Diplomarbeit

Aspektorientiertes Entwerfen mit “Aspectual Collaborations” – Entwicklung eines grafischen Editors für die repository-basierte Entwicklungsumgebung PIROL

Florian Hacker
August 2002

Technische Universität Berlin
Fakultät IV – Elektrotechnik und Informatik
Institut für Softwaretechnik und Theoretische Informatik
Fachgebiet Softwaretechnik
Prof. Dr.-Ing. Stefan Jähnichen

Erklärung:

Die selbständige und eigenhändige Anfertigung versichere ich an Eides statt.

Berlin, den 22. August 2002

____

Unterschrift

Inhaltsverzeichnis

Einleitung **1**
Thema dieser Arbeit 1
Struktur dieser Arbeit 1

1 Aspektorientierte Programmierung **3**
1.1 Motivation 3
1.2 Separation of Concerns 3
1.2.1 Crosscutting Concerns 4
1.3 Aspektorientierung 5
1.4 Hyper/J 5
1.4.1 Hyperspaces 6
1.4.2 Hyper/J 7
1.5 AspectJ 11
1.6 Composition Filters 13
1.7 Aspectual Collaborations 17

2 Aspektorientiertes Entwerfen **21**
2.1 Softwareentwurf 21
2.1.1 Umfeld 22
2.1.2 Ziele 26
2.1.3 Konzepte 29
2.1.4 Entwurfsqualität 31
2.2 Aspekte im Entwurf 38
2.2.1 Warum Aspekte in den Entwurf aufnehmen? 38
2.2.2 Aspekt-Arten im Entwurf 39
2.3 Aspectual Collaborations im Entwurf 41
2.3.1 Entwurfs-Szenarien 42
2.4 UML für Aspekte: UFA 43

3 PIROL **47**
3.1 Softwareentwicklungsumgebungen 47

3.1.1 Charakteristika 47
3.1.2 Integration 48
3.2 PIROL 50
3.2.1 Integration 50
3.2.2 Besondere Eigenschaften 54
3.2.3 Einbindung neuer Werkzeuge 55

4 UFA-Editor **57**
4.1 Anforderungsanalyse 57
4.1.1 Systembeschreibung 57
4.2 Erweiterung des PIROL Metamodells 58
4.2.1 Basisentitäten 59
4.2.2 Metamodell-Entwurf 59
4.2.3 Instanziierung des neuen Metamodells 62
4.3 GEF: Graph Editing Framework 62
4.3.1 Funktionsweise 64
4.3.2 Architektur 64
4.3.3 GraphModel und DefaultGraphModel 67
4.4 Erstellung des UFA-Editors mit Hilfe von GEF 68
4.4.1 UFA-Diagramm Konnektor 71
4.4.2 Repräsentation auf GEF-Seite 71
4.4.3 Anbindung von GEF an PIROL 74
4.4.4 Mediatoren 74
4.5 Beschreibung des UFA-Editors UFAEd 80
4.5.1 Werkzeugleiste 80
4.5.2 Kontextmenü 82
4.5.3 Details 83
4.6 Fazit zum Einsatz von GEF 84
4.6.1 Positives 84
4.6.2 Kritikpunkte 84
4.6.3 Fazit 85

5 Zusammenfassung und Ausblick **87**
5.1 Zusammenfassung 87
5.2 Ausblick 87

A Literaturverzeichnis **89**

B Anhang **95**
B.1 QMOOD Definitionen 95

Einleitung

Thema dieser Arbeit

Mit der aspektorientierten Programmierung wurde ein neues Paradigma gefunden, welches häufig wiederkehrende Probleme der derzeit vorherrschenden objektorientierten Entwicklung beheben soll: einige Belange (concerns) lassen sich in objektorientierten Zerlegungen nicht isolieren, vielmehr scheinen sie orthogonal zur objektorientierten Dekompositionsstruktur zu liegen – mit der Folge, dass solche Belange jeweils über viele Klassen verstreut realisiert werden müssen (scattering) und sich zudem untereinander, sowie mit den Kernaufgaben der jeweiligen Klassen verflechten (tangling). Diese sich überschneidenden Belange (crosscutting concerns) werden in der aspektorientierten Programmierung zu Entitäten ersten Grades erhoben, welche nun gleichberechtigt auf der Ebene der Klassen behandelt werden sollen: Aspekte.

Mit "Aspectual Collaborations" (Herrmann und Mezini, 2001a) steht ein Modell zur aspektorientierten Softwareentwicklung zur Verfügung, welches die getrennte Entwicklung wiederverwendbarer Aspekte ermöglicht und zudem auf eine Nahtlosigkeit zwischen Entwurf und Implementierung ausgelegt ist.

Ziel dieser Arbeit ist es, die Bedeutung und Verwendung von Aspekten auf der Entwurfsebene zu klären und einen grafischen Editor zur Modellierung von Aspectual Collaborations zu entwickeln. Der zu entwickelnde Editor soll sich als weiteres Werkzeug in die repository-basierte Softwareentwicklungsumgebung PIROL (Herrmann und Mezini, 2000) einfügen.

Struktur dieser Arbeit

Der erste Teil dieser Arbeit beschäftigt sich mit aspektorientierter Programmierung. Es wird die Motivation, die zu diesem neuen Paradigma geführt hat geklärt und daran anschließend werden vier unterschiedliche Modelle der aspektorientierten Programmierung einander gegenübergestellt.

Im zweiten Teil wird zunächst die Bedeutung des Entwurfes in der Softwareentwicklung erörtert, um dann den Besonderheiten des aspektorientierten Entwerfens gerecht zu werden. Anschließend wird mit UML For Aspects (UFA, siehe Herrmann,

2002a) eine Erweiterung der UML für den aspektorientierten Entwurf vorgestellt, die vom zu entwickelnden grafischen Editor unterstützt werden soll.

Der dritte Teil der Arbeit beschreibt die Softwareentwicklungsumgebung PIROL, in die der zu entwickelnde grafische Editor eingebettet werden soll. Dies geschieht entlang allgemein gültiger Integrationsdimensionen für Softwareentwicklungsumgebungen.

Der vierte Abschnitt erläutert die Entwicklung des in die Softwareentwicklungsumgebung PIROL eingebetteten grafischen Editors für den aspektorientierten Entwurf mit UFA.

1 Aspektorientierte Programmierung

1.1 Motivation

Die Informationstechnik ist heute allgegenwärtig. Aus dem öffentlichen Bereich ist sie nicht mehr wegzudenken und im wirtschaftlichen Bereich wird sie häufig unternehmenskritisch eingestuft. Auch kann festgestellt werden, dass die Lebensdauer von Software wesentlich größer ist, als von vielen angenommen wurde – spätestens der Jahreswechsel ins neue Jahrtausend hat dies eindrucksvoll gezeigt. Mit dieser gewandelten Bedeutung stellen sich für die Softwaretechnik spezielle Fragen mit wachsender Relevanz, die bisher nicht befriedigend gelöst werden konnten:

- Welche Techniken und Werkzeuge ermöglichen eine erfolgreiche *Erweiterbarkeit* und *Softwareevolution*?
- Wie können dem Anwendungsgebiet nach inhärent komplexe Softwaresysteme gestaltet werden, damit sie dennoch *handhabbar*, *wartbar* und *verständlich* bleiben?
- Wie kann Software konstruiert werden, damit sie *wiederverwendbar* ist und möglichst leicht auf andere Aufgabengebiete übertragen werden kann?
- Wie können Systeme entwickelt werden, die eine leichte *Anpassbarkeit* an unterschiedliche Kundenwünsche gewährleisten?

Diese Fragen haben zum einen die Entwicklung von Komponentenmodellen vorangetrieben, werfen aber auch die Frage auf, in wie weit die derzeitigen Hilfsmittel auf der programmiersprachlichen Ebene verbessert werden müssen, um diesen Zielen näher zu kommen.

1.2 Separation of Concerns

"Separation of Concerns" (Dijkstra, 1976) ist ein fundamentales Prinzip der Informatik. In verallgemeinerter Form beschreibt es die Dekomposition von Systemen in

einzelne Module, die jeweils nur die für einen bestimmten Zweck oder ein bestimmtes Ziel relevanten Informationen umfassen.

Die Modularisierung ist aus der Informatik nicht mehr weg zu denken. Sie produziert die zur Wiederverwendung geeigneten Bausteine und ermöglicht die unabhängige und gleichzeitige Arbeit an ihnen. Module helfen, Softwaresysteme handhabbar, kontrollierbar und verstehbar zu machen.

Concerns können unterschiedlichster Art sein. Sie können auf einer sehr *hohen Komplexitätsebene* angesiedelt sein, wie z.B. die Begriffe der Dienstgüte (quality of service) oder der Sicherheit. Sie können aber auch auf einer *niedrigen Komplexitätsebene* liegen, wie z.B. die Belange Caching und Buffering. Ebenso können Concerns *funktionaler* Natur sein und bestimmte Features beinhalten, wie z.B. den Datenexport oder die Druckausgabe. Genauso können sie aber auch *nichtfunktional* sein und systeminterne Anforderungen umfassen, wie Synchronisation oder Transaktionsmanagement. Bei der Einteilung in funktionale und nichtfunktionale Concerns gilt es, den gewählten Blickwinkel zu berücksichtigen, da dieser die Einteilung maßgeblich beeinflusst.

1.2.1 Crosscutting Concerns

Verschiedene Arbeiten (z.B. Harrison und Ossher (1993), Kiczales u. a. (1997)) weisen darauf hin, dass die derzeit verwendeten Techniken dem in der Informatik grundlegenden Konzept des "Separation of Concerns" und dementsprechender Modularisierungen noch nicht genügend gerecht werden. Es zeigt sich, dass es eine wiederkehrende Menge von Concerns gibt, deren Elemente sich mit den vorhandenen Techniken nicht modularisieren lassen, weil sie nicht entlang der verwendeten Strukturierungsdimensionen liegen.

Diese als *Crosscutting* bezeichnete Eigenschaft von Concerns führt zwangsweise dazu, dass die Realisierung – in der dem Problem nicht angemessenen Dekompositionsweise – über viele einzelne Module verstreut wird (*Scattering*). In den einzelnen Modulen selbst vermischen sich dann die Realisierungen *verschiedener* Concerns (*Tangling*).

Scattering

Tangling

In Abbildung 1.1 ist die Zerlegung eines Systems entlang einer Dimension, in der objektorientierten Programmierung beispielsweise der Klassen-Dimension, dargestellt. Auf die gewählte Dimension bezogen liegt das System nun in drei Module separiert vor, jedoch überschneiden die Concerns Logging, Synchronisation und Persistenz die etablierten Modulgrenzen. Richtet man den Blick auf ein einzelnes Modul, so ist zu erkennen, dass es jeweils Realisierungen mehrerer Concerns beinhaltet.

Im folgenden wird es um verschiedene Techniken gehen, die neue Mittel zum Umgang mit diesen "Crosscutting Concerns" bereitstellen.

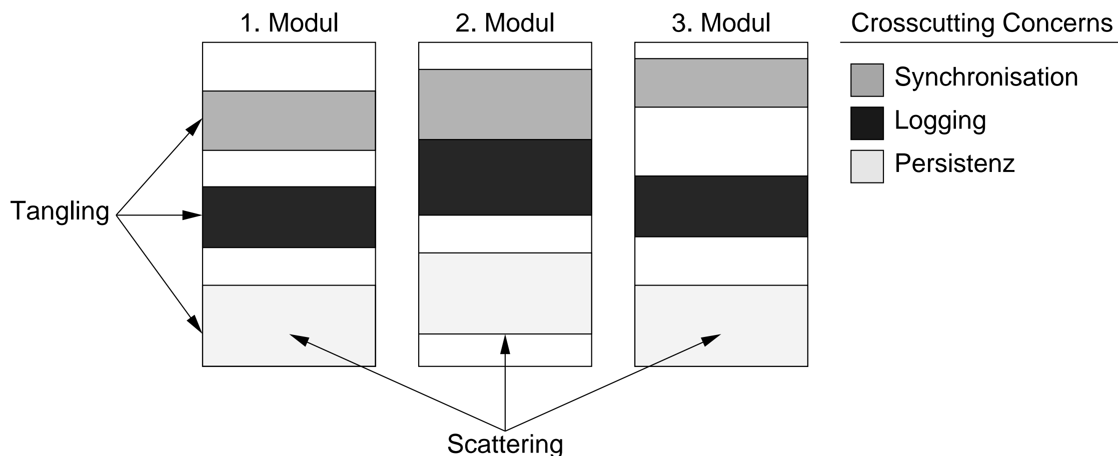


Abbildung 1.1: Crosscutting Concerns führen zu Tangling und Scattering

1.3 Aspektorientierung

Der Begriff der "aspektorientierten Programmierung" geht auf eine Veröffentlichung von Kiczales, Lamping, Menhdhekar, Maeda, Lopes, Loingtier und Irwin (1997) zurück und beschreibt das dort vorgestellte Programmiermodell. Im Einklang mit der Forschungsgemeinde (s.a. Elrad u. a., 2001; Ossher, 2002) wird in dieser Arbeit der Begriff in verallgemeinerter Form verwendet und soll das gesamte Gebiet bezeichnen, das sich mit den softwaretechnischen Mitteln zum Umgang mit "Crosscutting Concerns" beschäftigt.

Im folgenden werden vier aspektorientierte Ansätze vorgestellt, die sich auf unterschiedliche Weise bemühen, der "Crosscutting Concerns" Herr zu werden. Zwei der Problemstellung inhärente Schritte zur Lösung sind in den Beiträgen auf unterschiedliche Weise und in verschiedenen Ausprägungen realisiert:

Separation: Es werden Möglichkeiten geschaffen, Concerns voneinander zu trennen und mit ihnen im Weiteren als Entitäten ersten Grades (*first-class-entities*) zu arbeiten, sie werden also in neuartiger Form modularisiert.

Komposition: Die einzelnen Concerns werden auf spezifische Art und Weise wieder zu einem Ganzen zusammengefügt.

1.4 Hyper/J

Hyper/J (Tarr und Ossher, 2000) ist eine Umsetzung des von Tarr, Ossher, Harrison und Sutton Jr. (1999) vorgestellten Hyperspace-Modells zum mehrdimensionalen "Separation of Concerns" für die Programmiersprache Java.

1.4.1 Hyperspaces

Der Hyperspace-Ansatz geht sehr allgemein vor: Er abstrahiert von konkreten Dokumenttypen und lässt sich somit auf *alle* in der Softwareentwicklung anfallenden Artefakte (Anforderungen, Diagramme, etc.) anwenden. Diese Menge von Artefakten soll nun entlang der mannigfaltigen Concerns strukturiert werden. Zu welchem Zeitpunkt dies geschehen soll ist nicht festgelegt und auch nachträglich möglich.

Unit

Concern-Space

Alle Artefakte des Entwicklungsprozesses liegen in einer dem Problem und dem aktuellen Lösungsstand angemessenen Sprache vor. Die Palette reicht hier von natürlicher Sprache für die Anforderungsdefinitionen, über Diagrammnotationen der Analyse- und Designphasen bis hin zur Implementierung in Programmiersprachen. Die Artefakte lassen sich demnach weiter in syntaktische Einheiten der jeweiligen Sprachen zerlegen, die hier mit *Unit* bezeichnet werden. Alle Units zusammengenommen bilden also den gesamten Raum, den es mit den Concerns zu strukturieren gilt und der *Concern-Space* genannt wird.

Concern-Matrix

Concern-Dimension

Um diesen Concern-Space mit all seinen Units zu ordnen, wird er im Hyperspace-Ansatz als Matrix aufgefasst (*Concern-Matrix*). Die Matrix kann beliebig viele Achsen besitzen, von denen jede einzelne eine bestimmte Concern-Dimension darstellt. *Concern-Dimension*en fassen alle Concerns einer bestimmten Art zusammen, so enthielte z.B. eine Klassendatei-Dimension alle Concerns (hier also Klassendateien), die Klassendefinitionen enthalten. Eine Andere Dimension könnte alle Concerns enthalten, die einzelnen Features zugeordnet werden (wie z.B. "Ausgabe", "Darstellung" oder "Plausibilitätsprüfung").

Jede Dimension enthält jeweils *alle* Units des gesamten Concern-Space. Soweit eine Unit noch keinem speziellen Concern einer Dimension zugewiesen wurde, ist er einem in jeder Dimension automatisch vorhandenen "None" Concern zugeordnet. Jede Unit darf in einer Dimension immer nur genau einmal enthalten sein, also genau einem Concern zugeordnet werden.

Da die Unit in *allen* Concern-Dimensionen genau einmal enthalten ist, definiert ein Punkt in der Concern-Matrix (dem geordneten Hyperspace) die Unit in Bezug auf *alle* unterschiedlichen Concern-Dimensionen.

Mit dem durch beliebig viele Concern-Dimensionen aufgespannten Hyperspace steht also ein sehr mächtiges Mittel zur Verfügung, alle in der Softwareentwicklung anfallenden Units entlang der identifizierten Concerns zu strukturieren.

Diese neue Strukturierung hat jedoch die durch die dominante Dekompositionsdimension erzwungenen Modulgrenzen eingerissen. Um mit den nun nach beliebigen Concerns strukturierten Artefakten wieder sinnvoll arbeiten zu können, müssen sie erneut gekapselt werden. Dies geschieht in so genannten Hyperslices.

Hyperslice

deklarativ vollständig

Eine *Hyperslice* besteht aus beliebig vielen Units des gesamten Hyperspace. Eine Bedingung, die jede Hyperslice jedoch erfüllen muss ist, dass sie *deklarativ vollständig* sein muss. Damit wird gefordert, dass alle von den in dieser Hyperslice referenzierten

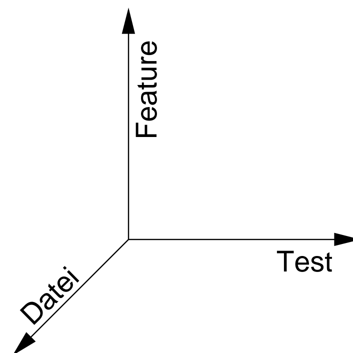


Abbildung 1.2: Concern-Matrix

Units entweder in dieser Hyperslice auch *definiert* sind, oder aber zumindest *deklariert* werden. Die deklarative Vollständigkeit bewirkt also eine Entkopplung der Hyperslice und ermöglicht damit die angestrebte Kapselung entlang beliebiger Grenzen.

Mit der Concern-Matrix ist es jetzt also möglich, beliebige Concerns zu identifizieren und ihnen Units der vorhandenen Artefakte zuzuordnen. Mit den Hyperslices wurden Module geschaffen, die durch ihre Eigenschaft der deklarativen Vollständigkeit die Grundvoraussetzung für die Integration mehrerer Hyperslices zu ganzen Systemen bieten.

Hypermodul

Diese Integration geschieht in so genannten *Hypermodul*en. Sie bestimmen, welche Hyperslices zu einem Modul integriert werden und nach welchen Regeln dies geschieht. Um am Ende ein lauffähiges System zu konstruieren, muss ein Hypermodul eine Menge von Hyperslices so kombinieren, dass jeder Unit-Deklaration auch eine passende Unit-Definition gegenübersteht.

Die Integrationsregeln legen fest, in welcher Weise die Units miteinander kombiniert werden. Im einfachsten Fall korrespondieren zwei namensgleiche Units aus zwei Hyperslices miteinander, von denen die eine die Unit lediglich deklariert und die andere sie implementiert.

Häufig sind jedoch zusätzliche Detailangaben notwendig, denn was soll z.B. geschehen, wenn *mehrere* Units eine Implementierung enthalten? Soll die Implementierung überschrieben werden? Wenn ja, welche? Müssen alle Implementierungen nacheinander ausgeführt werden? Wenn ja, in welcher Reihenfolge? Und falls die Signaturen nicht übereinstimmen, welche Parameter sollen weggelassen oder hinzugefügt oder welche Typumwandlungen vorgenommen werden?

1.4.2 Hyper/J

Hyper/J ist eine eigenständige Anwendung, die das Hyperspace-Konzept für Java umsetzt. Die Programmiersprache Java selbst bleibt dabei unverändert: Hyper/J arbeitet

mit den konventionellen Java-Klassendateien und kann demnach auch nachträglich zum Einsatz kommen. Die hyperspace-spezifischen Angaben (Units des Hyperspace, deren Zuordnung zu den Concerns der Concern-Dimensionen und die Integration zu Hypermodulen) werden gleich im einzelnen vorgestellt und finden ihren Platz in separaten Dateien. Aus den Java-Klassendateien und der Hyperspace-Konfiguration erstellt Hyper/J dann wieder Klassendateien. Zusätzlich wird ein Pseudo-Quelltext erzeugt, der zum Debuggen dient.

Im folgenden werden die einzelnen Elemente von Hyper/J kurz vorgestellt und danach auf ein kleines Beispiel-Szenario angewandt.

