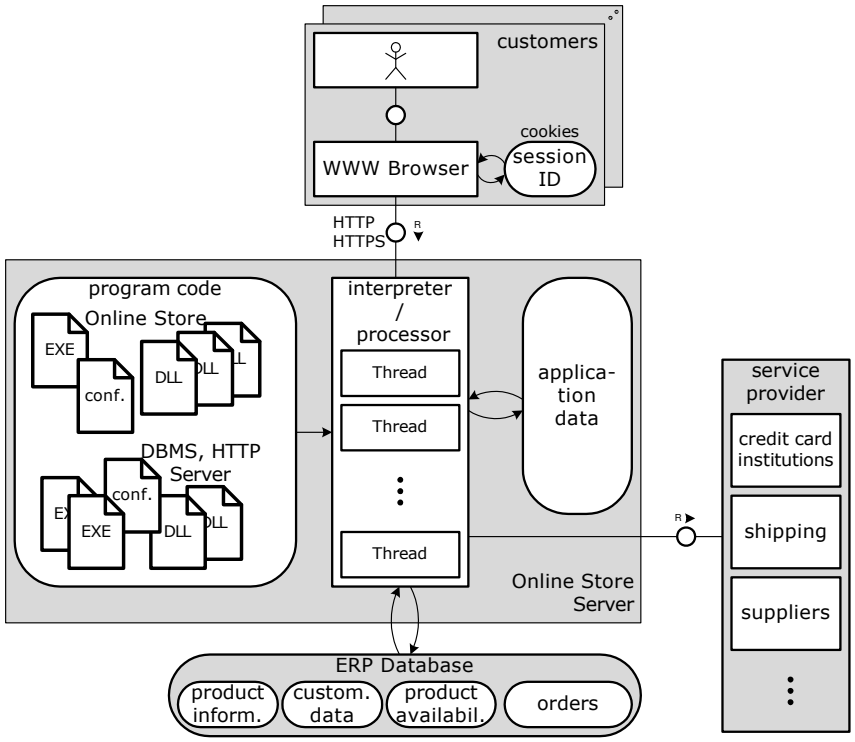


Abbildung 2.3: Online–Store: Realisierung des Online–Store Servers durch Ausführung von Code

Beispiel Online–Store: Das Trägersystem

Der Online Store Server wird nun als programmiertes System betrachtet. In Abbildung 2.3 sieht man nur noch einen Akteur im Online Store Server, nämlich den Abwickler, also den Interpreter des Programmcodes (üblicherweise der Prozessor). Die in Abbildung 2.2 noch vorhandenen Akteure wie etwa der HTTP Server werden jetzt durch Ausführung des entsprechenden Codes

vom Interpreter simuliert. Abbildung 2.3 zeigt also eine abstrakte Sicht des Trägersystems, das durch Ausführen des entsprechenden Codes zum gewünschten System wird.

2.1.3 Struktur der Software

Die Programmierung des Trägersystems erfolgt durch Software, die ins Trägersystem eingebracht wird (z.B. durch Einsetzen eines Speicherchips oder Kopieren von einer CD-ROM). Auch bei der Software kann man verschiedene Strukturen unterscheiden:

- Struktur der Quellen
  Für den Fall, dass es sich um eine compilierte Programmiersprache handelt, unterscheidet sich die Darstellung und Struktur der Quellen von der übersetzten Form, dem Code. Die Quellen werden in der Regel auch nicht zum Kunden ausgeliefert. Struktureinheiten von Quellen sind z.B. Module oder Prozeduren.
- Struktur des Codes
  Hier geht es um die Einheiten, in denen der vom Prozessor ausführbare Code strukturiert ist, z.B. in Form von Objektdateien und Bibliotheken. Deren Ort ist relevant, beispielsweise bei einer Ablage in einem Dateisystem. Bei seiner Ausführung ist der Code auch im Programmspeicher des Trägersystems sichtbar.

Ähnlichkeiten zwischen den Systemstrukturen und den Software–Strukturen sind allein schon für eine bessere Verständlichkeit der Software wünschenswert. Eine durchgängige Abbildung ist aber selten möglich; es gibt beispielsweise Strukturen in Software, die sich nicht auf Systemstrukturen zurückführen lassen, sondern durch die Arbeitsteilung der Entwickler, Vereinfachung von Installation oder Upgrades, oder auch Verbesserung der Performance begründet sind. Genauso wird ja nur ein Teil des Systems mit selbst entwickelter Software beschrieben; andere Teile sind entweder nicht programmiert oder werden durch Software anderer Hersteller realisiert.

Beispiel Online–Store: Die Software–Strukturen

Um über die Strukturen in der Software zu reden, muss man deren Entwicklung betrachten, wie in Abbildung 2.4 gezeigt. Hier geht es nicht mehr um Kunden und ERP–Daten, sondern um Entwickler mit ihren Werkzeugen, die Quelltexte und Software–Modelle schreiben und daraus Code erzeugen. Dabei ist zu beachten, dass es hier nur um die Entwicklung des Codes für den Online Store geht, denn der Code für HTTP Server, DBMS oder Betriebssystem wird als vorhanden vorausgesetzt und nicht weiter betrachtet.

Betrachtet man die Struktur der Quellen des Online Stores, dann wird man üblicherweise Software–Einheiten den Akteuren aus dem gewollten System (Abbildung 2.2) zuordnen können. Es gibt aber durchaus auch Einheiten, zu denen keine Abbildung auf die Systemstruktur existiert, genauso wie es Einheiten der Systemstruktur gibt, die nicht mit Software realisiert sind.

2.1.4 Strukturen zur Laufzeit

Bei der Ausführung des Programms im Trägersystem entstehen wiederum Strukturen innerhalb des Arbeitsspeichers, die sich auf verschiedenen Ebenen interpretieren lassen. Das reicht

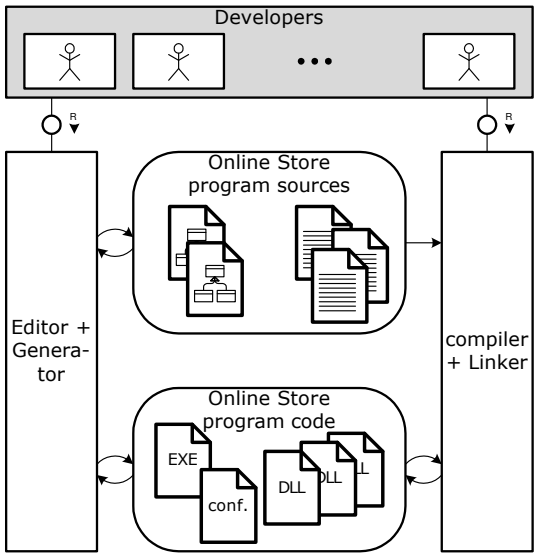


Abbildung 2.4: Online–Store: Entwicklung der Software für den Online Store

von einfachen Datenstrukturen über Objektstrukturen bis hin zu Threads und Prozessen, die miteinander in Beziehung stehen. Aus Sicht des Trägersystems ist alles Speicherbelegung. Fasst man z.B. die Software „Betriebssystem“ mit dem Trägersystem zusammen, erhält man ein neues, komplexeres Trägersystem, in dem die Strukturen im Arbeitsspeicher wieder anders zu interpretieren sind. Dieser Ebenenwechsel wurde detailliert beschrieben von Bungert in [Bun98].

In diesem Sinne handelt es sich bei Systemstrukturen in der Regel auch um Laufzeitstrukturen, da hier eine Abbildung beobachtbarer Speicherstrukturen eines programmierten Systems auf Systemstrukturen (auch auf hoher Betrachtungsebene) möglich ist. Insofern zeigen die Abbildungen 2.1, 2.2 und 2.3 jeweils Laufzeit–Strukturen auf verschiedenen Betrachtungsebenen

2.1.5 Trägersystem–Schichtung

2.1.5.1 Schichtung programmierter Systeme

Ist ein Trägersystem selbst wiederum durch ein programmiertes System realisiert, also Rollensystem für ein tiefer liegendes Trägersystem, spricht man von einer Trägersystem–Schichtung. Bei einem Wechsel der Betrachtungsebene wird also das Trägersystem für ein Rollensystem der höheren Ebene selbst wieder zum Rollensystem, das von einem Trägersystem einer niedrigeren Ebene realisiert wird[6]. In der Praxis taucht eine Trägersystem–Schichtung bereits dann auf, wenn ein Programm nicht direkt vom Hardware–Prozessor, sondern durch einen programmierten Interpreter ausgeführt wird.

Beispiel: Ein Workflow–System, das in einer Firma X eingesetzt wird, setzt Aufträge an verschiedene Service Provider ab und speichert die Zustände und Daten der einzelnen Workflow–Dokumente, wie in Abbildung 2.5 oben zu sehen. Das Verhalten wird durch ein Programm, nämlich eine Workflow–Definition beschrieben, die von einer programmierbaren Workflow–Maschine eingelesen und umgesetzt wird. Die Beschreibung der Workflows ist in Dokumenten

[6] Wendt verwendet hierfür den Begriff Rollenhuckepack [Wen91, S. 380].

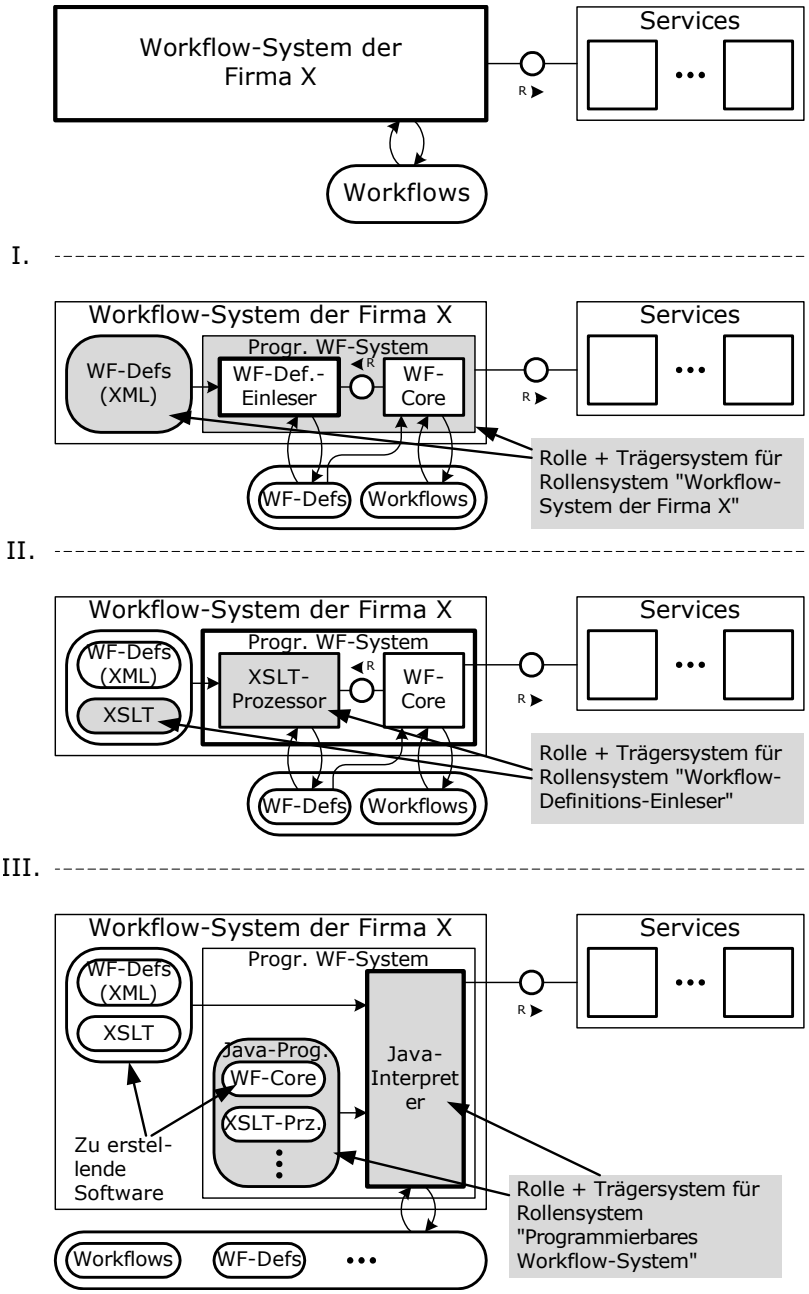


Abbildung 2.5: Trägersystemschichtung am Beispiel eines Workflow–Systems
(Trägersystem und dazugehöriges Programm sind jeweils grau gefärbt, das Rollensystem ist dick umrandet)

abgelegt, die gemäß eines XML–Formats strukturiert sind. Um diese einzulesen, benutzt das Workflow–System einen Standard–XSLT–Prozessor, der durch entsprechende Konfiguration, beschrieben in einem XSLT–Programm, die darin codierten Workflow–Definitionen in eine vom Workflow–Core auswertbare Form umsetzen kann (siehe Abbildung 2.5 Mitte). Workflow–System und der dazugehörige XSLT–Prozessor sind in Java geschrieben und benutzen weitere Bibliotheken. Der Java–Interpreter wiederum ist ein Programm, das auf einem Server ausgeführt wird (in Abbildung 2.5 nicht mehr dargestellt). Damit ergibt sich folgende Schichtung von Systemen:

	Rollensystem	Trägersystem	Software
I.	Workflow–System der Firma X	Programmierbares Workflow–System: Workflow–Def.–Einleser und Workflow–Core	*Workflow–Definition*
II.	Workflow–Definitions–Einleser	XSLT–Prozessor	*XSLT–Programm*
III.	Workflow–Core und XSLT–Prozessor	Java–Interpreter	*Workflow–Core*, XSLT–Prozessor, weitere Bibliotheken
IV.	Java–Interpreter	PC mit Peripherie	Java Runtime Environment, Betriebssystem und Bibliotheken

Die kursiv gedruckten Einträge der Spalte Software müssen entwickelt werden, der Rest kann in Form von Standard–Produkten benutzt werden. Während Workflow–Core und XSLT–Programm zum Lieferumfang des programmierbaren Workflow–Systems gehören, muss eine Workflow–Definition für jeden Kunden individuell geschrieben oder zumindest abgewandelt werden.

2.1.5.2 Vorteile der Schichtung

Die Stärke des Konzepts der Trägersystem–Schichtung liegt darin, dass das Trägersystem getrennt beschrieben und dann als bekannt vorausgesetzt werden kann, wenn die Anwendung, die mit diesem Trägersystem realisiert wird, beschrieben wird. Damit wird die Komplexität größerer Systeme reduziert. Sind die beteiligten Trägersysteme nämlich identifiziert, können die Artefakte des Systems, also die selbst zu entwickelnde Software, diesen zugeordnet werden. Während im Code oftmals erzwungenermaßen Aspekte verschiedener Betrachtungsebenen nebeneinander liegen, können diese mit Hilfe der konzeptionellen Modelle den verschiedenen Betrachtungsebenen zugeordnet werden.

2.2 Der Begriff Architektur im Umfeld programmierter Systeme

Der Begriff der Architektur wird im Umfeld der programmierten Systeme vielfältig verwendet. Die Ursprünge, den Vergleich mit der Bauarchitektur auf programmierte Systeme anzuwenden, liegen schon Anfang der 1960er Jahre.

Es gibt kein einheitliches Verständnis des Begriffs Architektur informationeller Systeme. Die wichtigsten Definitionen des Architekturbegriffs kann man einteilen über ihren Bezug zu Strukturen oder Prozessen. Die strukturbezogenen Definitionen machen dabei den Hauptteil aus, was aus der Analogie zur Bauarchitektur schließlich nahe liegt.

2.2.1 Strukturbezogene Definitionen

Die am weitesten verbreitete Interpretation des Architekturbegriffs bezieht sich auf die Strukturen, die man in der Software finden kann. Typische Begriffe sind hierbei Komponenten wie beispielsweise Programmbibliotheken, Module oder Subsysteme.

Stellvertretend für die Reihe strukturbezogener Architekturdefinitionen wird hier diejenige zitiert, die Eingang in die Norm IEEE 1471–2000 gefunden hat:

> Architektur ist die *grundlegende Organisation* eines Systems, verkörpert durch seine *Komponenten*, deren *Beziehungen* untereinander und zur Umgebung, sowie die Prinzipien, nach denen sein *Entwurf und seine Weiterentwicklung* vorzunehmen sind.[7]

In der Definition der ECBS Working Group[8] tauchen hingegen neben Komponenten auch Verbinder und Einschränkungen als Architektur–Elemente auf:

> Systemarchitektur ist die grundlegende und abstrakte Struktur eines Systems, die deren Verhalten bestimmt und aus Komponenten, Verbindern und zugehörigen Nebenbedingungen besteht.[9]

Diese Definition hebt sich auch von anderen ab, in denen die Architektur synonym zur statischen Struktur gesehen wird, von der die Verhaltensbeschreibung zu trennen ist.[10]

Die Pattern–Literatur folgt der strukturbezogenen Architektur–Definition:

> *Software–Architektur* wird als Beschreibung der Subsysteme und Komponenten eines Software–Systems sowie deren Beziehungen definiert, wobei verschiedene Sichten benutzt werden, um die relevanten Eigenschaften zu zeigen.[11]
>
> *Komponenten* sind gekapselte Systemteile mit Schnittstelle.[12] Sie werden in Anlehnung an Perry und Wolf [PW92] in drei Kategorien eingeteilt: Processing Elements, Data Elements und Connecting Elements.
>
> Die *Beziehungen* (Relationships) zwischen Komponenten werden in statische (im Quelltext sichtbare) und dynamische (erst zur Laufzeit zu beobachtende) unterteilt.

[7] „Architecture (is) the *fundamental organization* of a system embodied in its *components*, their *relationships* to each other, and to the environment, and the principles guiding its design and evolution.“ [IEE00]

[8] ECBS steht für Engineering of Computer Based Systems.

[9] „System architecture is a system's *fundamental, abstract structure* which determines its behavior defined in terms of *components, connectors and constraints*“

[10] Bass, Clements und Kazman definieren Software Architecture als statische Struktur der Software und das Verhalten der Komponenten: „The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and their relationships among them“
„[...] The behavior of each component is part of the architecture insofar as that behavior can be observed or discerned from the point of view of another component“ [BCK98, S. 23f]

[11] „A *software architecture* is a description of the subsystems and components of a software system and the relationships between them. Subsystems and components are typically specified in different views to show the relevant functional and non–functional properties of a software system. The software architecture of a system is an artifact. It is the result of the software design activity.“ [BMR+96, S. 384]

[12] „A *component* is an ecapsulated part of a software system. A component has an interface. Components serve as the building blocks for the structure of a system. At a programming–language level, components may be represented as modules, classes, objects or a set of related functions.“ [BMR+96, S. 385]

> Unter *Software Design* versteht man den Entwurfsprozess, der zur Software–Architektur führt.[13]

2.2.2 Prozessbezogene Definitionen

Bei den prozessbezogenen Definitionen geht es um den Prozess der Software–Entwicklung. Das Schaffen von Architektur wird hier als Voraussetzung für den weiteren Entwurf gesehen, das heißt, dass Entscheidungen, die die Arbeit der Entwickler betreffen, frühzeitig getroffen werden müssen und damit die Architektur bestimmen. Architektur beschreibt damit zwar ebenfalls Strukturen; ob Strukturen aber als Architektur bezeichnet werden können, hängt von ihrer Rolle im Prozess ab.

Rechtin beschreibt die Aufgabe eines Architekten als Beherrschung der Komplexität des Systems. Er muss die Komplexität reduzieren und Unklarheiten oder Mehrdeutigkeiten beseitigen, um zu handhabbaren Arbeitspaketen zu kommen.[14]

Unter das Schaffen von Architektur, „Architecting“, fällt also das Aufteilen des komplexen Gesamtsystems in einfachere Teilsysteme, die besser handhabbar sind und für die Arbeitspakete für Spezialisten bzw. kleine Gruppen gemacht werden können („Engineering“). Hierfür müssen allerdings eine Menge an Entscheidungen getroffen werden, die zum Teil große Auswirkungen auf die Arbeit der Einzelnen haben werden und für die Anforderungen von Kunden, Entwicklern und Management berücksichtigt werden müssen.

Im Rahmen seiner Dissertation, in der er sich mit dem Software–Entwicklungsprozess und Architektur–Beschreibungen auseinander setzt, definiert Frank Keller Systemarchitektur folgendermaßen:

> „Die *Architektur* eines programmierten Systems entspricht einer Menge von *Systemmodellen*, welche nur jene Strukturen umfassen, die Voraussetzung für die effiziente *arbeitsteilige Entwicklung und Evolution* dieses Systems sind.“ [Kel03, S. 23]

Hier stehen Modelle des Systems im Vordergrund, welche die Arbeitsteilung beim Entwickeln und Fortführen eines programmierten Systems ermöglichen. Neben dem Aspekt der Abstraktion („nur jene Strukturen...“) taucht hier also noch der Aspekt der Systemschaffung auf, der den Grad der Abstraktion bestimmt.

2.2.3 Zur Abgrenzung von Architektur und Design

Eng verbunden mit den verschiedenen Definitionen von Architektur ist auch die Schwierigkeit der Abgrenzung zum Design.

Bei den strukturbezogenen Architekturdefinitionen erfolgt die Abgrenzung in der Regel durch den Abstraktionsgrad der Komponenten, die die Architektur ausmachen. Eine Struktur aus

[13] „*Software design* is the activity performed by a software developer that results in the software architecture of a system. It is concerned with specifying the components of a software system and the relationship between them within given functional and non–functional properties.“ [BMR+96, S. 390]

[14] „The architect's problem is to reduce this complexity to a manageable degree, specificially to the point where the powerful techniques of engineering analysis can be brought to bear. [...] The architect, therefore, is not a 'general engineer' but a specialist in *reducing complexity, uncertainty, and ambiguity* to *workable concepts*.“[Rec91, S. 13]

(abstrakten) Komponenten wird als Architektur bezeichnet, Strukturen, die nur innerhalb dieser Komponenten zu finden sind, als Design. Diese Sichtweise spiegelt sich in den Definitionen zu Architektur– und Design–Patterns wider, die in Abschnitt 3.4 vorgestellt werden.

Bei den prozessbezogenen Architekturdefinitionen spielt sich Design erst dann ab, wenn die Architektur festgelegt und die Arbeitsteilung erfolgt ist. Für Design sind einzelne Entwickler oder kleine Gruppen zuständig, deren Entwurfsentscheidungen nur wenige betreffen.

Eden und Kazman gehen in [EK03] einen anderen Weg, indem sie formale Strukturaussagen über das System nach zwei Kriterien untersuchen:

- Ist die Beschreibung Intensional oder Extensional[15]?
- Sind deren Auswirkungen lokal oder nicht–lokal[16]?

Diese Kriterien können formal überprüft werden, wenn die Strukturaussagen vollständig formal vorliegen, also beispielsweise als formale Spezifikation oder als Programmcode.

Die Einordnung nach Architektur, Design oder Implementierung erfolgt wie in folgender Tabelle beschrieben:

Architektur	Intensional	nicht–lokal
Design	Intensional	lokal
Implementierung	Extensional	lokal

In der Praxis dürften dabei zwei Probleme auftauchen:

1. Alle Strukturaussagen zu einem System müssen formal vorliegen
2. Der Lokalitätsbegriff ist abhängig von der Betrachtungsebene und davon, was man alles dem System zuordnet.

2.3 Sichtenkonzepte

In einem informationellen System lassen sich eine Vielzahl verschiedener Strukturen identifizieren. Diese können nach ausgewählten Aspekten, den Sichten, gruppiert werden. Die Ansätze der verschiedenen Sichtenkonzepte unterscheiden sich nun darin, welche Sichten unterschieden werden.

Für diese Arbeit wird nur das „Siemens 4“ Sichtenkonzept betrachtet, da dort der Begriff der konzeptionellen Sicht geprägt wird. Eine tiefer gehende Untersuchung weiterer Sichtenkonzepte hat Keller in [Kel03] vorgenommen.

[15] *Intensional*: Beschreibung einer Menge durch Angabe von Eigenschaften der enthaltenen Exemplare
Extensional: Beschreibung einer Menge durch Aufzählung ihrer Exemplare. Programmcode ist für Eden und Kazman eine extensionale Beschreibung.

[16] „Lokal“ wird in [EK03] so definiert:
Wenn ein Design–Modell *m* die Strukturaussage *S* erfüllt, dann ist *S* auch erfüllt, wenn *m* Teil eines umfassenderen Design–Modells ist — der Rest des umfassenden Design–Modells ist nämlich nicht davon betroffen.

2.3.1 Das „Siemens 4“ Sichtenkonzept

Das „Siemens 4“ Sichtenkonzept von Soni, Nord und Hofmeister [SNH95] umfasst vier Sichttypen, die im später erschienenen Buch „Applied Software Architecture“ [HNS00] um die Beschreibung einer Vorgehensweise ergänzt wurden.

2.3.1.1 Architektur–Begriffe

Da Soni, Nord und Hofmeister Architektur–Sichten vorstellen, klären sie zu Anfang die Begriffswelt rund um Software–Architektur:

Der Begriff *Software Architecture* umfasst hier die Strukturen, die in einem bestimmten programmierten System zu finden sind, die aber ausreichend abstrakt sind, um deren Komplexität handhabbar zu machen. Es werden Typen von Elementen, deren Aufgaben, Interaktionen und die im System vorkommenden Exemplare beschrieben.[17]

Verallgemeinert man Software Architecture, kommt man zum *Architectural Style bzw. Pattern*, in dem nur Typen von Elementen und Interaktionen beschrieben werden.[18] Bei der *Reference Architecture* kommt die Festlegung auf eine bestimmte Anwendungsdomäne hinzu, beispielsweise auf den Compilerbau, sowie die Zuordnung von Funktionalität auf bestimmte Elemente.[19] *Product Line Architecture* wiederum betrifft eine Klasse von programmierten Systemen („Produkte“) und kann bereits Teile der Implementierung umfassen.[20] *Software Architecture* beschreibt schließlich ein konkretes System und enthält daher auch seine Implementierung.

2.3.1.2 Die vier Sichten

Die Autoren unterscheiden vier Architektursichten, die zu verschiedenen Zwecken und in verschiedenen Phasen des Entwicklungsprozesses eingesetzt werden und die miteinander in Verbindung stehen:

- Konzeptionelle Sicht (Conceptual View)
- Modulsicht (Module View)
- Trägersystemsicht (Execution View)
- Konfigurationsmanagement–Sicht (Code View)

Die *Konzeptionelle Sicht* (Conceptual View) hat den höchsten Abstraktionsgrad und bezieht sich auf das gewünschte System. Hier werden die Begriffe der Anwendungsdomäne verwendet.[21] Die Konzeptionelle Sicht stellt daher nur Systemstrukturen dar.

[17] [HNS00, S. 4ff]: „*Software Architecture* [is] the purposeful design plan of a system.“
„[It] applies to one system and describes the element types, how they interact, how functionality is mapped to them, and the instances that exist in the system.“
„*Software Architecture* [...] is an abstraction that helps manage complexity. [It is] not a comprehensive decomposition or refinement of the system [...]“

[18] „An Architectural Style/Pattern [...] defines element types and how they interact“. Beispiel: Pipes & Filters

[19] „Reference Architecture / Domain–specific SW Architecture [...] define element types and allowed interactions and apply them to a particular domain“

[20] „A Product Line Architecture [...] defines element types, how they interact, and how the product functionality is mapped to them. It may ... define some instances of the architecture elements“

[21] „[The Conceptual View] describes the system in terms of its major design elements and the relationships among them, ... usually tied closely to the application domain“ [HNS00, S. 12]

Die *Modulsicht* (Module View) befasst sich mit der Dekomposition des Systems in Module und deren Einordnung in Schichten.[22] Sie zeigt also keine Systemstrukturen, sondern die Ableitung von Software–Strukturen aus der konzeptionellen Sicht, genauer gesagt die Strukturen der Quellen.

Die *Trägersystemsicht* (Execution View) befasst sich mit der Verteilung von Prozessen bzw. Threads und Daten auf Rechnerknoten und deren Kommunikation.[23] Sie hat damit Auswirkungen auf die Systemstruktur auf niedriger Abstraktionshöhe. Natürlich muss auch die Software–Struktur dafür ausgelegt sein, diese Verteilung zu ermöglichen.

Die *Konfigurationsmanagement–Sicht* (Code View) zeigt die Aufteilung der Module in auslieferbare Einheiten, also Executables, Libraries, Resource files und ähnliches.[24] Sie betrifft damit ausschließlich die Struktur der Software, aber im Gegensatz zur Modulsicht nicht den Quellcode.

Abbildung 2.6 zeigt die Beziehungen der vier Sichten untereinander im Hinblick auf die Aufgaben im Entwicklungsprozess und der äußeren Faktoren.

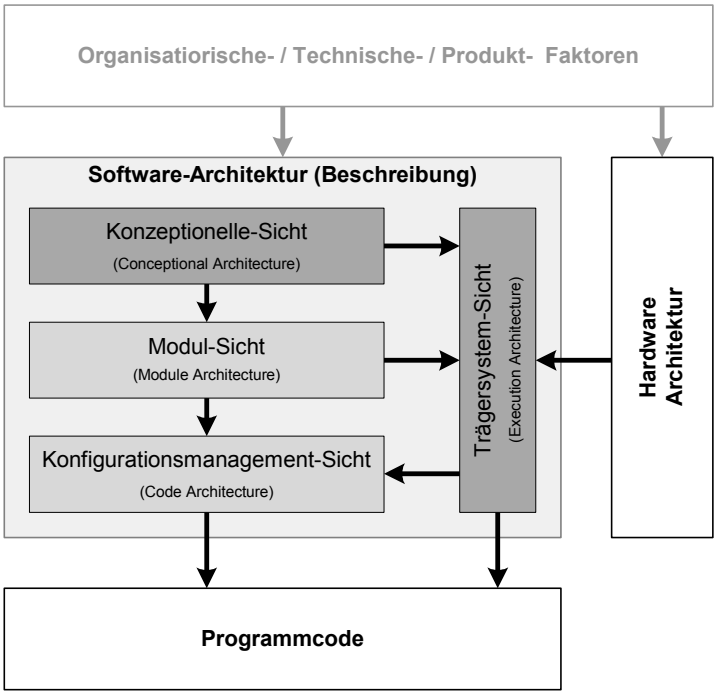


Abbildung 2.6: Das „Siemens 4“ Sichtenkonzept (Darstellung von Keller [Kel03] nach [HNS00, S.20])

Im Beispiel Online–Store aus Abschnitt 2.1 lassen sich die vier Sichten ebenfalls finden: Sowohl das gewollte System in Abbildung 2.1 als auch die implementierungsnähere Sicht in Abbildung 2.2 zeigen eine konzeptionelle Sicht. Modul– und Konfigurationsmanagement–Sichten betreffen die Quellen sowie den übersetzten und installierbaren Code in den beiden Speichern in Abbildung 2.4. Die Trägersystemsicht betrifft Abbildung 2.3, wobei hier die Abstraktion des Prozessor–Multiplex beim Multitasking vorgenommen wurde.

[22] „[The Module View is] the decomposition of the system and the partitioning of modules into layers“[HNS00, S. 11]

[23] „[The Execution View describes] how to allocate functional components to runtime entities, how to handle communication, coordination and synchronization among them“ [HNS00, S. 12]

[24] Code View: „The organization of the source code into object code, libraries and binaries, then in turn versions, files and directories affects the reusability of the code and the build time for the system.“ [HNS00, S. 11]

2.3.2 Mehrere Ebenen in der konzeptionellen Sicht

Hofmeister, Nord und Soni schlagen in [HNS00] vor, dass man ausgehend von den Kundenanforderungen ein konzeptionelles Modell erstellt, es hierarchisch verfeinert und daraus schließlich die Modul–Strukturen der Software ableitet. Es gibt hierbei aber durchaus auch Verfeinerungen, die nicht hierarchisch sind, weil sich die Strukturen ändern, nämlich die Implementierung mittels eines programmierten Systems oder mittels Multiplex. In diesen Fällen kann man, wie in Abschnitt 2.1.5 ausgeführt, von einem Wechsel der Betrachtungsebene sprechen.

Die Betrachtung der Implementierung führt aber nicht notwendigerweise auch direkt zum Code, denn erstens sind mehrere Stufen denkbar und zweitens kann es auch erforderlich sein, die Implementierung nicht–programmierter Komponenten zu berücksichtigen, beispielsweise das Protokoll, um einen Chipkartenleser anzusprechen.

Als Konsequenz folgt nun, dass auch dann von einer konzeptionellen Sicht gesprochen werden kann, wenn Komponenten auf realisierungsnäherer Ebene gezeigt werden.

Beispiele für konzeptionelle Sichten in mehreren Ebenen: Beim Online–Store tauchen in Abbildung 2.2 Standard–Komponenten wie HTTP–Server oder ein Datenbank–Verwaltungssystem auf, deren Bezug zur eigentlichen Anwendungsdomäne aus Abbildung 2.1.1 nur darin besteht, dass sie zur Realisierung eines Teils der Online–Store–Funktionalität benötigt werden.

In einer Firma gibt es typische Rollen in Produktion, Vertrieb, Buchhaltung oder Management. Um das Verhalten der Firma zu verstehen, ist es daher sinnvoll, ein konzeptionelles Modell mit Akteuren, die diese Rollen einnehmen, zu erstellen. Ist die Firma klein, füllen Mitarbeiter oft mehrere Rollen aus, beispielsweise macht ein Entwickler auch Akquisition und Support und ein Manager auch die Buchhaltung. In der Realität entsteht aus dem ersten konzeptionellen Modell ein weiteres konzeptionelles Modell, in dem die tatsächlich arbeitenden Personen vorkommen.

2.4 Fundamental Modeling Concepts (FMC)

2.4.1 FMC als Beschreibungsmittel für Systemstrukturen

Die Fundamental Modeling Concepts (FMC) bieten eine Begriffswelt und eine Darstellungstechnik zur Beschreibung informationeller Systeme. Dabei konzentriert sich die Darstellung auf Systemstrukturen. Aufbaustrukturen und daran erklärbare Abläufe werden in verschiedenen Diagrammtypen beschrieben, ebenso die Wertebereiche der Daten. Es werden wenige und einfache Darstellungselemente in den Diagrammtypen verwendet. Dadurch eignet sich FMC hervorragend für die Weitergabe von Wissen über informationelle System von Mensch zu Mensch.

Die Grundlagen von FMC wurden von Siegfried Wendt bereits 1974 im Rahmen eines Projekts mit Siemens geschaffen und später in vielen weiteren Industrieprojekten weiterentwickelt, sowie in diversen Publikationen veröffentlicht [Wen79, Wen82a, Wen82b, Wen91]. Die FMC–Website `fmc.hpi.uni-potsdam.de` bietet eine Vielzahl von Dokumenten und Veröffentlichungen zu diesem Thema. Der Name „Fundamental Modeling Concepts“ bzw. FMC wird dagegen erst seit 2002 verwendet und taucht erstmalig in einer Veröffentlichung über Wissenstransfer auf [KTG+02].

In dieser Arbeit bildet die Begriffswelt von FMC die Grundlage, um die verschiedenen Architektur–Definitionen und Begriffe aus der Pattern–Welt einzuordnen. Da Systemstrukturen in dieser Arbeit eine große Rolle spielen, werden auch bei der Betrachtung der Patterns in Kapitel 4 FMC–Aufbaubilder als Diagrammtyp zur Darstellung dieser Strukturen eingesetzt.

Die Besonderheit an FMC ist die Vorstellung, zuerst die Struktur des gewollten Systems mittels Aufbaumodellen darzustellen, und danach die Implementierung, also die Umsetzung in Hardware, Software und Infrastruktur zu betrachten. Die Aufbaumodelle sind dadurch abstrakter, da weiter von der Implementierung entfernt. Die so beschriebene Struktur des gewollten Systems ist keine „Software–Architektur“ — passender ist der Begriff „Systemarchitektur“, weil sie alle Beteiligten des Systems berücksichtigt.

2.4.2 FMC und das „Siemens 4“ Sichtenkonzept

Wie in Abschnitt 2.3 bereits gezeigt, werden sowohl in der konzeptionellen als auch in der Trägersystemsicht des „Siemens 4“ Sichtenkonzepts Systemstrukturen beschrieben. Die meisten FMC–Modelle wird man aufgrund ihres Abstraktionsgrads eher der konzeptionelle Sicht zuordnen, während sich die Trägersystemsicht für FMC–Modelle niedriger Betrachtungsebene eignet. Wegen der größeren Verbreitung des „Siemens 4“ Sichtenkonzepts kann mit dem Begriff „Conceptual Architecture“ also noch am ehesten eine Verbindung zu Systemstrukturen im Sinne der FMC erreicht werden.

In der konzeptionellen Sicht verwenden Soni, Nord und Hofmeister die Elemente Komponente und Verbinder, die mit den Akteuren und Kanälen der FMC Aufbaumodelle vergleichbar sind; allerdings nehmen Verbinder steuernde Aufgaben wahr, was bei FMC eindeutig Aufgabe von Akteuren ist. Weiterhin stellt FMC Speicherorte explizit dar, während bei „Siemens 4“ Daten nur in den Komponenten gespeichert werden können.

Der Betrachtungsebenenwechsel innerhalb der konzeptionellen Sicht ist ein mächtiges Mittel zur Komplexitätsreduktion, der von FMC gut unterstützt wird. Diese Möglichkeit fehlt bei „Siemens 4“.

Kapitel 3

Patterns

Patterns dienen der Weitergabe von Erfahrungswissen und stellen damit eine Bereicherung im Planungs– und Entwicklungsprozess eines informationellen Systems dar. In diesem Kapitel wird daher untersucht, was genau Patterns ausmacht, in welchen Formen und Kategorien sie auftreten und wie sie verwendet werden können.

3.1 Überblick

Ein Pattern ist die Beschreibung einer bewährten Lösung zu einem Problem, das in einem bestimmten Kontext auftritt.[1] Eng verbunden mit einem Problem sind Randbedingungen bzw. Trade–Offs (Forces), die in der Lösung berücksichtigt werden müssen. Da diese gegenläufig sein können, muss man bei der Lösung einen Kompromiss finden und ihn begründen. Wenn man nun die im Pattern vorgestellte Lösung auf das gegebene Problem anwendet, ergeben sich daraus Konsequenzen, die eventuell wiederum Ausgangspunkt für weitere Entwurfsentscheidungen und damit Kontext für weitere Patterns sind.

Nach einer kurzen Betrachtung der Geschichte der Patterns in Abschnitt 3.2 werden die Begriffe rund um Patterns im Abschnitt 3.3 vorgestellt.

Es gibt verschiedene Möglichkeiten, Patterns zu kategorisieren. Dazu gehören beispielsweise ihr Anwendungsbereich oder ihr Abstraktionsgrad. Einen Überblick über die Kategorien gibt Abschnitt 3.4. Die Kategorie der Architektur–Patterns für programmierte Systeme wird im Kapitel 4 vertieft behandelt.

Wie schon angedeutet führt die Anwendung einer Lösung oft zu einer Situation, in der weitere Probleme zu lösen sind. Derartige Zusammenhänge sind natürlich auch bei Patterns vorhanden. Deshalb ist es sinnvoll, Pattern nicht einzeln, sondern als System von Patterns für einen bestimmte Anwendungsbereich zu beschreiben. In diesem Zusammenhang taucht der Begriff der Pattern Language auf. Abschnitt 3.6 widmet sich diesem Thema.

Bei Patterns handelt es sich um Erfahrungen zur Problemlösung. Um später davon profitieren zu können, muss man die passenden Patterns anhand des Problems und seines Kontextes finden können. Aus diesem Grund werden Patterns gewöhnlich nach einem bestimmten Schema in einer festgelegten Form beschrieben. Die heute üblichen Pattern–Formen werden in Abschnitt 3.7 vorgestellt.

[1] „A pattern is a solution to a problem in a context“ (James Coplien in [Cop94, S. 19])

3.2 Geschichte

Die Verwendung des Begriffes *Pattern* (Muster) für technisches Design kommt aus dem Bereich der Bau–Architektur. Christopher Alexander brachte mit seinem Buch „A Pattern Language — Towns, Building, Construction“ [AIS+77] die Idee auf, dass gute, das heißt „lebendige“ Lösungen in der Architektur Gemeinsamkeiten haben, die ein Muster bilden.[2] Diese Gemeinsamkeiten aller guten Lösungen zu einem Problem stellt er in einem so genannten *Pattern* zusammen.

Ein Pattern beschreibt also die *Invariante* guter Lösungen zu einem Problem.

Alexander geht weiter: Eine gute Architektur könne man dadurch erhalten, dass man ein System von als gut empfundenen Teillösungen, den Patterns, in einer bestimmten Sequenz anwendet, um so ein lebendiges Gesamtbauwerk, beispielsweise einen Campus, zu erhalten. Mit dem Begriff der *Pattern Language* (Pattern–Sprache) unterstreicht er die Absicht, ein generatives Schema zu liefern, das zu guten Ergebnissen beim Entwurf führen soll. Dabei ist ihm wichtig, dass sich die Ergebnisse nicht notwendigerweise gleich aussehen müssen; Patterns geben Grundregeln vor, lassen aber gestalterische und künstlerische Freiheit zu. Ein Nebeneffekt besteht in der Schaffung einer gemeinsamen Sprachwelt, da die Namen der Patterns mit typischen Lösungen verbunden werden, so wie beispielsweise ein Elektroniker unter dem Begriff „Emitterschaltung“ einen bekannten und eindeutigen Schaltungstyp versteht.

