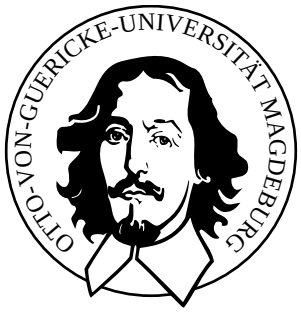
Laura Marnitz

Werkzeugunterstützung für die Merkmalorientierte Softwareentwicklung

Otto-von-Guericke-Universität Magdeburg



Fakultät für Informatik
Institut für Technische und Betriebliche Informationssysteme

Diplomarbeit

Werkzeugunterstützung für die Merkmalorientierte Softwareentwicklung

Autorin:

Laura Marnitz

8. September 2005

Betreuer:

Prof. Dr. rer. nat. habil. Gunter Saake

Dipl.-Wirtsch.-Inf. Thomas Leich

Dipl.-Inf. Sven Apel

Universität Magdeburg
Fakultät für Informatik
Postfach 4120, D–39016 Magdeburg

Marnitz, Laura:
Matrikelnummer: 157454
Werkzeugunterstützung für die Merkmalorientierte Softwareentwicklung
Diplomarbeit, Otto-von-Guericke Universität Magdeburg, 2005.

Danksagung

An dieser Stelle möchte ich meinen Dank den Menschen aussprechen, die mich bei der Erstellung dieser Diplomarbeit unterstützt haben. Dieser Dank gilt vor allem meinen Betreuern Prof. Dr. rer. nat. habil. Gunter Saake, Dipl.-Wirtsch.-Inf. Thomas Leich und Dipl.-Inf. Sven Apel. Intensive Diskussionen haben wesentlich zur Enstehung dieser Arbeit beigetragen und darüber hinaus eine gemeinsame Veröffntlichung ermöglicht. Dipl.-Inf. Marcel Götze möchte ich für wertvolle Hinweise im Bereich der Visualisierung danken.
Bei Anja, Petra, Sandra und Uta möchte ich mich für die gute „Betreuung" in den letzten Wochen bedanken und überhaupt für eine Studienzeit, die durch sie erst richtig schön wurde.
Meinen Eltern, meinem Freund und meinen Schwestern Hanna, Lena und Maria danke ich für die geduldige Begleitung meiner Diplomarbeit, ein offenes Ohr, zahlreiche Videokonferenzen und diverse Vorschläge bzgl. Rechtschreibung und Grammatik.

Inhaltsverzeichnis

Abbildungsverzeichnis

KAPITEL 1

Einleitung

1.1 Motivation

Softwareprodukte können in zwei Kategorien aufgeteilt werden. Es gibt Individualsoftware[1], die speziell auf die Belange des Kunden zugeschnitten wird und es gibt Standardsoftware[2], die durch Indivisualisierung bzw. Parametrisierung an den jeweiligen Kunden angepasst wird. Beide Ansätze haben Vor- und Nachteile. Maßgefertigte Software hat mehrere Vorteile: Sie kann optimal auf das Anwendungsgebiet und damit auf die Kundenwünsche zugeschnitten werden. Darüber hinaus können auch geschütztes Wissen oder spezielle Verfahren genutzt werden, die gegenüber anderen Konkurrenten einen Wettbewerbsfaktor darstellen. Diese individualisierte Softwareentwicklung ist jedoch sehr kostenintensiv. Da Wartung und Weiterentwicklung dieser Systeme ebenfalls teuer sind, entstehen häufig nur suboptimale Lösungen.
Standartsoftware wird durch die Verteilung der Kosten auf mehrere Kunden preiswerter. Dies wirkt sich ebenfalls positiv auf die Wartung und Weiterentwicklung aus. Diese Vorteile werden mit geringer Anpassbarkeit von Standardsoftware erkauft. Die Folge daraus kann sein, dass ganze Arbeitsprozesse angepasst werden müssen.
Im Bereich der eingebetteten Systeme können Standardprodukte nur selten eingesetzt werden. Durch die Bereiststellung der Funktionalität für eine möglichst breite Domäne entsteht häufig ein so großer Overhead, dass ein Einsatz auf den eingebetteten Geräte mit geringen Ressourcen nicht möglich ist.

1 engl: Special Purpose Software
2 engl: General Purpose Software

Eine mögliche Lösung bieten die seit längerer Zeit in der Softwaretechnik diskutierten Konzepte von Programmfamilien und Produktlinien. Sie ermöglichen die Austauschbarkeit von Teilen/Modulen zwischen verschiedenen Produkten. Diese Idee ist schon sehr alt. 1826 führte JOHN HALL nach 25 Jahren „Tüftelei" den Austauch von Teilen zwischen verschiedenen Gewehrvarianten im Musketenbau ein. Heute sind Produktlinie beispielsweiese im Fahrzeugbau europäischer Hersteller sehr akuell. So fahren viele Modelle der *Volkswagengruppe*[3], die sich außerlich kaum ähneln, auf der gleichen Bodengruppe, verwenden baugleiche oder nur minimal angepasste Motoren, Elektronikbauteile oder Getriebe. Dies ermöglich eine kostengünstige Produktion bei gleichzeitig hoher Variabltät der Produkte, die den sehr großen Individualisierungswünschen des europäischen Marktes entspicht.
Im Softwareproduktbereich steigt derzeitig auch das Bedürfnis nach indidualivierte Software. Dies hat unterschiedliche Gründe. So haben beispielsweise Unternehmen erkannt, dass individuell angepasste Software einen Wettbewerbsvorteil darstellt. Hersteller von Infrastruktursoftware im Bereich der eingebetteten Systeme haben festgestellt, dass die Variablität der Hardware und das Anwendungsspektrum dieser Geräte so stark zugenommen haben. Eine kosteneffektive Entwicklung solcher Software ist möglich, wenn sie über eine wohldefinierte Strategie erfolgt. Produktlinien in der Softwareentwicklung bieten ausreichend Flexibilität bzgl. einer Anwendungsdomäne bei gleichzeitig guter Unterstützung der Wiederverwenbarkeit von gemeinsamen genutzten Teilfunktionalitäten.

1.2 Problem- und Zielstellung

Für die Entwicklung von Produktlinien und Programmfamilien stehen eine Reihe von Methoden und Konzepten zur Verfügung. Eine prominente Vorgehensweise stellt das Domänen Engeering dar. Ziel des Domänen Engeerings ist die Entwicklung einer Familie von Programmen. Aufbauend auf einer gemeinsamen Basis sollen spezielle Familiemitglieder schrittweise abgeleitet werden.
Derzeitig existieren auch für die einzelnen Phasen: Analyse, Entwurf und Implementierung, verschiedene Verfahren. Die Integration der einzelnen Phasen in ein Werkzeug ist jedoch noch nicht umgesetzt worden. Dies führt bei der Komplexität solcher Programmfamilien zu Fehlern, Inkonsistenzen zwischen den Phasen und Mehrarbeit

3 Volkswagen, Seat, Skoda und Audi

und damit verbundene Kosten. Um die Entwicklung in Produktlinien und Programmfamilien einer breiten Masse von Entwicklern als Lösung für viele derzeitige Probleme bei der Erstellung variabler Software zu präsentieren, müssen Werkzeuge entwickelt werden, die den heutigen Werkzeugen der Objektorientierten-Entwicklung ähnlich sind.
Eine geeignete Basis für die Erstellung von Programmfamilien und Produktlinien bietet das AHEAD-Architektur- und Implementierungsmodell (BATORY et al. (BSR03)). Eine Integration diese Modells in den gesamten Prozess des Domänen Engeerings und die damit verbundene Unterstützung durch ein Werkzeug sind Thema der vorliegenden Diplomarbeit. Auf Grund der Komplexität des Entwicklungsprozess von Produktlinien und Programmfamilien steht eine geeinigte Automatisierung von Teilprozessen und eine optimierte Anwender-Interaktion und Visualisierung im Vordergrund dieser Arbeit.

Ziel dieser Diplomarbeit ist die Anforderungsanalyse, der Entwurf und die Implementierung eines Werkzeuges, das den Prozess des Domänen Engeerings konsistent zusammenführt. Hierbei soll die Unterstützung der Merkmalsorientierten Programmierung in besonderer Weise berücksichtigt werden.

1.3 Gliederung der Arbeit

Für das Verständnis dieser Arbeit sind softwaretechnische Kenntnisse der Programmfamilienentwicklung, sowie der Merkmalorientierten Programmierung notwendig. Die erforderlichen Grundlagen über existierende Methode und Verfahren werden in Kapitel 2 gegeben. Anschließend werden in Kapitel 3 kurz verschiedene Techniken der Informationsvisualisierung, insbesondere der Visualisierung von Software und hierarchischer Daten, zusammengefasst. Diese Informationen sind notwendig für die Lektüre der weiteren Arbeit, da das zu entwickelnde Werkzeug die Arbeit des Anwenders mithilfe der Informationsvisualisierung unterstützen soll. Kapitel 4 befasst sich mit der Analyse und dem Entwurf des zu entwickelnden Werkzeugs. Hierbei wird untersucht, wie die verschiedenen Phasen des Domain Engineerings integriert unterstützt werden können. Ein Schwerpunkt bildet hierbei die Frage mit welchen Mitteln die Merkmalorientierte Programmierung unter Verwendung der *AHEAD-Toolsuite* benutzerfreundlich gestaltet werden kann. Im folgenden Kapitel 5 wird die Umsetzung des

entwicklenden Prototyps als Eclipse-Plugin beschrieben. Die Arbeit schließt mit einer Zusammenfassung und einem Ausblick in Kapitel 6.

KAPITEL 2

Softwaretechnische Grundlagen

Schlüsselfaktoren in der Softwareentwicklung sind kurze Entwicklungszeiten, eine hohe Produktqualität, niedrige Entwicklungskosten und einfache Wartung. Daraus entsteht die Notwendigkeit Software so zu entwickeln, dass Teile der Software oder des Sourcecodes in anderen Systemen wiederverwendet werden können.

Wiederverwendung

Die genannten Ziele sollen durch *Wiederverwendung* bereits entwickelter (und getesteter) Softwaremodule erreicht werden, die Codefragmente kapseln. McIlroy (McI68) führte als Erster das Konzept einer formellen Softwarewiederverwendung ein und nahm hierfür die Hardwareindustrie als Vorbild:

> *„Software engineers should take a look at the hardware industry and establish a software component subindustry. The produced software components should be tailored to specific needs but be reusable in many software systems."*

Die genannten Ziele der Wiederverwendung können jedoch nur erreicht werden, wenn die Wiederverwendung von Software systematisch und formal geplant wird (Prieto-Diaz (PD93)). Ein Ansatz hierzu ist die Entwicklung von Programmfamilien.
In diesem Kapitel werden die softwaretechnischen Grundlagen der Programmfamilienentwicklung erarbeitet. Abschnitt 2.2 stellt den dreistufigen Prozess des Domain Engineerings vor, dessen Ziel die systematische Entwicklung einer Programmfamilie ist. Anschließend werden mit der *Merkmalorientierten Domänenanalyse* (Abschnitt 2.3), dem *Kollaborationsentwurf* (Abschnitt 2.4) und der *Merkmalorientierten Programmierung* (Abschnitt 2.6) eine Reihe verschiedener Methoden vorgestellt, mit deren Hilfe die einzelnen Phasen des Domain Engineerings konkret umgesetzt werden können.

2.1 Programmfamilien

Der Begriff *Programmfamilie* ist schon recht alt und wurde 1979 von PARNAS (Par76) geprägt. PARNAS definiert eine Programmfamilie als eine Menge von Anwendungen, deren gemeinsame Eigenschaften so weitreichend sind, dass es vorteilhaft ist, diese gemeinsamen Eigenschaften erst zu analysieren und zu nutzen, bevor ein einzelnes Mitglied der Familie analysiert wird. Die Mitglieder einer Programmfamilie haben natürlich auch unterschiedliche Eigenschaften. Diese Variationen resultieren aus notwendigen Anpassungen einer Anwendung, z. B. an eine bestimmte Nutzergruppe oder an die Implementierung. Variabilität ist ein wichtiges Qualitätsmerkmal von Programmfamilien.

Variabilität

Unter Variabilität versteht man die Möglichkeit, ein System zu verändern oder anzupassen. Je höher die Variabilität eines Systems ist, desto einfacher können notwendige Änderungen oder Anpassungen vorgenommen werden. Variabilität hängt eng mit der Wiederverwendung zusammen. Wenn Softwaremodule unter verschiedenen Umständen wiederverwendet werden sollen, muss Variabilität möglich sein, um notwendige Anpassungen vornehmen zu können. Variabilität wird durch die „Verzögerung" von Entwurfsentscheidungen möglich. Während der Entwicklung von Programmfamilien wird die Architektur des Systems früh festgelegt. Die Details der Implementierung werden jedoch bis zur Produktimplementierung verzögert. Diese „verzögerten" Entwurfsentscheidungen werden als *Variabilitätspunkte* bezeichnet (SVAHNBERG et al. (SvGB01)).

2.1.1 Programmfamilien vs. Produktlinien

Die Begriffe Programmfamilie und Produktlinie sind miteinander verwandt, haben jedoch im Kern unterschiedliche Bedeutungen[1].
Während die Gemeinsamkeiten innerhalb einer Programmfamilie eher technischer Natur sind, definieren sich die Mitglieder einer Produktlinie über einen gemeinsamen Markt. CLEMENTS et al. (CLN02) definieren eine Produktlinie als eine Menge von Softwareprodukten, welche die speziellen Ansprüche eines bestimmten Marktes erfüllen und eine Anzahl gemeinsamer Komponenten miteinander teilen.

1 Programmfamilien werden auch als *Produktfamilien* bezeichnet.

Die Bausteine einer Produktlinie sind im günstigsten Fall Produktfamilien. In diesem Fall kann eine Produktlinie aus einer oder mehreren Produktfamilien bestehen. Jede Produktfamilie kann dann in einer oder mehreren Produktlinien wiederverwendet werden (CZARNECKI et al. (CE00), BATORY et al. (BLHM02)).
Die Entwicklung von Programmfamilien und Produktlinien erfordert eine systematische Vorgehensweise. Das *Domain Engineering* ist ein dreiphasiger Prozess, mit dessen Hilfe die Programmfamilienentwicklung strukturiert umgesetzt werden kann.

2.2 Domain Engineering

NEIGHBORS (Nei80) macht deutlich, dass der Schlüssel für eine gute Wiederverwendung nicht die Wiederverwendung von Sourcecode ist, sondern die Wiederverwendung von Analyse- und Entwurfsergebnissen. Dies begründet er damit, dass nur „das Recycling" von Analyse- und Entwurfsinformationen eine explizite Darstellung der Struktur und Definition der verwendeten Komponenten bietet. Das Domain Engineering ist ein Softwareentwicklungsprozess, mit dessen Hilfe *Analyse-*, *Entwurfs-* und *Implementierungsinformationen* systematisch in wiederverwendbaren Einheiten zusammengefasst werden. Während der *Anwendungsentwicklung* werden diese wiederverwendbaren Einheiten dann genutzt, um ein konkretes Mitglied der Programmfamilie zu erstellen.

> *„Domain Engineering is the activity of collection, organizing, and storing past experience in building systems or parts of systems in a particular domain in the form of reusable assets (i.e., reusable work products), as well as providing an adequate means for reuse these assets (i.e., retrieval, qualification, dissemination, adaptation, assembly, and so on) when building new systems (*CZARNECKI *et al. (CE00))."*

Das Domain Engineering besteht aus den folgenden drei Teilschritten:

1. die Domänenanalyse,
2. die Domänenentwurf und
3. die Domänenimplementierung.

Parallel zum Domain Engineering verläuft die Anwendungsentwicklung. Ziel der Anwendungsentwicklung ist die Erzeugung eines konkreten Mitglieds der Programmfa-

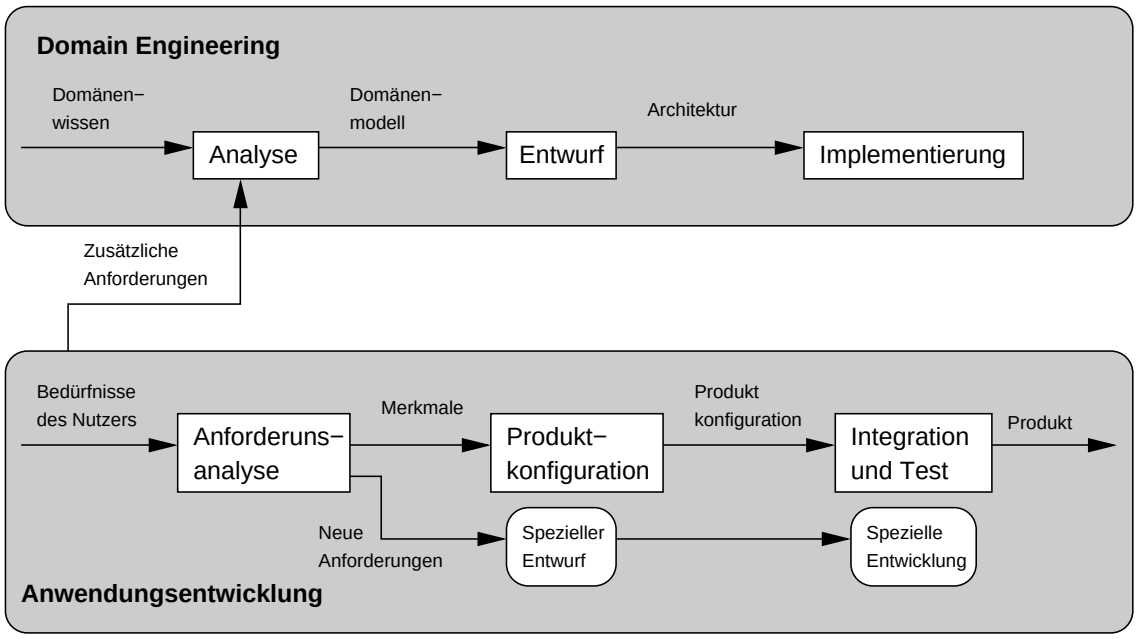


Abbildung 2.1: Die Phasen des Domain Engineerings und die Anwendungsentwicklung als parallele und miteinander interagierende Prozesse, adaptiert von CZARNECKI et al. (CE00).

milie. Hierbei werden die Softwareineinheiten, die während des Domain Engineerings erzeugt wurden, genutzt[2].

Zwingende Merkmale

Für die Umsetzung der drei Phasen des Domain Engineerings gibt es eine Vielzahl von Methoden. Eine Beschreibung aller Methoden würde den Rahmen dieser Arbeit sprengen, daher werden nur die für diese Arbeit relevanten Techniken detailliert dargestellt. Für weitere Methoden wird auf die entsprechende Literatur verwiesen.

2.2.1 Domänenanalyse

Das Hauptziel der Domänenanalyse ist die Identifizierung wiederverwendbarer Anforderungen. Dies setzt voraus, dass die Gemeinsamkeiten und Variabilitätspunkte innerhalb der Programmfamilien herausgearbeitet und in geeigneter Weise dargestellt werden. Hierzu ist ein tieferes Verständnis der *Domäne* notwendig, innerhalb der die Programmfamilie angesiedelt ist.

2 EISENECKER (CE00) fassen diese Beziehung schlagwortartig zusammen. Sie charakterisieren den Prozess des Domain Engineerings als *Entwicklung für die Wiederverwendung* und die Anwendungsentwicklung als *Entwicklung durch Wiederverwendung*.

In der Literatur gibt es eine Reihe unterschiedlicher Definitionen für den Begriff der *Domäne*. Diese Arbeit folgt der Definition von CZARNECKI et al. (CE00) : *Domäne*

„Domain: An area of knowledge

- *scoped to maximize the satisfaction of the requirements of its stakeholders,*
- *including a set of concepts and terminology understood by practitioners in that area, and*
- *including knowledge of how to build software systems (or parts of software systems) in that area."*

Wie bereits erwähnt, gibt es eine Vielzahl von Methoden für die Domänenanalyse. Die meisten Methoden beinhalten in der ein oder anderen Form die folgenden Teilschritte:

1. das Domain Scoping[3] und
2. die Domänenmodellierung.

Während des ersten Teilschritts erfolgt die Abgrenzung der Domäne und relevante Informationen über die Domäne werden zusammengetragen. Wenn die Domäne bezüglich einer Menge existierender Softwaresysteme, die Gemeinsamkeiten teilen, eingegrenzt wird, führt dies zur Entwicklung einer Programmfamilie. Wird die Domäne dagegen aufgrund einer Marketingstrategie eingegrenzt, so entsteht eine Produktlinie. Die zusammengetragenen Informationen werden in ein *Domänenmodell* integriert. Ein Domänenmodell ist eine explizite und implementierungsunabhängige Darstellung der Gemeinsamkeiten und Variationspunkte innerhalb der Domäne.

Geprägt wurde der Begriff der Domänenanalyse durch NEIGHBORS (Nei80), der mit *Draco* eine der ersten Methoden für die Domänenanalyse entwickelte. Die am häufigsten genutzten und am besten dokumentierten Methoden sind die *Merkmalorientierte Domänenanalyse*, die von KANG et al. (KCH$^+$90) entwickelt wurde und das *Organization Domain Modeling*, welches eine Weiterentwicklung der Merkmalorientierten Domänenanalyse durch SIMOS et al. (SCK$^+$96) darstellt. Als weitere bekanntere Methoden sind noch *Capture* von BAILIN (Bai93) und *DARE* [4] von FRAKES et al. (FPDF98) zu nennen.

Die Merkmalorientierte Domänenanalyse wird in Kapitel 2.3 näher erläutert, da sie ein zentraler Begriff dieser Arbeit ist. Ein sehr guter Überblick über weitere Methoden

3 Domänenabgrenzung

4 **D**omain **A**nalysis and **R**euse **E**nvironment

der Domänenanalyse und ihre zeitliche Einordnung findet sich bei CZARNECKI et al. (CE00).

2.2.2 Domänenentwurf

Ziel des Domänenentwurfs ist die Entwicklung einer gemeinsamen Architektur für die Mitglieder der Programmfamilie. Hierbei muss die Architektur die Variabilität innerhalb der Programmfamilie erhalten. Während des Domänenentwurfs wird ein Produktionsplan erstellt, der beschreibt, wie eine konkrete Applikation aus der Architektur und den wiederverwendbaren Einheiten erzeugt werden kann. Dazu gehören Beschreibungen der Schnittstellen für jeden Nutzer, Richtlinien für die Zusammenstellung der Komponenten und Regeln für Änderungsanforderungen. Der Kombinationsprozess der Komponenten kann manuell, semi-automatisch oder automatisch erfolgen.

2.2.3 Domänenimplementierung

Während der Domänenimplementierung werden die Architektur, die Komponenten und Werkzeuge, die während der Entwurfsphase theoretisch entwickelt wurden, umgesetzt. Dies umfasst z. B. auch die Dokumentation und Implementierung domänenspezifischer Sprachen und Generatoren.

Komponenten

Es ist ein übliches Vorgehen ein komplexes Problem in Teilprobleme zu zerlegen, um aus den Teillösungen die Lösung des gesamten Problems zu erhalten. Diese Strategie kann auch für die Erstellung von Software genutzt werden. Das Ziel ist die Erstellung komplexer Softwaresysteme durch die Kombination kleinerer Softwareeinheiten, die *Komponenten* genannt werden. Eine sehr allgemeine Definition einer Komponente liefern CZARNECKI et al. (CE00). Sie definieren Komponenten als Bausteine, aus denen sich Softwaresysteme zusammensetzen. Die Definition von SZYPERSKI (Szy98) ist weniger abstrakt:

> *„A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."*

Konkrete Beispiele für Komponenten sind *JavaBeans* oder *Mixin-Layers* (SMARAGDAKIS et al. (SB98), vgl. auch Abschnitt 2.4.3). Der Vorteil komponentenbasierter Verfahren besteht darin, dass sie die Anpassungsfähigkeit von Softwaresystemen erhöhen, da Systeme durch Hinzufügen, Entfernen oder Verändern von Komponenten einfach modifiziert werden können. Diese Eigenschaft machen den Einsatz von Komponenten zu einer der effektivsten Methoden, um Wiederverwendung zu gewährleisten.

Komponentenkombination

Die Kombination von Komponenten geschieht über Schnittstellen. Die Schnittstellen sind der Teil der Komponenten, der „von aussen" sichtbar ist (BATORY et al. (BO92)). Die Fähigkeit interne Details zu verbergen, wird als *Kapselung* bezeichnet und reduziert die Abhängigkeiten zwischen den Komponenten. Man unterscheidet *Whitebox*- und *Blackbox*-Komponenten. Whitebox-Komponenten erlauben dem Anwender die Anpassung und Veränderung ihrer Implementierung. Im Gegensatz hierzu verfügt der Nutzer bei Blackbox-Komponenten ausschließlich über Informationen bezüglich der Schnittstelle der Komponente und ihrer offiziellen (evtl. vertraglich vereinbarten) Spezifikation. Über die interne Realisierung der Komponente hat der Anwender keinerlei Kenntnisse. Besonders bei Blackbox-Komponenten hängt daher die Qualität einer Komponente von ihrer Schnittstellenspezifikation ab. Durch eine schlecht spezifizierte Schnittstelle kann die Wiederverwendung einer Komponente eingeschränkt und fehleranfällig werden.

Komponenten und Programmfamilien

Systematisch entwickelte Programmfamilien ermöglichen eine effiziente Wiederverwendung von Komponenten. Zugleich wird die Entwicklung von Programmfamilien durch Komponenten deutlich weniger zeit- und kostenintensiv, wenn die Eigenschaften, die während der Domänenanalyse identifiziert wurden, auf Komponenten oder Kombinationen von Komponenten abgebildet werden können. Gemeinsame Eigenschaften innerhalb der Programmfamilie werden auf wiederverwendbare Komponenten abgebildet. Dies gewährleistet die Gleichartigkeit identischer Eigenschaften innerhalb einer Programmfamilie. Wird eine Komponente, die Teil einer bestimmten Anwendung ist, verbessert, so kann diese Erweiterung einfach in andere Produkte, welche die gleiche Komponente enthalten, propagiert werden.

2.3 Merkmalorientierte Domänenanalyse

Die Merkmalorientierte Domänenanalyse (FODA[5]) wurde 1990 durch KANG et al. (KCH+90) am Carnegie Mellon Sofware Engineering Institute (SEI) entwickelt[6]. Sie führen die *Merkmalmodellierung* als zentrale Tätigkeit der Domänenanalyse ein. Die Merkmalorientierte Domänenanalyse beschreibt die Gemeinsamkeiten und Variationen innerhalb einer Programmfamilie mit Hilfe von *Merkmalen*.

2.3.1 Merkmale

Allgemein ausgedrückt sind Merkmale die markanten Eigenschaften einer Anwendung. KANG et al. (KCH+90) definieren Merkmale als Anforderung, die „von aussen" sichtbar sind. Interne Eigenschaften werden während der Merkmalmodellierung nicht berücksichtigt:

> *„Feature: A prominent or distinctive user-visible aspect, quality, or characteristic of a software system or systems."*

Diese Arbeit folgt der Merkmaldefinition von SIMOS et al. (SCK+96), da sie offener formuliert ist und auch Eigenschaften auf Implementierungsebene einschließt:

> *„Feature: A distinguishable characteristic of a concept (e.g., system, component, and so on) that is relevant to some stakeholder[7] or the concept."*

Merkmale vs. Concerns

Die Strukturierung einer Programmfamilie nach Merkmalen folgt einem bekannten und relativ alten Prinzip der Informatik, dem *Separation of Concerns*[8] (DIJKSTRA (Dij76)). CZARNECKI1998 (Cza98) fasst den Kern dieses Prinzips wie folgt zusammen:

> *„The principle acknowledges that we cannot deal with many issues at once, but rather with one at a time."*

5 engl. **F**eature **O**riented **D**omain **A**nalysis

6 Das SEI hat die *Product Line Practice*-Initiative (PLP) begründet, die als Ziel die Optimierung der Produktlinienentwicklung hat. Siehe auch: http://www.sei.cmu.edu/productlines/plp_init.html

7 *Stakeholder* sind alle Personen, Gruppen oder Organisationen, die von der Systementwicklung und natürlich auch vom Einsatz und Betrieb eines Systems in irgendeiner Weise betroffen sind. Dazu gehören auch Personen, die nicht bei der Systementwicklung mitwirken, aber das neue System z. B. nutzen oder in Betrieb halten.

8 Trennung von Belangen.

Die Trennung oder Kapselung von Belangen erleichtert die Änderungen, das Debugging und das Verstehen einer Anwendung. Ein Beispiel für einen Concern des Objektorientierten Entwurfs sind z. B. Klassen (Ossher (OT00)).
Für den Begriff Concern findet sich in der softwaretechnischen Literatur keine einheitliche Definition. (STANLEY et al. (SMSR02), TARR et al. (TO00), KANDÉ (Kan03)). Diese Arbeit folgt der Definition des *Institute of Electrical and Electronics Engineers, Inc. (IEEE)* (IEE00):

> *„Concerns are those interests which pertain to the system's development, its operation or any other aspects that are critical or otherwise important to one or more stakeholders."*

Crosscutting Concerns

Der Grundsatz des Separation of Concerns verlangt, dass Belange klar gekapselt, wohl lokalisiert und explizit repräsentiert werden. *Crosscutting Concerns* treten dann auf, wenn mit der gewählten Entwurfs- und Implementierungsmethode ein Belang nicht sauber gekaspelt werden kann. Ein Crosscutting Concern „schneidet" andere Belange und verletzen so den Grundsatz der Kapselung. Bekannte Beispiele für Crosscutting Concerns in der Objektorientierten Programmierung sind Synchronisations- oder Loggingfunktionalitäten.
Ein Ziel bei der Entwicklung neuer Techniken für das Software Engineering ist die vollständige Separierung aller Concerns, da dies mit einer optimalen Wiederverwendbarkeit einhergehen würde.