Concern-Space bevölkern

Im ersten Schritt der Verwendung von Hyper/J muss der Concern-Space mit den Units bevölkert werden, die später entlang der verschiedenen Concern-Dimensionen strukturiert werden sollen. Units sind in Hyper/J derzeit Java-Packages, Klassen, Interfaces und Klassen-Member. Sie werden durch Nennung ihrer Klasse oder der Klassendatei (bzw. Interface oder Interfacedatei) in einer *Hyperspace-Spezifikation*sdatei mit der Endung ".hs" aufgelistet:

Hyperspace-Spezifikation

```
hyperspace HyperspaceName
  composable class package1.class1, package2.class2;
  composable class package3.* except class3;
  composable class package4.class4 including subclasses;
  composable file "/user/florian/project/package5/*";
  uncomposable class package5.*;
```

Ein "*" kann in beiden Varianten als einfaches Wildcard benutzt werden. Falls bestimmte Elemente der Treffermenge des Wildcards ausgenommen werden sollen, kann dies mit `except` *`Class`* erreicht werden. Es gibt auch eine Möglichkeit, alle Subklassen einer speziellen Klasse in den Hyperspace aufzunehmen, ohne dass diese bereits bekannt sein müssen. Wird die Direktive `class ClassName including subclasses` angegeben, so durchsucht Hyper/J den gesamten Java-Klassenpfad nach Subklassen von ClassName und fügt diese zum Hyperspace hinzu. Die Schlüsselwörter `composable` und `uncomposable` geben an, ob die Klassen später von Hyper/J mit anderen Klassen integriert werden dürfen. Unerwünscht ist dies z.B. für Bibliotheksklassen, wie die Klassen der Java-Klassenbibliothek.

Concern-Space strukturieren

Der in der Hyperspace-Spezifikation geschaffene Concern-Space wird nun mit den sog. *Concern-Mapping*s strukturiert. In Concern-Mapping Dateien (mit der Endung

Concern-Mapping

".cm") werden die Concern-Dimensionen und deren Concerns definiert. Wenn Hyper/J die Concern-Mapping-Spezifikation verarbeitet, erzeugt es automatisch eine erste Dimension: die Klassendatei-Dimension. In ihr erscheinen die einzelnen Dateien als die Concerns der Dimension. Damit entspricht diese Dimension der von der objektorientierten Programmierung mit Java bekannten Dimension.

Die Zuordnung der Units (Packages, Klassen, Interfaces, Member) geschieht in einem Concern-Mapping folgender Form: `Unit-Art Unit : Dimension.Concern;` und könnte wie folgt aussehen:

```
package java.lang                         : Feature.Kernel;
package java.util                         : Feature.Utilities;
class java.util.zip.CRC32                 : Feature.Arithmetics;
interface java.util.zip.Checksum          : Feature.Integrity;
operation java.util.zip.Deflater.deflate: Feature.Compression;
field java.util.zip.Deflater.memberField: Feature.SomeFeature;
class java.util.zip.CRC32                 : Test.Util;
```

Die Direktiven werden nacheinander bearbeitet und später erfolgte Zuordnungen überschreiben die vorausgegangenen. Wie in der letzten Zeile des obigen Listings zu sehen ist, kann der Einfachheit halber in einer einzelnen Concern-Mapping Datei auch die Beschreibung mehrerer Dimensionen erfolgen. Das Überschreiben der Zuordnungen findet jedoch nur innerhalb der einzelnen Concern-Dimensionen statt. Im Beispiel ist die Unit `CRC32` in der Dimension `Feature` dem Concern `Arithmetics` zugeordnet und gleichzeitig in der `Test`-Dimension dem Concern `Util`.

Da in jeder einzelnen Hyperspace-Dimension *alle* Units enthalten sein müssen, werden die nicht explizit aufgeführten Units implizit dem *None-Concern* zugeordnet. Der Concern-Space ist dadurch nun in verschiedene Dimensionen und deren jeweilige Concerns geordnet worden.

None-Concern

Hyperslices zu Hypermodulen integrieren

Die Vorteile dieser Dekomposition ohne dominante Dekompositionsrichtung kommen natürlich erst mit der Integration der unterschiedlichen Concerns zum Tragen. Die Integration wird in der *Hypermodul-Spezifikation* textuell beschrieben und in einer Datei mit der Endung ".hm" festgehalten. Dort werden die zu integrierenden Concerns sowie die bei der Integration zu befolgenden Regeln festgelegt:

Hypermodul-Spezifikation

```
hypermodule HypermoduleName;
  hyperslices:
    Dimension1.Concern1;
    Dimension1.Concern2;
    Dimension2.Concern1
  relationships:
```

`mergeByName`

Hyperslices wurden auf Seite 6 als deklarativ vollständige Mengen von Units vorgestellt. Sie sind die Bausteine zur Integration von größeren Bausteinen bis hin zu kompletten Systemen. In Hyper/J werden Hyperslices derzeit nur durch die Concerns der einzelnen Dimensionen gebildet. Laut Hyper/J-Manual wird es in Zukunft möglich werden, Mengenoperationen (Vereinigungsmenge, Schnittmenge usw.) auf die Concerns anzuwenden.

Korrespondenz

Eine wichtige Integrationsbeziehung ist die *Korrespondenz*. Sie gibt an, welche einzelnen Hyperslices miteinander integriert werden sollen. Wie dies dann im einzelnen geschieht wird von der gewählten Kompositionsstrategie bestimmt.

Zur Integration von Hyperslices stehen drei grundlegende Kompositionsstrategien zur Auswahl, die jeweils durch zusätzliche Regeln an die tatsächlichen Gegebenheiten angepasst werden können. In der derzeit aktuellen Version 1.0 von Hyper/J sind diese noch nicht gänzlich realisiert (Tarr und Ossher, 2000, Kap. 4.5). Da sie jedoch konzeptionell wichtig sind und wohl bald realisiert werden, werden sie hier dennoch alle vorgestellt.

- `mergeByName` gibt an, dass die zu kombinierenden Units der Hyperslices anhand ihrer Namen korrespondieren sollen. Wenn zwei Units den gleichen Namen tragen, so werden sie miteinander zu einer neuen Unit integriert.

- `nonCorrespondingMerge` bedeutet, dass Units gleichen Namens nicht automatisch miteinander assoziiert werden.

- `overrideByName` gibt an, dass Units gleichen Namens einander entsprechen und durch Überschreiben miteinander kombiniert werden. Dies geschieht in der Reihenfolge, in der die Hyperslices in der Hypermodul-Spezifikation angegeben wurden.

Nachdem die zugrunde liegende Kombinations-Strategie ausgewählt wurde, kann sie mit den folgenden Direktiven detaillierter konfiguriert werden :

- `equate` lässt zwei Units zueinander korrespondieren. Dies ist z.B. sinnvoll, wenn sie unterschiedliche Namen tragen oder mit der Strategie `nonCorresponding-Merge` integriert wird.

- `order` legt die relative Ausführungsreihenfolge zweier Units fest. Dies kann angebracht sein, wenn z.B. Methoden kombiniert werden, von denen die eine die Basisfunktionalität enthält und die andere eine davon abhängige Zusatzfunktion.

- `rename` erlaubt es, Units unter einem anderen als ihrem ursprünglichen Namen in die Komposition aufzunehmen.

- `merge` lässt wie `equate` zwei Units zueinander korrespondieren und sorgt zusätzlich noch dafür, dass die Units auch kombiniert werden. Im Falle von `equate` hängt die Integration nachdem die Units zueinander in Beziehung gesetzt wurden noch von der zugrunde gelegten Integrationsstrategie ab.
- `noMerge` verhindert, dass zwei Units kombiniert werden bzw. dass eine Unit die andere überschreibt, selbst wenn die zugrunde gelegte Strategie dies verlangen würde.
- `override` legt fest, dass eine Unit die andere überschreibt.
- `match` lässt eine Unit mit einer durch einen Wildcard beschriebenen Menge von Units korrespondieren. Ebenso wie `equate` beeinflusst `match` nur, ob Unit-Paare zueinander korrespondieren und nicht deren Integration, die immer noch von der gewählten Integrationsstrategie abhängt.
- `bracket` umschließt die durch einen Wildcard ausgewählten Methoden mit zwei durch `before` und `after` angegebenen Methoden. Mittels `from` Direktive kann zusätzlich festgelegt werden, dass die hinzugefügten Methoden nur dann ausgeführt werden, wenn der Aufruf aus bestimmten Units kommt.
- `summary function` erlaubt es, eine besondere Methode zur Berechnung des Rückgabewertes festzulegen. Im Normalfall gibt Hyper/J immer das letzte Ergebnis der miteinander integrierten Methoden zurück. Die `summary function` bekommt als Parameter ein Array mit den Ergebnissen der einzelnen Methodenaufrufe übergeben und kann anhand dessen den geeigneten Rückgabewert bestimmen.

1.5 AspectJ

AspectJ (Kiczales u. a., 2001b) wurde entwickelt, um die von Kiczales u. a. (1997) vorgestellte "Aspektorientierte Programmierung" auch in der praktischen Anwendung zu erforschen. AspectJ erweitert Java um Sprachkonstrukte zur aspektorientierten Programmierung. AspectJ-Programme lassen sich mit dem AspectJ-Compiler in gewöhnliche Java-Classfiles übersetzen, die dann auf allen standardkonformen Java Virtual Machines laufen. Zudem ist AspectJ abwärtskompatibel, so dass alle Java-Programme auch gültige AspectJ-Programme darstellen. Für einige Entwicklungsumgebungen (JBuilder, Forte for Java, GNU Emacs, XEmacs, JDEE) werden Erweiterungen bereitgestellt, die die AspectJ Sprachkonstrukte integrieren.

In ihrer Arbeit zur aspektorientierten Programmierung stellten Kiczales u. a. fest, dass es eine bestimmte Klasse von Programmeigenschaften gibt, die sich mit den her-

kömmlichen Generalized-Procedure Sprachen[1] nicht geeignet (also dem Prinzip des "Separation of Concerns" nach) ausdrücken lassen. Diese spezifischen Programmeigenschaften werden *Aspekte* genannt.

Aspekte

AspectJ erweitert die Sprache Java um Elemente zum Formulieren dieser Aspekte. Sie werden analog zu gewöhnlichen Klassen in separaten Dateien gespeichert und durch den AspectJ-Compiler `ajc` in Java-Classfiles übersetzt.

Vier Elemente bilden die Grundlage zur Modularisierung von "Crosscutting Concerns" mit AspectJ: das Join Point Modell, Pointcuts, Advices und Introductions.

Join Points

Die *Join Points* beschreiben wohldefinierte Zeitpunkte in der Ausführung eines AspectJ-Programmes. Sie identifizieren beispielsweise den Zeitpunkt eines Methodenaufrufes oder den Zeitpunkt zu dem auf eine Membervariable zugegriffen wird. Das Join Point Modell stellt damit ein Gerüst bereit, an dem die separat vorgenommenen Realisierungen von "Crosscutting Concerns" im Programm verankert werden können.

Pointcuts

Pointcuts beschreiben bestimmte Ausprägungen von Join Points in einem AspectJ-Programm, z.B. den Aufruf des Konstruktors der Klasse K. Sie können benannt werden und legen fest, welche Methoden z.B. ein Methodenaufruf-Join Point im einzelnen betreffen soll. Pointcuts können auch aus mehreren Join Point Bezeichnern gebildet werden, die mit Boolescher Logik verknüpft werden.

Um beispielsweise in einer Grafikanwendung die Zeitpunkte zu definieren, an denen die Koordinaten eines Punktes gesetzt werden oder ein komplettes Zeichenelement platziert wird, könnte folgender Programmtext verwendet werden:

```
pointcut move():
  call(void Point.setX(int)) ||
  call(void Point.setY(int)) ||
  call(void Figure.place(int, int));
```

In diesem Beispiel wird ein Pointcut mit Namen `move` definiert, der alle Methodenaufrufe von `Point.setX()` und `Point.setY()` sowie von `Figure.place()` identifiziert.

Im Beispiel wird deutlich, dass sich Pointcuts nicht mehr an die bisherigen Modularisierungsgrenzen (hier die der Methoden und Klassen) halten müssen. Vielmehr ist es mit ihnen möglich, *neue* Grenzen (und damit Module) zu etablieren, die die objektorientierten Strukturen schneiden können, und dennoch die Implementierung separiert halten.

Advices

Advices enthalten zusätzliches Verhalten in einem AspectJ-Programm und legen fest, wie und unter welchen Bedingungen sich dieses Verhalten auswirkt. Um nach

[1] Alle Sprachen, deren zentrale Dekompositions- und Kompositionsmechanismen in irgendeiner Form auf generalisierten Prozeduren beruhen. Dazu gehören z.B. prozedurale, funktionale und objektorientierte Sprachen.

den im obigen Beispiel festgelegten Zeitpunkten z.B. alle Positionsänderungen zu protokollieren, dient folgender Advice:

```
after(): move() {
  System.out.println("Moved");
}
```

Die bisher beschriebenen Join Points, Pointcuts und Advices dienen dazu, an bestimmten Zeitpunkten der Ausführung eines AspectJ-Programmes zusätzliches Verhalten ausführen zu lassen. Dieser Mechanismus wird *dynamisches Crosscutting* genannt.

dynamisches Crosscutting

Daneben gibt es auch noch die Möglichkeit, a posteriori Typen ohne Eingriff in die Klassen selbst zu verändern (*open classes*). In AspectJ wird dieses *statische Crosscutting* durch Introductions ermöglicht.

open classes

statisches Crosscutting

Mit *Introductions* lassen sich Typ-Hierarchien verändern, in dem sie neue Features (Attribute und Methoden) einführen, die Liste der implementierten Interfaces erweitern oder eine Klasse von einer Elternklasse erben lassen.

Introductions

Um einer Klasse `Point` durch einen Aspekt z.B. einen Zähler für die Anzahl der Positionsänderungen hinzuzufügen, kann die folgende Introduction verwendet werden. In diesem Beispiel ist auch zu sehen, dass die Sichtbarkeit der ergänzten Features kontrolliert werden kann:

```
private int Point.moveCounter = 0;
private void Point.moved() {
  this.moveCounter++;
}
public void Point.resetMoveCounter() {
  this.moveCounter = 0;
}
```

1.6 Composition Filters

Das Composition Filters Modell (Aksit und Bergmans, 2001) stellt ebenfalls neue Mittel bereit, um das Problem der "Crosscutting Concerns" in den Griff zu bekommen. Die Grundlage des Modells bildet der der Objektorientierung immanente Nachrichtenaustausch.

Nachrichten dienen Objekten zur Kommunikation und bestimmen ihr Verhalten. Durch sie kann in der objektorientierten Programmierung beispielsweise auch das zentrale Konzept der Vererbung realisiert werden. Wenn in einer objektbasierten Umsetzung der Vererbung ein Objekt zum Ausführen einer Methode eine Nachricht

geschickt bekommt, führt es diese aus, sofern es eine entsprechende Methodendefinition enthält. Andernfalls schickt das Objekt die Nachricht an sein entsprechendes Elternobjekt weiter, damit dieses die Ausführung der Methode übernimmt.

An dieser Stelle wird deutlich, dass Nachrichten einen mächtigen Hebel zur Modifikation des Programmverhaltens darstellen. Um beispielsweise eine neue Vererbungsbeziehung zu realisieren reicht es aus, die Nachricht so zu verändern, dass sie nach bestimmten Kriterien nicht mehr dem Elternobjekt (oder nicht *nur* dem Elternobjekt) zugestellt wird, sondern einem anderen Objekt, welches dadurch quasi zum neuen Elternobjekt wird. Diese zentrale Bedeutung der Nachrichten macht sich das Composition Filters Modell zu Nutze, in dem es eine gezielte und strukturierte Manipulation von Nachrichten vorsieht.

Composition Filter

Das Composition Filters Modell stellt eine neue Art von Modulen bereit, die so genannten *Composition Filter* Klassen. Das Modell abstrahiert von konkreten Programmiersprachen und setzt lediglich den der objektorientierten Programmierung immanenten Nachrichtenaustausch voraus.

Composition Filters folgen dem in Abb. 1.3 gezeigten Aufbau. Ein Composition Filter aggregiert null oder mehr interne Klassen und sein Verhalten wird durch einen oder mehrere Filter bestimmt. Die internen Klassen können wiederum Composition Filter Klassen sein oder gewöhnliche, in einer beliebigen objektorientierten Sprache formulierte Klassen. Der Implementierungsteil der Composition Filter Klasse enthält das klassenspezifische Verhalten und ist ebenfalls nicht auf eine bestimmte Programmiersprache festgelegt.

Alle Nachrichten, die von Objekten dieser Composition Filter empfangen werden, müssen zuerst die Eingangsfilter (*input filters*) durchlaufen. Es gibt verschiedene Arten von Filtern, die unterschiedliche Aufgaben übernehmen. Allen Filtern ist jedoch gemein, dass sie einen Filterausdruck besitzen, der festlegt, ob der Filter zutrifft oder nicht – und damit bestimmt, ob die Akzeptierungs- oder die Ablehnungsaktion des Filters ausgeführt wird. In Tabelle 1.1 sind die verschiedenen Filter mit ihren jeweiligen Aktionen dargestellt.

Ein Filter hat die Form `Name:FilterTyp = {Filterausdruck}`. Im Beispiel (Abb. 1.3) trifft der Dispatch-Filter `f` auf alle Methoden zu, die in der internen Klasse `vparent` oder im Implementierungsteil der Filterklasse (referenziert durch `inner`) definiert sind.

Die erwähnte Modifikation der Vererbungshierarchie, die Einführung einer neuen Elternklasse, wäre damit bereits verwirklicht: `CF-Klasse` erbt damit praktisch von `vparent:P`, weil durch den Dispatch-Filter alle Nachrichten, die in der Signatur der internen Klasse `vparent` enthalten sind, auch an diese weitergeleitet werden. Die zweite eingehende Nachricht wird im Beispiel an den Implementierungsteil der Filterklasse weitergeleitet, da deren Signatur nicht in der internen Klasse `vparent` enthalten ist.

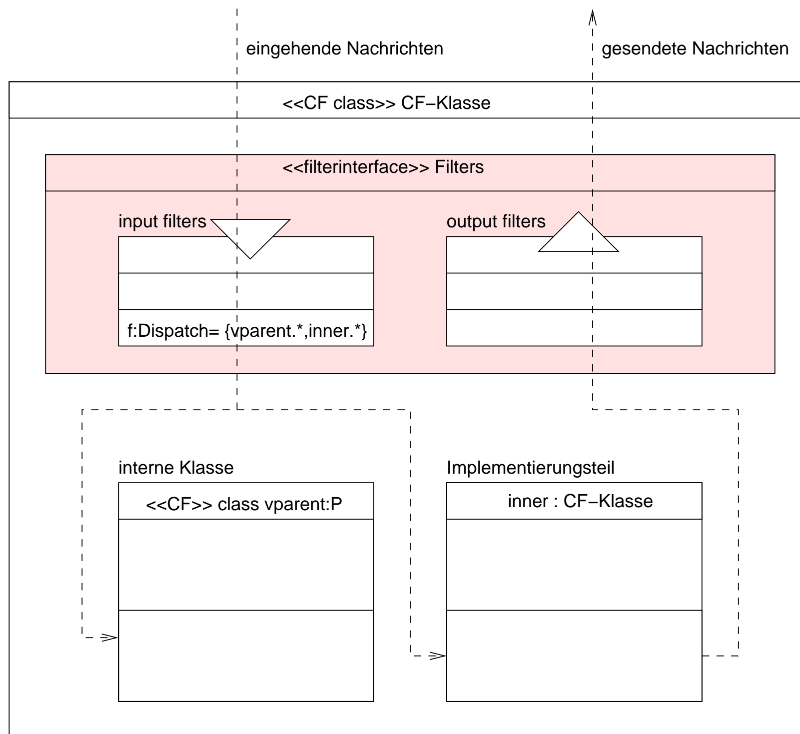


Abbildung 1.3: Aufbau eines Composition Filters

Filtertyp	**Akzeptierungs-Aktion**	**Ablehnungs-Aktion**
Dispatch	Nachricht an den Empfänger der Nachricht leiten.	Mit dem nächsten Filter fortfahren.
Error	Mit dem nächsten Filter fortfahren.	Eine Exception werfen.
Meta	Die Nachricht wird als Parameter einer (Meta-) Nachricht an ein bestimmtes Objekt gesendet. Das Zielobjekt kann die übergebene Nachricht dann inspizieren, modifizieren und ihre Ausführung wieder anstoßen.	Mit dem nächsten Filter fortfahren.
RealTime	Die RealTime-Eigenschaften der Nachricht werden angepasst.	Mit dem nächsten Filter fortfahren.
Wait	Mit dem nächsten Filter fortfahren.	Die Nachricht wird so lange in einer Warteschlange gehalten wie der Filterausdruck in einer Ablehnung resultiert.

Tabelle 1.1: Filter-Arten

Die Filterausdrücke können neben dem bisher gezeigten Testen auf Signaturen auch von der Auswertung Boolescher Funktionen abhängig gemacht werden. Dies ermöglicht zusätzlich zum Auswerten der Signatur (und damit statischer Informationen) auch die Auswertung des Zustandes (und damit dynamischer Informationen). Als Beispiel denke man an die (auch nachträgliche!) Realisierung eine Kapazitätsbegrenzung für eine Queue. Eingehende Nachrichten zum Hinzufügen eines Elementes würden dann je nach Ergebnis einer den Füllstand der Queue berücksichtigenden Funktion behandelt.