Kent Beck und Ward Cunningham übertrugen 1987 diese Idee auf objektorientierte Software–Entwicklung, da gerade hier einerseits Erfahrungswerte zum Finden der Klassen benötigt wurden, andererseits viele Standard–Probleme immer und immer wieder auftauchen und gelöst werden müssen [BC87]. Jim Coplien stellte daraufhin eine Sammlung von C++–Idiomen, also sprachspezifischen Patterns zusammen [Cop91]. Die Autoren Gamma, Helm, Johnson und Vlissides veröffentlichten mit „Design Patterns“ [GHJV94] das erste Buch zu Software–Patterns. Es entstand ein regelrechter Boom um Patterns, die bald auch für andere Bereiche als die objektorientierte Software–Entwicklung eingesetzt wurden (siehe Abschnitt 3.4).

Die Benutzung von Patterns in der Software–Entwicklung war überraschend für Alexander. In seiner Rede auf der Konferenz OOPSLA 1996 (ACM Conference on Object–Oriented Programs, Systems, Languages and Applications) beschreibt er, dass Patterns in der Software als Austauschformat für Ideen zur Programmierung benutzt werden.[3] Dort berichtet er auch, dass der Ansatz seiner Pattern Language zu kurz gegriffen war, um damit gute Architektur zu erreichen, denn auch der Einsatz vieler guter Teillösungen führt in der Praxis nicht notwendigerweise zu einem guten Gesamtentwurf. Danach verfolgte er einen neuen Ansatz, Regeln für grundlegende und immer wiederkehrende Eigenschaften und Proportionen der Dinge — vom Türknopf bis zum Gebäude — festzuhalten, was zur Buchreihe „The Nature of Order“ führte.

Die Pattern–Community, die sich nach wie vor auf Pattern–Workshops in der ganzen Welt trifft und Patterns zusammen trägt, verfolgt Alexanders neuen Ansatz zwar mit Interesse; dessen Anwendbarkeit auf Software oder Organisationsformen erscheint aber eher zweifelhaft.

[2] Da er keinen passenden Begriff für die Einschätzung der „Lebendigkeit“ des Resultats findet, nennt er es „Quality without a name“.

[3] „[...] the pattern concept, for you, is an inspiring format that is a good way of exchanging fragmentary atomic ideas about programming.“ [Ale99, S. 75]

3.3 Pattern–Bestandteile

Die wichtigsten Bestandteile eines Patterns sind nach Corfman bzw. Coplien [Cor98, Cop98]:

Name	Ein treffender und einprägsamer Name für das Pattern
Problem	Das Problem, für das das Pattern eine Lösung anbietet
Kontext	Der Kontext, in dem das Problem auftaucht und die Lösung anwendbar ist
Trade–Offs (Forces)	Bedingungen, die in der Lösung berücksichtigt werden sollten und oft nur Kompromisse zulassen
Lösung	Die Lösung des Problems, oft ein Kompromiss bezüglich der Trade–Offs
Konsequenzen	Welche neue Situation durch die Anwendung der beschriebenen Lösung entsteht und welche Trade–Offs wie berücksichtigt wurden.

Die Pattern–Formen, die in Abschnitt 3.7 beschrieben werden, lassen weitere Bestandteile in einer Pattern–Beschreibung zu, die aber nicht zu den Kern–Bestandteilen eines Patterns gezählt werden.

3.3.1 Das Problem, sein Kontext und die Trade–Offs

Die Beschreibung der zu lösenden Aufgabe wird unterteilt in drei Bestandteile, nämlich Problem, Kontext und Trade–Offs.

Problem

Hier wird in einer knappen und präzisen Beschreibung ausschließlich das Problem, das gelöst werden muss, geschildert. Es handelt sich also um die „harten“ Anforderungen, die auf jeden Fall erfüllt werden müssen.

Kontext

Unter Kontext versteht man hier Hintergrundinformationen zum Problem, also in welchem Zusammenhang dieses Problem auftritt. Der Kontext dient dazu, die Problemstellung zu präzisieren.

Welche Teile der Problembeschreibung in die Abschnitte „Problem“ und „Kontext“ gehören, beschreibt eine einfache Regel: Nach Anwendung der Lösung besteht das Problem nicht mehr, während der Kontext davon nicht beeinflusst wird.

Trade–Offs (oder: Forces)

Es gibt zu einem Problem und einem Kontext weitere Faktoren, die bei der Wahl der Lösung berücksichtigt werden sollten, aber möglicherweise in Konflikt miteinander stehen. Im Gegensatz zum Problem handelt es sich um „weiche“ Anforderungen, deren Erfüllung wünschenswert, aber nicht zwingend ist. Es gilt, mit der Lösung einen Kompromiss zu finden. Aus diesem Grund wird auch gerne von einer Spannung (tension) gesprochen, die durch die Trade–Offs aufgebaut wird. [4]

Die Entscheidungen bezüglich der Trade–Offs dienen dazu, alternative Lösungen zu bewerten.[5]

Zur Abgrenzung zwischen Kontext und Trade–Offs dient eine ähnliche Regel wie oben: Der Kontext wird durch die Anwendung der Lösung nicht beeinflusst, bei den Trade–Offs ist das zumindest möglich.[6]

Beispiel „Ein Frankfurter Geschäftsmann möchte nach Berlin reisen, weil er dort um 10:00 Uhr einen Vertrag mit einem Kunden verhandeln will. Die Zeit zwischen Abreise und Ankunft soll möglichst kurz sein, die Reise komfortabel und günstig sein. Was muss er tun?“

Problem	Eine Reise von Frankfurt nach Berlin mit Ankunft um 10:00 Uhr in der Stadt
Kontext	Beim Kunde soll ein Vertrag abgeschlossen werden. Beim Geschäftsmann handelt es sich um einen Manager, dessen Arbeitszeit teuer ist.
Trade–Offs	Die Reise soll schnell, komfortabel und preisgünstig sein und eine individuelle Startzeit zulassen.

3.3.2 Die Lösung und ihre Konsequenzen

Ein Pattern beschreibt eine Lösung zu einem Problem. Zur Bewertung der Lösung dient die Beschreibung der Konsequenzen, die sich aus ihrer Anwendung ergeben.

Lösung

Die durch das Pattern vorgeschlagene Lösung des Problems wird üblicherweise als Handlungsanweisung formuliert.

[4] „*Forces* may be boundary conditions, limiting factors, competing goals, or constraints imposed upon the set of alternatives you can realistically consider. They may be quantitative constraints on things like performance or size, or coupling. They may be principles or tenets like high cohesion + low coupling. [...] the forces section should be discussing constraints, and competing concerns about the *entire problem space*! A really strong set of forces will make it look like you are between a rock and a hard place.“ Brad Appleton in [App98]

[5] „All the forces (trade–offs) affect the different contexts [of the problem], but the strengths of these forces vary and some may have no impact at all. The best solution is the one that most completely resolves these forces and the tension between them. [...]
When finding a solution to a problem, a designer must weigh the advantages and disadvantages of each alternative, and make trade–offs to decide which solution is the best.“ [Cor98]

[6] Paul McKenney in [McK96]: „I currently use the following question to help me decide:
Is this thing something that choosing this pattern can affect (a force), or is this thing fixed regardless of the choice of pattern (a context)?
But when I use this question, I find that the forces at one level of abstraction (sometimes!) become the contexts of the next lower level of abstraction.
[...] Forces are those things that this pattern can change, contexts are those things that this pattern cannot change, and the pattern itself embodies the wisdom to know which is which.“

Konsequenzen

Im Abschnitt Konsequenzen werden zwei verschiedene Dinge beschrieben:

- Einerseits geht es um den gewählten Kompromiss bezüglich der Trade–Offs (deswegen wird der Abschnitt manchmal auch „Force Resolution“ genannt), und damit um die Vor– und Nachteile dieser Lösung.
- Andererseits wird auf Folge–Probleme (und günstigerweise auch auf passende Patterns) hingewiesen, weswegen auch von Verpflichtungen („Liabilities“) gesprochen wird.

In einer Pattern–Beschreibung gibt es im Abschnitt „Konsequenzen“ üblicherweise zwei Teile:

Benefits Welche Vorteile durch die Anwendung des Patterns entstehen und welche Trade–Offs dadurch aufgelöst wurden

Liabilities Die Nachteile dieser Lösung und welche Trade–Offs bestehen bleiben, sowie neue Probleme, die nun auch zu lösen sind, also neue Pflichten.

Beispiel „Der Frankfurter Geschäftsmann soll die Reise mit Flugzeug / Taxi machen und den ersten Flug nach Berlin ab Frankfurt nehmen. Zwar sind die Reisekosten dann hoch, dafür ist er rechtzeitig beim Kunden. Ansonsten müsste er entweder am Tag vorher fahren und in Berlin übernachten, oder den Nachtzug nehmen. Beides erfordert aber deutlich mehr Zeit.“

Lösung Nehmen Sie den ersten Linienflug ab Frankfurt und in Berlin ein Taxi.

Konsequenzen Der Zeitaufwand ist gering, und der Komfort hoch. Die Dichte der Linienflüge am Morgen ist ausreichend für eine kurze Wartezeit. Die Reisekosten sind hoch, eventuell muss das Reisebudget aufgestockt werden.

3.4 Kategorien von Patterns

Patterns waren ursprünglich für die Bau–Architektur gedacht. Durch die Übernahme der Pattern–Idee für Software–Systeme entstand bald ein Boom, in dem Patterns für weitere Bereiche gesammelt wurden. Dieser Abschnitt beschreibt die wichtigsten Kategorien von Patterns, um die Bandbreite der Arbeit der Pattern–Community vorzustellen.

3.4.1 Patterns für Software–Systeme

Hier geht es um Patterns, die Erfahrungen bei der Strukturierung von Software und bei der Lösung typischer Design–Probleme für Software–Entwickler festhalten.

3.4.1.1 Patterns für objektorientiertes Design

Gamma et al. unterscheiden in ihrem Buch „Design Patterns“ [GHJV94] Patterns nach ihrem Einsatzzweck hinsichtlich des Problems:

Creational Patterns: Lösungen für Probleme im Zusammenhang mit der Erzeugung von Objekten und Objektverbünden.[7]

[7] „Creational patterns concern the process of object creation“ [GHJV94, S.10].

Structural Patterns: Lösungen für Probleme, die Objektstrukturen erfordern, um die Dekomposition des Programms in Klassen zu erleichtern oder Objektstrukturen zur Laufzeit umbauen zu können.[8]

Behavioral Patterns: Lösungen für Probleme, um Verhaltensbeschreibungen geschickt aufzubauen und auf Klassen zu verteilen. [9]

Diese Klassifikation wurde von der Pattern–Community bald fallen gelassen, da sie zu sehr auf objektorientierte Technologie zugeschnitten war.

3.4.1.2 Architektur– und Design–Patterns

Buschmann et al. teilen in ihrem Buch „Pattern oriented Software Architecture“ (kurz: POSA) [BMR+96] Patterns in drei Kategorien ein, die an Granularität, Größenordnung und an Abstraktion ausgerichtet sind:

Architectural Patterns: Patterns zur grundlegenden Strukturierung des Softwaresystems. Die eigentliche Funktionalität steckt in Subsystemen, deren Einbindung durch das Pattern geregelt wird.[10]

Design Patterns: Patterns, die auf Ebene der Subsysteme oder Komponenten eines Software–Systems angewandt werden können. Im Gegensatz zu den Architektur–Patterns haben sie nur lokalen Einfluss auf das System. [11]

Idioms: Patterns, deren Lösungen nur für eine bestimmte Programmiersprache gelten.[12]

Beispiel: Übertragen auf den Maschinenbau hieße das: Mit einem Architektur–Pattern wird beschrieben, wie man beispielsweise bei einem Auto zweckmäßig Motor, Getriebe, Lenkung und Räder koppelt, so dass man auf die Vorderräder sowohl die Motorkraft zum Antrieb als auch den Lenkeinschlag bringen kann. Ein Design–Pattern beschreibt beispielsweise, wie man ein Getriebe auslegt. Ein Idiom behandelt beispielsweise die Auslegung der Lager eines Getriebes unter Verwendung eines speziellen Kugellagertyps.

Weiterhin grenzen die Autoren die Architektur–Patterns ab von Architektur–Stilen und Frameworks:

Architectural Style: Architektur–Stile ähneln den Architektur–Patterns, unterscheiden sich aber in folgenden Punkten: Sie sind relativ grob („overall structural framework“), während Architektur–Patterns in verschiedenen Detailgraden existieren. Weiterhin

[8] „Structural patterns deal with the composition of classes or objects“[GHJV94, S.10].

[9] „Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility“ [GHJV94, S.10].

[10] „An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.“ [BMR+96, S. 12].

[11] „A *design pattern* provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly–recurring structure of communicating components that solves a general design problem within a particular context.“ [BMR+96, S. 13]

[12] „An *idiom* is a low–level patterns specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between them using the features of the given language.“ [BMR+96, S. 14]

können Architektur–Patterns weitere Patterns enthalten oder mit einem anderen „interagieren“. Schließlich präsentieren Patterns eine Lösung für ein bestimmtes Entwurfsproblem. Diesen Fokus auf Problem und Lösung haben Architektur–Stile nicht.[13]
In Abschnitt 5.2.1 werden acht Architektur–Stile von Shaw vorgestellt.

Framework: Ein Framework ist ein halbfertiges Software–System mit Vorgaben zur Vervollständigung. Während ein Pattern nur Handlungsanweisungen als Lösung anbietet, gibt ein Framework Software vor, die an bestimmten Stellen nach vorgegebenen Regeln ergänzt werden muss.[14]

In einer Untersuchung objektorientierter Frameworks gibt Kleis zwei Definitionen für Frameworks an. In der architekturbezogenen ist ein Framework ein Software–Halbfertigprodukt, das eine Software–Architektur für eine Familie von Systemen oder Systemkomponenten vorgibt. Aus Sicht der Software–Strukturen ist ein Framework ein als Halbfertigprodukt vorliegendes Teilprogramm, das mittels Einschubmethoden realisierte Platzhalter für die geplante Ergänzung durch den Kunden enthält [Kle99, S. 119].

3.4.1.3 Analyse– und konzeptionelle Patterns

Fowler beschreibt in seinem Buch „Analysis Patterns“ [Fow97], dass man bereits bei der Analyse auf Erfahrungen in Form von Analyse–Patterns zurückgreifen sollte. Bei der Analyse wird das Problem mit Hilfe eines konzeptionellen Modells erfasst, im Unterschied zum Design, bei dem ein System als Lösung für das Problem entworfen wird. Gut gewählte Modelle ermöglichen eine Abstraktion und damit Vereinfachung des Problems. Analyse–Patterns beschreiben Daten, Relationen und Abläufe, die typisch für eine analysierte Domäne sind und unterstützen damit die Erstellung von Konzeptmodellen, die als Ausgangspunkt für das Design der Lösung dienen.[15].

Auch wenn es zunächst so klingt, erstellt Fowler bei der Analyse kein Systemmodell, aus dem klar wird, welche Aufgaben und Schnittstellen die zu bauende Komponente haben soll. Vielmehr beschreiben seine Analyse–Patterns Datenstrukturen und Prozesse, die im Problembereich vorkommen und sich gut in Software abbilden lassen. Für die von ihm beschriebenen Domänen aus dem Bereich der Betriebswirtschaft ist das aber auch ausreichend.

Auch Riehle und Züllighoven sprechen im Rahmen der Analyse von Patterns in konzeptionellen Modellen, die sie zuerst „Interpretations– und Gestaltungsmuster“ und später „Conceptual Patterns“ nennen. Es handelt sich dabei um Patterns, die während der Erfassung der Anforde-

[13] „An *architectural style* defines a family of software systems in terms of their structural organization. An architectural style expresses components and the relationships between them, with the constraints of their application, and the associated composition and design rules for their construction.“ [BMR+96, S. 394] basierend auf [PW92].

[14] „A *framework* is a partially complete software (sub–) system that is intended to be instantiated. It defines the architecture for a family of (sub–) systems and provides the basic building blocks to create them. It also defines the places where adaptions for specific functionality should be made. In an object–oriented environment a framework consists of abstract and concrete classes.“ [BMR+96, S. 396]

[15] „When doing analysis you are trying to understand the problem. [...] Analysis also involves looking begind the surface requirements to come up with a mental model of what is going on in the problem. [...] To do this we create a conceptual model—a mental model that allows us to understand and simplify the problem. Some kind of conceptual model is a necessary part of software development [...]. The difference is whether we think about conceptual modeling as a process in itself or as one aspect of the entire software design process.“ [Fow97, S. 1f.]

rungen (Requirements) und der Erstellung des konzeptionellen Modells relevant sind und in den Begriffen der untersuchten Domäne formuliert sind.[16]

3.4.2 Prozess– und Organisations–Patterns

Auch Erfahrungen in den Bereichen der Entwicklungsprozesse und Organisationsstrukturen von Firmen und Teams werden in Patterns festgehalten. James Coplien stellt in seinem Beitrag „A generative development — Process pattern language“ [Cop95] Prozess–Patterns vor, mit denen man Organisationsstrukturen und darin erforderliche Prozesse gestalten kann („[...] patterns that can be used to shape a new organization and its development processes“). So wie Alexanders Patterns zur Bauarchitektur auf Gebäude angewendet werden können, so sollen Prozess–Patterns auf die Organisationsstrukturen von Firmen anwendbar sein.

3.4.3 Didaktische Patterns

Erfahrungen mit der Weitergabe von Wissen an Schüler, Studenten und Schulungsteilnehmer bilden die Familie der didaktischen Patterns (auch: pedagogical patterns). In ihrem Beitrag „Patterns for Classroom Education“ [Ant96] stellt Dana Anthony beispielsweise ihre Erfahrungen mit der Konzeption eines Kurses über objektorientierte Technologie als Pattern–Sammlung vor.

3.5 Anti–Patterns

Während man mit Patterns den Ansatz verfolgt, bewährte Lösungen typischer Probleme weiterzugeben, werden mit Anti–Patterns naheliegende, aber nicht funktionierende Lösungen beschrieben, um diese zu vermeiden, falls man das dazugehörige Problem lösen soll. Anti–Patterns dienen also dazu, auch schlechte Erfahrungen weiterzugeben, damit andere sie vermeiden können. Sie kommen dann in Frage, wenn es viele mögliche Lösungen gibt, von denen aber die nahe liegenden schlecht sind, also Fallen darstellen.

In seinem Beitrag „Patterns and Antipatterns“ beschreibt Andrew Koenig Anti–Patterns als Patterns, die statt einer Lösung etwas anbieten, was zwar wie eine Lösung aussieht, aber nicht funktioniert.[17] Weiterhin schlägt er vor, Anti–Patterns mit Patterns zu koppeln, also neben die nahe liegende, nicht funktionierende Lösung eine echte, funktionierende zu stellen.

Anti–Patterns kommen in allen Pattern–Kategorien vor.

[16] „A *conceptual pattern* is a pattern whose form is described by means of the terms and concepts from an application domain.“ [RZ96].

[17] „An antipattern is just like a pattern, except that instead of a solution it gives something that looks superficially like a solution but isn't one.“ [Koe98]

3.6 Pattern–Systeme und Pattern Languages

Patterns stehen in der Regel nicht alleine. Alexander strukturierte die Bauarchitektur–Patterns durch ein Schema, das er *Pattern Language* nannte. Dahinter steht die Intention, dass man durch die Entscheidung, ein Pattern einzusetzen, auf neue Probleme stößt, für die es wiederum Patterns gibt.[18]

Auch im Bereich der Software–Patterns stellten Gamma et al. in [GHJV94, S. 10f] fest, dass es Beziehungen zwischen Patterns gibt und dass manche häufig zusammen mit anderen oder alternativ eingesetzt werden.

Coplien definiert eine Pattern Language als strukturierte Sammlung aufeinander aufbauender Patterns, mit deren Hilfe aus Anforderungen und Einschränkungen schließlich eine Architektur gewonnen werden soll. Dabei ergibt sich nach der Anwendung eines einzelnen Patterns immer ein resultierender Kontext, der wiederum Ausgangspunkt für die Anwendung des nächsten Patterns ist.[19]

Zu einer Pattern Language gehört immer auch einen Leitfaden (*Guideline*), in dem beschrieben wird, in welcher Reihenfolge welche Patterns anzuwenden sind und welche Alternativen und Kombinationsmöglichkeiten bestehen.

Völter, Schmid und Wolff klassifizieren in [VSW02, S.8f] folgende Kategorien zusammengehörender Patterns:

Compound Patterns (Zusammengesetzte Patterns) sind Patterns, die durch eine Kombination kleinerer bekannter Patterns entstehen.

Family of Patterns beschreibt eine Sammlung von Patterns, die für ein Problem verschiedene Lösungen anbietet.

System of Patterns beinhaltet eine Reihe von Patterns für einen bestimmten Problembereich. Häufig gibt es Beziehungen zwischen den Patterns.

Pattern Language ist ein System von Patterns mit Anwendungsvorschrift (Guideline) bezüglich Reihenfolge und Auswahl der Patterns, um ein bestimmtes Designziel zu erreichen. Die Beziehungen zwischen den Patterns sind hier am strengsten und die Patterns sind eventuell isoliert betrachtet wertlos.

[18] „An ordinary language like English is a system which allows us to create an infinite variety of one–dimensional combinations of words, called sentences. [...] A *pattern language* is a system which allows its users to create an infinite variety of those three–dimensional combinations of patterns which we call buildings, gardens, towns.

Thus, as in the case of natural languages, the pattern language is generative. It not only tells us the rules of arrangement, but shows us how to construct arrangements — as many as we want — which satisfy the rules.“ [Ale79, S. 185–186]

[19] „A pattern language is a structured collection of patterns that build on each other to transform needs and constraints into an architecture. It is not a programming language in any ordinary sense of the term, but is a prose document whose purpose is to guide and inform the designer. Patterns rarely stand alone. Each pattern works on a context, and transforms the system in that context to produce a new system in a new context. New problems arise in the new system and context, and the next „layer“ of patterns can be applied.

[...] Good pattern languages guide the designer toward useful architectures [...] Good architectures are durable, functional, and aesthetically pleasing, and a good combination of patterns can balance the forces on a system to strive toward these three goals.[...]“ [Cop98]

3.7 Pattern–Formen

Da mit einem Pattern Erfahrungen an andere weitergeben werden sollen, und ein einzelnes Pattern möglicherweise in einem Katalog von Patterns neben vielen anderen auftauchen kann, ist es wichtig, dass Patterns eine klare Struktur haben, die das Wesentliche verdeutlicht. Der Leser muss schnell erfassen können, um welches Problem in welchem Kontext es geht und wie der Lösungsansatz aussieht.

In der Pattern–Literatur haben sich eine Reihe von Möglichkeiten ausgebildet, eine Pattern–Beschreibung zu strukturieren. Eine Aufstellung von Pattern–Formen lässt sich im Portland Pattern Repository [Wik02] finden. Dabei dominieren stark gegliederte Pattern–Beschreibungen, da diese den Vorteil bieten, besser in einen Pattern–Katalog zu passen, der eine einfache Übersicht und Vergleichbarkeit verschiedener Patterns ermöglicht.

Im Folgenden werden vier verschiedene Pattern–Formen vorgestellt, die entweder typisch für eine ganze Klasse von Pattern–Formen, oder, wie im Fall der „Gang–of–Four“ Form, aus historischen Gründen interessant sind.

3.7.1 „Alexandrische“ Form

Diese Form, wie in Abbildung 3.1 skizziert, wurde von Christopher Alexander in seinem Buch „A Pattern Language“ [AIS+77] über Bau–Architektur verwendet.

Name des Patterns	**SMALL MEETING ROOMS**
Ein typisches Bild	
Prolog: Kontext	[...] Investigation of meeting rooms show that the best distribution — both size and position — is rather unexpected.
	* * *
Das Problem: Kurzfassung	**The larger meetings are, the less people get out of them.**
Das Problem: Details und Trade-offs (Forces)	We first discuss the sheer size of meetings. [...]
	Therefore:
Die Lösung: Kurzfassung	**Make at least 70% of all meeting rooms really small — for 12 people or less. Locate them in the most public parts of the bulding [...].**
Die Lösung: Details, Konsequenzen	[...]
	* * *
Beispiele, verwandte oder ähnliche Patterns	[...] People will feel best if many of the chairs are different, to suit different temperaments and moods and shapes and sizes — DIFFERENT CHAIRS (251). [...]

Abbildung 3.1: Die „Alexandrische“ Form aus „A Pattern Language“ am Beispiel SMALL MEETING ROOMS [AIS+77]

Zuerst kommt ein Bild, meist ein Foto, das typisch für dieses Pattern ist. Ein einleitender Abschnitt beschreibt den Kontext und setzt es in Beziehung mit anderen Patterns. Drei „Diamonds“ bzw. Sterne (* * *) dienen dazu, den Anfang des eigentlichen Patterns zu markieren.

Ein fett gedruckter Abschnitt beschreibt kurz den Kern des Problems. Danach wird das Problem ausführlich beschrieben und diskutiert, auch mit Studien und Statistiken belegt. Den Abschluss bildet das Wort „Therefore“ (daher), das die Lösung einleitet.

Der Kern der Lösung wird in Form von Anweisungen in einem kurzen fett gedruckten Abschnitt skizziert, und dann gegebenenfalls in weiteren Absätzen ausgeführt. Ein Diagramm veranschaulicht die Lösung. Drei „Diamonds“ bzw. Sterne (* * *) schließen jetzt das Pattern ab. Der abschließende Absatz dient nun dazu, auf Beispiele und weiterführende Patterns zu verweisen.

Diese Form der Pattern–Beschreibung erfordert neben gutem Verständnis der Domäne auch eine gewisse schriftstellerische Begabung. In der Pattern–Literatur zieht man daher meist andere Formen vor, in denen eine strengere Struktur mit Absätzen und vorgegebenen Überschriften verwendet werden.

3.7.2 „Gang–of–Four“ Form

Die „Gang–of–Four“ Form wurde von Gamma, Helm, Johnson und Vlissides in ihrem ersten Buch über Design–Patterns [GHJV94] eingeführt.

Name	COMPOSITE
Classification	Object Structural
Intent	Compose Objects into tree structures to represent part–whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Also Known As (Aliases)	—
Motivation	Graphical applications use both primitives and containers [...]
Applicability	Use the Composite pattern when you want to represent part–whole–hierarchies of objects [...]
Structure	[Class diagram, Object diagram]
Participants	Component, Leaf, Composite, Client [...]
Collaborations	Clients use the Component class interface to interact with objects in the composite structure. [...]
Consequences	Benefits and liabilites of the Composite pattern: [...]
Implementation	Issues related to the implementation of the Composite pattern [...]
Sample Code	[...]
Known Uses	Smalltalk Model/View/Controller (MVC), [...]
Related Patterns	DECORATOR, FLYWEIGHT, ITERATOR, VISITOR [...]

Abbildung 3.2: Die „Gang–of–Four“ Form am Beispiel des COMPOSITE Patterns [GHJV94, S. 163ff]

Schon bald erkannte die junge Pattern–Community, dass diese Form überarbeitet werden muss. So fehlt beispielsweise ein eigener Abschnitt für die Problembeschreibung, und die Form ist zu sehr auf objektorientierte Technologie und damit verbundene Anwendungen, beispielsweise Benutzeroberflächen (GUIs), spezialisiert.

3.7.3 „POSA“–Form

Die „POSA“–Form wird von Buschmann, Meunier, Rohnert, Sommerlad und Stal in ihrem Buch „Pattern Oriented Software Architecture“ [BMR+96] verwendet und im zweiten Band [SSRB00] verfeinert.

Name	REACTOR
Abstract	Allows event–driven applications to demultiplex and dispatch service requests [...]
Also knwon as (Aliases)	Dispatcher, Notifier
Example	Server for distributed logging servive [...]
Context	An event–driven application receives multiple service requests simultaneously, but processes them synchronously and serially.
Problem and Forces	A server must demultiplex and dispatch service requests. [...] Forces: Improve scalability and latency, maximize throughput, minimize effort to integrate new services [...]
Solution	Synchronously wait for events, decouple demultiplexing and dispatching from application [...]
Structure	Handles, Demultiplexer, Event Handler, Reactor [...] [Class diagram]
Dynamics	Application registers Event Handler [...] [Sequence diagram]
Implementation	1. Define the event handler interface [...]
Example resolved	Our loggin server uses a singleton reactor [...]
Variants	Thread–safe reactor , Concurrent Event Handlers, [...]
Known Uses	InterViews, Xt toolkit, ACE Framework, [...]
Consequences	Benefits: Separation of concerns, Modularity, [...] Liabilities: Restricted applicability, Non–pre–emptive, [...]
See also	OBSERVER, PUBLISHER–SUBSCRIBER, ACTIVE OBJECT, [...]
Acknowledgements or Credits	John Vlissides, [...]

Abbildung 3.3: Die „POSA“ Form am Beispiel des REACTOR Patterns [SSRB00, S. 179ff]

Die Beschreibung der Patterns in [BMR+96] und [SSRB00] ist mit 25–30 Seiten pro Pattern recht umfangreich. Das gilt insbesondere für den Abschnitt zur Implementierung.

3.7.4 Kanonische Form

Die Kanonische Form schreibt eine strenge Gliederung unter Verwendung der dargestellten Überschriften vor. Sie geht zurück auf Jim Coplien, der 1994 vorschlug, das Wesen der Patterns nach Christopher Alexander unabhängig von der verwendeten Form zu halten. Daher muss es als kleinsten gemeinsamen Nenner in der Beschreibung eines Patterns zumindest eine klare Definition des Problems, der Trade–Offs (Forces) und der Lösung geben.

Im Kontext werden die Situation und Umgebung beschrieben, in denen dieses Pattern anwendbar ist. Der resultierende Kontext wiederum beschreibt, wie der Einsatz des Patterns die Trade–Offs abdeckt und welche Vorteile und welche Konsequenzen sich daraus ergeben. Unter „Ra-

tionale“ können schließlich weitere Gründe für die Wahl der Lösung angeführt werden. Diese Form der Patternbeschreibung ist — in Abwandlungen — am häufigsten anzutreffen.

Name	ARCHITECT CONTROLS PRODUCT
Alias (Also knwon as)	—
Problem	A product designed by many individuals lacks elegance and cohesiveness.
Context	An organization of Developers that needs strategic technical direction.
Forces	Totalitarian control is viewed by most development teams as a draconian measure. The right information must flow through the right roles.
Solution	Create an Architect role. The Architect should advise and control Developers, and should communicate closely with them. The Architect should also be in close touch with the Customer.
Example	—
Resulting Context	[...]
Benefits	This provides technical focus, and a rallying point for technical work as well as market–related work. [...]
Liabilities	Resentment can build against a totalitarian Architect; use patterns like REVIEW THE ARCHITECTURE to temper this one.
Rationale	[...] While the Developer controls the process, the Architect controls the product. The Architect is a „chief Developer“ (See also ARCHITECT ALSO IMPLEMENTS). His responsibilities include understanding requirements, framing the major system structure, and controlling the long–term evolution of that structure. [...]
Known Uses	—
Related Patterns	—

Abbildung 3.4: Die Kanonische Form am Beispiel des Prozess–Patterns ARCHITECT CONTROLS PRODUCT [Cop95]

Kapitel 4

Architektur– und Design–Patterns in der Literatur

Architektur–Patterns sind eine Kategorie von Patterns, die im Zusammenhang mit der Architektur von Softwaresystemen verwendet werden — siehe auch Abschnitt 3.4. Sie gewinnen zunehmend an Bedeutung, da die zu entwickelnden Systeme immer größer und komplexer werden und Erfahrungen mit der Planung ihrer Struktur, also ihrer Architektur, daher immer wertvoller werden.

Untersuchungsgegenstand dieses Kapitels ist eine Auswahl von Architektur– und Design–Patterns aus der einschlägigen Pattern–Literatur (siehe Abschnitt 4.1). Sie werden im Abschnitt 4.2 in kompakter Form vorgestellt, wobei zur Darstellung durchgängig FMC–Aufbaubilder eingesetzt werden, da hier der Schwerpunkt auf die Systemstrukturen gelegt wird.

Die hier verwendete thematische Einteilung der Patterns unterscheidet sich von der Einordnung in der Literatur, wobei allerdings auch dort die thematische Einordnung mancher Patterns schwer zu entscheiden oder zumindest uneinheitlich ist. Dieses Thema wird in Abschnitt 4.3 behandelt.

Da der Begriff der Architektur von Softwaresystemen alles andere als eindeutig ist (siehe Abschnitt 2.2), wird in Abschnitt 4.5 anhand der vorgestellten Patterns geprüft, wie die Autoren von Architektur–Patterns diesen Begriff verstehen und ihn vom Design abgrenzen.

Patterns stehen nicht alleine. In der betrachteten Literatur werden sie in ein Pattern–System oder eine Pattern Language eingeordnet und in Beziehung zu anderen Patterns gesetzt. Die dabei auffallenden Unterschiede in der Abstraktionshöhe der Patterns werden in Abschnitt 4.4 behandelt. Abschnitt 4.6 zeigt die Schwierigkeiten, die durch die Einordnung von Patterns verschiedener Abstraktionshöhe in ein Pattern–System entstehen.

Schließlich werden die gewonnenen Erkenntnisse in Abschnitt 4.7 zusammengefasst und bewertet.

4.1 Pattern–Literatur

Nachdem Gamma et al. in „Design Patterns“ [GHJV94] Patterns für das Design objektorientierter Software vorgestellt hatten, zeigten Shaw und Garlan in „Software Architecture“ [SG96], dass es auch im Bereich der Software–Architektur Patterns bzw. Stile gibt, die Shaw in kompakter Form auf einem Pattern–Workshop vorstellte [Sha96].

Buschmann et al. stellen im Buch „Pattern–oriented Software Architecture“ (POSA1) [BMR+96] zusammen mit ihrer Sammlung ausführlich beschriebener Patterns ihre Definition des Begriffs Architektur–Pattern vor. Der zweite Band mit dem Untertitel „Patterns for Concurrent and Networked Objects“ (POSA2) [SSRB00] konzentriert sich auf die Bereiche Multitasking und Verteilung, in denen System–Architektur eine größere Rolle spielt und die als schwieriger zu handhaben gelten. Das Buch „Server Component Patterns“ von Völters et al. [VSW02] zielt wiederum auf Patterns im Bereich der komponentenorientierten Application Server, daher auch der Untertitel „Component Infrastructures Illustrated with EJB“. Viele Patterns kommen aus dem Bereich Enterprise Java Beans und werden mit Java–Beispielen belegt. Weniger auf Architektur als auf die Beherrschung von Nebenläufigkeit ist das Buch „Concurrent Programming in Java — Design Principles and Patterns“ von Lea [Lea99] ausgerichtet.

Die Bücher sind das Ergebnis der Arbeit einer aktiven Community, die mit speziell auf Patterns ausgerichteten Workshops, den so genannten PLoPs (Pattern Languages of Programs), Patterns und Systeme von Patterns in Form von Aufsätzen vorstellen und diskutieren. Die Workshops finden unter Namen wie PLoP, EuroPLoP, ChiliPLoP, VikingPLoP, etc. in verschiedenen Teilen der Welt statt. Auch die Patterns der betrachteten Bücher wurden zum großen Teil auf solchen Workshops vorgestellt. Manche Veröffentlichungen sind recht umfangreich und werden zum Teil über Jahre hinweg erweitert. Diese Workshops bieten auch die Möglichkeit, Patterns neu zu strukturieren und darzustellen, wie es beispielsweise Buschmann und Henney in [BH02] getan haben.

4.2 Eine Auswahl von Patterns

4.2.1 Überblick und Begründung der Auswahl

Im Folgenden werden typische und bekannte Patterns vorgestellt, überwiegend aus den beiden Büchern „Pattern oriented Software Architecture“ (POSA1+2, [BMR+96, SSRB00]). In der getroffenen Auswahl befinden sich neben den Architektur–Patterns noch Design–Patterns, bei denen der Einfluss auf Systemstrukturen zum Teil deutlich ist. Anhand dieser Zusammenstellung soll die Kategorisierung in Architektur– und Design–Pattern und damit auch der Architekturbegriff nachvollzogen werden. Die Patterns dürften neben den Design–Patterns von Gamma et al. zu den bekanntesten gehören. Außerdem ist die Trennung in Architektur– und Design–Patterns hier am deutlichsten durchgeführt worden.

Die nachfolgende thematische Einteilung wurde durch den Autor vorgenommen und unterscheidet sich teilweise von derjenigen der Pattern–Autoren. Eine Diskussion der thematischen Kategorisierung findet in Abschnitt 4.3 statt. Die Themen sind:

- Verarbeitung von Daten
- Event–verarbeitende Systeme
- Erweiterbare und änderbare Systeme
- Client–Server–Kommunikation
- Netzwerk–Kommunikation und Verteilung
- Multitasking und asynchrone Dienste

Bei der folgenden Vorstellung der Themen werden die Patterns in tabellarischer Form skizziert, wobei die Kategorisierung in Architektur– und Design–Pattern aus der Literatur übernommen wurde. Danach wird jedes Pattern einheitlich mit einem kurzen Text und einem FMC–Diagramm beschrieben. Die Beschriftung der Diagramme erfolgt in englischer Sprache, da nahezu die gesamte Pattern–Literatur in Englisch verfasst ist und damit die Vergleichbarkeit mit den Darstellungen in der Literatur gewahrt bleibt. Die in der Literatur verwendete Darstellung von Architektur–Patterns ist Thema von Kapitel 5, in dem auch Original–Bilder abgebildet sind.

Verarbeitung von Daten (4.2.2)

In dieser Kategorie geht es um die Frage, wie eine Anwendung Daten verarbeitet. Dabei geht man davon aus, dass es viele verschiedene Algorithmen gibt, die auf die Daten angewendet werden sollen. Diese Algorithmen sollen in eigenen Modulen gekapselt und somit unabhängig voneinander und austauschbar sein. Man benötigt daher eine Architektur, die es einfach macht, Module mit Algorithmen zu integrieren.

Zwei Möglichkeiten werden vorgestellt: Entweder die Daten sind zentral vorhanden und eine Steuer–Komponente ruft nacheinander die Module auf, die auf den zentralen Daten operieren (BLACKBOARD), oder die Module verarbeiten Datenströme, so dass sie hintereinander gehängt werden können (PIPES AND FILTERS). Bei letzterem erspart man sich die Steuerung, da der Ablauf durch die Verkettung der Module vorgegeben ist. Interessanterweise kann man auch mit zentralen Daten eine PIPES AND FILTERS–Verarbeitung realisieren — es hängt letzten Endes ja nur an der Steuerung.

Pattern	**Problem mit Kontext**	**Lösung: Konzept, Struktur und Ablauf für Anwendung, Infrastruktur**
BLACKBOARD Architektur	**Kontext:** Verarbeitung von Daten **Problem:** Die Auswahl und Reihenfolge der Operationen ist von den Daten abhängig und muss evtl. bei jedem Schritt neu ermittelt werden.	**Konzept:** Koordinierte Operationen auf gemeinsamen Speicher **Struktur & Ablauf:** Mehrere Module zur Datenverarbeitung, die Operationen auf einem gemeinsamen Speicher durchführen, sowie ein Koordinator, der die Module aufruft und deren Ergebnisse bewertet. **Infrastruktur:** Gemeinsamer Speicher
PIPES AND FILTERS Architektur	**Kontext:** Verarbeitung von Datenströmen **Problem:** Änder– und erweiterbare Reihenfolge von Operationen	**Konzept:** Verkettung von Operationen zur Datenstromverarbeitung **Struktur & Ablauf:** Mit Pipes verkettete Filter zur Datenstromverarbeitung; Ablauf durch Topologie vorgegeben. **Infrastruktur:** Pipes zur Kopplung der Filter.

Event–verarbeitende Systeme (4.2.3)

Eine Anwendung (meist ein Service Provider, oder aber auch ein Client) muss auf Events reagieren. Bei PUBLISHER–SUBSCRIBER löst ein Client einen Event aus, wodurch der Publisher Aufträge an alle registrierten Event–Behandler (Subscriber) richtet. Dieses Prinzip setzt sich in den nächsten Patterns fort.

Die Quellen der Events können bei den Clients (REACTOR) und in von der Anwendung aufgerufenen Services (PROACTOR) liegen.

Die Patterns PROACTOR und LEADER / FOLLOWERS ermöglichen die nebenläufige Behandlung externer Events, wobei ersteres asynchron aufrufbare Basisdienste und letzteres Multitasking vom Betriebssystem erfordert.