Merkmalinteraktion

Analog zu Concerns bilden auch Merkmale nur selten vollständig voneinander unabhängige Einheiten. Andernfalls wäre es wenig sinnvoll sie in ein Produkt zu integrieren (LEE et al. (LKL02)). Das Löschen oder die Änderung eines Merkmals kann somit Auswirkungen auf andere Merkmale haben. Diese *Merkmalinteraktionen* erschweren die Abbildung einzelner Merkmale auf Komponenten der Implementierung, da die miteinander interagierenden Merkmale bei der Umsetzung schwierig zu kapseln sind [9].

9 GIBSON (Gib97) charakterisiert Merkmalinteraktion wie folgt: *„Feature interaction is defined as a characteristic of a system whose complete behavior does not satisfy the separate specifications of all its features."*

2.3.2 Merkmalmodellierung

Ziel der Merkmalmodellierung ist die Identifizierung der Merkmale, welche die Systeme innerhalb einer Domäne gemeinsam haben oder voneinander unterscheiden. Die Merkmalmodellierung hilft bei der Vermeidung zweier Probleme:

1. die Nicht-Integration wichtiger Merkmale und Variationspunkte und
2. die Integration von Merkmalen und Variationspunkten, die nicht benutzt werden und unnötige Komplexität und Kosten verursachen.

Ein Merkmalmodell bietet eine abstrakte (implementierungsunabhänige) und explizite Repräsentation der Variationspunkte innerhalb der Programmfamilie. Dargestellt werden solche Modelle in Baumdiagrammen. Die erste Notation für ein solches Baumdiagramm wurde durch KANG et al. (KCH+90) vorgestellt. Im Rahmen dieser Arbeit wird die erweiterte Notation von CZARNECKI et al. (CE00) verwendet, da sie eine detailreichere Modellierung zulässt. Dieses Modell unterscheidet vier verschiedene Merkmalstypen[10]:

- zwingend
- optional
- alternativ
- oder

Optionale Merkmale, alternative Merkmale und Oder-Merkmale stellen Variationspunkte innerhalb der Programmfamilie dar. Ein Leitfaden für die Merkmalmodellierung findet sich bei CZARNECKI et al. (CE00), sowie bei LEE et al. (LKL02).

Merkmalsdiagramm

Ein Merkmaldiagramm besteht aus einer Menge von Knoten, Kanten und Kantendekorationen. Durch die Anordnung der Merkmale in einem Baumdiagramm wird eine hierarchische Klassifizierung der Merkmale vorgenommen, wobei der Abstraktionsgrad von der Wurzel zu den Blättern abnimmt. Der Name des Wurzelmerkmals entspricht daher der Bezeichnung der Programmfamilie (KANG et al. (KCH+90)). Bei Verbindung zweier Merkmale durch eine Kante, kennzeichnen die Kantendekorationen den Merkmalstyp der Kinderknoten. Abbildung 2.2 zeigt das Merkmaldiagramm für einen Speichermanager.

10 Eine andere mögliche Kategorisierung findet sich bei SVAHNBERG et al. (SvGB01).

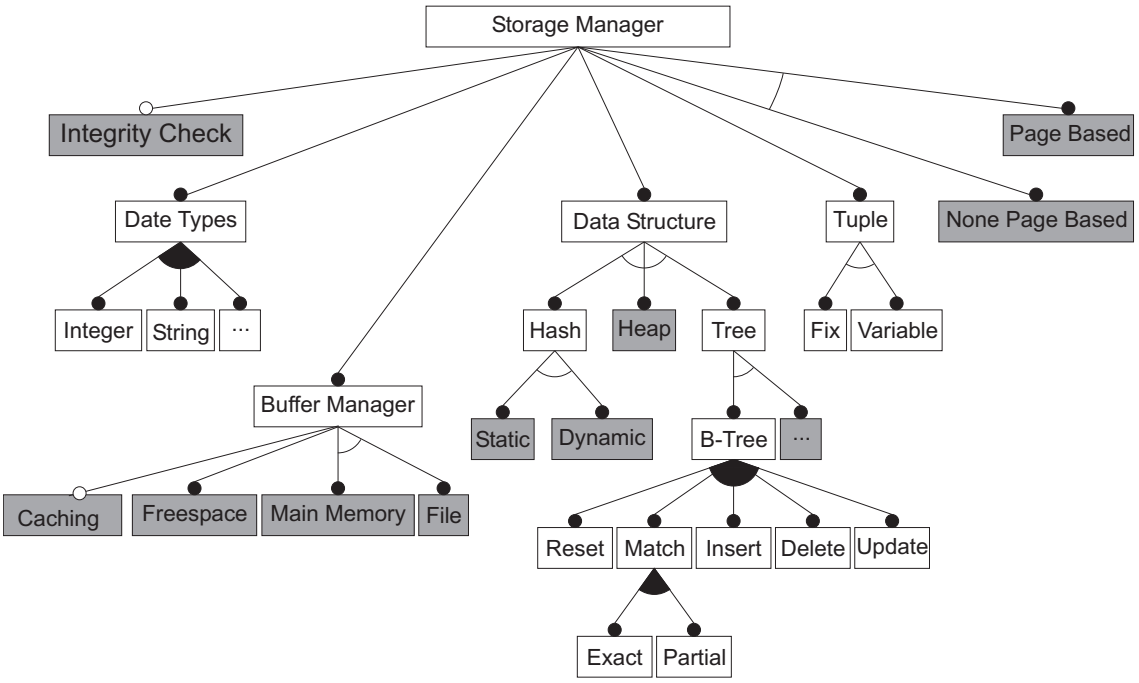


Abbildung 2.2: Merkmaldiagramm eines Speichermanagers, adaptiert von LEICH et al. (LAS05).

Zwingende Merkmale

Ein zwingendes Merkmal ist Teil einer Anwendung, die aus einer Programmfamilie erzeugt werden kann, wenn das Elternmerkmal auch Teil der Anwendung ist. Gekennzeichnet wird ein zwingendes Merkmal durch einen schwarz gefüllten Kreis (vgl. Abbildung 2.3).

Optionale Merkmale

Optionale Merkmale können Teil einer Anwendung sein, wenn der Elternknoten auch Teil der Anwendung ist. Ein optionales Merkmal wird durch einen leeren Kreis gekennzeichnet (vgl. Abbildung 2.3).

Alternative Merkmale

Ein alternatives Merkmal gehört immer zu einer Menge Merkmale, die ein gemeinsames Elternmerkmal besitzen. Wenn das Elternmerkmal einer solchen Menge ausgewählt wurde, darf höchstens ein Merkmal dieser Menge Teil einer Anwendung sein. Eine Menge alternativer Merkmale wird durch einen Bogen gekennzeichnet (vgl. Abbildung 2.3).

Oder-Merkmale

Oder-Merkmale gehören ebenfalls immer zu einer Menge von Oder-Merkmalen, die ein gemeinsames Elternmerkmal besitzen. Wenn der Elternknoten einer solchen Menge ausgewählt wurde, können ein oder mehrere Merkmale der Menge ausgewählt

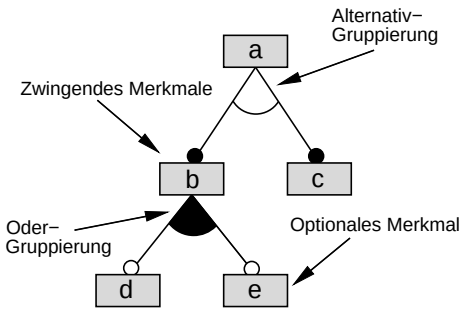


Abbildung 2.3: Schematische Darstellung eines Merkmaldiagramms.

werden. Eine Menge von Oder-Merkmalen wird durch einen gefüllten Bogen gekennzeichnet (vgl. Abbildung 2.3).

2.4 Kollaborationsbasierter Entwurf

Der Kollaborationsbasierte Entwurf ist eine Möglichkeit den Domänenentwurf umzusetzen. Diese Form des Entwurfs gehört zu den objektorientierten Entwürfen. Es wird davon ausgegangen, dass der Leser bereits über Kenntnisse der Objektorientierte Programmierung verfügt, so dass in diesem Abschnitt nur eine kurze Zusammenfassung der fundamentalen Konzepte und Begriffe der Objektorientierten Programmierung gegeben wird.

2.4.1 Konzepte der Objektorientierten Programmierung

Das zentrale Abstraktions- bzw. Modularisierungselement der Objektorientierten Programmierung ist das *Objekt*. Ein Objekt kapselt Daten und Operationen, die auf diesen Daten ausgeführt werden. Die Implementierung der Daten und Operationen eines Objektes ist nach außen unsichtbar. Der direkte Zugriff auf die interne Datenstruktur wird unterbunden und erfolgt stattdessen über definierte Schnittstellen. Damit erfüllen Objekte grundlegende Eigenschaften einer Komponente (vgl. Abschnitt 2.2.3). *Klassen* sind bei der Objektorientierten Programmierung Muster, welche die Daten und Operationen eines Objektes beschreiben. Durch Instantiierung einer Klasse werden neue Objekte erzeugt. Weitere wichtige Kernkonzepte der Objektorientierten Programmierung sind *Vererbung* und *Polymorphismus*.

Vererbung ermöglicht die Schaffung neuer Objekte unter Wiederverwendung bereits existierender Objekte. Ein Objekt kann Daten und Operationen von ihrer Superklasse erben. Durch *Überschreibung* können Funktionen angepasst oder neue Daten und Operationen hinzugefügt werden. Auf diese Weise entsteht eine Vererbungshierarchie, deren Spezialisierung mit jeder Vererbung zunimmt. *Vererbung*

Als *Polymorphie*[11] wird ein Mechanismus bezeichnet, durch den ein Objekt in verschiedenen Kontexten eingesetzt werden kann, solange es die „Rolle" der jeweils geforderten Basisklasse erfüllt. Wenn z. B. eine Klasse die Funktion seiner Superklasse überschreibt, kann diese Klasse immer noch als Typ seiner Superklasse eingesetzt werden. JOHNSON et al. (JF88) stellen klar, dass auch bei der Objektorientierte Programmierung wiederverwendbare Komponenten nicht zwangsläufig entstehen, sondern auch hier der eigentlichen Programmierung ein sorgfältiger Entwurf der eigentlichen Implementierung vorausgehen muss. Sie beschreiben Frameworks als geeignete Entwurfsmethode, um Wiederverwendung zu gewährleisten. Ein Framework ist ein wiederverwendbares Softwareelement, dass die Architektur und gemeinsame Funktionalitäten der Systeme beschreibt, die aus dem Framework entwickelt werden sollen. JOHNSON et al. setzten Frameworks mithilfe abstrakter Klassen um. Die abstrakten Klassen kapseln die Hauptkomponenten der Systeme. Konkretisiert wurden diese Komponenten dann durch Subklassen. *Polymorphie*

Objektorientierte Programmierung und Concerns

Die Objektorientierte Programmierung ermöglicht eine gute Modularisierung von Concerns, so lange diese genau im System lokalisiert und separiert werden können. Es gibt jedoch eine Reihe von Concerns (vgl. Abschnitt 2.3.1), die durch die meisten Entwurfs- und Implementierungsverfahren der Objektorientierten Programmierung nicht modularisierbar sind. Es entstehen Klassen, die von mehreren Concerns „geschnitten" werden. Die Folge sind zwei Phänomene, welche die Wartung, Wiederverwendbarkeit und Anpassungsfähigkeit eines Softwaresystems erschweren:

- **Code Scattering**: Beim Code Scattering ist der Code eins Concerns über mehrere Klassen verstreut, die eigentlich ein anderes Concern implementieren.
- **Code Tangeling**: Code Tangeling bezeichnet den Umstand, dass eine Klasse mehrere Concerns implementiert.

11 Vielgestaltigkeit

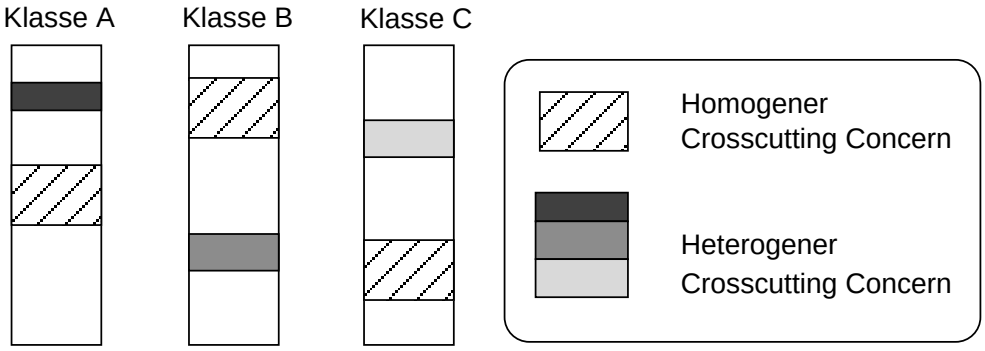


Abbildung 2.4: Lokalisierung homogener und heterogener Crosscutting Concerns im Quellcode objektorientierter Klassen.

2.4.2 Kollaborationen

Mithilfe des kollaborationsbasierten Entwurfs kann eine bestimmte Gruppe von Crosscutting Concerns in modularisierte Entwurfseinheiten zusammengefasst werden. Crosscutting Concerns können in *homogene* und *heterogene* Crosscutting Concerns eingeteilt werden (Colyer et al. (CRB04)).

- **Homogene Crosscutting Concerns** : Concerns, bei denen das gleiche oder sehr ähnliche Verhalten an vielen unterschiedlichen Stellen des Kontrollflusses des Softwaresystems auftauchen muss[12].
- **Heterogene Crosscutting Concerns** : Concerns, die viele unterschiedliche Stellen des Softwaresystems beeinflussen, deren Verhalten jedoch an den betroffenen Stellen unterschiedlich ist[13] (vgl. Abbildung 2.4).

Der kollaborationsbasierte Entwurf ermöglicht die Kapselung heterogener Crosscutting Concerns :

> *„Viewed in terms of design modularity, collaboration-based design acknowledges that a unit of functionality (module) is neither a whole object nor a part of it, but can cross-cut several different objects."* (SMARAGDAKIS et al. (SB98))

Das zentrale Absraktionselement bei dieser Entwurfsmethode sind *Kollaborationen*. Eine Kollaboration besteht aus einer Menge von Klassen und einem Protokoll, das festlegt wie diese Klassen miteinander interagieren. Der Teil einer Klasse, der dieses Protokoll ausführt, nennt man *Rolle*. Jede Klasse repräsentiert eine Menge von Rol-

12 z. B. *Logging* oder Performanzüberwachung
13 z. B. die Unterstützung eines Anwendungsfalles, der mehrere Module innerhalb des Softwaresystems umfasst.

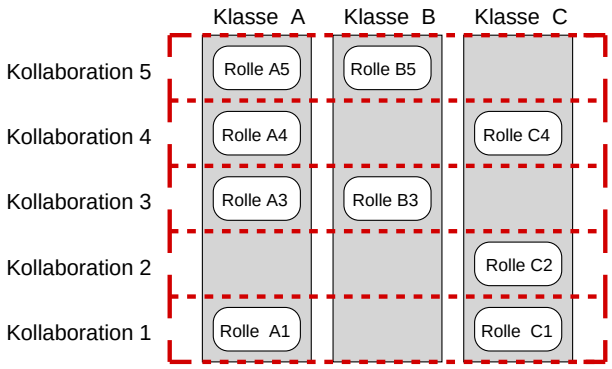


Abbildung 2.5: Schematische Darstellung eines Kollaborationsentwurfs.

len, die ein bestimmtes Verhalten der Klasse kapseln. Daher werden Rollen auch als *Klassenfragmente* bezeichnet. Eine Klasse ist also in der Regel Teil mehrerer Kollaborationen. Jede Kollaboration umfasst umgekehrt eine Menge von Rollen, die Beziehungen zwischen Klassen kapseln. Ziel des kollaborationsbasierten Entwurfs ist es, eine Anwendung durch eine Menge unabhängiger Kollaborationen zu beschreiben, die alle Abhängigkeiten zwischen den Klassen kapseln.
Abbildung 2.5 zeigt die Überlagerung von Kollaborationen und Klassen. Kollaborationen werden in Schichten übereinander gelegt, um eine lauffähige Anwendung zu erhalten[14]. Dabei schneidet jede Kollaboration eine oder mehrere Klassen.
Bei der Implementierung von Kollaborationen steht der Entwickler vor der Herausforderung die Modularität des Entwurfs bei der Implementierung zu erhalten. Frameworks werden häufig genutzt, um kollaborationsbasierte Entwürfe zu implementieren. BATORY et al. (BCS99) haben verdeutlicht, dass dieser Ansatz üblicherweise nicht die Modularität des Entwurfs erhält [15].

14 Die Begriffe Kollaborationen und Schichten werden daher auch synonym verwendet.

15 Der Grund hierfür liegt darin, dass Frameworks die Wiederverwendung abstrakter Klassen unterstützen, jedoch keine Möglichkeit vorsehen, um eine Menge konkreter Klassen zu implementieren und nach Belieben einzusetzen. Eine feste Implementierung innerhalb eines Frameworks würde die Komplexität der Frameworkinstanzen, die dieses Merkmal nicht benötigen unnötig erhöhen. Daher muss dieses Merkmal in jeder Frameworkinstanz neu implementiert werden. Dies hat Code-Scattering zur Folge. Ein beschreibendes Kostenmodell findet sich in SMARAGDAKIS (Sma99).

2.4.3 Implementierung durch Mixin-Layer

SMARAGDAKIS et al. (SB98) nutzen Mixin-Layer zur Implementierung von Kollaborationen. Der Begriff Layer spiegelt den schichtenbasierten Aufbau der Kollaborationen wider. Mixins sind besondere Klassen, aus denen die Schichten bestehen.

Mixins

Die Hauptidee von Mixins besteht darin, dass ein Mixin mit verschiedenen Superklassen instantiiert werden kann. In der Objektorientierten Programmierung können Subklassen üblicher Weise nur zusammen mit ihren Superklassen definiert werden. Mixins werden daher auch als *abstrakte Subklassen* bezeichnet (BRACHA et al. (BC90)). Hierdurch kann die implementierte Funktionalität eines Mixins zu vielen verschiedenen Klassen hinzugefügt werden. Mixins können einfach durch parametrisierte Vererbung umgesetzt werden (VANHILST et al. (VN96b)).
Die ersten, die Mixins zur Implementierung von Kollaborationen nutzten, waren VANHILST et al. (VN96b),(VN96a). Sie bildeten Mixins auf zu implementierende Rollen ab. Abbildung 2.6 zeigt die Umsetzung einer Rolle in C++-Syntax als Mixin.

```
1 template <class RoleSuper, class OA, class OC>
2   class B4 : public RoleSuper{
3     /*role implementation, using OA, OC*/
4   }
```

Abbildung 2.6: Umsetzung einer Rolle in C++-Syntax als Mixin.

Aufgrund der feingranularen Abbildung zwischen Rollen und Mixins, werden die Schichten nicht explizit repräsentiert, so dass dieser Ansatz nicht skalierbar ist. VANHILST et al. schlugen als Lösung die Abbildung einer Kollaboration auf eine Implementierungseinheit vor.
SMARAGDAKIS et al. (SB98) nutzten Mixin-Layer als Implementierungseinheit. Mixin-Layer sind Mixins, die aus mehreren Mixins bestehen. Gekapselte Mixins werden als *innere Mixins* bezeichnet, die kapselnden Mixins entsprechend als *äußere Mixins*. Innere Mixins können wie jede andere Variable geerbt werden. Ein äußerer Layer wird dann als Mixin-Layer bezeichnet, wenn der Parameter (Superklasse) des äußeren Mixins alle Parameter (Superklassen) der inneren Mixins kapselt[16]. Vorteile von Mixin-Layer sind:

- die Erhaltung der Entwurfsstruktur,

16 Innere Mixins können wiederum Mixin-Layer sein.

- die Wiederverwendbarkeit,
- die Austauschbarkeit und
- die Skalierbarkeit.

Der Erhalt der Entwurfsstruktur vereinfacht die Wartung erheblich. Wenn der Entwurf verändert wird, können notwendige Änderungen im Code schnell und isoliert vorgenommen werden. Ein Nachteil der Umsetzung durch Mixin-Layer besteht darin, dass die Umsetzung mittels Templates, sowie parametrisierter Vererbung sehr komplex und schwierig zu überschauen ist.

Ein weiterer wichtiger Gesichtspunkt ist die Sicherstellung, dass Mixin-Layer korrekt miteinander kombiniert werden. Nicht jede Kombination ist sinnvoll. Mixin-Layer müssen häufig in einer bestimmten Reihenfolge angeordnet werden oder können nicht mehr als einmal genutzt werden. Diese Problematik und eine mögliche Lösung werden im folgenden Abschnitt betrachtet.

2.5 GenVoca

GenVoca ist ein Entwurfs- und Implementierungsmodell, um Familien hierarchischer Systeme aus wiederverwendbaren Komponenten zusammenzusetzen (BATORY et al. (BO92)). Es basiert auf einer relativ alten Entwurfsmethode, die von DIJKSTRA (Dij76) vorgeschlagen wurde, der *schrittweisen Verfeinerung*.

Schrittweise Verfeinerung

Die schrittweise Verfeinerung ist ein Top-Down-Entwurfsprozess, bei dem eine Anwendung durch eine Folge von Verfeinerungsschritten entwickelt wird. Beginnend mit einer sehr abstrakten Beschreibung der Funktionalität, werden jedem Verfeinerungsschritt weitere Informationen über eine Funktionalität hinzugefügt, bis der Detaillierungsgrad hoch genug ist, um die Funktionalität zu programmieren. Bei GenVoca sind Verfeinerungen weniger feingranular. Jede Verfeinerung entspricht hier einer Komponente, die jeweils aus einer Menge von Klassen und/oder Funktionen besteht, so dass auch komplexe Anwendungen aus wenigen Verfeinerungen zusammengesetzt werden können. Die Verfeinerungen werden in *Schichten* übereinandergelegt, um eine lauffähige Anwendung zu erhalten. GenVoca und der kollaborationsbasierte Entwurf sind eng miteinander verknüpft, da eine GenVoca-Verfeinerung als Kollaboration betrachtet werden kann (BATORY et al. (BCS99)).

GenVoca-Notation

Um die Komposition von Schichten für die Erstellung einer konkreten Anwendung verständlich ausdrücken zu können, bietet GenVoca eine einfache Notation, in der die Kompositionen als Gleichung ausgedrückt werden. Programme werden als Werte und Verfeinerungen als Funktionen ausgedrückt.
Betrachtet werden die folgenden Konstanten f und g, die Programme mit unterschiedlichen Komponenten repräsentierten:

Konstante f: Programm mit Komponente f
Konstante g: Programm mit Komponente g

Eine Verfeinerung ist eine Funktion, die ein Programm als Eingabe und als Ausgabe ein um eine Komponente erweitertes Programm hat. Dabei repräsentiert eine Funktion eine Komponente und seine Implementierung, so dass verschiedene Funktionen für eine Komponente existieren können, wenn die Implementierung unterschiedlich ist:

i(x) : Fügt Komponente i zu Programm x hinzu.
j(x) : Fügt Komponente j zu Programm x hinzu.

Verschiedene Gleichungen definieren eine Programmfamilie:

$\text{Programm}_1 =:$	i(f) ;	Programm_1 besitzt Komponente i und f
$\text{Programm}_2 =:$	j(g) ;	Programm_2 besitzt Komponente j und g
$\text{Programm}_3 =:$	i(j(f)) ;	Programm_3 besitzt Komponente i, j und g

Design Rule Checking

Auch bei einer syntaktisch korrekten Schichtenkombination ist es möglich, dass semantische Regeln für die Kombination verletzt werden. Diese Regeln werden als *Design Rules* bezeichnet. Um die semantische Korrektheit zu überprüfen, muss jede Schicht domänen-spezifische Informationen über Voraussetzungen und Restriktionen

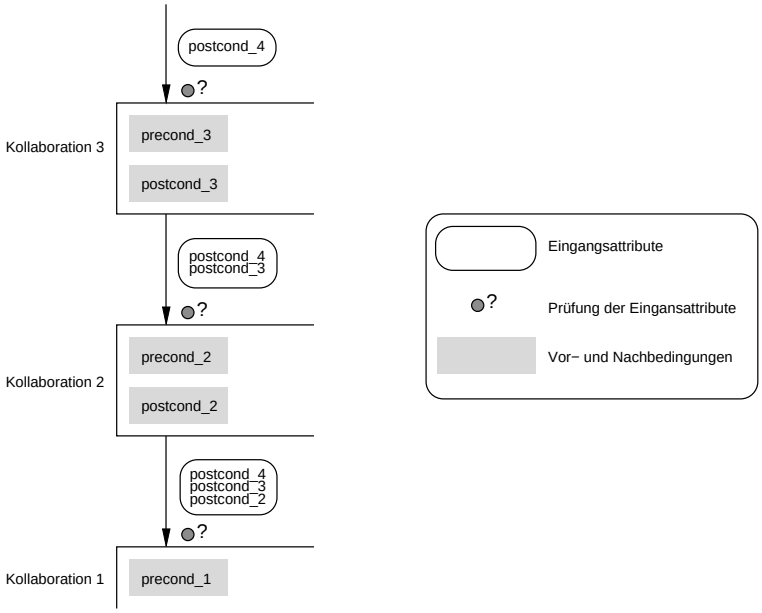


Abbildung 2.7: Darstellung der Design Rule-Überprüfung in Top-Down-Richtung.

hinsichtlich der Nutzung dieser Komponente bereitstellen. Diese Informationen werden während der Analysephase zusammengetragen. Die Regelüberprüfung dieser Informationen wird als *Design Rule Checking* bezeichnet (BATORY et al. (BG97)).

Vor- und Nachbedingungen

Bezüglich der Design Rules unterscheidet man zwischen Vorbedingungen für die Verwendung einer Komponente und ihren Nachbedingungen. Nachbedingungen einer Komponente müssen nach Ausführung einer Operation erfüllt sein und bilden die Vorbedingungen für andere Komponenten.[17] Üblicherweise sind die Eigenschaften, die in Richtung höherer Schichten propagiert werden andere, als die, die in Richtung niedrigerer Schichten propagiert werden. [18]

Vorgehen

Die eigentliche Design Rule-Überprüfung ist ein relativ einfacher Vorgang, der Top-Down und Bottom-Up durchgeführt wird. Je nach Durchführungsrichtung werden die Vor- und Nachbedingungen als *Pre-* und *Postconditions* oder als *Pre-* und *Postrestrictions*

17 Design Rules müssen nicht immer lokal, d. h. von der benachbarten Komponente erfüllt werden, sondern können auch von weiter entfernten Komponenten erfüllt werden.

18 Die Ausdrücke „höher" und „niedriger" beziehen sich auf die Position einer Komponente innerhalb der Gleichung. Die äusserste Komponente ist die höchste und die innerste Komponente die niedrigste.

bezeichnet. Eine schematische Darstellung des Top-Down-Vorgangs ist in Abbildung 2.7 zu sehen. Betrachtet wird die Kollaboration 3 der Abbildung. Diese Schicht besitzt eine Postcondition `postcond_3` und einer Precondition `precond_3`. Direkt überhalb der Kollaboration 3 hat eine Attributenmenge Gültigkeit, welche die `postcond_4` einer vorhergehenden Kollaboration enthält. Wenn die Attributmenge die Vorbedingung `precond_3` erfüllt wird Kollaboration 3 korrekt eingesetzt. Anschließend wird die Attributmenge um die Nachbedingung `postcond_3` erweitert und zur darunter liegenden Schicht propagiert.
Der Bottom-Up-Prozess wird analog in umgekehrter Richtung für die Prerestrictions und Postristrictions durchgeführt.

2.6 Merkmalorientierte Programmierung und AHEAD

Der Begriff der Merkmalorientierten Programmierung (FOP[19]) wurde von PREHOFER (Pre97) eingeführt. Die Merkmalorientierte Programmierung ist eine relativ junge Methode der Softwareentwicklung und kann als Weiterentwicklung der Objektorientierten Programmierung betrachtet werden. Sie wird vor allem im Bereich der Produktlinienentwicklung angewendet. Wie bei der Merkmalorientierten Domänenanalyse besteht die Grundidee darin, Applikationen aufgrund ihrer Merkmale zu spezifizieren.

2.6.1 AHEAD

AHEAD[20] ist ein Entwurfsmodell für die Umsetzung der Merkmalorientierten Programmierung. AHEAD wurde von BATORY et al. (BSR03) vorgestellt und stellt eine Weiterentwicklung von GenVoca dar. Die Grundidee besteht darin, dass eine Schicht sich rekursiv aus weiteren Schichten zusammensetzen kann und dadurch eine Kapselungshierarchie definiert.

Schrittweise Merkmalverfeinerung

Das AHEAD-Modell basiert ebenfalls auf dem Kollaborationsentwurf. Jede Verfeinerungsschicht kapselt ein Merkmal. Man spricht daher auch von *schrittweiser Merkmalverfeinerung*[21].