Die gezielte und strukturierte Manipulation von Nachrichten mit speziellen Filter-Modulen stellt sich als ein mächtiges Werkzeug abseits der von der Objektorientierung bekannten Modularisierungen dar. Bisher wurden die Verhaltensaspekte der Composition Filter beschrieben.

Superimposition

Um die in den Filtern formulierten "Crosscutting Concerns" mit den einzelnen Concerns zu verbinden wird der Mechanismus der *Superimposition* eingeführt (Abb. 1.4). Die Superimposition erlaubt es, Filterinterfaces zu anderen Concerns hinzuzufügen. Dazu wird in der Superimpositions-Spezifikation festgelegt, welche Filterinterfaces welchen Concerns angeheftet werden:

```
selectors
  selected = { *=Class1, *=Class2 };
Filterinterfaces
  selected <- Filters;
```

Es werden dem Selektor `selected` alle Instanzen der Concerns `Class1` und `Class2` zugeordnet. Die Superimpositions-Spezifikation wird beendet, in dem festgelegt wird, dass das Filterinterface `Filters` in alle Instanzen der durch den Selektor `selected` ausgewählten Concerns eingeführt wird.

Die Superimposition stellt damit ein neues Hilfsmittel zur Komposition dar. In den Composition Filter Klassen können "Crosscutting Concerns" als eigene Module realisiert werden. Die Superimposition verbindet diese neuen Module mit den bisher verwendeten objektorientierten Modulen.

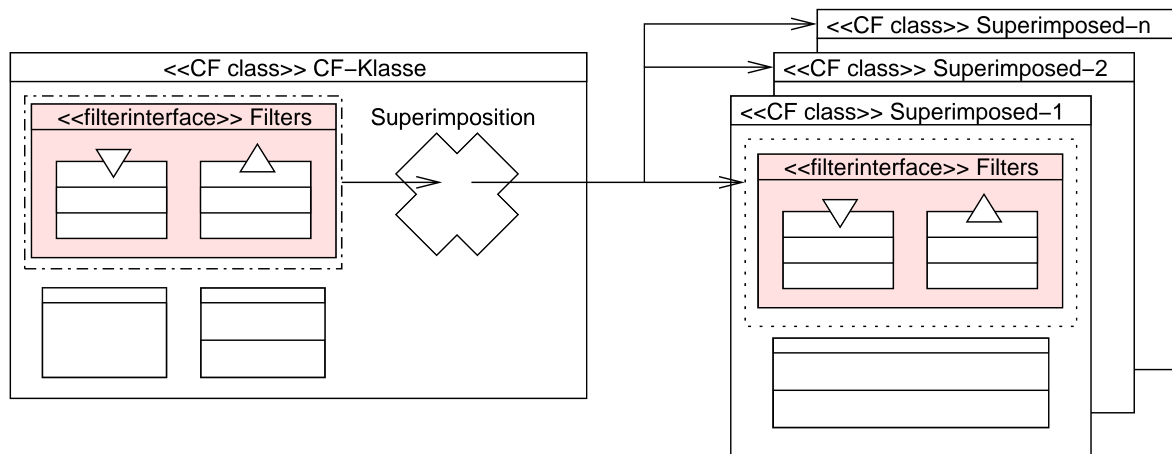


Abbildung 1.4: Superimposition eines Composition Filters

1.7 Aspectual Collaborations

Das Modell der Aspectual Collaborations (Herrmann und Mezini, 2001a) wurde ausgehend von der Arbeit an Aspectual Components (Lieberherr, Lorenz und Mezini, 1999) entwickelt. In der Sprache LAC (Herrmann und Mezini, 2001b) wurden Aspectual Components erstmals implementiert und das Modell wurde anhand der gesammelten Erfahrungen weiterentwickelt und verfeinert.

Das Modell

Aspectual Collaborations erleichtern den Umgang mit den "Crosscutting Concerns" im Sinne des "Separation of Concerns" ebenfalls durch die Einführung neuer Module. Ähnlich der Programmierung mit "Adaptive Methods" (Lieberherr u. a., 2001) berücksichtigt das Aspectual Collaborations Modell in besonderer Weise, dass Concerns häufig die *Kollaboration* verschiedener Klassen betreffen und es genau aus diesem Grund zum Scattering und Tangling kommt.

Kollaborationen werden als neuer Modul-Typ eingeführt. Kollaborationen sind am ehesten mit Paketen (packages) zu vergleichen, denn sie umfassen in der Regel mehrere Klassen und kapseln deren Zusammenspiel. Die einzelnen Klassen innerhalb einer Kollaboration repräsentieren die in der Kollaboration auftretenden *Rollen*.

Kollaborationen

Rollen

Die einzelnen Teilnehmer der Kollaboration (participants) dürfen unvollständig implementiert sein. Gewöhnliche Klassen kennen nur das Konzept der *abstrakten* Methoden, welches es ihnen erlaubt, Teile der Implementierung vorerst offen zu lassen. Diese Teile werden dann von Subklassen implementiert.

Im Gegensatz dazu können die Teilnehmer einer Kollaboration Methoden deklarieren und verwenden, deren Implementierung später an anderer Stelle erfolgt und

nicht auf Vererbung angewiesen ist. Ein Kollaborationsteilnehmer wird also gegen eine erwartete Schnittstelle (*expected interface*) implementiert. Kollaborationen sind dennoch in sich geschlossene Module, da sie zwar an einigen Stellen unvollständig implementiert, aber dennoch *deklarativ vollständig* sind. Kollaborationen beschreiben also das Zusammenspiel verschiedener Rollen, enthalten die Teile der für das gemeinsame Wirken notwendigen Implementierung und besitzen mit der erwarteten Schnittstelle klar definierte *open spots*.

expected interface

Natürlich können Kollaborationen auch vollständig definierte Klassen beinhalten, die speziell in der Kollaboration benötigt werden. Dies sind dann gewöhnliche Klassen, die nur innerhalb der Kollaboration sichtbar sind, wie innere Klassen in Java.

Eine Kollaboration muss nicht nur ein Behälter für die teilnehmenden Rollen sein. Sie kann auch Eigenschaften besitzen, die sich auf die Kollaboration als solche beziehen. Dies können sowohl Attribute, als auch Methoden sein, die der Kollaboration als eigener Einheit zugehörig sind.

Mit Kollaborationen stehen also am Ende des "Separation of Concerns" Module zur Verfügung, die kollaborative Aspekte einzeln beschreiben können. Mindestens so wichtig wie die Möglichkeit zur Modularisierung ist es aber, die separat behandelten Concerns wieder zu vollständigen Systemen integrieren zu können.

Konnektoren

Im Aspectual Collaborations Modell übernehmen *Konnektoren* die Aufgabe, das in einer Kollaboration beschriebene kollaborative Verhalten der teilnehmenden Rollen an Klassen von Basispaketen zu binden. Dies geschieht transparent, so dass der Einsatz von Kollaborationen auch nachträglich erfolgen kann, weil keine speziellen Vorkehrungen in den Basispaketen getroffen werden müssen.

Konnektoren binden Kollaborationen an Basispakete und spezifizieren dazu die erforderlichen Details auf drei Ebenen. Auf der *Ebene von Klassen* werden Rollen der Kollaboration mit Klassen eines Basispaketes identifiziert. Damit wird festgelegt, dass eine Basisklasse K die Rolle R einer Kollaboration übernimmt.

Auf der *Ebene von Methoden* werden Zuordnungen gemacht, die zum einen Methodenaufrufe aus der Kollaboration heraus und zum anderen Methodenaufrufe in die Kollaboration hinein beschreiben.

callouts

Die so genannten *callouts* legen für die in der Kollaboration bisher lediglich deklarierten Methoden fest, an welche Methode einer Basisklasse ihre Aufrufe innerhalb der Kollaboration *delegiert* werden sollen.

callins

Aufrufe, die von bestimmten Stellen eines Basispaketes in die Kollaboration hinein erfolgen sollen, werden über *callins* definiert. Wie in AspectJ können diese callin-Bindungen festlegen, wann ein in eine Basisklasse eingewobener Aufruf in die Kollaboration bezüglich der Methode erfolgen soll (`before`, `after`, `replace`).

Auf der dritten Ebene, der *Ebene der Parameter*, können die für die Methodenaufrufe notwendigen Parameter den jeweiligen Signaturen entsprechend angepasst werden.

Ein Anwendungsbeispiel

Als Beispiel soll eine Anwendung für die Wetterüberwachung an einem Flughafen betrachtet werden (Abb. 1.5). Der Wettersensor `WeatherSensor` überwacht die Wetterlage (Temperatur, Luftdruck, Niederschlag) und gibt sie sobald eine signifikante Änderung eintritt zeilenweise auf dem Telex-Drucker `TelexPrinter` des Towers aus. Um nun die Wetterdaten weiterer Sensoren auf dem Telex-Drucker ausgeben zu können, muss eine einfache Synchronisation geschaffen werden, die dafür Sorge trägt, dass die einzelnen Ausgabezeilen der verschiedenen Sensoren nicht durcheinander geraten.

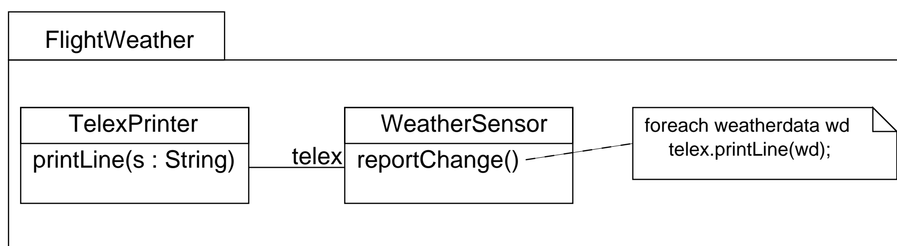


Abbildung 1.5: Paket FlightWeather

Eine naheliegende Lösung besteht darin, dem `TelexPrinter` und dem `WeatherSensor` die benötigte Synchronisations-Funktionalität hinzuzufügen. Allerdings erkennt man dabei sofort, dass man auf dem besten Wege ist, einen *einzelnen* Concern (die Synchronisation) auf *mehrere* Module zu verteilen (Scattering). Zudem scheint der Concern “Synchronisation” nicht wirklich zu den Aufgaben eines Wettersensors oder eines Telex-Druckers zu gehören, mit der Folge, dass sich die Realisierung dieses Concerns mit den eigentlichen Aufgaben der zwei Module zwangsweise vermischen würde (Tangling).

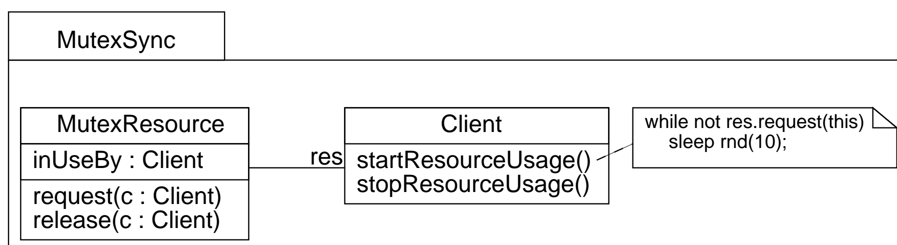


Abbildung 1.6: Paket MutexSync

Mit Aspectual Collaborations lässt sich, unabhängig von der eben beschriebenen Anwendung, das allgemeine Konzept zur Synchronisation der Nutzung einer exklusiv verfügbaren Resource in einer Kollaboration formulieren (Abb. 1.6). Die Kollaboration beschreibt das gemeinsame Verhalten einer exklusiv nutzbaren Resource

(MutexResource) und ihres Clients. Ein Klient versucht vor der Nutzung in startResourceUsage() durch Aufrufen von request(), den exklusiven Zugriff von der Resource zugeteilt zu bekommen. Ist dies – evtl. erst nach einer Wartezeit – geschehen, kann er die Resource nutzen und beendet die Nutzung der Resource in seiner Methode stopResourceUsage(), die ihrerseits die Resource über release() wieder freigibt.

Um das FlightWeather Paket mit diesem Synchronisationsverhalten auszustatten, müssen die beiden Pakete nicht verändert werden. Es ist lediglich ein Konnektor zu definieren, der angibt, welche Klassen aus dem Paket FlightWeather nun welche Rollen aus der Kollaboration MutexSync übernehmen sollen (Abb. 1.7).

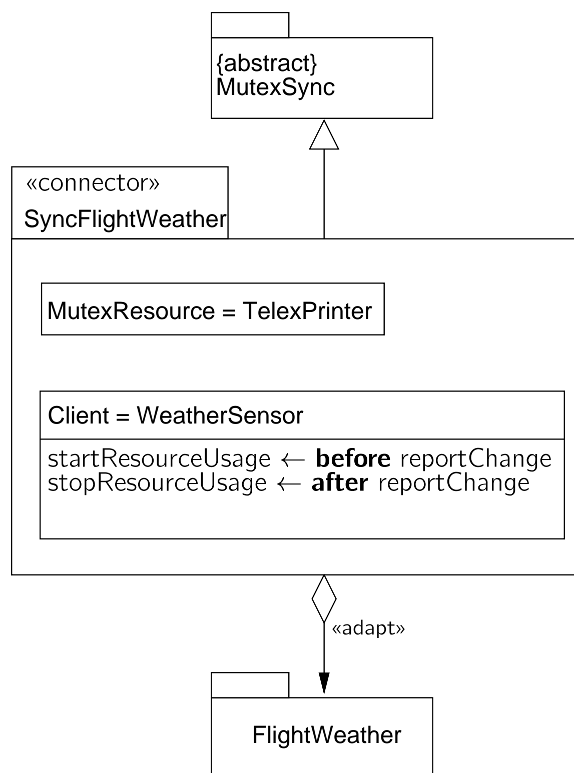


Abbildung 1.7: Ein Konnektor bindet die Synchronisations-Kollaboration an ein Basispaket

Der Konnektor SyncFlightWeather verfeinert die Kollaboration MutexSync und adaptiert das Basispaket FlightWeather. Die Details der Bindung werden innerhalb des Konnektors definiert: der Telex-Drucker soll die Rolle der exklusiv verfügbaren Resource spielen, der Wettersensor die Rolle des Klienten übernehmen. Immer *bevor* im Wettersensor die Methode reportChange() ausgeführt wird, soll die Methode startResourceUsage() der Klientenrolle aufgerufen werden (callin). Immer *nach* der Ausführung der Methode reportChange() wird stopResourceUsage() aufgerufen, in der die Resource wieder frei gegeben wird.

2 Aspektorientiertes Entwerfen

In diesem Teil der Arbeit wird das Entwerfen mit Aspectual Collaborations genauer erörtert. Um dies in sinnvoller Weise durchführen zu können ist es unabdingbar, den Kontext zu analysieren, in den der Softwareentwurf eingebettet ist. Denn nur wenn die Randbedingungen und Ziele des Kontextes verstanden werden, können die von Aspectual Collaborations gebotenen Neuerungen angemessen eingeordnet werden. Zunächst geht es demnach um den Softwareentwurf an sich und danach um Aspekte, deren konkrete Ausprägung Aspectual Collaborations und die Entwurfsnotation UFA (UML for Aspects).

2.1 Softwareentwurf

Laut der Definition eines IEEE Standards zur Terminologie in der Softwaretechnik (IEEE Std 610.12-1990) bezeichnet der Softwareentwurf sowohl den Prozess in dem die Architektur, Komponenten, Schnittstellen und andere Charakteristika eines Systems oder einer Komponente definiert werden, als auch das Ergebnis dieses Prozesses[1].

Dass der Softwareentwurf sprachübergreifend sowohl den Prozess als auch das Produkt bezeichnet, bringt passend zum Ausdruck, dass das eine nicht ohne das andere betrachtet werden kann. Der Entwurf als Prozess und der Entwurf als Produkt stehen in enger Wechselbeziehung zueinander.

Gleichzeitig darf nicht übersehen werden, dass ein Entwurf für sich alleine genommen keinen eigenen Wert besitzt. Nach welchen Kriterien sollte man ihn auch bewerten?

Auf der Suche nach charakterisierenden Merkmalen des Entwurfes wird deutlich, dass der Entwurf selbst erstaunlich gestaltlos bleibt. Erst wenn man den Blick auf seine Umgebung ausdehnt, erkennt man, dass er vollkommen in ihr eingeschlossen ist und erst durch diesen ihn umgebenden Kontext seine Form bekommt. Die Einbettung in ein größeres Ganzes lässt den Softwareentwurf Gestalt annehmen.

[1] "Design: (1) The process of defining the architecture, components, interfaces, and other characteristics of a system or component. (2) The result of the process in (1)."

Wenn im Folgenden also zunächst das Umfeld des Softwareentwurfes betrachtet wird, so sollte immer bedacht werden, dass dieses Umfeld mit seinen unterschiedlichen Anforderungen dem Softwareentwurf seine Form gibt (Abb. 2.1). Auf indirekte Weise wird also im folgenden Abschnitt der Softwareentwurf selbst beschrieben, oder genauer: der Bereich, den der Softwareentwurf ausfüllt.

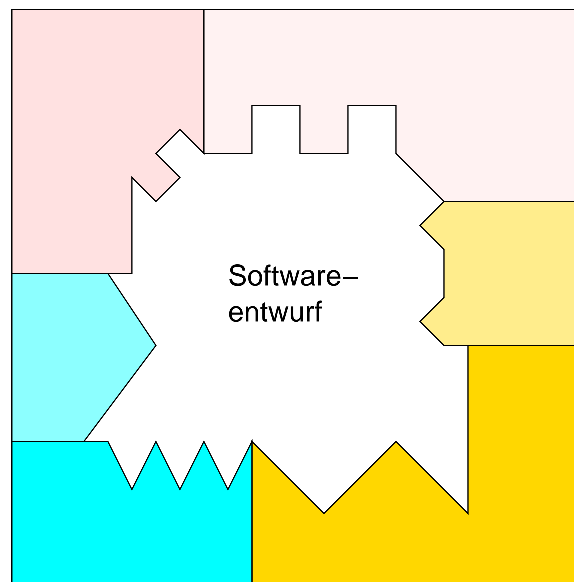


Abbildung 2.1: Form des Softwareentwurfes

2.1.1 Umfeld

Wie im vorigen Abschnitt erläutert wurde kommt dem Umfeld des Softwareentwurfes eine besondere Bedeutung zu. Der Softwareentwurf tritt immer im Kontext eines Softwareentwicklungsprozesses auf, dessen Ziel die Herstellung eines Software-Produktes ist. In Abbildung 2.2 ist die logische Abfolge der den Softwareentwurf umgebenden Phasen des Entwicklungsprozesses und die dem Entwicklungsprozess vorgelagerte Aktivität der Systementwicklung (*system engineering*) dargestellt.

Die logische Abfolge beschreibt den generellen Verlauf des Entwicklungsprozesses von dessen Initialisierung bis zum Produkt. Die Phasen der tatsächlichen Durchführung lassen sich nicht strikt linear anordnen. Vielmehr gehört es zu den zentralen Erkenntnissen der Softwaretechnik, dass eine Rückkopplung zwischen den einzelnen Phasen unabdingbar ist. Die im Entwicklungsprozess in nachfolgenden Schritten gewonnen Erkenntnisse und die damit verbundenen neuen Fragestellungen müssen in einer den Kontext berücksichtigenden Sichtweise betrachtet werden. Dies kann nur mit der umfassenderen Sichtweise der vorgelagerten Phase geschehen und verlangt daher nach Zyklen.

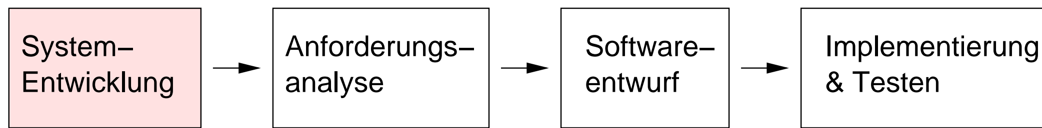


Abbildung 2.2: logische Abfolge der umgebenden Entwicklungsphasen

Systementwicklung

Softwareentwicklung findet in dem durch die Systementwicklung gesteckten Rahmen statt. Die Systementwicklung analysiert, entwirft und organisiert computerbasierte Systeme. Den Kontext der Systementwicklung kann ein gewerbliches Unternehmen mit seinen Geschäftsprozessen bilden oder ein bestimmtes zu erstellendes Produkt, wie z.B. ein Verkehrsleitsystem.

Je nachdem ob ein Unternehmen oder ein Produkt betrachtet wird, spricht man von *information engineering* bzw. *product engineering*. Unter dem Begriff "Systementwicklung" sollen im weiteren beide Teilgebiete zusammengefasst werden, da diese Unterscheidung für die Verbindung zur Softwareentwicklung von untergeordneter Bedeutung ist.