Pattern	**Problem mit Kontext**	**Lösung: Konzept, Struktur und Ablauf für Anwendung, Infrastruktur**
PUBLISHER–SUBSCRIBER Design	**Kontext:** Eine offene Anzahl von Komponenten soll bei einem Event benachrichtigt werden. **Problem:** Wie umgeht man, dass ein Event–Auslöser alle Interessenten kennen muss, um sie zu benachrichtigen?	**Konzept:** Dienst zum indirekten Prozeduraufruf bzw. zur Broadcast–Kommunikation **Struktur & Ablauf:** Eine offene Anzahl von Event–Behandler–Prozeduren wird von einem Event–Auslöser über den Publisher angestoßen. **Infrastruktur:** Dispatcher mit Registrierung.
REACTOR Architektur	**Kontext:** Event–getriebene Anwendung (z.B. Service Provider) empfängt versch. Event–Typen von versch. Event–Quellen **Problem:** Demultiplex und Dispatch der Events an Event–Behandler ohne Multitasking	**Konzept:** Reaktive Anwendung mit zentralem Event–Dispatch **Struktur & Ablauf:** Service Provider besteht aus Event–Behandlern, die von einem zentralen Dispatcher (Reactor) bei Auftreten von Events aufgerufen werden. **Infrastruktur:** Dispatcher mit Registrierung.
PROACTOR Architektur	**Kontext:** Service Provider als reaktive Anwendung, Verfügbarkeit asynchroner Dienste (z.B. I/O) **Problem:** Wie können Aufträge nebenläufig bearbeitet werden, ohne Multitasking zu benutzen?	**Konzept:** Kooperatives Multitasking auf Anwendungsebene unter Nutzung asynchroner Dienstaufrufe, sowie Reactor für Auftragsrückmeldungen **Struktur & Ablauf:** Die Anwendung besteht aus Event–Behandlern für Rückmeldungen von Diensten, die diese zuvor aufgerufen haben. Nach Aufruf eines Dienstes erfolgt die Rückgabe an die Hauptschleife. **Infrastruktur:** Dispatcher mit Registrierung, evtl. Completion Queue
LEADER / FOLLOWERS Architektur	**Kontext:** Service Provider als reaktive Anwendung mit Multitasking **Problem:** Wie erreicht man hohe Performance, also kurze Antwortzeiten bei nebenläufigen Aufträgen?	**Konzept:** Effiziente Ressourcennutzung durch Task–Pool und Rollenwechsel. **Struktur & Ablauf:** Task Pool, Rollen Listener & Worker; Rollenwechsel der Tasks umgeht Performance–Verluste durch Kontextwechsel **Infrastruktur:** Worker Pool sowie ein Mutex oder ähnliches zur Auswahl des nächsten Listeners

Erweiterbare und änderbare Systeme (4.2.4)

Programmierte Systeme wachsen und verändern sich mit der Zeit. Das betrifft nicht nur die Zeit nach der Auslieferung, bereits beim Entwurf kommen häufig nachträgliche Anforderun-

gen. Es ist daher wichtig, bereits bei der Planung Möglichkeiten zur nachträglichen Änderung und Erweiterung vorzusehen.

Mit dem LAYERS Pattern wird vorgeschlagen, Prozeduren und Module einer Anwendung zu gruppieren und deren Aufrufbeziehungen einzuschränken, um sie später leichter austauschen zu können.

Beim REFLECTION Pattern geht man davon aus, dass sich manche Datenstrukturen und Schnittstellen mit der Zeit ändern. Um dann nicht alle Stellen im Code anfassen zu müssen, führt man Metainformationen über Datenstrukturen und Schnittstellen ein und führt vor jedem Zugriff oder Aufruf eine Abfrage dieser Metainformationen durch.

Um eine bestehende Anwendung nachträglich erweitern zu können, kann man den Weg über Plug–Ins wählen. Der Hauptablauf ist durch die Anwendung vorgegeben, an bestimmten Stellen (Hooks) können sich nun die so genannten *Interceptors* einklinken, die die Funktionalität des Plug–Ins in die Anwendung einbetten.

Damit eng verbunden ist der Wunsch, eine Anwendung um Funktionalität zu erweitern, ohne diese neu übersetzen oder gar neu starten zu müssen. Der COMPONENT CONFIGURATOR verwaltet das dafür erforderliche dynamische Laden von Code.

Eher für Basissysteme ist der MICROKERNEL gedacht. Die Idee hierbei ist, ähnlich wie bei INTERCEPTOR, Funktionalität in austauschbare Module auszulagern. Der Unterschied besteht darin, dass als Anwendungskern nur noch ein Kommunikationsdienst mit ein paar Basisdiensten übrigbleibt, nämlich der Microkernel.

Pattern	**Problem mit Kontext**	**Lösung: Konzept, Struktur und Ablauf für Anwendung, Infrastruktur**
LAYERS Architektur	**Kontext:** Dekomposition einer Anwendung. Die Aufgaben, die eine Anwendung erledigen muss, lassen sich gruppieren. **Problem:** Wie strukturiert man eine Anwendung, so dass stabile Schnittstellen entstehen und nachträgliche Änderungen lokal bleiben?	**Konzept:** Schichtung der Aufgaben bzw. Module, Restriktion der Aufrufbeziehungen. **Struktur & Ablauf:** — **Infrastruktur:** —
REFLECTION Architektur	**Kontext:** Datenstrukturen und Schnittstellen ändern sich mit der Zeit. **Problem:** Wie umgeht man dabei Code–Änderungen bzw. wie kommt man mit verschiedenen Versionen zurecht?	**Konzept:** Handhabung variabler Datenstrukturen und Interfaces über Metainformationen **Ablauf**: Vor jedem Zugriff oder Aufruf Abfrage und Auswertung von Metainformationen **Infrastruktur:** Auskunftsdienst für Meta–Informationen.
INTERCEPTOR Architektur	**Kontext:** Erweiterbare Anwendungen und Frameworks **Problem:** Wie kann Funktionalität transparent und nachträglich erweitert oder angepasst werden?	**Konzept:** Event–Behandler zur Anwendungserweiterung, durch Hooks des Application Core angestoßen. **Struktur & Ablauf:** Aufteilung in Application Core mit Hooks und Plug–Ins mit Interceptors, also Event–Behandler für Hook–Events **Infrastruktur:** Dispatcher mit Registrierung.

Pattern	Problem mit Kontext	Lösung: Konzept, Struktur und Ablauf für Anwendung, Infrastruktur
COMPONENT CONFIGURATOR Design	**Kontext:** Erweiterbare Anwendungen und Frameworks **Problem:** Wie kann die Funktionalität einer Anwendung zur Laufzeit flexibel erweitert oder angepasst werden?	**Konzept:** Nachladen von Code. **Struktur & Ablauf:** Funktionalität ist in nachladbare Code–Einheiten gepackt, die dynamisch geladen werden **Infrastruktur:** Lader mit Component Registry.
MICROKERNEL Architektur	**Kontext:** Flexible Basis–Systeme **Problem:** Wie strukturiert man ein Basis–System, das flexibel änder– und erweiterbar sein soll und mehrere APIs für seine Dienste zulässt?	**Konzept:** Minimale Kerndienste, Funktionalität in modularen Diensten mit einheitlichen Schnittstellen, die einen zentralen Kommunikationsdienst nutzen **Struktur & Ablauf:** Modulare Dienste (Internal & External Servers), die über Microkernel kommunizieren. External Servers stellen APIs für externe Aufrufe zur Verfügung. **Infrastruktur:** Microkernel mit Basisdiensten zur Kommunikation

Client–Server–Kommunikation (4.2.5)

In dieser Kategorie geht es darum, wie ein Client Zugang zum Service Provider bekommt. Er kann die Kommunikation über einen PROXY durchführen, so dass es ihm egal ist, wie dieser mit dem Server kommuniziert, oder er kann einen DISPATCHER um Vermittlung bitten, wobei er nach Vermittlung direkt mit dem Server kommuniziert.

Pattern	Problem mit Kontext	Lösung: Konzept, Struktur und Ablauf für Anwendung, Infrastruktur
PROXY Design	**Kontext:** Client–Server–Kommunikation **Problem:** Wie kontrolliert man die Kommunikation zwischen Client und Server bzw. vereinfacht sie aus Sicht des Clients?	**Konzept:** Kommunikation über Stellvertreter (Proxy) **Struktur & Ablauf:** Proxy zwischen Client und Server gibt Aufträge und Rückmeldungen weiter. **Infrastruktur:** (Proxy)
CLIENT–DISPATCHER–SERVER Design	**Kontext:** Client–Server–Kommunikation, veränderlicher Ort oder Identität eines Servers **Problem:** Wie kann ein Client direkt (also nicht über einen Proxy) mit einem Server kommunizieren, ohne dessen Adresse oder Identität vorher zu kennen?	**Konzept:** Kommunikation über Vermittler (der nur Verbindung herstellt) **Struktur & Ablauf:** Client beauftragt Dispatcher mit Vermittlung zum Server. Nach Vermittlung erfolgt direkte Kommunikation zwischen Client und Server **Infrastruktur:** Dispatcher mit Service Registry.

Netzwerk–Kommunikation und Verteilung (4.2.6)

Die Kommunikation über ein Netzwerk erfordert Aufrufe, die man gerne kapseln will. FORWARDER–RECEIVER bietet eine nachrichtenorientierte Kommunikation, ACCEPTOR–

CONNECTOR kapselt den Verbindungsaufbau bei einer verbindungsorientierten Kommunikation.

Der BROKER kombiniert nun PROXY, DISPATCHER (naming service) und FORWARDER–RECEIVER, um eine Kommunikations–Infrastruktur für Clients und Service Provider in einem potentiell verteilten System anzubieten.

Pattern	**Problem mit Kontext**	**Lösung: Konzept, Struktur und Ablauf für Anwendung, Infrastruktur**
FORWARDER–RECEIVER Design	**Kontext:** Paketorientierte Peer–to–Peer–Kommunikation. **Problem:** Wie verbirgt man die Wahl des Transportprotokolls vor der Anwendung?	**Konzept:** Allgemeiner Dienst für paketorientierte Kommunikation **Struktur & Ablauf:** Kommunikationspartner nutzen Messaging Service **Infrastruktur:** Messaging Service.
ACCEPTOR–CONNECTOR Design	**Kontext:** Verbindungsorientierte Peer–to–Peer–Kommunikation. **Problem:** Wie verbirgt man die Wahl des Transportprotokolls vor der Anwendung?	**Konzept:** Allgemeiner Dienst für verbindungsorientierte Kommunikation **Struktur & Ablauf:** Kommunikationspartner nutzen Dienst (Acceptor / Connector) zum Aufbau einer Verbindung bzw. Warten auf Verbindungsanfrage **Infrastruktur:** Verbindungsdienst: Connector (Client) und Acceptor (Server)
BROKER Architektur	**Kontext**: Verteilte Anwendung in heterogenem Umfeld **Problem:** Wie ermöglicht man eine ortstransparente Kommunikation der Anwendungskomponenten?	**Konzept:** Dienst zur netzwerktransparenten Client–Server–Kommunikation **Struktur & Ablauf:** Komponenten (Client und Server) nutzen den BROKER (Kommunikationsdienst und Dispatcher) über Proxies **Infrastruktur:** Broker mit Proxies, Dispatcher und Bridges

Multitasking und asynchrone Dienste (4.2.7)

Die Benutzung mehrerer Threads innerhalb eines Systems oder in Verteilung impliziert asynchrone Kommunikation: Es werden Aufträge abgesetzt, die von den Service Providern nebenläufig zum Auftraggeber bearbeitet werden und deren Ergebnis nach einer "Ende"–Meldung später abgeholt werden kann.

Benutzt man Multitasking (mit Prozessen oder Threads), muss die Anwendung nicht mehr einen eigenen Event–Loop mitbringen, sondern kann blockierende Aufrufe benutzen. HALF–SYNC / HALF–ASYNC beschreibt, wie blockierende Aufrufe für asynchrone Dienste ermöglicht werden.

ACTIVE OBJECT beschreibt Auftrags–Rückmelde–Mechanismen zwischen verschiedenen Threads. MONITOR OBJECT wiederum zeigt einen Sperrmechanismus, um den Zugriff von Threads auf Ressourcen zu serialisieren.

Bei ASYNCHRONOUS COMPLETION TOKEN will man die Antworten zu asynchron abgesetzten Aufträgen diesen wieder zuordnen können.

Nach Buschmann und Henney muss die Entscheidung für eine Multitasking–Architektur früh gefällt werden [BH02, S. 37].

Pattern	Problem mit Kontext	Lösung: Konzept, Struktur und Ablauf für Anwendung, Infrastruktur
HALF–SYNC/HALF–ASYNC Architektur	**Kontext:** Asynchrones I/O und Multitasking **Problem:** Wie kann man Multitasking benutzen, ohne auf die Vorteile asynchroner I/O zu verzichten?	**Konzept:** Trennung der Anwendung in asynchrone und synchrone Dienste, die über Queues gekoppelt werden **Struktur & Ablauf:** Synchrone Dienste: Mehrere Threads, die blockierende (synchrone) Aufrufe absetzen. Asynchrone Dienste: Ein Thread mit Hauptschleife. Kommunikation der Dienste über Queues **Infrastruktur:** Queues zur Kopplung der asynchronen mit den synchronen Diensten
ASYNCHRONOUS COMPLETION TOKEN Design	**Kontext:** Nutzung asynchroner Dienste **Problem:** Wie können eintreffende Rückmeldungen asynchroner Dienste den abgesetzten Aufträgen zugeordnet werden?	**Konzept:** Zuordnung von Auftragsrückmeldungen zu Aufträgen über Token. **Ablauf:** Mit Aufruf ein Token verschicken und vermerken, bei Rückmeldung über Token die Zuordnung herstellen. **Infrastruktur:** Registry zur Zuordnung von Rückmeldungen zu Aufrufen.
ACTIVE OBJECT Design	**Kontext:** Nebenläufigkeit mit Multithreading, Objekte als Service Provider **Problem:** Die Aufrufe verschiedener Threads bezüglich eines Objekts müssen synchronisiert werden, sollen den Aufrufer aber nicht blockieren.	**Konzept:** Nutzung von Multithreading zur Ausführung von Prozeduren in einem anderen Thread, Inter–Thread–Kommunikation **Ablauf:** Bei Aufruf wird ein Objekt (Future) für das Ergebnis angelegt und ein Eintrag in eine Queue (Activation List) geschrieben, die von einem Scheduler im objekteigenen Thread ausgelesen wird. **Infrastruktur:** Objektbezogener Scheduler und Queues (Activation Lists) zur Inter–Thread–Kommunikation
MONITOR OBJECT Design	**Kontext:** Objektorientierung und Multithreading **Problem:** Wie verhindert man, dass mehrere Threads gleichzeitig ein Objekt manipulieren?	**Konzept:** Synchronisation mehrerer Threads bezüglich Zugriff auf eine Ressource (Objekt) über Sperren. **Ablauf:** Vor Zugriff auf ein Objekt muss eine Sperre (Monitor) angefordert werden, nach Zugriff wird sie zurückgegeben **Infrastruktur:** Verwaltung objektbezogener Sperren

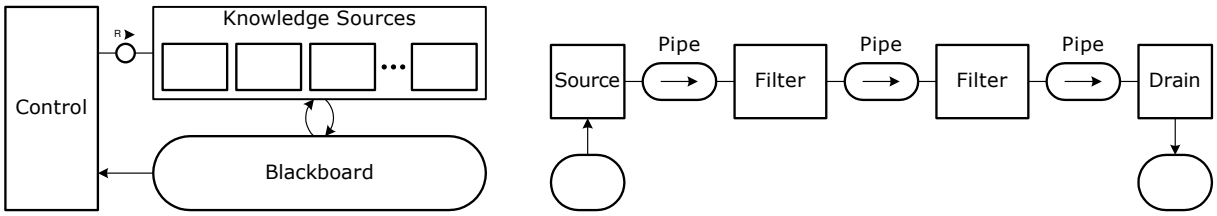
Manche der aufgeführten Patterns wurden bereits in einem vom Autor geleiteten Seminar von Studenten des Hasso–Plattner–Instituts untersucht und verdichtet. Ihre Ausarbeitungen sind in einem technischen Bericht [GK03] zusammengefasst. Da es für diese Arbeit aber hauptsächlich auf die Idee hinter den einzelnen Patterns ankommt, wurde im folgenden eine deutlich kompaktere Darstellung gewählt.

4.2.2 Verarbeitung von Daten

4.2.2.1 Blackboard

(Architektur–Pattern im Kapitel „From Mud to Structure“ in [BMR+96, S. 71])

Es sollen Daten verarbeitet werden, wobei die erforderlichen Operationen nicht vorher feststehen, sondern abhängig von den Daten und den Ergebnissen vorheriger Operationen ermittelt werden.



(a) BLACKBOARD (b) PIPES AND FILTERS

Abbildung 4.1: Patterns zur Verarbeitung von Daten: BLACKBOARD und PIPES AND FILTERS

Dazu operieren mehrere datenverarbeitende Module (im Pattern „Knowledge Sources“ genannt) auf einem gemeinsamen Datenspeicher („Blackboard“), siehe Abbildung 4.1(a). Immer genau ein Modul wird von der Steuerung („Control“) aktiviert, je nach aktuellem Bearbeitungszustand der Daten im Blackboard. Das Pattern legt die Schnittstelle zwischen Knowledge Sources und Control fest und bestimmt, dass nur die Knowledge Sources Operationen auf den Daten durchführen.

Das Pattern kommt aus dem Bereich der künstlichen Intelligenz (KI) zum Lösen von Problemen, wobei der Lösungsweg nicht vorher feststeht, sondern durch Probieren von Algorithmen und Bewertung ihrer Ergebnisse entsteht. Man kann es aber auch anders interpretieren: Die Knowledge Sources führen Operationen durch, die durch ein Steuerwerk (Control) gesteuert werden, und das Blackboard enthält die Daten, auf denen operiert wird. Damit ergibt sich der in Software realisierte Steuerkreis nach Wendt [Wen70][Wen91, S. 299].

4.2.2.2 Pipes and Filters

(Architektur–Pattern im Kapiteln „From Mud to Structure“ [BMR+96, S. 53])

Es sollen Operationen auf Datenströmen durchgeführt werden, wobei deren Reihenfolge und Austauschbarkeit möglichst flexibel sein sollte.

Bei PIPES AND FILTERS findet die Datenverarbeitung in Modulen einheitlicher Schnittstelle statt, den Filtern. Diese werden über Pipes verbunden und bilden damit eine Filterkette, in der verschiedene Operationen auf einem Datenstrom durchgeführt werden können, siehe Abbildung 4.1(b). Die Reihenfolge der Operationen ergibt sich durch die Topologie der Filterkette, deren Aufbau aber nicht Thema des Patterns ist.

Bei einer „push“–Kette werden die Filter beim Eintreffen der Daten aktiviert, also von der Datenquelle zur Senke. Bei einer „pull“–Kette erfolgt der Aufruf von der Senke her, die die Daten anfordert. Die Anforderung wird dabei von Filter zu Filter bis zur Quelle weitergegeben, die schließlich Daten liefern kann, welche dann durch die Filterkette „gezogen“ werden.

Aktive Filter sind Komponenten, die Daten von ihrer Quelle anfordern („pull“) und sie an ihre Senke weitergeben („push“). Eine Verkettung aktiver und damit potentiell nebenläufig arbeitender Filter–Module setzt wiederum eine Inter–Task–Kommunikation durch puffernde Pipes voraus.

4.2.3 Event–verarbeitende Systeme

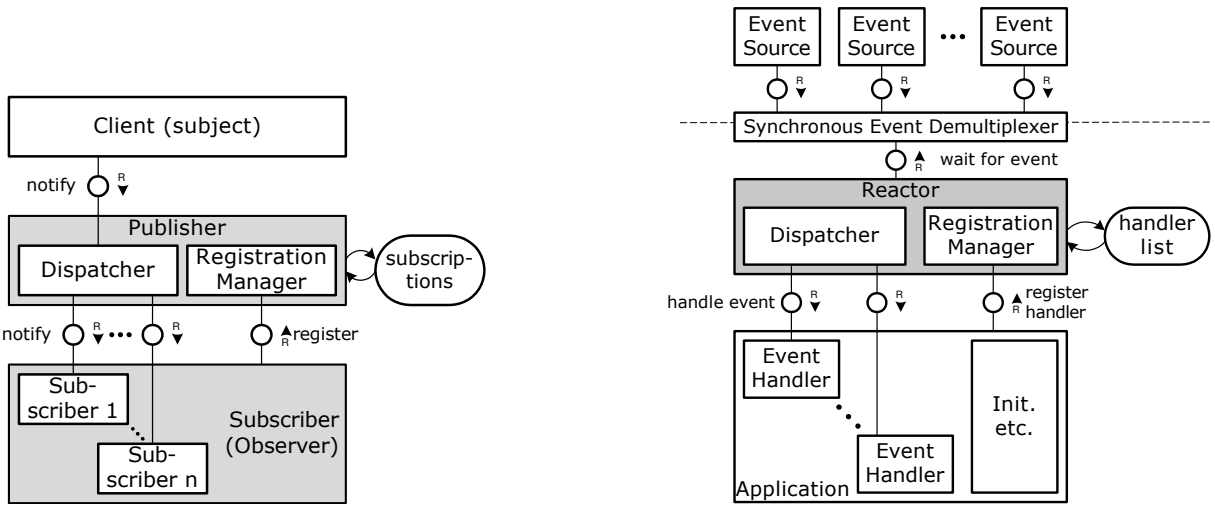
4.2.3.1 Publisher–Subscriber

(Design–Pattern im Kapitel „Communication“ [BMR+96, S. 339], basierend auf OBSERVER [GHJV94, S. 293])

Ein Client möchte eine offene Anzahl an Interessenten benachrichtigen können, ohne diese kennen zu müssen.

Er benutzt dazu einen Publisher, um Mitteilungen an Abonnenten (Subscriber) zu senden (notify), siehe Abbildung 4.2(a). Der Publisher realisiert damit einen *Broadcast*–Kanal zur Kommunikation mit den Subscribern. Bei dieser Art der Kommunikation ist dem Client verborgen, welche und wie viele Empfänger seine Nachricht haben wird.

Ein typisches Einsatzgebiet sind Systeme, deren Akteure Events auslösen und auf solche reagieren.



(a) PUBLISHER–SUBSCRIBER (b) REACTOR

Abbildung 4.2: Patterns für Event–verarbeitende Systeme: PUBLISHER–SUBSCRIBER und REACTOR

4.2.3.2 Reactor

(Architektur–Pattern im Kapitel „Event Handling“ [SSRB00, S. 179])
Eine Anwendung soll auf verschiedene Arten nebenläufig eintreffender Events reagieren, indem passende Behandler–Prozeduren aufgerufen werden.

Mit Hilfe des REACTORS verarbeitet eine Anwendung von außen kommende Events. Der Reactor führt den Dispatch der Events an den oder die dafür registrierten Event Handler durch, wie in Abbildung 4.2(b) gezeigt. Es handelt sich dabei also um einen Publisher–Subscriber–Mechanismus, der um einen Demultiplexer, wie etwa der `select()`–Aufruf[1] zum blockierenden Warten auf Events, erweitert wurde.

Für eine darauf basierende Anwendung ergibt sich das klassische Event–getriebene System, in dem Reactor und Application einen gemeinsamen Thread benutzen. Die Registrierung der Event Handler kann sogar lediglich einmalig zur Initialisierung erfolgen, muss also nicht dynamisch sein.

4.2.3.3 Proactor

(Architektur–Pattern im Kapitel „Event Handling“ [SSRB00, S. 215])

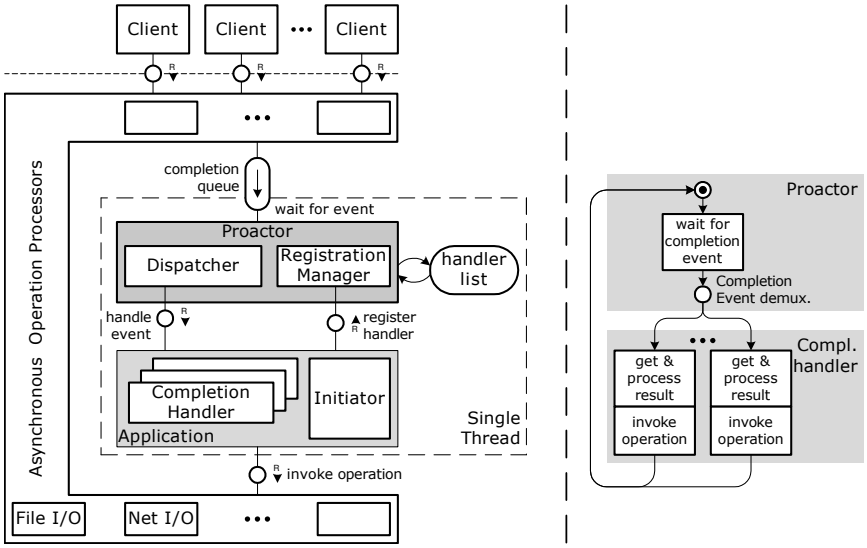


Abbildung 4.3: Patterns für Event–verarbeitende Systeme: PROACTOR

Eine reaktive Anwendung (z.B. ein Service Provider) soll nebenläufig Aufträge bearbeiten, ohne dafür Multitasking des Betriebssystems zu verwenden. Es stehen allerdings asynchron aufrufbare Dienste zur Verfügung, die zum Abschluss einer Operation eine Rückmeldung in eine Queue stellen.

Nachdem die Completion Handler registriert wurden, setzt der Initiator den ersten Auftrag ab. Danach verteilt der Proactor genauso wie beim REACTOR Events auf die Completion Handler,

[1] Mit dem Betriebssystem–Aufruf `select()` kann ein Prozess oder Thread eine Menge von Datei–Deskriptoren auf Status–Änderung überwachen. Mit den Dateideskriptoren können auch Netzwerk–Sockets oder Pipes zu anderen Prozessen oder Threads verbunden sein. Ein optionaler Timeout–Parameter kann die Wartezeit begrenzen.

die ihrerseits wieder Dienste asynchron aufrufen, siehe Abbildung 4.3. Die Completion Queue erfasst nicht nur Completion Events, sondern auch von den Clients ausgelöste Events wie beispielsweise Aufträge.

Der Proactor führt einen Event Loop aus und hat damit die Rolle eines Schedulers für kooperatives Multitasking. Eine zufrieden stellende Quasi–Nebenläufigkeit wird aber nur dann erreicht, wenn die Completion Handler nur kurze Verarbeitungsschritte durchführen, während die Langläufer von asynchron aufgerufenen Diensten durchgeführt werden. Komplexere Anwendungen werden hier, wie bei allen Event–getriebenen Anwendungen, leicht unübersichtlich.

4.2.3.4 Leader / Followers

(Architektur–Pattern im Kapitel „Concurrency“[SSRB00, S. 447])

Ein Service Provider soll unter Ausnutzung von Multitasking möglichst effizient nebenläufig Requests von Clients bearbeiten können.

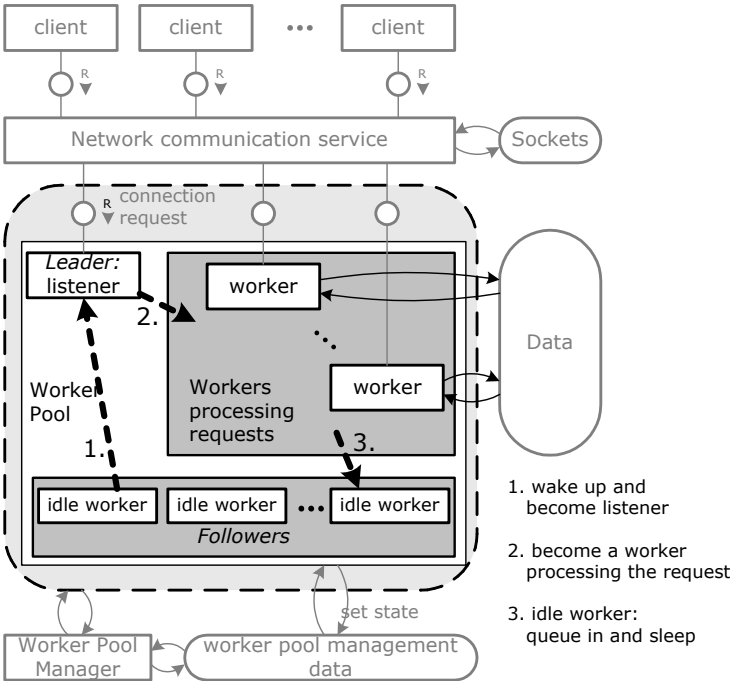


Abbildung 4.4: Patterns für Event–verarbeitende Systeme: LEADER / FOLLOWERS

Die Grundlage des LEADER / FOLLOWERS Patterns ist die Trennung der Aufgaben in Listener und Worker, wie z.B. in LISTENER / WORKER in Abschnitt 7.2.1 beschrieben. Weiterhin sollen Worker Tasks nicht erst bei Bedarf erzeugt werden, sondern bereits vorher im Zustand *idle* zur Verfügung stehen (WORKER POOL). Anstatt nun, wie in JOB QUEUE, zwischen Listener und Idle Worker eine Queue zur Übergabe des nächsten Jobs zu benutzen, wechseln die Tasks ihre Rollen, so dass jeder mal zum Listener wird und bei Eintreffen eines Requests die Rolle wechselt und zum Worker wird und damit den Job “mitnimmt”. Die Rolle des Listeners wird vom nächsten idle Worker übernommen. Abbildung 4.4 zeigt den Aufbau, in den die Zustandsübergänge der Worker integriert wurden. Man erkennt darin auch eine Warteschlange

aus Idle Workern. Damit immer nur einer der Wartenden zum Listener werden kann, ist ein Sperr–Mechanismus erforderlich.

In Kapitel 7 wird gezeigt, wie sich die Elemente des LEADER / FOLLOWERS Patterns, so wie es in [SSRB00, S. 447] vorgestellt wird, in mehrere, grundlegende Patterns unterteilen lassen, was damit auch die Alternativen deutlicher hervorhebt.

4.2.4 Erweiterbare und änderbare Systeme

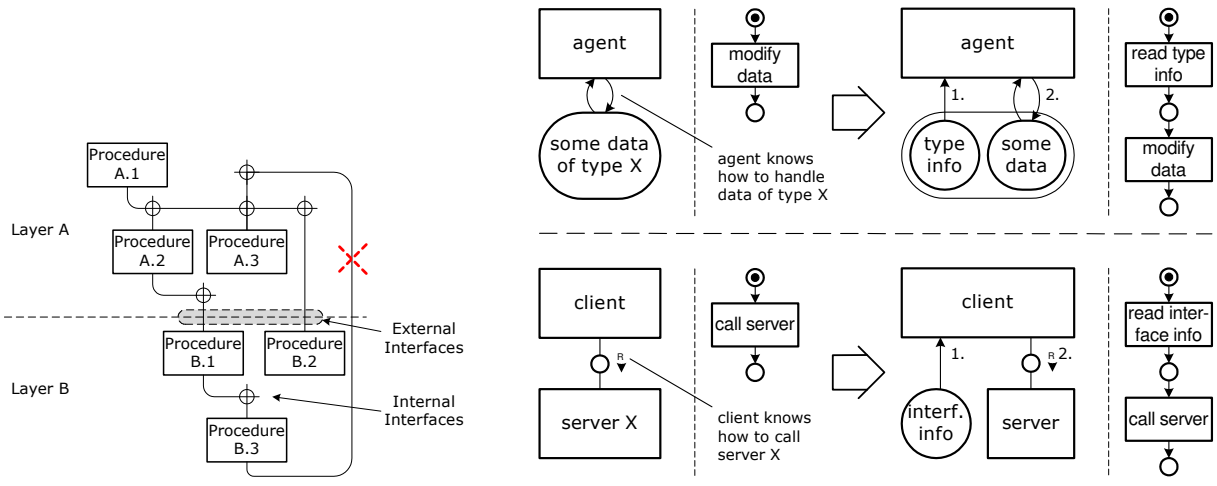
4.2.4.1 Layers

(Architektur–Pattern im Kapitel „From Mud to Structure“ [BMR+96, S. 31])

Die Dekomposition einer größeren Anwendung in Module soll zu möglichst überschaubaren Abhängigkeiten zwischen den Modulen und zu einer guten Änder– und Erweiterbarkeit führen.

Hinter LAYERS steht das Konzept der Kapselung. Layers legen die Gruppierung der Module und ihre Aufrufbeziehungen fest: Module und ihre Prozeduren werden in Layers gruppiert. Aus einer Prozedur eines Layers dürfen nur Prozeduren des gleichen oder des nächst–niedrigeren Layers aufgerufen werden. Für den Aufruf über eine Layer–Grenze hinweg sind auch nur bestimmte Prozeduren freigegeben (externe Schnittstelle). Abbildung 4.5(a) zeigt eine beispielhafte Aufrufschichtung, wobei die Aufrufbeziehung ganz rechts (von B.3 nach A.3) nicht erlaubt ist.

LAYERS ist eher ein Stil als ein Pattern (siehe Abschnitt 3.4.1.2).



(a) LAYERS (b) REFLECTION

Abbildung 4.5: Patterns für erweiterbare und änderbare Systeme: LAYERS und REFLECTION

4.2.4.2 Reflection

(Architektur–Pattern im Kapitel „Adaptable Systems“ [BMR+96, S. 193])
Von Version zu Version können sich in einer Anwendung Datentypen und Schnittstellen ändern. Um dann nicht alle Programmteile neu übersetzen zu müssen, legt man Metainformationen über Datentypen und Schnittstellen an, die die Anwendung zur Laufzeit abfragen muss, bevor ein Zugriff oder ein Aufruf erfolgt (Meta–Object Protokoll).

In Abbildung 4.5(b) ist links oben beispielhaft ein Akteur (agent) dargestellt, der modifizierend auf Daten in einem Speicher zugreift. Üblicherweise kennt der Akteur die Struktur der Daten, da sie im Fall von Software etwa durch den Compiler in den Zugriffsprozeduren berücksichtigt wurde. Beim REFLECTION Pattern kann man dieses Wissen nicht voraussetzen und sieht zusätzliche Informationen über die Daten im Speicher vor, üblicherweise Typinformationen (Metadaten). Im rechten Teil der Abbildung ist daher ein weiterer Speicher mit Typinformationen dazugekommen, den der Akteur immer zuerst lesen muss, bevor er weiß, wie er die Daten dieses Typs modifizieren kann. Diese Erweiterung muss an allen Stellen in der Systemstruktur vorgenommen werden, an denen auf derartige Daten zugegriffen werden muss.

Das REFLECTION–Pattern erlaubt neben den Meta–Informationen über die Daten auch solche über Schnittstellen. Aus diesem Grund kann man es auch in die Kategorie Client–Server–Kommunikation einordnen. Die Plug–In–Infos bei INTERCEPTOR oder Informationen aus dem Component Repository des COMPONENT CONFIGURATOR sind dafür Beispiele.

4.2.4.3 Interceptor

(Architektur–Pattern im Kapitel „Service Access and Configuration“ [SSRB00, S. 109])

Eine Anwendung soll nachträglich erweitert bzw. eigene Funktionalität einfach in ein Framework integriert werden können.

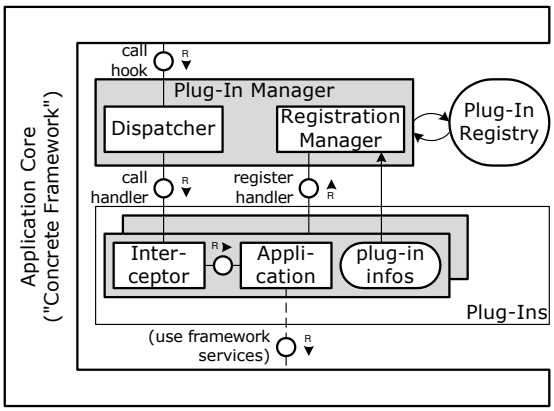


Abbildung 4.6: Patterns für erweiterbare und änderbare Systeme: INTERCEPTOR

In der Anwendung bzw. im Framework (Application Core) werden Hooks definiert, also Stellen im Ablauf, zu denen dafür registrierte Plug–In–Prozeduren, falls vorhanden, aufgerufen werden. Die Erweiterungen bzw. die eigentliche Anwendungsfunktionalität werden in Form von Plug–Ins integriert. Ein REACTOR verteilt die Aufrufe des Application Core an die Event

Handler der Plug–Ins, die *Interceptors*, die weitere Aufrufe innerhalb des Plug–Ins durchführen können, siehe Abbildung 4.6. Die Komponente eines Plug–Ins, die die eigentliche Funktionalität zur Verfügung stellt, wird im Pattern *Application* genannt.

Die Abbildung stellt abweichend von [SSRB00, S. 109] noch dar, dass ein Plug–In Informationen über sich zur Verfügung stellt (Plug–in Infos) und dass es wiederum Aufrufe an den Application Core richten kann. Diese Erweiterungen sind so typisch für Plug–Ins, dass sie hier aufgenommen wurden.[2]

Um Plug–Ins dynamisch einer Anwendung hinzufügen zu können, wird ein Lader für Code benötigt — siehe COMPONENT CONFIGURATOR weiter unten.

4.2.4.4 Component Configurator

(Design–Pattern im Kapitel „Service Access and Configuration“ [SSRB00, S. 75])

Ermöglicht man die Erweiterung einer Anwendung durch dynamisches Nachladen von Code, muss man auch für eine Buchführung über diese Codestücke sorgen und diese der Anwendung zur Verfügung stellen.

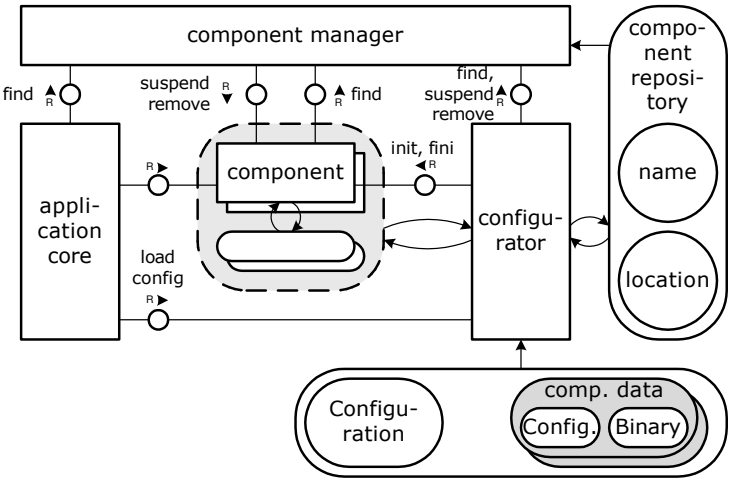


Abbildung 4.7: Patterns für erweiterbare und änderbare Systeme: COMPONENT CONFIGURATOR

Das Pattern COMPONENT CONFIGURATOR legt die Schnittstellen fest, die zum Laden und Entfernen einer Komponente, üblicherweise eines Plug–Ins, benutzt werden sollen. Weiterhin soll ein Configurator ein Repository mit Informationen über geladene Komponenten verwalten, siehe auch Abbildung 4.7.

Legt also INTERCEPTOR fest, dass Funktionalität mit Plug–Ins erweitert werden kann, zeigt der COMPONENT CONFIGURATOR, wie diese zu laden und verwalten sind.[3]

[2] Vergleiche dazu auch die Untersuchung der Plug–In–Architektur des Apache HTTP Servers in [GKKS04].
[3] Eine ausführliche Untersuchung des Patterns im Rahmen eines Seminars ist in [RG03] zu finden.

4.2.4.5 Microkernel

(Architektur–Pattern im Kapitel „Adaptable Systems“ [BMR+96, S. 171])
Es sollen (Basis–)Dienste auf flexible Weise zur Verfügung gestellt werden, wobei mehrere Schnittstellen zur Benutzung der Dienste unterstützt werden. Die Dienste sollen einfach austauschbar und erweiterbar sein.

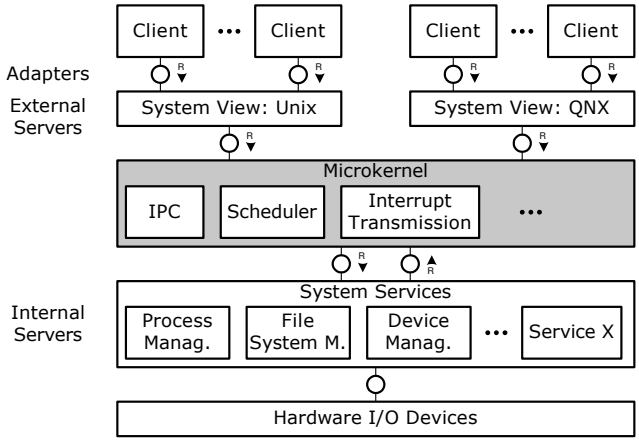


Abbildung 4.8: Patterns für erweiterbare und änderbare Systeme: MICROKERNEL

Der Microkernel in Abbildung 4.8 stellt nur elementare Dienste zur Verfügung und dient ansonsten zur Kommunikation. Die eigentliche Funktionalität liegt in den so genannten Internal Servers. Die External Servers und Adapter (APIs) stellen den Clients, also Anwendungsprogrammen, wählbare Sichten auf das System zur Verfügung.

In [BH02] wird dieses Pattern nicht auf die Betriebssystemebene eingeschränkt: Hier geht es um eine Anwendung, die Dienste zur Verfügung stellt, wobei verschiedene Clients aus Evolutionsgründen verschiedene Versionen dieser Dienste ansprechen. Die External Server stellen also verschiedene Versionen der Schnittstellen zur Verfügung. Der Microkernel stellt die Dienst–Basis zur Verfügung, während die Internal Server Plug–Ins für versionsspezifische Dienste darstellen (Vergleiche INTERCEPTOR). Diese zweite Interpretation des MICROKERNEL–Patterns unterscheidet sich damit deutlich von der ursprünglichen.

4.2.5 Client–Server–Kommunikation

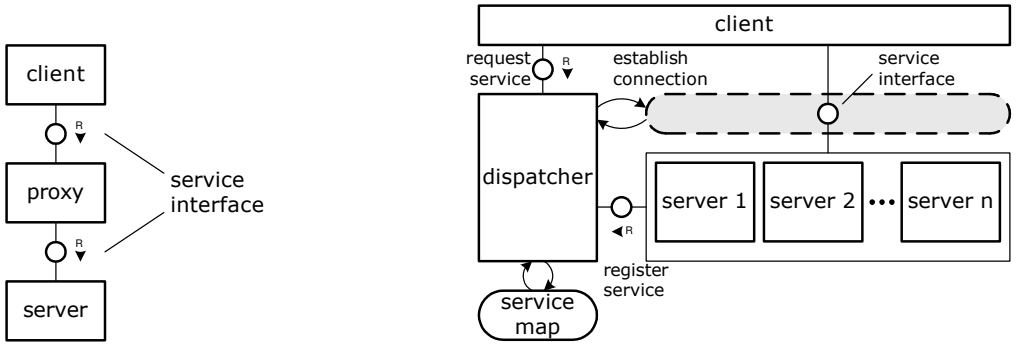
4.2.5.1 Proxy

(Design–Pattern im Kapitel „Access Control“ [BMR+96, S. 263])

Ein Client soll keinen direkten Zugang zum Service Provider bekommen, beispielsweise wegen der Durchsetzung von Zugangsbeschränkungen oder um den Ort des Service Providers variabel zu halten.