19 engl. **F**eature **O**riented **P**rogramming
20 engl. **A**lgebraic **H**ierarchical **E**quations for **A**pplication **D**esign
21 Die Begriffe *Merkmal*, *Schicht* und *Kollaboration* werden daher im Zusammenhang mit AHEAD häufig synonym genutzt.

Eine andere Erweiterung des AHEAD-Modells gegenüber GenVoca besteht darin, dass AHEAD berücksichtigt, dass Software nicht nur aus Code, sondern auch aus vielen weiteren Dateiartefakten, wie Dokumentation, UML-Diagrammen oder Testszenarien bestehen kann. Jedes Merkmal kapselt daher nicht mehr nur Klassenfragmente, sondern auch Fragmente der übrigen Implementierungsartefakte. [22]

AHEAD-Notation

Analog zu GenVoca werden Basisartefakte nach dem AHEAD-Modell durch Konstanten repräsentiert und Verfeinerung der Artefakte durch Funktionen. Ein Artefakt, dass aus einer Kette verschiedener Verfeinerungen resultiert wird also als eine Serie von Funktionen beschrieben, die auf eine Konstante angewendet werden. Es wird jedoch ein expliziter Verfeinerungsoperator • und der Begriff des *Kollektivs* eingeführt. Ein Kollektiv kapselt alle Softwareartefakte, die zu einer Schicht gehören und wird als Menge ausgedrückt:

$$\begin{aligned} &\text{Konstante } h: \quad h = a_f, b_f, c_f \ ; \\ &\text{Funktion } \ f: \quad f = a_h, b_h, d_h \ ; \end{aligned}$$

Statt der Schreibweise $h(f)$ wird die Komposition von h und f mit Hilfe des neueingeführten Verfeinerungsoperator • ausgedrückt:

$$h \bullet f$$

Die Komposition von Kollektiven erfolgt rekursiv:

$$\begin{aligned} h \bullet f &= \{a_h, b_h, d_h\} \bullet \{a_f, b_f, c_f\} \\ &= \{a_h \bullet a_f, b_h \bullet b_f, c_f, d_h\} \end{aligned}$$

Der Kompositionsoperator • ist polymorph. Für jeden Artefakttyp gibt es eine eigene Implementierung dieses Operators.

22 Wurde z. B. der Sourcecode verfeinert, muss in den meisten Fällen auch die Dokumentation geändert werden.

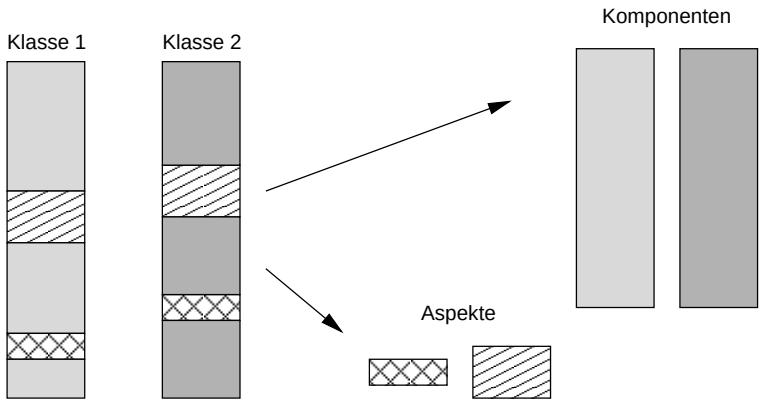


Abbildung 2.8: Zerlegung eines Systems in Komponenten und Aspekte.

2.7 Aspektorientierte Programmierung

Die Aspektorientierte Programmierung (AOP) ist eine Entwurfs- und Implementierungsmethode, welche die Modularisierung *homogener* Crosscutting Concerns ermöglicht (KLM+97). Ziel der Aspektorientierten Programmierung ist die Zerlegung eines Systems in Komponenten und Aspekte (vgl. Abbildung 2.8). Aspekte sind die Implementierungseinheiten, welche die homogenen querschneidenden Belange kapseln, während die Komponenten die Kernbelange umfassen. Aspektorientierte Programmierung nutzt Klassen der Objektorientierung, um die Komponenten zu implementieren und stellt einen Mechanismus zur Verfügung, durch den Aspekte von Komponenten getrennt implementiert werden können. Hierdurch werden die Probleme umgangen, die üblicherweise bei der Implementierung homogener Crosscutting Concerns durch objektorientierte Techniken wie Java oder C++ entstehen (vgl. Abschnitt 2.4.1). Die Aspektorientierte Softwareentwicklung kann grob in drei Teilabschnitte gegliedert werden.

1. die Dekomposition in Aspekte und Komponenten,
2. die Implementierung von Aspekten und Komponenten,
3. die Rekombination von Aspekten und Komponenten.

In Schritt eins werden alle Aspekte und Komponenten einer Anwendung identifiziert. Dann werden alle Module unabhängig voneinander implementiert. Für diese Implementierung können Standardsprachen der Objektorientierten Programmierung wie C++ oder Java verwendet werden. Anschließend müssen die implementierten Belange wieder zusammengeführt werden. Dieser Teilschritt wird auch als *weben* bezeich-

net. Für eine korrekte Rekombination muss festgelegt werden, auf welche Komponenten ein Aspekt einwirken soll. Diese Regeln werden zusammen mit dem eigentlichen Aspektcode definiert und werden *Join Points* genannt. Komponenten haben kein „Wissen" über ihre Aspekte, da ansonsten die Unabhängigkeit verletzt wäre. Join Points müssen nicht explizite Konstrukte im Sourcecode einer Komponente sein, es können auch Elemente der Programmsemantik als Join Points genutzt werden. Die Integration von Komponenten und Aspekten wird unter Verwendung von *Aspektwebern* durchgeführt.
Mittlerweile gibt es eine Reihe von Erweiterungen der bekannten objektorientierten Sprachen um spezifische Konstrukte der Aspektorientierten Programmierung: AspectJ für Java (LOPES et al. (LK98)), AspectC++ für C++ (SPINCZYK et al. (SGSP02)) und AspectS für Smalltalk (HIRSCHFELD (Hir02)).

KAPITEL 3

Grundlagen der Visualisierung

In diesem Kapitel werden Grundlagen der Informationsvisualisierung erläutert. Hierfür werden zunächst wichtige Visualisierungstechniken vorgestellt, wobei ein besonderer Schwerpunkt auf der Visualisierung hierarchischer Daten liegt. Der abschließende Teil beschäftigt sich kurz mit der Softwarevisualisierung, einem Teilbereich der Informationsvisualisierung.

Mithilfe der elektronischen Datenverarbeitung können heute in kürzester Zeit sehr große Datenmengen ermittelt und verarbeitet werden. Die Herausforderung besteht darin, diese Daten so aufzubereiten, dass sie durch den Menschen möglichst schnell und korrekt ausgewertet werden können. Numerische Methoden wie *Data Mining* oder *Knowledge Discovery*, werten die Daten für den Anwender u.a. mit statistischen Methoden aus. Das Ziel der Informationsvisualisierung besteht darin, abstrakte Informationen aus beliebigen Informationssystemen, strukturell aufzubereiten und unter Berücksichtigung der menschlichen Wahrnehmung grafisch darzustellen, so dass der Betrachter seine eigenen Schlüsse ziehen kann. Hierfür verknüpft die Informationsvisualisierung eine große Anzahl verschiedener wissenschaftlicher Disziplinen wie Computergaphik, Bild-und Signalverarbeitung, Computer Vision, Mensch-Computer-Interaktion, Wahrnehmungspsychologie und Design miteinander.

Eine besondere Rolle spielt hierbei auch die Interaktion mit den dargestellten Daten:

> *„The field of computer-based information visualization [..] is about creating tools that exploit the human visual system to help people explore or explain data. Interacting with a carefully designed visual representation of data can help us form mental models that let us perform specific tasks more effectively."*
> (MUNZNER (Mun02))

WARE (War00) beschreibt fünf Aufgaben der Datenanalyse, bei der die Informationsvisualisierung den Menschen unterstützen kann:

- das Verständnis sehr großer Datenmengen,
- die Wahrnehmung wichtiger Eigenschaften und Muster,
- der Untersuchung der Daten auf Fehler bei der Datensammlung,
- das Verständnis von globalen und lokalen Eigenschaften und
- die Interpretation der Daten und der Hypothesenbildung.

3.1 Visualisierungstechniken

Bei der Informationsvisualisierung werden in der Regel abstrakte Daten abgebildet, für die es keine Entsprechung in der physischen Welt gibt. Eine der Herausforderungen der Informationsvisualisierung besteht darin für diese Daten eine angemessene visuelle Metapher zu finden. Allgemeingültige Regeln für die Effektivität einer Visualisierung können nur schwer formuliert werden, da sie von einer Reihe verschiedener Kriterien beeinflusst werden:

- die Wahrnehmungfähigkeit des Betrachters,
- die Visualisierungsziele (z. B. Analyse vs. Ergebnisdarstellung) und
- die Eigenschaften der darzustellenden Daten (Dimensionalität, Wertebereich).

Generelle Aussagen über die Wirkung einer *grafischen Variablen* sind jedoch möglich. Der Begriff der grafischen Variable wurde von BERTIN (Ber83) geprägt. BERTIN hat mit der *grafischen Semiologie* eine grundlegende Taxonomie für die Darstellung zweidimensionaler Objekte entwickelt. Für die drei Grafikprimitive *Punkt, Linie* und *Fläche* unterscheidet er die sechs Variablen *Größe, Helligkeit, Muster, Farbe, Richtung* und *Form*, durch deren Anpassung die Primitive je nach zu vermittelnder Information modifiziert werden können: Weitere Informationen werden durch räumliche (1D, 2D, 3D) und zeitliche (Animation) Variablen kodiert. Eine empririsch ermittelte Rangfolge bezüglich der Effektivität dieser und weiterer grafischer Merkmale in Verbindung mit drei verschiedenen Datentypen stellt MACKINLAY (Mac88) vor. Diese Rangfolge ist in Abbildung 3.1 dargestellt.

Die aufgelisteten Eigenschaften gehören zu einer Gruppe grafischer Merkmale, die durch den Betrachter präattentiv, also ohne gerichtete Aufmerksamkeit, erkannt werden. Präattentiv wahrgenommene Eigenschaften werden sehr schnell und präzise erkannt, so dass sie besonders geeignet sind, um den Blick des Betrachters auf Bereiche

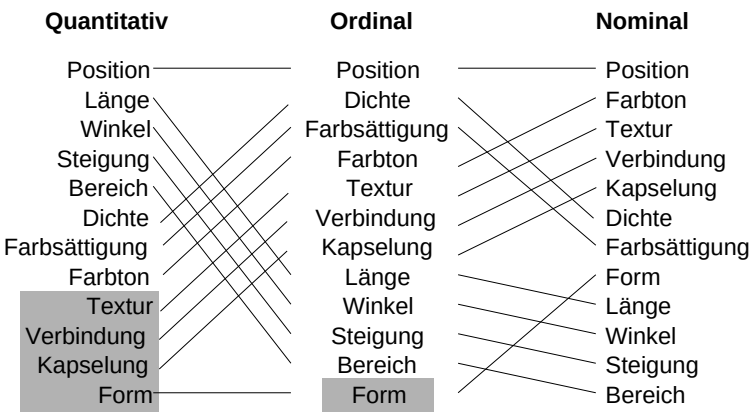


Abbildung 3.1: Effektivitätsrangfolge grafischer Merkmale. Die grau unterlegten Merkmale sind für den jeweiligen Datentyp nicht von Bedeutung, adaptiert von MACKINLAY (Mac88).

zu lenken, die von besonderem Interesse sind (vgl. Abbildung 3.3). An diesem Beispiel für das Zusammespiel zwischen Visualisierung und menschlicher Wahrnehmung wird deutlich, dass umfassende Kenntnisse der menschlichen Wahrnehmung für die Erzeugung einer effektiven Visualisierung unumgänglich sind[1].

3.1.1 Information Seeking Mantra

Das *Visual Information Seeking Mantra* von SHNEIDERMAN (Shn96) umfasst eine Reihe von Richtlinien für die Erstellung einer interaktiven Visualisierung:

„Overview first, zoom and filter, then detail-on-demand."

Das Zitat benennt die grundlegenden Schritte einer interaktiven Datenexploration. Aus einer Übersicht gewinnt der Betrachter einen ersten Eindruck von der Anzahl der Daten und den groben Zusammenhängen zwischen ihnen („*Overview first*"). Globale Muster innerhalb der Datenmenge werden häufig erst aus diesem Blickwinkel sichtbar. Aus der Übersicht muss der Anwender die Bereiche auswählen können, die für ihn von besonderem Interessen sind, und über die er weitere Informationen erhalten möchte. Filtermechanismen sind notwendig um irrelevante Details auszublenden („*zoom and filter*"). Bei großen Datenmenge ist die Darstellung ergänzender Informationen zu Datenpunkten innerhalb einer Visualisierung schwierig, ohne die Ansicht zu „überfrachten". Daher werden solche Details nur auf Anfrage des Anwenders darge-

1 Ein Überblick über weitere Zusammenhänge zwischen Visualisierung und visueller Wahrnehmung ist bei WARE (War00) zu finden.

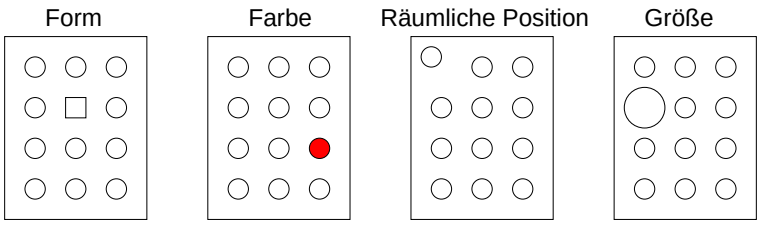


Abbildung 3.2: Auswahl präattentiv wahrgenommener Merkmale.

stellt („*then detail-on-demand*"). Eine typische Umsetzung hierfür sind Tooltips oder Popup-Fenster.

Focus & Context vs. Detail & Overview

Zwei etablierte Techniken der Informationsvisualisierung ermöglichen eine Kombination von Übersicht und Detaildarstellung. Bei einer *Focus & Context*-Darstellung liegen die wichtigen Daten im Fokus der Darstellung und werden detailliert dargestellt. Unwichtigere Gebiete liegen am Rand des Darstellungsbereiches und werden nur komprimiert angezeigt, ermöglichen dem Betrachter jedoch die Einordnung der im Fokus liegenden Informationen in den Kontext. Diese Technik wurde z. B. bei der *Perspective Wall* angewendet (MACKINLAY et al. (MRC91), vgl. Abbildung 3.3), die der Darstellung linearer Daten dient. Bei *Detail-and-Overview*-Techniken werden Übersicht und Detaildarstellung nicht in eine Ansicht integriert sondern in zwei verschiedenen Fenstern dargestellt. Beide Ansichten werden parallel oder sequentiell dargestellt. Detail-and-Overview-Techniken werden häufig mit zwei Methoden verknüpft, die *Navigational Slaving* und *Linking* genannte werden. Beim Navigational Slaving werden Bewegungen eines Fensters in das andere Fenster propagiert. Das Linking verknüpft Daten beider Fenster miteinander, so dass Aktionen, die auf dem Element des einen Fensters durchgeführt werden auch auf dem korrelierenden Element der anderen Ansicht ausgeführt werden.

3.2 Visualisierung hierarchischer Daten

Die traditionelle Darstellung hierarchischer Daten erfolgt als Baumdiagramm, das die hierarchische Struktur der Elemente mittels Verbindungen zwischen Knoten und Kanten darstellt. Der Nachteil dieser Visualisierung besteht in einer sehr schlechten Ausnutzung der Darstellungsfläche, so dass nur eine kleine (Teil-)Menge von Daten, trotz vieler Freiflächen, auf einen Blick zu erkennen ist. Weiterhin sind Informationen zu den Elementen nur schwierig darstellbar, ohne dass die Ansicht unübersichtlich wird.

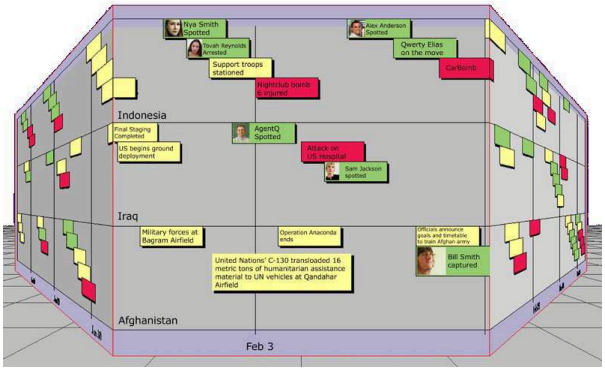


Abbildung 3.3: Originalansicht der *Perspective Wall*. Quelle: RAO (Rao96)

Zur Behebung oder Milderung dieses Problems wurden eine Reihe unterschiedlicher Darstellungsformen für hierarchische Daten entwickelt. Hierbei finden sich vor allem drei unterschiedliche Layout-Techniken (NEUMANN (Neu04)):

- **Einfaches Baumlayout**: Ästhetische Kriterien wie minimiale Kantenüberschneidungen oder die Anordnung von Knoten der gleichen Hierarchietiefe auf einer Ebene stehen im Vordergrund. Eine gute Ausnutzung der Darstellungsfläche ist nicht das Hauptziel. Das Scrollen ist eine notwenige Funktionalität, um alle Bereiche des Baumes betrachte zu können.
- **Einfaches, komprimiertes Baumlayout**: Das einfache Baumlayout wird kompakter gestaltet. Visualisierungssysteme mit komprimierten Baumlayouts müssen immer Interaktionsmechanismen zur Verfügung stellen, mit deren Hilfe eine nicht-komprimierte Sicht auf Bereiche, die von Interesse sind, ermöglicht werden.
- **Geschachteltes Baumlayout**: Im Vordergrund steht vor allem eine effiziente Raumausnutzung. Die Hierarchie wird durch die Schachtelung der Hierarchieelemente impliziert. Bei geschachtelten Baumlayouts werden Funktionalitäten benötigt, mit denen wichtige Bereiche herangezoomt oder herausgefiltert werden können.

Ergänzend zu den Layouttechniken werden häufig Focus & Context-Elemente integriert

Die in den folgenden Abschnitten kurz vorgestellten Darstellungsvarianten setzen die genannten Layoutvarianten einzeln oder in kombinierter Form um. Sie wurden aus-

gewählt, da sie zu den bekanntesten Hierarchievisualisierungen gehören oder die genannten Layouttechniken besonders gut demonstrieren[2].

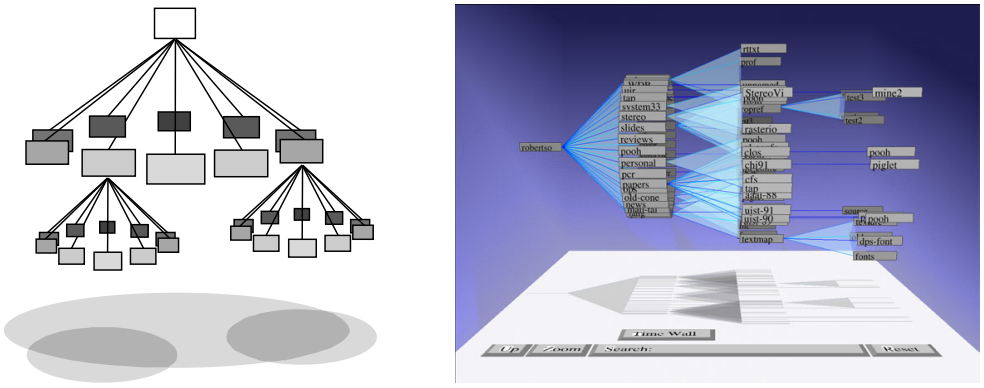
3.2.1 Cone und Cam Trees

Cone Trees und Cam Trees wurden 1991 von ROBERTSON et al. (RMC91) vorgestellt und sind eine der ersten dreidimensionalen Hierarchievisualisierungen. Dreidimensionale Darstellungen versuchen durch Erweiterung des Darstellungsraumes um eine Dimension mehr Elemente der Hierarchie darzustellen. Die Knoten des Cone Trees werden entlang eines dreidimensionalen Kegels angeordnet. An der Spitze des Kegels befindet sich die Wurzel des (Teil-) Baumes, während die Kinderknoten am unteren Rand des Kegels liegen. Die Höhe des Kegels ist auf jeder Ebene des Baumes gleich, während der Durchmesser zu den Blättern hin abnimmt, damit alle Knoten auf der Darstellungsfläche angeordnet werden können. Zusätzlich ist eine Focus & Context-Technik integriert, da auf der Vorderseite liegende Knoten größer dargestellt werden als auf der Rückseite liegende Knoten. Der Unterschied zwischen Cone Trees und Cam Trees besteht in der Ausrichtung der Kegel (vgl. Abbildung 4.6). Cam Trees haben den Vorteil, dass die Knoten mehr Informationen über die Knoten darstellen können als Cone Trees, da die Knoten größer dargestellt werden können. Ein grundsätzliches Problem dreidimensionaler Darstellungen ist die Verdeckung von Knoten und Kanten. Dieses Problem soll durch Transparenz und die Darstellung zusätzlicher Schatten, die eine zweidimensionale Repräsentation des Baumes darstellen, gelöst werden. Zusätzlich ist ein Interaktionsmechanismus nötig, der es dem Betrachter gestattet den Baum so zu rotieren, dass ein Bereich, der von Interesse ist verdeckungsfrei dargestellt wird. Im günstigsten Fall wird diese Rotation so animiert, dass der Anwender die Rotation mitverfolgen kann und nicht die Orientierung verliert.

3.2.2 Hyperbolic Tree Browser

Der Hyperbolic Browser wurde am *Xerox Palo Alto Research Center* durch LAMPING et al. (LRP95) entwickelt. Der Hyperbolic Browser kann aufgrund einer Focus & Context-Technik sehr große Hierarchien darstellen. Grundlage ist die hyperbolische Geometrie. Sie ist eine nicht-euklidische Geometrie, in der parallele Linien voneinander weg

2 Ein sehr guter Überblick über weitere hierarchische Visualisierung, aber auch über die Informationsvisualisierung im Allgemeinen findet sich bei NEUMANN (Neu04).

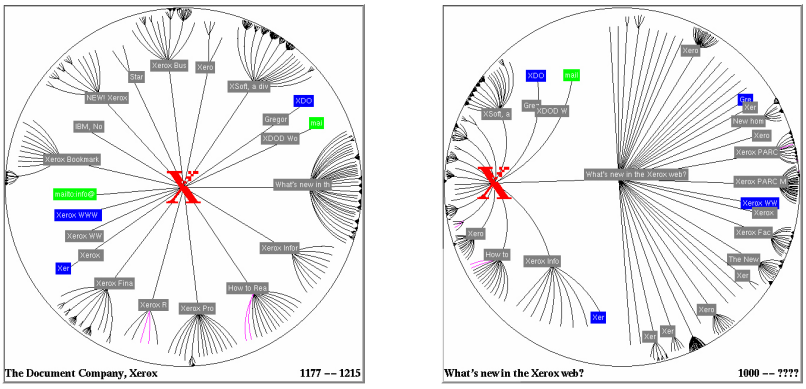


(a) Schematische Darstellung eines Cone Trees.

(b) Original Cam Tree-Ansicht des *Information Visualizer*.

Abbildung 3.4: Cone Tree und Cam Tree. Quelle: RAO (Rao96).

laufen. Daher wächst der Umfang eines Kreises auf der hyperbolischen Ebene exponentiell mit seinem Radius, so dass der verfüügbare Raum ebenfalls exponentiell mit zunehmender Entfernung wächst. Im Fokus der Darstellung liegt die Wurzel des Baumes. Die Kinderknoten werden sternenförmig auf eine zweidimensionale kreisförmige Fläche projiziert. Der größte Teil der Daten liegt, sehr klein dargestellt, am Rand der Fläche und bildet den Kontext. Nur im Zentrum der Ansicht liegende Knoten werden so groß dargestellt, dass auch Details zu erkennen sind (vgl. Abbildung 3.5). Durch die Selektion interessanter Bereiche wird der Fokus der Visualisierung entsprechend angepasst. Auch dieser Vorgang wird animiert dargestellt.



(a) Zentrierte Wurzel

(b) Wurzel am linken Rand

Abbildung 3.5: Zwei Ansichten des *Hyperbolic Tree Browser*. Quelle: RAO (Rao96).

Abbildung 3.6: Schematische Darstellung einer *Treemap*. Quelle: NEUMANN (Neu04).

3.2.3 Treemap

Treemaps wurden 1991 durch Shneiderman et al. (JS91) vorgestellt. Sie sind eine Visualisierungsform hierarchischer Daten, die versucht den Darstellungsbereich besonders effizient auszunutzen, indem sie die Hierarchie als eine Folge geschachtelter Rechtecke darstellt. Die Größe der Rechtecke kodiert ein Attribut des Knotens. Bei Darstellung einer Verzeichnishierarchie kann dies z. B. die Dateigröße sein. Der Layout-Algorithmus unterteilt den Darstellungsbereich rekursiv in vertikale und horizontale Rechtecke. Der Wurzelknoten der Hierarchie wird auf das Basisrechteck abgebildet, das den Darstellungsbereich ausfüllt. Dieses Basisrechteck wird dann in n weitere Rechtecke vertikal geteilt, wenn n die Anzahl der Kinderknoten der Wurzel ist. Diese Rechtecke werden horizontal entsprechend der dann folgenden Kinderknoten geteilt, usw. Auf ungeraden Ebenen erfolgt die Aufteilung immer in vertikaler und auf geraden Ebenen der Hierarchie immer in horizontaler Richtung. Neben der effizienten Raumausnutzung haben Treemaps den großen Vorteil, dass aufgrund der rechteckigen und – im Vergleich zu nicht-geschachtelten Darstellungen – großen Knotendarstellung zusätzliche Informationen durch Farbe oder Text zur Verfügung gestellt werden können. Ein Nachteil besteht darin, dass bei einer großen Anzahl von Knoten die Darstellungsfläche so stark unterteilt ist, dass einzelne Knoten nur noch schwer zu erkennen und auch zu selektieren sind. Grundsätzlich ist die globale Hierarchiestruktur bei Treemaps schwieriger zu erkennen als bei der traditionellen Baumansicht (Kobsa (Kob04)).

3.3 Softwarevisualisierung

Softwaresysteme werden immer komplexer, so dass die Entwicklung und das Verständnis dieser Anwendungen immer schwieriger wird. Softwarevisualisierung, als Teilbereich der Informationsvisualisierung, kann bei Tätigkeiten wie Analyse, Entwurf, Testen, Debugging und Wartung eines Softwareprodukts unterstützen. Insgesamt ist das Feld der Softwarevisualisierung noch relativ unstrukturiert. So wurde erst 2003 mit der *VISSOFT* [3] die erste eigenständige Konferenzen für den Bereich der Softwarevisualisierung ins Leben gerufen.
Eine häufig zitierte Defintion von Softwarvisualisierung stammt von PRICE et al. (PBS93):

> *„Softwarevisualization is [..] the use of the crafts of typography, graphic design, animation, and cinematography with modern-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software."*

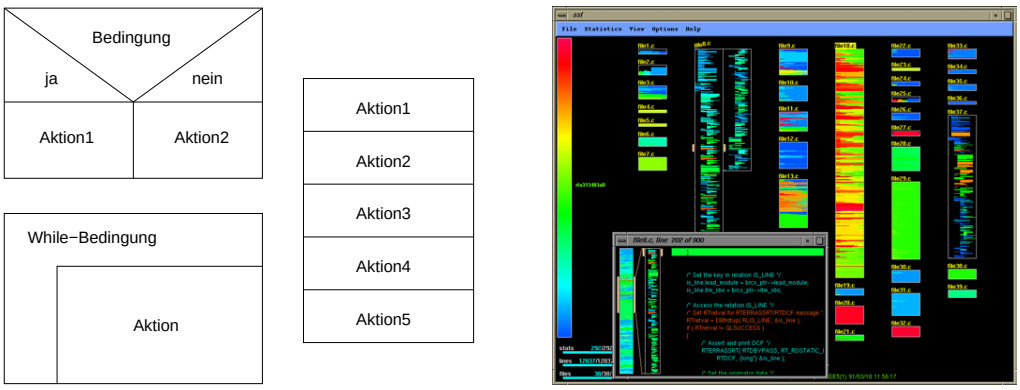
Bereits 1947 demonstrierten GOLDSTEIN et al. Flussdiagramme. Diese von Hand erstellten Diagramm waren die erste abstrakte Darstellung von Zusammenhängen in Softwareprogrammen und sollten bei der Planung und dem Entwurf von Software unterstützen. Anfang der sechziger Jahren wurden die ersten Flussdiagramme automatisch aus Software erzeugt. Dies wurde notwendig, da die Softwareprogramme mit den Entwurfsdokumenten immer seltener übereinstimmten. Des Weiteren stieg die Komplexität der Software. Durch die Einführung der strukturierten Programmierung mussten neue Darstellungsformen für eine verständliche Darstellung der Software gefunden werden. Sehr bekannt wurden in diesem Zusammenhang Nassi-Shneiderman-Diagramme (auch *Struktogramme* genannt) (NASSI et al. (NS73), vgl. Abbildung 3.7(a)). 1966 stellte KNOWLTON erste Animationstechniken in Form von Filmen vor. Sie sollten die bisherige statische Struktur der Darstellung um dynamische Aspekte erweitern. Ein weiterer Meilenstein der Softwarevisualisierung war in den 70iger Jahren die Einführung der sogenannten *Pretty-Printer*-Funktionalität. Sie ermöglichte durch Einrückungen, Leerzeichen und Leerzeilen eine bessere Visualisierung des Codes.
Durch die Darstellung von Bitmapgrafiken und die Einführungen von grafischen Benutzeroberflächen in den 80iger Jahren entstand ein eigenes Forschungsfeld für die Softwarevisualisierung.