Ein von der Systementwicklung zu erstellendes computerbasiertes System bedient sich verschiedener Elemente, um die benötigte Informationsverarbeitung zu organisieren. Zu den wichtigsten gehören: Menschen, Software, Hardware, Daten und Prozesse.

Festgehalten werden muss, dass die Softwareentwicklung durch die Systemdefinition der Systementwicklung einen festen inhaltlichen Rahmen zugewiesen bekommt, den sie nicht von sich aus verlassen kann. Die grundlegenden Aufgaben der Systemkomponenten in deren übergeordneter Organisation werden im Zuge der Systementwicklung bestimmt.

Anforderungsanalyse

In der Anforderungsanalyse (*requirements engineering*) wird das von der Systementwicklung abgesteckte Gebiet unter verschiedenen Blickwinkeln analysiert. Dies geschieht, um die Frage zu beantworten: *Was* soll das zu erstellende Softwaresystem tun?

Die Anforderungsspezifikation spielt in der Entwicklung größerer Systeme verschiedene Rollen (Faulk, 1997):

- Für den Kunden dokumentieren die Anforderungen typischerweise, was ihm geliefert werden soll und können als vertragliche Basis für die Entwicklung dienen.
- Für Manager können die Anforderungen eine Basis für den Zeitplan bilden und ein Maß zur Fortschrittsmessung darstellen.

- Für Software Designer spezifizieren die Anforderungen das Entwurfsziel.
- Für Programmierer definieren die Anforderungen den Rahmen der möglichen Lösungen und bestimmen die zu produzierenden Ausgaben.
- Für Qualitätssicherer bilden die Anforderungen die Grundlage, um die Validierung, Testplanung und Verifikation durchzuführen.

Ausgehend von der Systemdefinition erfolgt die Erarbeitung der Anforderungen, deren Quellen vielfältig sind. Neben dem Kunden und den zukünftigen Nutzern des Systems können z.B. auch die vorhandene Informations- und Kommunikationsstruktur oder gesetzliche Vorschriften spezifische Anforderungen stellen. Meist wird daher zwischen *funktionalen* und *nichtfunktionalen* Anforderungen unterschieden.

Die Ausarbeitung der Anforderungsanalyse muss besonders sorgfältig geschehen, denn eine auf falschen oder unvollständigen Annahmen beruhende Lösung mag noch so perfekt realisiert sein, wenn sie das falsche Problem löst, ist sie dennoch wertlos. Auch kostet die Umsetzung zu spät identifizierter Anforderungen ein Vielfaches. Boehm und Basili (2001) halten in Abhängigkeit von der Größe des Projektes beispielsweise Faktoren von fünf bis 100 für realistisch.

Zentraler Bestandteil in der Durchführung der Anforderungsanalyse ist die intensive Kommunikation mit dem Kunden und den anderen relevanten Interessengruppen. In einem ersten Schritt geht es darum, ein gemeinsames Verständnis des Problems und der für die Problemlösung relevanten Objekte[1] des Problemraumes zu erfassen.

Abstrahiert man von den unterschiedlichen Methoden und ihren konkreten Modellen, so lässt sich feststellen, dass es in der Analysephase um die Erstellung eines Objektmodells und eines Benutzungsmodells geht (Jacobsen u. a., 1998). Das Objektmodell beschreibt ein mentales Modell, das die Benutzergruppe vom Problembereich hat. Mit diesem Modell wird ein gemeinsames Verständnis des Problembereiches angestrebt. Das Benutzungsmodell beschreibt die Interaktion des Benutzers mit dem zu entwickelnden System und stellt den angestrebten Lösungsweg unter Verwendung der Elemente des Objektmodells dar.

Unter Bezugnahme auf die erstellten Modelle werden die Anforderungen formuliert. Schrittweise werden sie vervollständigt und konkretisiert und die Modelle entsprechend angepasst. Es wird versucht alle unklaren Formulierungen zu beseitigen und implizite Annahmen explizit zu machen. Alle Anforderungen sollten so deutlich formuliert sein, dass jeweils eindeutig entschieden werden kann, ob sie erfüllt sind.

[1] Der Terminus "Objekt" bezeichnet im Kontext der Anforderungsanalyse nicht notwendigerweise Objekte im Sinne der Objektorientierung. Primär sind unter Objekten alle materiellen und konzeptuellen Gegenstände des Problembereiches zu verstehen, unabhängig davon wie sie in der konkreten Methode repräsentiert werden.

Das Produkt der Anforderungsanalyse ist die Anforderungsspezifikation. Bevor diese Spezifikation abgeschlossen wird, muss sie auf ihre Qualität hin begutachtet werden. Es soll sichergestellt werden, dass die Anforderungsspezifikation alle relevanten Anforderungen berücksichtigt und dass die erstellten Modelle konsistent, korrekt und valide sind.

Häufig dient die fertige Anforderungsspezifikation als vertragliche Basis für die Erstellung des geplanten Systems. Dies macht auch noch einmal deutlich, dass die Anforderungsanalyse eine von Kunden und Entwicklern gemeinsam zu leistende Arbeit ist. Das Fachwissen des Kunden ist für die korrekte Erstellung der Anforderungen unerlässlich. Den Entwicklern kommt die Aufgabe zu, dieses Wissen so zu systematisieren und zu formulieren, dass es den Kontext und die Aufgaben des anvisierten Systems treffend beschreibt.

Softwareentwurf

Das Ziel des Softwareentwurfes ist es, ein implementierbares Modell des Softwaresystems zu entwerfen, welches die analysierten Anforderungen erfüllt. Die Trennung zwischen Analyse und Entwurf lässt sich pragmatisch so formulieren: Die Analyse behandelt die Dinge, die den Kunden interessieren; der Entwurf behandelt zusätzlich Details der Umsetzung, die für den Kunden nicht von Belang sind[1].

In der Entwurfsphase dient die Anforderungsdefinition als Grundlage zur Erarbeitung einer Lösung. Da es in der Softwareentwicklung für gewöhnlich nicht *eine* "richtige" Lösung gibt, sondern viele mögliche, dient der Entwurf auch dazu, verschiedene Alternativen zu modellieren und auf ihre Eignung hin zu analysieren. An dieser Stelle kommen zu den bereits analysierten Anforderungen des Kunden auch Interessen der Entwickler hinzu. Der Entwurf kann z.B. maßgeblich davon beeinflusst werden, ob auf bereits bestehende Komponenten zurückgegriffen werden soll. Ebenso kann es im Interesse der Entwickler liegen, bestimmte Komponenten bereits auf die Wiederverwendung in anderen Kontexten hin zu optimieren.

Der Softwareentwurf selbst lässt sich in zwei Hauptbereiche einteilen: Zum einen wird die Softwarearchitektur entworfen (*architectural design*) und zum anderen werden die einzelnen Komponenten so weit beschrieben, dass sie implementiert werden können (*detailed design*).

Die Softwarearchitektur beschreibt die Subsysteme und Komponenten des Softwaresystems und ihre Beziehungen untereinander (Buschmann u. a., 1996, S. 384). Es

[1] Es muss beachtet werden, dass diese Formulierung versucht, den *thematischen* Kern der beiden Phasen zu treffen und keine Aussage über den Prozess darstellt. Wie in Abschnitt 2.1.1 betont wurde, interagieren die einzelnen Phasen miteinander, so dass sich aus dem Softwareentwurf heraus auch Fragen ergeben, die im Kontext der Anforderungen mit dem Kunden beantwortet werden müssen. Dies tritt z.B. auf, wenn sich in der Umsetzung herausstellt, dass bestimmte Anforderungen nur auf Kosten anderer zu realisieren sind und zwischen ihnen eine Abwägung getroffen werden muss.

wird von Implementierungsdetails wie Algorithmen und Datenstrukturen abstrahiert und die Interaktion einzelner als Black-Boxes betrachteter Komponenten modelliert.

Der detaillierte Entwurf ist als direkte Vorstufe zur Implementierung zu sehen. Die Realisierung der im Architekturmodell identifizierten Komponenten wird jetzt vorbereitet. Das Innenleben der Komponenten wird nun bis auf die Modulebene der anvisierten Programmiersprachen beschrieben und die zur Kommunikation verwendeten Datenstrukturen werden spezifiziert.

Implementierung und Testen

In der Implementierungsphase wird das Architekturmodell und das Modell des detaillierten Entwurfes in die Form von Programmtexten gebracht und getestet. Die Grenze zwischen dem detailliertem Entwurf und der Implementierung lässt sich nicht eindeutig ziehen. Zum einen hängt sie von der eingesetzten Entwicklungsmethode ab, die den gewünschten Detaillierungsgrad des Entwurfes vorgibt. Zum anderen scheint sich diese Grenze auch durch die heute verfügbaren Werkzeuge, die einen nahtlosen Übergang vom Entwurf zur Implementierung anstreben, immer weiter aufzulösen. Das Spektrum reicht dabei von automatischer Quelltextgenerierung bis zu den Bemühungen der Object Management Group (OMG), die modellgetriebene Architektur (Model Driven Architecture, MDA) zu etablieren.

Die MDA soll die Integration von Softwaresystemen so einfach wie möglich machen. Erreicht werden soll dies durch die volle Spezifikation der Systemfunktionalität in Modellen, die dann in einem weiteren Schritt auf unterschiedliche konkrete Plattformen gemappt werden können. Zur Formulierung der Modelle gibt die MDA Richtlinien und setzt auf Standards wie UML, MOF, CORBA, J2EE, XML und .NET. Gerade für große und langlebige Systeme ist die Aussicht darauf, dass sie in ihrem Lebenszyklus auf Modellebene mit weiteren Systemen integriert und auf die aktuellen Plattformen abgebildet werden können, verlockend.

2.1.2 Ziele

In den vorigen Abschnitten wurde der Softwareentwurf im Kontext des Entwicklungsprozesses beschrieben. Der Entwurf soll eine Lösung zu den ermittelten Anforderungen in einem Modell formulieren, das sich auf eine Implementierungssprache abbilden lässt. Daneben soll das Modell auch ein Werkzeug sein, das den Weg zu solch einer Lösung dadurch ebnet, dass es hilft, konkrete Lösungsideen zu kommunizieren, zu analysieren und zu evaluieren.

Die Aufgaben des Softwareentwurfes sind demnach umrissen. Die bisherige Beschreibung hat allerdings noch nicht die Frage nach den innerhalb des Entwurfes verfolgten Zielen beantwortet.

Im Weiteren wird es vorrangig um die Produkte des Softwareentwurfes und die Konzepte deren Erstellung gehen und weniger um die Prozess-Sicht auf den Softwareentwurf. Der Blick auf den Softwareentwurf als Schritt im Prozess deutet in die Richtung der zahlreichen Entwurfsmethoden mit ihren unterschiedlichen Repräsentationen, Vorgehensweisen und Heuristiken.

Die innerhalb des Entwurfes verfolgten Ziele leiten sich zum Großteil von den Anforderungen an das zu erstellende Produkt ab, welches natürlich von hoher Qualität sein soll. Aber was genau ist eigentlich Softwarequalität?

Softwarequalität

Softwarequalität eindeutig zu definieren ist schwierig. Dies liegt darin begründet, dass es sich beim Begriff "Qualität" um einen übergreifenden Ausdruck handelt. Qualität ist nichts Absolutes, sondern immer von den vom Betrachter gewählten Kriterien abhängig. Im IEEE Standard für Softwarequalitäts-Metrik-Methodologien (IEEE Std 1061-1998) wird Softwarequalität auch als "der Grad, zu dem Software eine gewünschte Kombination von Qualitätsmerkmalen besitzt"[1] beschrieben.

In einer Betrachtung von Softwarequalität benennt Bertrand Meyer zwei unterschiedliche Klassen von Qualitätsmerkmalen, die auch schon unter anderer Bezeichnung von McCall (McCall, 1977) unterschieden wurden: *externe* und *interne* Qualitätsmerkmale (Meyer, 1997).

Zu den externen Qualitätsmerkmalen gehören alle Merkmale, deren Vorhandensein oder deren Mangel von den Benutzern wahrgenommen werden können. Zu den Benutzern zählen neben den tatsächlich mit der Software interagierenden Menschen auch diejenigen, die auf der Seite des Kunden in anderer Weise Anforderungen an die Software stellen, wie z.B. Systementwickler, die die Software an andere Systeme anbinden müssen.

Zu den internen Qualitätsmerkmalen gehören Eigenschaften wie Modularität oder Lesbarkeit, die nur von den Softwareentwicklern wahrgenommen werden können, da sie den Zugriff auf die Quelltexte haben. Diese Merkmale wirken sich auf die Lernbarkeit, Änderbarkeit und Erweiterbarkeit der Software aus und werden insbesondere in späteren Phasen des Software-Lebenszyklus, wie Wartung und Erweiterung, relevant.

Am Ende zählen für das Produkt natürlich nur die externen Qualitäten; die internen Qualitätsmerkmale sind nur in so fern relevant, als dass sie bei der Sicherstellung der externen helfen sollen. Demnach werden im Folgenden die externen Qualitätsmerkmale beschrieben, da sie die übergeordneten Ziele des Softwareentwurfes darstellen. Meyer definiert die folgenden externen Qualitätsmerkmale:

[1] Software quality is the degree to which software possesses a desired combination of quality attributes.

1. *Korrektheit* ist die Eigenschaft von Softwareprodukten exakt die Aufgaben zu erfüllen, die durch ihre Spezifikation festgelegt wurden.
2. *Robustheit* ist die Eigenschaft von Softwaresystemen, angemessen auf unnormale Bedingungen zu reagieren.
3. *Erweiterbarkeit* ist die Möglichkeit, Softwareprodukte an geänderte Spezifikationen anzupassen.
4. *Wiederverwendbarkeit* ist die Eigenschaft von Softwarekomponenten, zur Erstellung vieler verschiedener Anwendungen beizutragen.
5. *Kompatibilität* ist die Leichtigkeit, mit der Softwarekomponenten mit anderen kombiniert werden können.
6. *Effizienz* ist die Eigenschaft von Softwaresystemen, so wenig Anforderungen wie möglich an Hardware-Resourcen, wie Rechenzeit, Speicherplatz oder Bandbreite bei Kommunikationsgeräten, zu stellen.
7. *Portierbarkeit* ist die Leichtigkeit mit der Softwareprodukte auf verschiedene Hard- und Softwareplattformen übertragen werden kann.
8. *Benutzerfreundlichkeit* ist die Leichtigkeit mit der Menschen mit verschiedenen Hintergründen und Qualifikationen die Benutzung von Softwareprodukten lernen und zur Problemlösung einsetzen können. Dazu gehört auch der Aufwand zur Installation, Benutzung und Überwachung.
9. *Funktionalität* ist der Grad an gebotenen Möglichkeiten eines Systems.
10. *Pünktlichkeit* ist die Eigenschaft eines Softwaresystems, veröffentlicht zu werden, wenn oder bevor der Benutzer es benötigt.

Bis auf "Pünktlichkeit" finden sich diese Merkmale auch in einer weiteren Definition von Softwarequalität in der Norm ISO/IEC 9126-1:2001. Sie beschreibt in sechs Kategorien mit insgesamt 27 Unterpunkten Merkmale und Eigenschaften von Software für Software-Qualitätsmodelle:

1. *Funktionalität* umfasst Charakteristiken, die sich auf den grundsätzlichen Zweck beziehen, für den die Software entwickelt wurde (Suitability, Accuracy, Interoperability, Security, Functionality Compliance).
2. *Zuverlässigkeit* beinhaltet die Fähigkeit von Software, ihre Leistung auf stetem Niveau unter bestimmten Bedingungen für eine bestimmte Zeitdauer aufrecht zu erhalten (Maturity, Fault tolerance, Recoverability, Reliability Compliance).

3. *Benutzerfreundlichkeit* umfasst die Eigenschaften einer Software, die die Mühe zur Nutzung und zum individuellen Erlernen dieser Nutzung für eine definierte oder implizit angenommene Benutzergruppe beschreiben (Understandibility, Learnability, Operability, Attractiveness, Usability Compliance).

4. *Effizienz* umspannt die Charakteristiken, die sich auf den Performanz-Grad von Software und die unter bestimmten Bedingungen benötigten Resourcen beziehen (Time behavior, Resource utilization, Efficiency Compliance).

5. *Wartbarkeit* beinhaltet Merkmale, die den Aufwand beschreiben, der zur Durchführung von Änderungen, Korrekturen und Verbesserungen oder zur Anpassung an Änderungen in der Softwareumgebung, der Anforderungen oder der funktionalen Spezifikationen nötig ist (Analyzability, Changeability, Stability, Testability, Maintainability Compliance).

6. *Portierbarkeit* umfasst die Charakteristiken von Software, die die Übertragung von Software von einer Organisation oder Hardware oder Softwareumgebung auf eine andere betreffen (Adaptability, Installability, Co-existence, Replaceability, Partability Compliance).

Durch diese Definitionen wird das Spektrum der durch die gesamte Entwicklung hindurch zu verfolgenden Ziele deutlich. Das Erreichen dieser Ziele sollte demnach auch in der Phase des Softwareentwurfes angestrebt werden.

2.1.3 Konzepte

Wie eingangs bereits beschrieben wurde, wird die Form des Softwareentwurfes entscheidend von den Anforderungen seiner Umgebung bestimmt. Damit ist es natürlich auch schwierig, Details am Softwareentwurf selbst zu identifizieren und diese zu beleuchten. Der Weg zu einer weiteren Charakterisierung des Softwareentwurfes führt daher über eine handvoll Konzepte, die sich zum Erreichen der Ziele als hilfreich erwiesen haben.

Da die im Folgenden beschriebenen Konzepte seit ihrer Entdeckung bis heute ihre Nützlichkeit beweisen, können sie mit gutem Grund als fundamentale Konzepte des Softwareentwurfes betrachtet werden. Diese Konzepte helfen zum einen, den Umgang mit den komplex gewordenen Softwaresystemen zu erleichtern. Zum anderen stellen sie Mittel dar, um die oben genannten Software-Qualitätsmerkmale zu verbessern.

Abstraktion

Durch das Konzept der *Abstraktion* werden beim Blick auf einen bestimmten Gegenstand der Betrachtung alle nicht essenziellen Details ausgeblendet. Aus der Tatsache, dass die Bedeutung “essenzieller Details” aus verschiedenen Blickwinkeln und zu

unterschiedlichen Zeitpunkten betrachtet variiert, folgt, dass es verschiedene Ebenen der Abstraktion gibt. Diese unterscheiden sich in ihrem jeweiligen Abstraktionsgrad.

Pressman identifiziert drei Arten der Abstraktion (Pressman, 1997): *Prozedurale Abstraktionen* sind benannte Sequenzen von Instruktionen; kaffee_kochen könnte demnach von den für die Zubereitung notwendigen Schritten abstrahieren. *Datenabstraktionen* betreffen die einzelnen Datenobjekte; ein Beispiel sind abstrakte Datentypen. Die dritte Art der Abstraktion ist die *Kontrollabstraktion*; sie beschreibt die Programmkontrolle und blendet deren Implementierungsdetails aus (z.B. Semaphoren).

Diese Abstraktionen dienen in der Softwareentwicklung dazu, die Komplexität der Aufgabenlösung in den Griff zu bekommen. Ausgehend von einer abstrakten Ebene erfolgt die Schrittweise Konkretisierung bis zur Implementierung.

Verfeinerung

Das Konzept der *Verfeinerung* bildet den Gegenpol zur Abstraktion. Letzteres ermöglicht dem Entwickler über Details hinwegzusehen wohingegen durch die Verfeinerung diese Details schrittweise erarbeitet werden.

Dekomposition

Die *Dekomposition* ist ein Konzept, welches ebenfalls dazu dient mit der begegneten Komplexität umzugehen. Durch die Zerlegung eines großen Problems in mehrere kleine Teilprobleme wird dessen Lösung erleichtert ("divide and conquer").

Modularität

Das Konzept der *Modularität* geht mit dem der Dekomposition einher. Module strukturieren ein System in einzelne, z.B. aus einer Dekomposition hervorgegangene, Komponenten. Durch diese strukturierende Wirkung helfen sie ebenfalls beim Umgang mit der Komplexität. Zudem ist die Modularität der Schlüssel zu vielen der oben genannten Software-Qualitätsmerkmale wie Erweiterbarkeit, Wiederverwendbarkeit, Testbarkeit oder Portabilität.

Zwei wichtige Charakteristika von Modulen sind *Kohäsion* und *Kopplung*. Kohäsion bezeichnet den inneren Zusammenhalt eines einzelnen Modules. Damit wird beschrieben, in wie weit die einzelnen Aufgaben und Teile eines Modules zusammengehören. Die Kohäsion ist z.B. sehr gering, wenn die einzelnen Teile nur zufällig in einem Modul zusammengefasst sind. Ein höherer Grad von Kohäsion liegt vor, wenn die einzelnen Teile z.B. logisch, sequenziell oder funktional zusammengehören.