Ein Client benutzt zur Kommunikation mit einem Service Provider einen Stellvertreter (Proxy), der ihm die gleiche Schnittstelle (Service Interface) zur Verfügung stellt, siehe Abbildung 4.9(a).

Aus Sicht des Clients gibt es also keinen Unterschied. Die Benutzung eines Proxys bringt viele Möglichkeiten, zum Beispiel Zugangskontrolle, Pufferung, Verteilung oder bedarfsgetriebenes Laden. Wie der Client an den Proxy kommt, gehört nicht zum Pattern.



(a) PROXY (b) CLIENT–DISPATCHER–SERVER

Abbildung 4.9: Patterns für die Client–Server–Kommunikation: PROXY und CLIENT–DISPATCHER–SERVER

4.2.5.2 Client–Dispatcher–Server

(Design–Pattern im Kapitel „Communication“ [BMR+96, S. 323])

Es soll die Möglichkeit geschaffen werden, dass Aufträge von unterschiedlichen Dienst–Anbietern ausgeführt werden können, ohne dass der Client diese kennen muss.

Dafür wendet sich der Client an einen Dispatcher, der ihn an einen Service Provider vermittelt. Mit diesem kommuniziert der Client, bis sein Auftrag erledigt ist, siehe Abbildung 4.9(b).

Im Gegensatz zu PROXY stellt der Vermittler nur den Kontakt zwischen Client und Server her und ist danach nicht mehr beteiligt.

4.2.6 Netzwerk–Kommunikation und Verteilung

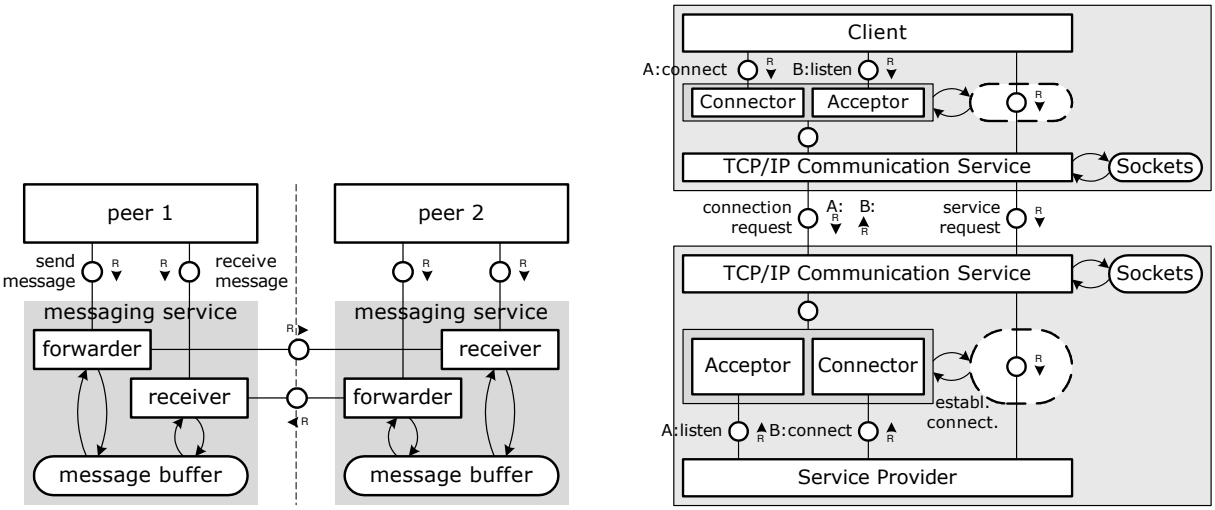
4.2.6.1 Forwarder–Receiver

(Design–Pattern im Kapitel „Communication“ [BMR+96, S. 307])

Paketorientierte Netzwerk–Zugriffe sollen für die Anwendung gekapselt werden.

Statt direkt miteinander zu kommunizieren, benutzen zwei Peers (gleichberechtigte Kommunikationspartner) dafür einen Messaging Service, nämlich je einen Forwarder für die ausgehenden und einen Receiver für die eingehenden Nachrichten, wie in Abbildung 4.10(a) dargestellt. Die Messaging Services kapseln die eigentliche Kommunikations–Infrastruktur und deren Protokolle.

Im Gegensatz zu PROXY wird hier aber die Schnittstelle des Partners nicht repliziert — vielmehr handelt es sich um eine generische paketorientierte Kommunikationsschnittstelle.



(a) FORWARDER–RECEIVER (b) ACCEPTOR–CONNECTOR

Abbildung 4.10: Patterns zu Netzwerk–Kommunikation und Verteilung: FORWARDER–RECEIVER und ACCEPTOR–CONNECTOR

4.2.6.2 Acceptor–Connector

(Design–Pattern im Kapitel „Event Handling“ [SSRB00, S. 285])

Der Aufbau einer Netzwerkverbindung zwischen Client und Server soll von der restlichen verbindungsorientierten Kommunikation getrennt werden.

Dazu dient auf der einen Seite der Connector, der eine Verbindungsanfrage stellt, auf der anderen der Acceptor, der auf Verbindungsanfragen wartet. Nach erfolgreicher Operation von Connector und Acceptor besteht eine Verbindung zwischen Client und Server, die diese für ihre Kommunikation nutzen können.

Es gibt also zwei Arten von Requests: Connection Requests zum Aufbau einer Verbindung, und Service Requests vom Client zum Server über diese Verbindung. Wer die Initiative zum Verbindungsaufbau ergreift, ist offen. Abbildung 4.10(b) zeigt beide Möglichkeiten (A: Client, B: Server baut Verbindung auf).

4.2.6.3 Broker

(Architektur–Pattern im Kapitel „Distributed Systems“ [BMR+96, S. 99])

Services sollen netzwerktransparent zur Verfügung gestellt werden.

Dafür sorgt ein Broker, der einerseits wie ein Dispatcher den eigentlichen Service Provider ermittelt. Die Kommunikation erfolgt über Proxies und den Broker, so dass es für einen Client egal ist, ob es sich um lokale Aufrufe handelt oder dafür ein Netzwerkdienst benutzt werden muss.[4] Siehe dazu Abbildung 4.11.

[4] Im Rahmen eines Seminars fand auch ein Vergleich des Broker–Patterns mit CORBA, DCOM und EJB statt [HL03].

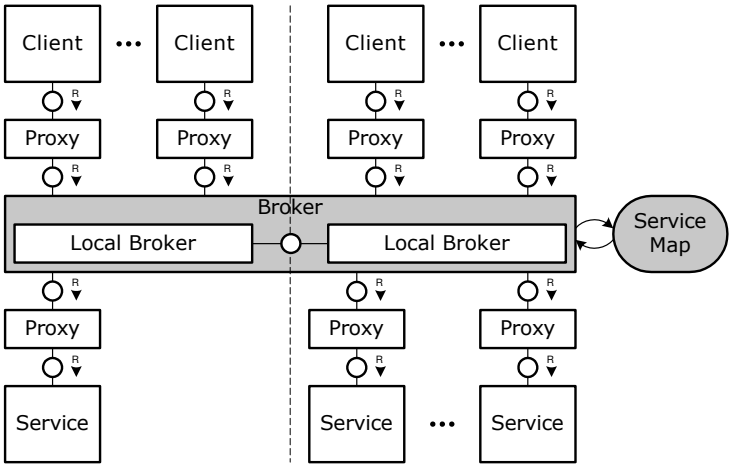


Abbildung 4.11: Patterns zu Netzwerk–Kommunikation und Verteilung: BROKER

4.2.7 Multitasking und Asynchrone Dienste

4.2.7.1 Half–Sync / Half–Async

(Architektur–Pattern im Kapitel „Concurrency“ [SSRB00, S. 423])

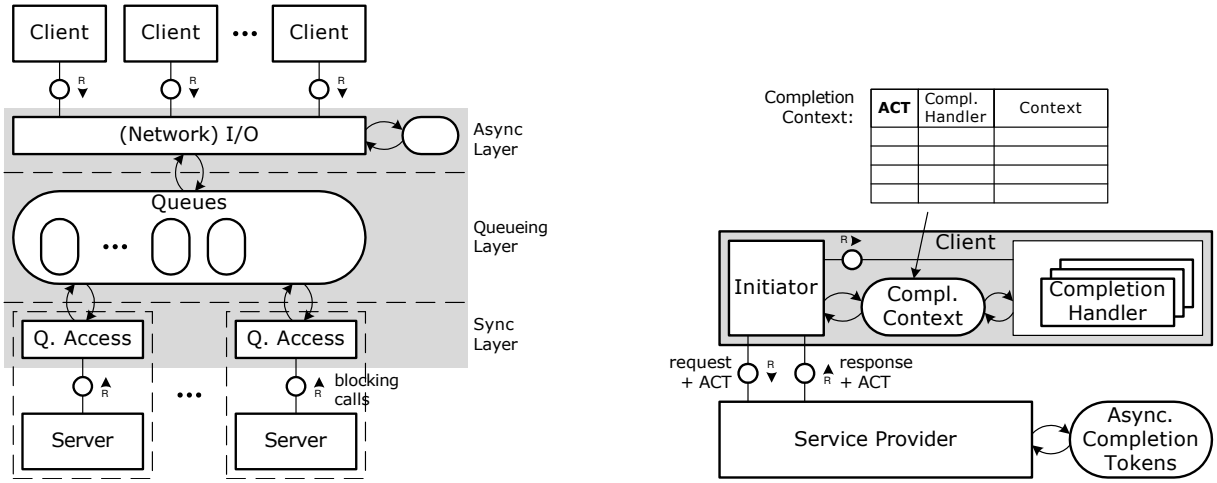
Die Benutzung von Multitasking ermöglicht es, Anwendungen einfacher zu schreiben, da Event—Loops und Scheduling vom Betriebssystem übernommen werden. Die Benutzung asynchroner Dienste hat aber zur Folge, dass man sich dennoch um die abgesetzten Aufträge und um Polling zu kümmern hat.

Das HALF–SYNC / HALF–ASYNC Pattern behandelt die Benutzung asynchroner Services wie etwa File–I/O durch Tasks, indem dafür blockierende Auftrags–/Rückmelde–Schnittstellen zur Verfügung gestellt werden. Die „synchrone“ Welt besteht damit aus vielen Threads, die durch I/O blockiert sein können, während die asynchrone Welt aus einem oder wenigen Threads besteht, die Interrupt–Behandlung machen.

In [SSRB00, S. 423] wird das Beispiel eines Netzwerk–Servers gebracht: Das Netzwerk–I/O ist asynchron aufzurufen. Von außen kommende Requests werden in eine Queue geschrieben und von „synchronen“ Threads aus der Queue geholt und bearbeitet. Die Antwort wird über die Queue zum Netzwerk–I/O zurückgeschrieben und zum Client zurückgesendet, siehe Abbildung 4.12(a)

Zur Realisierung der Queue wird das MONITOR OBJECT Pattern vorgeschlagen (wegen der Serialisierung der Zugriffe), für die “synchronen” Services ACTIVE OBJECT (wegen der Nebenläufigkeit der Services).

Gängige Betriebssysteme mit Multitasking bieten üblicherweise diese blockierenden Aufrufe und Queues an. Es besteht auch eine gewisse Ähnlichkeit zum PROACTOR (Abbildung 4.3): Die Completion Queue verbindet auch dort asynchrone Basisdienste mit dem Proactor, der zusammen mit der Anwendung in einer Task ausgeführt wird. Im Unterschied dazu sieht HALF–SYNC / HALF–ASYNC vor, dass auch mehrere nebenläufige „synchrone“ Tasks eine Queue benutzen können.



(a) HALF–SYNC / HALF–ASYNC (b) ASYNCHRONOUS COMPLETION TOKEN

Abbildung 4.12: Patterns zu Multitasking und Asynchrone Dienste: HALF–SYNC / HALF–ASYNC und ASYNCHRONOUS COMPLETION TOKEN

In [BH02, S. 39] heißt es, dass in einer Anwendung manche Services synchron und manche asynchron aufgerufen werden. Um das sauber auseinander zu halten, sollen alle asynchronen, systemnahen Aufrufe in einen Layer, alle synchronen, abstrakteren in einen anderen Layer sortiert werden, und zur Entkoppelung die Kommunikation der beiden Layer über einen Queuing Layer geregelt werden. Es handelt sich aber nicht um das LAYERS Pattern im engeren Sinn, denn dieses impliziert, dass zwar aus dem synchronen Layer in den Queuing Layer gerufen werden darf und von dort in den asynchronen Layer, aber nicht umgekehrt.

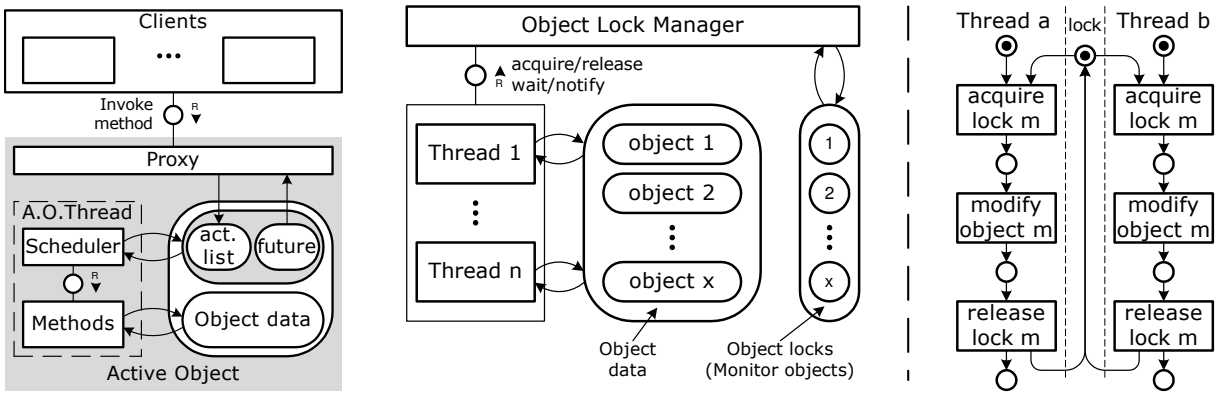
Half–Sync / Half–Reactive Weiter hinten in der Pattern–Beschreibung in [SSRB00, S. 440] wird die Variante HALF–SYNC / HALF–REACTIVE beschrieben, die das Pattern aus der Betriebssystem–Ebene hoch in die Anwendungsebene hievt. Hier wird beschrieben, wie ein "Reactive Thread" Events aufnimmt und Aufträge in eine (oder mehrere) Queues schreibt, die von "synchronen" Service Threads gelesen und bearbeitet werden — siehe dazu das JOB QUEUE Pattern in Abschnitt 7.2.5.

4.2.7.2 Asynchronous Completion Token

(Design–Pattern im Kapitel „Event Handling" [SSRB00, S. 261])

Benutzt eine Anwendung Services, die asynchron aufgerufen werden, setzt sie zuerst Aufträge ab, um später auf deren "Fertig"–Meldung (Completion) zu reagieren. Hat eine Anwendung mehrere Aufträge an einen Service Provider gerichtet, muss sie feststellen können, zu welchem Auftrag eine empfangene Antwort gehört.

Das Pattern ASYNCHRONOUS COMPLETION TOKEN schlägt nun vor, zusammen mit einem Auftrag eine ID, nämlich das Asynchronous Completion Token (ACT), mitzugeben, das der Service Provieder mit der Antwort unverändert zurückschickt. Anhand dieser ID ermittelt der Initiator (in der Rolle des Dispatchers, siehe auch PROACTOR) den Kontext des Auftrags und den Behandler der Antwort — den Completion Handler, siehe Abbildung 4.12(b).



(a) ACTIVE OBJECT (b) MONITOR OBJECT

Abbildung 4.13: Patterns zu Multitasking und Asynchrone Dienste: ACTIVE OBJECT und MONITOR OBJECT

4.2.7.3 Active Object

(Design–Pattern im Kapitel „Concurrency“ [SSRB00, S. 369])

Arbeiten mehrere Threads mit den gleichen Objekten, geht die Datenkapselung der Objekte verloren, da ja mehrere Threads gleichzeitig Operationen bezüglich eines Objekts durchführen können (siehe auch MONITOR OBJECT).

Setzt man das ACTIVE OBJECT Pattern ein, bewahrt man die Kapselung, indem nur ein Thread Operationen auf Objektdaten ausführt. Dazu werden, wie auch bei JOB QUEUE, Bearbeitungsaufträge durch die Client Threads in einen Puffer (Activation List) geschrieben. Ein eigener Scheduler liest den Puffer und führt die Aufträge nacheinander aus, wie in Abbildung 4.13(a) dargestellt. Für die Übergabe der Ergebnisse werden Datensätze, so genannte *Futures* angelegt. Der Proxy dient dazu, nach außen eine einfache Methodenschnittstelle anzubieten und damit das Einstellen der Aufträge in den Puffer zu kapseln. Da der Proxy in verschiedenen Threads benutzt werden kann, müssen die Operationen auf dem Puffer natürlich über Sperren synchronisiert werden.

Eine Verallgemeinerung des Patterns führt zur Inter–Thread–Kommunikation über Queues.

Thread Pool Active Object Als Variante wird in [SSRB00, S. 393] das THREAD POOL ACTIVE OBJECT erwähnt, bei dem der Request Buffer (activation list) nicht von einem, sondern von mehreren Threads pro Objekt gelesen wird, die damit nebenläufig Aufträge ausführen können. Pro Auftrag wird also ein Thread aus dem Pool benutzt, bis entweder der Pool keine Threads oder der Request Buffer keine Aufträge mehr hat. Die ursprüngliche Intention, nebenläufige Zugriffe auf Objektdaten zu verhindern, ist damit natürlich ad absurdum geführt — das beschriebene Prinzip gehört eigentlich in ein eigenes Pattern (siehe dazu auch WORKER POOL in Abschnitt 7.2.3).

4.2.7.4 Monitor Object

(Design–Pattern im Kapitel „Concurrency“ [SSRB00, S. 399])

Auch hier geht es darum, die Datenkapselung eines Objekts zu bewahren, wenn mehrere Threads Operationen ausführen wollen.

Ein MONITOR OBJECT lässt immer nur genau einen Thread zur Ausführung von Methoden auf einem Objekt zu. Dafür wird eine objektweite Sperre benötigt, sowie ein Mechanismus zum Blockieren eines aufrufenden Threads, falls die Sperre nicht verfügbar ist. Dazu kommt noch ein wait/notify Mechanismus, über den Threads miteinander kommunizieren können.

Abbildung 4.13(b) zeigt eine Systemsicht, bei der Objekte auf Datensätze im Speicher reduziert sind, auf die mehrere Threads ungehindert zugreifen können.[5] Der Schutz erfolgt nun darin, dass vor einem Zugriff auf Objektdaten mit `acquire` eine Sperre für dieses Objekt angefordert wird, die nach erfolgreicher Operation mit `release` zurückgegeben wird.

In Java kann man Methoden mit dem Schlüsselwort `synchronized` markieren, wodurch der Compiler am Anfang und Ende der Methode Anweisungen zur Anforderung und Rückgabe der Objekt–Sperre einbaut.

4.3 Thematische Kategorisierungen

Die in Abschnitt 4.2 vorgestellten Patterns sind thematisch zum Teil anders sortiert als in der Literatur, wobei sich bei manchen Patterns die Einordnungen auch innerhalb verschiedener Veröffentlichungen unterscheiden.

Die thematische Kategorisierung der vorgestellten Patterns wird anhand dreier Veröffentlichungen untersucht: Dem Buch „Pattern–Oriented Software Architecture“ von Buschman et al. [BMR+96], dem zweiten Band von Schmidt, Buschmann et al. [SSRB00], schließlich dem Workshop–Beitrag von Buschmann und Henney [BH02]. Alle drei bringen die Patterns in Form eines Pattern–Systems oder einer Pattern Language in Beziehung.

4.3.1 Pattern–Oriented Software Architecture 1 (POSA1, 1996)

Tabelle 4.7 aus [BMR+96, S. 366] zeigt die Einordnung der dort vorgestellten Patterns, die um Patterns aus „Design Patterns“ [GHJV94] erweitert wurde.

Es fällt auf, dass für die Architektur–Patterns andere Kategorien gewählt wurden als für die Design Patterns. Das liegt wahrscheinlich daran, dass mit Architektur–Patterns ein anderes Ziel verfolgt wird als mit Design–Patterns, nämlich ein Framework oder eine Basisstruktur für das Software–System vorzugeben, während Design–Patterns ja für speziellere Probleme zuständig sind.

Fast alle im Buch vorgestellten Patterns tauchen auch in der thematischen Kategorisierung des zweiten Bands wieder auf:

[5] Es gibt mehrere Möglichkeiten, Objekte in Aufbaustrukturen darzustellen. Die hier benutzte Datensatz–Sicht ist für Betrachtungen im Zusammenhang mit Multithreading angemessen. Eine ausführliche Darstellung zu diesem Thema gibt [TG03].

	Architectural Pattern	Design Pattern	Idiom
From mud to structure	**Layers** **Pipes and Filters** **Blackboard**	Interpreter	
Distributed Systems	**Broker** **(Pipes and Filters)** **(Microkernel)**		
Interactive Systems	**Model-View-Controller** **Presentation-Abstraction-Control**		
Adaptable Systems	**Microkernel** **Reflection**		
Structural Decomposition		**Whole-Part** Composite	
Organization of Work		**Master-Slave** Chain of Responsibility Command Mediator	
Access Control		**Proxy** Facade Iterator	
Management		**Command Processor** **View Handler** Memento	
Communication		**Forwarder-Receiver** **Client-Dispatcher-Server** **Publisher-Subscriber**	
Resource Handling		Flyweight	**Counted Pointer**

Tabelle 4.7: Kategorisierung der Patterns nach POSA 1 [BMR+96, S. 366] (1996)
Die fett gedruckten Patterns werden im Buch beschrieben, während die nicht fett gedruckten aus „Design Patterns“ von Gamma et al. [GHJV94] stammen. Die Patterns PIPES AND FILTERS und MICROKERNEL können in zwei Kategorien eingeordnet werden

4.3.2 Pattern–Oriented Software Architecture 2 (POSA2, 2000)

Tabelle 4.8 zeigt die Kategorisierung der Patterns aus [SSRB00, S. 525].

	Architectural Pattern	**Design Pattern**	**Idiom**
Base-line Architecture	**Broker** **Layers** **Microkernel**		
Communication	**Pipes and Filters**	Abstract Session **Command Processor** **Forwarder-Receiver** Observer Remote Operation Serializer	
Initialization		Activator **Client-Dispatcher-Server** Evictor Locator Object Lifetime Manager	
Service Access and Configuration	**Interceptor**	**Component Configurator** **Extension Interface** Half Object plus Protocol Manager-Agent **Proxy** **Wrapper Facade**	
Event Handling	**Proactor** **Reactor**	**Acceptor-Connector** **Asynchr. Completion Token** Event Notification Observer **Publisher-Subscriber**	
Synchronization	Object Synchronizer [SPM96]	Balking Code Locking, Data Locking Guarded Suspension **Double-checked Locking Optim.** Reader-Writer Locking Specific Notification **Strategized Locking** **Thread-Safe Interface**	**Scoped Locking**
Concurrency	**Half-Sync/Half-Async** **Leader-Followers**	**Active Object** **Master-Slave** **Monitor Object** Producer-Consumer Scheduler Two-Phase-Termination **Thread-Specific Storage**	

Tabelle 4.8: Kategorisierung der Patterns nach POSA 2 [SSRB00, S. 525] (2000)
Die fett gedruckten Patterns stammen aus POSA 1 und 2, die übrigen aus verschiedenen Quellen

Patterns wurden gegenüber dem ersten Band umsortiert. Waren LAYERS, PIPES AND FILTERS und BLACKBOARD noch zusammen in der Kategorie „From mud to structure“ beschrieben, fallen jetzt nur LAYERS, BROKER und MICROKERNEL in die Kategorie „Base line architecture“, geben also vor, in welcher Form Funktionalität in das System integriert werden kann — sei es rein auf Code–Seite (LAYERS) oder über Strukturen, die zur Laufzeit relevant sind, weil sie für die Kommunikation benötigt werden (MICROKERNEL, BROKER). Manche Einordnungen sind diskussionswürdig, so findet man etwa PIPES AND FILTERS in der Kategorie „Communication“ und CLIENT–DISPATCHER–SERVER in „Initialization“.

Einige Patterns, wie z.B. BLACKBOARD, MODEL–VIEW–CONTROLLER, PRESENTATION–ABSTRATION–CONTROL oder REFLECTION tauchen nicht mehr auf, da es in diesem Band the-

matisch um nebenläufige und verteilte Systeme geht („Patterns for Concurrent and Networked Objects“).

4.3.3 Buschmann/Henney (2002)

Tabelle 4.9 zeigt eine spätere Kategorisierung aus dem Workshop–Beitrag „A Distributed Computing Pattern Language“ [BH02] von Buschmann und Henney, in dem ebenfalls nur Patterns vorkommen, die für Distributed Computing interessant sind. Diese Einschränkung ermöglicht es, sie in einer Pattern Language (siehe Abschnitt 3.6) anzuordnen. Hier werden die Application Infrastructure Patterns übrigens auch *Strategic Patterns* genannt.

	Architectural Pattern	**Design Pattern**
Distribution Infrastructure	**Broker**	**Client--Dispatcher--Server**
Application Infrastructure	**Layers** **Pipes and Filters** **Blackboard** **Model-View-Controller** **Presentation-Abstraction-Control** **Reflection** **Microkernel**	
Concurrency	**Half-Sync / Half-Async** **Leader / Followers**	**Active Object** **Monitor Object** Guarded Suspension
Event Handling	**Reactor** **Proactor**	**Acceptor-Connector** **Async. Completion Token**

Tabelle 4.9: Kategorisierung der Patterns nach Buschmann und Henney [BH02] (2002)
Die fett gedruckten Patterns stammen aus POSA 1 und 2, GUARDED SUSPENSION aus [Lea99, S. 87ff]

4.3.4 Bewertung der thematischen Kategorisierung

Bei der thematischen Kategorisierung der vorgestellten Patterns fällt auf, dass es bei manchen deutliche Unterschiede gibt, die zum Teil durch den unterschiedlichen Autorenkreis bedingt sind oder auch durch die Zeit, die zwischen den Publikationen vergangen ist und in der die Autoren weitere Erfahrungen mit dem Einsatz der Patterns gesammelt haben. Weiterhin muss beachtet werden, dass der Einsatzzweck eine Rolle spielt, also die Anwendungsdomäne, für die das Pattern–System zusammengestellt wurde.

Beispielsweise wurde das Pattern PIPES AND FILTERS in die Kategorien *From Mud to Structure, Distributed Systems, Communication* und *Application Infrastructure* eingeordnet. Das hängt damit zusammen, dass bei der Benutzung von Netzwerkkommunikation Datenströme eine wichtige Rolle spielen, andererseits die Modularisierung der Datenstromverarbeitung in Form der Filter die Software strukturiert, und schließlich mit den Pipes ein Basiskonzept von Betriebssystemen vorgestellt wird.

Ein anderes Beispiel sind Event Handling Patterns wie REACTOR und PROACTOR, zu denen in der Auswahl in Abschnitt 4.2.3 auch LEADER / FOLLOWERS gezählt wird, während dieses Pattern in der Literatur ([SSRB00] und [BH02]) unter *Concurrency* eingeordnet wird. Der Zweck

von LEADER / FOLLOWERS ist aber der selbe wie bei den anderen beiden Patterns, nämlich die Verarbeitung von Aufträgen. Während bei REACTOR eine nebenläufige Bearbeitung nicht möglich ist, benutzt der PROACTOR für diesen Zweck asynchrones I/O und LEADER / FOLLOWERS Multitasking.
Soll man diese Patterns nun nach ihrem Einsatzzweck oder nach den verwendeten technischen Mitteln sortieren?

Die Schwierigkeit der thematischen Kategorisierung von Architektur–Patterns besteht also darin, dass diese verschiedene Bestandteile haben, die verschiedenen Kategorien zugeordnet werden können.

Doch selbst wenn man eine klare, redundanzfreie thematische Einteilung der Patterns hat, läuft man dennoch Gefahr, dass ein Leser diese Kategorisierung nicht nachvollzieht und damit ein Pattern für sein Problem nicht findet.

Die Bandbreite der vorgestellten Patterns ist groß, so dass leicht der Eindruck eines „Gemischtwarenladens“ entsteht. Es ist wahrscheinlich in der Praxis besser, unterschiedliche Pattern–Systeme oder Pattern Languages für eng umrissene Problembereiche zusammenzustellen und dabei bewusst in Kauf zu nehmen, dass grundlegende Patterns in verschiedenen Systemen immer wieder vorkommen. Dadurch entfällt auch die Notwendigkeit einer allzu breiten thematischen Kategorisierung — schließlich ist diese bereits durch den Problembereich, den die Pattern Language abdeckt, vorgegeben.

4.4 Verschiedene Abstraktionsgrade von Patterns

Bei der Untersuchung der vorgestellten Patterns fallen durchaus Unterschiede bezüglich ihres Abstraktionsgrads auf. Während eine Reihe von Patterns recht allgemeine Konzepte beschreibt, die ohne weiteres auf beliebige informationelle Systeme anwendbar sind, beschreiben andere Basis–Mechanismen, die in gängigen Betriebssystemen meist bereits implementiert sind.

Die Patterns aus Abschnitt 4.2 sollen daher nach steigendem Abstraktionsgrad sortiert werden:

Basis–Funktionalität
: Die im Pattern beschriebenen Konzepte betreffen Basis–Mechanismen, wie sie in Betriebssystemen und Abwicklern für Programme zu finden sind, beispielsweise die Sperren von MONITOR OBJECT.

Konzepte für programmierte Systeme
: Diese Patterns sind nur in programmierten Systemen anwendbar, beispielsweise der COMPONENT CONFIGURATOR, mit dem das Nachladen von Code organisiert wird.

Allgemeine Konzepte
: Die in den Patterns beschriebenen Konzepte lassen sich auch auf nicht programmierte informationelle Systeme anwenden, beispielsweise wird PUBLISHER–SUBSCRIBER bei Zeitschriftenabonnements verwendet.

	Allgemeine Konzepte	Konzepte für progr. Systeme	Basis–Funktionalität (Teil des Betriebssystems oder der Programmiersprache)
Verarbeitung von Daten	BLACKBOARD PIPES AND FILTERS		Pipes aus PIPES AND FILTERS
Event–verarbeitende Systeme	HALF–SYNC / HALF–REACTIVE LEADER / FOLLOWERS PROACTOR PUBLISHER–SUBSCRIBER REACTOR		Event Demultiplexer aus REACTOR
Erweiterbare / Änderbare Systeme	INTERCEPTOR	REFLECTION COMPONENT CONFIGURATOR MICROKERNEL (LAYERS)	
Client–Server–Kommunikation	PROXY CLIENT–DISPATCHER–SERVER		
Netzwerk–Kommunikation		BROKER	ACCEPTOR–CONNECTOR FORWARDER–RECEIVER
Multitasking & asychr. Dienste	ASYNC. COMPLETION TOKEN	THREAD POOL ACTIVE OBJECT	HALF–SYNC / HALF–ASYNC MONITOR OBJECT

Betrachtet man auch Software–Strukturen, kann man neben der Ebene „Basis–Funktionalität“ auch eine Ebene „Sprach–Erweiterung“ ausmachen, deren Patterns nur im Rahmen einer bestimmten Programmiertechnik verwendbar sind. Beispiel: ABSTRACT FACTORY [GHJV94, S. 87] ist ein Pattern, das nur im Zusammenhang mit objektorientierten Sprachen sinnvoll einsetzbar ist.

4.5 Architektur– oder Design–Pattern?

In der betrachteten Literatur [BMR+96, SSRB00, BH02] werden die Patterns klar in Architektur– und Design–Patterns unterteilt. Im Gegensatz zur oben betrachteten thematischen Kategorisierung ist diese Einteilung über den Zeitraum der verschiedenen Veröffentlichungen keinen Änderungen unterworfen. Das führt zu zwei möglichen Schlussfolgerungen: Entweder gibt es klare und gut nachvollziehbare Regeln, nach denen ein Pattern entweder ein Architektur–Pattern, ein Design–Pattern oder ein Idiom ist (siehe Abschnitt 3.4.1.2), oder diese Einteilung spielt mittlerweile keine wichtige Rolle mehr. Völters et al. merken in „Server Component Patterns“ zwar an, dass die Unterscheidung in Architektur– und Design–Patterns auch in ihrem Pattern–System möglich ist, wenden diese aber nicht an [VSW02, S. 5].

Bei vielen der in Abschnitt 4.2 vorgestellten Patterns werden Systemstrukturen und dort auftretende Mechanismen beschrieben. Manche davon sind als Design–Patterns, andere als Architektur–Patterns bezeichnet. Diese Unterteilung soll hier nachvollzogen werden.

4.5.1 Architektur–Patterns: Definitionen

In „Pattern–oriented Software Architecture“ (POSA1) werden Architektur–Patterns definiert als Patterns zur grundlegenden Strukturierung des Softwaresystems. Die eigentliche Funktionalität steckt in Subsystemen, deren Einbindung durch das Pattern geregelt wird:

> „An *architectural pattern* expresses a fundamental structural organization schema for software systems. It provides a set of subsystems, specifies their responsibilities, and includes rules and guidelines for organizing the relationships between them.“ [BMR+96, S. 12].

Betrachtet man die in Abschnitt 4.2 beschriebenen Patterns unter Berücksichtigung dieser Definition, so kann man folgendes feststellen:

Es ist ein wichtiges Zeichen eines Architektur–Patterns, dass zu den Bestandteilen der beschriebenen Struktur Subsysteme oder Komponenten gehören, deren Funktionalität *nicht* vom Pattern festgelegt wird. Beispielsweise legt INTERCEPTOR nur fest, wie der Aufrufmechanismus zwischen Application Core und Plug–In aussieht, aber nicht, welche Art von Funktionalität im Application Core oder Plug–In realisiert wird.

Mit einem Architektur–Pattern wird daher ein Prinzip beschrieben, nach dem eine Anwendung funktionieren soll. Dabei gibt es einerseits eine Struktur aus Subsystemen und Komponenten, die die Funktionalität der Anwendung kapseln, und andererseits eine eventuell erforderliche Infrastruktur. Beispielsweise ist bei den Patterns REACTOR, PROACTOR und INTERCEPTOR als Infrastruktur jeweils ein Dispatcher und Registration Manager erforderlich, um registrierte Prozeduren zur Behandlung von Events aufzurufen. Die Anwendungsfunktionalität steckt dagegen in den Event–Behandlern.

Damit kann man die Definition der Architektur–Patterns auch so umformulieren:

> Ein Architektur–Pattern beschreibt ein *Konzept* zum Aufbau einer Anwendung. Darin wird eine *Struktur* als Gerüst vorgegeben, in die die *Funktionalität* der Anwendung *eingebettet* werden kann, sowie ein Prinzip, nach dem deren *Aufruf erfolgt*. Dafür ist meist eine *Infrastruktur* erforderlich, deren Implementierung wiederum entweder komplett im Pattern beschrieben, oder als vorhanden vorausgesetzt wird. In jedem Fall werden *Schnittstellen* definiert, die benutzt werden müssen, um die Funktionalität im System nutzen zu können.

Kriterien für ein Architektur–Pattern sind also:

- Eine (System– oder auch Software–) Struktur, in die die Funktionalität (Datenverarbeitung, Auftragsbehandlung) in Form von Modulen / Subsystemen eingebettet werden kann (vergleiche dazu auch den Framework–Begriff aus Abschnitt 3.4.1.2)
- Ein sich daraus ergebender Hauptablauf, nach dem die Module aufgerufen werden
- Definition von Schnittstellen und Protokollen, die einen größeren Teil des Systems betreffen

- Schaffung der Grundlage für Flexibilität und Änderbarkeit

Man kann diese Kriterien gut am Beispiel des INTERCEPTOR–Patterns zeigen, das Voraussetzungen für die nachträgliche Erweiterbarkeit einer Anwendung schafft: Die Struktur für die Anwendung besteht einerseits aus einem Anwendungskern mit Hooks und andererseits aus Plug–Ins mit Event–Behandlern, die beim Auftreten eines Hooks aufgerufen werden. Zentrale Infrastrukturkomponenten sind der Dispatcher und die Plug–In–Registrierung. Ein Plug–In muss eine festgelegte Schnittstelle anbieten und nach der Registrierung auf Aufrufe vom Anwendungskern warten.

4.5.2 Architektur– und Design–Patterns in POSA

Die Tabellen in Abschnitt 4.2 zeigen für alle dort vorgestellten Architektur– und Design–Patterns, welche Aussagen über Konzept, Struktur und Ablauf der Anwendung, sowie die dafür benötigte Infrastruktur gemacht werden, wobei ausschließlich die Systemstrukturen betrachtet werden.

Überprüft man nun die Kategorisierung in Architektur– oder Design–Pattern für alle der in Abschnitt 4.2 vorgestellten Patterns, dann ist die Entscheidung bei manchen nur schwer nachzuvollziehen. Dazu gehören die folgenden Architektur–Patterns, die nach den aufgeführten Kriterien eher Design–Patterns oder Stile sind:

LAYERS Es werden keine Aussagen über Funktionalität, Schnittstellen oder Infrastruktur gemacht. Es handelt sich eher um einen Stil oder eine allgemeine Qualität wie etwa Kapselung oder eine einheitliche Namensregelung für Bezeichner.

REFLECTION Die Einführung einer Indirektionsstufe betrifft natürlich zentrale Schnittstellen und damit große Teile eines Programms. Es gibt aber keinen Hauptablauf oder eine Struktur zur Einbettung von Funktionalität.

HALF–SYNC / HALF–ASYNC Dieses Pattern gibt als Struktur vor, Anwendungsteile mit synchronen (blockierenden) Service–Aufrufen in eigenen Threads von solchen mit asynchronen Aufrufen zu trennen und deren Kommunikation über Queues zu regeln. Ansonsten wird keine Struktur oder Ablauf definiert.

4.5.3 Das Lokalitätskriterium nach Eden und Kazman

Eden und Kazman demonstrieren in [EK03] ihre Intensional– und Lokalitätskriterien anhand von Patterns (siehe auch Abschnitt 2.2.3).

Die Design Patterns aus der Sammlung von Gamma et al. [GHJV94] erfüllen alle die Kriterien intensional und lokal und beschreiben damit Design. Das trifft auch auf die Umsetzung des MODEL–VIEW–CONTROLLER–Patterns in der Java Swing Bibliothek zu, obwohl dieses Pattern in [BMR+96] und [BH02] als Architektur–Pattern aufgeführt wird. Das Lokalitätskriterium bezieht sich auf die Spezifikation, üblicherweise auf den Code.

Die Architektur–Patterns LAYERED ARCHITECTURE und PIPES AND FILTERS aus [SG96] beschreiben dagegen Architektur, da hier das Kriterium „nicht–lokal“ erfüllt ist. Eden und Kazman gehen hierbei davon aus, dass im Beispiel der PIPES AND FILTERS das gesamte System

diesem Muster folgen muss, und Ausnahmen explizit als solche gekennzeichnet werden müssen.

In der Praxis stellt sich die Frage, wo die Systemgrenzen sind, die das Lokalitätskriterium betrifft. Die Vorstellung, dass sich jede Zeile Code zu einem Programm beispielsweise in das Pattern PIPES AND FILTERS einordnen ließe, ist praxisfern. Dagegen kann man durchaus Subsysteme abgrenzen, in denen ein Architektur–Pattern gilt.

4.6 Beziehungen zwischen Patterns

Eine gute Darstellung von Pattern–Beziehungen ist wichtig, da bei der Suche nach geeigneten Patterns nicht nur die thematische Kategorisierung, sondern auch deren Beziehungen zu anderen Patterns eine Rolle spielt.

In den betrachteten Publikationen werden Beziehungen zwischen Patterns, ob Architektur– oder Design–Patterns, anhand von Pattern–Systemen bzw. Pattern Languages beschrieben. Diese Beziehungen werden einerseits über die Beschreibung des Kontexts und der Konsequenzen zu einem Pattern bereits aufgebaut, andererseits dienen Grafiken dazu, eine Übersicht über die Patterns und ihre Beziehungen zu geben.

4.6.1 Beispiel: Concurrency Patterns nach Buschmann und Henney (2002)

4.6.1.1 Die Patterns und ihre Beziehungen

Buschmann und Henney haben in [BH02] eine Pattern Language zum Thema Verteilte Systeme aus den Patterns aus POSA 1 und 2 zusammengestellt und diese damit in eine deutlich engere Beziehung gestellt. Abbildung 4.14 aus [BH02, S.38] zeigt die Beziehungen der Concurrency Patterns HALF–SYNC / HALF–ASYNC, LEADER / FOLLOWERS, MONITOR OBJECT, ACTIVE OBJECT und GUARDED SUSPENSION (alle dunkel gefärbt) zu den anderen.