3 engl. **I**nternational **W**orkshop on **Vis**ualizing **Sof**tware for Understanding and Analysis

PRICE et al. (PBS93) unterscheiden zwei Bereiche der Softwarevisualisierung:

- **Programmvisualisierung**: Darstellung implementierungsnaher Eigenschaften. Visualisierung von Quelltext und den darin verwendeten Datenstrukturen in statischer oder dynamischer Form verstanden.
- **Algorithmenvisualisierung**: Wird häufig zu Lernzwecken eingesetzt, z. B. Darstellung eines Sortieralgorithmus. Es gibt sowohl statische als auch dynamische (animierte) Visualisierungen.

In der Regel sind Softwarevisualisierungen in ein Werkzeug eingebettet. In dem folgenden Abschnitt wird nur das *SeeSoft*-Visualisierungssystem kurz erläutert, da es die Basis für eine Visualisierung in dieser Arbeit darstellt. Ein Überblick über weitere Werkzeuge für die Softwarevisualisierung findet sich bei ZEHLER (Zeh04).



(a) *Nassi-Shneiderman*-Diagramme

(b) Originalansicht des *SeesSoft*-Werkzeugs. Quelle: RAO (Rao96)

Abbildung 3.7: Statische Softwarevisualisierungen.

3.3.1 SeeSoft

SeeSoft wurde 1992 durch EICK et al. (ESS92) vorgestellt. SeeSoft ist ein Werkzeug, mit dem Strukturen von mehr als 50.000 Zeilen Quellcode übersichtlich dargestellt werden können. Abbildung 3.7(b) zeigt ein Beispiel. Die Quellcodedateien werden durch Spalten repräsentiert, wobei die Länge der Spalten die Größe der Datei widerspiegelt. Jede Programmzeile einer Datei wird als Pixellinie innerhalb der Spalte dargestellt. Mit Hilfe verschiedener Linienfarben werden unterschiedliche statistische Informationen über den Quelltext angezeigt, etwa den Zeitpunkt der letzten Änderung

einer Zeile. Um den Quellcode darzustellen kann eine kleine Linse über die Spalten geführt werden. Der entsprechende Quellcode wird dann in einem separaten Fenster angezeigt. Das SeeSoft-Werkzeug kann für eine Vielzahl verschiedener Anwendungsbereiche in der Softwareentwicklung genutzt werden, u. a. für das Projekt-Management, Qualitätssicherung und Systemtest, für die Optimierung des Codes, Verständnis des Codes oder das Training neuer Entwickler.

KAPITEL 4

Analyse und Entwurf

Die Merkmalorientierte Programmierung ist ein Verfahren, um eine Programmfamilie zu implementieren. Für eine systematische Entwicklung der Programmfamilie wird die Merkmalorientierte Programmierung in einen Domain Engineering-Prozess eingebettet. Ziel dieser Arbeit ist die Entwicklung eines Werkzeugs, das beide Methoden integriert und unterstützt. Soweit zum aktuelle Zeitpunkt bekannt ist, existiert noch kein Werkzeug, welches dies umsetzt. Das Werkzeug *pure::variance* der *pure-systems GmbH* verknüpft ebenfalls alle Phasen des Domain Engineerings miteinander. Diese Anwendung verfolgt jedoch einen allgmeinen Ansatz, der keine spezielle Programmiermethodik unterstützt (BEUCHE et al. (BPSP03)). Daher kann auch kein gezielter Austausch von Informationen erfolgen.
Für eine gezielte Unterstützung bei dem Austausch von Wissen zwischen den Entwicklungsphasen ist jedoch die Festlegung auf eine Programmiermethodik notwendig. Im folgenden Abschnitt wird kurz zusammengefasst welche Techniken neben dem Domain Engineering und der Merkmalorientierten Programmierung genutzt werden sollen. Anschließend werden die Probleme aufgezeigt, die durch fehlende Werkzeugunterstützung bei der Durchführung des Domain Engineerings und der Merkmalorientierten Programmierung entstehen können. Auf der Grundlage dieser Ergebnisse werden mögliche Lösung für die Umsetzung eines solchen Werkzeugs erörtert.

4.1 Ausgangslage

Das Domain Engineering besteht aus drei Phasen, für die jeweils ein konkretes Verfahren gewählt werden muss. Zentraler Gegenstand der Untersuchungen dieser Di-

plomarbeit ist die Werkzeugunterstützung der Merkmalorienierten Programmierung. Analog zur Objektorientierte Programmierung gibt es jedoch unterschiedliche Umsetzungen und Methoden für die Durchführung. Das am weitesten entwickelte Werkzeug für die Merkmalorientierte Programmierung ist die AHEAD-Toolsuite, die auf dem vorgestellten AHEAD-Modell basiert (vgl. Abschnitt 2.6.1). Da die AHEAD-Toolsuite frei verfügbar ist wird sie zur Umsetzung der Merkmalorientierten Programmierung genutzt werden.

4.1.1 Entwicklungsprozess einer Programmfamilie

Die Art des Entwurfs und der Implementierung werden durch Verwendung der AHEAD-Toolsuite festgelegt. Das AHEAD-Modell basiert auf der schrittweisen Merkmalverfeinerung. Merkmale werden als Kollaborationen repräsentiert. Für die Implementierung des Entwurfs beinhaltet die AHEAD-Toolsuite eine Reihe unterschiedlicher Funktionalitäten (vgl. Abschnitt 4.1.2). Wichtigstes Instrument ist die Java-Erweiterung *Jak*, mit deren Hilfe Merkmale/Kollaborationen[1] als Mixin-Layer umgesetzt werden können.
Als Verfahren für die Domänenanalyse wurde die Merkmalorientierte Domänenanalyse . Da die zentralen Abstraktionseinheiten der Merkmalorientierte Domänenanalyse ebenfalls Merkmale sind, lässt sich diese Methode ausgezeichnet mit der Merkmalorientierten Programmierung verknüpfen.

Merkmalorientierte Softwareentwicklung

Aufgrund der gewählten Methoden wird eine Programmfamilie nun in jeder Phase des Domain Engineerings auf der Basis ihrer Merkmale betrachtet. Daher kann der Entwicklungsprozess nun auch als *Merkmalorientierte Softwarentwicklung* bezeichnet werden.

4.1.2 Funktionalität der AHEAD-Toolsuite

In diesem Teilabschnitt wird kurz der Funktionsumfang der AHEAD-Toolsuite erläutert, die bei der Umsetzung des AHEAD-Modells Hilfestellung gibt. Die AHEAD-Toolsuite besteht aus einer Sammlung verschiedener Werkzeuge. Die folgende Liste enthält die wichtigsten Tools:

- **Composer:** Werkzeug zur Kombination von Merkmalen,

1 Die Begriffe Merkmal und Kollaboration bzw. Schicht können nicht immer klar getrennt werden, da sie sehr eng miteinander verknüpft sind und sich z. T. gegenseitig bedingen.

- **Jampack und Mixin:** Werkzeuge zur Kombination von Jak-Dateien,
- **Unmixin:** zerlegt Jak-Files, die aus der Komposition mehrerer Jak-Dateien mittels *Mixin* enstanden sind, wieder in ihre einzelnen Merkmale,
- **Jak2java:** konvertiert Jak-Dateien in Java-Dateien,
- **DRC:** kombiniert Design Rules,
- **ModelExplorer:** Werkzeug, mit dem merkmalbasierte Entwürfe kombiniert, durchsucht und erstellt werden können.

Die wichtigste Komponente der Toolsuite ist die Java-Erweiterung *Jak*. Sie erweitert Java u.a. um das Schlüsselwort *refines*, durch das Verfeinerungen auf Codeebene gekennzeichnet werden. Abbildungen 4.1 und 4.2 stellen die Verfeinerung der Klasse `Calc` dar. Ziel ist die Erstellung eines Taschenrechners (BATORY (Bat04)).

```
1 refines class Calc {
2   void divide() {
3     e0 = e0.divide( e1 );
4     e1 = e2;
5   }
6 }
```

Abbildung 4.1: Verfeinerung um Division.

```
1 refines class Calc {
2   void add() {
3     e0 = e0.add(e1);
4     e1 = e2;
5   }
6 }
```

Abbildung 4.2: Verfeinerung um Addition.

Komposition von Merkmalen

Die Kapselungshierarchie der Merkmale und ihrer Implementierungsartefakte wird als Verzeichnishierarchie repräsentiert, so dass die Verfeinerung von Merkmalen – oder die Kombination von Jak-Dateien – als Kombination von Verzeichnissen beschrieben werden kann. Abbildung 4.3 zeigt wie Merkmale durch rekursive Zusammenfassung von Verzeichnissen kombiniert werden. Die Reihenfolge, in der Verzeichnisse miteinander kombiniert werden sollen, wird in *Equation-Files* festgehalten. Der eigentliche Kombinationsprozess erfolgt unter Verwendung zwei verschiedener Werkzeuge. Abbildung 4.4 zeigt den Kombinationsprozess für Jak-Dateien. Zunächst werden die Jak-Klassenfragmente der verschiedenen Schichten mittels Jampack oder Mixin kombiniert. Mithilfe des *Jak2java*-Werkzeugs wird diese kombinierte Jak-Datei in eine Java-Datei überführt. Aus dieser Java-Datei kann dann unter Verwendung der bekannten Java-Werkzeuge eine ausführbare Datei erzeugt werden.

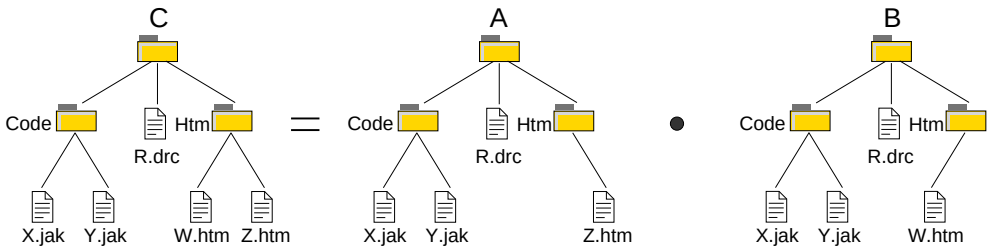


Abbildung 4.3: Kombination von Merkmalen durch Kombination von Verzeichnissen, welche die Implementierungsartefakte der Merkmale enthalten.

Jampack vs. Mixin

Jampack und *Mixin* sind zwei Implementierungen des Kompositionsoperators für Jak-Dateien. Der Unterschied zwischen *Jampack* und *Mixin* besteht in der Durchführung der Code-Kombination. Der Jampack-Algorithmus fügt Verfeinerungen, durch direkte Veränderung der Basisartefakte zusammen. Die Grenzen zwischen den einzelnen Verfeinerungen werden dabei nicht erhalten. Wird die kombinierte Jak-Datei z. B. zur Fehlerbehebung verändert, muss diese Änderung manuell rückpropagiert werden. Dieser Vorgang ist fehleranfällig und zeitintensiv. Mixin behebt diesen Nachteil, indem es eine Vererbungshierarchie entsprechend der Verfeinerungskette erstellt. Abgesehen von der terminalen Klasse sind alle Klassen entlang der Vererbungshierarchie abstrakt. Mithilfe des Tools *Unmixin*, kann eine mit Mixin kombinierte Jak-Datei daher wieder in die elementaren Jak-Files zerlegt werden.

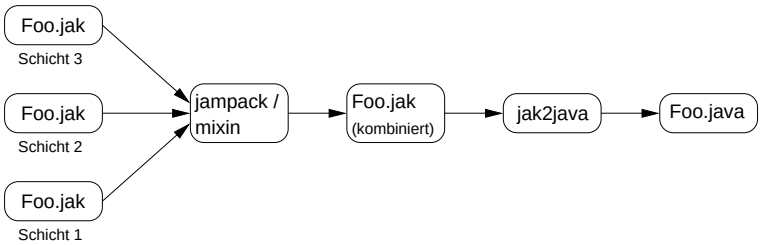


Abbildung 4.4: Erzeugung einer Java-Datei aus Jak-Klassenfragmenten.

ModelExplorer

Die AHEAD-Toolsuite enthält eine Vielzahl unterschiedlicher Anwendungen. Diese stellen jedoch hauptsächlich die Grundfunktionalitäten zur Verfügung, welche für die Merkmalorientierte Programmierung minimal notwendig sind. Darüber hinausgehende Werkzeugunterstützung mit einer grafischen Nutzeroberfläche bietet lediglich der integrierte *ModelExplorer* (vgl. Abbildung 4.8). Neben den grundlegenden Editor-

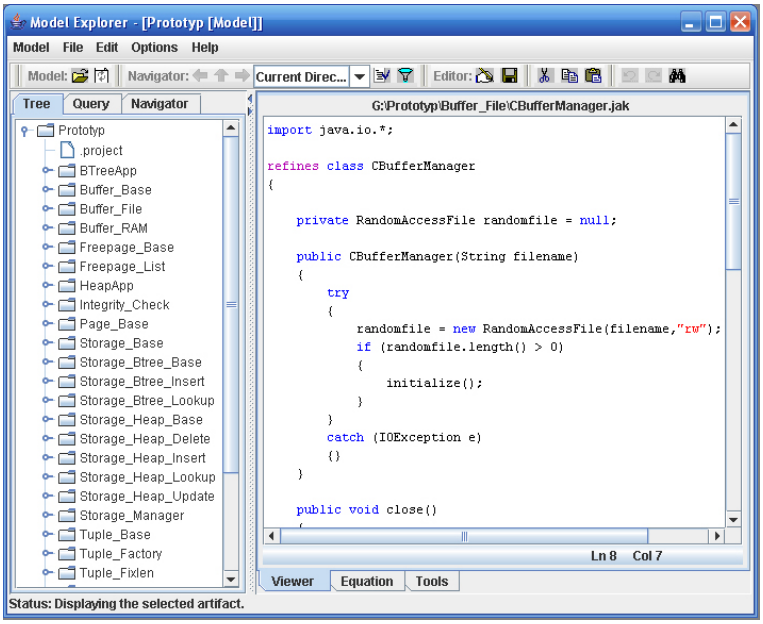
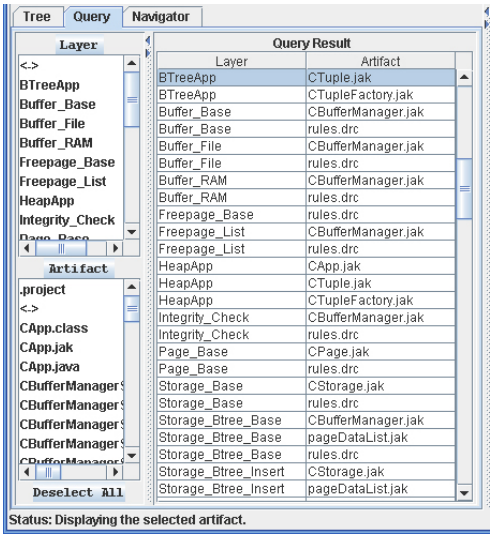
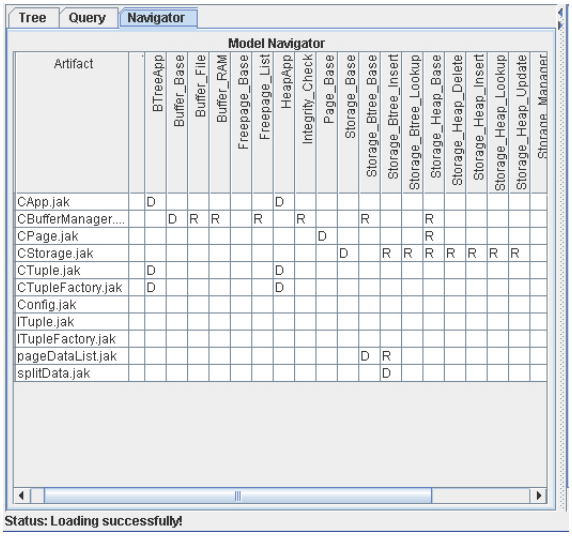


Abbildung 4.5: Originalansicht des *ModelExplorers*.

funktionen mit Dateibrowser und Texteditor, stellt der ModelExplorer noch zwei weitere Ansichten auf Merkmale und ihre Implementierungsartefakte zur Verfügung. Eine dieser Ansichten stellt Merkmale und Artefakte als Relation dar (vgl. Abbildung 4.6(a)). Die dritte Ansicht bildet Merkmale und Artefakte als Matrix ab (vgl. Abbildung 4.6(b)). Alle drei Ansichten können zur Dateinavigation genutzt werden. Die Darstellung als Verzeichnisbaum spiegelt die Kapselungshierarchie der Merkmale wider. Die Repräsentation als Relation stellt Merkmale und ihre Implementierungsartefakte gegenüber. Diese Darstellungsweise bietet jedoch keine nennenswerten Vorteile gegenüber der des Verzeichnisbaums und ist zu unstrukturiert, um einen Überblick über die Daten zu gewinnen. Interessanter ist die Matrixansicht, da sie einen Überblick darüber schafft, welches Merkmal welche Implementierunsartefakte verfeinert oder neu definiert. Neudefinitionen werden in der Matrix mit **D** für *definition* gekennzeichnet, Verfeinerungen werden mit **R** für *refinement* verdeutlicht.
Weitere Funktionalitäten des ModelExplorers sind u.a. die Erzeugung von Equation-Dateien durch Auswahl der entsprechenden Verzeichnisse und die Möglichkeit verschiedene Werkzeuge der AHEAD-Toolsuite aufzurufen.



(a) Relationenansicht des *ModelExplorers*.



(b) Matrix-Ansicht des *ModelExplorers*.

Abbildung 4.6: Repräsentation der Merkmale und Implementierungseinheiten im *ModelExplorer*.

Design Rules

Die Design Rules, die bei Nutzung der AHEAD-Toolsuite spezifiziert werden können basieren auf einer attributiven Grammatik, welche die Attribute *Int* und *Bool* nutzt. Abbildung 5.8 zeigt ein Beispiel. In Zeile eins wird der Name der Schicht `foo` deklariert, für die diese Design Rule gilt. Darunter werden die Attribute `Bool c` und `Int d` festgelegt. Die Schlüsselwörter `flowleft` und `flowright` legen fest, in welche Richtung die Attribute propagiert werden. Dieser Vorgang ist in Abbbildung 4.8 schematisch dargestellt: `flowleft` kennzeichnet Attribute, die von rechts nach links durch die Verfeinerungshierarchie propagiert werden bzw. von unten nach oben. `flowright` kennzeichnet entsprechend die entgegengesetzten Richtungen (vgl. auch Abschnitt 2.5). Durch die Schlüsswörter `requires` und `provides` wird spezifiziert, ob die betroffenen Attribute Vor- oder Nachbedingungen sind.

```
1 layer     foo;
2 flowright Bool c;
3 flowleft  Int  d;
4 requires  flowright c;
5 provides  flowleft d >= 10;
```

Abbildung 4.7: Design Rule-Spezifikation

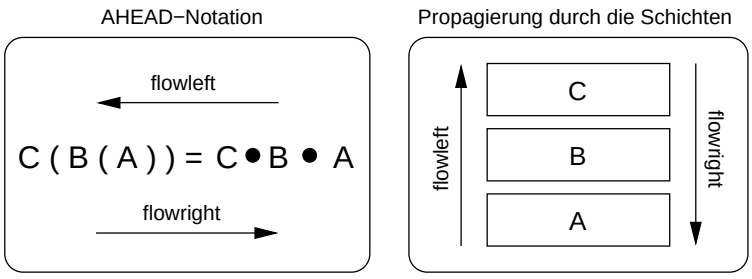


Abbildung 4.8: Propagierungsrichtungen

4.2 Problemanalyse und Anforderungsdefinition

Im folgenden Abschnitt wird eine strukturierte Problemanlayse und die sich daraus ergebenden Anforderungen an die Werkzeugunterstützung diskutiert. Hierzu werden zunächst einzelne Phasen des Entwicklungsprozesses isoliert voneinander betrachtet. Anschließend werden die Probleme und Anforderungen erörtert, die sich aus der mangelnden Integration der Teilprozesse ergeben. Während den Betrachtungen wird von der Entwicklung großer Programmfamilien mit bis zu 1000 Merkmalen ausgegangen. Die Erfahrungen bei der Entwicklung von Programmfamilien haben gezeigt, dass diese Anzahl von Merkmalen leicht erreicht werden kann (LEICH et al. (LAS05)).

4.2.1 Merkmalmodellierung

Kern der Merkmalmodellierung ist die Erstellung eines Merkmaldiagramms. Die vorgestellte Notation basiert auf der traditionellen Baumdarstellung.

Darstellungskomplexität

Aufgrund des einfachen Baumlayouts ergibt sich das Problem der ineffizienten Raumausnutzung (vgl. Abschnitt 3.2) erläutert wurde. Um auch große Merkmaldiagramme übersichtlich und mit Detailinformationen darstellen zu können, muss sowohl der Aufbau als auch die Exploration des Diagramms mit Techniken der Informationsvisualisierung unterstützt werden wie sie in Kapitel 3 vorgestellt wurden.
Es gibt zwei neuere Werkzeuge für den Aufbau von Merkmaldiagrammen, das *Feature Modeling Plug-in* von ANTKIEWICZ et al. und *Pure::Variants* der *pure-systems GmbH*. Beide Anwendungen versuchen den Merkmalbaum platzsparender als Dateibaum anzuordnen. Dies hat jedoch den Nachteil, dass Gruppierungen der Merkmale durch Einfügung neuer Knoten dargestellt werden müssen. Diese Methode erhöht die Kom-

plexität der Darstellung unnötig. Zudem sind Gruppierungen so nur schwierig „auf den ersten Blick" erkennbar.

4.2.2 Kollaborationsentwurf

Die Identifizierung von mehr als 1000 Merkmalen bei der Entwicklung von Programmfamilien ist keine Seltenheit. Zu diesem Merkmalen werden in jeder Phase des Domain Engineerings eine Reihe weiterer Eigenschaften und Beziehungen gesammelt. Die Komplexität dieser Daten ist ohne Werkzeugunterstützung für den Anwender kaum noch überschau- und händelbar.

Merkmalinteraktion
Merkmale interagieren auf unterschiedliche Weise auf Analyse-, Entwurfs- und Implementierungsebene miteinander. Während des Entwurfs benötigt der Entwickler einen Überblick über alle bereits identifizierten Beziehungen. Wichtig ist in diesem Zusammenhang die Sicherstellung, dass durch Entwurfsentscheidungen des Entwicklers keine inkonsistenten Informationen bezüglich der Merkmalinteraktionen entstehen.

Merkmalrepräsentation
Dem AHEAD-Modell liegen Kollaborationen und Merkmale als zentrale Abstraktionseinheiten zugrunde. Eine adäquate Darstellung muss die Verfeinerungshierarchie der Merkmale und ihrer Implementierungsartefakte widerspiegeln, sowie darüber Auskunft geben, welche Merkmale miteinander kombiniert werden können oder müssen. Keine der drei Ansichten, die der ModelExplorer bietet, wird diesen Anforderungen gerecht. Die textuelle Festlegung von Beziehungseigenschaften in Design Rules, wie es das AHEAD-Modell vorsieht, sind ebenfalls nicht geeignet, um einen schnellen Überblick zu gewinnen.

4.2.3 Merkmalorientierte Programmierung

Während der Umsetzung ist Sourcecode nach wie vor die wichtigste Repräsentationsform einer Software. Aus diesem Grund besteht eine der wichtigsten Aufgaben einer integrierten Entwicklungsumgebung darin, dem Entwickler den Sourcecode strukturiert darzustellen und ihm die einfache Navigation an die gewünschten Stellen des Codes zu ermöglichen.

Variabilität der Verfeinerungshierachie
Quellcodeergänzungen sind heute Standart jeder Entwicklungsumgebung. Die Basissprache der AHEAD-Toolsuite ist die Jak-Sprache. Sie unterscheidet sich nur in einigen sehr wenigen Schlüsselwörtern von Java. Eine einfache Erweiterung vorhandener Wizards und Sourcecode-Ergänzungsfunktionlitäten, die für Java zur Verfügung stehen, ist dennoch nicht möglich. Dies liegt an der dynamischen Verfeinungsstruktur, die keine eindeutige Festlegung der Reihenfolge der Schichten zulässt.

Verfeinerungsstruktur auf Quellcodeebene
Während der Implementierung ensteht eine Verfeinerungsstruktur auf Quellcodebene, da jedes Klassenfragment neue Variablen und Funktionen einführen, sowie unterschiedliche Funktionen verfeinern kann. Diese Zusammenhänge sind durch Betrachtung des Quellcodes nur schwer zu überschauen. Daher ist eine Visualisierung wünschenswert, die einen guten Überblick und dabei gleichzeitig die Navigation durch den Quellcode ermöglicht.

Design Rules
Die Sicherung der semantischen Korrektheit bei der Kombination von Merkmalen wird durch Überprüfung von Design Rules umgesetzt. Die Erzeugung der Design Rules ist daher ein Kernelement der Softwareentwicklung. Das AHEAD-Modell sieht vor, dass der Entwickler die Design Rules selbständig unter Zuhilfenahme einer attributiven Grammatik formuliert. Die AHEAD-Toolsuite bietet hierfür keinerlei Unterstützung. Dies bedeutet einen hohen manuellen Aufwand durch den Programmierer und ist zudem sehr fehleranfällig, da es schwierig ist über alle Beziehungen den Überblick zu behalten. Eine Automatisierung und eine visuelle Unterstützung bei der Design Rule-Erzeugung sind daher sinnvoll. Ein weiteres Problem besteht darin, dass das AHEAD-Modell die Erzeugung von Design Rules erst auf Implementierungsebene vorsieht. Eine große Anzahl semantischer Informationen wird jedoch bereits in den vorhergehenden Entwicklungsphasen zusammengetragen, so dass es sinnvoll ist die Design Rules sukzessiv während der verschiedenen Phasen zu erweitern.

4.2.4 Integration des Entwicklungsprozesses

Durch verschiedene Konzepte des Domain Engineerings werden dem Softwareentwickler Methoden und Verfahren vorgeben, nach denen er Programmfamilien entwickeln kann. Leider existieren derzeitig nur fragmentarische Ansätze, um diese Me-

thoden und Konzepte durch eine integrierte Werkzeugunterstützung konsistent und zu einem hohen Grad automatisiert durchzuführen. Bei der Entwicklung großer Programmfamilien entstehen die folgenden zwei Problemfelder.

Mehraufwand und Inkonsistenzen
Die Basis der Merkmalorientierten Softwareentwicklung ist der dreiphasige Domain Engineering-Prozess. Da kein Werkzeug für eine integrierte Abarbeitung aller Phasen zur Verfügung steht, müssen die Ergebnisse von den vorhergehenden Phasen manuell in die nachfolgenden Phasen des Prozesses übernommen werden. Dies verursacht einen erheblichen Aufwand und ist zudem noch sehr fehleranfällig. So gibt es z. B. zum derzeitigen Zeitpunkt kein Werkzeug, dass die Ergebnisse (z. B. Merkmalbezeichnungen, Eigenschaften und Beziehungen zwischen den Merkmalen) der Merkmalanalyse in den Entwurf übeträgt. Der Entwickler hat nun in der Entwurfsphase die Aufgabe alle bereits in der Analysephase erlangten Ergebnisse in den Entwurfsprozess zu integrieren. Dies ist nicht nur unnötiger Mehraufwand, sondern der Hauptgrund für inkonsistente Wissenszustände zwischen den Phasen. Des Weiteren ist die Softwareentwicklung selten ein streng vorwärts gerichteter Prozess. Oft werden komplexe Zusammenhänge zwischen Merkmalen erst auf Implementierungsebene entdeckt. Dies macht eine Rückpropagierung dieser Informationen in vorhergehende Phasen nötig.

Komplexität der Konfigurierung
Das AHEAD-Modell beschreibt eine konkrete Anwendung der Programmfamilie durch eine Gleichung. Der AHEAD-Model Explorer unterstützt die Generierung dieser Gleichungen (Equation-Dateien) dadurch, dass der Anwender die gewünschten Verzeichnisse (Merkmale/Schichten) interaktiv auswählen kann. Daraus wird dann automatisch eine Equation-Datei erzeugt. Die Verfeinerungsreihenfolge muss jedoch weiterhin der Entwickler festlegen. Hierfür benötigt er detailliertes Wissen über die gesamte Architektur und die Implementierung der einzelnen Merkmale. Für Programmfamilien mit sehr vielen Merkmalen und ihren komplexen Beziehungen ist dies durch einen Menschen kaum zu gewährleisten. Weiterhin muss der Entwickler einen Überblick über die in den Design Rules festgelegten Beziehungen haben. Die in den Equations festgelegten Zusammenstellung von Merkmalen wird durch die AHEAD-Toolsuite *nachträglich* auf Konsistenz bzgl. der Design Rules überprüft. Werden Fehler erkannt, muss der Anwender die Reihenfolge verändern oder Merkmale hinzufügen/entfernen. Je größer die Programmfamilie und die erstellte Konfiguration ist, desto eher und häufiger wird sich dieser Vorgang wiederholen. Ziel muss es sein diesen Prozess der

Konfigurierung auf eine abstraktere Ebene zu überführen und während der Konfigurierung möglichst viele (visuelle) Hinweise auf Beziehungen zu geben.