Die Kopplung beschreibt den Grad der Abhängigkeit zwischen verschiedenen Modulen. Das Spektrum der möglichen Ausprägungen reicht von keiner direkten Kopplung über Datenkopplung (Parameterübergabe) bis hin zur inhaltlichen Kopplung, bei der ein Modul auf den Inhalt eines anderen Modules zugreift oder an eine bestimmte Stelle dessen Modulkörper verzweigt.

Um den größten Nutzen aus dem Konzept der Modularität zu ziehen, sollten Module eine möglichst große Kohäsion aufweisen und untereinander möglichst lose gekoppelt sein.

Module sollten auch dem *Open-Closed principle* (Meyer, 1997) gehorchen: Sie sollten *offen* sein, damit sie sich gut erweitern lassen. Gleichzeitig sollten Module

geschlossen sein, so dass sie für die effektive Nutzung von anderen Modulen aus zur Verfügung stehen.

Die Forderung nach Offenheit entstammt der Erfahrung, dass Module in ihrer Lebenszeit den sich ändernden Bedingungen angepasst werden müssen. Geschlossenheit (und damit z.B. die Stabilität von Schnittstellen oder die Verfügbarkeit als Bibliothek) wird insbesondere zum Bauen ganzer Systeme benötigt. Die Nutzung von Modulen als Bausteine eines Systems ist natürlich nur dann effizient, wenn sie keine immer wiederkehrenden Änderungen an den Stellen ihrer Verwendung verursacht und keinen erheblichen Kompilierungsaufwand mit sich bringt.

Information Hiding

Das Prinzip des *Information Hiding* hängt eng mit der angestrebten Modularität zusammen. Es besagt, dass Module nur über eine explizite Schnittstelle miteinander kommunizieren und Module ihre Interna wie Daten oder Funktionen vor dem direkten Zugriff von außen schützen. Typische zu versteckende Informationen sind zum Beispiel die Repräsentationen der in einem Modul verwendeten Datentypen und die eingesetzten Algorithmen.

Die Vorteile des Information Hiding liegen darin, dass Aufgaben von ihren Realisierungen entkoppelt werden. Ähnlich wie bei der Modularität wird damit die Änderbarkeit, Erweiterbarkeit, Testbarkeit usw. verbessert.

Lokalität

Eine weitere anzustrebende Eigenschaft ist *Lokalität*. Sie drückt aus, dass Elemente (Variablen, Methoden, Klassen etc.) nur in einem bestimmten Kontext verfügbar sind, nämlich genau in dem Kontext, in dem sie benötigt werden. Je höher die Lokalität ist, desto einfacher lassen sich Änderungen durchführen, weil der von der Änderung potenziell betroffene Bereich schon durch die Lokalität des Elementes begrenzt wird.

Hierarchien

Hierarchien stellen ein weiteres Konzept der Softwareentwicklung dar. Hierarchien strukturieren Mengen in übergeordnete Elemente und Elemente, die diesen untergeordnet werden. Ein untergeordnetes Element kann wiederum übergeordnetes Element für weitere ihm untergeordnete Elemente sein.

Hierarchien begrenzen durch die Einteilung in über- und untergeordnete Elemente die zwischen diesen Elementen möglichen Beziehungen. Diese Einschränkung steigert die Lokalität der in der Hierarchie geordneten Inhalte.

Durch Hierarchien können z.B. Programmstrukturen, Architekturen oder Datenstrukturen organisiert werden. Die erhöhte Lokalität führt in diesen Fällen durch die klarere Struktur zu besserem Programmverständnis, besserer Erweiterbarkeit, verringertem Korrekturaufwand usw.

2.1.4 Entwurfsqualität

Durch die Beschreibung des Softwareentwurfes im Hinblick auf seinen Kontext, seine Ziele und die in seiner Durchführung etablierten Konzepte hat der Softwareentwurf eine deutlichere Form bekommen. Wie es im Inneren des Softwareentwurfes aussieht

soll in diesem Abschnitt geklärt werden. Denn die Entwurfsziele sind nun zwar benannt, aber *wie* sieht eigentlich ein guter Entwurf aus, das heißt, ein Entwurf, der zu einem Produkt mit der erwarteten Softwarequalität führt?

Die oben beschriebenen Konzepte sind ein erster Hinweis in diese Richtung. Ihre Anwendung hat sich bewährt, so dass sie zumeist Eingang in die verwendeten Entwicklungswerkzeuge gefunden haben. Ein Beispiel dafür ist die objektorientierte Programmierung. Bei der Arbeit an einem Entwurf helfen diese Konzepte jedoch nur sehr bedingt weiter, da sie selbst keine konkreten Anwendungshinweise liefern. Der Raum, der für die Lösung mit den auf diesen Konzepten basierenden Werkzeugen noch zur Verfügung steht, ist immer noch so groß, dass genug Platz für schlechte Lösungen bleibt.

Der Weg zur hochqualitativen Software ist schwierig, weil sich die Softwareentwicklung einer algorithmischen Lösung entzieht. Aus diesem Grund kann es auch keine Lösungsanleitungen geben. Die in der Softwareentwicklung gesammelten Erfahrungen werden daher in Form von Heuristiken oder wiederverwendbaren Ideen wie Entwurfsmustern festgehalten. Die Hoffnung ist, dass durch die Berücksichtigung der Heuristiken und durch die Verwendung allgemein akzeptierter Teillösungen die gewünschte Qualität erreicht wird. Auch in der Qualitätssicherung bedient man sich dieser Empfehlungen. Die Qualität eines Entwurfes wird in Entwurfs-Reviews von mehreren Menschen begutachtet und diskutiert.

Aufschlussreicher sind jedoch Metriken, weil sie direkt bestimmbar und quantifizierend sind. Für die Charakterisierung von Programmtexten lassen sich einfach Metriken finden. Sie messen z.B. die Anzahl der Programmzeilen, Funktionen oder die Vererbungstiefe und Vererbungsbreite. Wie allerdings die Qualität eines Softwareentwurfes gemessen werden kann ist noch Gegenstand der Forschung. Es ist nicht klar ersichtlich, wie sich die abstrakten Qualitätsmerkmale wie Wiederverwendbarkeit oder Erweiterbarkeit eines Entwurfes messen lassen.

QMOOD

Eine umfangreiche Arbeit zu dieser Frage haben Bansiya und Davis (2002) durchgeführt. Ihr "Quality Model for Object-Oriented Design" (QMOOD) genanntes Modell spannt den Bogen von Qualitätsattributen des Entwurfes über die Eigenschaften objektorientierter Entwürfe und objektorientierte Entwurfs-Metriken bis zu den (den Entwurf konstituierenden) objektorientierten Komponenten (Abb. 2.3).

Eine Stärke des Modells, die es für die hier durchgeführte Annäherung an die Softwareentwurfs-Qualität so interessant macht, liegt darin, dass seine Validität nachgewiesen werden konnte. Die Erfüllung dieses Gütekriteriums bedeutet, dass die Messungen des Modells auch genau das Beabsichtige messen, nämlich die Qualität des Softwareentwurfes und keine andere Größe.

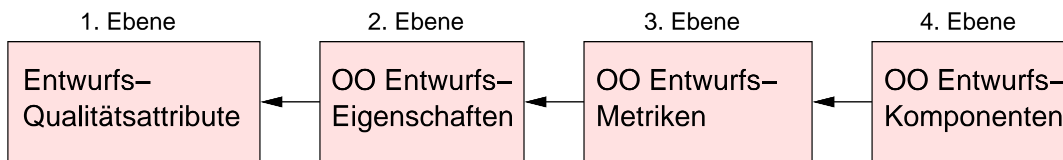


Abbildung 2.3: QMOOD Ebenen

Auf der ersten Ebene werden Qualitätsattribute von Softwareentwürfen definiert. Diese wurden von den in der ISO 9126 genannten Attributen der Softwarequalität (vgl. Abschnitt 2.1.2) abgeleitet. Dabei fielen Attribute heraus, die offensichtlich zu eng an die Implementierung gebunden sind wie z.B. "Verlässlichkeit". Andere Attribute wurden so abgewandelt, dass sie besser auf den Entwurf passen. So wurde das Attribut "Effizienz" zu "Effektivität" umgedeutet. Die sechs Qualitätsattribute eines Softwareentwurfes sind[1]:

1. *Wiederverwendbarkeit* betrachtet das Vorhandensein objektorientierter Entwurfs-Charakteristiken, die einen Entwurf ohne großen Aufwand auf ein neues Problem anwendbar machen.

2. *Flexibilität* ist ein Merkmal, welches die Einarbeitung von Änderungen in den Entwurf ermöglicht. Es beschreibt die Möglichkeit eines Entwurfes, so angepasst zu werden, dass er funktional ähnliche Aufgaben erfüllt.

3. *Verständlichkeit* bezeichnet die Eigenschaften eines Entwurfes, die es erlauben, ihn leicht zu lernen und zu verstehen. Die Verständlichkeit steht in direktem Verhältnis zur Struktur des Entwurfes.

4. *Funktionalität* beschreibt die den einzelnen Klassen eines Entwurfes zugeordneten Aufgaben, die durch ihre Schnittstellen veröffentlicht werden.

5. *Erweiterbarkeit* bezieht sich auf das Vorhandensein und die Nutzung von Eigenschaften eines bestehenden Entwurfes, die die Einarbeitung neuer Anforderungen in den Entwurf erlauben.

6. *Effektivität* bezieht sich auf die Fähigkeit eines Entwurfes, die angestrebte Funktionalität und das angestrebte Verhalten durch die Verwendung objektorientierter Entwurfskonzepte und -techniken zu erreichen.

Auf der zweiten Ebene wurden wichtige Entwurfseigenschaften ausgewählt, die durch die Inspektion des Entwurfes ermittelt werden können. Betrachtet werden die interne und externe Struktur, die Funktionalität und die Beziehungen von Klassen,

[1] Die englische Originalversion ist Tabelle B.1 des Anhangs auf Seite 95 zu entnehmen.

Methoden und Attributen. Die gewählten Eigenschaften sind *Entwurfsgröße*, *Hierarchien*, *Abstraktion*, *Kapselung*, *Kopplung*, *Kohäsion*, *Komposition*, *Vererbung*, *Polymorphie*, *Dienste* und *Komplexität*[1].

Auf der dritten Ebene werden objektorientierte Entwurfs-Metriken definiert. Dazu wurden bestehende objektorientierte Metriken auf ihre Eignung für die Quantifizierung der auf der zweiten Ebene definierten Entwurfseigenschaften hin untersucht. Nicht für alle Eigenschaften existierten bereits Metriken und die vorhandenen setzten oft eine nahezu vollständige Implementierung voraus. Daher wurden von den elf letztendlich verwendeten Metriken fünf neu eingeführt[2]:

1. *DSC (Design Size in Classes)*: Anzahl der Klassen.
2. *NOH (Number of Hierarchies)*: Anzahl der Hierarchien.
3. *ANA (Average Number of Ancestors)*: Durchschnitt der Anzahlen der Vorfahren.
4. *DAM (Data Access Metrics)*: Das Verhältnis der Anzahl der als privat und protected deklarierten Attribute zur Gesamtzahl der Attribute einer Klasse. Gewünscht ist ein hoher DAM-Wert.
5. *DCC (Direct Class Coupling)*: Zählt die Anzahl verschiedener von einer Klasse aus referenzierter anderer Klassen in ihren Attributen und Methodensignaturen.
6. *CAM (Cohesion Among Methods of a Class)*: Berechnet die Kohäsion der Methoden einer Klasse, basierend auf deren Parameterlisten. Es werden die Schnittmengen der Methodenparameter einer Methode mit der größten unabhängigen Menge aller Parametertypen der Klasse gebildet und deren Quotienten aufsummiert.
7. *MOA (Measure of Aggregation)*: Misst den Grad der durch Attribute realisierten Teil-Ganzes Beziehungen. Gezählt wird die Anzahl der Deklarationen benutzerdefinierter Typen.
8. *MFA (Measure of Functional Abstraction)*: Das Verhältnis der Methoden einer Klasse, die geerbt wurden, zu den für die Methoden der Klasse zugreifbaren Methoden.
9. *NOP (Number of Polymorphic Methods)*: Anzahl der als polymorph deklarierten Methoden.
10. *CIS (Class Interface Size)*: Anzahl der Methoden der öffentlichen Schnittstelle.

[1] Die genauen Definitionen sind Tabelle B.2 des Anhangs auf Seite 96 zu entnehmen.
[2] Die genauen Definitionen sind Tabelle B.3 des Anhangs auf Seite 97 zu entnehmen.

11. *NOM (Number of Methods)*: Gesamtzahl der Methoden einer Klasse.

Auf der vierten Ebene werden schließlich die den Softwareentwurf konstituierenden Komponenten und ihre qualitätstragenden Eigenschaften identifiziert. Die Komponenten sind die *Attribute*, die *Methoden* und die *Klassen* sowie deren *Beziehungen* zueinander. Außerdem werden die *Klassenhierarchien* einbezogen.

Zur Ermittlung der qualitätstragenden Eigenschaften wurden zu den verwendeten Attributen, Methoden und Klassen unter anderem jeweils Überlegungen zur Kapselung (wie sind Attribute und Methoden gekapselt?), zu den benutzten Parametern (Anzahl der Parameter, Parametertypen?) und zu den verwendeten Typen angestellt (handelt es sich um Basistypen oder um benutzerdefinierte Typen?).

Die Mengen der identifizierten qualitätstragenden Eigenschaften für Attribute, Methoden usw. sind groß, jedoch sind ihnen auch viele Elemente gemein. Diese Eigenschaften wurden daraufhin untersucht, ob sie zum Zeitpunkt des Entwurfes schon vorliegen und somit verwendet werden können. Die übriggebliebenen Eigenschaften, wie z.B. Kapselung, deckten sich annähernd mit den Entwurfseigenschaften (S. 34) und wurden diesen zugeordnet.

Die Verbindung der dritten Ebene zur zweiten Ebene wurde ebenfalls in Eins-zu-eins-Beziehungen hergestellt. Eine Metrik wird, wie in Tabelle 2.1 dargestellt, einer der Entwurfseigenschaften zugeordnet.

Entwurfseigenschaft	**Entwurfs-Metrik**
Entwurfsgröße	Design Size in Classes (DSC)
Hierarchien	Number of Hierarchies (NOH)
Abstraktion	Average Number of Ancestors (ANA)
Kapselung	Data Access Metric (DAM)
Kopplung	Direct Class Coupling (DCC)
Kohäsion	Cohesion Among Methods of a Class (CAM)
Komposition	Measure of Aggregation (MOA)
Vererbung	Measure of Functional Abstraction (MFA)
Polymorphie	Number of Polymorphic Methods (NOP)
Dienste	Class Interface Size (CIS)
Komplexität	Number of Methods (NOM)

Tabelle 2.1: Entwurfseigenschaften und ihre Metriken

Um die Verbindung zwischen der zweiten und ersten Ebene herzustellen, also die Eigenschaften eines Entwurfes mit den Entwurfs-Qualitätsattributen zu assoziieren, wurden die vorhandene Literatur und Veröffentlichungen zu objektorientierter Softwareentwicklung herangezogen. Die dort formulierten Ansichten und angenommenen

Zusammenhänge zwischen den Qualitätsattributen eines Softwareentwurfes und den Eigenschaften des Entwurfes ergeben das in Tabelle 2.2 dargestellte Bild. Das $\oplus$ bezeichnet einen positiven Zusammenhang zwischen der Entwurfseigenschaft und dem Qualitätsattribut des Entwurfes und das $\ominus$ eine Auswirkung in der umgekehrten Richtung.

	Wiederverwendbarkeit	**Flexibilität**	**Verständlichkeit**	**Funktionalität**	**Erweiterbarkeit**	**Effektivität**
Entwurfsgröße	$\oplus$		$\ominus$	$\oplus$		
Hierarchien				$\oplus$		
Abstraktion			$\ominus$		$\oplus$	$\oplus$
Kapselung		$\oplus$	$\oplus$			$\oplus$
Kopplung	$\ominus$	$\ominus$	$\ominus$		$\ominus$	
Kohäsion	$\oplus$		$\oplus$	$\oplus$		
Komposition		$\oplus$				$\oplus$
Vererbung					$\oplus$	$\oplus$
Polymorphie		$\oplus$	$\ominus$	$\oplus$	$\oplus$	$\oplus$
Dienste	$\oplus$			$\oplus$		
Komplexität			$\ominus$			

Tabelle 2.2: Zusammenhang von Qualitätsattributen und Entwurfseigenschaften

Bei der Auswertung des so schwierig in Regeln zu formulierenden Wissens hat sich herausgestellt, dass die *Wiederverwendbarkeit* stark durch die Entwurfsgröße, die vorhandene Kopplung und Kohäsion sowie die Anzahl der angebotenen Dienste bestimmt wird. Die *Flexibilität* wird durch Kapselung, Komposition und Polymorphie erhöht, wohingegen die Kopplung diese verringert.

Auf die *Verständlichkeit* des Softwareentwurfes üben die Mehrzahl der identifizierten Entwurfsattribute einen Einfluss aus. Mit steigender Größe, Abstraktion, Kopplung, Polymorphie und Komplexität sinkt die Verstehbarkeit. Eine hohe Kapselung und hohe Kohäsion wirken sich dagegen positiv auf die Verständlichkeit aus. Die *Funktionalität* wird maßgeblich von der Entwurfsgröße, den vorhandenen Hierarchien, der Kohäsion, der Polymorphie und den angebotenen Diensten befördert.

Die *Erweiterbarkeit* wird positiv beeinflusst, wenn der Abstraktionsgrad hoch liegt und Gebrauch von Vererbung und Polymorphie gemacht wird. Einbußen in der Erweiterbarkeit sind insbesondere bei hoher Kopplung zu erwarten. Die effektive Nutzung der objektorientierten Techniken (*Effektivität*) wird durch die Nutzung von Abstraktion, Kapselung, Komposition, Vererbung und Polymorphie sichergestellt.

Die im QMOOD analysierten Zusammenhänge von Entwurfs-Qualitätsattributen bis zu den objektorientierten Entwurfs-Komponenten vermitteln ein gutes Bild vom

Inneren des Softwareentwurfes und dessen, was es für einen Entwurf bedeutet, qualitativ hochwertig zu sein.

Das Ziel dieser Beschreibung von QMOOD darin lag, die Qualität eines Entwurfes an sich und deren wichtigsten Einflussfaktoren zu verdeutlichen. Dazu war die modellbasierte und quantitative Herangehensweise von QMOOD an dieses Thema besonders geeignet, angesichts der Alternative, lediglich die Best-Practices des Entwurfes zu wiederholen. Das valide Modell zeigt, dass sich die Qualität eines wie im Versuch verwendeten Entwurfes grundsätzlich durch entsprechende Entwurfs-Metriken messen lässt.

An dieser Stelle sei noch eine kritische Anmerkung gemacht, die die gewonnenen Erkenntnisse jedoch nicht schmälert.

Der Versuchsaufbau orientiert sich an der Programmiersprache C++. Dies passt zu dem während der Entwicklung vorgenommenen Abgleich der gewählten Metriken an aufeinanderfolgenden Versionen realer Applikations-Frameworks (Microsoft Foundation Classes und Borland Object Windows Library). Eine Besonderheit von C++ (und C#) ist jedoch, dass Methoden explizit als virtuell (virtual) deklariert werden müssen, damit sie dynamisch gebunden werden und somit den sinnvollen Einsatz von Polymorphie erst ermöglichen. Die Entwurfseigenschaft *Polymorphie* ist leider speziell auf diese Klasse von Sprachen zugeschnitten. Da z.B. in Eiffel und Java alle Methoden implizit als virtuell behandelt werden, ist bei Entwürfen mit diesen Zielsprachen die Bestimmung der Eigenschaft "Polymorphie" nicht notwendigerweise möglich.[1]

Fazit

Ein qualitativ hochwertiger Entwurf kann als notwendige Bedingung für hochqualitative Software gesehen werden. Hinreichend ist ein guter Entwurf leider nicht.

Bei der Betrachtung der Softwarequalitätsanforderungen scheinen sich Kategorien wie Benutzerfreundlichkeit, Funktionalität oder Effizienz der Inspektion im Entwurf zu entziehen. Sie lassen sich nicht anhand eines isoliert vorliegenden Entwurfes ablesen sondern ergeben sich erst aus dem Verhältnis des Entwurfes zu seinem Kontext. Das Maß der Funktionalität beispielsweise lässt sich erst durch die Gegenüberstellung mit den Anforderungen ermitteln.

Dass sich einige Bereiche der Softwarequalität im Entwurf nicht direkt ablesen lassen liegt auch darin begründet, dass sich deren Inhalte häufig nicht mit vertretbarem Aufwand formalisieren lassen. Diese Qualitätsanforderungen im Entwurfsmodell zu repräsentieren ist dadurch erschwert.

[1] In Java ist die Bestimmung möglich, falls alle Methoden explizit als `final` deklariert sind, die nicht polymorph sein sollen. Ob die Ergebnisse mit denen eines C++-Entwurfes vergleichbar sind und ob das Polymorphie-Maß in Java noch große Aussagekraft besitzt, ist jedoch zweifelhaft.