Die Kanten zeigen Beziehungen zwischen Patterns, nämlich welche weiteren Aufgaben mit welchem Pattern zu lösen sind, wenn man ein Pattern einsetzt. Beispielsweise führen einige Kanten auf MONITOR OBJECT, nämlich falls Serialisierung bzw. Synchronisierung benötigt wird. Bei mehreren wegführenden Kanten mit gleicher Beschriftung handelt es sich um Alternativen.

4.6.1.2 Das Nebeneinander von Patterns verschiedener Ebenen

Bei Buschmann / Henney werden HALF–SYNC / HALF–ASYNC und LEADER / FOLLOWERS als Alternativen dargestellt. Wie in Abschnitt 4.4 gezeigt, sind diese beiden Patterns aber nicht auf einer Ebene vergleichbar:

Half–Sync / Half–Async steht für ein Prinzip, nämlich die Trennung der Anwendung in einen synchronen Teil, in dem Threads nebenläufig agieren, einen asynchronen Teil, in dem Ein–/Ausgabedienste ausgeführt werden, sowie deren Kopplung mittels Queues und blockierendem Warten. Hier werden Betriebssystemmittel als Infrastruktur vorausgesetzt. In dem im Abschnitt 4.2.7.1 gezeigten Beispiel, dem Socket Layer, bietet das Betriebssystem bereits alles an, inklusive Queue.

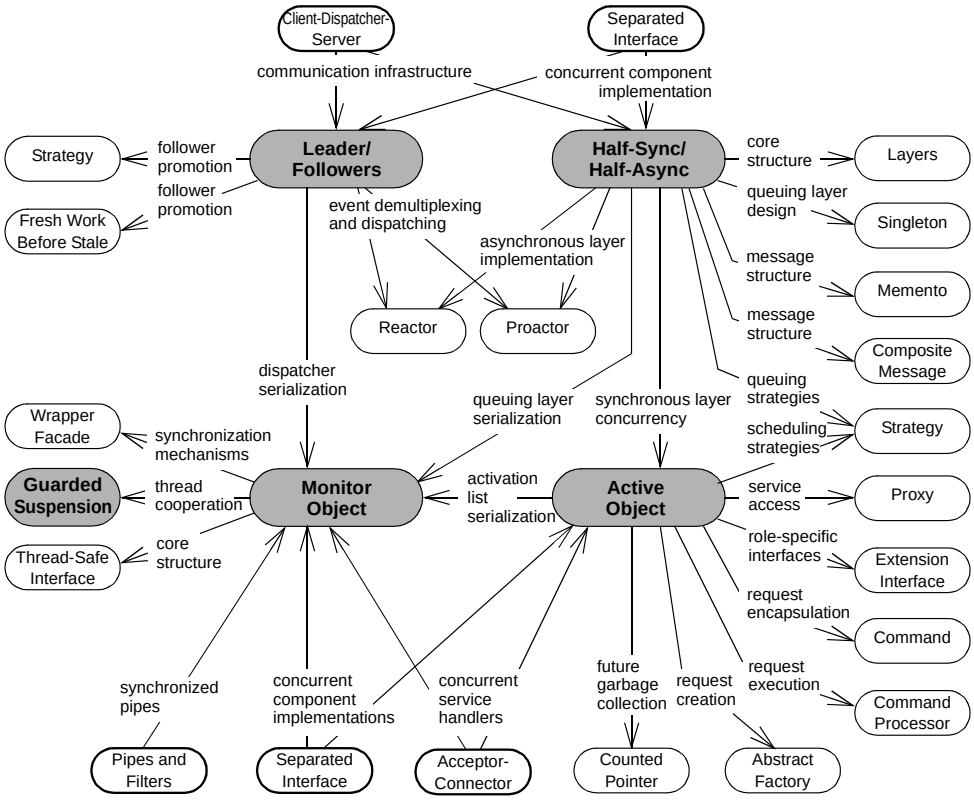


Abbildung 4.14: Darstellung der Pattern Language zu den Concurrency Patterns nach Buschmann und Henney [BH02, S.38] (2002)
Die grau gefärbten Patterns werden im Kapitel „Concurrency Patterns“ in [BH02] beschrieben, die anderen tauchen entweder in weiteren Kapiteln auf (z.B. REACTOR/PROACTOR) oder sind nicht Teil der Pattern Language (z.B. MEMENTO)

Half–Sync / Half–Reactive Dieses Pattern wird als Variante von HALF–SYNC / HALF–ASYNC vorgestellt. Ein Listener–Thread reagiert auf Events von außen, ruft die Event Handler aber nicht im eigenen Thread–Kontext auf, sondern schickt Aufträge über Queues an andere Threads, die schließlich die Event Handler ausführen. Das ist aber eigentlich keine Variante, sondern ein eigenständiges Pattern, wobei auch die Infrastruktur, nämlich die Job Queues, auf der Anwendungsebene zu finden ist.

Hier wird wieder das Konzept einer reaktiven Anwendung, wie in REACTOR, beschrieben. Der Unterschied besteht darin, dass die Event Handler in eigenen Threads ausgeführt werden.

Leader / Followers Direkt daneben steht das LEADER / FOLLOWERS Pattern: Statt wie bei HALF–SYNC / HALF–REACTIVE Aufträge von einem Event–Fänger (Listener) in Queues zu schieben, behandelt der Listener ein Event selbst und gibt die Rolle des Listeners an den nächsten wartenden Thread ab. Für die Strukturierung der Anwendung gilt das gleiche wie bei HALF–SYNC / HALF–REACTIVE, nämlich die Event–Behandler in eigenen Threads auszuführen.

4.6.2 Vorschlag: Schichtung von Patterns

Nimmt man das Beispiel Socket Layer aus der Beschreibung des LEADER / FOLLOWERS Patterns, dann baut das Pattern eigentlich auf dem HALF–SYNC / HALF–ASYNC Pattern auf: Der Listener gehört zum synchronen Teil, der Socket Layer des Betriebssystems zum asynchronen Teil. Bei einem ankommenden Request werden die Request–Informationen in eine Queue geschrieben und der Listener geweckt, der die Informationen dann auswerten und die Verbindung aufbauen kann.

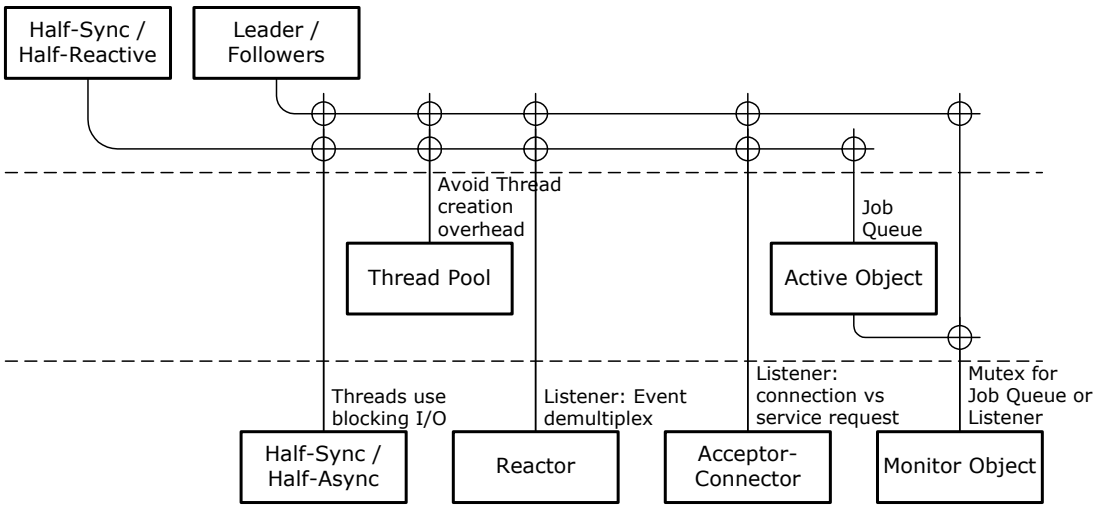


Abbildung 4.15: Schichtungsbild zu den Patterns LEADER / FOLLOWERS und HALF–SYNC / HALF–ASYNC

Die Schichtung in Abbildung 4.15 zeigt am Beispiel der Patterns HALF–SYNC / HALF–REACTIVE und LEADER / FOLLOWERS, wie diese auf anderen Patterns aufbauen, indem sie die dort beschriebenen Prinzipien benutzen, die in der Abbildung als Kommentar an einer Kante auftauchen. Ganz unten stehen Betriebssystem–Mechanismen, die eingesetzt werden, beispielsweise blockierende Ein–/Ausgabe–Operationen oder Sperren für Tasks. In der Mitte

sind Patterns, die für bestimmte Strategien stehen, wie der THREAD POOL [SSRB00, S. 393, in: Active Object] zur Vermeidung von Verzögerungen durch bedarfsgetriebene Erzeugung von Threads oder ACTIVE OBJECT für die Inter–Thread–Kommunikation über eine Queue (Activation List). Oben schließlich stehen die beiden Patterns als echte Alternative nebeneinander. Eine weiterführende Behandlung dieser Patterns erfolgt in Kapitel 7.

In gleicher Weise kann auch das BROKER Pattern betrachtet werden, in dem eine Gesamtlösung für ein verteilungstransparentes Client–Server–System vorgestellt wird. Hier findet man Bestandteile wie PROXY oder CLIENT–DISPATCHER–SERVER wieder.

4.7 Architektur–Patterns: Bewertung

Ein Architektur–Pattern gibt ein Konzept zum Aufbau einer Anwendung vor. Daraus resultieren eine Struktur, in die die Anwendungsfunktionalität eingebettet werden kann, sowie ein Gesamtablauf für die Anwendung. Weiterhin ist eine bestimmte Infrastruktur erforderlich, um das beschriebene Konzept umzusetzen.

Die verschiedenen Bestandteile eines Architektur–Patterns führen zu folgenden Problemen:

- Wofür steht ein Architektur–Pattern? Für das Konzept, die Anwendungsstruktur, die Infrastrukturkomponenten oder alles zusammen? Passt der Pattern–Name zur Lösung?
- Ist die Unterscheidung in Architektur– und Design–Pattern in der Praxis relevant? Wenn ja, dann müssten Architektur–Patterns am Anfang der Planungsphase eingesetzt werden. Nützt dann aber die detailreiche Implementierungsbeschreibung?
- In welchen Themenkreis ist ein Architektur–Pattern einzuordnen?
  Die Schwierigkeit der thematischen Kategorisierung von Architektur–Patterns (siehe Abschnitt 4.3) besteht darin, dass diese verschiedene Bestandteile haben, die verschiedenen Kategorien zugeordnet werden können.
- Welche Rolle spielen Software–Strukturen und Code?
  In den meisten Architektur– und in vielen Design–Patterns werden Konzepte beschrieben, die die Systemstrukturen beeinflussen. Dabei sind aber viele konkrete Implementierungen denkbar, weshalb Software–Strukturen im Pattern nur eine untergeordnete Rolle spielen sollten (siehe auch Kapitel 5 zum Thema Darstellung).

Im folgenden werden die Hauptaspekte der Benennung und der Gefahr der Überfrachtung näher betrachtet.

4.7.1 Wofür steht ein Architektur–Pattern?

Wie bereits in Abschnitt 4.5 gezeigt, beschreibt ein Architektur–Pattern ein Konzept, aus dem sich die Teile Anwendungsstruktur, Hauptablauf und erforderliche Infrastruktur ergeben. Damit sollte der Name eines Architektur–Patterns für genau dieses Konzept stehen.

Die Benennung eines Architektur–Patterns erfolgt aber häufig nach einer Infrastruktur–Komponente, z.B. Broker, Microkernel, Reactor oder Blackboard. Das führt dazu, dass der Name des Architektur–Patterns auch dann verwendet wird, wenn damit eigentlich eine darin

benutzte Infrastruktur–Komponente gemeint ist. Beispielsweise wird in [BH02, S. 14] vom Pattern LAYERS Bezug auf PIPES AND FILTERS genommen, mit der Begründung, dass damit die inter–Layer Kommunikation realisiert werden kann. Damit ist aber eigentlich gemeint, dass Dienste verschiedener Layers über die Infrastruktur–Komponente *Pipe* miteinander kommunizieren können.

Wird dagegen auf das Pattern PROACTOR verwiesen, ist damit die Strukturierung der Anwendung in Event–Behandler für Auftrags–Rückmeldungen gemeint, zumal die Infrastruktur–Komponenten (Dispatcher mit Registrierung) sich nicht von denen der Patterns REACTOR und INTERCEPTOR unterscheiden.

Die unterschiedliche thematische Einordnung der Patterns resultiert auch aus dem Umstand, dass mehrere Bestandteile der Lösung beschrieben werden. PIPES AND FILTERS taucht in den Kategorien Application Infrastructure, Distributed Systems und Communication auf. Die Pipe transportiert natürlich Datenströme, das gleiche gilt für TCP Netzwerkverbindungen. Das Konzept für die Anwendung ist aber die modulare Datenstromverarbeitung.

4.7.2 Überfrachtete Patterns

Manche Varianten von Patterns stellen durchaus eigenständige Patterns dar. Beispiele dafür sind ACTIVE OBJECT mit der Variante THREAD POOL sowie HALF–SYNC / HALF–ASYNC mit der Variante HALF–SYNC / HALF–REACTIVE, die dazu noch verschiedenen Ebenen zuzuordnen sind.

Daneben gibt es große Patterns, die aus anderen zusammengesetzt sind, beispielsweise BROKER, sowie Patterns, die eigentlich wie ein Design–Pattern eine Teillösung beschreiben, aber im Kontext eines Architektur–Patterns stehen, beispielsweise LEADER / FOLLOWERS im Kontext von REACTOR (Konzept der Event–getriebenen Anwendung).

Die in Abschnitt 4.6.2 gezeigte Schichtung von Patterns stellt eine Möglichkeit zur Strukturierung von Patterns dar, die innerhalb einer Pattern Language genutzt werden kann. Ihr Einsatz hat möglicherweise zur Folge, dass eine Pattern Language nur wenige Architektur–Patterns zur Anwendungsstrukturierung und mehrere, teils alternative Design–Patterns für die Infrastruktur enthält.

4.7.3 Fazit

Ein Problem an bestehenden Architektur–Patterns ist der Umstand, dass als Lösung neben dem Strukturierungskonzept für das (Sub–) System auch die Realisierung der dazu erforderlichen Infrastruktur beschrieben wird und diese meist auch Namensgeber des Patterns ist.

Es ist daher nahe liegend, diese beiden Teile eines Architektur–Patterns voneinander zu trennen, besonders wenn es mehrere Alternativen für die Realisierung der Infrastruktur gibt. Das führt allerdings dazu, dass ein Pattern zur Strukturierung der Anwendung nur mit einem Pattern zur Beschreibung der Infrastruktur zusammen sinnvoll ist, also innerhalb eines Pattern–Systems stehen muss. Der Vorteil eines Pattern–Systems besteht allerdings darin, dass man es besser auf einen bestimmten Themenbereich abstimmen und alternative Lösungen mit ihren Vor– und Nachteilen einfacher identifizieren kann. In Kapitel 7 ist als Beispiel eine Pattern Language für Multitasking Server beschrieben.

Schließlich beschreiben nicht nur Architektur–Patterns wichtige und grundlegende Lösungen für Probleme, die früh von den Architekten gelöst werden müssen, um eine arbeitsteilige Entwicklung, Flexibilität und Änderbarkeit zu erlauben.

Kapitel 5

Darstellung von Architektur–Patterns in der Literatur

Im Abschnitt 4.2 des vorigen Kapitels wurden eine Reihe von Patterns in knapper Form vorgestellt, wobei der Schwerpunkt darauf lag, die Idee hinter einem Pattern zu vermitteln. In der Pattern–Literatur steht üblicherweise deutlich mehr Platz zur Verfügung, in dem neben Problem, Trade–Offs und Lösung auch Beispiele und Code–Stücke gezeigt werden.

In diesem Kapitel geht es nun darum, Pattern–Beschreibungen dahingehend zu untersuchen, wie die Idee hinter dem Pattern vermittelt wird. Dazu sind besonders die verwendeten Diagramme und Grafiken von Interesse, da diese, ähnlich wie der Name eines Patterns, wesentlich zur Einprägsamkeit beitragen. So verwendet Christopher Alexander für jedes Pattern in seinem Bauarchitektur–Buch [AIS+77] ein Foto als Aufhänger, das die Idee möglichst einprägsam machen soll (siehe auch Abbildung 3.1). Die (technische) Lösung wird dort mit einer kleinen Skizze illustriert.

Bei der grafischen Darstellung von Patterns im Software–Bereich überwiegen Diagrammtypen aus dem objektorientierten Entwurf, üblicherweise Klassen– und Sequenzdiagramme nach Booch oder UML. Das liegt wohl auch daran, dass die ersten Patterns für Software von Gamma et al. [GHJV94] Probleme im Entwurf objektorientierter Programme behandelten.

Mittlerweile gibt es viele Varianten für die grafische Darstellung von Patterns, da Klassen– und Sequenzdiagramme für viele Patterns keine geeignete Anschauung bieten. Neben eher formalen Ansätzen wie den Class–Responsibility–Collaborator Cards (CRC–Cards) in POSA1/2 [BMR+96, SSRB00] gibt es einige halb– oder nicht–formale Diagrammtypen bis hin zu den typischen Clip–Art–Aufbaudiagrammen.

Wie bereits in Abschnitt 4.7 gezeigt, werden in einem Architektur–Pattern zwei Ideen vermittelt, nämlich eine zur Strukturierung der Anwendungsfunktionalität und eine zur Realisierung der dazu erforderlichen Infrastruktur. Im folgenden wird daher untersucht, ob und wie diese Ideen grafisch vermittelt werden.

Gegenstand der Untersuchung sind ausgewählte Architektur–Patterns, zu denen es mehrere Darstellungen gibt. Für diese werden betrachtet:

- Die erste Veröffentlichung in Form eines Artikels für einen Pattern–Workshop (PLoP)
- Die überarbeitete Fassung für ein Pattern–Buch
- Eventuell eine weitere Bearbeitung, oft als Bestandteil eines Pattern–Systems (Pattern Language)

5.1 Einsatz von Grafiken in Pattern–Beschreibungen

Eine Pattern–Beschreibung folgt in ihrer Struktur üblicherweise einer Pattern–Form, wie in Abschnitt 3.7 beschrieben. Grafiken, Diagramme und Code–Stücke tauchen in den folgenden Abschnitten auf:

Problem — Um das Problem zu verdeutlichen, wird am Anfang oft ein Beispiel gebracht, das durch eine *Clip–Art–Grafik* illustriert wird.

Lösung — Die Lösung wird in den meisten Patterns auch mit Diagrammen dargestellt. Hierbei ist eine Unterteilung in *Structure, Dynamics* und *Implementation* häufig anzutreffen:

- Structure — Die Komponenten der Lösung werden mit einem *Klassendiagramm* und in POSA1/2 auch mit *CRC–Cards*[1] dargestellt.
- Dynamics — Für Abläufe werden überwiegend *Sequenzdiagramme*, manchmal unterstützt durch *Zustandsdiagramme*, verwendet.
- Implementation — In der schrittweisen Anleitung zur Umsetzung der Lösung tauchen oft *Code–Stücke* auf.

Beispiel — Das Beispiel vom Anfang wird nun mit der vorgestellten Lösung gezeigt, wobei neben einer entsprechend erweiterten Clip–Art–Grafik auch Code–Stücke gezeigt werden.

Die Darstellung von Architektur–Patterns in der Literatur unterscheidet sich kaum von der Darstellung von Design–Patterns.

5.2 Eine Auswahl von Original–Darstellungen

Aus Platzgründen sollen an dieser Stelle nur ein paar typische Vertreter für die grafische Darstellung von Patterns in der Literatur gezeigt werden.

Shaw und Garlan veröffentlichten in ihrem Buch „Software Architectur“ [SG96] eine Reihe von Architektur–Stilen, die Shaw auf einem Pattern–Workshop [Sha96] in sehr kompakter Form und mit einer eigenen Notation präsentierte, die von der der Design Patterns deutlich abweicht. Zu zwei dieser Patterns, nämlich PIPES AND FILTERS und BLACKBOARD (Beschreibung in Abschnitt 4.2.2), werden spätere Darstellungen, die entweder UML oder eine Objekt–Notation verwenden, gezeigt.

Für den Bereich der Concurrency Patterns stehen die Vertreter HALF–SYNC / HALF–ASYNC und LEADER / FOLLOWERS (Beschreibungen in den Abschnitten 4.2.3.4 und 4.2.7.1). Der Bereich wurde gewählt, weil hier eine grafische Darstellung nebenläufiger Tasks erforderlich ist.

[1] CRC–Cards (Class–Responsibility–Collaborators) wurden von Beck und Cunningham als Mittel zur Unterstützung objektorientierten Denkens in der Lehre eingeführt [BC89].

5.2.1 „Some Patterns for Software Architectures“ (Shaw 1996)

Mary Shaw stellt in ihrem Beitrag „Some Patterns for Software Architectures“ [Sha96] Architektur–Patterns[2] vor, die die Strukturierung einer Anwendung betreffen („The purpose of each of these patterns is to impose an overall structure for a software system or subsystem [...]“). In der Beschreibung wird das Component–Connector–Modell verwendet: Shaw und Garlan verwenden die Begriffe *Component* für Systemelemente, die Information verarbeiten, und *Connector* für die Interaktion zwischen diesen Elementen.[3]

Folgende Architektur–Patterns werden in diesem Aufsatz vorgestellt:

Architektur–Pattern	**Kurzbeschreibung (Abbildung auf Komponenten und Konnektoren)**
PIPELINE	(später: PIPES AND FILTERS) Die Verarbeitung eines Datenstroms wird durch Filterkomponenten, die wegen ihrer einheitlichen Schnittstellen beliebig hintereinander geschaltet werden können, geleistet. Die Konnektoren leiten Datenströme, beispielsweise UNIX pipes.
DATA ABSTRACTION	Kapselung durch Abstract Data Types (Komponenten), Datenzugriff durch Prozeduraufruf (Konnektor).
COMMUNICATING PROCESSES	Nebenläufige kommunizierende Prozesse stellen die Komponenten dar, die Nachrichten, die zwischen ihnen ausgetauscht werden, die Konnektoren.
IMPLICIT INVOCATION	(Event–driven communication) Publish–subscribe–Mechanismen zwischen Event–auslösenden Komponenten ermöglichen indirekte Prozeduraufrufe (Konnektoren).
REPOSITORY	Shared memory (als spezielle Komponente) für mehrere Komponenten, die auf den Daten Berechnungen durchführen. Als Konnektoren werden hier die Datenzugriffe, direkt oder indirekt, genannt.
INTERPRETER	Abwickler für Programme. Die Komponenten sind der Interpreter (Prozessor) sowie die Speicher für das Programm, den Abwicklungszustand und die Daten, auf denen der Interpreter laut Programm operieren soll. Konnektoren sind hier Datenzugriffe und Prozeduraufrufe.
MAIN PROGRAM AND SUBROUTINES	Die Aufteilung eines Programms in Unterprogramme (sic!). Komponenten sind Prozeduren und explizit sichtbare Daten, Konnektoren die Prozeduraufrufe und gemeinsame Nutzung von Daten.
LAYERED ARCHITECTURE	Hierarchie von Modulschichten, in jeder Schicht gibt es bestimmte Abstraktionen und Zuständigkeiten. Ein Zugriff ist nur über die Schnittstellen auf die nächstliegende Schicht möglich. Die Komponenten sind hier zusammengesetzte Einheiten (composites), meist Prozeduren. Konnektoren sind meist Prozeduraufrufe.

[2] Shaw und Garlan verwenden in [SG96] die Begriffe *Pattern* und *Stil* synonym. Tatsächlich ist der Begriff des Stils angemessener, auch im Sinne der Definition in Abschnitt 3.4.1.2. Es ist ein „guter Stil“, seine Software in Unterprogramme zu gliedern und deren Aufrufbeziehungen nach Möglichkeit streng hierarchisch zu gliedern. Für ein Pattern ist das aber zu wenig, denn hier fehlt das konkrete Problem, das mit dieser Lösung behoben werden könnte.

[3] „[...] to treat an architecture of a specific system as a collection of computational components — or simply *components* — together with a description of the interactions among these components — the *connectors*.“ [SG96, S. 20]

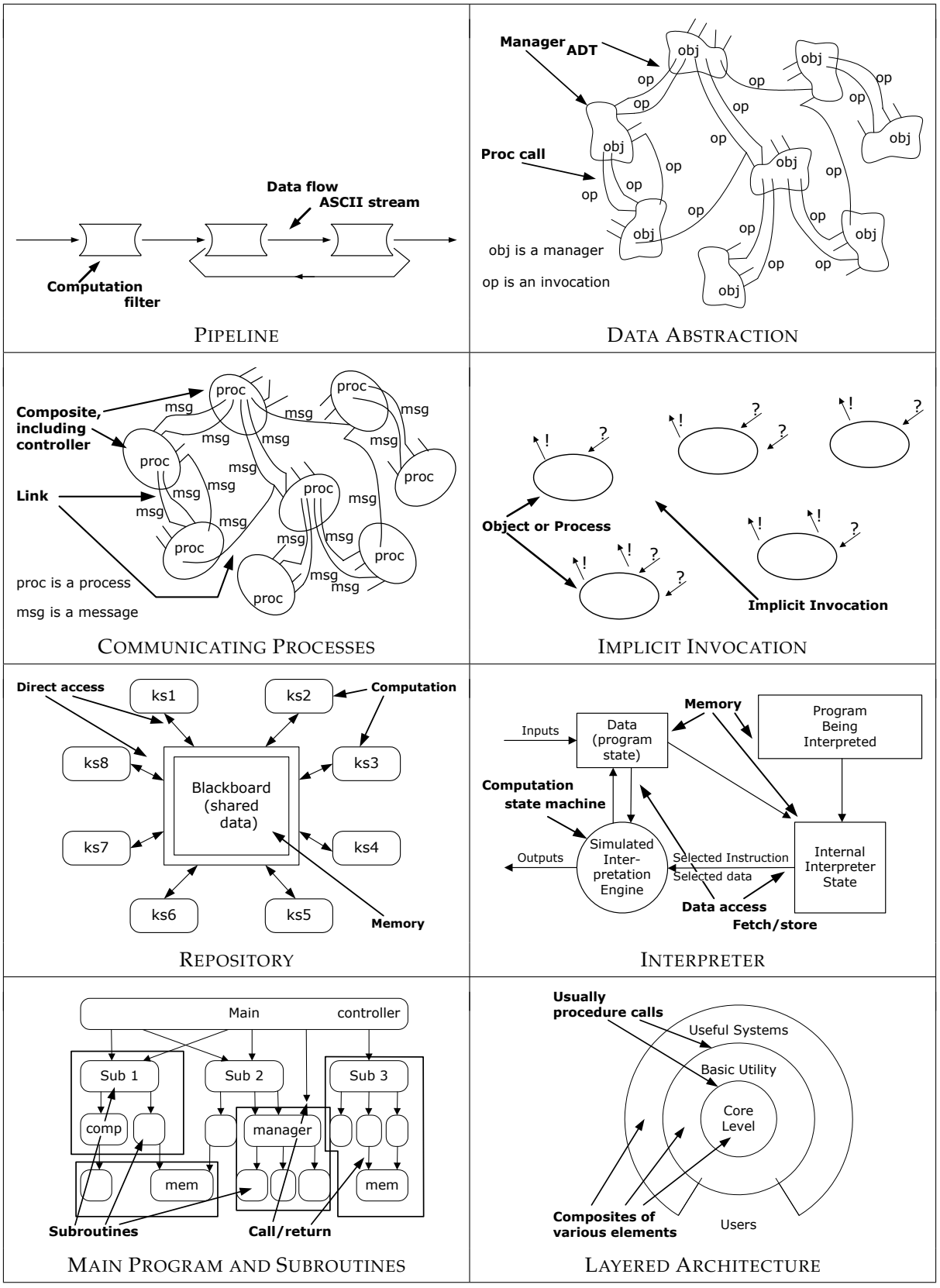


Abbildung 5.1: Original–Darstellung der Architektur–Patterns von Shaw [Sha96, S. 259-266]

Mit einer Seite pro Pattern benutzt Mary Shaw eine sehr knappe Form. Dabei verwendet sie für jedes Pattern ein Bild, um dessen Einprägsamkeit zu verstärken. Die Struktur eines Patterns ist sehr streng gegliedert in Problem – Context – Solution – Significant Variants – Examples. Die Lösung besteht aus der Liste System Model – Components – Connectors – Control Structure und dem Bild. Die Abbildung 5.1 zeigt die Bilder zu allen vorgestellten Patterns.

Bewertung

Was an den Bildern auffällt, ist ihre uneinheitliche Notation. Die Abbildungen zu REPOSITORY und INTERPRETER verwenden noch eine einheitliche Notation — Rechtecke für Speicher, Kreise bzw. Rechtecke mit abgerundeten Ecken für aktive Komponenten. Die Darstellung von DATA ABSTRACTION und COMMUNICATING PROCESSES wirkt überfrachtet, weil jeder Knoten und jede Kante redundant beschriftet ist. Wenig hilfreich für das Verständnis sind nach Meinung des Autors die Abbildungen IMPLICIT INVOCATION und LAYERED ARCHITECTURE.

5.2.2 Das Architektur–Pattern PIPES AND FILTERS

5.2.2.1 Darstellung von Shaw (1996)

Shaw stellt in Abbildung 5.1 nur die Filter dar. Pipes tauchen nicht auf, auch im Text werden als Konnektoren nur die Datenströme bezeichnet, welche beispielsweise über UNIX pipelines transportiert werden.

5.2.2.2 Darstellung in POSA 1 (1996)

Buschmann et al. benutzen in „Pattern oriented Software Architecture“ [BMR+96] UML–Klassendiagramme und Class–Responsibility–Collaborator–Cards (CRC Cards) für die Struktur und UML–Sequenzdiagramme und UML–Zustandsdiagramme für das Verhalten.

Für das Pattern PIPES AND FILTERS werden für die Struktur nur CRC Cards benutzt (Abbildung 5.2), während ein UML–Klassendiagramm ein Anwendungsbeispiel zeigt. Zum Verhalten werden mehrere Szenarien für „push“ und „pull“–Filterketten gezeigt; Abbildung 5.3 zeigt eine Kombination aus „push/pull“: Aktive Filterelemente, die selbst die Eingabedaten holen und die Ausgabedaten wegschreiben, sowie puffernde Pipes.

5.2.2.3 Darstellung von Buschmann/Henney (2002)

Frank Buschmann und Kevlin Henney haben 2002 für eine Pattern Language Architektur–Patterns in kompakter Form dargestellt [BH02]. Dabei haben sie eine eigene Notation eingeführt, in der in Abbildung 5.4 das PIPES AND FILTERS Pattern dargestellt ist. Ein Element ist umrandet von einem schattierten Vieleck und hat weiß gefärbte, kleinere Vielecke, die auf einer Seite heraus stehen und die üblicherweise mit Methodennamen beschriftet sind. Kanten mit Pfeilen stehen für Aufrufe oder Datenfluss. Die Farbe der äußeren Vielecke hat ebenfalls eine Bedeutung — hellgrau bedeutet Datenspeicher, dunkelgrau Datenverarbeiter.

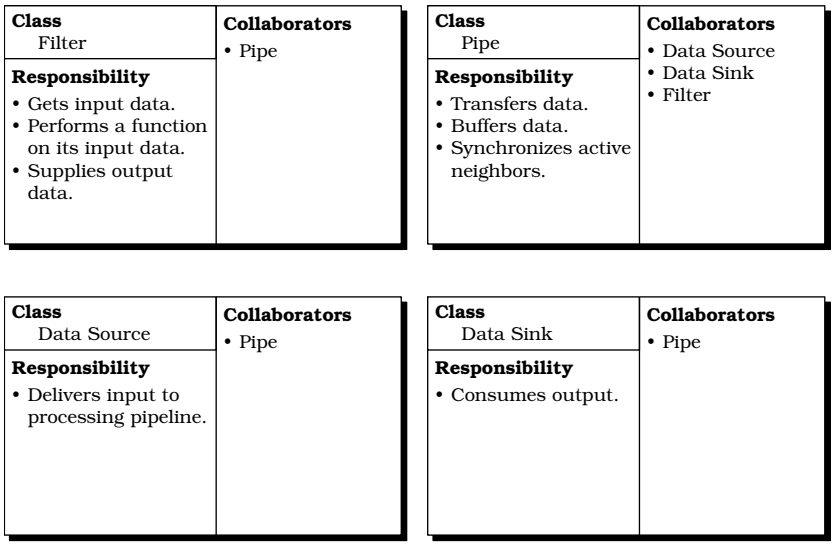


Abbildung 5.2: PIPES AND FILTERS nach POSA 1: CRC Cards [BMR+96, S. 56]

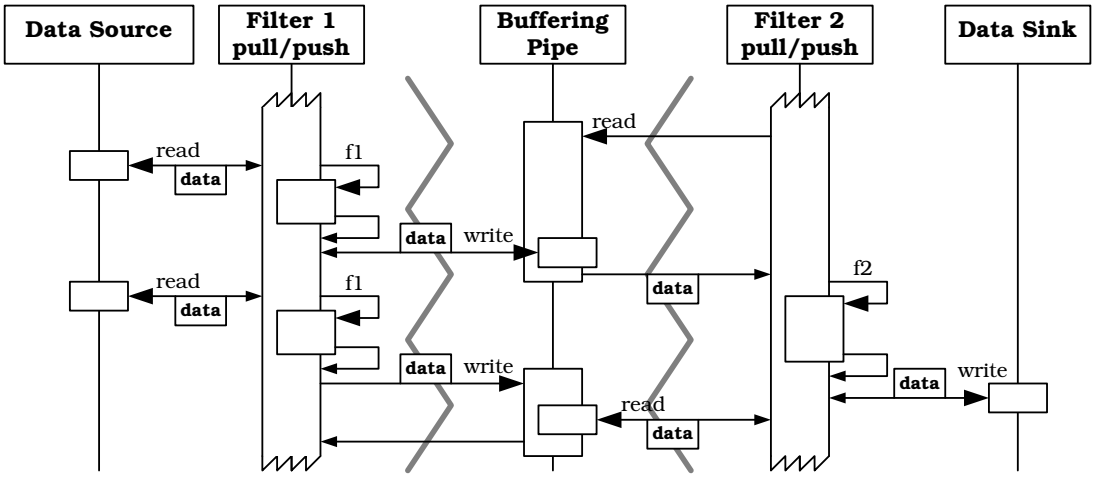


Abbildung 5.3: PIPES AND FILTERS nach POSA 1: Sequenzdiagramm [BMR+96, S. 59]

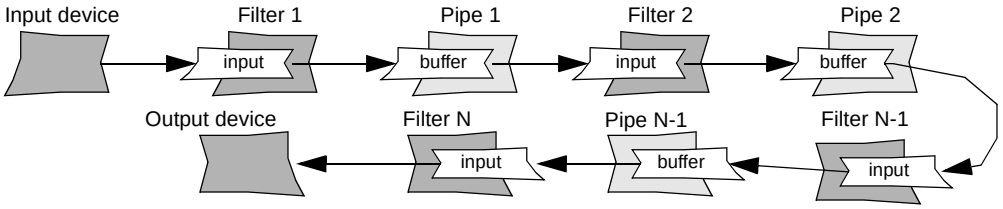


Abbildung 5.4: PIPES AND FILTERS nach Buschmann / Henney [BH02, S. 20]

5.2.3 Das Architektur–Pattern BLACKBOARD

5.2.3.1 Darstellung von Mary Shaw (1996)

In [Sha96] illustriert Shaw das Pattern REPOSITORY mit einem Aufbaustrukturbild (siehe Abbildung 5.1). Dabei stehen die abgerundeten Knoten für Verarbeitungseinheiten und der eckige für den gemeinsamen Speicher. Die Aussage des Bildes beschränkt sich darauf, dass mehrere „ks“ (Knowledge Sources) direkten Zugang zu gemeinsamen Daten (Blackboard) haben und Berechnungen durchführen.

5.2.3.2 Darstellung in POSA 1 (1996)

Im Gegensatz zum REPOSITORY von Shaw führen Buschmann et al. in [BMR+96] für das Pattern BLACKBOARD ein Element zur Koordination ein: „Control“. Zusätzlich zu den CRC Cards (Abbildung 5.5) benutzen sie ein UML–Klassendiagramm (Abbildung 5.6). Wegen der Verwendung objektorientierter Technologie ist ein direkter Zugriff auf die Daten nicht vorgesehen, hierfür sind die Zugriffsmethoden „inspect“ und „update“ zu benutzen.

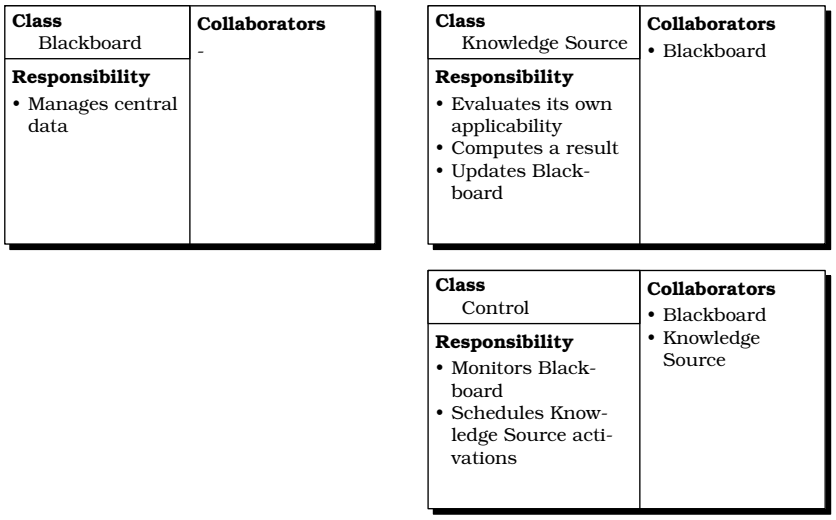


Abbildung 5.5: BLACKBOARD nach POSA1: CRC Cards [BMR+96, S. 77f]

Der Ablauf wird an einem Sequenz–Diagramm für ein Spracherkennungs–Beispiel gezeigt (hier nicht dargestellt).

5.2.3.3 Darstellung von Buschmann/Henney (2002)

Buschmann und Henney stellen in Abbildung 5.7 Aufbau und Ablauf kombiniert dar. Die Komponenten haben wieder verschiedene Farben, nämlich dunkel für aktiv, hell für passiv. An den Pfeilen stehen Zahlen in Kreisen, um die Reihenfolge beim Ablauf darzustellen. Zusätzlich wird im oberen Vieleck die Operation „run“ der Komponente „Control“ verfeinert, indem in Pseudo–Code deren Schritte gezeigt werden.

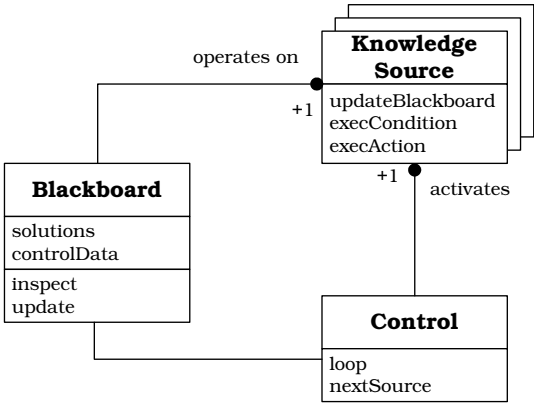


Abbildung 5.6: BLACKBOARD nach POSA1: Klassendiagramm [BMR+96, S. 79]

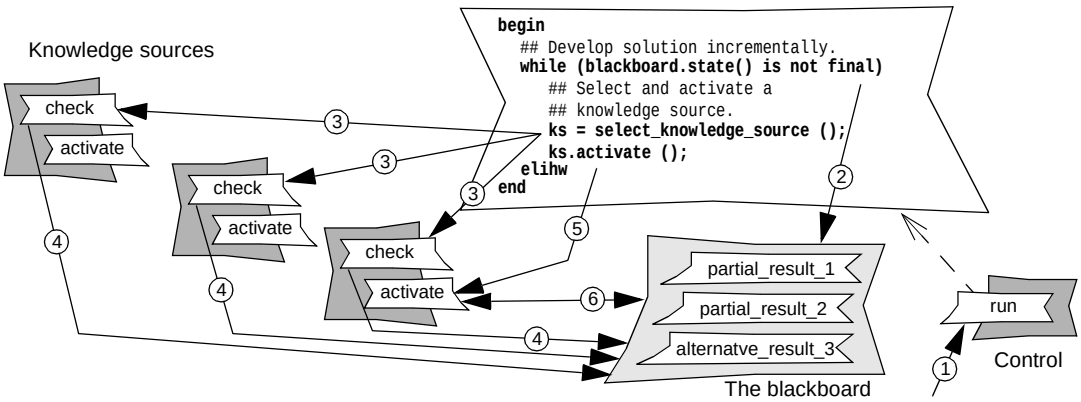


Abbildung 5.7: BLACKBOARD nach Buschmann / Henney [BH02, S. 22]

5.2.4 Das Architektur–Pattern HALF–SYNC / HALF–ASYNC

5.2.4.1 Darstellung von Schmidt auf der PLOPD2 (1996)

Dieses Pattern tauchte zum ersten Mal als „Architectural Pattern for Efficient and Well–Structured Concurrent I/O“ auf dem zweiten Pattern–Workshop PLOP auf [SC96].