4.3 Entwurf der Basisfunktionlitäten

In den folgenden Teilabschnitten werden die Grundfunktionalitäten für die drei Phasen des Domain Engineerings, sowie verwendete Methoden und Techniken, beschrieben.

4.3.1 Merkmalmodellierung

CZARNECKI et al. (CE00) unterscheiden drei Hauptfunktionen, die ein Werkzeug für die Merkmalmodellierung zur Verfügung stellen muss:

1. die Unterstützung einer Diagrammnotation,
2. die Verknüpfung des Merkmalmodells mit anderen Modellen und
3. die Verwaltung zusätzlicher Informationen, die von Interesse sein können[2].

Die Erstellung eines Merkmaldiagramms ist das Hauptziel der Merkmalmodellierung und steht daher in diesem Abschnitt im Mittelpunkt. Das Hauptziel der Merkmalorientierte Domänenanalyse ist die Erstellung eines Merkmaldiagramms. Um diese Arbeit einzugrenzen werden nur Funktionalitäten für den Aufbau eines Merkmaldiagramms betrachtet.

Diagrammansicht

Die Diagrammnotation von CZARNECKI et al. (CE00) bietet grundsätzlich eine übersichtliche und intuitive Darstellung der Zusammenhänge und Beziehungen zwischen den Merkmalen. Zudem ist sie in der vorgestellten Form oder leicht abgewandelt sehr häufig in der Literatur zu finden, so dass sie bereits einen großen Bekanntheitsgrad unter potentiellen Anwendern hat. Aufgrund des einfachen Baumlayouts kommt es jedoch bei einer großen Anzahl darzustellender Merkmale zu den bereits mehrfach erläuterten Problemen (vgl. Abschnitt 4.2.1), so dass Darstellungsalternativen in Betracht gezogen werden müssen.

2 CZARNECKI et al. (CE00) listen eine Reihe von Informationen auf, die neben dem Merkmaldiagramm das Merkmalmodell komplettieren. Dazu gehören u. a. semantische Beschreibungen der Merkmale, Begründung für den Einsatz eines Merkmals und Beispielsysteme.

Darstellungsalternativen

Bei der Beurteilung der Darstellungsalternativen stellt sich vor allem die Frage, ob neben der hierarchischen Eltern-Kind-Beziehung auch die Gruppierungen und Attribute angemessen dargestellt werden können. Bei der Betrachtung eines Merkmaldiagramms ist es darüber hinaus wichtig, mehrere Hierarchieebenen und deren Attributierung auf einen Blick erfassen zu können, damit Aussagen über Merkmalinteraktionen getroffen werden können.

Treemap
Der Vorteil einer Treemap-Visualisierung bestünde darin, dass zusätzliche Informationen zu den Merkmalen dargestellt werden könnten. Die Attributierung der Merkmale als optional oder zwingend könnte z. B. durch Farbe repräsentiert werden. Schwieriger ist die Gruppierung von Merkmalen, da die Schachtelung und enge räumliche Anordnung der Knoten bereits eine Form der Gruppierung suggeriert. Eine weitere Gruppierung von Merkmalen ist nur schwierig umsetzbar ohne die Übersichtlichkeit der Darstellung negativ zu beeinflussen. In jedem Fall müssten Gruppierungen in Abhängigkeit des gerade selektierten Knotens aus- und eingeblendet werden. Dies gestaltet die Erfassung der Merkmalinteraktionen schwierig. Hinzu kommt, dass es bei größeren Hierarchiedarstellungen schwierig ist, aufgrund der engen räumlichen Anordnung mehrere Hierarchiestufen zu überschauen. Aus den genannten Gründen ist eine Treemap-Visualisierung ungeeignet für ein Merkmaldiagramm.

Cone und Cam Tree
Eine Adaption der Diagrammnotation auf Cone oder Cam Tree ist einfach, da sie die grundsätzliche Struktur der traditionellen Baumdarstellung mit Knoten und Kantenverbindungen erhalten. Grundsätzlich käme für eine Adaption nur der Cam Tree in Frage, da die Knoten des Baumes die Merkmalnamen aufnehmen müssen. Dies wäre beim Cone Tree nicht möglich. Das Hauptproblem bei dreidimensionalen Darstellungen ist die Verdeckung (vgl. Abschnitt 3.2.1). Bei einem Merkmaldiagramm wird dieses Problem bedeutender, da weitere grafische Hinweise integriert werden müssen und dadurch weitere Teile der Rückseite verdeckt werden. Bei einem Cone Tree, der nur aus Knoten und Kanten besteht, können noch relativ viele Informationen über Knoten auf der Rückseite des Baumes aus dem Zusammenhang erschlossen werden und so den Kontext bilden. Die Betrachtung der Merkmale und ihre Attributierungen und Gruppierungen erfordert dagegen eine relativ störungsfreie Ansicht. Besonders

in den unteren Hierarchiebereichen, bei denen der Kegelradius relativ klein ist, können größere Gruppierungen möglicherweise nicht mehr als Ganzes dargestellt werden. Für kleiner Merkmalbäume könnte ein Cam Tree eine Alternative sein. Für große Programmfamilien ist die dreidimensionale Darstellung weniger geeignet.

Hyperbolische Ebene
Die Anordnung des Diagramms auf der hyperbolischen Ebene wäre eine weitere Möglichkeit, um große Merkmalbäume darzustellen. Aufgrund der kreisförmigen Ausrichtung des Baumes besteht hier allerdings das Problem, dass es schwierig ist Merkmale mehrerer Hierarchiestufen zusammen darzustellen. Eine Darstellung von mehr als drei Hierarchiestufen ist kaum zu erreichen (vgl. Abbildung 3.5) .

Insgesamt erfüllt die in den softwaretechnischen Grundlagen vorgestellte Visualisierung eines Merkmaldiagramms die Anforderungen, die zu Beginn dieses Abschnitts genannt wurden, am besten. Um die beschriebenen Nachteile aufzufangen, müssen zusätzliche Visualisierungs- und Interaktionsmechanismen integriert werden, um das Diagramm z. B. komprimierter darstellen zu können.

Editier- und Explorationsfunktionalität

Für den interaktiven Aufbau des Diagramms müssen Grundfunktionalitäten, wie das Erzeugen, Löschen und Anordnen von Knoten integriert werden. Darüber hinaus sind Funktionen für die automatische Anordnung des Baumes, wie z. B. die Ausrichtung an einem Raster nützlich. Um die Nachteile der traditionellen Baumansicht zu mildern werden, angelehnt and das *Information Seeking Mantra*[3] von SHNEIDERMAN (Shn96), drei Strategien verfolgt:

- der Wechsel zwischen Übersicht und Detailansicht,
- die Anpassung der Darstellungskomplexität und
- die Verwendung von *Detail-on-Demand*-Mechanismen.

Der Wechsel zwischen Übersicht und Detailansicht kann durch einen Zoom-Mechanismus, *Focus & Context*- oder *Detail & Overview*-Techniken realisiert werden. Die beiden letzt genannten Methoden haben dabei den Vorteil, dass sie keinen Wechsel zwischen beiden Ansichten erforderlich machen.

3 *„Overview first, zoom and filter, then detail-on-demand"* (vgl. auch Abschnitt 3.1.1).

Bei der Betrachtung oder Bearbeitung des Diagramms wird es häufiger der Fall sein, dass nur ein bestimmter Teilbaum für den Entwickler von Interesse ist. In diesen Fällen ist es nützlich, wenn der Nutzer die Komplexität der Ansicht nach seinen Wünschen anpassen kann, indem er Teilbäume ein- und wieder ausblendet, oder bestimmte innere Knoten/Merkmale „zusammenzieht". Weitere Filterungsmechanismen, die Merkmale mit bestimmten Attributen ausblenden, sind auch möglich aber wenig sinnvoll, da hierdurch die Struktur des Baumes aufgelöst wird. *Detail-on-Demand*-Mechanismen haben den Vorteil, dass durch sie zusätzliche Informationen zu den Merkmalen oder dem Diagramm dargestellt werden können, ohne dass die Darstellung überladen oder die Notationskonventionen verletzt werden. Detail-on-Demand-Informationen können textuell oder durch grafische Hinweise dargestellt werden. Semantische Zusatzinformationen über Merkmale müssen als Text z. B. als Tooltips dargestellt werde. Strukturelle Zusammenhänge werden dagegen besser durch grafische Hervorhebung (z. B. durch Farbe) betroffener Merkmale dargestellt. Ein Beispiel hierfür wäre die Hervorhebung aller optionalen Merkmale.

4.3.2 Kollaborationsentwurf

Während der Entwurfsphase werden die umzusetzenden Implementierungsartefakte und weitere Beziehungseigenschaften der Merkmale festgelegt. Während dieses Vorgangs ist eine visuelle und interaktiv veränderbare Darstellung der entwickelten Ergebnisse nützlich. Eine Entwurfsansicht, deren Elemente interaktiv erzeugt und angeordnet, aber auch wieder verworfen werden können, helfen dem Entwickler den Entwurf schrittweise zu entwickeln.
Eine geeignete Repräsentation muss Merkmale und Implementierungsartefakte so darstellen, dass alle Beziehungen zwischen Merkmalen und Implementierungsdateien eindeutig und einfach erkennbar sind. Es gibt eine Reihe unterschiedlicher Zusammenhänge, die dargestellt werden müssen:

1. die Zuordnung zwischen Merkmalen und ihren Implementierungsdateien,
2. die Zuordnung der Implementierungsdateien, welche das selbe Merkmale implementieren,
3. die Verfeinerungshierarchie auf Merkmalebene,
4. die Verfeinerungshierarchie auf Dateiebene und
5. die Korrelation zwischen sich ausschließenden und gegenseitig bedingenden Merkmalen.

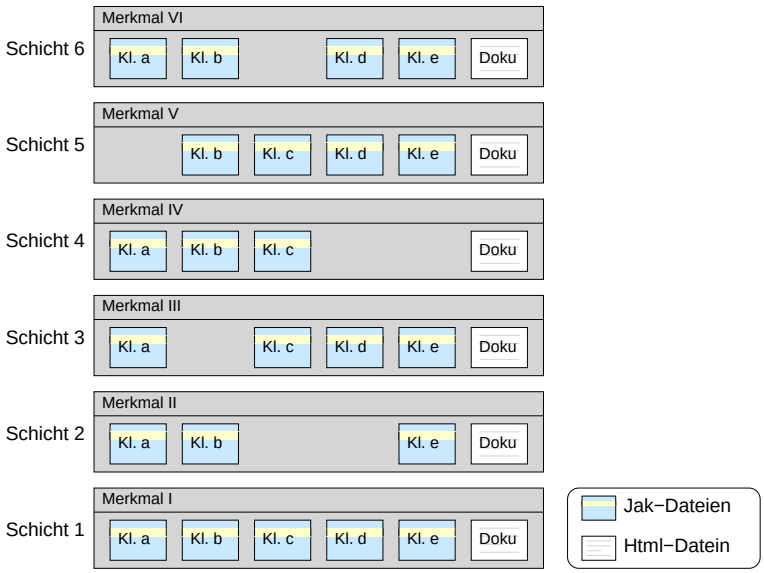


Abbildung 4.9: Repräsentation von Merkmalen als Kollaborationshierarchie. Jedes Merkmal besteht aus einer Reihe von Jak-Klassenfragmenten und einer Html-Dokumentation.

Die schematische Darstellung der Kollaborationen und ihrer Verfeinerungshierarchie aus Abbildung 2.5 bietet einen guten Ansatzpunkt, da diese Darstellung die ersten vier Anforderungen erfüllt. Darüber hinaus hat der Gebrauch dieser Darstellungsform eine Reihe weiterer Vorteile:

- sie ordnet Merkmale und Dateien in den Entwurfskontext ein,
- sie nutzt die Bildschirmfläche effizient aus,
- sie ist aus der Fachliteratur über den Kollaborationenentwurf oder die Merkmalorientierte Programmierung bekannt und
- sie kann einfach um zusätzliche Informationen durch Text oder grafische Hinweise erweitert werden.

Abbildung 4.9 zeigt ein mögliches Beispiel. Die Merkmale werden als Schichten repräsentiert. Auf den Schichten sind die verschiedenen Implementierungsartefakte abgebildet und die Farbe oder Textur der Dateirepräsentationen geben Auskunft über den dargestellten Dateityp.
Wurde der Entwurf festgelegt, kann die Darstellung der Schichten und Artefakte während der Implementierung neben dem Dateibaum als zusätzliches Navigationsinstrument dienen.

Darstellung von Merkmalinteraktionen

Alle Merkmale und Implementierungsartefakte sollen als eine kompakte Kollaborationenhierarchie dargestellt werden. Es gibt jedoch Merkmale, die sich nicht gemeinsam oder in einer eindeutigen Reihenfolge in die Verfeinerungshierarchie einordnen lassen. Das erstgenannte Problem tritt bei sich gegenseitig auschließenden Merkmalen auf. Das zweite Problem wird durch austauschbare Merkmale verursacht, deren Reihenfolge irrelevant ist. Für eine korrekte Repräsentation der Merkmale ergibt sich daher die Notwendigkeit eines Mechanismus, durch den sich gegenseitig ausschließende oder austauschbare Merkmale eindeutig erkennbar sind. Andernfalls würde dem Entwickler eine Verfeinerungshierarchie suggeriert werden, die nicht erlaubt bzw. nicht eindeutig ist.
Wenn Merkmale nicht gemeinsam dargestellt werden dürfen, kann dies durch eine Art Detail-on-Demand-Funktionalität gelöst werden. Ist eine Schicht/ein Merkmal selektiert worden, werden alle Schichten minimiert, die mit der selektierten Schicht nicht Teil einer Konfiguration sein dürfen. Hierdurch sind diese Schichten eindeutig als ausgeschlossen gekennzeichnet. Die Schichten werden wieder maximiert, wenn sie durch den Anwender selektiert werden oder eine andere Schicht ausgewählt wurde, welche die minimierten Schichten nicht mehr ausschließt. Die Kennzeichnung austauschbarer Merkmale/Schichten erfolgt als „optische Zusammenfassung" durch eine Kantenverbindung (vgl. Abbildung 4.10).

Editier- und Explorationsfunktionalität

Um den Entwurf festzulegen muss der Entwickler, neben Grundfunktionalitäten zur Erzeugung von Schichten und Dateien, die Möglichkeit haben die Beziehungen zwischen den Merkmalen/Schichten festzulegen. Zum einen die Festlegung sich ausschließender Merkmale und zum anderen die Bestimmung austauschbarer Merkmale. Da der fertige Entwurf während der Implementierung als Navigationsinstrument dienen soll, muss jede Dateirepräsentation der Entwurfsansicht mit einer Dateirepräsentation des Betriebssystems verknüpft werden.
Die Entwurfsansicht ermöglicht eine deutlich effizientere Nutzung des Darstellungsbereiches als das Merkmaldiagramm. Trotzdem sollten auch hier zusätzliche Interaktionsmöglichkeiten vorgesehen werden, welche die Exploration der Daten verbessern. Hierbei werden die gleichen drei Strategien angewendet wie in Abschnitt 4.3.1.

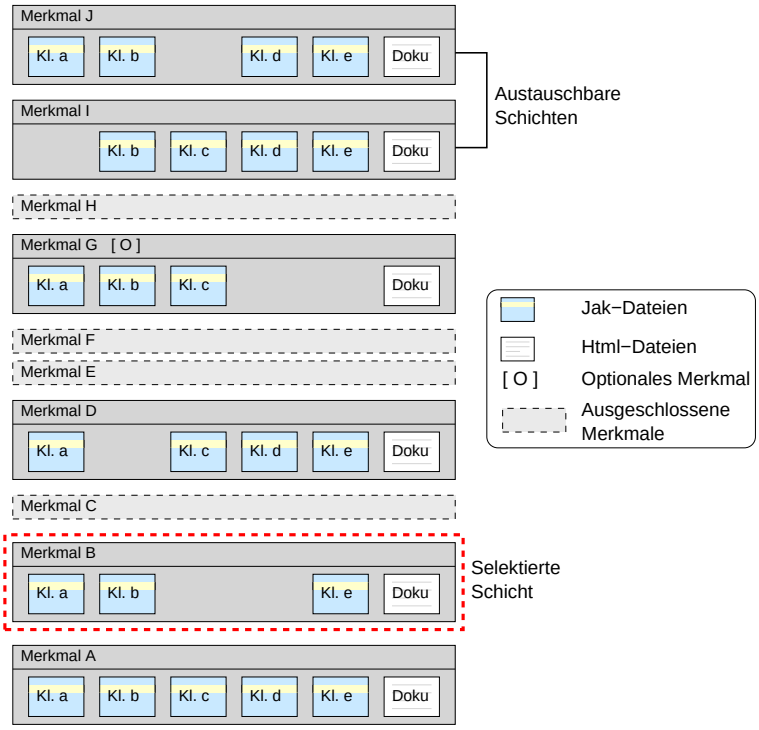


Abbildung 4.10: Entwurfsansicht mit optionalen Merkmalen und ausgeschlossenen Merkmalen bzgl. des selektierten Merkmals A.

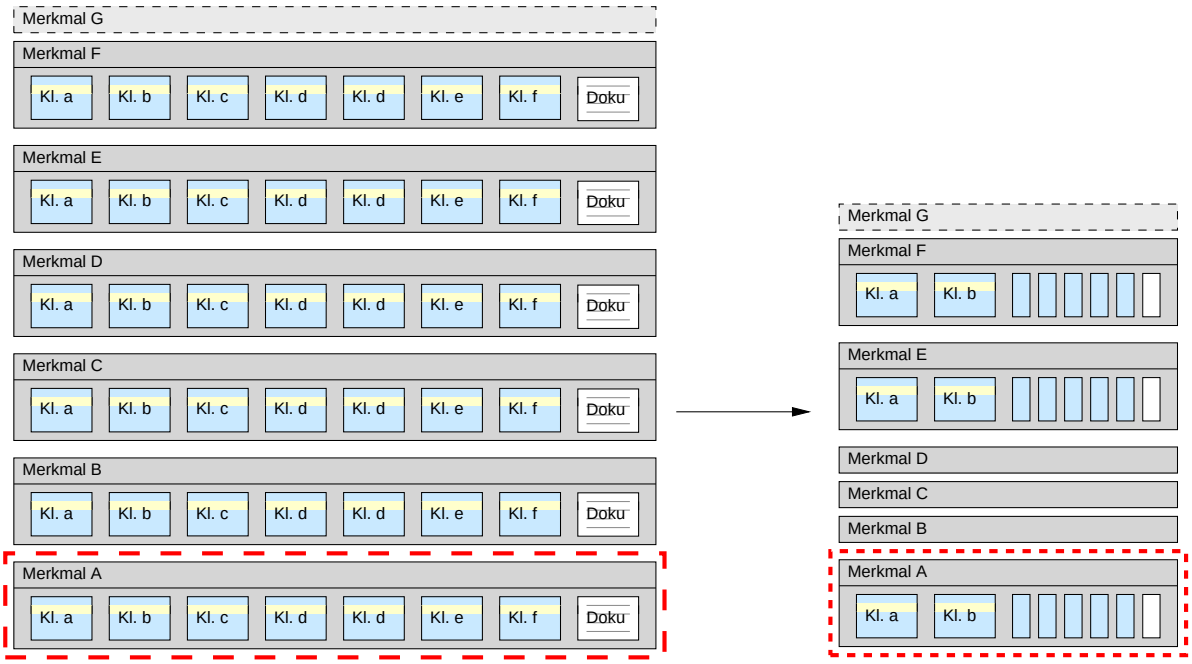


Abbildung 4.11: Reduktion der Darstellungskomplexität.

Um die Komplexität der Ansicht zu verändern, können Schichten und Dateiansichten durch den Anwender interaktiv minimiert oder maximiert werden. Abbildung 4.11 zeigt wie die Größe der Ansicht durch diese Funktionalität in vertikaler und horizontaler Richtung angepasst werden kann. Durch Ausblendung einiger Schichten oder Klassenfragmente nach bestimmten Filterkriterien kann die Darstellung ebenfalls den Wünschen des Entwicklers angepasst werden. Mögliche Filterkriterien sind der Dateityp, der Dateiname oder die Attributierung der Schichten/Merkmale ensprechend ihrer Analyseergebnisse.
Auch für die Entwurfsansicht können *Detail-on-Demand*-Mechanismen verwendet werden. Wichtige zusätzliche textuelle Informationen, die über Tooltips dargestellt werden können, sind z. B. die Design Rules der Schichten. Informationen entsprechend der oben genannten Filterkriterien können durch Farbe oder anderweitige grafische Markierungen hervorgehoben werden.

4.3.3 Merkmalorientierte Programmierung

In diesem Abschnitt werden nur Grundfunktionalitäten für einen Jak-Editor und eine mögliche Visualisierung der Verfeinerungsstruktur auf Quellcodebene vorgestellt. Wie

eine werkzeugunterstützte Erzeugung von Design Rules umgesetzt werden kann wird erst in Abschnitt 4.4.1 zusammen mit den Integrationsfunktionalitäten erörtert, da die Erstellung der Design Rules nicht mehr ausschließlich während der Implementierung, sondern auch während der beiden anderen Phasen erfolgen soll.

Jakarta-Editor

Die Haupttätigkeit der Implementierungsphase ist die Umsetzung der Merkmale mit der Programmiersprache *Jak*, so dass in das zu entwickelnde Werkzeug ein Jak-Editor integriert werden muss. Die Minimalanforderung an den Editor sind die üblichen Editorfunktionen wie Syntax-Highlighting, Kopieren, Einfügen, Suchen etc.
Bei Standartentwicklungsumgebungen für die Objektorientierte Programmierung wie *Eclipse* oder *Microsoft.net* sind Quellcodeergänzungen und Navigation durch die Vererbungshierarchie aus dem Quellcode heraus üblich. Diese Techniken können auch für Jak-spezifische Eigenheiten der Programmmierung genutzt werden. Durch die Verfeinerung von Methoden oder Konstruktoren werden im Quellcode, immer wieder die gleichen Methoden- und Konstruktorensignaturen genutzt. Daher ist es sinnvoll den Entwickler bei der Erzeugung dieser Codefragmente durch Quellcodeergänzung zu unterstützen.
Analog zu der Navigation durch die Vererbungshierarchie kann die Jak-Implementierung unterstützt werden, wenn aus dem Quellcode heraus durch die Verfeinerungshierarchie navigiert werden kann. Der Nutzer selektiert eine Signatur im Quellcode und kann dann zu dem darüber oder darunter liegenden Klassenfragmenten „springen". Die notwendigen Informationen über die Verfeinerungshierarchie können aus dem Entwurf gewonnnen werden.

Codevisualisierung

Die vorgestellte Entwurfsansicht bietet eine gute Übersicht über die Verfeinerungshierarchie auf Dateiebene. Bei der Implementierung entsteht jedoch auch eine Verfeinerungsstruktur auf Quellcodebene. Jedes Klassenfragment kann neue Variablen und Funktionen einführen, sowie unterschiedliche Funktionen verfeinern. Diese Zusammenhänge sind durch Betrachtung des Quellcodes nur schwer zu überschauen. Daher ist eine Visualisierung wünschenswert, die einen guten Überblick und dabei gleichzeitig die Navigation durch den Quellcode ermöglicht. Für eine übersichtliche und

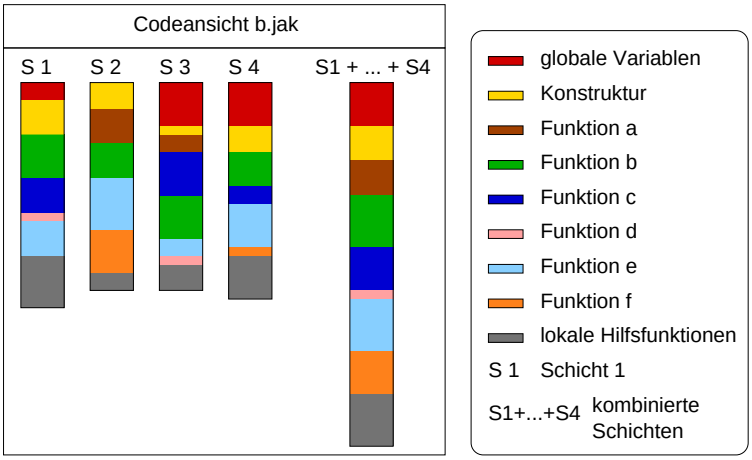


Abbildung 4.12: Codevisualisierung

vergleichbare Darstellung bietet sich die SeeSoft-Visualisierung als Grundlage an (vgl. Abschnitt 3.3.1). Abbildung 4.12 zeigt, wie die Symbolik der SeeSoft-Visualisierung genutzt wird, um die Verfeinerungsstruktur mehrerer Jak-Dateien darzustellen.
Jeder Balken repräsentiert ein Klassenfragment, das die gleich Klasse verfeinert. Die Reihenfolge der Balken von links nach rechts entspricht der Darstellungsreihenfolge der Schichten von unten nach oben in der globalen Entwurfsansicht. Die Höhe der Balken repräsentiert die Größe der Jak-Dateien. Jeder Streifen innerhalb eines Balkens steht für eine Codefragmentklasse (Konstruktor, globale Variablen, „öffentliche" Funktionen, „private" Hilfsfunktionen). Globale Variablen und Hilfsfunktionen, die nicht verfeinert werden, werden als rote bzw. graue Streifen dargestellt. Alle anderen Fragmente bekommen eine eigene Farbe. Die Höhe der Streifen kodiert die Größe des Codefragments. Aus Abbildung 4.12 können nun sehr leicht eine Reihe von Informationen über die Verfeinerungsstruktur abgeleitet werden. Alle Schichten verfeinern den Konstruktor, Schicht 2 führt keine weitere globale Variable ein, fügt jedoch die Funktion a hinzu. Funktion a wird durch Schicht 3 weiter verfeinert.

Debugging

Die SeeSoft-Darstellung kann auch für Debugging-Zwecke genutzt werden. Wie bereits erwähnt wurde, müssen notwendige Änderungen im Quellcode einer kombinierten Jak-Datei in die Klassenfragmente rückpropagiert werden. Zu diesem Zweck muss der Entwickler wissen, aus welchem Klassenfragment der betroffene Teil stammt.

4.4 Entwurf der Integrationsfunktionalität

In den drei Phasen des Domain Engineerings (Analyse, Entwurf und Implementierung) werden neue Informationen über die zu entwickelnde Programmfamilie gesammelt. Im Idealfall ergänzen sich diese Informationen zu einer umfassenden und konsistenten Beschreibung der Programmfamilie. Dieses Wissen wird dann während der Konfigurierungsphase genutzt, um eine Anwendung aus den Merkmalen der entwickelten Progammfamilie zu erstellen.

4.4.1 Automatische Design Rule-Erzeugung

Design Rules sollen bereits während der Analyse und des Entwurfs erzeugt werden. Ziel dieses Abschnittes ist es Regeln zu entwickeln, um dem Anwender automatisch Vorschläge für Design Rules unterbreiten zu können. Da die Interaktionen sehr vielfältig sind, werden nur die Beziehungen betrachtet, die durch das Merkmaldiagramm abgebildet werden können.

Merkmaldiagramm

Betrachtet wird das zwingende Merkmal `current`. Für dieses Merkmal wird zunächst eine Design Rule angelegt, die folgende Layer-und Attributdefinitionen enthält:

```
layer current;
flowleft Bool currentUp;
flowright Bool currentDown;
provides flowleft currentUp;
provides flowright currentDown;
```

Durch die Bereitstellung der Attribute `currentUp` und `currentDown` wird die Existenz von Merkmal `current` in beide Richtungen der Hierarchie (nach oben bzw. nach unten) signalisiert.

Elternmerkmal

Hat Merkmal `current` ein Elternmerkmal `parent`, muss sichergestellt werden, dass Merkmal `current` nur in Verbindung mit seinem Elternmerkmal genutzt wird:

```
extern[4] flowleft parentUp;
requires flowleft parentUp;
```

Alternativ-Gruppierung

Ist ein Merkmal Teil einer Alternativ-Gruppierung, muss sichergestellt werden, dass genau ein Merkmal aus der Gruppierung ausgewählt wird. Hierzu werden zwei weitere Attribute im Elternmerkmal `parent` erzeugt:

```
flowleft Bool parent;
flowright Bool children;
provides flowleft parent;
requires flowright !children;
```

Bei allen Merkmalen der Gruppierung wird Folgendes eingefügt:

```
extern flowleft Bool parent;
extern flowright Bool children;
requires flowleft parent;
provides flowleft !parent;
provides flowright !children
```

Alle Kindermerkmale der Gruppierung erwarten das Attribut `parent` nicht-negiert. Das erste Merkmal der Gruppierung, das eingefügt wird negiert das Attribut jedoch, so dass ein Fehler auftritt, wenn ein weiteres Merkmal der Gruppierung eingefügt wird. Damit festgestellt wird, ob überhaupt ein Merkmal der Gruppierung eingefügt wurde, setzt das Elternmerkmal voraus, dass das Attribut `children` durch eines seiner Kindermerkmale negiert wurde.
Bei Alternativ-Gruppierungen, die mindestens ein optionales Merkmal enthalten, muss nur gewährleistet sein, dass höchstens ein Merkmal aus der Gruppierung ausgewählt wird [5].