2.2 Aspekte im Entwurf

2.2.1 Warum Aspekte in den Entwurf aufnehmen?

Die Aspektorientierung ist zunächst eine Technik, die aus der Programmierung heraus entstanden ist. Sie wurde entwickelt, um dem Problem des Scattering und Tangling zu begegnen. Die Motivation, Aspekte auch im Entwurf einzuführen, hat verschiedene Gründe, die im Weiteren näher erläutert werden.

Entwicklungsprozess

Nahtlosigkeit

Mit Blick auf den Entwicklungsprozess lassen sich einige Anforderungen identifizieren, die die Verwendung von Aspekten auf der Ebene des Entwurfes fordern. Eine angestrebte Eigenschaft von Entwicklungsprozessen ist deren *Nahtlosigkeit*. Wenn eine neue Technik Einzug in die Implementierung hält, so muss der Einfluss auf die Nahtlosigkeit beim Übergang vom Entwurf zur Implementierung untersucht werden. Techniken wie z.B. der Einsatz von Inline-Assembler würden diese Nahtlosigkeit nicht beeinflussen. Da die Aspektorientierung jedoch in erheblichem Maße die *Struktur* der Software betreffen kann, wird die Nahtlosigkeit geschmälert. Die Nahtlosigkeit der einzelnen Schritte des Entwicklungsprozesses ist ein Motivationsfaktor, Aspekte auch im Entwurf zu berücksichtigen.

Nachvollziehbarkeit

Hinzu kommt der Wunsch nach einer möglichst guten *Nachvollziehbarkeit* (Traceability). Die Nachvollziehbarkeit fordert, dass durch den gesamten Entwicklungsprozess hindurch ersichtlich ist, in welcher Beziehung die Artefakte der verschiedenen Prozessschritte zueinander stehen. In besonderem Maße trifft dies auf die Beziehungen der Artefakte der Anforderungsanalyse zu den Artefakten des Entwurfes bis zu denen der Implementierung zu. Da Aspekte die Struktur der Software betreffen, sind sie auch der Nachvollziehbarkeit halber in den Entwurf aufzunehmen.

inkrementelle Entwicklung

Round-trip Engineering

Um eine *inkrementelle Entwicklung* und effektives *Round-trip Engineering* umzusetzen, sind ebenso Ausdrucksmittel für die aspektorientierten Konzepte auf der Stufe des Entwurfes gefordert. Zum einen wiederum, weil Aspekte die Struktur der Software prägen können. Zum anderen, weil es – wie für Klassen – Mittel geben muss, diese auf einer abstrakteren und undetaillierteren Ebene zu beschreiben und weiterzuentwickeln.

Entwurfsaktivität

Werkzeug im Entwurf

Von der Entwurfsaktivität an sich aus betrachtet sprechen ebenfalls mehrere Argumente für die Einbeziehung der Aspekte in den Entwurf. Da Aspekte Mittel zum Separation of Concerns darstellen und eine weitere Modularisierung ermöglichen, sind sie auch als *Werkzeug im Entwurf* von Nutzen. Auf sie treffen alle Vorteile des bereits

beschriebenen Modularisierungskonzeptes zu. Insbesondere können Aspekte helfen, die *Komplexität zu reduzieren*.

Reduktion der Komplexität

Die Verfügbarkeit von Aspekten im Entwurf kann der *Dokumentation* der Software hilfreich sein und ihre Erlernbarkeit erleichtern. Aspekte können auf Entwurfsebene auch *besser kommuniziert werden* als auf der Ebene der Implementierung. Dies rührt zum einen daher, dass im Entwurfsprozess ausgiebig von Visualisierungen Gebrauch gemacht wird. Zum anderen helfen dabei die im Entwurf üblichen unterschiedlichen Sichten, die je nach Intention jeweils auf bestimmte Details verzichten.

Dokumentation

Kommunizierbarkeit der Aspekte

Diese Eigenschaften befördern auch die *Wiederverwendbarkeit* von Aspekten. Daneben ist die gute Kommunizierbarkeit von Aspekten – und damit die Verfügbarkeit im Entwurf – eine Grundvoraussetzung für denkbare *CAOTS* (components and aspects off-the-shelf[1]).

Wiederverwendbarkeit

CAOTS

Fazit

Gründe für die Verwendung von Aspekten im Entwurf gibt es demnach mehr als genug. Letztendlich können Aspekte einmal so selbstverständliche Elemente des Entwurfes werden, wie es derzeit Klassen und Beziehungen sind.

2.2.2 Aspekt-Arten im Entwurf

Bei genauerer Betrachtung besitzen verschiedene Aspekte für den Entwurf durchaus unterschiedliche Bedeutung. Allen Repräsentationen von Aspekten im Entwurf ist gemein, dass sie zu deren Implementierung führen sollen. Darüber hinaus scheint es zwei Gruppen von Aspekten zu geben, die sich durch die Qualität der in ihnen enthaltenen Aspekte in Bezug auf den Entwurf unterscheiden.

Simple Aspekte

Auf der einen Seite gibt es Aspekte, die sehr einfach aufgebaut sind. Diese Aspekte betreffen zwar häufig viele Klassen und sind ohne Frage nützlich, um das Scattering und Tangling des realisierten Aspektes zu verhindern. Doch obwohl sie viele Klassen betreffen, bleiben diese Aspekte auf eine bestimmte Art substanzlos.

Ein Beispiel dafür ist das Logging. Dieser bereits klassische Aspekt ist von seiner Funktionalität her äußerst simpel. Die Einfachheit zeigt sich nicht in dem geringen, bei seiner Ausführung zu veranschlagenden Aufwand, sondern primär durch seinen Aufbau. Die Funktionalität des Loggings ist für die jeweils betroffenen Klassen *lokal*, es wird keine komplexe Funktionalität im Sinne eines Zusammenspiels unterschiedlicher, miteinander interagierender Klassen bewirkt.

lokale Funktionalität

[1]Diese Bezeichnung wurde von Constantinides und Skotiniotis (2002) übernommen.

Auch die von Kiczales u. a. (2001a) als Appetitanreger auf AspectJ beschriebene Gruppe der so genannten Entwicklungs-Aspekte fallen in diese Kategorie. Sie sind wie ihr Name andeutet insbesondere bei der Entwicklung hilfreich und ermöglichen einen leichten Einstieg in aspektorientierte Techniken, da sie nur während der Entwicklung aktiviert sind, aber im ausgelieferten Produkt entfallen können. In diese Menge fallen Aspekte wie Tracing oder das von Meyers Eiffel populär gemachte Design By Contract. Auch diese Aspekte helfen, das Crosscutting und Tangling zu verhindern, aber sie sind analog zu den vorigen Aspekten sehr simpel aufgebaut, wiederum in dem Sinne, dass ihre Funktionalität nur sehr *lokal* ist.

Den bisher erwähnten Aspekten ist zusätzlich gemein, dass sie auch intern sehr unkompliziert aufgebaut sind. Sie sorgen demnach weder für ein substanzielles Zusammenspiel externer Klassen noch bringen sie selbst eine komplexe interne Struktur mit um ihre Aufgabe zu erfüllen.

Komplexe Aspekte

Im Gegensatz zu den zuvor beschriebenen Aspekten weisen komplexe Aspekte beziehungsreichere Strukturen auf. Charakteristischerweise steuern sie eine substanzielle Funktionalität für die zu entwickelnde Software bei.

Beispiele dafür sind die Synchronisation oder ein Aspekt, der das Updaten der Darstellung einer Model-View Architektur übernimmt. Der Synchronisations-Aspekt trägt maßgeblich zur Funktionalität eines Systems bei. Zusätzlich ist diese Funktionalität – im Gegensatz zu den simplen Aspekten – nicht lokal begrenzt: zur Verwirklichung der Synchronisation muss der Aspekt mit den zu synchronisierenden Elementen kommunizieren. Dadurch ist er in wesentlich stärkerem Maße Träger von Struktur.

Komplexe Aspekte stellen nicht nur die Vorstufe einer Implementierungstechnik zum Verhindern von Scattering und Tangling dar. Vielmehr drücken sie in besonderer Weise *Strukturen* aus und sind damit insbesondere für den Entwurf relevant, da Strukturen für diesen ein Hauptthema sind.

Eine weitere Motivation, komplexe Aspekte im Entwurf zu behandeln liegt in ihrer immanenten Komplexität. Gerade wenn sie zur Erledigung ihrer Aufgaben nicht oder nur in geringer Weise auf die Kommunikation mit externen Elementen angewiesen sind bedeutet dies, dass sie die benötigten Dienste selbst mitbringen müssen. Damit sind sie ein klassischer Kandidat für die schrittweise Verfeinerung im Entwurf, analog zum Vorgehen bei Klassen, die ihre Funktionalität aus inneren Klassen komponieren.

Bindungen

Zur Unterscheidung der Aspekt-Arten wurden die für das Inkraftsetzen der Aspekte nötigen Bindungen bisher nicht herangezogen. Aber auch die Bindungen der simplen und komplexen Aspekte streben in zwei Richtungen auseinander.

Simple Aspekte besitzen durch ihre lokale Funktionalität häufig kleine Schnittstellen. Dies macht sie von der technischen Seite betrachtet recht allgemein einsetzbar und erfordert wenig Aufwand beim Erstellen der Bindungen. Aufwendiger wird es jedoch, wenn ein simpler Aspekt tatsächlich global eingesetzt werden soll. Dann kann die Bindung allein durch die schiere Menge der hergestellten Verbindungen unübersichtlich werden.

Komplexe Aspekte stellen höhere Anforderungen an ihre Bindungen, die von ihren größeren Schnittstellen und der stärkeren Einbindung in die Funktionalität des Systems herrühren. Hier ist es nicht primär die Quantität der Bindungen, die zur Komplexität beiträgt, sondern deren Substanz. Je größer die Schnittstellen sind, desto wahrscheinlicher müssen auch die Bindungen deren Unterschiede überbrücken.

Fazit

Es scheint Aspekte unterschiedlicher Qualität zu geben, die hier mit simpel und komplex bezeichnet wurden. Beide Aspektarten können im Entwurf mit der Berechtigung berücksichtigt werden, dass ihnen als Technik der Programmierung auch eine Repräsentation in früheren Phasen des Entwicklungsprozesses zusteht.

Wirklich interessant sind für den Entwurf jedoch die komplexen Aspekte. Dies ist der Fall, da sie in ihrem Inneren komplex aufgebaut sein können oder ihre Komplexität von der durch ihre Kommunikation mit externen Elementen induzierten Struktur herrührt. Dadurch repräsentieren sie im Entwurf nicht nur einen späteren Teil der Implementierung, sondern sind selbst Objekt des Entwurfes und nehmen wie Klassen am Entwurfsprozess teil, werden verfeinert usw.

Bindungen fordern ihre Berücksichtigung im Entwurf zum einen durch ihre mögliche große Anzahl. Größere Bedeutung kommt allerdings den Bindungen der komplexen Aspekte im Entwurf zu, da sie durch die größeren zu überbrückenden Unterschiede ebenfalls komplexer sind.

2.3 Aspectual Collaborations im Entwurf

Aspectual Collaborations stellen wie in Abschnitt 1.7 beschrieben Module zum Kapseln kollaborativen Verhaltens dar. Damit stellen sie genau solche Mittel bereit, die sich zum Formulieren von komplexen Aspekten eignen.[1]

Die Berücksichtigung von Aspectual Collaborations im Entwurf kann nur ein erster Schritt sein und wird auch dazu dienen, mehr über den Umgang mit Aspekten zu lernen. Es müssen erst Erfahrungen gesammelt werden, um die genaue Rolle von

[1] Simple Aspekte können natürlich auch formuliert werden, sie stellen quasi eine Untermenge der komplexen Aspekte dar und besitzen eine besonders einfache Struktur.

Aspectual Collaborations im Entwurf zu erarbeiten. Diese neuen Erfahrungen werden auch Einfluss auf die verwendeten Entwurfs-Methoden nehmen und der Betrachtung von Klassen analoge Vorgehensweisen und Heuristiken hervorbringen.

2.3.1 Entwurfs-Szenarien

Die Herangehensweise an den Entwurf mit Aspectual Collaborations hängt von der konkreten Einsatzsituation ab, da von ihr bestimmte Vorgaben gemacht werden. Diese Randbedingungen beeinflussen insbesondere zwei Dimensionen beim Entwurf mit Aspectual Collaborations: die Funktionalität der Kollaboration und ihre Struktur.

Freiheitsgrade

Es lassen sich vier grundsätzliche Situationen identifizieren, die sich unterschiedlich auf die für den Entwurf der Kollaboration zur Verfügung stehenden *Freiheitsgrade* auswirken (Tab. 2.3). Je größer der Freiheitsgrad ist, desto umfangreichere Entscheidungen können in der jeweiligen Dimension beim Entwerfen getroffen werden.

Gerade am Beginn des Einsatzes von Aspectual Collaborations kann es durchaus hilfreich sein, bestimmten Restriktionen ihrer Verwendung ausgesetzt zu sein. Dies ist der Fall, weil die Beschränkungen auch ein Gerüst darstellen und durch sie weniger Entscheidungen gefällt werden müssen (zu deren Treffen noch Erfahrungen fehlen).

Szenario	**Funktionalität**	**Struktur**
Erweiterung	⊖	⊖
Refactoring	⊕	⊖
CAOTS	⊖	⊕
Erstentwicklung	⊕	⊕

Tabelle 2.3: Freiheitsgrade für Aspectual Collaborations

Erweiterung

Bei der *Erweiterung* eines bestehenden Systems werden beide Freiheitsgrade durch immanente Vorgaben restringiert. Da die Erweiterung zu einem bestimmten Zweck erfolgt, ist die (hier durch Aspectual Collaborations) zu realisierende Funktionalität bereits vorgegeben. Auf die Struktur, mit der eine Kollaboration zusammenarbeiten wird, kann ebenso wenig Einfluss genommen werden, da diese bereits existiert und nicht verändert werden soll.

Refactoring

Das *Refactoring* eines Systems macht im Gegensatz zur Erweiterung zunächst keine Vorgaben bezüglich der in der Kollaboration zu realisierenden Funktionalität. Natürlich muss die Gesamtfunktionalität des Systems erhalten bleiben, aber welche Teile dieser Funktionalität im einzelnen nach dem Refactoring von einer Kollaboration übernommen werden, bleibt dem Entwickler überlassen. Die Entscheidungsgrundlage dafür liegt in der vorgefundenen Struktur der Anwendung.

CAOTS

Bei der Entwicklung von *CAOTS* (commercial aspects and components off-the-

shelf) ist die Wahl der Funktionalität durch ihren Anwendungszweck gebunden. Der Struktur der Kollaboration werden allerdings keine konkreten Vorgaben gemacht.

Erstentwicklung

Die *Erstentwicklung* von Systemen bietet den größten Entscheidungsspielraum bezüglich der Funktionalität und der Struktur von Aspectual Collaborations und würde am stärksten von speziellen Entwurfsrichtlinien profitieren.

2.4 UML für Aspekte: UFA

UFA (Herrmann, 2002a) ist eine Erweiterung der UML, die es ermöglicht, Aspekte zu modellieren, die auf den Konzepten der Aspectual Collaborations basieren. Die wichtigsten Elemente des Aspectual Collaborations Modells (vgl. Abschnitt 1.7) sind *Kollaborationen* und *Rollen* sowie *Konnektoren*. Für diese Elemente stellt UFA graphische Repräsentationen zum Modellieren des Softwareentwurfes bereit. Da der Softwareentwurf für gewöhnlich schrittweise, durch die Verfeinerung eines zunächst abstrakten Modells erfolgt, werden diese Darstellungen von UFA auch in verschiedenen Detailstufen angeboten.

Pakete repräsentieren Kollaborationen

Kollaborationen werden in UFA durch eine erweiterte Form von UML-Paketen repräsentiert. Da in einer Kollaboration verschiedene Rollen beschrieben werden, die man auch als unvollständige, nur auf einen Aspekt hin implementierte Klassen betrachten kann, bieten sich Pakete für diese Darstellung an. Die Besonderheit, dass in einer Kollaboration keine vollständige Implementierung geliefert werden muss, drückt sich darin aus, dass diese Pakete mit der Eigenschaft `{abstract}` markiert werden.

In Abbildung 2.4 ist das Beispiel der Flugwetterüberwachung aus Abschnitt 1.7 auf abstraktem Niveau dargestellt. Die Kollaboration mit der Synchronisationsfunktionalität ist durch das abstrakte Paket `MutexSync` repräsentiert und die Details der einzelnen Pakete sind ausgeblendet.

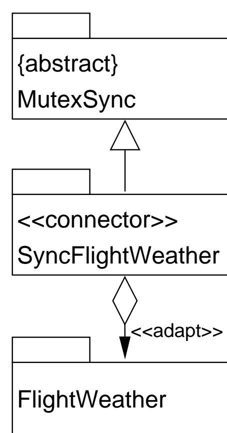


Abbildung 2.4: Anwendung der MutexSync-Kollaboration auf das Flight-Weather Paket

Pakete besitzen Attribute und Methoden

Zusätzlich zu den Eigenschaften von reinen UML Paketen dürfen UFA Pakete auch eigene Attribute und Methoden besitzen und an Vererbungsbeziehungen teilnehmen. Attribute und Methoden werden – analog zu Klassen – in eigenen, durch eine horizontale Linie abgetrennten Bereichen dargestellt. Wenn ein Paket von einem anderen Paket erbt, so erbt es alle Eigenschaften des Elternpaketes und die in ihm enthaltenen Klassen. Um eine geerbte Klasse zu spezialisieren, reicht es aus, eine Klasse des gleichen Namens mit den zusätzlichen Eigenschaften in das Kindpaket aufzunehmen.

Paket-Vererbung

Der Konnektor in Abbildung 2.4 spezialisiert das Kollaborations-Paket `MutexSync` für die Anwendung auf das `Flight-Weather` Paket.

Adaptions-Beziehung

Die *Adaptions-Beziehung* zwischen Paketen ist eine weitere neu eingeführte Beziehung. Sie drückt aus, dass das adaptierende Paket ein anderes benutzt und beeinflusst. Da diese Beziehung gerichtet ist, hat das adaptierte Paket keine Kenntnis von einer eventuellen Adaption.

Im Beispiel adaptiert der Konnektor das `FlightWeather` Paket ohne dessen Wissen und Zutun. Das Paket wird durch die Bindung an die Kollaboration derart verändert, dass der Zugriff mehrerer Klienten durch die Synchronisation der Resourcennutzung möglich wird.

Der in Abbildung 2.4 dargestellte Entwurf ist auf einer recht abstrakten Ebene angesiedelt, weil die Darstellung keine Details der Paket-Komposition erkennen lässt. Auf einer detaillierteren Stufe kann auch der Inhalt von Paketen entworfen werden. Dazu wird im folgenden Beispiel der im abstrakten Paket `MutexSync` gekapselte Synchronisations-Aspekt in einem weiteren Anwendungsgebiet eingesetzt.

Ein Reisebüro `TravelAgency` besitzt ein Buchungssystem, welches über ein Faxgerät die Reiseanbieter über erfolgte Buchungen informiert. Eine Synchronisation der Nutzung des Faxgerätes wird notwendig, da die Mitarbeiter des Reisebüros potenzielle Kunden nun auch per Fax über attraktive Reiseangebote informieren sollen. Um diese Werbemaßnahme vorerst zu testen, soll das für die Buchung verwendete Faxgerät auch dem Versand von Reiseangeboten dienen. Eine Randbedingung bei dieser Mitbenutzung ist, dass das Faxgerät zu drei Vierteln der Buchung dienen soll und nur maximal zu einem Viertel zum Informationsversand.

Die Synchronisation ist im Paket `MutexSync` bereits gelöst. Die Anforderung zum Führen einer Statistik der Nutzung kann sehr einfach im Konnektor geschehen, der das Reisebüro mit dem Synchronisations-Aspekt ausstattet.

Im Entwurf des Konnektors in Abbildung 2.5 sind die Bindungsdetails auf Klassenebene dargestellt. Die Rolle `MutexResource` wird von der Klasse `Fax` des adaptierten Paketes `TravelAgency` gespielt. Die Klientenrolle wird von mehreren, nicht miteinander verwandten Klassen übernommen: dem Buchungssystem und den Angestellten. `Client=` wird für die beiden nicht verwandten Klassen `BookingSystem` und `Employee` zu einem gemeinsamen Supertyp.