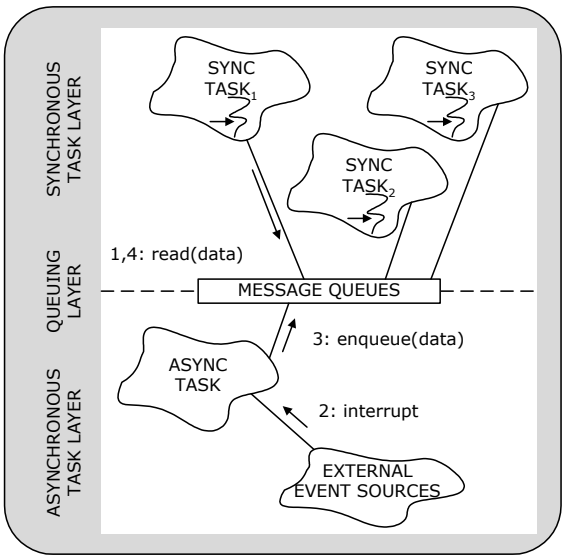


Abbildung 5.8: HALF–SYNC / HALF–ASYNC nach Schmidt: Strukturdiagramm [SC96]

Das in Abbildung 5.8 wiedergegebene Diagramm zeigt die beteiligten Tasks, die Queues sowie die Event Sources, sowie ihre Zuordnung zu den Layers. Die Pfeile und Zahlen der Beschriftung deuten den Ablauf an.

5.2.4.2 Darstellung in POSA2 (2000)

An der Darstellung dieses Patterns in POSA2 [SSRB00, S. 421-445] wird besonders deutlich, dass die Benutzung von Diagrammtypen aus dem objektorientierten Entwurf zur Darstellung von Multitasking–Systemstrukturen nicht geeignet ist.

Mit den CRC–Cards in Abbildung 5.9 werden die Beteiligten vorgestellt. Das Klassendiagramm aus Abbildung 5.10 beschreibt die gleiche Aufbaustruktur wie Abbildung 5.8, wobei jetzt alle Knoten die gleiche Form (Rechteck) haben und die Verwendung gestrichelter Kanten und der spitzen Klammern „<< >>“ um die Kantenbeschriftung herum die Lesbarkeit beeinträchtigen.

Das einzige Sequenzdiagramm dieser Pattern–Beschreibung (Abbildung 5.11) ist etwas unglücklich gewählt, weil kein Unterschied zwischen asynchronem und synchronem Service sichtbar ist (der synchrone Service blockiert nämlich eigentlich beim Lesen aus der Queue, bis der asynchrone etwas in die Queue schiebt).

Das HALF–SYNC / HALF–REACTIVE Pattern wird nur als Variante mit einem kurzen Textabschnitt erwähnt. Eine Darstellung dieses Patterns, die übrigens keine OO–Notation verwendet, ist an einer anderen Stelle, nämlich in der Pattern–Beschreibung zu LEADER / FOLLOWERS zu finden (Abbildung 5.12).

Class Synchronous Task Layer	*Collaborator*
Responsibility • Executes high-level processing tasks synchronously	• Queuing Layer

Class Asynchronous Task Layer	*Collaborator*
Responsibility • Executes low-level processing tasks asynchronously	• Queuing Layer

Class Queuing Layer	*Collaborator*
Responsibility • Provides a buffering between the synchronous task layer and the asynchronous task layer	• Asynchronous Task Layer • Synchronous Task Layer

Class External I/O Source	*Collaborator*
Responsibility • Generates events received and processed by the asynchronous task layer	• Asynchronous Task Layer

Abbildung 5.9: HALF–SYNC / HALF–ASYNC nach POSA2: CRC–Cards [SSRB00, S. 426f]

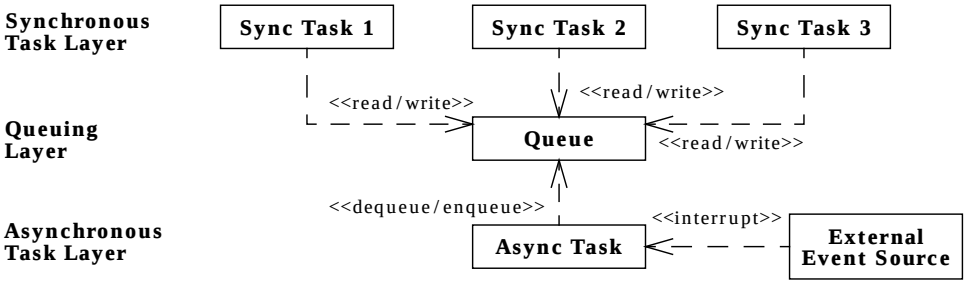


Abbildung 5.10: HALF–SYNC / HALF–ASYNC nach POSA2: Klassendiagramm [SSRB00, S. 427]

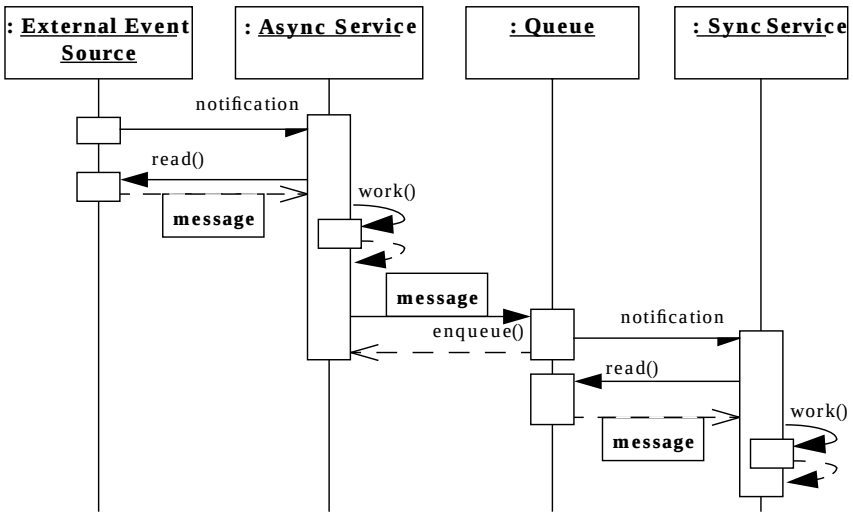


Abbildung 5.11: HALF–SYNC / HALF–ASYNC nach POSA2: Sequenzdiagramm [SSRB00, S. 428]

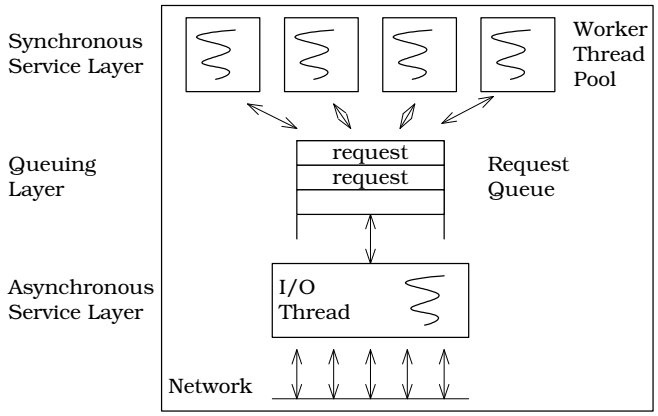


Abbildung 5.12: HALF–SYNC / HALF–REACTIVE nach POSA2 (bei: LEADER / FOLLOWERS) [SSRB00, S. 448]

5.2.4.3 Darstellung bei Buschmann/Henney (2002)

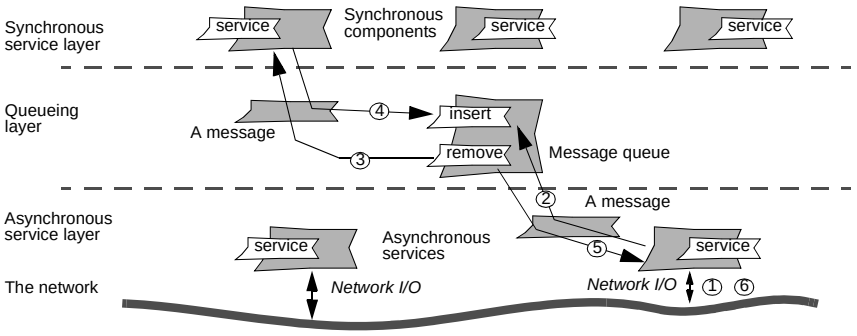


Abbildung 5.13: HALF–SYNC / HALF–ASYNC nach Buschmann / Henney [BH02, S. 40]

Die Darstellung des HALF–SYNC / HALF–ASYNC Patterns bei Buschmann / Henney in Abbildung 5.13 ähnelt wieder der ursprünglichen (Abbildung 5.8). Die Pfeile zeigen die Datenflussrichtung an. Die Message Queue wurde nicht als passiv (helle Farbe) dargestellt, da nur über die Methoden „insert“ und „remove“ darauf zugegriffen wird.

5.2.5 Das Architektur–Pattern LEADER / FOLLOWERS

5.2.5.1 Darstellung in POSA 2 (2000)

Das Pattern zu LEADER / FOLLOWERS in POSA 2 [SSRB00, S. 447–474] geht zurück auf eine Veröffentlichung von Douglas Schmidt et al. [SOK+00], in dem die gleiche Darstellung wie später in POSA 2 verwendet wird.

Als Einstieg wird begründet, warum die Alternative, das HALF–SYNC / HALF–REACTIVE Pattern, dargestellt in einem Aufbaubild (siehe Abbildung 5.12), nicht immer geeignet ist.

Zunächst werden die Beteiligten im Text und durch CRC–Cards beschrieben (siehe Abbildung 5.14). Die Beziehungen zwischen den Beteiligten werden danach durch ein Klassen–Diagramm (siehe Abbildung 5.15) illustriert.

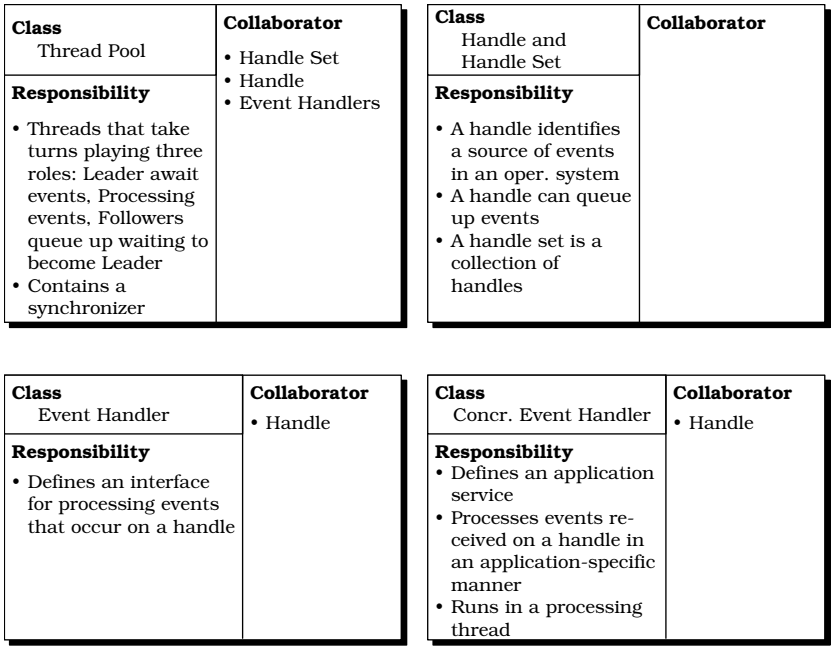


Abbildung 5.14: LEADER / FOLLOWERS nach POSA2: CRC Cards [SSRB00, S. 452f]

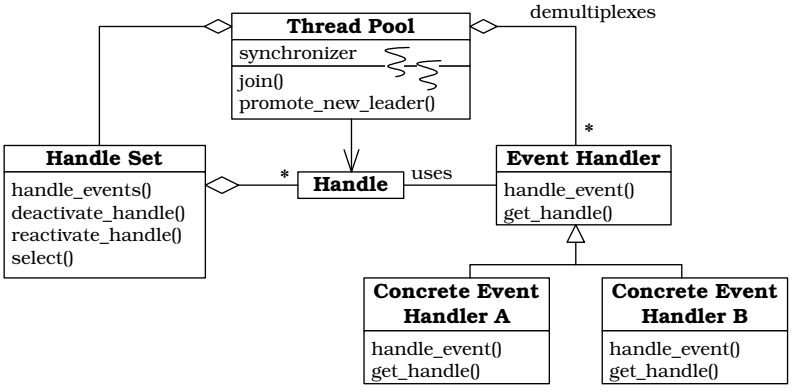


Abbildung 5.15: LEADER / FOLLOWERS nach POSA2: Klassendiagramm [SSRB00, S. 454]

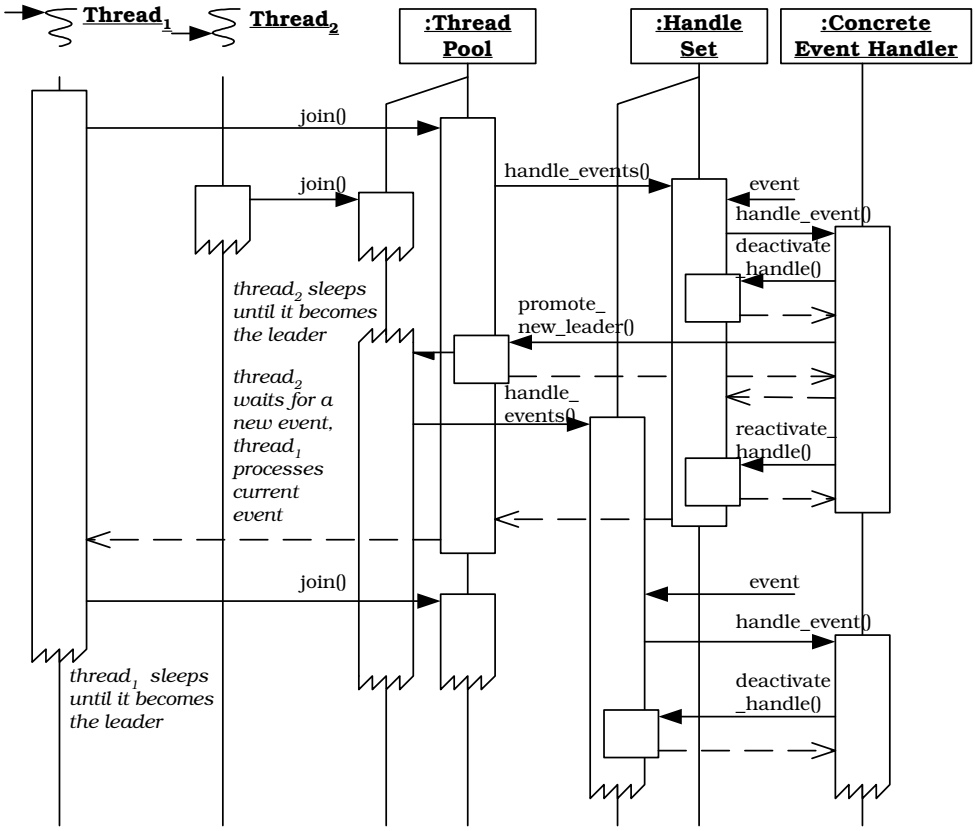


Abbildung 5.16: LEADER / FOLLOWERS nach POSA2: Sequenzdiagramm [SSRB00, S. 455]

Im Abschnitt „Dynamics“ geht es darum, wie die Beteiligten zusammenspielen. Zur Illustration dient ein Sequenzdiagramm (Abbildung 5.16). Dabei bekommen Threads eigene Swimlanes, obwohl Threads und Objekte völlig verschiedene Dinge sind.

Zur Erklärung des Diagramms: Mit `join()` bewirbt sich ein Thread um die Rolle als Leader, der mit dem Aufruf von `handle_events()` am Handle Set auf Events von außen wartet. Falls ein Event angezeigt wird, wird der Concrete Event Handler von diesem Thread ausgeführt, der mit `promote_new_leader()` die Leader–Rolle an den nächsten wartenden Thread abgibt. Nachdem das Event behandelt wurde, reiht sich der Thread mit `join()` wieder in die Reihe der Followers ein.

Die Events von außen werden durch Pfeile aus dem Nichts am Handle Set dargestellt. Dass der Concrete Event Handler einmal von Thread 1 und dann von Thread 2 ausgeführt wird, ist nur durch Nachvollziehen der Aufrufe erkennbar.

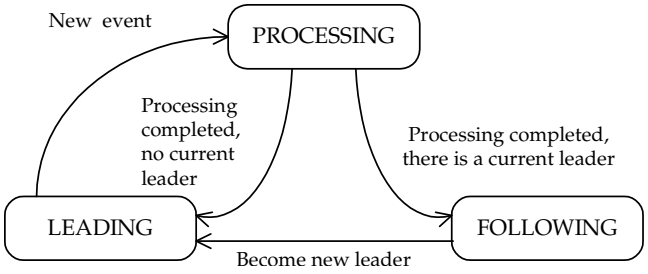


Abbildung 5.17: LEADER / FOLLOWERS nach POSA2: Zustände eines Threads [SSRB00, S. 456]

Da in diesem Pattern die Threads ihre Rollen wechseln, verdeutlicht ein Zustandsdiagramm (Abbildung 5.17) diese Tatsache. Leider taucht es erst hinter den andern Diagrammen auf.

5.2.5.2 Darstellung von Buschmann/Henney (2002)

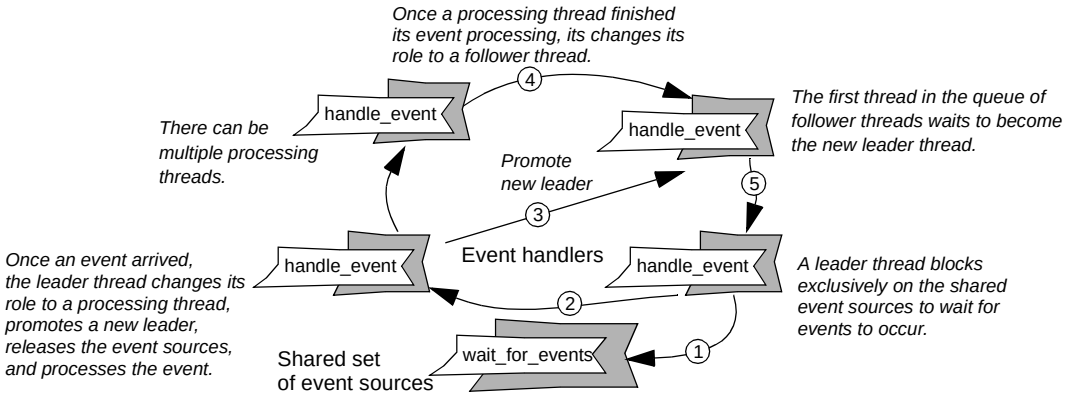


Abbildung 5.18: LEADER / FOLLOWERS nach Buschmann / Henney [BH02, S. 42]

Abbildung 5.18 zeigt die Darstellung von Buschmann und Henney. Es werden die Rollen und Zustände eines Threads dargestellt, während unklar bleibt, in welcher Umgebung diese sich befinden. Die Zugriffe (z.B. 1: wait_for_events) und Zustandsübergänge sind durch Pfeile mit Zahlen markiert.

5.3 Bewertung der grafischen Darstellungen

5.3.1 Zweckmäßige Beschreibung von Systemstrukturen

Diagrammtypen aus dem objektorientierten Entwurf

Wenn es um Systemstrukturen geht, ist die Verwendung von Diagrammtypen aus dem objektorientierten Entwurf in POSA1/2 [BMR+96, SSRB00] nur wenig geeignet, das Konzept eines solchen Patterns wiederzugeben:

Aus den CRC–Cards lässt sich zwar eine Struktur ermitteln, diese ist aber grafisch nicht sichtbar; außerdem gibt es hier Elemente, die nur eine gemeinsame Schnittstelle ermöglichen sollen, zur Laufzeit aber nicht auftauchen (Beispiel: Die Klasse „Event Handler“ in Abbildung 5.14).

Die Klassendiagramme sind zur Darstellung der Systemstrukturen wenig geeignet, weil die dort wichtigen Sachverhalte besser an exemplarischen Aufbaustrukturen gezeigt werden können und weil Konzepte der Software–Strukturen, beispielsweise Vererbung, hier nahezu keine Rolle spielen.

Sequenzdiagramme zeigen auch, welche Komponenten zur Laufzeit beteiligt sind. Es gibt sie aber nur in einer Granularität, nämlich zur Darstellung von Prozeduraufrufen zwischen oder innerhalb von Objekten. Außerdem zeigen sie nur exemplarische Abläufe.

Die Verwendung von Klassendiagrammen im Abschnitt „Structure“ einer Pattern–Beschreibung trägt dazu bei, dass es meist keine eigenen Bilder gibt, die die Laufzeit–Entitäten mit ihren Verbindungen zeigen, obwohl diese für das Verständnis der (Beispiel–) Abläufe hilfreich wären. Interessanterweise bildet das Design–Patterns–Buch von Gamma et al. [GHJV94] eine Ausnahme, denn hier werden meist Objekt– und Datenstrukturen explizit dargestellt.

Andere Diagrammtypen

Nützlicher für das Verständnis der Systemstrukturen sind die freieren Notationen bei Buschmann / Henney [BH02] und die Clip–Art–Grafiken, weil sie meist Aufbaustrukturen zeigen.

Völter, Schmid und Wolff schreiben in ihrem Buch „Server Component Patterns“, dass die UML für Struktur–Diagramme nicht viel Unterstützung bietet und dass die Verwendung von UML um jeden Preis über Stereotypen und Kommentare nicht hilfreich ist.[4]

Das legt den Schluss nahe, dass Patterns für Bereiche, in denen es nicht um Probleme im Zusammenhang mit objektorientierter Programmierung oder Datenstrukturierung geht, nicht unbedingt primär mit den OO–typischen Diagrammtypen, beispielsweise aus der UML, dargestellt werden sollten.

Nach Meinung des Autors tragen gerade bei Architektur–Patterns die Darstellungsmittel für (objektorientierte) Software–Strukturen nicht zum besseren Verständnis der Patterns bei. Sicher lassen sich mit UML–Klassendiagrammen beliebige Strukturen darstellen, weil Klassendiagramme letztendlich erweiterte Entity–Relationship–Diagramme (ERD) sind. Das Problem dabei ist aber, dass der Leser wegen der Notation als Klassendiagramm auch zuerst Aussagen über programmiersprachliche Klassen erwartet.

[4] „For many of the structural diagram [...] we use non–UML notations, because UML does not provide very much help here, and forcing everything to be UML by using stereotypes and comments does not help either.“ [VSW02, S. xviii]

Erfahrungen aus der Architekturentwicklung für ein großes Software–Projekt bei Alcatel zeigen, dass die Beteiligten die verschiedenen Notationen von Systemstruktur–Diagrammen (z.B. Aufbaubilder) und Software–Struktur–Diagrammen (z.B. UML Klassendiagrammen) als große Erleichterung für das Verständnis sehen — man weiß sofort, womit man es gerade zu tun hat.

5.3.2 Grafiken als Merkhilfe

Ist der Detailgrad eines Diagramms zu hoch, kann man es nur schwer erfassen. Ein einfaches Aufbaubild, das die Idee des Patterns vermittelt, ist daher besser als ein Bild, das bereits Details der Implementierung darstellt. Das gilt besonders dann, wenn es mehrere Möglichkeiten der Implementierung gibt.

Beispiel: Das Klassendiagramm von LEADER / FOLLOWERS (Abbildung 5.15) zeigt eine mögliche Aufteilung bei objektorientierter Implementierung. Stellt man die Idee aber in einem Aufbaubild wie in Abbildung 4.4 dar, dann ist viel leichter zu erkennen, dass beispielsweise auch das Preforking Multi–Processing–Modell des Apache HTTP Servers diesem Prinzip folgt, nur dass hier Prozesse statt Threads und ein Mutex als Synchronizer verwendet werden (näheres dazu in Abschnitt 7.3.2.1).

Die Einprägsamkeit eines Patterns hängt nach Meinung des Autors unmittelbar von der Darstellung des Patterns ab, insbesondere der grafischen Darstellung. Ein Diagramm sollte klar verständlich sein und damit die Idee des Patterns merkbar machen. Innerhalb eines Systems von Patterns sollten sowohl eine einheitliche Notation als auch ein einheitliches Layout verwendet werden, um die Unterschiede und Gemeinsamkeiten zwischen den verschiedenen Patterns hervorzuheben.

Kapitel 6

Konzeptionelle Patterns für Systemstrukturen

6.1 Motivation

In der Architektur–Phase, in der ein System geplant und Aufgaben verteilt werden, sind nicht nur Architektur–Patterns relevant. Da das gewollte System über seine Systemstrukturen auf mehreren Abstraktionsebenen modelliert und verfeinert wird, sind alle Erfahrungen in Pattern–Form nützlich, die Systemstrukturen behandeln. Für viele der in Abschnitt 4.2 vorgestellten Architektur– und Design–Patterns gilt, dass sie in Problem, Kontext und Lösung Systemstrukturen beschreiben. Im Gegensatz zu vielen Design–Patterns für den objektorientierten Entwurf, beispielsweise von Gamma et al. [GHJV94], sind ihre Auswirkungen auf Software–Strukturen daher nur gering.

Die anwendungsstrukturierenden Eigenschaften der Architektur–Patterns, wie sie in Kapitel 4 vorgestellt wurden, spielen in der späten Architektur–Phase eine Rolle, nämlich wenn der Entwurf technisch konkret wird und bevor der Übergang zu Software–Strukturen und das Abgrenzen von Arbeitspaketen für die Entwickler ansteht.

In der Pattern–Literatur wird, wie in Kapitel 5 gezeigt, bei der Darstellung der Schwerpunkt auf Software–Strukturen gelegt, die zur Darstellung der Lösungsidee im Bereich der Systemstrukturen oft wenig geeignet sind.

In Abschnitt 4.7 wurde zudem gezeigt, dass Architektur–Patterns schwer zu kategorisieren sind, weil bei dieser Art von Patterns die Lösung mehrere Bestandteile hat, die aber auf unterschiedliche Abstraktionsebenen und sogar in verschiedene Themenbereichen eingeordnet werden müssen.

Damit ergeben sich folgende Konsequenzen:

1. Patterns, die Lösungen im Bereich der Systemstrukturen beschreiben und die für die Architektur–Phase nützlich sein können, sollten als solche erkennbar sein. Für diese wird die Kategorie der *konzeptionellen Patterns* eingeführt.

2. Die Systemstrukturen müssen explizit grafisch dargestellt werden, um die Lösungsidee eines solchen Patterns zu verdeutlichen. Als Darstellungsmittel bietet sich die Notation der Fundamental Modeling Concepts (FMC, siehe Abschnitt 2.4) an, da diese für die Beschreibung von Systemstrukturen entwickelt wurden.

3. Ein Architektur–Pattern sollte aufgetrennt werden in ein anwendungsstrukturierendes Pattern und ein Design–Pattern für die Infrastruktur.

6.2 Konzeptionelle Patterns als Kategorie

6.2.1 Definition

Ein *konzeptionelles Pattern* im Bereich informationeller Systeme zeichnet sich dadurch aus, dass sowohl Problem als auch Lösung im Bereich der Systemstrukturen zu finden sind. Die Lösung beschreibt Akteure, ihre Kommunikation und ihr Verhalten. Daraus lassen sich wiederum Schnittstellen für die Software ableiten. Die Lösung ist daher unabhängig von einer bestimmten Programmiertechnik und eventuell sogar von der Realisierung durch Software.

Diese Definition konzeptioneller Patterns geht deutlich weiter als die von Riehle und Züllighoven, die sich auf die Analyse–Phase bezieht und durch die Einschränkung der Begriffe auf die Anwendungs–Domäne keine technischen Systemstrukturen zulässt.[1] Den Conceptual Patterns werden hier nämlich Design Patterns gegenübergestellt, mit denen das konzeptionelle Modell auf das Design–Modell übertragen werden soll, in dem die Beschränkungen durch eine Realisierung mit Software berücksichtigt werden müssen.[2] Design–Patterns werden mit Darstellungsmitteln für Software–Konstrukte beschrieben.[3]

Nach der oben vorgestellten Definition müssen Problem und Lösung im Bereich der Systemstrukturen zu finden sein, wobei die Anwendung des Patterns in der Systemstruktur nachvollziehbar sein muss. Um nun für ein existierendes Pattern zu entscheiden, ob es sich um ein konzeptionelles Pattern handelt, prüft man zuerst, ob eine sinnvolle Modellierung der Systemstrukturen möglich ist; hat die Anwendung des Patterns Auswirkungen auf die Systemstruktur, handelt es sich um ein konzeptionelles Pattern.

In Anlehnung an Abschnitt 2.1 seien hier ein paar Kriterien genannt, die für Systemstrukturen erfüllt sein müssen:

- Das Modell umfasst mehrere Akteure, die potentiell nebenläufig agieren können.
- Die Akteure kommunizieren miteinander über Kanäle oder Speicher.
- Eine Abbildung der Akteure auf Menschen ist zumindest denkbar.

Viele Patterns für objektorientierte Programmierung (z.B. aus [GHJV94]) beschreiben Lösungen, die in der Aufbaumodellierung auf die Nutzung von Schnittstellen oder die Strukturierung von Speicher durch Objektstrukturen abstrahiert werden können.[4] Genauso kann man viele durch Algorithmen beschriebene Operationen in der Systemstruktur auf Manipulationen von Speicherstrukturen reduzieren.

[1] „A *conceptual pattern* is a pattern whose form is described by means of the terms and concepts from an application domain.“ „[...] conceptual patterns comprise both a kind of world view and a guideline for perceiving, interpreting and changing the world.“[RZ96]

[2] „[...] we need a model which relates to the conceptual models of the application domain but takes into account the need for reformulating this conceptual model in terms of the formal restrictions of a software system“[RZ96]

[3] „A design pattern is a pattern whose form is described by means of software design constructs, for example objects, classes, inheritance, aggregation and use–relationships.“[RZ96]

[4] Das Problem der Aufbaumodellierung objektorientiert programmierter Systeme wird in [TG03] vertieft.

Die Wahl des angemessenen Abstraktionsgrads ist ein generelles Problem der Modellierung, insbesondere bei Systemstrukturen. Im Fall der konzeptionellen Patterns kann von einem angemessenen Abstraktionsgrad dann gesprochen werden, wenn in den Systemstrukturen sowohl das Problem als solches erkennbar als auch die Lösung nachvollziehbar ist.

6.2.2 Verschiedene Abstraktionsgrade konzeptioneller Patterns

Konzeptionelle Patterns lassen sich auch in die in Abschnitt 4.4 beschriebenen Abstraktionsgrade einordnen: Sie können auf hoher Ebene *(Allgemeine Konzepte)* angesiedelt sein, auf der eine Realisierung per programmiertem System gleichwertig neben anderen Möglichkeiten steht, oder auf niedrigerer Ebene *(Konzepte für programmierte Systeme)* bereits spezielle Umgebungen in programmierten Systemen voraussetzen, in denen es beispielsweise Multitasking oder eine bestimmte Netzwerk–Schnittstelle gibt. In jedem Fall sind konzeptionelle Patterns unabhängig von Programmiertechniken und Software–Strukturen.

6.2.3 Zusammenhang mit Architektur– und Design–Patterns

Wie in Kapitel 4 gezeigt, ist die Bandbreite der vorgestellten Patterns bezüglich Systemstrukturen groß. Die ursprüngliche Annahme, dass nur Architektur–Patterns Lösungen innerhalb der Systemstrukturen anbieten, während Design–Patterns hauptsächlich Software–Strukturen behandeln, wurde dort widerlegt. Es gibt einige Design–Patterns, die als konzeptionelle Patterns eingestuft werden können, wenngleich der Anteil konzeptioneller Patterns bei den Architektur–Patterns deutlich höher ist.

Mit LAYERS und REFLECTION fallen im Gegenzug zwei Architektur–Patterns der Literatur nicht in die Kategorie der konzeptionellen Patterns. Beide schlagen Lösungen vor, die nur im Zusammenhang mit Programmierung sinnvoll verwendbar sind und die die Nutzung von Aufrufschnittstellen betreffen (siehe auch Abschnitt 4.5). Zwar lassen sich im Fall des REFLECTION Patterns in der Systemstruktur durchaus Schnittstellen und Zugriffe auf Speicher identifizieren, die von diesem Mechanismus Gebrauch machen, doch hat der Einsatz dieses Patterns keine Auswirkung auf die Systemstruktur.

6.3 Grafische Darstellung

Aufbaubilder (Block diagrams) in der Notation der Fundamental Modeling Concepts (FMC, siehe Abschnitt 2.4) sind zur Darstellung von Systemstrukturen auf höheren Abstraktionsebenen ausgelegt.[5] Eine Abbildung auf Realisierungskomponenten der Laufzeitstruktur ist möglich; so ist auf niedriger Ebene beispielsweise ein Akteur auf einen Thread abbildbar.

Ein weiterer Vorteil an Aufbaubildern ist die Möglichkeit, den Rest des Systems beziehungsweise die Umgebung darzustellen, obwohl sie nicht Teil der Lösung, sondern Teil des Problems bzw. Kontexts sind. FMC bietet dabei auf der einen Seite die erforderliche Präzision, auf der anderen Seite die Möglichkeit, Sachverhalte bewusst offen zu lassen und ist damit gut geeignet, Dinge darzustellen, die für das Verständnis wichtig sind, aber nicht an dieser Stelle spezifiziert werden sollen (oder können).

[5] Die FMC–Website `fmc.hpi.uni-potsdam.de` bietet eine Vielzahl von Dokumenten und Veröffentlichungen zur grafischen Darstellung von Systemstrukturen.

Alle Patterns in den Kapiteln 4 und 7 sind mit FMC–Aufbaubildern, teilweise ergänzt um Petri–Netze für die Abläufe, dargestellt.

6.4 Anwendungsgebiete

6.4.1 Unterstützung der Architekturphase

Konzeptionelle Patterns dienen dazu, den Entwurf im Bereich der Systemstrukturen zu unterstützen. Durch die Konzentration auf Systemstrukturen eignen sich konzeptionelle Patterns für einen frühen Einsatz in der Planungsphase, als Kommunikationsmittel, aber auch für die Untersuchung und Bewertung bestehender Systeme.

Es handelt sich bei konzeptionellen Patterns nicht um eine neue Sorte von Patterns, sondern um eine neue Kategorisierung, in die viele bestehende Patterns passen. Eine Überarbeitung ihrer Beschreibung und grafischen Darstellung mit dem Fokus auf die Systemstrukturen, so wie in Abschnitt 4.2 geschehen, führt dazu, dass sie besser in der Architekturphase einsetzbar sind.

Sobald man von Systemstrukturen mit Hilfe von Patterns Software–Strukturen ableiten kann, ist der Begriff *Bridging Patterns* angebracht. Beispielsweise gibt es verschiedene Möglichkeiten, Akteure und Speicher auf Objekte bei Verwendung objektorientierter Technologie abzubilden [TG03]. Bridging Patterns stehen nicht im Fokus dieser Arbeit, sondern stellen ein eigenes Forschungsthema dar.

6.4.2 Einbettung in Pattern Languages

Patterns, die grundlegende Prinzipien zur Anwendungsstrukturierung beschreiben, also Architektur–Patterns, sind isoliert betrachtet häufig nicht hilfreich. Aus diesem Grund werden sie in der Literatur häufig in einem Pattern–System oder einer Pattern Language betrachtet. Das ist umso wichtiger, je kleiner Patterns werden, was bei der vorgeschlagenen Auftrennung von Architektur–Patterns der Fall ist.

Es wird daher vorgeschlagen, Pattern Languages für bestimmte Problembereiche zusammenzustellen, die konzeptionelle und anwendungsstrukturierende Patterns enthalten. Eine derartige Zusammenstellung hat den Vorteil, dass zusammen mit dem Problembereich ein gemeinsamer Kontext, beispielsweise die Umgebung, nur einmal beschrieben werden muss. Es können alternative oder konsekutive Lösungen vorgestellt werden oder auch Anti–Patterns, also nahe liegende, aber nicht funktionierende Lösungen. Der Leitfaden (Guideline) bestimmt die Reihenfolge der Pattern–Anwendung und unterstützt bei den Entscheidungen für eine der Alternativen.

Das folgende Kapitel zeigt beispielhaft eine Pattern Language konzeptioneller Patterns.

Kapitel 7

Konzeptionelle Server–Patterns

In diesem Kapitel wird ein System konzeptioneller Patterns vorgestellt, das verschiedene Möglichkeiten für den Einsatz von Multitasking für auftragsbearbeitende Server zeigt. Die Pattern Language basiert auf Untersuchungen diverser Server–Produkte und wurde erstmals auf dem Pattern–Workshop VikingPLoP 2003 in Bergen (Norwegen) präsentiert [GT03]. Eine vergleichbare Gegenüberstellung der Multitasking–Strategien von CORBA–Servern wurde von Schmidt in [Sch98] veröffentlicht.

Alle Patterns der folgenden Pattern Language beschreiben Systemstrukturen und fallen damit in die Kategorie der konzeptionellen Patterns. Ein Architektur–Pattern im Sinne der Definition nach Abschnitt 4.5.1 befindet sich aber nicht darunter. Das Beispiel des Apache HTTP Servers in Abschnitt 7.3.2 zeigt aber, welche Architektur–Patterns in diesem Zusammenhang denkbar sind.

7.1 Vorstellung der Domäne

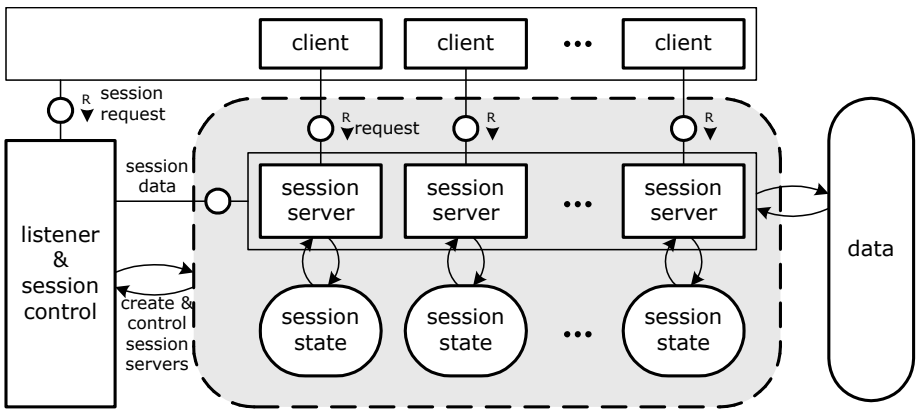


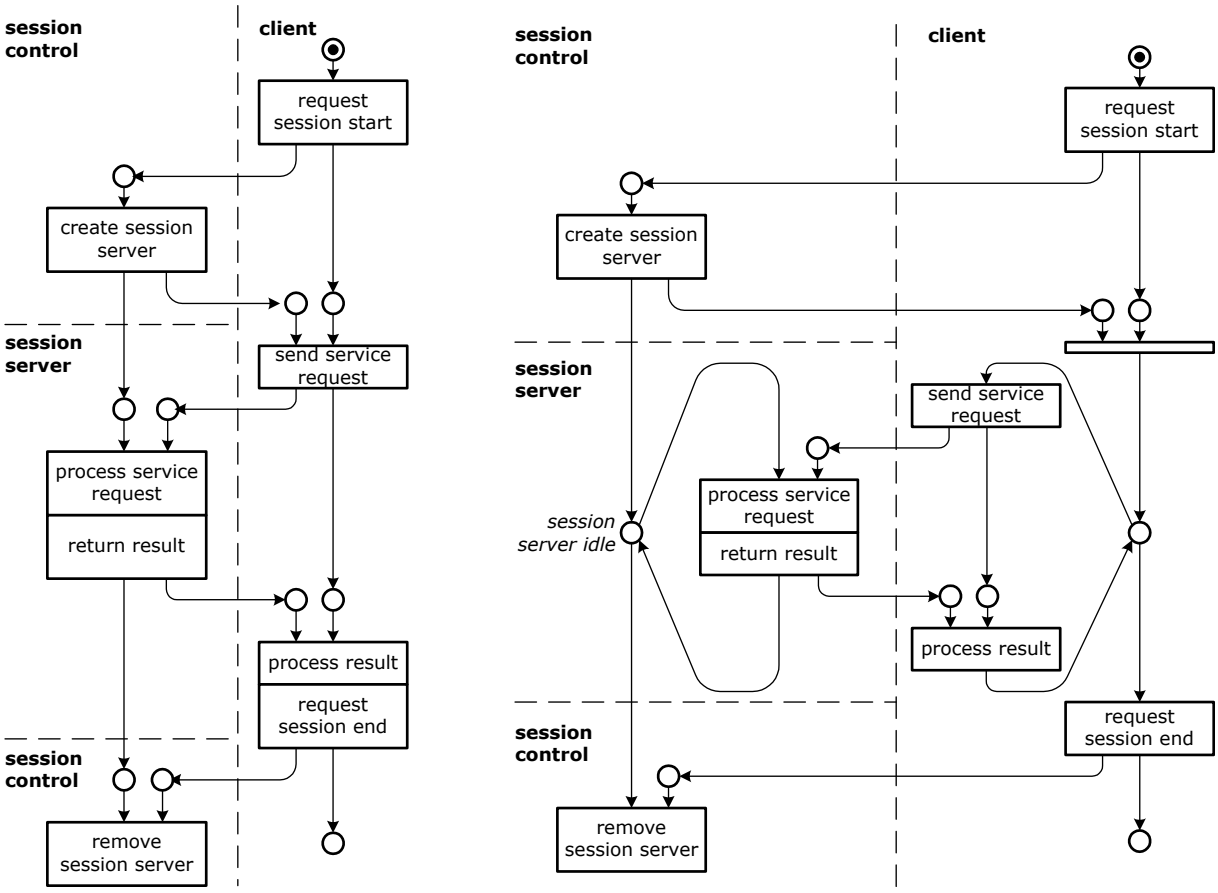
Abbildung 7.1: Clients und auftragsbearbeitende Session Server in einem System

7.1.1 Das gewünschte System

Abbildung 7.1 zeigt das gewünschte System: Mehrere Clients können unabhängig voneinander Aufträge über einen Kanal an einen ihnen allein zugeordneten Server schicken, der sich einen

Sitzungskontext hält, so dass der Client nicht mit jedem Auftrag den gesamten Kontext mitschicken muss. Idealerweise sollte ein Client erst dann etwas von konkurrierenden Aufträgen anderer Clients mitbekommen, wenn es Zugriffskonflikte auf gemeinsame Ressourcen gibt (im Bild: Schreibzugriffe auf gemeinsame Daten).