4 Das Schlüsselwort *extern* bedeutet, dass das Attribut in einer anderen Design Rule, hier die des Elternmerkmals, definiert wurde.

5 Eine Alternativ-Gruppierung mit mindestens einem optionalen Merkmal kann wie eine Alternativ-Gruppierung behandelt werden, die nur aus optionalen Merkmalen besteht (CZARNECKI et al. (CE00)).

Oder-Gruppierung

Bei einer Oder-Gruppierung, deren Merkmale alle zwingend sind, muss ebenfalls nur sichergestellt werden, dass mindestens eines dieser Merkmale integriert wird. Dies erfolgt analog zur Alternativ-Gruppierung[6].

Kindermerkmal

Bezüglich der Kindermerkmale müssen nur die Merkmale betrachtet werden, die zwingend sind, oder Teil einer Gruppierung, die kein optionales Merkmal enthält. Ist ein Kind zwingend und nicht Teil einer Gruppierung muss sichergestellt werden, dass es ebenfalls ausgewählt wird:

```
extern flowright Bool childDown;
requires flowright childDown;
```

Sind die Kinderknoten gruppiert, müssen die bereits erläuterten zusätzlichen Attribute eingefügt werden.
Die aufgestellten Regeln bilden nur einen Bruchteil der möglichen Beziehungen, innerhalb einer Programmfamilie ab und können noch stark erweitert werden. Weiterhin muss evaluiert werden, inwiefern diese Regeln in der Praxis genutzt werden können. Dies ist jedoch nicht Thema dieser Arbeit.
Die Design Rules stellen eine sehr kompakte Zusammenfassung der Beziehungen innerhalb der Programmfamilie dar, die jedoch für den Anwender schwierig zu überschauen ist. Andererseits ermöglicht ihre Kompaktheit, bei einer festgelegten Schreibweise, eine schnelle und automatische Auswertung, bei der Widersprüche oder Wiederholungen in den Design Rules erkannt werden können.

4.4.2 Sammlung und Propagierung von Wissen

Die Voraussetzung für eine aufeinander aufbauende Wissenssammlung in den verschiedenen Phasen des Domain Engineerings ist die Präsenz bereits gesammelter Informationen in den nachfolgenden Phasen. Ohne einen entsprechenden Überblick über bereits gefundene Zusammenhänge ist die Wahrscheinlichkeit sehr hoch, dass redundante oder sogar widersprüchliche Informationen zusammengetragen werden.
Während der Merkmalorientierten Domänenanalyse wird z. B. sehr viel Wissen über die Merkmale und ihre Beziehungen zueinander zusammengetragen. Zwischen den

6 Oder-Gruppierungen mit mindestens einem optionalen Merkmal können in nicht-gruppierte, optionale Merkmale aufgelöst werden (CZARNECKI et al. (CE00)).

Merkmalen der Analyse und den Schichten des Entwurfs besteht jedoch keine direkte Kopplung, so dass Wissen aus der Analyse nicht notwendiger Weise in den Entwurfsprozess integriert wird und so inkonsistente Wissenszustände zwischen den Phasen entstehen können. Dies ist eines der zentralen Probleme, die mit dieser Arbeit gelöst werden sollen.
Ziel dieses Abschnitts ist die Erstellung einer Heuristik, mit deren Hilfe die Propagierung von Wissen automatisch durchgeführt werden kann. Durch die automatisch (Vorwärts-) Propagierung soll eine Wissensbasis für die nachfolgenden Phasen geschaffen werden. Aufgrund der unterschiedlichen Abstraktionsgrade muss diese Basis als *Vorschlag* betrachtet werden. Wie gut diese Basis genutzt werden kann hängt in großem Umfang von der Qualität der Analyse ab.

Propagierung und Konsistenz-erhaltung

Wissenspropagierung und Konsistenzerhaltung sind eng miteinander verknüpft, da das eine ohne das andere nicht möglich ist. Die Rückpropagierung von Wissen, also die Übertragung von Wissen in eine vorhergehende Phase, dient sogar ausschließlich der Konsistenzerhaltung. Durch den Aufbau des Merkmaldiagramms und der Entwurfsansicht, wird eine interne Repräsentation der Analyse- und Entwurfsergebnisse geschaffen, die es möglich macht, die Wissensübertragung wenigstens (semi-) automatisch durchzuführen. Die Automatisierung der Propagierungsvorgänge ist wichtig, da hierdurch Inkonsistenten aufgrund falscher Entwicklerentscheidungen vermieden werden können. Bei nicht-automatisierbaren Vorgängen besteht die Strategie darin, Entscheidungen des Entwicklers in die richtigen Bahnen zu lenken oder, wenn dies auch nicht möglich ist, zu versuchen seine Entscheidungen nachträglich zu überprüfen. Im Laufe der weiteren Betrachtungen sollen folgende Fragen beantwortet werden:

- Wie können die Abstraktionseinheiten der verschiedenen Phasen aufeinander abgebildet werden?
- Wie können die Beziehungen zwischen den Abstraktionseinheiten auf andere Phasen übertragen werden?
- Welche Abbildungsvorgänge sind automatisierbar?
- Wie wird die Konsistenz bei nicht-automatisierbaren Vorgängen sichergestellt?

Abbildung der Abstraktionseinheiten

Merkmale der Analyse müssen eindeutig den Schichten des Entwurfs zugeordnet werden können, Schichten des Entwurfs auf Merkmale der Implementierung usw. Die gewählten Abbildungsregeln basieren u. a. auf dem AHEAD-Modell. Das AHEAD-Modell

bildet die Kollaborationen des Entwurfs direkt auf die Merkmale der Implementierung ab. Das heisst, dass jede Schicht des Entwurfs durch genau ein Merkmal der Implementierung umgesetzt wird. Nach SOCHOS et al. (SPR04) sollten Merkmale der Analyse und Entwurfseinheiten möglichst auch 1:1 abgebildet werden, um ein möglichst langlebiges und gut wartbares System zu erhalten. Folglich müssten auch Merkmale der Analyse 1:1 auf Merkmale der Implementierung abgebildet. Diese Abbildungsvorschriften sind sehr einfach und können daher leicht durch ein Werkzeug überprüft werden[7].
Da im Diagramm auch Merkmale vorhanden sein können, die nur der Strukturierung dienen und nicht implementiert werden sollen, kann die Erzeugung der Entwurfsschichten nicht automatisch, ohne vorherige Auswahl der gewünschten Merkmale, erfolgen.
Jeder Schicht des Entwurfs muss jedoch ein Merkmal der Analyse zugeordnet werden, bevor sie in den Entwurf eingefügt werden kann. Durch diese Bedingung wird die Rückpropagierung neuer Informationen sichergestellt. Wenn die Notwendigkeit einer neuen Schicht oder Variante während des Entwurfs erkannt wird, kann diese Schicht erst integriert werden, wenn ein entsprechendes Merkmal in den Baum eingefügt wurde. Diese Konsistenzbedingung wird also durch „Lenkung" des Entwicklers festgelegt. Die Erzeugung von Merkmalen der Implementierung erfolgt analog auf der Basis des Entwurfs.

Automatisierte Abbildung von Beziehungseigenschaften

Aufgrund der Tatsache, dass Merkmalinteraktionen sehr vielfältig sind, ist die Abbildung von Beziehungseigenschaften weniger offentsichtlich. Statt die Struktur des Merkmalbaumes oder der Entwurfsansicht direkt auszuwerten, können auch die Design Rules als Informationsgrundlage genutzt werden, da sie eine kompakte Darstellung der Beziehungseigenschaften sind.

Analyse – Entwurf

Wenn sich Merkmale der Analyse ausschließen muss diese Eigenschaft auf die entsprechenden Schichten übertragen werden. Diese Beziehungsinformationen aus der Analyse werden als Grundlage für den Entwurf genutzt. Als Basis für die Verfeinerungshierarchie wird die Hierarchiestruktur des Baumes genutzt. Abbildung 4.13 zeigt zwei Beispiele für die Wissenspropagierung zwischen Analyse und Entwurf. Dargestellt sind

7 An dieser Stelle muss festgehalten werden, dass dies ein idealisierter Ansatz ist, der in der Praxis möglicherweise nicht immer geeignet ist.

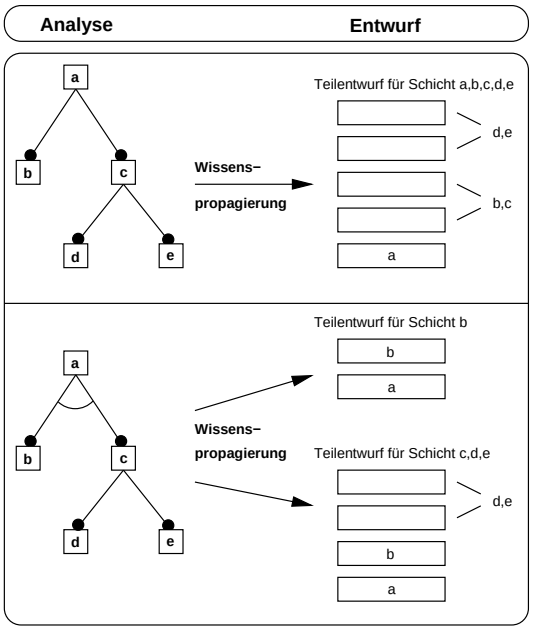


Abbildung 4.13: Heuristik für die Wissenspropagierung zwischen Analyse und Entwurf.

zwei Merkmaldiagramme und die Teilentwürfe, die daraus abgeleitet wurden. In einem ersten Schritt wird für jedes Merkmal eine Schicht im Entwurf erzeugt. Dann können die Beziehungen der Merkmale innerhalb des Modells auf den Entwurf übertragen werden. Im oberen Teil der Abbildung bedingen sich z. B. alle Merkmale gegenseitig. Diese Eigenschaft wird auf den Entwurf übertragen, so dass alle Schichten gemeinsam Teil eines Kollaborations-Stacks sind. Im unteren Teil der Abbildung dagegen schließen sich Merkmal `b` und Merkmal `c` gegenseitig aus. Daher können auch die korrespondierenden Schichten nicht gemeinsam Teil einer Verfeinerungshierarchie sein. Aus der Position der Merkmale im Baum wird eine Halbordnung der Schichten gewonnen. Im Merkmalbaum nimmt der Abstraktionsgrad der Merkmale von der Wurzel zu den Blättern hin ab. In der Verfeinerungshierarchie des Entwurfs nimmt der Abstraktionsgrad von der unteren Schicht hin zu der obersten Schicht ebenfalls ab. Daher wird das Wurzelmerkmal auf die unterste Schicht oder Basisschicht der Verfeinerungshierarchie abgebildet. Direkt über der Basisschicht liegen zunächst die Schichten, deren korrespondierende Merkmale direkte Nachfolger des Wurzelmerkmals sind. Die Basis wird durch Schichten verfeinert, deren korrespondierende Merkmale direkte Nachfolger des Wurzelmerkmals sind usw. Über die Reihenfolge der Schichten, deren Merkmale auf der gleichen Ebene des Diagramms liegen, können jedoch keine Aussagen

getroffen werden, da sie den gleichen Abstraktionsgrad besitzen. Dies kann im Gegensatz zu den beiden vorgestellten Abbildungsvorgänge nicht automatisch durchgeführt werden.

Analyse – Implementierung

Die AHEAD-Toolsuite repräsentiert Merkmale als Verzeichnisse. Zueinander in Beziehung gesetzt werden die Merkmale durch die Verzeichnishierarchie. Aufgrund der 1:1-Abbildung zwischen Merkmalen der Analyse und Merkmalen der Implementierung kann die Hierarchiestruktur des Merkmaldiagramms als Grundlage genutzt werden, um automatisch eine Verzeichnishierarchie für die Implementierung zu generieren.

Entwurf – Implementierung

Da schon während der Erstellung des Entwurfs die notwendigen Implementierungsartefakt erzeugt werden, wird bereits während des Entwurfs das Grundgerüst für die Implementierung geschaffen. Hierdurch gehen Entwurf und Implementierung fließend ineinander über.

Rückpropagierung

Die Rückpropagierung dient ausschließlich der Konsistenzerhaltung. Sie erfolgt, wenn in einer der vorhergehenden Phasen wichtige Eigenschaften der Programmfamilie noch nicht beachtet oder fehlerhaft dargestellt wurden und dies in nachfolgenden Phasen erkannt wird. Wenn z. B. während der Implementierung festgestellt wird, dass ein Merkmal weiter differenziert werden muss, sollten diese veränderten Anforderungen auch in den Entwurf und die Analyse eingefügt werden. Dabei ist zu beachten, dass nur Informationen rückpropagiert werden können, die sinnvoll in die vorhergehenden Phasen integriert werden können. Dies ist besonders schwierig bei der Rückpropagierung in das Merkmaldiagramm. Im Prinzip können die im Merkmaldiagramm dargestellten Eigenschaften bis auf Implementierungsebene aufgefächert werden. Dies ist jedoch nicht sinnvoll, da das Merkmaldiagramm eine implementierungsunabhängige Darstellung einer Programmfamilie sein soll. Werden auf Implementierungsebene neue Informationen gefunden, muss abgewogen werden, ob dieses Wissen einen Abstraktionsgrad besitzt, der angemessen in das Merkmaldiagramm integriert werden kann oder nicht. Auch wenn der Abstraktionsgrad zu niedrig ist, muss immer noch geklärt werden, ob diese Information sich in abstrahierter Form im Merkmaldiagramm wiederfindet. Diese Fragen sind nicht automatisch entscheidbar und müssen durch den Entwickler festgelegt werden.

4.5 Konfigurierungskomplexität

Ziel der Produktfamilienentwicklung ist es, bereits implementierte Komponenten wiederzuverwenden und so den Aufwand für die Erstellung einer neuen Anwendung zu vereinfachen. Während der Konfigurierung stellen sich zwei wichtige Fragen: Welche Komponenten enthält die Programmfamilie und wie muss ich diese miteinander kombinieren? Die Konfigurierung kann zeitlich weit entfernt von der Implementierung liegen. Daher muss es das Ziel sein eine konkrete Konfiguration möglichst deklarativ, also ohne tieferes Implementierungswissen, beschreiben zu können[8]. Hierfür ist ein Werkzeug notwendig, das eine Schnittstelle zwischen abstrakter Beschreibung und Quellcode schafft. Die Equation-Dateien der AHEAD-Toolsuite sind ein erster Schritt, erreichen jedoch noch nicht den notwendigen Abstraktionsgrad für eine vollständig implementierungsunabhängige Konfigurierung. Mit dem Merkmaldiagramm ist bereits eine abstrakte und kompakte Repräsentation der Programmfamilie vorhanden, die sehr gut als Benutzerschnittstelle für die interaktive Auswahl der gewünschten Merkmale genutzt werden kann. In dem Diagramm werden jedoch nur die Beziehungen zwischen den Merkmalen dargestellt, die während der Analysephase identifiziert wurden. Für den Konfigurierungsprozess sollten jedoch auch die Beziehungsinformationen des Entwurfs und der Implementierung in das Diagramm integriert werden. Eine naheliegende Möglichkeit für die Darstellung dieser Beziehungen im Diagramm wäre das Einfügen weiterer Verbindungslinien zwischen den betroffenen Merkmalen. Dies würde jedoch zu einem unübersichtlichen und nur noch schwer verständlichen Diagramm führen. Statt dessen wird eine Funktionalität zur Verfügung gestellt, mit deren Hilfe sich der Entwickler zu selektierten Merkmalen anzeigen lassen kann, welche Merkmale ebenfalls ausgewählt werden müssen/können oder nicht ausgewählt werden dürfen, wenn die selektierten Merkmale Teil einer Konfiguration sein sollen. Da die notwendigen Informationen durch Auswertung der Design Rules gewonnen werden, umfassen die Informationen alle drei Entwicklungsphasen. Die Kennzeichnung erfolgt durch farbliche Markierung der Merkmale. Hierbei wird auch auf Widersprüche hingewiesen, wenn zwei Merkmale markiert sind, die sich gegenseitig ausschliessen. In Abbildung 4.14 wurden die Merkmale a und b selektiert. Die grün markierten

8 Einen ähnlichen Ansatz wird in der Anwendung *pure::variance* verfolgt. Sie verknüpft ebenfalls die abstrakte Beschreibungen einer Programmfamilie mit der Low-Level-Beschreibung der Implementierung. Semantische Bedingungen für die Konfigurierung von Komponenten können in *PROLOG* festgehalten werden (BEUCHE et al. (BPSP03)).

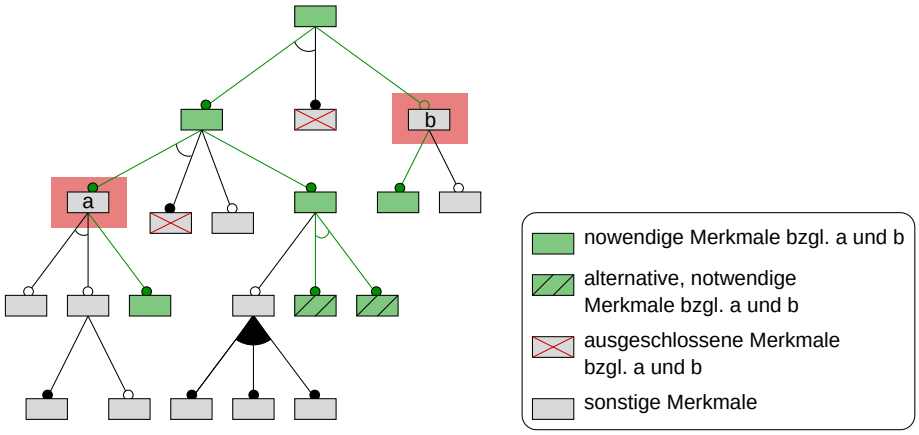


Abbildung 4.14: Heuristik für die Wissenspropagierung zwischen Analyse und Entwurf

Merkmale müssen gemeinsam mit Merkmal a und Merkmal b Teil einer gemeinsamen Konfiguration sein. Die mit einem roten Kreuz gekennzeichneten Merkmale dürfen dagegen nicht hinzugefügt werden. Es gibt eine Reihe weiterer Möglichkeiten, mit denen das Werkzeug eine korrekte Konfigurierung unterstützen kann:

- die Nicht-Selektierbarkeit ausgeschlossener Merkmale,
- die automatische Ergänzung benötigter Merkmale und
- die Überprüfung der potenziellen Konfiguration, bevor ein neues Merkmal hinzugefügt wird.

Nach Auswahl der gewünschten Merkmale für eine Konfiguration erzeugt das Werkzeug automatisch eine Equation-Datei, die dann von der AHEAD-Toolsuite genutzt werden kann.

4.6 Zusammenfassung

In diesem Kapitel wurde analysiert wie das Domain Engineering und die Merkmalorientierte Programmierung integriert unterstützt werden können. Als Verfahren für die Analyse, den Entwurf und die Implentierung wurden die Domänenanalyse, der Kollaborationsentwurf bzw. die Java-Erweiterung *Jak* der AHEAD-Toolsuite genutzt. Die Problemanalyse wurde in vier Teilbereiche gegliedert. Die ersten drei Bereiche beschäftigen sich mit den Problemen, die in den drei Entwicklungsphasen isoliert voneinander auftreten. Der vierte Bereich umfasst die Probleme, die sich durch die integrierte Unterstützung aller drei Phasen ergeben. Tabelle 4.1 fasst die auftretenden Probleme zusammen.

Merkmal-modellierung	Kollaborationen entwurf	FOP + AHEAD	Integration
Darstellungs-komplexität	Merkmal-interaktion Merkmal-repräsentation	Verfeinerungs-variabilität Quellcode - verfeinerung Design Rule-komplexität	Inkonsistenzen u. Mehraufwand Konfigurierungs-komplexität

Tabelle 4.1: Auftretende Problembereiche bei der integrierten Unterstützung der Merkmalorientierten Softwareentwicklung.

Nach der Problemanalyse wurden mögliche Lösungen diskutiert. Um die Darstellungskomplexität während der Erstellung und Betrachtung eines Merkmaldiagramms zu vermindern, wurden verschiedene Visualisierungs- und Interaktionsmöglichkeiten erörtet. Für den Kollaborationsentwurf ist eine grafische Repräsentation entwickelt worden, die Merkmale und ihre Implementierungsartefakte in den Kollaborationsentwurf einordnet und einen Überblick über die bestehenden Merkmalinteraktionen ermöglicht. Die Programmierung mittels der AHEAD-Toolsuite soll in mehrfacher Hinsicht unterstützt werden. Angelehnt an die SeeSoft-Visualisierung kann die Verfeinerungshierarchie auf Quellcodeebene dargestellt werde. Dies gibt dem Entwickler Hilfestellung bei dem Verständnis der Verfeinerungsstruktur und der Navigation durch den Quellcode.
Die automatische Erzeugung der Design Rules wurde im Zusammenhang mit der Integration der Entwicklungsprozesse behandelt, da die Erstellung der Design Rules nicht mehr erst in der Implementierung, sondern bereits in der Analyse und dem Entwurf erfolgen soll. Hierdurch wird eine kompakte Repräsentation der Merkmalinteraktionen innerhalb jeder Entwicklungsphase gewonnen. Für die automatische Erfassung von Design Rules auf Analyseebene wurden eine Reihe von Regeln aufgestellt.
Um Inkonsistenzen zwischen den Phasen und Mehraufwand zu vermeiden ist eine Abbildungsheuristik für die automatische Wissenspropagierung zwischen den Phasen entwickelt worden, insbesondere zwischen der Analyse und dem Entwurf. Mithilfe dieser Heuristik können, auf der Basis der Design Rules, Vorschläge für den Entwurf unterbreitet werden.

Der letzte Abschnitt beschäftigt sich mit der Frage wie die Konfigurierung einer konkreten Anwendung benutzerfreundlicher gestaltet werden kann. Als Lösung wird vorgeschlagen den Konfigurierungsvorgang auf eine abstraktere Ebene zu heben, indem das Merkmaldiagramm von dem Anwender als Schnittstelle für die Auswahl der gewünschten Merkmale genutzt wird. Auf der Basis der Auswahl, die der Anwender getroffen hat, werden Equation-Files dann automatisch erzeugt.

KAPITEL 5

Umsetzung

Aufgrund der vielfältigen Aufgaben hat das zu entwickelnde Werkzeug alle Kennzeichen einer integrierten Entwicklungsumgebung (IDE[1].). Die vollständige Neuimplementierung typischer Funktionalitäten wie die des Texteditors, Dateinavigators oder des Versionsmanagements wäre bereits mit sehr viel Aufwand verbunden, ohne dass die eigentlich zu implementierenden Aufgaben umgesetzt würden. Daher sind die Entwurfsergebnisse des vorherigen Kapitels in die *Eclipse Java-IDE* als *Plugin* integriert worden. Die Eclipse Java-IDE wurde aus folgenden Gründen ausgewählt:

- Eclipse ist als Open-Source-Werkzeug frei verfügbar und erweiterbar,
- Eclipse bietet bereits eine Vielzahl nützlicher Funktionen, die, unabhängig von der jeweiligen Programmiersprache, die Implementierung unterstützen[2],
- Eclipse ist sehr einfach erweiterbar, da der Kern der Plattform auf Erweiterung ausgelegt ist,
- aus den kombinierten *Jak*-Dateien werden *Java*-Dateien erzeugt, so dass auch die Java-Funktionalitäten der IDE genutzt werden können.

In diesem Kapitel wird die Umsetzung des zu entwickelnden Werkzeugs als Eclipse-Plugin beschrieben. Hierfür wird eine kurze Einführung zu Eclipse und die Plugin-Entwicklung gegeben, sowie über weitere genutzte Bibliotheken und Plugins. Anschließend wird erläutert wie die einzelnen Phasen der *Merkmalorientierten Softwareentwicklung* auf der Basis des Entwurfs umgesetzt wurden.

1 engl. **I**ntegrated **D**evelopment **E**nvironment
2 z. B. Ressourcen-Navigator, CVS-Client, Konsole,etc.

5.1 Einführung in Eclipse

Eclipse ist eine Open-Source-Anwendung, deren Entwicklung 2001 durch *IBM* initiiert wurde. Zum aktuellen Zeitpunkt wird Eclipse durch eine Nonprofit-Organisation verwaltet, deren weitere Mitglieder u. a. *Hewlett Packard*, *Intel* und die *SAP AG* sind. Die Organisation entstand, nachdem im November 2001 IBM den Quellcode einer Entwicklungsumgebung in die neu gegründete Open-Source-Gemeinde *Eclipse.org* eingebracht hatte. Die *Eclipse Foundation* gibt mit Eclipse, genauer gesagt mit der *Eclipse Rich Client Plattform* (RCP), eine erweiterbare Entwicklungsumgebung und eine Plattform für die Integration von Programmierwerkzeugen als Open Source heraus.
Am bekanntesten ist die Nutzung von Eclipse als Java-IDE. Grundsätzlich kann Eclipse jedoch als Plattform „für alles Mögliche" angesehen werden. Die Grundlage für diese Vielseitigkeit ist die Plugin-Architektur von Eclipse, die es einfach erlaubt neue Funktionalitäten zu integrieren.

5.1.1 Plugin-Entwicklung für Eclipse

Fast der vollständige Funktionsumfang, der durch die *Eclipse-Workbench* zur Verfügung gestellt wird, ist in Plugins gekapselt. Der Plattformkern von Eclipse wird durch das Plugin *org.eclipse.core.runtime* gebildet, das für die Ausführung aller anderen Plugins zuständig ist. Die Eclipse Java-IDE ist lediglich ein sehr bekanntes Beispiel für ein Eclipse-Plugin. Die allermeisten der derzeitig vorhandenen Plugins erweitern die vorhandene Java-IDE und dienen in irgendeiner Form der Anwendungsentwicklung. Die vornehmliche Aufgabe dieser Plugins ist die Erweiterung und Anpassung von Editoren, Übersichten, Debugger-Funtionalitäten, etc.
Ein Eclipse-Plugin besteht üblicherweise aus einem Java-Archiv und einigen zusätzlichen Dateien wie Bildern und Hilfetexten. Zwingend erforderlich für jedes Plugin ist das *Plugin-Manifest*, eine XML-Datei mit dem Namen *plugin.xml*, welches die Konfiguration des Plugins und seine Integration in die Plattform beschreibt. In dem Manifest wird festgehalten welche existierenden Plugins benötigt und welche erweitert werden.

Erweiterungspunkte

Die Grundlage für die Erweiterung bereits bestehender Plugins bilden *Erweiterungspunkte*[3]. Über diese Punkte „klinken" sich neue Plugins in die bestehende Architektur, genauer gesagt in bereits existierende und erweiterbare Plugins, ein. Die Erweite-

3 engl. Extension Points.

rungspunkte werden durch die erweiterbaren Plugins definiert und geben vor, welche Schnittstellen implementiert und welche Eigenschaften gesetzt werden müssen. Ein Beispiel für die Erweiterung eines Plugins stellt Abbildung 5.1 dar. Es zeigt exemplarisch die Erweiterung der Java-IDE um einen Jak-Editor. Dieses XML-Fragment legt fest, welche Klasse den Jak-Editor implementiert, den Namen des Editors und seine ID. Anhand der Dateierweiterung wird festgelegt, welche Dateitypen der Editor öffnen kann und ob der Editor als Standarteditor für diesen Typ verwendet werden soll. Abschließend wird festgehalten welches Icon dem Editor in der Taskleiste von Eclipse zugeordnet werden soll.

```
1  <extension
2    <editor
3      class="fos.editors.JakEditor"
4      name="Jak-Editor"
5      id="fos.editors.jakEditor"
6      extensions="jak"
7      default="true"
8      icon="icons/jakeditor.gif"/>
9    </editor>
10 </extension>
```

Abbildung 5.1: XML-Fragment für die Erweiterung der Eclipse-Platfrom um einen Jak-Edior.

SWT und JFace

Eclipse stellt die Java-Bibliotheken *SWT*[4] und *JFace* zur Verfügung. Sie stellen eine Alternative zu den Java-Bibliotheken *AWT* bzw. *Swing* von *Sun Microsystems* dar. SWT wurde entworfen um einen einfachen und portablen Zugang zu den Benutzerschnittstellen des jeweiligen Betriebssystems zu erlangen. JFace ist ein *GUI*[5]-Framework, das auf dem SWT aufbaut und erweiterte Unterstützung für Editoren, Dialoge und Wizards bietet. Mithilfe dieses Frameworks ist eine schnelle Entwicklung relativ komplexer Anwendungen mit hochwertigen und einheitlichen Benutzeroberflächen möglich.
Die SWT-Bibliothek ist vollständig unabhängig von den Sun-Bibliotheken AWT und Swing und kann auch in Eclipse-fremden Anwendungen genutzt werden. Ein Unterschied zwischen den Eclipse- und Sun-Bibliotheken besteht darin, dass das Aussehen der Nutzeroberfläche durch die Eclipse-Bibliotheken nicht nachgebildet wird, sondern direkt auf den nativen Betriebssystem-Widgets aufsetzt.

4 engl. **S**tandard **W**idget **T**oolkit.
5 engl. **G**raphical **U**ser **I**nterface: Grafische Benutzeroberfläche.