Die geforderte Nutzungsstatistik wird über Paket-Eigenschaften realisiert. Der

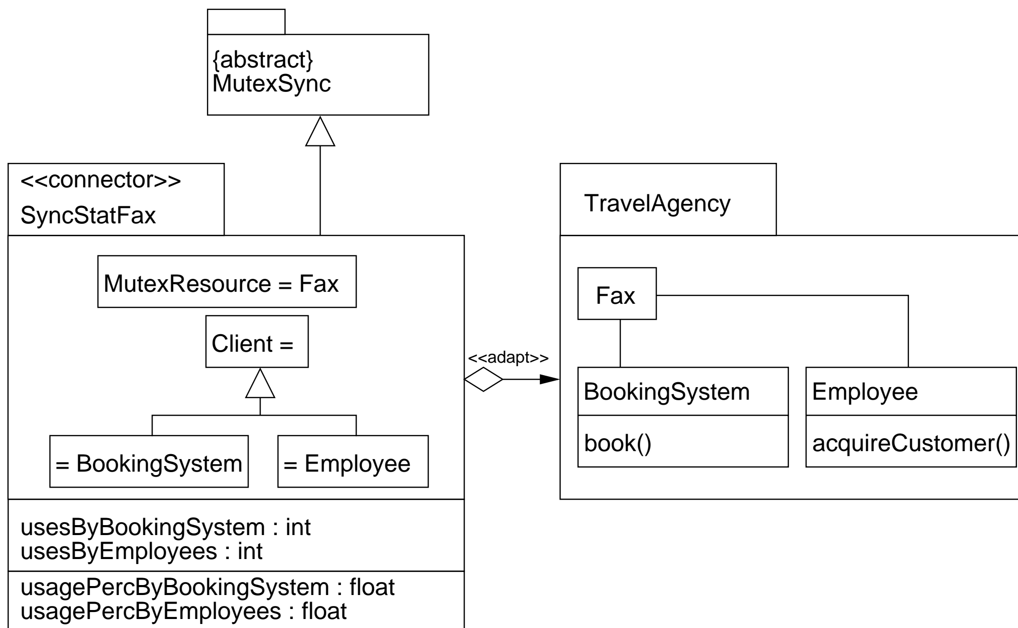


Abbildung 2.5: Details der Bindung des Synchronisations-Aspektes an das Reisebüro

Konnektor `SyncStatFax` verfügt über zwei Attribute zum Zählen der Faxnutzung. Außerdem stellt der Konnektor zwei Methoden bereit, die Auskunft über den prozentualen Anteil an der Nutzung des Faxgerätes des Buchungssystems bzw. der Mitarbeiter geben.

Fazit

In diesem Kapitel wurde deutlich, dass für die aspektorientierte Softwareentwicklung neben der Weiterentwicklung von aspektorientierten Techniken auf der programmiersprachlichen Ebene auch die Berücksichtigung der Aspektorientierung in früheren Phasen des Entwicklungsprozesses nötig ist. Es wurden Überlegungen zur Rolle von Aspekten im Entwurf gemacht und erkannt, dass insbesondere komplexe Aspekte dort repräsentiert werden sollten. Schließlich wurde mit UFA eine geeignete Entwurfsnotation für Aspekte basierend auf dem Aspectual Collaborations Modell vorgestellt.

3 PIROL

3.1 Softwareentwicklungsumgebungen

Ein Ziel der Softwaretechnik ist es von je her, Methoden und Techniken zu entwickeln, die die Softwareproduktion möglichst effizient, planbar und kontrollierbar werden lassen. Was läge in diesem Fall näher, als eine passende Software zu entwickeln, die beim Erreichen dieses Zieles hilft?

Softwareentwicklungsumgebungen (SEU) sind demnach Systeme, die die Softwareentwicklung unterstützen sollen. Sie sollen die von den einzelnen Werkzeugen in den jeweiligen Phasen des Entwicklungsprozesses gebotene Funktionalität unter einem gemeinsamen Dach bereitstellen und für die Integration der Werkzeuge sorgen.

Mittlerweile existieren eine Vielzahl von Softwareentwicklungsumgebungen sowie Abkürzungen zu deren Klassifizierung, die jedoch je nach Autor unterschiedliche Bedeutungen tragen.

Genannt seien an dieser Stelle nur folgende: SEE (Software Engineering Environment), ISEE (Integrated Software Engineering Environment), IPSE (Integrated Process Support Environment), CASE (Computer Aided Software Engineering), ICASE (Integrated Computer Aided Software Engineering), SDE (Software Development Environment), ISF (Integrated Software Factory) und SLCSE (Software Life Cycle Support Environment).

Die Tatsache, dass zu den aufgeführten Begriffen keine exakten Definitionen vorliegen darf nicht davon ablenken, dass sich Softwareentwicklungsumgebungen inhaltlich erheblich voneinander unterscheiden können.

3.1.1 Charakteristika

unterstützte Methoden

Ein erstes Differenzierungsmerkmal von Softwareentwicklungsumgebungen ist das Vorhandensein bzw. Fehlen der Unterstützung unterschiedlicher *Entwicklungsmethoden*. Einige Umgebungen unterstützen lediglich einen spezifischen Prozess, wohingegen andere prozessunabhängig ausgelegt sind und der verwendeten Methode angepasst werden können. Die prozessunabhängigen Umgebungen unterscheiden sich wiederum im Umfang ihrer Anpassungsfähigkeiten.

abgedeckte Phasen

Softwareentwicklungsumgebungen unterscheiden sich auch darin, welche *Phasen* der Softwareentwicklung abgedeckt werden. Zum Beispiel werden Upper CASE Tools von Lower CASE Tools unterschieden. Erstere beziehen sich auf frühe Phasen des Entwicklungsprozesses wie Analyse und Entwurf, letztere auf die späteren Phasen wie Implementierung, Testen oder Reverse Engineering.

Gruppen-unterstützung

Ein weiteres Unterscheidungsmerkmal lässt sich in den beabsichtigten Benutzergruppen ausmachen. Es gibt Softwareentwicklungsumgebungen, die für die Verwendung von mehreren Nutzern in verschiedenen Gruppen und Rollen ausgelegt sind und solche, die lediglich Einzelnutzer unterstützen.

unterstützte Aktivitäten

Welche *Aktivitäten* des Prozesses werden durch Werkzeuge unterstützt? Denkbar sind Umgebungen, die sich auf die Kernaktivitäten der Entwicklung wie Analyse, Entwurf, Implementierung und Testen konzentrieren. Daneben können aber auch Projektmanagement-Aktivitäten, wie die Verwaltung der Projektresourcen (Personal, Kosten, Zeit etc.) oder Aktivitäten der Qualitätssicherung von einer SEU unterstützt werden.

Integrationsgrad

Ein fünftes Unterscheidungsmerkmal stellt die von der Softwareentwicklungsumgebung realisierte *Integration* dar, die im folgenden Abschnitt detaillierter betrachtet wird.

3.1.2 Integration

Seit der Entwicklung der ersten Softwareentwicklungsumgebungen ist die Integration der von der Umgebung umfassten Werkzeuge ein Kernanliegen. Die unterschiedlichen Werkzeuge sollen so zusammenarbeiten, dass die Softwareentwicklung möglichst nahtlos durch alle Phasen des Prozesses geführt wird. Die zyklische Natur der Softwareentwicklung stellt dabei besondere Anforderungen an den Umgang mit den erfassten Informationen, da diese in unterschiedlichen Phasen mit unterschiedlichen Werkzeugen (und differierenden Zielsetzungen) bearbeitet werden.

In einer Arbeit zur Werkzeugintegration hat Wasserman (1989) genauer untersucht, welche Arten der Integration unterschieden werden können. Die von ihm beschriebenen Integrationsarten sind bis heute gültig und finden sich in unterschiedlichen Arbeiten wieder, u.a. auch im ECMA Referenzmodell für Frameworks von Softwareentwicklungsumgebungen (ECMA TR/55, 1993). Letzterer Standard lässt die Plattformintegration jedoch außen vor und fügt ergänzend die Frameworkintegration hinzu.

Datenintegration

Unter Datenintegration wird die Möglichkeit verstanden, Informationen innerhalb der Entwicklungsumgebung von verschiedenen Werkzeugen aus gemeinsam zu nutzen.

Dazu gehört auch die Organisation des Umgangs mit Daten, die zwar nur von einem Werkzeug verwendet werden, aber in einer bestimmten Beziehung zu den restlichen Daten des Systems stehen. Insbesondere gilt es, die Konsistenz der Daten sicherzustellen und Redundanzen zu vermeiden.

Die Datenintegration kann auf unterschiedlichem Niveau realisiert sein. Bereits die Möglichkeit, Daten von verschiedenen Werkzeugen über das Speichern und Laden von Dateien auszutauschen, stellt eine Form der Datenintegration dar. Nötig ist dazu natürlich ein gemeinsames Datenschema oder das Vorhandensein geeigneter Export- und Importfilter, die die notwendigen Transformationen durchführen. Mächtigere Mittel zur Datenintegration sind beispielsweise Datenbanken oder spezielle Object Management Systeme.

Kontrollintegration

Idealerweise arbeiten die Werkzeuge einer Softwareentwicklungsumgebung untereinander Hand in Hand und unterstützen außerdem die flexible Kombination ihrer Funktionalitäten.

Ersteres bedeutet zum Beispiel, dass Werkzeuge in ihrer Kontrolle so integriert sind, dass eine Änderung von Daten in einem Werkzeug die anderen Werkzeuge zu entsprechenden Reaktionen veranlasst. Dies kann eine Benachrichtigung für den Nutzer sein oder lediglich die Aktualisierung der angezeigten Informationen.

Zur Kontrollintegration gehört ebenso, dass Werkzeuge die Dienste anderer Werkzeuge in Anspruch nehmen können. So könnte beispielsweise die Editor-Komponente eines Werkzeugs von anderen Werkzeugen verwendet werden, wenn Textinformationen zu manipulieren sind.

Die Kontrollintegration erfordert insbesondere die Kommunikation der Werkzeuge untereinander und setzt damit einen geeigneten Kommunikationsmechanismus voraus.

Prozessintegration

Prozessintegration beschreibt die Möglichkeit, die Funktionalität der in der Softwareentwicklungsumgebung gebündelten Werkzeuge in Abhängigkeit von einem definierten Entwicklungsprozess zu steuern.

Zu den einfachen Forderungen gehört zum Beispiel die Rechteverwaltung verschiedener Benutzer und Benutzergruppen. Aber auch komplexere, im Prozess festgeschriebene Abläufe, wie das automatische Weiterleiten einer zum Testen freigegebenen Komponente an einen für diesen Teil des Systems qualifizierten Mitarbeiter, der freie Kapazitäten hat und sich im Augenblick nicht am Strand in der Sonne räkelt.

Präsentationsintegration

Die Werkzeuge einer Softwareentwicklungsumgebung sollten auch in Hinsicht auf ihre Benutzeroberflächen und die verwendeten Interaktionsformen aufeinander abgestimmt sein.

Die Präsentationsintegration erleichtert zum einen das Erlernen des Umgangs mit den unterschiedlichen Werkzeugen, da in einem Werkzeug gesammelte Erfahrungen auf die Funktionsweise der anderen übertragen werden können. Zum anderen wird die Nutzung der Werkzeuge effizienter und wirkt weniger belastend, da beim Wechsel des verwendeten Werkzeugs während der Arbeit keine Vergegenwärtigung der genauen Funktionsweise erfolgen muss. Nicht zuletzt hat die gelungene Gestaltung von Benutzerschnittstellen heute deutlichen Einfluss auf die Produktakzeptanz gewonnen.

Frameworkintegration

Die Frameworkintegration bezieht sich darauf, inwieweit ein Framework seine Nutzung durch geeignete Mechanismen unterstützt. Ist es möglich, Werkzeuge hinzuzufügen und unbenötigte zu entfernen? Kann die dem Benutzer von den Werkzeugen bereitgestellte Funktionalität dem Benutzerstatus entsprechend angepasst werden? Werden neue Werkzeuge so integriert, dass sie mit den bereits vorhandenen zusammenarbeiten können?

3.2 PIROL

3.2.1 Integration

PIROL (Project Integrating Reference Object Library) ist ein an der TU Berlin entwickeltes Framework für Softwareentwicklungsumgebungen (Groth, Herrmann, Jähnichen und Koch, 1995). Es unterstützt die Erstellung integrierter Entwicklungsumgebungen zur Entwicklung objektorientierter Software und stellt Mittel zur Daten-, Kontroll- und Präsentationsintegration bereit[1].

Die offene Struktur macht vielfältige Erweiterungen möglich. Dies können zum einen neue Werkzeuge sein, zum anderen aber auch auf tieferen Ebenen liegende Erweiterungen wie beispielsweise die Unterstützung für einen spezifischen Prozess.

Datenintegration

Repository

Den Kern von PIROL bildet das so genannte *Repository*, welches alle in der Ent-

[1] Die für die Prozessintegration benötigten Mechanismen werden bereits durch die Kontrollintegration bereitgestellt

wicklungsumgebung relevanten Informationen enthält. Das Repository übernimmt die Aufgabe, diese Daten zu speichern und sichert darüber hinaus die Konsistenz der Daten, um insbesondere das Arbeiten mehrerer Benutzer zu ermöglichen.

strukturelle Dekomposition

PIROL bedient sich dazu des Konzeptes der *strukturellen Dekomposition* (Abb. 3.1). Danach bestehen alle Dokumente aus einer Menge atomarer Datenelemente im Repository, die als Referenz-Objekte (RO) bezeichnet werden.

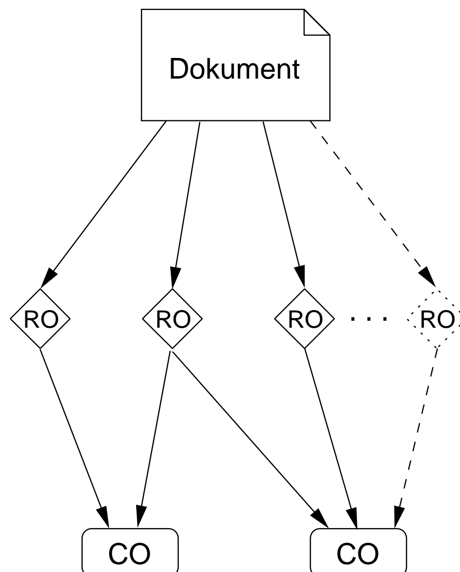


Abbildung 3.1: Strukturelle Dekomposition in ROs und deren Aggregation zu COs

unterschiedliche Sichten

Um unterschiedlichen Werkzeugen *unterschiedliche Sichten* auf das Repository und seine Inhalte zu ermöglichen, werden Referenz-Objekte zu konzeptuellen Objekten (CO) aggregiert. COs repräsentieren für die Werkzeuge also die Dokumente und sind aus den Daten der ROs aufgebaut. Ein einzelnes RO kann Teil mehrerer Sichten auf die Daten des Repositorys sein und sorgt damit für die angestrebte Konsistenz.

Metamodell

Die Basis für die Datenintegration ist ein *Metamodell*, welches öffentlich ist und von den verschiedenen Werkzeugen gemeinsam genutzt werden kann. Das Metamodell selbst ist ein objektorientiertes Klassenmodell und definiert Meta-Klassen der objektorientierten Softwareentwicklung, wie zum Beispiel CLASSIFIER, SUBSYSTEM, RELATION oder INSTANCE.

In der Nutzung durch die Werkzeuge unterscheidet sich das Repository von einer relationalen Datenbank dahingehend, dass im Metamodell nicht nur die statische Struktur definiert wird, sondern auch das Verhalten der Objekte der Meta-Klassen. Damit ähnelt es objektorientierten Datenbank-Management Systemen (OODBMS), bietet aber noch einige zusätzliche Eigenschaften, auf die in Abschnitt 3.2.2 eingegangen wird.

Der Zugriff auf das Repository erfolgt durch die Werkzeuge nie direkt, sondern vermittelt durch die *Workbench*. Diese enthält einen Interpreter für die Repository Sprache Lua/P (Herrmann, 2000) und stellt damit quasi ein ausführbares Metamodell zur Verfügung. Für jede Anwendersitzung wird eine Workbench gestartet, die alle von diesem Nutzer verwendeten Werkzeuge bedient.

Workbench

Die Verwendung der *Repository Sprache Lua/P* trägt entscheidend dazu bei, dass Werkzeuge flexibel in die Umgebung integriert werden können. Über die in Lua/P realisierten Konnektoren lassen sich beispielsweise auch Werkzeuge integrieren, deren zugrundeliegendes Modell sich von dem in der Entwicklungsumgebung verwendeten Modell unterscheidet, ohne die zu integrierenden Modelle dabei selbst ändern zu müssen.

Repository Sprache Lua/P

Kontrollintegration

Wie beschrieben erfordert die Kontrollintegration insbesondere einen Kommunikationsmechanismus, damit die Werkzeuge miteinander arbeiten können. In PIROL kommt ein *Nachrichtenserver* zum Einsatz, der ein Multicast-Protokoll bereitstellt und von den Clients per Socketverbindung angesprochen wird. Die Zugriffsfunktionen werden den Clients durch eine Bibliothek bereitgestellt. Sowohl die Werkzeuge als auch die Workbench sind Clients des Nachrichtenservers, der sich ihnen gegenüber transparent verhält (Abb. 3.2).

Nachrichten-server

Bei Änderungen an den Repository-Objekten sendet die Workbench eine entsprechende Nachricht auf dem Nachrichtenkanal. Den einzelnen Werkzeugen obliegt es, sich für die Benachrichtigung über Änderungen an den von ihnen bearbeiteten Objekten zu registrieren.

Werkzeuge, die in Java implementiert werden, können auf eine Bibliothek von Proxyklassen zurückgreifen, die die Klassen des Metamodells kapselt und die Kommunikation mit der Workbench übernimmt. Werden Änderungen an den Proxyobjekten vorgenommen, so wirken sich diese direkt auf die in der Workbench enthaltenen Basis-Objekte aus.

Werkzeuge kommunizieren nie direkt miteinander, sondern immer vermittelt durch die Workbench. Das PIROL Metamodell enthält ein Paket TOOLS, welches die installierten Werkzeuge repräsentiert. Über die Workbench ist dann der Zugriff auf die installierten und eventuell auch gerade aktiven Werkzeuge möglich.

Metamodell Repräsentation-der Werkzeuge

Prozessintegration

Die grundlegende Prozessintegration wird über Klassen des PIROL Metamodells ermöglicht. Die Metamodell-Klassen GROUP, PERSON und ROLE stellen die Mittel dar, um Benutzergruppen und deren Mitglieder mit ihren jeweils eingenommenen

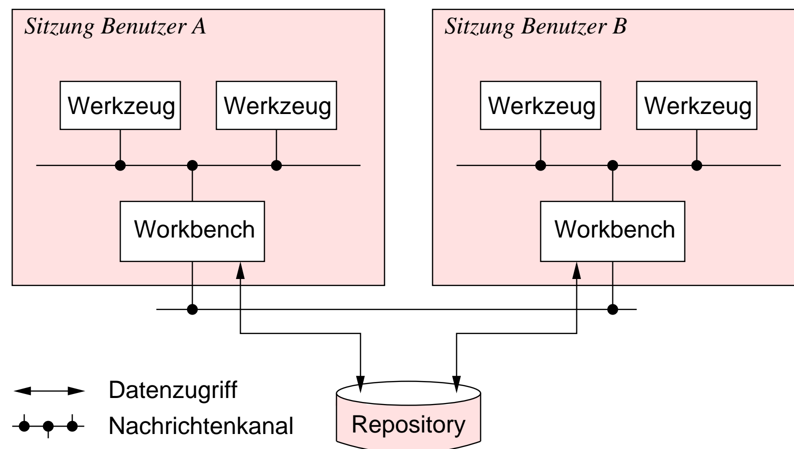


Abbildung 3.2: PIROL Architektur

Rollen zu repräsentieren. Über die Klassen STATE, TRANSITION und PERMISSION_GUARD werden Zustände und Transitionen sowie Wächter, die die Zulässigkeit der Transitionen sicherstellen, widergespiegelt. Im Rahmen einer Diplomarbeit von Sven Siewczynski wird derzeit ein Projekt-Management Werkzeug entwickelt, welches auch Verbesserungen des Metamodells an diesem Punkt mit sich bringt.

Präsentationsintegration

Die Präsentationsintegration ist bisher kein Schwerpunkt in der Entwicklung von PIROL gewesen, so dass es noch keinen Standard für die graphischen Oberflächen gibt.

Ein für die integrierte Präsentation der Werkzeuge nützliches Instrument wird jedoch von der Workbench in Form eines *Kontextmenüs* bereitgestellt. Es kann von den Werkzeugen zu jedem Repository-Objekt angefordert werden und wird dem RO entsprechend konfiguriert.

Kontextmenü

Über dieses Menü sind Workbench-Kommandos wie `select`, `replace`, `print` und `SendTo` zu erreichen, die somit prinzipiell in allen Werkzeugen zur Verfügung stehen. Vom genauen Typ des gewählten ROs hängt z.B. ab, welche Einträge in das Submenü `Launch` aufgenommen werden, so dass dort nur passende Werkzeuge zum Starten angeboten werden.

Frameworkintegration

In PIROL werden Werkzeuge als TOOL-Objekte des Metamodells von der Workbench verwaltet. Dem zur Verfügung stehenden Werkzeug-Pool können jederzeit Werkzeuge hinzugefügt oder entnommen werden. Durch ihre Repräsentation im Me-

tamodell sind auch nachträglich installierte Werkzeuge vollständig in die Umgebung integriert. Sie können von der Workbench aus gestartet werden und bieten sich für geeignete RO-Typen zu deren Bearbeitung an.