Um an einen Session Server zu kommen, muss ein Client zuvor eine Sitzung beim Listener anfordern. Dieser baut ihm eine Verbindung zu seinem persönlichen Session Server auf, die er so lange für Aufträge benutzen kann, bis die Sitzung beendet ist.



(a) Single Request Session

(b) Multiple Request Session

Abbildung 7.2: Single und Multiple Request Sessions

7.1.2 Sitzungen und Aufträge

Innerhalb einer Sitzung können ein oder mehrere Aufträge im selben Kontext erteilt werden. Für technische Server ist dabei der Sonderfall interessant, dass in einer Sitzung nur ein Auftrag erteilt wird (Single Request Session), denn in diesem Fall ergibt sich nicht die Notwendigkeit, einen Session Context bis zum Eingang des nächsten Auftrags zu halten.

Abbildung 7.2 zeigt die beiden Möglichkeiten im Ablauf zwischen Client und Server.

7.1.3 Verbindungsaufbau und Antwortzeiten

Die in den folgenden Patterns vorgestellten Lösungen behandeln allesamt die Art, wie ein Client zu seinem Session Server kommt, an den dieser seinen Auftrag richten kann. Dafür muss der Client zuerst eine Verbindung zum Server aufbauen, indem er eine Verbindungsanfrage an den Listener richtet. Nimmt der Listener die Verbindungsanfrage an, wird dadurch die Verbindung zwischen Client und Server aufgebaut. Über diese Verbindung kann jetzt der Client seinen Auftrag an den Session Server stellen.

Da in allen Lösungen nach erfolgtem Verbindungsaufbau eine Server–Task als Session Server exklusiv zur Abarbeitung eines Client–Auftrags zur Verfügung steht, unterscheidet sich die Dauer der eigentlichen Auftragsbearbeitung nicht. Die Unterschiede bestehen vielmehr in der Dauer des Verbindungsaufbaus und wie lange es dauert, bis eine Server–Task den Auftrag des Clients entgegen nehmen kann.

Aus Sicht des Clients ergeben sich damit folgende Zeiten:

t_{conn} *Connect Time*
Die Dauer zwischen Senden einer (TCP–) Verbindungsanfrage und Aufbau der Verbindung

t_{res1} *First Response Time*
Zeit für die Antwort auf den ersten Auftrag nach Verbindungsaufbau

t_{res2+} *Next Response Time*
Zeit für die Antwort auf weitere Aufträge über eine bestehende Verbindung

An diesen Zeiten werden die verschiedenen Lösungen gemessen. Aus Sicht des Servers sind aber andere Zeiten relevant, die bei der Beschreibung von LISTENER / WORKER im Abschnitt 7.2.1 vorgestellt werden.

7.2 Die Pattern Language

Pattern	Problem	Lösung
LISTENER / WORKER	Wie stellt man mit Hilfe von Multitasking nebenläufige Session Server für eine variable Anzahl von Clients zur Verfügung?	Stelle eine Task als Listener für Connection Requests und viele für die Service Requests bereit.
FORKING SERVER	Wie setzt man mit einfachen Mitteln und niedrigem Ressourcen–Verbrauch einen Listener–Worker Server um?	Eine Master Server Task ist Listener und erzeugt bei einem Connection Request eine Worker Task, die den Service Request bearbeitet.
WORKER POOL	Wie erreicht man eine kürzere Antwortzeit bei einem Listener / Worker Server?	Eine bestimmte Anzahl von Worker Tasks werden bei Server–Start erzeugt und je eine wird aktiviert, sobald ein Service Request eintrifft.

Pattern	Problem	Lösung
WORKER POOL MANAGER	Wie verwaltet man den Worker Pool und passt ihn bei möglichst geringem Ressourcen–Verbrauch an die aktuelle Last an?	Der Worker Pool Manager erzeugt und beendet Worker Tasks abhängig von der aktuellen Server–Last und sorgt für ausreichende Kapazitäten für eine kurze Antwortzeit.
JOB QUEUE	Worker Pool: Wie übergibt der Listener einen Job (in Form einer Verbindung zum Client) an eine Worker Task?	Der Listener schiebt den Job in eine Queue, und aktiviert damit den ersten wartenden Worker, der den Job entnimmt. Ist kein wartender Worker da, füllt sich die Queue.
LEADER / FOLLOWERS	Worker Pool: Wie übergibt der Listener einen Job (in Form eine Verbindung zum Client) an eine Worker Task?	Der Listener wechselt seine Rolle und wird selbst zum Worker, muss also nichts übergeben. Der nächste wartende Worker wird dann zum Listener.
SESSION CONTEXT MANAGER	Wie können Multiple Request Sessions mit Workern erreicht werden, die immer genau einen Service Request bearbeiten und dann eventuell einen anderen Client bedienen?	Verwalte die Sessions mit einem Session Context Manager und identifiziere die Session anhand einer vom Client gelieferten ID oder anhand der Verbindung zum Client.

7.2.1 Listener / Worker

Kontext Ein Server soll mit Hilfe von Multitasking Dienste für eine offene Anzahl an Clients nebenläufig anbieten. Es wird ein verbindungsorientiertes Netzwerkprotokoll (z.B. TCP/IP) verwendet.

Problem Wie benutzt man Tasks (Prozesse oder Threads) zur nebenläufigen Bearbeitung von Connection– und Service–Requests?

Trade–Offs

- Die Zeit t_{conn}zwischen Connection Request und Verbindungsaufbau durch den Server sollte kurz sein. Ein Client sollte nicht abgewiesen werden, solange der Server noch ausreichende Rechenkapazität hat.
- Connection Requests treffen auf einem Server Port ein, der nicht von mehreren Tasks gleichzeitig bedient werden kann. Es muss daher sichergestellt sein, dass zu einem Zeitpunkt nur eine Task diesen Port bedient.

Lösung Unterteile die Tasks im Server in Listener und Worker Tasks:

Listener Eine Task überwacht den oder die Server Ports, baut bei einem Connection Request die Verbindung zum Client auf und übergibt diese Verbindung an einen Worker.

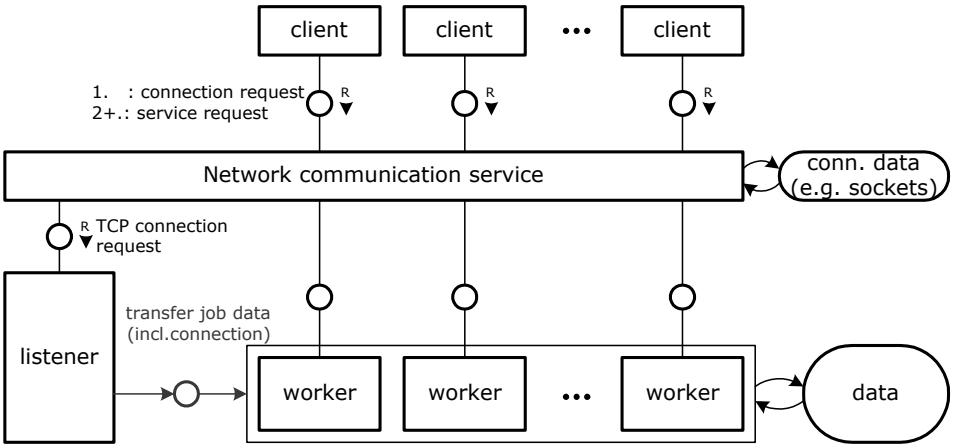


Abbildung 7.3: LISTENER / WORKER

Worker Idealerweise gibt es pro Client eine Worker Task, die Service Requests vom Client über die Verbindung zwischen ihnen annimmt, den Auftrag bearbeitet und das Ergebnis über die Verbindung zurückschickt.

Konsequenzen

Nutzen Der Verbindungsaufbau erfolgt schnell, da der Listener nur die Aufgabe hat, auf Connection Requests zu reagieren.

Pflichten Es müssen noch folgende Fragen geklärt werden

- Wann werden die Worker Tasks erzeugt?
- Wie übergibt der Listener die Verbindung zum Client an einen Worker?

Durch die Wahl des LISTENER / WORKER Patterns haben vier Zeitintervalle Einfluss auf die Antwortzeit des Servers:

t_{listen} *Listener Response Time*
Die Dauer zwischen Empfang einer (TCP–) Verbindungsanfrage und Aufbau der Verbindung durch den Listener.

$t_{latency}$ *Listener Latency*
Die Zeit, die der Listener nach einem Verbindungsaufbau benötigt, bis er die nächste Verbindungsanfrage entgegennehmen kann.

$t_{handover}$ *Connection Handover Time*
Die Dauer zwischen Aufbau der Verbindung durch den Listener und der Bereitschaft des Workers, einen Auftrag über diese Verbindung entgegenzunehmen.

t_{worker} *Worker Response Time*
Die Dauer zwischen Empfang eines Auftrags und Versenden der Antwort durch den Worker.

In der Annahme, dass bei allen Patterns ein Auftrag exklusiv von einer Worker Task bearbeitet wird, unterscheidet sich das vierte Zeitintervall t_{worker} bei keinem der folgenden Patterns. Das gleiche gilt für die Listener Response Time t_{listen}, wenn der Listener die Verbindung nach Erhalt der Verbindungsanfrage selbst aufbaut.

Betrachtet man die Antwortzeiten aus Sicht des Clients (siehe Abschnitt 7.1.3), dann ergeben sich folgende Zusammenhänge:

t_{conn} Connect Time
Die Dauer zwischen Senden einer (TCP–) Verbindungsanfrage und Aufbau der Verbindung hängt gänzlich vom Listener ab. Im günstigsten Fall gilt $t_{conn} = t_{listen}$, der maximale Durchsatz des Listeners beträgt $\frac{1}{t_{listen}+t_{latency}}$.

t_{res1} First Response Time
Für den Zeitbedarf für die Antwort auf den ersten Auftrag nach Verbindungsaufbau gilt $t_{res1} = t_{handover} + t_{worker}$. In der Connection Handover Time können sich die Patterns deutlich unterscheiden.

t_{res2+} Next Response Time
Die Zeit für die Antwort auf weitere Aufträge über eine bestehende Verbindung entspricht der Worker Response Time t_{worker} und unterscheidet sich daher bei den folgenden Patterns nicht.

7.2.2 Forking Server

Auch: THREAD PER REQUEST [Sch98, S. 56]

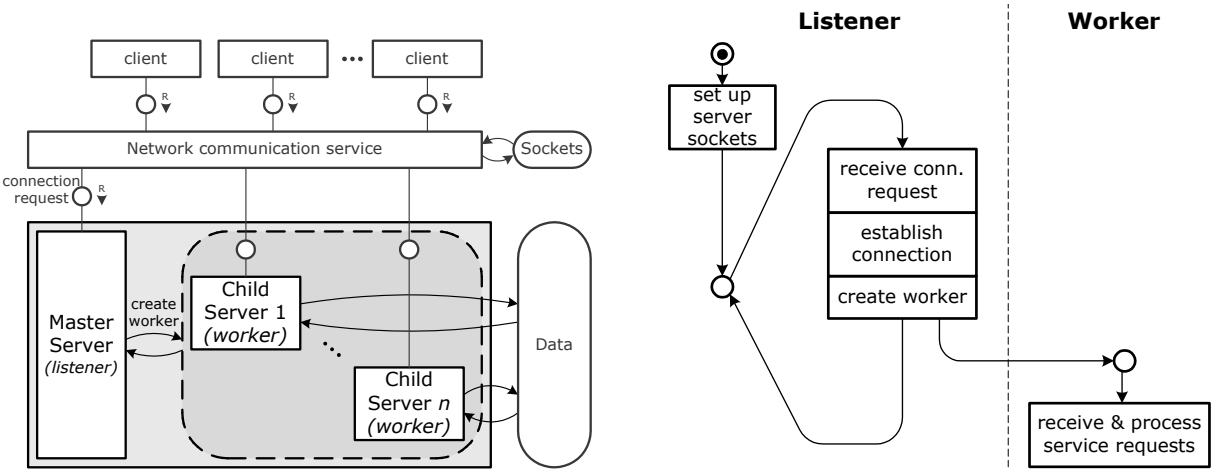
Kontext Man wendet für einen auftragsbearbeitenden Server das LISTENER / WORKER Pattern an.

Problem Wie handhabt man möglichst einfach die Erzeugung der Worker Tasks und die Übergabe der Client–Verbindung?

Trade–Offs

- Jede Betriebssystem–Task (Prozesse oder Threads) verbraucht Ressourcen wie Speicher und Rechenzeit, auch im inaktiven Zustand.
- Werden Betriebssystem–Prozesse als Tasks benutzt, wird die Übergabe einer TCP–Verbindung zwischen zwei Prozessen zum Problem.

Lösung Starte eine Task als Master Server, die die Rolle des Listeners übernimmt. Nachdem diese einen Connection Request angenommen hat, erzeugt sie eine Worker Task, die Service Requests auf dieser Verbindung bearbeitet und sich danach beendet. Im Fall von Betriebssystem–Prozessen unter UNIX erfolgt die Erzeugung mit `fork()`, wobei der Worker Prozess als Klon des Masters auch Zugang zur Verbindung zum Client hat. Das löst das Problem der Übergabe zwischen Listener und Worker.



(a) Aufbau (b) Ablauf

Abbildung 7.4: FORKING SERVER

Konsequenzen

Nutzen

- Die Übergabe der Client–Verbindung zwischen Listener und Worker funktioniert auch bei Verwendung von Betriebssystem–Prozessen.
- Die Anzahl der Tasks richtet sich genau nach dem aktuellen Bedarf

Pflichten Die Antwortzeit des Servers, genauer gesagt $t_{latency}$ und $t_{handover}$ hängen von der Zeit ab, die das Erzeugen einer Task erfordert. Kommen viele Requests an, bei denen jeder für sich schnell beantwortet ist, sorgt das ständige Erzeugen von Tasks und deren Beseitigung nach ihrem Ende für eine hohe Rechnerlast und damit für eine längere Antwortzeit.

7.2.3 Worker Pool

Auch: THREAD POOL ACTIVE OBJECT (Abschnitt 4.2.7.3 bzw. [SSRB00, S. 393].

Kontext Man wendet für einen auftragsbearbeitenden Server das LISTENER / WORKER Pattern an.

Problem Wie erreicht man eine möglichst kurze Antwortzeit?

Trade–Offs

- Erzeugt man, wie beim FORKING SERVER, eine Worker–Task erst bei Annahme eines Connection Requests, erhöht das die Zeit zur Übergabe der Client–Verbindung an den Worker ($t_{handover}$). Weiterhin ist in dieser Zeit der Listener blockiert, was $t_{latency}$ erhöht.
- Jede Task, ob aktiv oder schlafend, verbraucht System–Ressourcen.

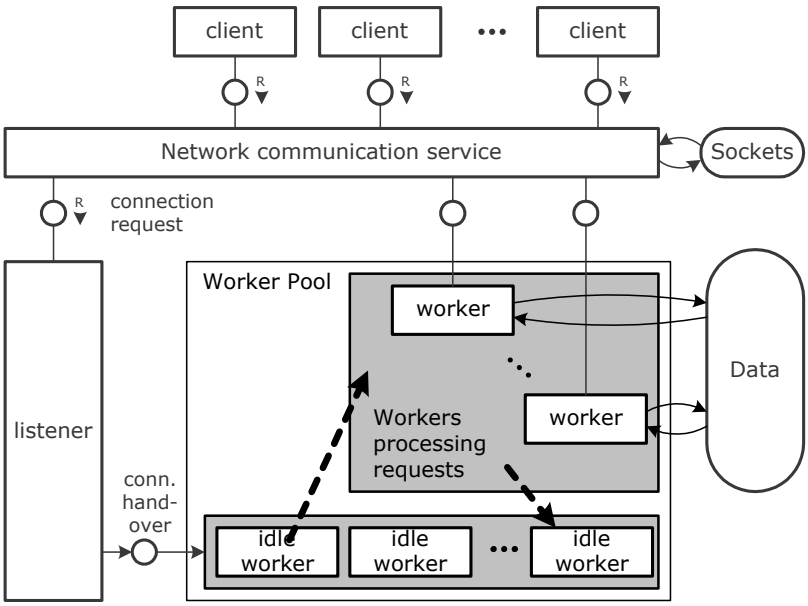


Abbildung 7.5: WORKER POOL

Lösung Erzeuge bei Start des Servers eine bestimmte Anzahl von Worker Tasks, die inaktiv sind, bis eine vom Listener einen Job (eine Client–Verbindung) bekommt. Damit entfällt die Zeit zur Erzeugung einer Worker Task.

Konsequenzen

Nutzen

- Es geht bei der Annahme von Requests keine Zeit mehr für die Erzeugung von Worker Tasks verloren. Damit wird $t_{latency}$ klein, was den Durchsatz des Listeners erhöht.
- Die Anzahl der Worker Tasks kann begrenzt und damit für die Maschinen–Ressourcen (CPUs, Speicher) optimiert werden. Mit einem Session Context Management (siehe Abschnitt 7.2.7) können wenige Worker viele Clients bedienen.

Pflichten

- Die Worker Tasks im Pool müssen beim Server–Start erzeugt und beim Herunterfahren beendet werden.

- Die Übergabe der Client–Verbindung muss jetzt von der Listener–Task zu einer bereits bestehenden Worker–Task erfolgen, was Einfluss auf $t_{handover}$hat. Der beim FORKING SERVER angewendete Trick des Klonens von Prozessen mit `fork()` funktioniert hier nicht.

- Je nach Situation existieren mehr Tasks als erforderlich und verbrauchen Ressourcen.

- Variiert die Server–Last stark, ist eine statische Anzahl an Worker Tasks eventuell ungeeignet. Hier sollte ein WORKER POOL MANAGER eingesetzt werden.

7.2.4 Worker Pool Manager

Kontext Man wendet das WORKER POOL Pattern an.

Problem Wie verwaltet man die Worker im Worker Pool?

Trade–Offs

- Bei Start des Servers muss eine initiale Anzahl an Workern erzeugt werden, bei Ende des Servers müssen alle Worker des Pools beendet werden. Falls sich ein Worker unerwartet beendet (z.B. im Fehlerfall), muss er durch einen neuen ersetzt werden. Das erfordert eine eigene Task zur Verwaltung und Überwachung, die aber wiederum Ressourcen für die Worker blockiert.

- Um zu vermeiden, dass bei Annahme eines Connection Requests ein Worker erzeugt werden muss, weil kein inaktiver mehr da ist, muss sichergestellt werden, dass immer eine bestimmte Anzahl inaktiver Worker Tasks bereit steht, solange es die verfügbaren Ressourcen zulassen. Allerdings verbraucht jede Worker Task Ressourcen. Existieren zu viele inaktive Worker, sollten diese beendet werden.

Lösung Ein spezieller Worker Pool Manager erzeugt, beendet und überwacht Worker Tasks im Pool und trägt diese in eine Liste ein. Jeder Worker trägt in diese Liste ein, womit er gerade beschäftigt oder ob er inaktiv wird. Mit Hilfe dieser Liste stellt der Manager fest, ob die Anzahl der inaktiven Worker zu groß oder zu klein ist und kann dementsprechend entweder Worker Tasks beenden oder neue erzeugen. Beim Herunterfahren beendet er alle Worker Tasks.

Weiterhin lässt sich der Worker Pool Manager vom Betriebssystem mitteilen, ob sich ein Worker beendet hat. Ist das unerwartet geschehen, kann der Manager diesen durch einen neuen Worker ersetzen.

Konsequenzen

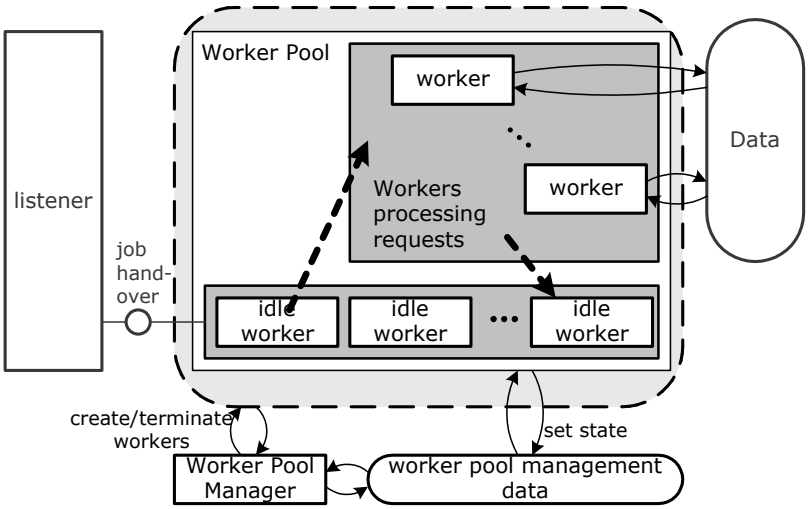


Abbildung 7.6: WORKER POOL MANAGER: Aufbau
(Der obere Teil (Client, Network Communication Service) wurde zur Vereinfachung weggelassen)

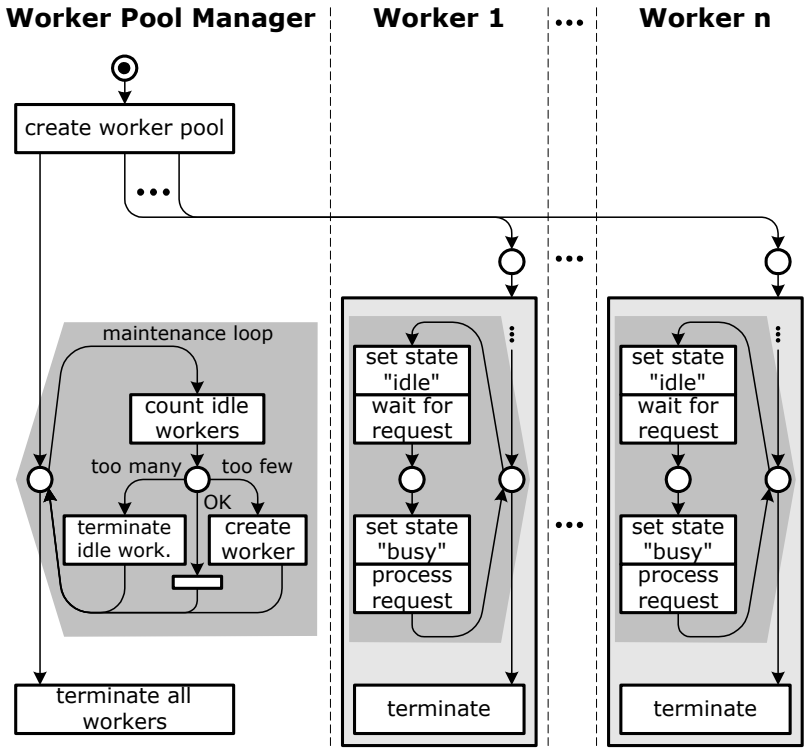


Abbildung 7.7: WORKER POOL MANAGER: Ablauf

Nutzen

- Der Administrator kann den Worker Pool Manager zur Steuerung des Servers benutzen, also zum Starten, Herunterfahren und Anpassen des Ressourcenverbrauchs im Betrieb.
- Der Worker Pool Manager ermöglicht auch bei starker Laständerung die Anpassung des Pools an die aktuelle Server–Last.
- Die Überwachung der Worker Tasks ermöglicht den Dauerbetrieb des Servers.

Pflichten

- Die Service Request–Bearbeitung muss erweitert werden um die Mitteilung, in welchem Zustand der Worker gerade ist. Auf die dafür erforderliche Liste greifen alle Worker und der Manager nebenläufig zu. Hier muss verhindert werden, dass sich die Tasks beim Zugriff blockieren.
- Der Worker Pool Manager wird über eine zusätzliche Task realisiert, die ihrerseits wieder Rechenzeit und Speicher verbraucht.

7.2.5 Job Queue

Auch: HALF–SYNC / HALF–REACTIVE (siehe Abschnitt 4.2.7.1 bzw. [SSRB00, S. 440])

Kontext Man wendet das LISTENER / WORKER und das WORKER POOL Pattern an.

Problem Wie übergibt der Listener eine Client–Verbindung an einen Worker?

Trade–Offs

- Um schnell auf den nächsten Connection Request reagieren zu können, sollte die Übergabe aus Sicht des Listeners schnell erfolgen (niedrige $t_{latency}$).
- Der Wechsel zwischen zwei Tasks erfordert eine gewisse Zeit.
- Bei Verwendung von Betriebssystem–Prozessen ist eine Übergabe einer Client–Verbindung zwischen Prozessen nicht möglich.

Lösung Eine Job Queue dient zur Übergabe der Client–Verbindung zwischen Listener und Worker. Alle inaktiven Worker warten darauf, dass ein Job in die Queue kommt. Nach Annahme eines Connection Requests schiebt der Listener die Client–Verbindung als Job in die Queue und weckt damit den ersten inaktiven Worker, der den Job aus der Queue nimmt und mit dem Client kommuniziert. Falls keine inaktiven Worker Tasks warten, füllt sich die Queue mit jedem Connection Request.

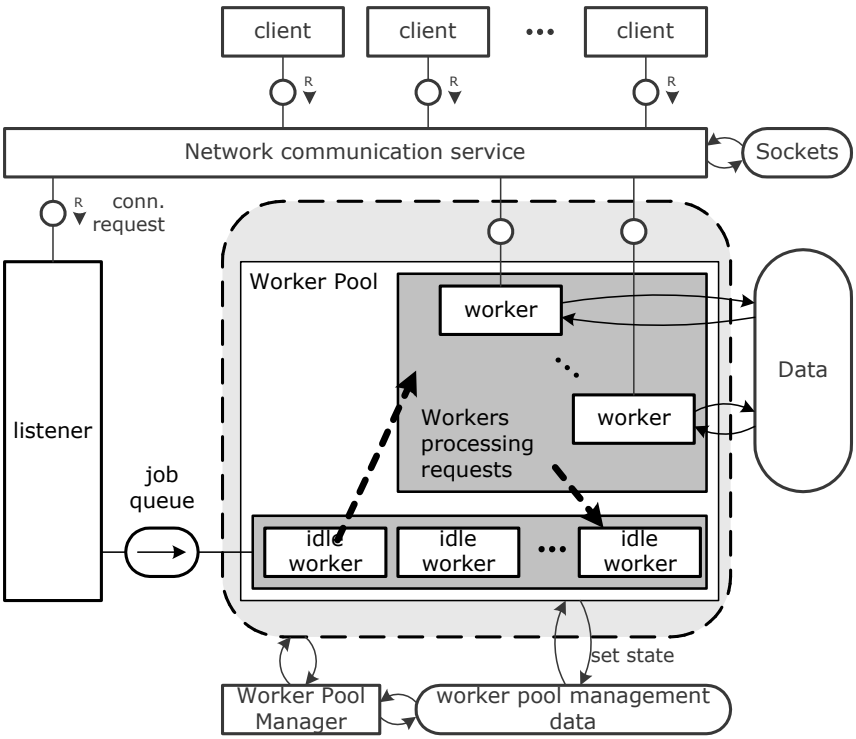


Abbildung 7.8: JOB QUEUE: Aufbau

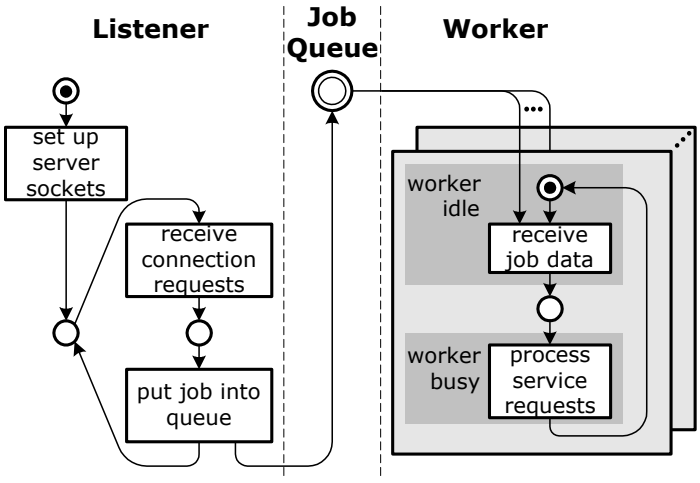


Abbildung 7.9: JOB QUEUE: Ablauf

Konsequenzen

Nutzen

- Der Listener nimmt mit der kürzest möglichen Antwortzeit Connection Requests an (niedrige $t_{latency}$).
- Durch die Job Queue sind auch Worker Pools mit statischer Anzahl an Workern effizient. Hier muss ein Client eventuell etwas warten, bis sein Service Request angenommen und bearbeitet wird.

Pflichten

- Zwischen Aufbau der Verbindung durch den Listener und Annahme des Service Requests durch den Worker findet ein Task–Wechsel statt, der die Antwortzeit, genauer gesagt $t_{handover}$, erhöhen kann.
- Das Pattern ist nicht anwendbar, wenn Betriebssystem–Prozesse benutzt werden, da hier die Client–Verbindung nicht zwischen Listener und Worker übertragen werden kann.

7.2.6 Leader / Followers

(Siehe 4.2.3.4 Abschnitt bzw. [SSRB00, S. 447])

Kontext Man wendet das LISTENER / WORKER und das WORKER POOL Pattern an.

Problem Wie übergibt der Listener eine Client–Verbindung an einen Worker?

Trade–Offs

- Um schnell auf den nächsten Connection Request reagieren zu können, sollte die Übergabe aus Sicht des Listeners schnell erfolgen (niedrige $t_{handover}$).
- Der Wechsel zwischen zwei Tasks erfordert eine gewisse Zeit.
- Bei Verwendung von Betriebssystem–Prozessen ist eine Übergabe einer Client–Verbindung zwischen Prozessen nicht möglich.

Lösung Die Listener Task wechselt bei Annahme eines Connection Requests ihre Rolle und wird zum Worker. Die nächste inaktive Worker Task wird aktiviert und spielt jetzt den Listener. Damit entfällt die Notwendigkeit, die Client–Verbindung zwischen zwei Tasks zu übergeben.

Konsequenzen

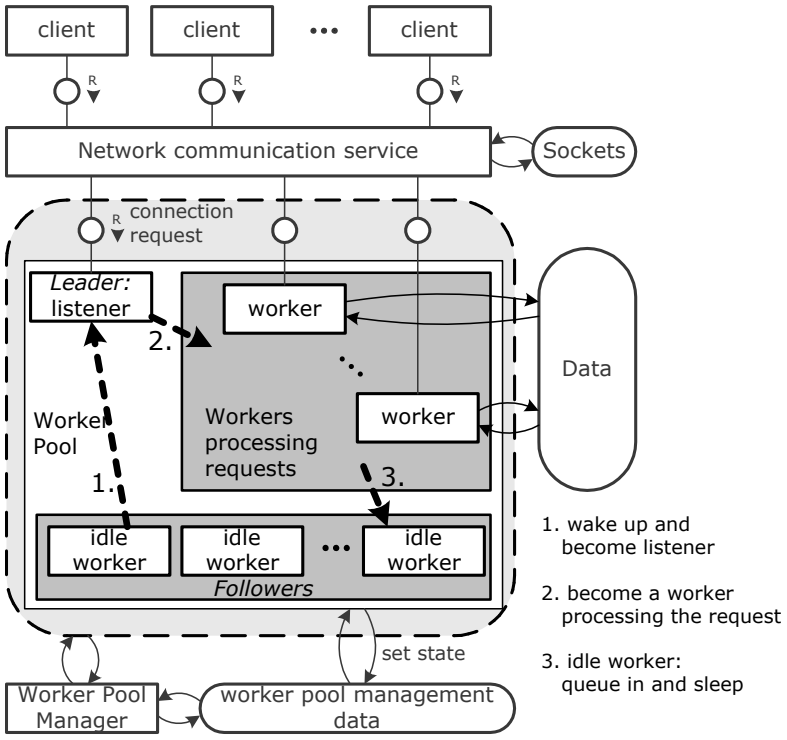


Abbildung 7.10: LEADER / FOLLOWERS: Aufbau

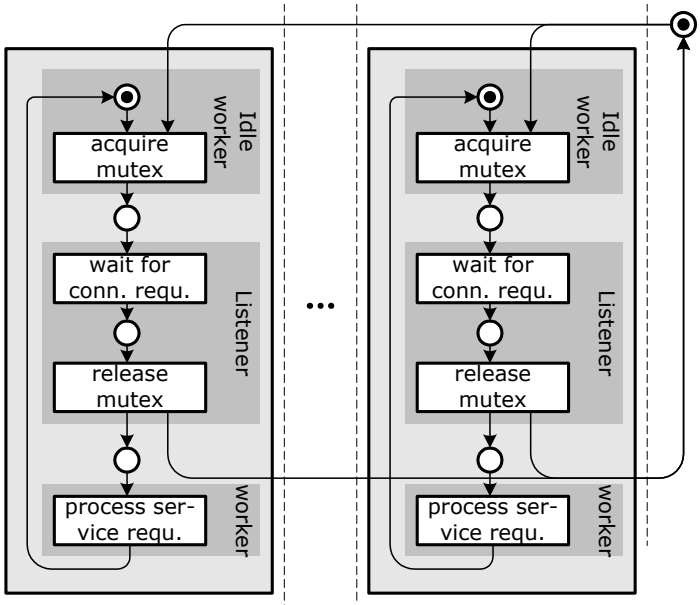


Abbildung 7.11: LEADER / FOLLOWERS: Ablauf

Nutzen

- Es findet kein Task–Wechsel zwischen Aufbau der Verbindung durch den Listener und Annahme des Service Requests durch den Worker statt, damit ist die Antwortzeit, genauer gesagt $t_{handover}$ kurz. Auch ist eine Realisierung mit Betriebssystem–Prozessen möglich.

Pflichten

- Durch den Rollen– und Task–Wechsel kann die Annahmezeit des Listeners variieren ($t_{latency}$). Falls keine inaktive Worker Task bereit ist, bleibt die Rolle des Listeners sogar vakant, so dass der Server keine Connection Requests annimmt. Um das zu verhindern, ist eine dynamische Regelung des Worker Pools durch einen WORKER POOL MANAGER erforderlich.
- Die Server–Ports, auf denen die Connection Requests eintreffen, müssen für alle Tasks zugreifbar sein, wobei sichergestellt werden muss, dass nur die Task in der Rolle des Listeners auch tatsächlich darauf zugreift.

7.2.7 Session Context Manager

Kontext Man wendet für einen auftragsbearbeitenden Server das LISTENER / WORKER Pattern an und möchte Multiple Request Sessions handhaben, also den Kontext zwischen mehreren Requests eines Clients erhalten.

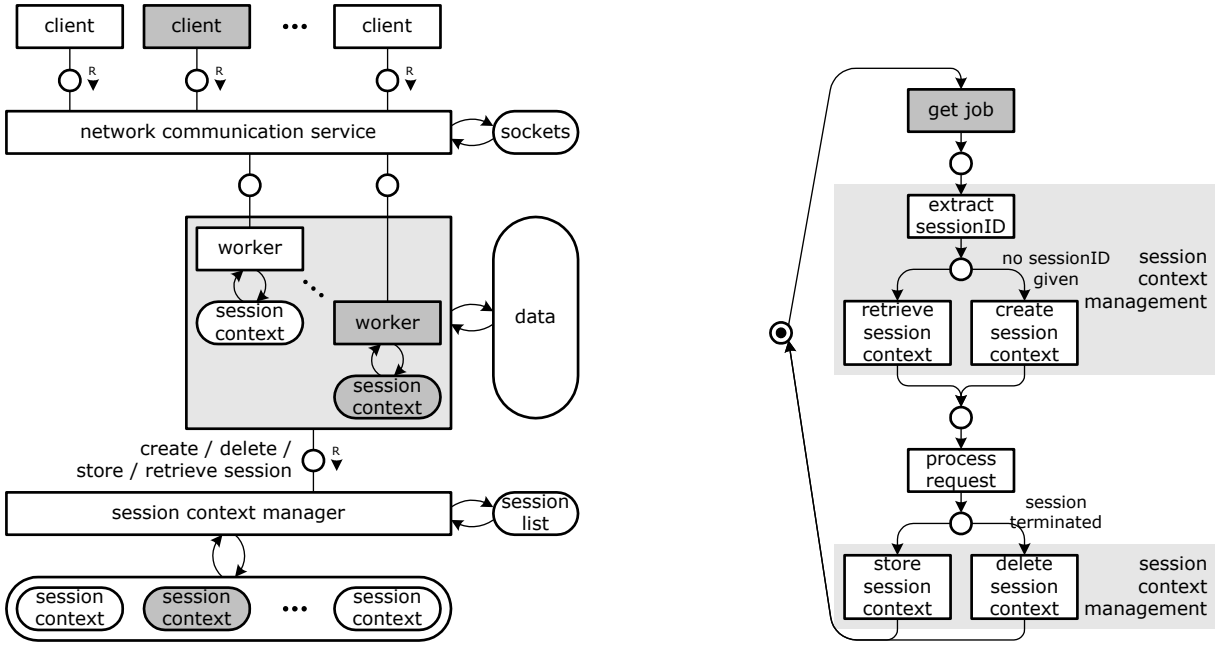
Problem Wie kann ein Worker Multiple Request Sessions bearbeiten, ohne exklusiv einem Client zugeordnet zu sein?

Trade–Offs

- Ist ein Worker genau einem Client für dessen Session zugeordnet, verbringt er einen größeren Teil der Zeit mit Warten auf den nächsten Service Request und blockiert damit Ressourcen.
- Reißt die Verbindung ab oder beendet sich die Worker Task wegen eines Fehlers, sind auch die Sitzungsdaten verloren.

Lösung Ein Session Context Manager verwaltet Zustand und Daten einer Multiple Request Session. Ein Worker bearbeitet immer genau einen Service Request, indem er den zugehörigen Kontext vom Manager anfordert, den Service Request beantwortet und den geänderten Kontext an den Manager zurück gibt. Zur Identifikation der Session wird eine ID verwendet, die der Client mit dem Service Request zusammen übertragen muss (wie beim ASYNCHRONOUS COMPLETION TOKEN in Abschnitt 4.2.7.2).

Konsequenzen



(a) Aufbau

(b) Ablauf

Abbildung 7.12: SESSION CONTEXT MANAGER

Nutzen

- Die Worker Tasks werden optimal ausgenutzt, da sie nicht zwischen zwei Service Requests eines Clients für andere Requests blockiert sind.
- Da ein Client seine Session über eine ID identifizieren muss, ist im Fall eines Verbindungsabbruchs oder im Fehlerfall einer Worker Task das Wiederherstellen einer Session möglich. Das gilt auch für den Fall, dass nicht erwünscht ist, dass über die Dauer der Session die Verbindung gehalten wird, beispielsweise weil diese mehrere Tage lang dauern kann.

Pflichten

- Ein Client muss eine Session ID speichern und mit jedem Service Request mitschicken.
- Man muss damit rechnen, dass eine Session nicht sauber beendet wird. Das erfordert eine Garbage Collection im Session Context Speicher.
- Die Antwortzeit innerhalb einer Session erhöht sich, da bei jedem Service Request erst der Session Context angefordert werden muss.

7.2.8 Leitfaden zur Wahl der Patterns

1. Kläre den Kontext für Listener / Worker.

   - Dient der Server zur Auftragsbearbeitung (Request processing) oder nicht (beispielsweise Streaming)? Sollen Multitasking–Mittel des Betriebssystems (Prozesse oder Threads, Inter–Task–Kommunikation) benutzt werden? Falls nicht, ist das Pattern–System eventuell unpassend.
   - Umfasst eine Session mehr als einen Request? Dann ziehe Punkt 4, Context Management, in Betracht.

2. Wähle ein Pattern zur Task–Nutzung.
   Ist ein niedriger Ressourcen–Verbrauch wichtiger als die Antwortzeit? Wähle für diesen Fall FORKING SERVER, ansonsten WORKER POOL.

3. Falls die Entscheidung für eine WORKER POOL Lösung gefällt wurde:

   - Wähle ein Pattern zum Job Transfer. Falls es einfacher ist, Job–Daten zwischen Tasks zu übermitteln als die Rollen der Tasks (insbesondere des Listeners) zu wechseln, dann benutze eine JOB QUEUE, ansonsten LEADER / FOLLOWER.
   - Gibt es starke Schwankungen in der Anzahl nebenläufig eintreffender Requests? Dann verwende einen dynamischen Worker Pool, der von einem WORKER POOL MANAGER überwacht wird.
   - Bestimme eine Strategie zur Auswahl der nächsten Worker Task für einen Job bzw. die Listener–Rolle: FIFO, LIFO[1], nach Prioritäten oder durch das Betriebssystem bestimmt.

4. Falls mehrere Requests in einer Session auftreten können:

   - Erlaubt es die Anzahl nebenläufiger Sessions und ihre Dauer, pro Session exklusiv eine Task zu reservieren? Falls nicht, verwalte die Kontextinformationen der Sessions mit einem SESSION CONTEXT MANAGER.