Ressourcen in Eclipse

Zu Beginn jeder Eclipse-Sitzung wird der Nutzer aufgefordert den *Workspace*[6] festzulegen. Hierfür wählt der Anwender ein Verzeichnis im lokalen Dateisystem aus. Der Dateinavigator von Eclipse zeigt nicht das gesamte lokale Verzeichnissystem, sondern nur die Ressourcen, die im Workspace liegen. Der Pfad einer Ressource im Workspace und der im lokalen Dateisystem ist meist ab dem Workspace-Verzeichnis gleich. Bei Ressourcen, die durch einen Importiervorgang in den Workspace von Eclipse geladen wurden, können sich beide Adressen jedoch unterscheiden. Daraus folgt, dass jede Ressource von Eclipse zwei Dateiadressen besitzt. Eine Adresse im lokalen Dateisystem und eine Adresse im Eclipse-Workspace. Eclipse kennt die drei Ressourcentypen *Projekt*, *Verzeichnis* und *Datei*. Projekte werden im lokalen Dateisystem als Verzeichnisse abgebildet und bilden die Wurzel im Workspace. Sie enthalten alle Verzeichnisse und Dateien, die für die Umsetzung einer Aufgabe notwendig sind.

5.1.2 Konzepte der Nutzeroberfläche der Eclipse Java-IDE

Das Hauptfenster von Eclipse wird als *Workbench* bezeichnet. Die Workbench enthält üblicherweise einen Editor, um den eine Reihe von *Views* angeordnet sind. Abbildung 5.2 zeigt ein typisches Beispiel. Dargestellt wird hier die Java-Perspektive. Eine *Perspektive* ist eine bestimmte Zusammenstellung von Editoren, Views und Menüs, die genau an eine bestimmte Aufgabe angepasst ist. In der Mitte der dargestellten Workbench ist der Java-Editor zu, an dessen Seiten verschiedene Views angeordnet sind. Editoren erlauben es Objekte zu öffnen, zu editieren und abzuspeichern. Views haben im Gegensatz zu Editoren keine Eingabequelle, sondern zeigen Zustandsdaten des aktiven Editors oder der Workbench an[7]. Am linken Rand der Workbench ist der *Package-Explorer* angeordnet. Er dient der Navigtion durch die angelegten Java-Dateien. An der rechten Seite des Editors ist die *Outline-View* angeordnet. Sie gibt eine strukturierte Übersicht auf den im Editor geöffneten Quellcode. Durch Selektion einzelner Fragmente in der Outline-View ist zudem eine Navigation durch das geöffnete Dokument möglich. Unterhalb des Editors ist die Konsole geöffnet, welche Ausgaben der implementierten Anwendung wiedergibt. Die grafische Nutzeroberfläche von Eclip-

6 Arbeitsbereich

7 Die Eclipse Java IDE enthält eine Reihe verschiedener Views, die unterschiedlichste Aufgaben erfüllen können, z. B. für die Textsuche, Fehlerausgabe, Aufgabenverwaltung oder für die Versionsverwaltung mittels CVS.

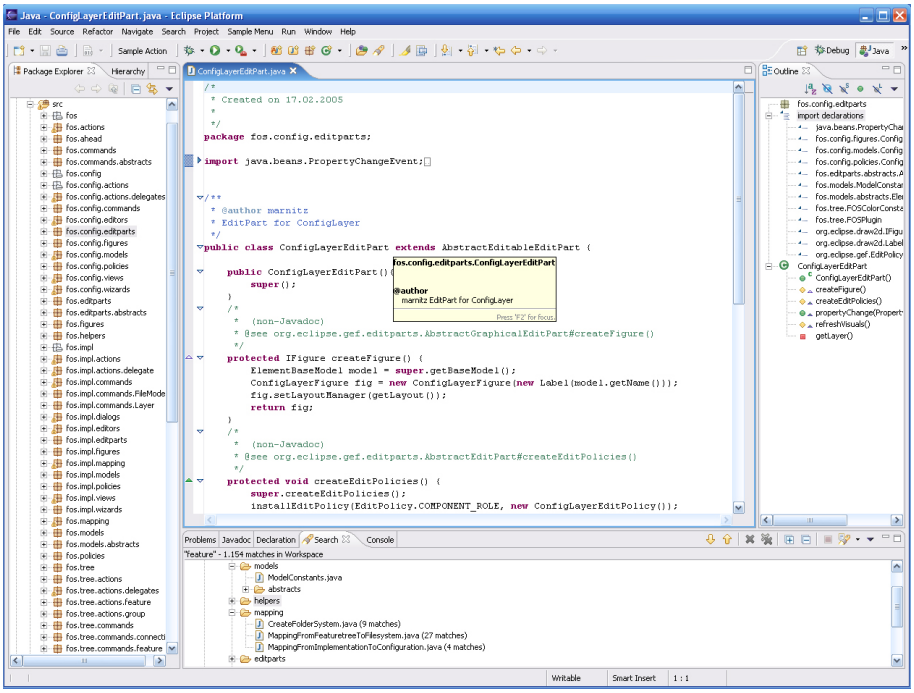


Abbildung 5.2: Originalansicht der Eclipse-Workbench mit geöffneter Java-Perspektive.

se ist sehr flexibel. Die Views können interaktiv um den Editor angeordnet werden. Durch das Hinzufügen oder Schließen von Views, kann die gerade aktive Perspektive nach den Wünschen des Entwicklers individuell verändert werden. Diese angepasste Perspektive kann durch Speicherung persistent gemacht werden.
Eclipse bietet eine Reihe von Erweiterungspunkten, mit denen es um neue Editoren, Views und Perspektiven erweitert werden kann. Editoren und Views haben (Popup-) Menüs und Toolbars, die ihnen zugeordnet sind. Es gibt ebenfalls Erweiterungspunkte, um neue Aktionen zu diesen Menüs und Toolbars hinzuzufügen.

5.2 Graphical Environment Framework (GEF)

Für eine effiziente Erstellung eines Merkmaldiagramms,ebenso wie für die Erstellung der Entwurfsansicht, ist die Existenz eines grafischen Editors die Voraussetzung. Ein weiterer Vorteil der Nutzung von Eclipse besteht darin, dass es aufgrund der großen Nutzergemeinde auch für speziellere Probleme bereits eine Reihe von Plugin-Lösungen oder Bibliotheken gibt, die frei genutzt und angepasst werden können. Auch für die Implementierung grafischer Diagrammeditoren gibt es bereits eine Bibliothek, die ge-

nutzt werden kann: das *Graphical Environment Framework* (GEF). Abgesehen vom Jak-Editor sind alle implementierten und neu integrierten Editoren unter Zuhilfenahme dieses Frameworks umgesetzt worden. GEF basiert auf einer *Model View Controller*-Architectur (MVC)[8]. Ziel dieser Architektur ist ein flexibles Programmdesign, um später notwendige Änderungen oder Erweiterungen einfach gestalten und die einzelnen Komponenten wiederverwenden zu können[9]. Die Grundannahme des GEF ist die Existenz eines Modells, welches grafisch dargestellt und interaktiv editiert werden soll. Die Steuerung, auch als *EditPart* bezeichnet, verknüpft ein Modell mit seiner grafischen Repräsentation und passt die Darstellung an den jeweiligen Zustand des Modells an. Wird ein Modell verändert löst es ein entsprechendes `PropertyChangeEvent` aus. Wenn der EditPart des Modells dieses Ereignis kennt passt er die Darstellung entsprechend.
EditParts sind ebenfalls zuständig für die Anpassung von Modellen, da sie die Objekte eines Diagramms repräsentieren, mit denen der Anwender interagiert. Jede Interaktion (z. B. das Ziehen der Maus, einfach Klicken oder Doppelklicken) löst eine bestimmte Anfrage oder *Request* aus: z. B. die Anfrage ein neues Modell zu erzeugen, ein Modell zu löschen oder die Position eines Modells zu verändern. Kann der betroffene EditPart diesen Request verarbeiten löst er ein *K*ommando oder *command* aus, welches die gewünschte Veränderung an dem Modell kapselt. Auf welche Interaktionen oder Requests ein EditPart reagieren kann wird durch seine *Policys* festgelegt. Je nach Art der angemeldeten Policys kann ein EditPart verschiedene Nutzeraktionen verarbeiten. Dies hat den Vorteil, dass EditParts nur auf sinnvolle Interaktionen reagieren.

5.2.1 Draw2D

Die Anordnung und das Rendering der grafischen Repräsentationen werden durch das *Draw2d*-Paket übernommen, welches Teil von GEF ist und verschiedene Zeichen- und Layoutalgorithmen enthält. Draw2D basiert auf dem SWT von Eclipse. Draw2D soll im Bereich des SWT die Rolle des *Java 2D* innerhalb der AWT/Swing-Bibliothek übernehmen. Draw2D ist Java 2D jedoch in den Gestaltungsmöglichkeiten unterlegen, weil es keine komplexen 2D-Transformationen wie Translation, Rotation, Skalierung oder Scherung ermöglicht. Da Java 2D auschließlich für die Nutzung mit AWT oder Swing

8 etwa: Modell-Darstellungs-Steuerungs-Architektur.
9 Für weiterführende Informationen zur MVC-Architektur vgl. REENSKAUG (Ree03).

entworfen wurde, stehen die genannten 2D-Transformationen bei der Visualisierung nicht zur Verfügung.

5.3 Umsetzung der Merkmalmodellierung

Die Implementierung des Merkmaldiagramms unter Verwendung des GEF benötigt drei verschiedene Modelle, je eines für die Merkmale, Verbindungskanten und Gruppierungen. In diesen Modellen werden alle Informationen gespeichert, die notwendig sind für den Aufbau des Diagramms.

5.3.1 Aufbau des Merkmaldiagramms

Der Diagrammaufbau erfolgt per Mauseingabe im neu integrierten Diagrammeditor (vgl. Abbildung 5.3). Um ein neues Merkmaldiagramm erzeugen zu können wurde der *New*-Wizard von Eclipse erweitert. Am linken Rand des Editors ist eine Menüleiste positioniert, welche die Funktionen für die Erzeugung der Diagramm-Modelle enthält. Soll z. B. eine neues Merkmal erstellt werden, wird die entsprechende Funktion zunächst selektiert. Dann kann das Merkmal im Editor durch einen Mausklick an der Position des Mauszeigers erzeugt werden, da die Mausaktion eine entsprechende Anfrage an den *BasiseditPart* (den Hintergrund) auslöst. Kanten und Gruppierungen werden erzeugt, indem Anfang- und Endmerkmale durch Mausklick festgelegt werden. Mit einem Kontextmenü werden weitere Editierfunktionen, wie „redo", „undo", löschen, sowie Festlegung der Attributierung zur Verfügung gestellt. Der Name eines Merkmals kann durch Editierung der Merkmalrepräsentationen festgelegt werden.

Design Rule-Erzeugung

Aus dem Merkmaldiagramm werden automatisch Design Rules entsprechend der im Entwurf vorgestellten Regeln erzeugt (vgl. Abschnitt 4.4.1). Diese müssen als Dateien im lokalen Dateisystem dort gespeichert werdem, wo später die Implementierungsdateien der Schichten abgelegt werden. Daher wird zusammen mit dem Merkmaldiagramm ein Verzeichnisbaum im Workspace aufgebaut, der die Hierarchie des Diagramms widerspiegelt. Eine erzeugte Design Rule wird in dem Verzeichnis abgespeichert, das dem Merkmal entspricht, zu dem die Design Rule erstellt wurde. Während des Entwurfs werden dann die Implementierungsdateien für dieses Merkmal eben-

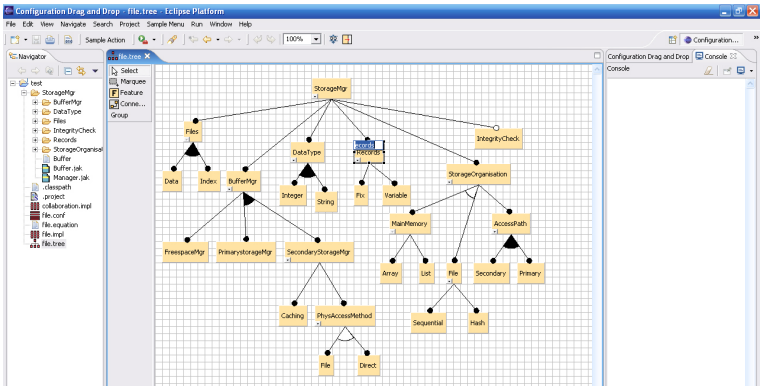


Abbildung 5.3: Originalansicht des neu-integrierten Diagrammeditors für die Erstellung eines Merkmaldiagramms

falls in diesem Verzeichnis gespeichert. Wird ein Merkmal im Diagramm gelöscht, umgesetzt oder umbenannt, werden die äquivalenten Aktionen auf dem zugehörigen Verzeichnis und den enthaltenen Dateien ausgeführt. Hierdurch wird sichergestellt, dass das Merkmaldiagramm und der Verzeichnisbaum immer konsistent sind.

5.3.2 Exploration, Views und Perspektiven

Für die Exploration wurden mehrere Interaktionsmöglichkeiten integriert. Betrachtet wird das Diagramm in Abbildung 5.4.

Zoomen

Abbildung 5.4 zeigt das gleichen Diagramm wie in Abbildung 5.3. Die Ansicht des Baumes ist durch zoomen jedoch um 25% verkleinert dargestellt. Insgesamt stehen acht verschiedene Zoomstufen zur Verfügung. Ebenfalls möglich ist eine automatische Anpassung des Baumes an die Höhe oder Breite des Darstellungsbereiches.

Anpassung der Baumansicht

Die Ausrichtung des Baumes kann manuell, semi-automatisch an einem Raster oder automatisch geschehen. Darüber hinaus ist es möglich Teilbäume aus- und wieder einzublenden. Die Wurzel dieser Teilbäume wird dann grau dargestellt.

Tooltips

Die Tooltips geben momentan Auskuft über Level, Anzahl der Kinder, den „Pfad" des Merkmals innerhalb des Diagramms, sowie über die aktuelle Attributierung. Möglicherweise werden sich in der praktischen Erprobungsphase weitere Informationen als wichtig herausstellen. In der Mitte ist ein Tooltip für das Merkmal *PrimaryStorageMgr* zu sehen.

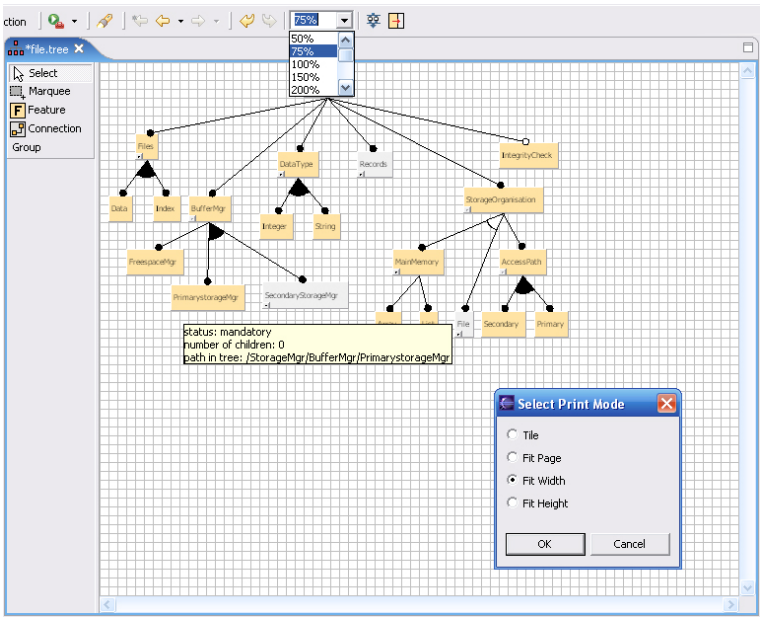


Abbildung 5.4: Um 25% verkleinerte Ansicht des Merkmaldiagramms, mit ausgeblendeten Merkmalen, Drucker-Dialog, Tooltips und Liste mit Zoomwerten.

Drucken

Für die Dokumentation ist es wichtig das Diagramm ausdrucken zu können. Unten rechts in Abbildung 5.4 ist der Druck-Dialog zu sehen. Für das Drucken kann eine von vier Optionen ausgewählt werden: *Tile*, *Fit Page*, *Fit Width* und *Fit Height*. Je nach Auswahl der Optionen wird das Bild vor dem Drucken unterschiedlich skaliert[10].

Detail & Overview

Views sind in Eclipse ein wichtiges Konzept, um weiterführende Informationen über den Editorinhalt darzustellen. Bereits erwähnt wurde die Content-Outline, welche als Navigationsinstrument, sowie zur Übersicht dient. Auch für den Diagrammeditor wurde eine Content-Outline integriert. In Abbildung 5.6 ist die Outline-View links neben dem Editor angeordnet. Die Outline-View stellt das Diagramm im Editor als Thumbnail dar. Das transparente Rechteck markiert den Ausschnitt, der im Editor sichtbar ist. Durch Verschiebung des Rechtecks wird auch die Ansicht im Editor entsprechend angepasst. Die Darstellung des Merkmalbaumes als Thumbnail macht es nicht möglich die Elemente aus der Übersicht mit denen im Editor zu verknüpfen und dadurch Merkmale gezielt zu selektieren. Zu diesem Zweck wurde eine weitere Übersicht integriert, die das Merkmaldiagramm als Verzeichnisbaum darstellt. Diese ist rechts von dem

10 Die Ausrichtung des Merkmalbaumes, sowie das Zoomen und die Druckfunktionalität wurden in einem parallel laufenden Laborpraktikum weiterentwickelt.

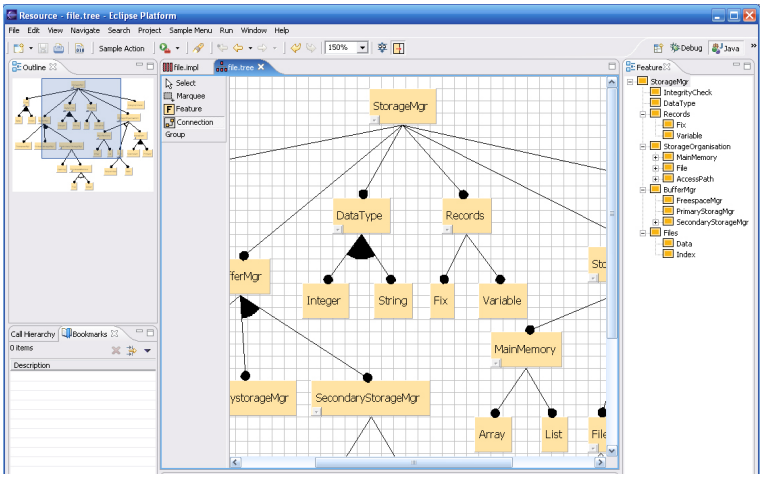


Abbildung 5.5: Outline-View als Übersicht und Navigationsinstrument der Hauptansicht.

Editor angeordnet. Für die automatische Anordnung des Editors und beider Views wurde eine neue Perspektive definiert.

5.4 Umsetzung der Entwurfsansicht

Für die Entwurfsansicht sind lediglich zwei Modelle notwendig. Eines für die Schichten und eines für die Dateirepräsentationen, die sich auf den Schichten befinden. Für die Entwurfsphase wurde ebenfalls ein neuer Editor in Eclipse integriert. Am linken Rand der Abbildung ist die Outline-View angeordnet.

Schichtenerzeugung

Wie bereits im vorherigen Kapitel erläutert wurde, darf eine Schicht nur dann erzeugt werden, wenn sie einem Merkmal der Analyse zugeordnet werden kann. Um dies sicherzustellen werden Schichten nicht durch direkte interaktive Platzierung mit der Maus in der Entwurfsansicht erzeugt, sondern auf der Basis des Merkmaldiagramms. Dieser Vorgang geschieht per Drag & Drop. Zu implementierende Merkmale werden im Diagramm selektiert und können dann in eine View gezogen werden. In der View wird in reduzierter Form die aktuelle Entwurfsansicht dargestellt. Vor der Erzeugung einer Schicht wird überprüft, ob das Merkmal bereits einer Schicht zugeordnet wurde. Ist dies nicht der Fall wird eine Schicht in der View und in der Entwurfsansicht erzeugt. Um die spätere Zuordnung einfach zu gestalten, haben korrespondierende Merkmale und Schichten den gleichen Namen. Der beschriebene Drag & Drop-Vorgang ist auch direkt von der Analyseansicht hin zur Entwurfsansicht möglich. Der „Umweg" über ei-

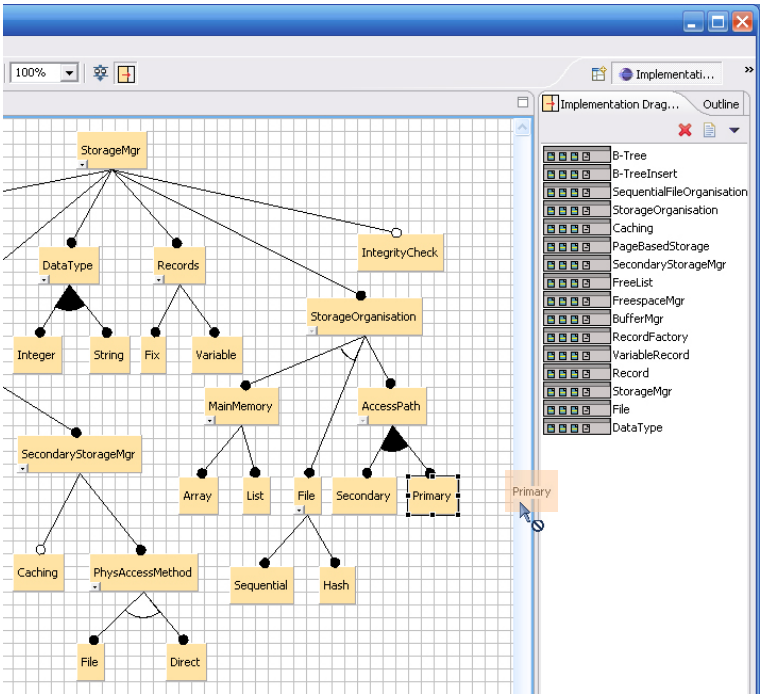


Abbildung 5.6: Drag & Drop-Vorgang für die Erzeugung der Entwurfsansicht.

ne View geschieht aus mehreren Gründen. Um dem Nutzer die Anordnung der Quell– und Zielfenster zu ersparen wurde eine *Drag & Drop-Perspektive* definiert, mit der die benötigten Fenster automatisch in geeigneter Weise angeordnet werden. Bei der Definition einer neuen Perspektive kann jedoch nur ein Editor festgelegt werden, um den beliebig viele Views angeordnet sind, so dass die automatische Anordnung zweier Editoren nicht möglich ist. Hinzu kommt, dass der Inhalt von zwei Editoren, die nebeneinander angeordnet sind, nur in Ausnahmefällen vollständig dargestellt werden kann. Da bei dem Drag & Drop-Vorgang vor allem die Details des Merkmaldiagramms notwendig sind ist die reduzierte Ansicht der Schichtendarstellung in einer View ausreichend.

Schichtenanordnung

Abbildung 5.6 zeigt den Drag & Drop-Vorgang und Abbildung 5.7 die daraus entstandene Entwurfsansicht. Die minimierte Schicht und die selektierte Schicht schließen sich gegenseitig aus. Diese Information wurde aus den Design Rules gewonnen, die automatisch während der Analyse erzeugt wurden, ebenso wie die Reihenfolge der Schichten. Auf der Basis dieses automatischen „Vorschlags" kann die Entwurfsansicht dann weiter verfeinert werden. Informationen über sich ausschließende Schichten, die noch nicht in diesen Design Rules festgehalten wurden, können durch Selektion

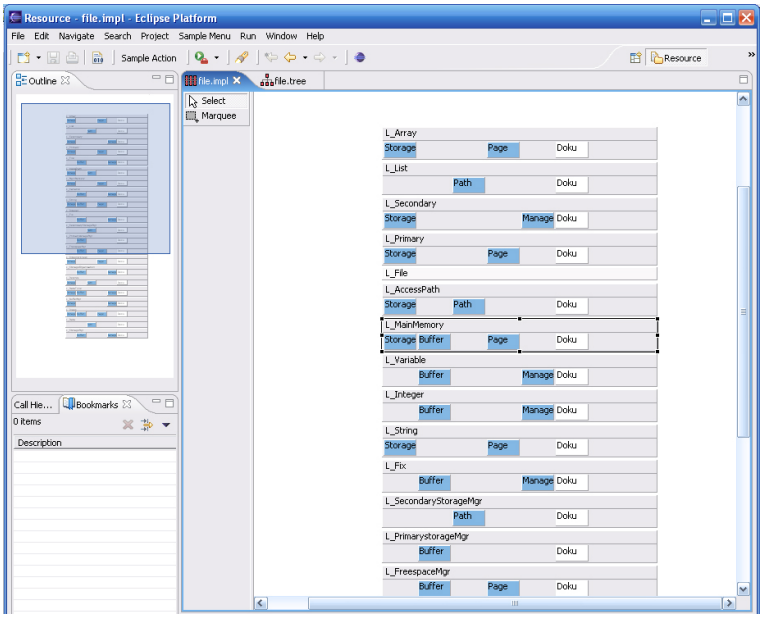


Abbildung 5.7: Entwurfsansicht

der entsprechenden Kollaborationen gekennzeichnet werden. Das gleiche gilt für austauschbare Kollaborationen. Um die Reihenfolge der Schichten weiter anzupassen, können sie durch Auswahl der entsprechenden Pfeiltasten auf der Tastatur nach oben und unten verschoben werden. Die Design Rules können durch Aufrufen eines entsprechenden Formulars weiter verfeinert werden (vgl. Abbildung 5.8).

5.4.1 Erzeugung und Anordnung von Dateirepräsentationen und Implementierungsdateien

Die Erzeugung von Dateirepräsentationen und die der Implementierungsdateien ist miteinander gekoppelt. Nach Auswahl einer oder mehrerer Schichten kann eine Funktion aufgerufen werden, die zunächst einen Dialog öffnet, in dem der Name der zu erzeugenden Datei eingegeben wird. Aufgrund der Verfeinerungen werden häufig gleiche Dateinnamen genutzt. Daher enthält der Dialog eine Liste mit den Namen der Dateien, die bereits erzeugt wurden. Die Adresse der erzeugten Implementierungsdateien wird automatisch festgelegt (vgl. Abschnitt 5.3.1). Die gleichzeitige Erstellung von grafischer Repräsentation und lokaler Datei gewährleistet, dass die Entwurfsansicht während der Implementierung als Navigationsinstrument genutzt werden kann.

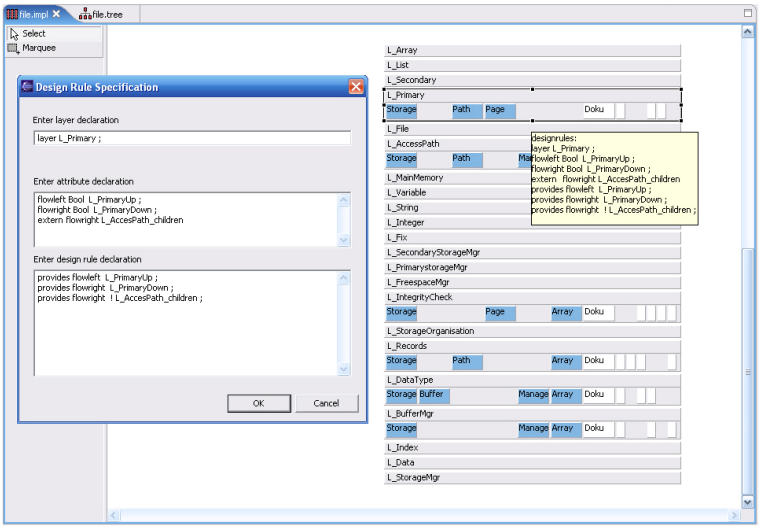


Abbildung 5.8: Outline-View als Übersicht und Navigationsinstrument der Hauptansicht.

Die Anordnung neu erzeugter Dateirepräsentationen auf ihren Schichten erfolgt ebenfalls automtisch. Zunächst wird untersucht, ob ein Dateiname bereits auf einer anderen Schicht existiert. Ist dies der Fall, wird die neue Datei an der gleichen horizontalen Position auf der Schicht platziert. Andernfalls wird sie am rechten Rand der Schicht angeordnet, die gegebenenfalls vergrößert wird. Damit alle Schichten die gleiche Länge haben wirkt sich solch eine Vergrößerung immer auf alle Schichten aus.

5.4.2 Exploration, Views und Perspektiven

Für die weitere Exploration wurden im Wesentlichen die gleichen Funktionalitäten und Views (mit eigener Perspektive) wie für den Analyseeditor implementiert. Der Anwender kann zoomen, die Ansicht drucken, sowie über Tooltips weitere Informationen anzeigen lassen. Die Tooltips der Schichten zeigen ihre Design Rules an, die Tooltips der Dateirepräsentationen den Pfad der korrespondierenden Datei innerhalb des Eclipse-Workspaces.
Um die Komplexität der Ansicht anzupassen, gibt es die im Entwurf vorgestellte Möglichkeit Schichten und Dateien zu minimieren oder zu expandieren. Hierfür können auch verschiedenen Filterkriterien benannt werden.

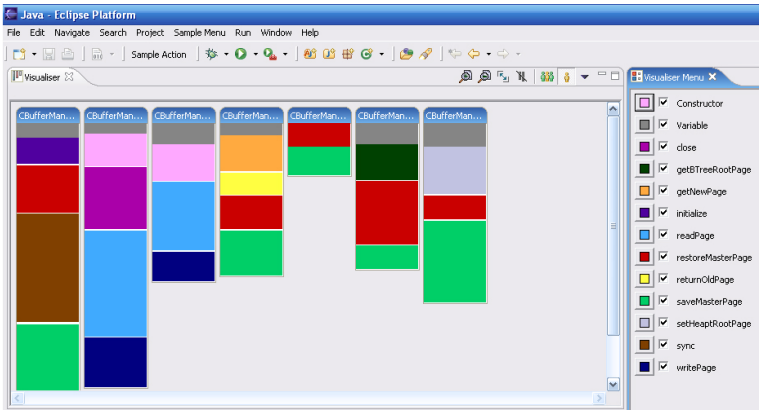


Abbildung 5.9: Visualisierung der Verfeinerungshierarchie auf Codeebene.

5.5 Codevisualisierung

Auch für die Seesoft-basierte Datenvisualisierung wurde bereits ein Eclipse-Plugin entwickelt. Die im Entwurf vorgestellte und an Seesoft angelehnte Visualisierung wurde mithilfe des *Visualizer Plugins* umgesetzt. Dieses Plugin wurde ursprünglich entwickelt, um die aspektorientierte Programmierung zu unterstützten, indem es Aspekte im Code visualisiert. Dann wurde es so erweitert, dass beliebige Daten abgebildet werden können.
Abbildung 5.9 zeigt die Visualisierung mehrerer Klassenfragmente einer Implementierung, die merkmalsorientiert entworfen wurden. Das Plugin besteht aus zwei separaten Views. Eine View enthält die eigentliche Visualisierung, die zweite View ein Funktionsmenü. Dieses Funktionsmenü listet alle Klassen von Daten auf, die als Streifen in unterschiedlichen Farben dargestellt werden sollen. In diesem Fall sind es der Konstruktor und verschiedene Funktionen, die verfeinert werden. Weiterhin ermöglicht das Menü die Farbauswahl für die Streifen, sowie das Ein- und Ausblenden bestimmter Datenklassen. Diese Funktionalität ist wichtig, da die Darstellung bei sehr vielen unterschiedlichen Funkionen sonst durch die Vielzahl verschiedener Farben unübersichtlich wird. Die Streifen können per Mauseingabe selektiert werden. Hierdurch öffnet sich die Quellcodedatei an der Position, die durch den Streifen repräsentiert wird.
Um das Visualiser Plugin zu erweitern stellt es einen Erweiterungspunkt zur Verfügung, der die Implementierung zweier Schnittstellen erfordert:

- `IContentProvider`: Die Implementierung dieser Klasse bestimmt die Anzahl und Größe der Balken, die dargestellt werden.
- `IMarkupProvider`: Die Implementierung dieser Klasse legt fest, welche Streifen, in welcher Reihenfolge und Größe auf den Balken plaziert werden.

Da der Visualizer nur die Klassenfragmente einer Klasse anzeigt, muss die gewünschte Klasse zunächst durch den Anwender ausgewählt werden. Dies geschieht durch Auswahl einer Jak-Datei. Dann ermittelt die Implementierung der Klasse `IContentProvider` die Anzahl aller weiteren Klassenfragmente und zeichnet die entsprechende Anzahl von Balken. Die Länge der Balken wird in Abhängigkeit von der Dateigröße der jeweiligen Implementierungsdatei ermittelt.
Die Ermittlung der Streifen bzw. der Anzahl unterschiedlicher Codefragmente innerhalb der Implementierungsdatei erfodert, dass der Quellcode geparst wird, da die einzelnen Code-Fragmente herausgearbeitet werden müssen. Für jede neue globale Funktion muss ein Streifentyp angelegt werden, ebenso für alle Konstruktoren. Variablen und lokale Funktionen werden dagegen jeweils zu einem Streifentyp zusammengefasst, da sie nicht verfeinert werden. Die Breite und Position des Streifens ist abhängig von der Länge bzw. Position des Codefragments. Das Parsen des Jak-Quellcodes wurde noch nicht umgesetzt. Der entwickelte Protoyp kann jedoch aus einer Datei, in der die notwendigen Informationen zusammengefasst sind eine Visualisierung erzeugen.

5.6 Konfigurierung

Der Anwender soll eine Konfiguration auf Grundlage des Merkmalbaumes erstellen können. Daher erfolgt die Konfigurierung analog zur Erzeugung der Entwurfsansicht per Drag & Drop. Der Anwender selektiert die gewünschten Merkmale und zieht sie in eine View, in der dann die entsprechenden Schichten erzeugt werden. Dies geschieht nur, wenn zu dem selektierten Merkmal eine Implementierung vorhanden ist. Anhand der Design Rules werden die Schichten dann automatisch in die korrekte Reihenfolge gebracht. Um diesen Vorgang zu unterstützten, wurden die im Entwurf erarbeiteten Mechanismen integriert. Abbildung 5.10 zeigt ihre Umsetzung. Die Merkmale *SecondaryStorageMgr*, *Primary* und *String* wurden selektiert. Die grün eingefärbten Merkmale müssen zusammen mit diesen drei Merkmalen Teil der Konfiguration sein. Ausgeschlossene Merkmale sind hellgrau dargestellt. Aus der Konfigurationsansicht heraus kann für die Konfigurationen ein *Equation*-Datei erzeugt werden. Vorher wird

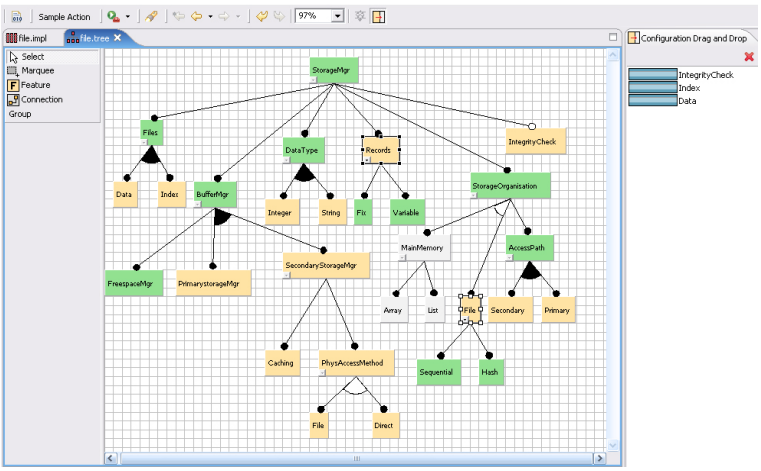


Abbildung 5.10: Ansicht des Merkmaldiagramms während der Konfiguration.

noch einmal anhand der Design Rules überprüft, ob eine gültige Konfiguration ausgewählt wurde. Andernfalls wird eine Fehlermeldung ausgegeben. Diese Konfigurierungssicht wurde eingeführt, um auch Zwischenstände abspeichern zu können oder zusätzliche Annotationen zu ermöglichen. Abbildung 5.11 zeigt die Konfigurationsansicht. Am rechten Rand ist eine automatisch erzeugte Equation-Datei abgebildet. Mithilfe des geöffneten Dialogs können die gewünschten AHEAD-Werkzeuge aufgerufen werden.

5.7 Zusammenfassung

In diesem Kapitel wurde die Umsetzung der im Entwurf entwickelten Lösungen erläutert. Die Unterstützung der Merkmalorientierten Softwareentwicklung wurde als Plugin in die *Eclipse Java-IDE* integriert. Die Java-IDE von Eclipse wurde als Basis genutzt, da sie bereits eine große Anzahl wichtiger Funktionalitäten für die Programmierung zur Verfügung stellt. Es wurden vier neue Editoren integriert, ein Jak-Editor und je ein grafischer Diagrammeditor für eine der Entwicklungsphasen. Der Aufbau eines Merkmaldiagramms erfolgt interaktiv. Um den Umgang mit großen Bäumen zu erleichtern wurden verschiedene Techniken genutzt (Zoom, Detail & Overview, Navigational Slaving, Ein- und Ausblenden von Teilbäumen, sowie Tooltips und (semi-) automatische Ausrichtung des Diagramms). Während des Aufbaus des Merkmaldiagramms wird ein Verzeichnisbaum aufgebaut, der die Diagrammhierarchie widerspiegelt. In diesem

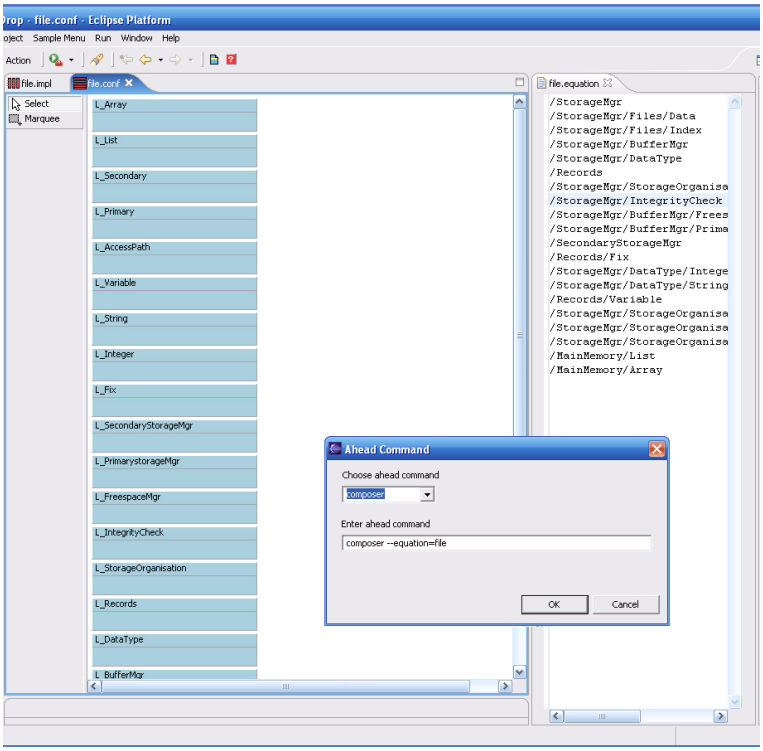


Abbildung 5.11: Konfigurationsansicht.

Verzeichnisbaum werden Design Rules und Implementierungsdateien automatisch abgelegt. Design Rules können für einen Merkmalbaum automatisch erzeugt werden.
Die Erzeugung der Entwurfsansicht geschieht auf der Basis des Merkmaldiagramms durch einen Drag & Drop-Vorgang. Hierdurch wird sichergestellt, dass Schichten des Entwurfs immer einem Merkmal der Analyse zugeordnet werden. Anhand der Design Rules, die während der Analyse erzeugt wurden, wird eine erste Anordnung der Schichten vorgenommen. Diese kann dann durch den Anwender weiter verfeinert werden. Die Dateirepräsentationen auf den Schichten werden zusammen mit den repräsentierten Dateien erzeugt. Die Dateien werden automatisch im erzeugten Verzeichnisbaum abgelegt. Um die Verfeinerungshierarchie auf Quellcodeebene darzustellen wurde mithilfe des Visualizer Plugins eine Seesoft-ähnliche Visualisierung integriert. Der Konfigurationsprozess wird durch Auswahl der gewünschten Merkmale im Merkmaldiagramm durchgeführt. Die Anordnung der Schichten und die Erzeugung der Equation-Dateien erfolgt automatisch durch Auswertung der Design Rules.

KAPITEL 6

Zusammenfassung und Ausblick

Ziel dieser Arbeit war der Entwurf eines Werkzeugs, das die Entwicklung von Programmfamilien und Produktlinien unterstützt. Die Basis für diese Entwicklung sollten der dreistufige Prozess (Analyse, Entwurf und Implementierung) des Domain Engineerings, sowie die Merkmalorientierte Programmierung bilden. Nach einer Einleitung und Motivation der Notwendigkeit für die Werkzeugunterstützung des Programmfamilien-Engineerings, wurden verschiedene Techniken und Verfahren zur Umsetzung von Programmfamilien und Produktlinien vorgestellt. Dabei lag der Fokus auf den für diese Arbeit wichtigen Methoden: der Merkmalorientierten Domänenanalyse, dem Kollaborationsentwurf und der Merkmalorientierten Programmierung. In dem darauf folgenden Kapitel wurden Visualisierungs- und Interaktionstechniken im Allgemeinen und die Visualisierung hierarchischer Daten, im Speziellen präsentiert. Des Weiteren wurde der Bereich der Softwarevisualisierung kurz zusammengefasst.
Kapitel 4 präsentierte den Entwurf des Werkzeugs. Hierzu wurde zunächst die AHEAD-Toolsuite vorgestellt, die als Werkzeug für die Merkmalorientierte Programmierung integriert werden sollte. Ausgehend von den Teilprozessen wurden dann sukzessiv die Schwächen des vorhandenen Werkzeuges aufgezeigt. Die fehlende Integration des Gesamtprozesses und die daraus entstehenden Mehrfacharbeiten sind die Hauptursache für entstehende Inkonsistenzen. Auf Grund der Komplexität des Entwicklungsprozess von Produktlinien und Programmfamilien stand eine geeinigte Automatisierung von Teilprozessen und eine optimierte Anwenderinteraktion und Visualisierung im Vordergrund dieser Arbeit. Für eine automatisierte Propagierung von Wissen wurden eine Reihe von Heuristiken entwickelt, die eine Abbildung zwischen den Abstraktionseinheiten und ihren Interaktionen von einer Phase des Domain Engineerings auf nach-

folgende Phasen ermöglichen sollen. Die Erzeugung der Design Rules wurde auf alle Phasen des Domain Engineerings erweitert. Daher kann die automatische Wissensübertragung nun auf der Basis von Design Rules erfolgen. Um die Erstellung von Design Rules benutzerfreundlicher zu gestalten wurden Regeln für die automatische Generierung von Design Rules aufgestellt. Hierbei ist jedoch nur ein Ausschnitt der möglichen Merkmalinteraktionen betrachtet worden.
Kapitel 5 erläuterte die Umsetzung der im Entwurf vorgestellten Lösungen. Die Umsetzung erfolgte als Plugin in die Eclipse Java-IDE. Zu diesem Zweck wurden eine Reihe grafischer Editoren und Views in die Eclipse-Workbench neu-integriert. Insbesondere wurde eine Seesoft-ähnliche Visualisierung eingefügt, mit deren Hilfe die Verfeinerungsstruktur auf Quellcodeebene dargestellt werden kann. Der grafische Editor für die Erstellung eines Merkmaldiagramms ermöglicht einen interaktiven Aufbau der Analyseansicht. Die Entwurfsansicht wird dagegen auf der Grundlage des Merkmaldiagramms aufgebaut und zunächst automatisch erzeugt. Bei diesem Vorgang wird auf der Basis der entwickelten Heuristiken automatisch ein „Vorschlag" für den Entwurf präsentiert, der dann weiter verfeinert werden kann. In ähnlicher Weise wird der Prozess der Konfigurierung umgesetzt. Eine Konfiguration wird ebenfalls auf der Basis des Merkmaldiagramms erstellt. Bei diesem Vorgang wird der Anwender durch eine Reihe unterschiedlicher Mechanismen unterstützt, um eine korrekte Konfiguration aus der Vielzahl der verschiedenen Merkmale erstellen zu können.

Ausblick

Die wichtigste noch ausstehende Aufgabe besteht in der Durchführung einer Nutzerstudie, die folgende Fragen beantworten sollte:

- Sind die entwickelten Abbildungsregeln und Heuristiken in der Praxis tragfähig?
- Sind die integrierten Visualisierung- und Interaktionsmechanismen ausreichend? Wenn nicht:
- Welche Funktionalitäten können noch integriert werden, um die Benutzerfreundlichkeit zu steigern?

Eine wichtige zusätliche Erweiterung wäre die Entwicklung einer Debugging-Funktion für die Sprache *Jak*. Weiterhin sollte die automatische Erstellung von Design Rules auf den Entwurf erweitert werden. Im Hinblick auf die Codevisualisierung muss das Parsen des Quellcodes noch umgesetzt werden.
Mit *FeatureC++* haben APEL et al. (ALRS05) eine Erweiterung der Sprache C++ entwickelt, die ebenfalls Merkmalorientierte Programmierung unterstützt. Um auch homogene Crosscutting Concerns kapseln zu können, wurden in FeatureC++ darüber

hinaus auch Konzepte der Aspektorientierten Programmmierung integriert. Denkbar ist eine Erweiterung der vorgestellten IDE um die Unterstützung von FeatureC++. Hierbei müssten die besonderen Anforderung, die sich aus der Integration der Aspektorientierten Programmierung ergeben, berücksichtigt werden.

Literaturverzeichnis

[ALRS05] Apel, S.; Leich, T.; Rosenmüller, M.; Saake, G.: FeatureC++: Feature-Oriented and Aspect-Oriented Programming in C++. Technischer Bericht, Department of Computer Science, Otto-von-Guericke University, Magdeburg, 2005.

[Bai93] Bailin, S.: Domain Analysis with KAPTUR. In *Tutorials of TRI-Ada'93*, 1993.

[Bat04] Batory, D.: *A Tutorial on Feature Oriented Programming and the AHEAD Tool Suite (ATS)*. Department of Computer Sciences University of Texas at Austin, Austin, Texas, 78712 U.S.A., September 2004.

[BC90] Bracha, G.; Cook, W.: Mixin-based inheritance. In *Conference on Object Oriented Programming Systems Languages and Applications*, S. 303 – 311. ACM Press, 1990.

[BCS99] Batory, D.; Cardone, R.; Smaragdakis, Y.: Object-Oriented Frameworks and Product-Lines. In *1st Software Product-Line Conference*. Denver, Colorado, August 1999.

[Ber83] Bertin, J.: *Semiology of Graphics: Diagrams Networks Maps.* The University of Wisconsin Press, Madison, WI, USA., 1983.

[BG97] Batory, D.; Geraci, B.: Composition Validation and Subjectivity in GenVoca Generators. In *IEEE Transactions on Software Engineering*, S. 67–82, Februar 1997.

[BLHM02] Batory, D.; Lopez-Herrejon, R. E.; Martin, J.-P.: Generating Product-Lines of Product-Families. In *17 th IEEE International Conference on Automated Software Engineering (ASE'02)*, 2002.

[BO92] Batory, D.; O'Malley, S.: The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, Band 1, S. 355 – 398, 1992.

[BPSP03] Beuche, D.; Papajewski, H.; Schröder-Preikschat, W.: Variability Management with Feature Models. In *Proceedings of the Workshop on Software Variability Management*, S. p. 72–83, 2003.

[BSR03] Batory, D.; Sarvela, J. N.; Rauschmayer, A.: Scaling Step-Wise Refinement. In *ICSE 2003*, 2003.

[CE00] Czarnecki, K.; Eisenecker, U. W.: *Generative Programming Methods, Tools and Applications*. Addison-Wesley, 2000.

[CLN02] Clements, P.; Linda Northrop, L.: *Software Product Lines: Practices and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2001, 2002.

[CRB04] Colyer, A.; Rashid, A.; Blair, G.: On the Separation of Concerns in Program Families. Technischer Bericht Nr. COMP-001-2004, Computing Department, Lancaster University, 2004.

[Cza98] Czarnecki, K.: *Generative Programming. Principles and Techniques of Software Engineering Based on Automated Configuration and Fragment-Based Component Models*. Dissertation, Technische Universität Ilmenau, Oktober 1998.

[Dij76] Dijkstra, E.: *A Discipline of Programming*. Prentice-Hall, 1976.

[ESS92] Eick, S.; Steffen, J.; Summer, E.: Seesoft – A Tool For Visualizing Line Oriented Software Statistics. In *IEEE Transactions on Software Engineering*, S. 957–968, 1992.

[FPDF98] Frakes, W.; Prieto-Diaz, R.; Fox, C.: DARE: Domain Analysis and Reuse Environment. *Annals of Software Engineering*, Band 5, S. 125–151, 1998.

[Gib97] Gibson, J. P.: Feature Requirements Models: Understanding Interactions. In *Feature Interactions In Telecommunications IV*. IOS Press, Montreal, Canada, Juni 1997.

[Hir02] Hirschfeld, R.: AspectS – Aspect-Oriented Programming with Squeak. In *Proceedings of Net.ObjectDays (NODe)*, S. 219–235, 2002.

[IEE00] IEEE: IEEE Recommended Practice for Architectural Description of Software-Intensive Systems. The Institute of Electrical and Electronics Engineers, Inc., September 2000.

[JF88] Johnson, R. E.; Foote, B.: Designing Reusable Classes. *Journal of Object-Oriented Programming*, Band 1, Nr. 2, S. 22–35, 1988.

[JS91] Johnson, B.; Shneiderman, B.: Treemaps: A Space-filling Approach to the Visualization of Hierarchical Information Structures. In *Proceedings of IEEE Visualization '91*, S. 284–291. IEEE Press, Los Alamitos, CA, USA, October 1991.

[Kan03] Kandé, M. M.: *A Concern-Oriented Approach to Software Archtitecure*. Dissertation, Technische Universität Berlin, 2003.

[KCH+90] Kang, K. C.; Cohen, S. G.; Hess, J. A.; Novak, W. E.; Peterson, A.: Feature-Oriented Domain Analysis (FODA) Feasibility Study. Technischer Bericht Nr. CMU/SEI-90-TR-21 ESD-90-TR-222, Software Engineering Institute Carnegie Mellon University, Pittsburgh, Pennsylvania 15213, USA, November 1990.

[KLM+97] Kiczales, G.; Lamping, J.; Mendhekar, A.; Maeda, C.; Lopes, C. V.; Loingtier, J.-M.; Irwin, J.: Aspect-Oriented Programming. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*. Springer-Verlag LNCS 1241, Finland, Juni 1997.

[Kob04] Kobsa, A.: User Experiments with Tree Visualization Systems. In *IEEE Symposium on Information Visualization 2004*, S. 9 – 16, Oktober 2004.

[LAS05] Leich, T.; Apel, S.; Saake, G.: Using Step-Wise Refinement to Build a Flexible Lightweight Storage Manager. In *Proceedings of the 9th East-European Conference on Advances in Databases and Information Systems (ADBIS'05)*, September 12-15. Lecture Notes in Computer Science (LNCS), Springer, 2005.

[LK98] Lopes, C. V.; Kiczales, G.: Recent Developments in AspectJ. In *ECOOP'98 Workshop Reader (Aspect-Oriented Programming Workshop)*. Springer-Verlag LNCS 1543, 1998.

[LKL02] Lee, K.; Kang, K. C.; Lee, J.: Concepts and Guidelines of Feature Modeling for Product Line Software Engineering. In Gacek, C. (Hrsg.): *Software Reuse: Methods, Techniques, and Tools: Proceedings of the Seventh Reuse Conference (ICSR7)*, S. 62–77, April 2002.

[LRP95] Lamping, J.; Rao, R.; Pirolli, P.: A Focus+Context Technique Based on Hyperbolic Geometry for Visualizing Large Hierarchies. In Katz, I. R.; Mack, R.; Marks, L. (Hrsg.): *Proceedings of the Conference on Human Factors in Computing Systems, CHI'95*, S. 401–408. ACM Press, New York, NY, USA, Mai 1995.

[Mac88] Mackinlay, J.: Applying a Theory of Graphical Presentation to the Graphic Design of User Interfaces. In *Proceedings of the 1st annual ACM SIGGRAPH symposium on User Interface Software table of contents*, 1988.

[McI68] McIlroy, M. D.: Mass-Produced Software Components. In Naur, P.; Randell, B. (Hrsg.): *Software Engineering Concepts and Techniques*, S. 88–98. North Atlantic Treaty Organisation (NATO) Conference on SoftwareEngineering, Garmisch-Partenkirchen, Oktober 1968.

[MRC91] Macklinay, J. D.; Robertson, G. G.; Card, S. K.: The Perspective Wall: Detail and Context Smoothly Integrated. In Robertson, S. P.; Olson, G. M.; Olson, J. S. (Hrsg.): *Proceedings of the Conference on Human Factors in Computing Systems, CHI'9 ACM Press.*, S. 173–179. ACM Press, New York, NY, USA, 1991.

[Mun02] Munzner, T.: Information Visualization. In *IEEE Computer Graphics and Applications Special Issue on Information Visualization*, S. 20–21, 2002.

[Nei80] Neighbors, J. M.: *Software Construction Using Components*. Dissertation, Department of Information and Computer Science, University of California, 1980.

[Neu04] Neumann, P.: Focus+Context Visualization of Relations in Hierarchical Data. Master's thesis, Otto-von-Guericke-Universität Magdeburg,Fakultät

für Informatik, 2004.

[NS73] Nassi, I.; Shneiderman, B.: Flowchart Technique for Structured Programming. In *ACM SIGPLAN Notices*, S. 12–26, 1973.

[OT00] Ossher, H.; Tarr, P.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In *In Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.

[Par76] Parnas, D. L.: On the Design and Development of Program Families. *IEEE Transactions On Software Engineering*, Band 1, S. 1–9, March 1976.

[PBS93] Price, B.; Baecker, R.; Smal, I.: A Principled Taxonomy of Software Visualization. *Journal of Visual Languages and Computing*, Band 4, Nr. 3, S. 211–266, 1993.

[PD93] Prieto-Diaz, R.: Status Report: Software Reusability. In *IEEE Software*, S. 61 – 66. IEEE Computer Society Press, 1993.

[Pre97] Prehofer, C.: Feature Oriented Programming. A Fresh Look at Objects. In Aksit, M.; Matsuoka, S. (Hrsg.): *ECOOP '97 - Object-Oriented Programming*, S. 419–443. Springer-Verlag, June 1997.

[Rao96] Raos, R.: Information Visualization and the Next Generation Workspace, Januar 1996. Xerox Palo Alto Research Center.

[Ree03] Reenskaug, T.: The Model-View-Controller (MVC). Its Past and Present. University of Oslo, 2003.

[RMC91] Robertson, G. G.; Mackinlay, J. D.; Card, S. K.: Cone Trees: Animated 3D Visualizations of Hierarchical Information. In *Proceedings of the Conference on Human Factors in Computing Systems, CHI'91*, S. 189–194. ACM Press, New York, NY, USA, 1991.

[SB98] Smaragdakis, Y.; Batory, D.: Implementing Layered Designs with Mixin Layers. In Jul, E. (Hrsg.): *Proceedings of ECOOP'98, LNCS 1445*, S. 550–570. Springer-Verlag Heidelberg Berlin, 1998.

[SCK+96] Simos, M.; Creps, D.; Klinger, C.; Levine, L.; Allemang, D.: Organization Domain Modeling (ODM) Guidebook, Version 2.0. Technischer Bericht, Technical Report for STARS, STARS-VC-A025/001/00, Juni 1996.

[SGSP02] Spinczyk, O.; Gal, A.; Schröder-Preikschat, W.: AspectC++: An Aspect-Oriented Extension to C++. In *Proceedings of the 40th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS Pacific 2002)*, S. 18–21, 2002.

[Shn96] Shneiderman, B.: The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations. In *Proceedings of the 1996 IEEE Symposium on Visual Languages*, S. 336, 1996.

[Sma99] Smaragdakis, I.: *Implementing Large-Scale Object-Oriented Components*. Dissertation, The University of Texas at Austin, Dezember 1999.

[SMSR02] Stanley M. Sutton, J.; Rouvellou, I.: Modeling of software concerns in Cosmos. In *Proceedings of the 1st international conference on Aspect-oriented software developmen*, S. 127 – 133, 2002.

[SPR04] Sochos, P.; Philippow, I.; Riebisch, M.: Feature-Oriented Development of Software Product Lines: Mapping Feature Models to the Architecture. In Weske, M.; Liggesmeyer, P. (Hrsg.): *Object-Oriented and Internet-Based Technologies*, S. 138 – 152. Springer, LNCS 3263, 2004.

[SvGB01] Svahnberg, M.; Gurp, J. v.; Bosch, J.: On the Notion of Variability in Software Product Lines. In *Proceedings Working IEEE/IFIP Conference on Software Architecture (WICSA 2001)*, August 2001.

[Szy98] Szyperski, C.: *Component Software, Beyond Object-Oriented Programming*. Addison-Wesley, 1998.

[TO00] Tarr, P.; Ossher, H.: Multi-Dimensional Separation of Concerns and The Hyperspace Approach. In Kluwer (Hrsg.): *Proceedings of the Symposium on Software Architectures and Component Technology: The State of the Art in Software Development*, 2000.

[VN96a] VanHilst, M.; Notkin, D.: Using C++ Templates to Implement Role-Based Designs. In *Lecture Notes In Computer Science. Proceedings of the Second*

JSSST International Symposium on Object Technologies for Advanced Software, S. 22 – 37. Springer-Verlag, 1996.

[VN96b] VanHilst, M.; Notkin, D.: Using Role Components to Implement Collaboration-Based Designs. In *OOPSLA '96 CA, USA*. ACM Press, 1996.

[War00] Ware, C.: *Information Visualization. Perception for design.* Morgan Kaufmann Publishers, San Francisco, 2000.

[Zeh04] Zehler, T.: Software-Visualisierung, März 2004. Virtuelles Software Engineering Kompetenzzentrum.

Selbständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Arbeit selbständig und nur mit erlaubten Hilfsmitteln angefertigt habe.

Magdeburg, 8. September 2005

Laura Marnitz