Die Pattern Language behandelt nur Strategien, eintreffende Requests auf Tasks zu verteilen. Zur Strukturierung der Request–Bearbeitung sind andere Patterns erforderlich, z.B. INTERCEPTOR. Nebenläufigkeit muss man im Folgenden aber nur noch insofern berücksichtigen, dass es bei der Request–Bearbeitung konkurrierende Zugriffe auf gemeinsame Ressourcen wie z.B. Dateien geben kann. Die folgenden Beispiele zeigen, in welchen Produkten die vorgestellten konzeptionellen Patterns zu finden sind.

7.3 Anwendungsbeispiele

7.3.1 Der Internet Daemon (inetd)

Ein typischer Vertreter eines FORKING SERVERS ist der Internet Daemon (inetd), der in den meisten UNIX–Systemen zu finden ist. Seine Aufgabe ist es, auf einer Reihe von Ports auf Anfragen zu warten. Kommt eine Anfrage, startet er einen Server–Prozess, der diese bedient.

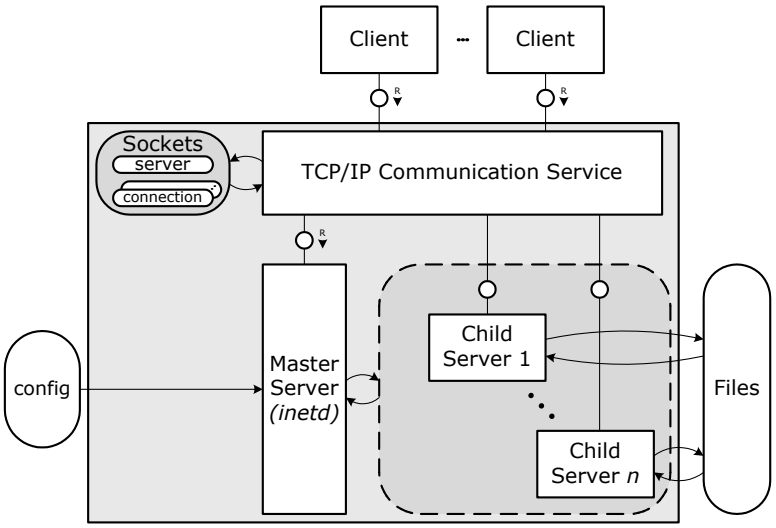


Abbildung 7.13: Der Internet Daemon (inetd) — ein typischer FORKING SERVER

Damit kann man die Anzahl der Prozesse verringern, die auf Anfragen warten. Abbildung 7.13 zeigt das zugehörige Aufbaubild.

In einer Konfigurationsdatei (config) wird festgelegt, welche Ports beobachtet werden sollen und welches Programm für Anfragen auf welchem Port gestartet werden soll. Bei TCP–Kommunikation baut der inetd als Master Server die Verbindung auf, startet den Server–Prozess als Kopie seiner selbst mit `fork()` und lädt in diese Kopie das angegebene Programm mit `exec()`. Der Verbindungs–Socket und damit die TCP–Verbindung zum Client ist danach noch über File–Deskriptoren erreichbar, da diese beim `exec()` nicht gelöscht werden.

7.3.2 Der Apache HTTP Server

Der Apache HTTP Server ist der am weitesten verbreitete Server im World Wide Web (WWW). Da die Quelltexte komplett offen liegen, eignet er sich hervorragend als Untersuchungsgegenstand für Server–Technologie. Im Rahmen mehrerer Seminare wurden die Kern–Bestandteile untersucht und modelliert und die Ergebnisse schließlich online veröffentlicht [GKKS04].

Der Apache HTTP Server wurde auf viele Plattformen portiert, wobei es größere Unterschiede beim Multitasking–Modell (Prozesse und/oder Threads) und bei der Synchronisation von nebenläufigen Zugriffen auf Ressourcen gibt. Weiterhin wird er für verschiedenste Zwecke eingesetzt — vom lokalen Server für die Online–Hilfe bis zum Web–Hoster für hunderte von Web–Domains pro Server.

Diese Vielfalt wird durch einem sehr modularen Aufbau des Servers ermöglicht; tatsächlich können die Module, die ihn überhaupt zu einem rudimentären HTTP Server machen, zur Laufzeit als Plug–Ins geladen werden (vergleiche dazu auch INTERCEPTOR und COMPONENT CONFIGURATOR). Auch die Anpassung an die jeweilige Plattform und ihr Multitasking–Modell erfolgt über eine spezielle Sorte von Modulen, die so genannten Multi–Processing Modules (MPM). Auf diese Weise können verschiedene Multitasking–Strategien durch Austausch des MPMs benutzt werden.

[1] FIFO: First–In, First–Out, also Warteschlangen–Prinzip; LIFO: Last–In, First–Out, also Stapel–Prinzip

Im folgenden werden zwei MPMs vorgestellt, nämlich das Preforking MPM und das Worker MPM. Allen Apache–MPMs ist gemeinsam, dass sie LISTENER / WORKER und WORKER POOL verwenden. Ein FORKING SERVER kommt nicht in Betracht, da HTTP nur Single–Request Sessions zulässt, HTTP–Aufträge häufig auftreten und nur eine kurze Bearbeitungszeit erfordern. Damit wäre die Startzeit eventuell größer als die Bearbeitungszeit. Weiterhin ist in allen MPMs der Listener vom WORKER POOL MANAGER getrennt. Das so genannte Scoreboard enthält die Verwaltungsdaten zum Worker Pool, in dem alle Worker ihren aktuellen Zustand vermerken.

7.3.2.1 Das Apache Preforking Modell

Seit den ersten Versionen aus dem Jahr 1995 benutzt der Apache HTTP Server die so genannte Preforking–Strategie, die auf das LEADER / FOLLOWERS Pattern hinausläuft: Ein Master Server erzeugt beim Start des Server per `fork()` eine Menge von Child Server Prozessen, die den WORKER POOL bilden. Da das geschieht, bevor überhaupt auf HTTP Requests gewartet wird, heißt diese Strategie *Pre*forking.

Ein Child Server bekommt den sogenannten *accept mutex*, übernimmt die Rolle des Listeners und wartet auf HTTP Requests. Sobald ein solcher eintrifft, gibt der Child Server die Rolle des Listeners an den nächsten wartenden ab, indem er den accept mutex abgibt, und bearbeitet den HTTP Request, um sich danach wieder um die Rolle des Listeners zu bemühen.

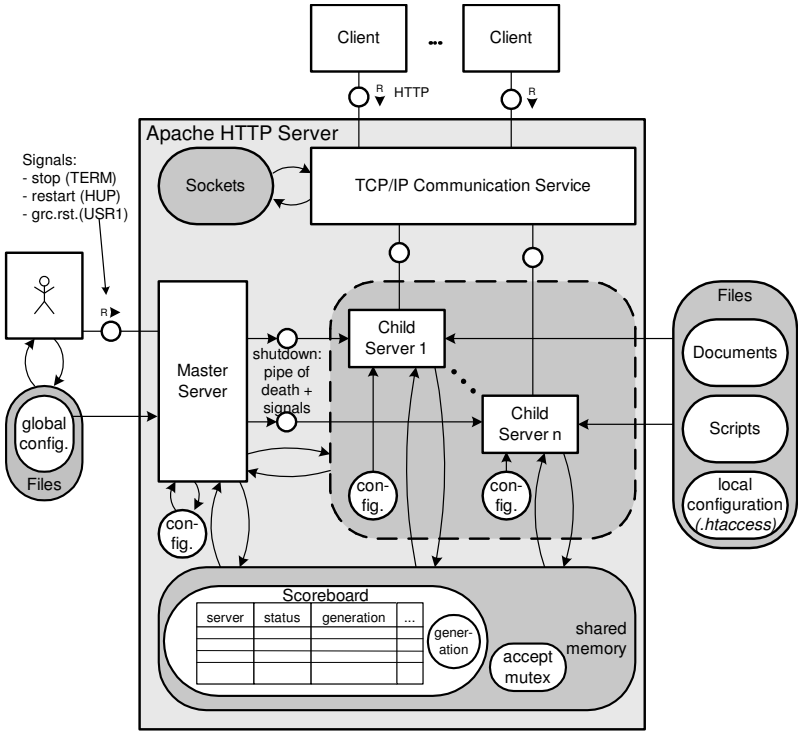


Abbildung 7.14: Apache Preforking MPM

Abbildung 7.14 zeigt die Systemstruktur des Apache HTTP Servers auf Prozess–Ebene. Der Master Server hat die Rolle des WORKER POOL MANAGERS: Er überwacht die Child Server Prozesse, startet neue, falls sich einer unerwartet beendet oder falls zu wenig Reserve im Pool ist, oder beendet manche, wenn zu viele untätig sind. Weiterhin liest er die globale Server–Konfiguration ein, was durch die hohe Konfigurierbarkeit des Servers und aller seiner Module

ein aufwändiger Vorgang ist. Da er die Child Server Prozesse per `fork()` als Kopie von sich erzeugt, beinhalten diese damit auch die eingelesene Server–Konfiguration. Das hat aber auch zur Folge, dass bei einer Änderung an den Konfigurationsdateien alle Server–Prozesse neu erzeugt werden müssen.

Der Master Server ist auch Repräsentant des Servers für den Administrator, der über ihn den Server startet, das Neu–Einlesen der Konfiguration verlangt oder den Server herunter fährt. Das Scoreboard (im Bild unten) dient zur Buchführung über alle Child Server Prozesse. Bei Erzeugung legt der Master Server für jeden Child Server einen Eintrag an. Ein Child Server wiederum trägt dort seinen aktuellen Zustand ein. Daran erkennt der Master Server, welcher Child Server untätig ist und kann dann gegebenenfalls Prozesse starten oder beenden.

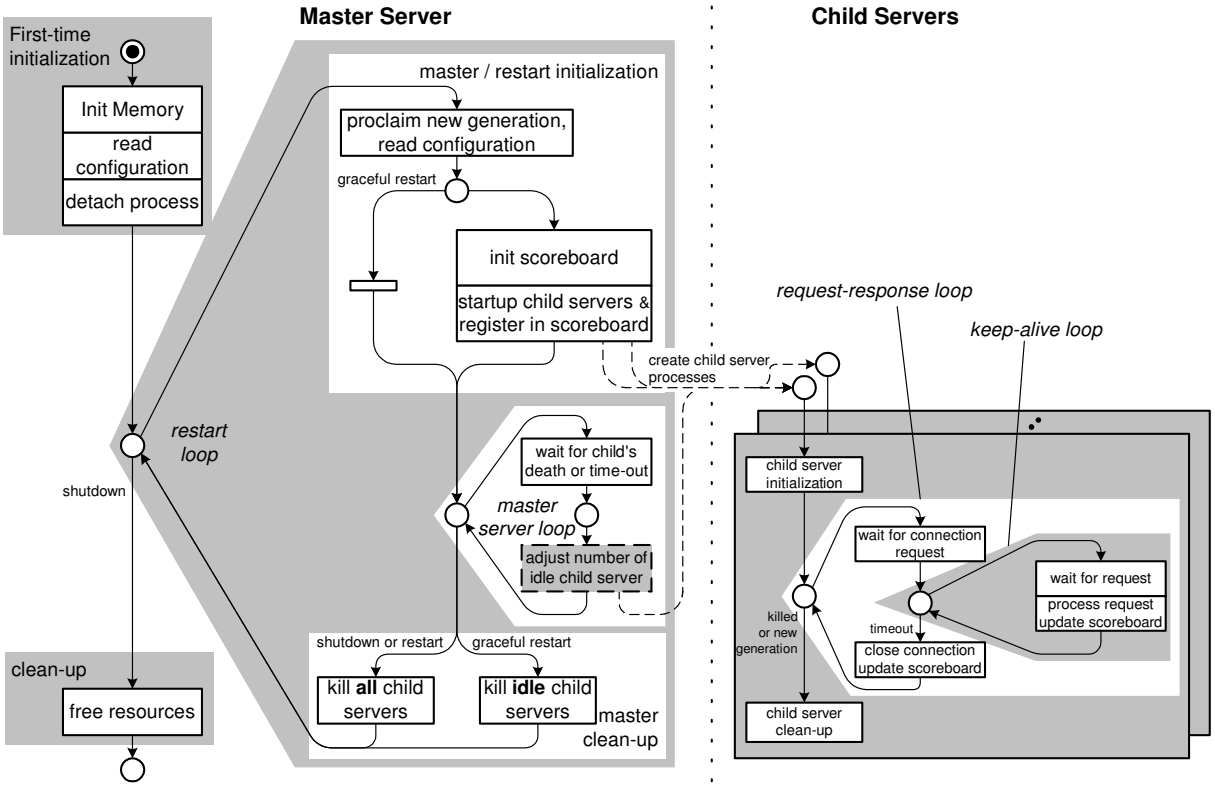


Abbildung 7.15: Apache Haupt–Ablauf

Die Speicherstelle `generation` im Scoreboard und in jedem Child Server–Eintrag zeigt an, welche Generation (Version der Konfiguration) aktuell ist und zu welcher jeder einzelne Child Server gehört; sie dient zum nahtlosen Übergang auf eine geänderte Server–Konfiguration: Damit eine neue Konfiguration aktiv wird, muss der Master Server die bestehenden Child Server Prozesse beenden und durch neue ersetzen. Um den laufenden Betrieb nicht zu stören, beendet er aber nur diejenigen, die ohnehin gerade untätig sind (Zustand *idle* im Scoreboard). Er erhöht die Variable `generation` und kopiert deren Wert in die Scoreboard–Einträge der neu erzeugten Child Server. Ein Child Server überprüft nach erfolgreicher Bearbeitung eines HTTP Requests, ob die eigene Generation noch mit der in der Variablen `generation` übereinstimmt. Falls nicht, beendet er sich und wird daraufhin vom Master Server durch einen neuen ersetzt.

Abbildung 7.15 zeigt den Hauptablauf des Apache HTTP Servers. Er besteht aus zwei geschachtelten Schleifen für den Master Server und die Child Server. In der äußeren Schleife des Master Servers (restart loop) erfolgt das Einlesen der Konfigurationsdateien und das Er-

zeugen der Child Server Prozesse, sowie das Beenden dieser Prozesse am Schleifenende. Die äußere Schleife wird bei jedem Neustart (üblicherweise zum Aktivieren einer neuen Konfiguration) durchlaufen. Der nahtlose Übergang zur neuen Konfiguration heißt hier *graceful restart*. Die innere Schleife, den Master Server Loop, durchläuft der Master Server in regelmäßigen Abständen, um die Anzahl der Child Server der aktuellen Server–Last anzupassen.

Die Child Server erledigen die eigentliche Aufgabe eines HTTP Servers. In ihrer äußeren Schleife, dem Request–Response Loop, durchlaufen sie die Rollen Listener, Worker und Idle Worker. Erst in der inneren (Keep–alive Loop) werden HTTP–Requests bearbeitet. Der Name der inneren Schleife kommt daher, dass zwar bei HTTP mit einem Request auch die Sitzung abgeschlossen ist (also keine Kontextdaten am Server gespeichert werden müssen), dass aber üblicherweise mehrere Requests aufeinander folgen. Um nicht für jeden Request eine neue TCP–Verbindung aufbauen zu müssen, wird die bestehende so lange offen gehalten (keep alive), bis für einen bestimmten Zeitraum keine Requests mehr ankommen. So lange steht ein Child Server Prozess exklusiv für diese Verbindung zur Verfügung.

7.3.3 Das Apache Worker Modell

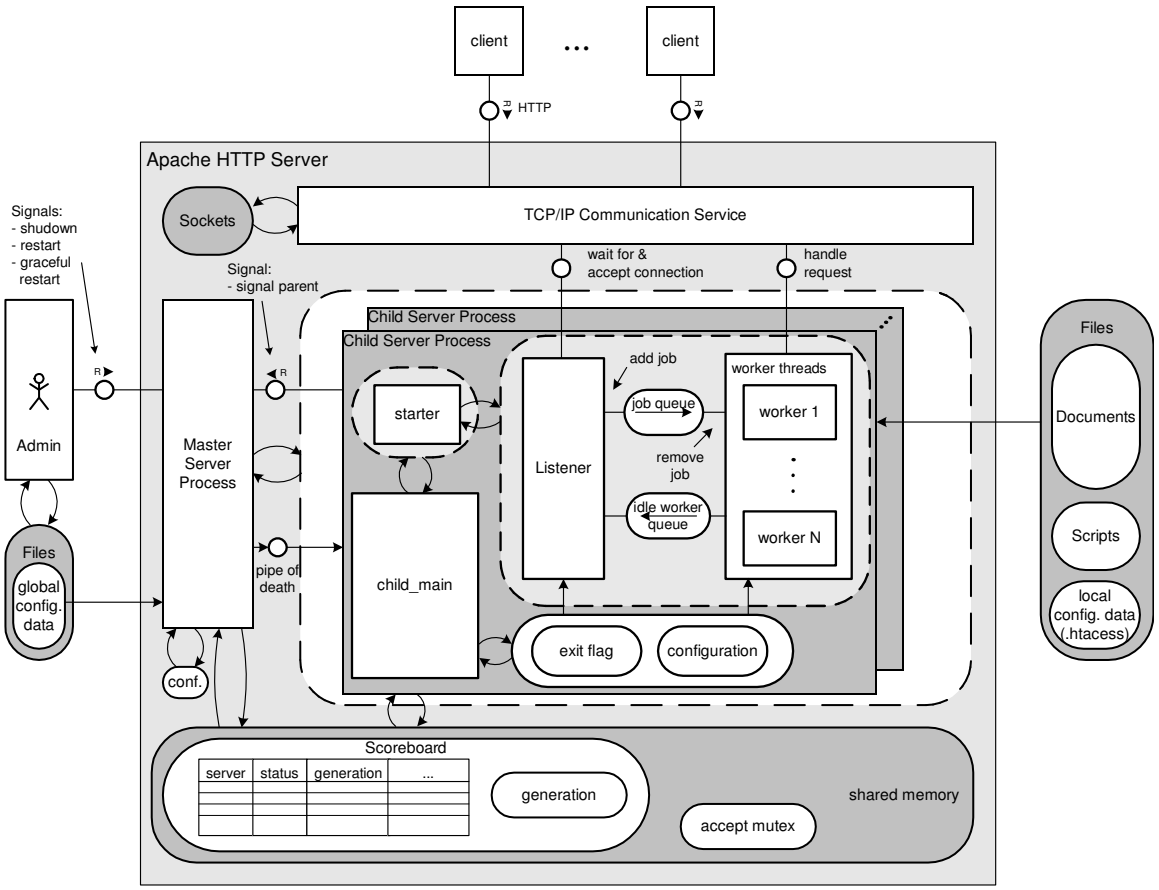


Abbildung 7.16: Apache Worker MPM

Das so genannte Worker MPM des Apache HTTP Servers nutzt eine Kombination aus Threads und Prozessen, um einerseits eine höhere Geschwindigkeit durch geringeren Aufwand zum Kontextwechsel zwischen Threads zu erzielen, andererseits aber die Robustheit der Prozesse

zu nutzen, die durch ein unerwartetes Ende eines anderen Prozesses nicht beeinträchtigt werden.

In Abbildung 7.16 haben Child Server–Prozesse im Gegensatz zum Preforking MPM eine innere Struktur, bestehend aus einem Listener Thread, einem Pool von Worker Threads und einer Job Queue. Der Thread Child_main ist für Start (Erzeugen des Starter Threads, der seinerseits Listener und Worker Threads sowie die Queues erzeugt) und Ende (setzen des Exit Flags) zuständig. Da die Anzahl der Worker Threads fest ist, gibt es keine Anpassung zur Laufzeit des Servers.

In Erweiterung des JOB QUEUE Patterns gibt es hier noch eine Idle Worker Queue, die folgenden Zweck hat: In jedem Child Server Prozess existiert ja ein Listener Thread, der sich über den *Accept Mutex* um den Zugang zu den Server Sockets bemüht, um auf neue Verbindungsanfragen warten zu können. Es wäre aber fatal, wenn ein Listener den Zugang bekommt, wenn gerade alle Worker Threads in seinem Prozess beschäftigt sind und damit kein Worker da wäre, um den nächsten HTTP Request zu bearbeiten. Deshalb tragen sich Idle Workers in die Idle Worker Queue ein, womit der Listener weiß, dass Worker Threads zur Verfügung stehen und er sich um den Zugang zu den Server Sockets bemühen kann.

Der Master Server hat die gleichen Aufgaben wie im Preforking Modell, nämlich WORKER POOL MANAGER, Konfigurationseinleser und Repräsentant des Servers zu sein.

7.3.4 Dispatcher und Taskhandler im SAP R/3 Basissystem

Das R/3–System von SAP ist ein skalierbares Enterprise Resource Planning (ERP) System mit einer Drei–Ebenen–Architektur: Eine zentrale Datenbank, eine variable Anzahl von Application Servern sowie eine große Anzahl von Thin Clients (SAP GUI).

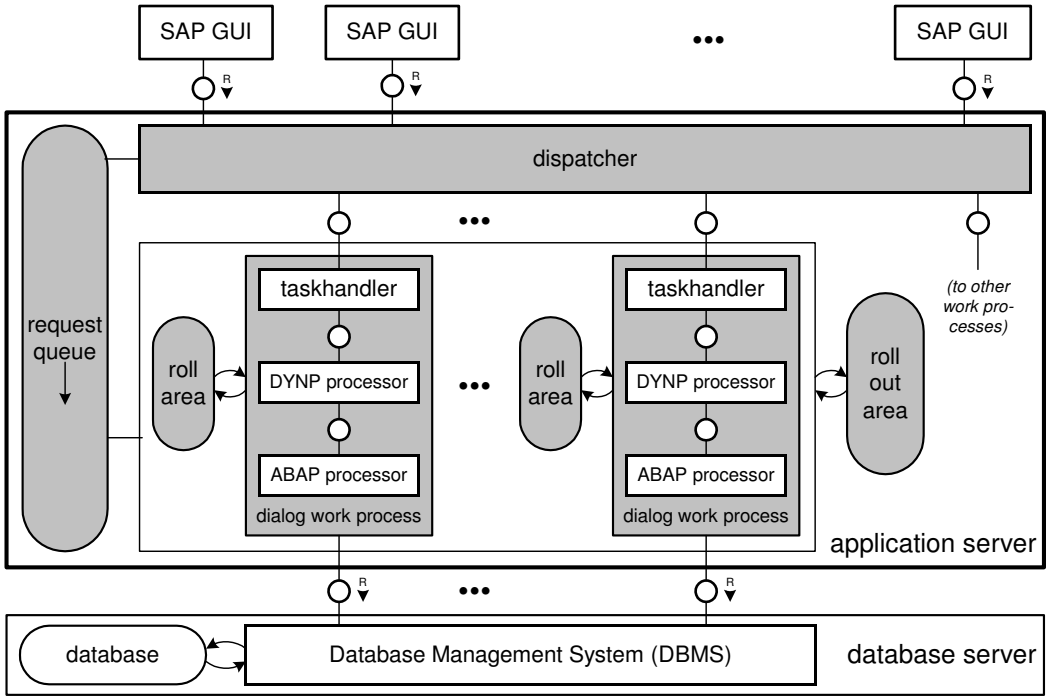


Abbildung 7.17: SAP R/3 Basis: Dispatcher und Taskhandler

Die drei Ebenen sind auch in Abbildung 7.17 gezeigt, wobei hier der Schwerpunkt auf dem Basis–System innerhalb eines Application Servers liegt. Aus dem Blickwinkel eines auftragsbearbeitenden Servers folgt die R/3–Basis dem WORKER POOL Pattern. Während aber im Beispiel des Apache HTTP Servers immer mehr Server Prozesse existieren als Clients Verbindungen zum Server offen haben, gibt es beim R/3 eine relativ geringe Anzahl von Servern, die so genannten Work–Prozesse, die im Zeitmultiplex Verarbeitungsschritte für die verschiedenen Clients durchführen.

Der *Dispatcher* empfängt einen Request und schiebt ihn in die Request Queue. Der nächste freie Work–Prozess holt den nächsten Request aus der Queue, genauso wie in JOB QUEUE. Da sich im R/3 eine Sitzung (dort auch „Modus“ genannt) üblicherweise über mehrere Dialogschritte erstreckt, müssen bei jedem Request Sitzungsdaten geladen, modifiziert und wieder gespeichert werden. Diese Aufgabe, wie im SESSION CONTEXT MANAGER Pattern beschrieben, übernimmt der *Task Handler*, der die Sitzungskontexte („Roll Area“) in der Roll Out[2] Area speichert. Die R/3–Basis benutzt eine feste Anzahl von Work–Prozessen, in denen über Zeitmultiplex ein kooperatives Multitasking stattfindet. Werden Work–Prozesse blockiert, etwa durch Debugging, sinkt dadurch auch die Gesamtleistung des Application Servers.

[2]Der Name „Roll Out Area“ kommt aus der Anfangszeit des R/3, als der Sitzungskontext („Roll Area“) vom Hauptspeicher auf einen Massenspeicher „heraus gerollt“ wurde. Schon bald verwendete man stattdessen shared memory, wodurch sich das Laden und Speichern des Sitzungskontexts auf das Ein– und Ausblenden von Speicherbereichen reduziert.

Kapitel 8

Fazit und Ausblick

Systemstrukturen spielen für die Architekturphase, in der die wichtigen Strukturen geplant und die spätere Arbeitsteilung ermöglicht werden, eine entscheidende Rolle, denn ihre Modellierung ermöglicht eine ausreichende Abstraktion unter Berücksichtigung des Gesamtsystems. Durch die Tragweite der Entscheidungen in der Architekturphase können Patterns ein wertvolles Mittel für den Entwurfsprozess darstellen. Patterns können zur Weitergabe von Erfahrungswissen in der Planung und im Entwurf Software–intensiver Systeme eingesetzt werden. Dabei erscheint vor allem die Kategorie der Architektur–Patterns interessant, da ihr Zweck darin besteht, grundlegende Strukturen eines Systems festzulegen.

Eine genauere Untersuchung gängiger Architektur–Patterns zeigt aber, dass die Kombination aus anwendungsstrukturierendem Konzept und der Beschreibung der dazu erforderlichen Infrastruktur zu Problemen führt: Häufig ist nicht klar, wofür ein solches Pattern steht und damit auch nicht, in welche thematische Kategorie es gehört. Weiterhin werden für die Beschreibung und grafische Darstellung von Patterns meist Mittel zur Darstellung von Software–Strukturen verwendet, auch wenn Konzepte im Bereich der Systemstrukturen darzustellen sind. Darunter leiden neben der Einprägsamkeit eines Patterns auch dessen Wiedererkennbarkeit in einem untersuchten System.

In der Architekturphase spielen aber nicht nur Architektur–Patterns eine Rolle: Den wenigen Architektur–Patterns stehen nämlich eine ganze Reihe von Design–Patterns gegenüber, die ebenfalls Lösungen im Bereich der Systemstrukturen beschreiben, und damit auch in der Architektur–Phase nützlich sein können.

Die Lösung stellen konzeptionelle Patterns dar. Es handelt sich dabei um eine Kategorie von Patterns, die Lösungen im Bereich der Systemstrukturen beschreiben und in deren Beschreibung mit FMC–Diagrammen auch eine angemessene Darstellung verwendet wird. Sie eignen sich zur Verwendung in der Architekturphase.

Ein Großteil der Architektur–Patterns und eine Reihe von Design–Patterns fällt in die Kategorie der konzeptionellen Patterns, so dass hier nur eine Aufbereitung mit Schwerpunkt auf der Darstellung der Systemstrukturen erforderlich ist, so wie in Abschnitt 4.2 geschehen. Um einerseits eine Überfrachtung der Patterns zu vermeiden und andererseits das Finden des passenden Patterns zu erleichtern, ist es sinnvoll, für abgegrenzte Themenbereiche *Pattern Languages* zusammenzustellen, die neben den durch die Patterns beschriebenen Lösungen auch Entwurfsentscheidungen und Alternativen darstellen. Die Pattern Language für Multitasking Server in Kapitel 7 ist eine Anregung, für ein abgegrenztes Themengebiet verschiedene Entwurfsmöglichkeiten mit deren Vor– und Nachteilen mit Hilfe einer Pattern Language zu präsentieren.

Die strenge Form eines Patterns hat schließlich den Vorteil, dass sich ein Autor Gedanken um die präzise Formulierung von Problem, Lösung und Konsequenzen machen muss. Auch wenn für einen Bericht oder eine Dokumentation nicht die Pattern–Form gefordert wird, erhöht diese Präzisierung die Verständlichkeit der vorgestellten Lösung erheblich.

Auch wenn konzeptionelle Patterns Planung und Entwurf auf höheren Abstraktionsebenen erlauben, müssen irgendwann Software–Strukturen von den Systemstrukturen abgeleitet werden. An dieser Stelle kommen andere Patterns zum Tragen, die *Bridging Patterns*. Zu den Bridging Patterns besteht weiterer Forschungsbedarf.

Abbildungsverzeichnis

Literaturverzeichnis

[AIS+77] ALEXANDER, Christopher ; ISHIKAWA, Sara ; SILVERSTEIN, Murray ; JACOBSON, M. ; FIKSDAHL-KING, I. ; ANGEL, S.: *A Pattern Language — Towns, Buildings, Construction*. New York : Oxford University Press, 1977 (Center for Environmental Structure Series). – ISBN 0–19–501919–9

[Ale79] ALEXANDER, Christopher: *The timeless way of building*. New York : Oxford University Press, 1979

[Ale99] ALEXANDER, Christopher: The origins of pattern theory. In: *IEEE Software* 16 (1999), September/October, Nr. 5, S. 71–81

[Ant96] ANTHONY, Dana L. G.: Patterns for Classroom Education. In: VLISSIDES, John M. (Hrsg.) ; COPLIEN, James O. (Hrsg.) ; SCHMIDT, Douglas C. (Hrsg.): *Pattern Languages of Program Design* Bd. 2. 2. Addison–Wesley, 1996. – ISBN 0–201–89527–7, S. 391–406

[App98] APPLETON, Brad: Forces Vs. Consequences. In: *Portland Pattern Repository* (1998), November. – http://c2.com/cgi/wiki?ForcesVsConsequences

[BC87] BECK, Kent ; CUNNINGHAM, Ward: Using Pattern Languages for Object-Oriented Programs / Computer Research Laboratory, Tektronix, Inc. 1987 (CR-87-43). – Forschungsbericht. (OOPSLA-87 Workshop on the Specification and Design for Object-Oriented Programming)

[BC89] BECK, Kent ; CUNNINGHAM, Ward: A Laboratory for Teachnig Object–oriented Thinking. In: *Proceedings of OOPSLA '89, Special Issue of SIGPLAN Notices* 24 (1989), Oktober, Nr. 10, S. 1–6

[BCK98] BASS, Len ; CLEMENTS, Paul ; KAZMAN, Rick: *Software Architecture in Practice*. Addison–Wesley, 1998 (The SEI Series in Software Engineering)

[BH02] BUSCHMANN, Frank ; HENNEY, Kevlin: A Distributed Computing Pattern Language. In: *Proceedings of the EuroPLoP 2002*, 2002

[BMR+96] BUSCHMANN, Frank ; MEUNIER, Regine ; ROHNERT, Hans ; SOMMERLAD, Peter ; STAL, Michael: *Pattern-oriented Software Architecture — A System of Patterns*. Bd. 1. John Wiley & Sons, 1996. – ISBN 0–471–95869–7

[Bun98] BUNGERT, Andreas: *Beschreibung programmierter Systeme mittels Hierarchien intuitiv verständlicher Modelle*, Universität Kaiserslautern, Diss., Juni 1998. – Shaker Verlag Aachen, ISBN 3-8265-3911-7

[Cop91] COPLIEN, James O.: *Advanced C++ Programming Styles and Idioms*. Addison–Wesley, August 1991. – ISBN 0–201–54855–0

[Cop94] COPLIEN, James O.: Generative pattern languages: An emerging direction of software design. In: *SIGS C++ Report Magazine* (1994), Juli, S. 18–22

[Cop95] COPLIEN, James O.: A generative development - Process pattern language. In: COPLIEN, James O. (Hrsg.) ; SCHMIDT, Douglas C. (Hrsg.): *Pattern Languages of Program Design* Bd. 1. Addison–Wesley, 1995. – ISBN 0–201–60734–4, S. 183–237

[Cop98] COPLIEN, James O.: Software design patterns: Common questions and answers. In: RISING, Linda (Hrsg.): *The Patterns Handbook*. New York : Cambridge University Press, 1998. – ISBN 0–521–64818–1, S. 311–320

[Cor98] CORFMAN, Russell: An Overview of Patterns. In: RISING, Linda (Hrsg.): *The Patterns Handbook*. New York : Cambridge University Press, 1998. – ISBN 0–521–64818–1, S. 19–29

[EK03] EDEN, Ammon H. ; KAZMAN, Rick: Architecture, Design, Implementation. In: *The 25th International Conference on Software Engineering (ICSE'03)*. Portland, OR, USA, Mai 2003, S. 149–159

[Fow97] FOWLER, Martin: *Analysis Patterns: Reusable Object Models*. Addison–Wesley, 1997 (Series in Object–Oriented Software Engineering). – ISBN 0–201–89542–0

[GHJV94] GAMMA, Erich ; HELM, Richard ; JOHNSON, Raph ; VLISSIDES, John: *Design Patterns — Elements of Reusable Object–Oriented Software*. Addison–Wesley, 1994 (Professional Computing Series). – ISBN 0–201–63361–2

[GK03] GRÖNE, Bernhard (Hrsg.) ; KELLER, Frank (Hrsg.): Conceptual Architecture Patterns / Hasso-Plattner-Institut für Softwaresystemtechnik an der Universität Potsdam. Potsdam, 2003 (2). – Forschungsbericht. – ISBN 3–935024–98–3

[GKKS04] GRÖNE, Bernhard ; KNÖPFEL, Andreas ; KUGEL, Rudolf ; SCHMIDT, Oliver: The Apache Modelling Project / Hasso-Plattner-Institut für Softwaresystemtechnik. Potsdam, 2004 (5). – Forschungsbericht. Web site: apache.hpi.uni-potsdam.de. – ISSN 1613–5653

[GT03] GRÖNE, Bernhard ; TABELING, Peter: A System of Conceptual Architecture Patterns for Concurrent Request Processing Servers. In: *Proceedings of the Second Nordic Conference on Pattern Languages of Programs (VikingPLoP 2003)*. Bergen, Norway, 2003

[HL03] HÜBNER, Konrad ; LÜCK, Einar: Modeling of the Broker Architectural Framework. In: GRÖNE, Bernhard (Hrsg.) ; KELLER, Frank (Hrsg.): *Conceptual Architecture Patterns*. Potsdam, 2003 (2). – ISBN 3–935024–98–3, S. 12–17

[HNS00] HOFMEISTER, Christine ; NORD, Robert ; SONI, Dilip: *Applied Software Architecture*. Addison–Wesley, 2000 (Object technology series). – ISBN 0–201–32571–3

[IEE00] IEEE STANDARD 1471-2000. *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*. 2000

[Kel03] KELLER, Frank: *Über die Rolle von Architekturbeschreibungen im Software–Entwicklungsprozess*, Universität Potsdam, Diss., August 2003

[Kle99] KLEIS, Wolfram: *Konzepte zur verständlichen Beschreibung objektorientierter Frameworks*, Universität Kaiserslautern, Diss., 1999. – Shaker Verlag Aachen, ISBN 3-8265-6754-4

[Kno04] KNOEPFEL, Andreas: *Konzepte der Beschreibung interaktiver Systeme*, Hasso–Plattner–Institut für Softwaresystemtechnik an der Universität Potsdam, Diss., 2004

[Koe98] KOENIG, Andrew: Patterns and Antipatterns. In: LINDA RISING (Hrsg.): *The Patterns Handbook*. Cambridge University Press, 1998. – ISBN 0–521–64818–1, S. 383–389

[KTG+02] KELLER, Frank ; TABELING, Peter ; GRÖNE, Bernhard ; KNÖPFEL, Andreas ; SCHMIDT, Oliver ; KUGEL, Rudolf ; APFELBACHER, Rémy: Improving knowledge transfer at the architectural level: Concepts and notations. In: ARABNIA, Hamid R. (Hrsg.) ; MUN, Youngsong (Hrsg.): *Proceedings of the SERP'02, the international conference on software engineering research and practice, las vegas*, CSREA Press, Juni 2002. – ISBN 1–892512–99–8, S. 101–107

[Lea99] LEA, Doug: *Concurrent Programming in Java: Design Principles and Patterns*. 2nd Edition. Addison–Wesley, 1999 (The Java Series). – ISBN 0–201–31009–0

[McK96] MCKENNEY, Paul: Force Context Duality. In: *Portland Pattern Repository* (1996), Mai. – http://c2.com/cgi/wiki?ForceContextDuality

[PW92] PERRY, Dewayne E. ; WOLF, Alexander L.: Foundations for the Study of Software Architecture. In: *ACM SIGSOFT, Software Engineering Notes* 17 (1992), Oktober, Nr. 4, S. 40–52

[Rec91] RECHTIN, Eberhard: *Systems Architecting*. Englewood Cliffs NJ, USA : Prentice Hall, 1991

[RG03] RÖCK, Stefan ; GIERAK, Alexander: Evaluating and Extending the Component Configurator Pattern. In: GRÖNE, Bernhard (Hrsg.) ; KELLER, Frank (Hrsg.): *Conceptual Architecture Patterns*. Potsdam, 2003 (2). – ISBN 3–935024–98–3, S. 28–34

[RZ96] RIEHLE, Dirk ; ZÜLLIGHOVEN, Heinz: Understanding and Using Patterns in Software Development. In: *Theory and Practice of Object Systems* 2 (1996), Nr. 1, S. 3–13

[SC96] SCHMIDT, Douglas C. ; CRANOR, Charles D.: Half-Sync/Half-Async: An Architectural Pattern for Efficient and Well-Structured Concurrent I/O. In: VLISSIDES, John M. (Hrsg.) ; COPLIEN, James O. (Hrsg.) ; KERTH, Norman L. (Hrsg.): *Pattern Languages of Program Design 2*, Addison–Wesley, 1996. – ISBN 0–201–89527–7, S. 437–459

[Sch98] SCHMIDT, Douglas C.: Evaluating architectures for multithreaded object request brokers. In: *Communications of the ACM* 41 (1998), Nr. 10, S. 54–60. – ISSN 0001–0782

[SG96] SHAW, Mary ; GARLAN, David: *Software Architecture*. Prentice-Hall, 1996. – ISBN 0–13–182957–2

[Sha96] SHAW, Mary: Some Patterns for Software Architectures. In: VLISSIDES, John M. (Hrsg.) ; COPLIEN, James O. (Hrsg.) ; KERTH, Norman L. (Hrsg.): *Pattern Languages of Program Design 2*, Addison–Wesley, 1996. – ISBN 0–201–89527–7, S. 255–269

[SNH95] SONI, Dilip ; NORD, Robert L. ; HOFMEISTER, Christine: Software Architecture in Industrial Applications. In: *International Conference on Software Engineering*, 1995, S. 196–207

[SOK+00] SCHMIDT, Douglas C. ; O'RYAN, Carlos ; KIRCHER, Michael ; PYARALI, Irfan ; BUSCHMANN, Frank: Leader/Followers: A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching. In: *Proceedings of the 7th Pattern Languages of Programs Conference*. Allerton Park, Illinois, USA, August 2000

[SPM96] SILVA, Antonio R. ; PEREIRA, Joao ; MARQUES, Jose A.: Object Synchronization Pattern. In: *Proceedings of the EuroPLoP 1996*, 1996

[SSRB00] SCHMIDT, Douglas ; STAL, Michael ; ROHNERT, Hans ; BUSCHMANN, Frank: *Pattern-oriented Software Architecture Vol. 2 — Patterns for Concurrent and Networked Objects*. John Wiley & Sons, 2000 (Series in Software Design Patterns). – ISBN 0–471–60695–2

[TG03] TABELING, Peter ; GRÖNE, Bernhard: Mappings between Object–oriented Technology and Architecture–based models. In: AL-ANI, Ban (Hrsg.) ; ARABNIA, Hamid R. (Hrsg.) ; MUN, Youngsong (Hrsg.): *Proceedings of the SERP'03, the international conference on software engineering research and practice, Las Vegas* Bd. II, CSREA Press, Juni 2003. – ISBN 1–932415–20–3, S. 568–574

[VSW02] VÖLTER, Markus ; SCHMID, Alexander ; WOLFF, Eberhard: *Server component patterns — Component infrastructures illustrated with EJB*. John Wiley & Sons, 2002 (Series in Software Design Patterns). – ISBN 0–470–84319–5

[Wen70] WENDT, Siegfried: Eine Methode zum Entwurf komplexer Schaltwerke unter Verwendung spezieller Ablaufdiagramme. In: *Elektronische Rechenanlagen* 12 (1970), Nr. 6, S. 314–323

[Wen79] WENDT, Siegfried: The programmed action module: An element for system modelling. In: *Digital Processes* 5 (1979), S. 213–222

[Wen82a] WENDT, Siegfried: Einführung in die Begriffswelt allgemeiner Netzsysteme. In: *Regelungstechnik* 30 (1982), Nr. 1

[Wen82b] WENDT, Siegfried: Der Kommunikationsansatz in der Software-Technik. In: *Data Report* 17 (1982), Nr. 4

[Wen91] WENDT, Siegfried: *Nichtphysikalische Grundlagen der Informationstechnik - Interpretierte Formalismen*. 2. Heidelberg : Springer Verlag, 1991. – ISBN 3–540–54452–6

[Wik02] WIKI: Pattern Forms. In: *Portland Pattern Repository* (2002), August. – http://c2.com/cgi/wiki?PatternForms

Index