3.2.2 Besondere Eigenschaften

In Abschnitt 3.2.1 wurde bereits auf die Verwandtschaft des PIROL-Repositorys zu OODBMS hingewiesen. Einige der zusätzlichen Eigenschaften sollen hier kurz erläutert werden, da sie auch im Implementierungsteil dieser Arbeit zum Einsatz kamen.

Bewachte Attribute

guarded attributes

Bewachte Attribute (*guarded attributes*) sind ein Mittel, um die Datenintegrität im Repository sicherzustellen. Wird zu einem Attribut ein Wächter (guard) definiert, so bekommt dieser die Kontrolle beim Auslesen des Attributes und bei Zuweisungen an das Attribut übergeben. Der Wächter kann dann entscheiden, ob das Attribut gesetzt wird und es steht ihm frei, beliebige Aktionen auszuführen. Dies können reine Inspektionen des Repositorys sein, aber ebenso gezielte Seiteneffekte, die sich auch auf Programme außerhalb des Repositorys beziehen können. Der Wert könnte einfach gesetzt, vor dem Setzen geeignet verändert oder ignoriert und der Aufruf mit einem Fehler quittiert werden.

Abgeleitete Attribute

derived attributes

Abgeleitete Attribute (*derived attributes*) dienen dazu, Redundanzen im Repository zu vermeiden ohne gleichzeitig auf die Eigenschaft von Attributen verzichten zu müssen, dass deren Zustand überwacht werden kann.

An bestimmten Stellen eines Datenmodells stellt sich die Frage, ob eine Eigenschaft einer Klasse besser als Attribut oder als Funktion realisiert werden sollte, wobei beide Wahlmöglichkeiten an dieser Stelle Nachteile mit sich bringen.

Dies ist vor allem bei Attributen der Fall, die sich aus mehreren anderen Attributen zusammensetzen, wie z.B. eine Methoden-Signatur. Die Signatur einer Methode setzt sich aus dem Methodennamen, deren Argumenten und deren Ergebnistyp zusammen.

Wird die Signatur der Methode als Attribut der Klasse realisiert, so enthält sie Informationen, die bereits an anderer Stelle vorhanden sind und führt damit unerwünschte Redundanzen ein.

Demgegenüber steht die Möglichkeit, die Signatur als Funktion zu realisieren, die den Wert aus den entsprechenden Attributen berechnet. Allerdings gibt es in diesem Fall keine Möglichkeit mehr, sich über Zustandsänderungen der Signatur informieren zu lassen, da diese ja aus einer Berechnung hervorging.

Abgeleitete Attribute bringen diese beiden Pole zusammen. Zum einen sind sie nicht persistent und vermeiden dadurch Redundanzen. Zum anderen können sie wie Attribute angemeldete Observer informieren, wenn sich ihr Zustand geändert hat.

3.2.3 Einbindung neuer Werkzeuge

Zur Einbindung neuer Werkzeuge in PIROL können Maßnahmen auf unterschiedlichen Ebenen nötig sein, die im Folgenden beschrieben werden.

Metamodell

Auf dieser untersten Ebene muss sichergestellt sein, dass das Metamodell geeignete Klassen für die von den neuen Werkzeugen verarbeiteten Informationen bereit hält. Für die Repräsentation der herkömmlichen objektorientierten Strukturen wie Klassen, Schnittstellen, Attributen oder Methoden kann auf bereits existierende Meta-Klassen zurückgegriffen werden.

Sollen mit einem Werkzeug jedoch neue Konzepte unterstützt werden, die über die bekannten objektorientierten hinausgehen, so muss das Metamodell entsprechend angepasst werden. Dies kann durch die Änderung bestehender Klassen geschehen oder durch das Hinzufügen neuer Klassen, die sinnvollerweise bereits implementierte Basiseigenschaften durch Vererbung für sich nutzen sollten.

Dynamische Konnektoren

Eingangs wurde auf das Prinzip der strukturellen Dekomposition hingewiesen, welches unterschiedliche Sichten auf die Daten ermöglicht und primär dem Konsistenzerhalt dient (s. 3.2.1). Ein weiteres Mittel, um den Werkzeugen unterschiedliche Sichten auf die Informationen zu ermöglichen sind die in Lua/P realisierten dynamischen Konnektoren (Herrmann und Mezini, 2001c).

Insbesondere bei der Einbindung bereits bestehender Werkzeuge wird eine Lücke zwischen den auf Seite von PIROL und auf Seite des Werkzeugs verwendeten Datenmodellen klaffen. Im einfachsten Fall hat das Werkzeug Daten zu einem Objekt zu speichern, die auf der Meta-Ebene bisher nicht modelliert sind und womöglich nur für die Sicht dieses speziellen Werkzeugs auf die Informationen relevant sind. Dabei könnte es sich beispielsweise um die Darstellungsinformation eines grafischen Werkzeugs handeln, die die Liniendicke eines Objektes festlegt.

dynamic view connector

view object

Ein dynamischer Konnektor (*dynamic view connector*, DVC) kann die Sicht auf bestimmte Objekttypen mit zusätzlichen Informationen anreichern. Dies geschieht durch das so genannte Lifting, welches ein RO zu einem VO (*view object*) erhebt, das die zusätzlichen Eigenschaften besitzt. In der Verwendung unterscheiden sich die

durch einen DVC gelieferten VOs nicht von den gewöhnlichen ROs. Die zusätzlichen Daten sind ebenso persistent und werden dem RO bei Verwendung des DVCs einfach angeheftet. Da sich das RO aus Sicht der restlichen Werkzeuge (ohne den DVC) in keiner Weise verändert hat, müssen an diesen auch keine Modifikationen vorgenommen werden.

Java-Schnittstelle

Derzeit sind die meisten Werkzeuge für PIROL in Java implementiert. Ermöglicht wird dies durch eine Java-Klassenbibliothek, die den Zugriff auf die Objekte der Workbench kapselt. Für alle Klassen des Metamodells, die in der Workbench in Lua/P implementiert sind, existieren auf Java-Seite entsprechende *Proxyklassen*. Die Proxys besitzen die gleichen Methoden wie ihre Pendants in der Workbench und zusätzlich Getter- und Settermethoden für den Zugriff auf die Attribute.

Proxyklassen

Wie bei der Kontrollintegration in Abschnitt 3.2.1 beschrieben sendet die Workbench für alle Änderungen an Repository-Objekten eine Nachricht auf den Nachrichtenkanal. Für Java-Werkzeuge stehen die *Observer-Klassen* `ChangedPattern` und `ListChangeObserver` zur Verfügung, um sich für die Benachrichtigung über Änderungen an Repository-Objekten zu registrieren.

Observer-Klassen

Durch ein `ChangedPattern` kann zu einem bestimmten Attribut eines Repository-Objektes ein Callback installiert werden, der bei allen Änderungen des Attributes aktiviert wird.

Bei der Überwachung eines Listenattributes ist ein `ChangedPattern` Observer weniger sinnvoll, da er seinen Callback lediglich auslösen würde, wenn das Listenattribut selbst verändert wird[1]. In der Regel sind jedoch Änderungen an der Liste zu verfolgen, so dass an dieser Stelle ein `ListChangeObserver` zum Einsatz kommen sollte.

Ein Listen-Observer installiert folgende Callbacks, die bei den entsprechenden Veränderungen an der Liste aufgerufen werden: `do_append`, `do_delete`, `do_insert` und `do_replace`.

[1] vgl. den Unterschied zwischen Wert- und Referenz-Semantik

4 UFA-Editor

Da im Laufe dieser Arbeit das Aspectual Collaborations Modell zum *Object Teams* Modell (Herrmann, 2002b) weiterentwickelt wurde, wird im Folgenden das Vokabular von Object Teams verwendet. Der wichtigste Unterschied zu Aspectual Collaborations besteht darin, dass die Trennung von Kollaboration und Konnektor technisch nicht mehr notwendig ist, wenn sie auch methodisch wünschenswert bleibt. Ein Team umfasst damit die Eigenschaften von Kollaboration und Konnektor.

Object Teams

4.1 Anforderungsanalyse

4.1.1 Systembeschreibung

Ziel der Entwicklung ist ein grafischer Editor zum Erstellen und Bearbeiten von aspektorientierten Entwürfen, der möglichst nahtlos in die repository-basierte SEU PIROL integriert ist. Zur grafischen Repräsentation der zu bearbeitenden Entwürfe soll UFA (UML for Aspects) zum Einsatz kommen.

Funktionalität

Der Editor soll alle Elemente eines UFA-Entwurfes erstellen und bearbeiten können. Dazu gehört das Anlegen von Paketen und Teams sowie die Vererbung zwischen Teams und das Adaptieren von Paketen durch Teams.

Innerhalb von Teams und Paketen sollen Klassen und Rollen angelegt und bearbeitet werden können. Die vorgenannten Elemente sollen außerdem an Vererbungsbeziehungen teilnehmen können.

Da für einen aspektorientierten Entwurf in Form eines UFA-Diagramms auch textuelle Angaben benötigt werden (z.B. Klassen-, Rollen- und Methodennamen), die nicht wie das Platzieren von grafischen Elementen durch Aktionen mit der Maus erfolgen können, muss eine angemessene Bearbeitungsmöglichkeit für diese Elemente gefunden werden.

Integration in PIROL

Eine Kernanforderung an den zu entwickelnden Editor liegt in der nahtlosen Integration in die SEU PIROL. Es ist das Ziel, die von PIROL zur Datenintegration bereitgestellten Mechanismen vollständig zu nutzen und direkt auf den Repository-Daten zu arbeiten, um Redundanzen und die damit einhergehenden Konsistenzprobleme zu vermeiden.

Benutzeroberfläche

Der Editor soll eine zeitgemäße grafische Oberfläche bieten, die eine möglichst intuitive Bedienung ermöglicht. Dazu gehört die Verwendung von Standard-GUI Komponenten, die den Nutzern aus anderen Anwendungen dieses Typs bekannt sind.

Im Einzelnen sollen Befehle über Menüs zu erreichen sein und Tastatur-Shortcuts für die wichtigsten Aktionen existieren. Grafische Elemente sollen über eine Werkzeugleiste erreicht werden können und deren Erstellung möglichst intuitiv machen.

4.2 Erweiterung des PIROL Metamodells

Wie in Abschnitt 3.2.1 beschrieben wurde, basiert PIROLs Datenintegration auf einem Metamodell für die objektorientierte Softwareentwicklung. Um aspektorientierte Entwürfe in sinnvoller Weise im Repository repräsentieren zu können, muss das Metamodell geeignet erweitert werden.

Wichtig ist dabei, dass sich diese Erweiterung so in das Metamodell integriert, dass die von den bestehenden Werkzeugen verwendeten Strukturen nicht zerstört werden. Das bedeutet insbesondere, dass keine vorhandenen Features entfernt, verändert oder umbenannt werden. Erweiterungen an bestehenden Metamodell-Klassen sind hingegen ohne Einschränkungen möglich und sollen insbesondere zum Erhalt der Redundanzfreiheit vorgenommen werden.

Das Ziel der Metamodell-Erweiterung ist zwar, UFA Diagramme im Repository abspeichern zu können, jedoch sollen nicht alle von einem Editor für UFA Diagramme benötigten Informationen direkt im Metamodell repräsentiert werden. Vielmehr ist eine Erweiterung des Metamodells gefordert, die die Basiskonzepte von Object Teams modelliert und für alle Arten von Werkzeugen nutzbar ist. UFA-Diagramme stellen lediglich eine spezielle Sicht auf diese Basisinformationen dar und haben daher im Metamodell selbst keinen Platz. Alle sichtspezifischen, über die Basisinformationen hinausgehenden Eigenschaften finden ihren Platz in einem Konnektor, der in Abschnitt 4.4.1 näher beschrieben wird.

4.2.1 Basisentitäten

Im Folgenden werden die im Metamodell zu repräsentierenden Entitäten beschrieben und anschließend ihre Integration ins PIROL Metamodell vorgestellt. Bei der Beschreibung wird ausdrücklich Bezug genommen auf die bereits im Metamodell enthaltenen Klassen, um zu einer harmonischen Einordnung der neuen Elemente zu gelangen.

Teams

Teams tragen sowohl Eigenschaften von Paketen als auch von Klassen. Wie Pakete sind Teams Container für andere Elemente, in diesem Fall für Klassen und Rollen. Mit Klassen haben Teams gemeinsam, dass sie eigene Attribute und Methoden besitzen können. Außerdem können Teams an zwei neuen Arten von Beziehungen teilnehmen: der Vererbungsbeziehung zwischen Teams und der Adaptionsbeziehung zwischen Teams und Paketen.

Rollen

Rollen gleichen im Object Teams Modell vom Prinzip her unvollständig implementierten Klassen. Im Unterschied zu Klassen kann ihre Unvollständigkeit jedoch durch explizite Bindung an eine Basisklasse behoben werden. Eine weitere Besonderheit von Rollen tritt bei der Spezialisierung von Teams zu Tage: Rollen erben implizit von namensgleichen Rollen in Super-Teams.

Bindungen

Durch Bindungen werden Rollen an Basisobjekte gebunden, die die von der Rolle noch benötigte Funktionalität bereitstellen. Die Bindungsdetails werden in Form von callout- und callin-Bindungen spezifiziert.

4.2.2 Metamodell-Entwurf

Paket PRODUCT

Für die Metamodell-Repräsentation von Object Teams musste das Paket PRODUCT nur an einer Stelle erweitert werden: in der Klasse ROUTINE wurde das Attribut *is_callin* eingeführt, um Callin-Methoden als solche kennzeichnen zu können (Abb. 4.1).

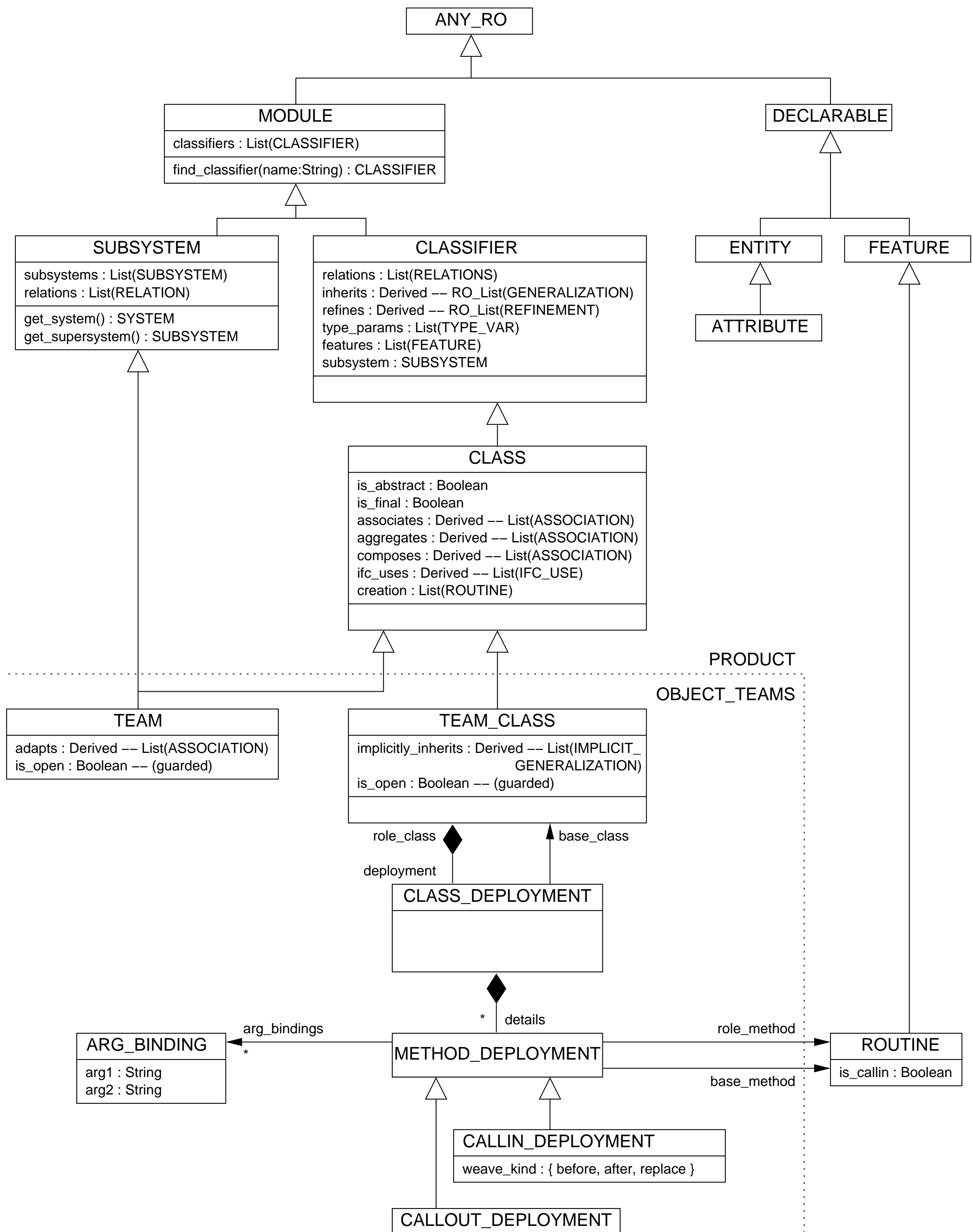


Abbildung 4.1: Ausschnitt aus den Paketen PRODUCT und OBJECT_TEAMS

Paket RELATIONS

Um die Adaptionsbeziehung zwischen Teams und Paketen modellieren zu können, wurde das am UML Metamodell angelehnte Attribut *aggreg_indicator* der ASSOCIATION_ROLE um den Indikator *adaptation* erweitert. Außerdem stellte sich heraus, dass die Attribute *supplier* und *client* in RELATION vom Typ CLASSIFIER unnötig speziell sind und so die Adaptionsbeziehung zwischen den Objekten der Typen SUBSYSTEM und TEAM nicht erlaubt hätten. Dementsprechend wurde der Typ dieser Attribute zu MODULE verallgemeinert.

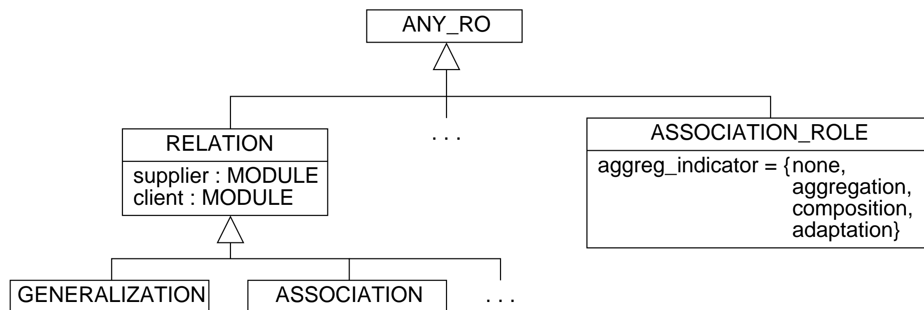


Abbildung 4.2: Ausschnitt aus dem erweiterten PIROL Metamodell Paket RELATIONS

Paket OBJECT_TEAMS

Alle anderen Erweiterungen wurden in das neue Paket OBJECT_TEAMS aufgenommen (Abb. 4.1). Die Klasse TEAM erbt von SUBSYSTEM und CLASS und fügt das Attribut *is_open* hinzu. Über das Attribut *adapts* kann auf die Liste der Adaptionsbeziehungen zugegriffen werden, welche aus der Liste der Relationen abgeleitet wird.

Da Rollen im wesentlichen Klassen gleichen, wurde die neue Klasse TEAM_CLASS von CLASS abgeleitet und um die drei Attribute *implicitly_inherits*, *is_open* und *deployment* erweitert. Mit Hilfe von Attributwächtern werden unter anderem an den Attributen *subsystem* und *is_abstract* einige Konsistenzregeln sichergestellt. Beispielsweise dürfen TEAM_CLASSes nur innerhalb von TEAMs definiert werden und wenn eine TEAM_CLASS abstrakt ist und keine konkrete Subklasse besitzt, so hat dies zur Folge, dass auch das sie umgebende TEAM als abstrakt gekennzeichnet wird.

Um eine Teamklasse an eine Basisklasse binden zu können, besitzt TEAM_CLASS das Attribut *deployment* vom Typ CLASS_DEPLOYMENT, welches eine eindeutige Beziehung zwischen einer Rollenklasse und einer Basisklasse modelliert. Die Bindungsdetails in Form von Zuordnungen von Rollenmethoden zu Basismethoden

werden durch Objekte der Klassen CALLIN_DEPLOYMENT und CALLOUT_DEPLOYMENT übernommen.

4.2.3 Instanziierung des neuen Metamodells

Um die Eignung des Metamodell-Entwurfes zu überprüfen, wird ein Ausschnitt einer Instanziierung des in Abbildung 4.3 gezeigten Entwurfes in einem Objektdiagramm des neuen Metamodells dargestellt. Die resultierenden Instanzen der Metamodell-Klassen sind im Objektdiagramm in Abbildung 4.4 gezeigt.

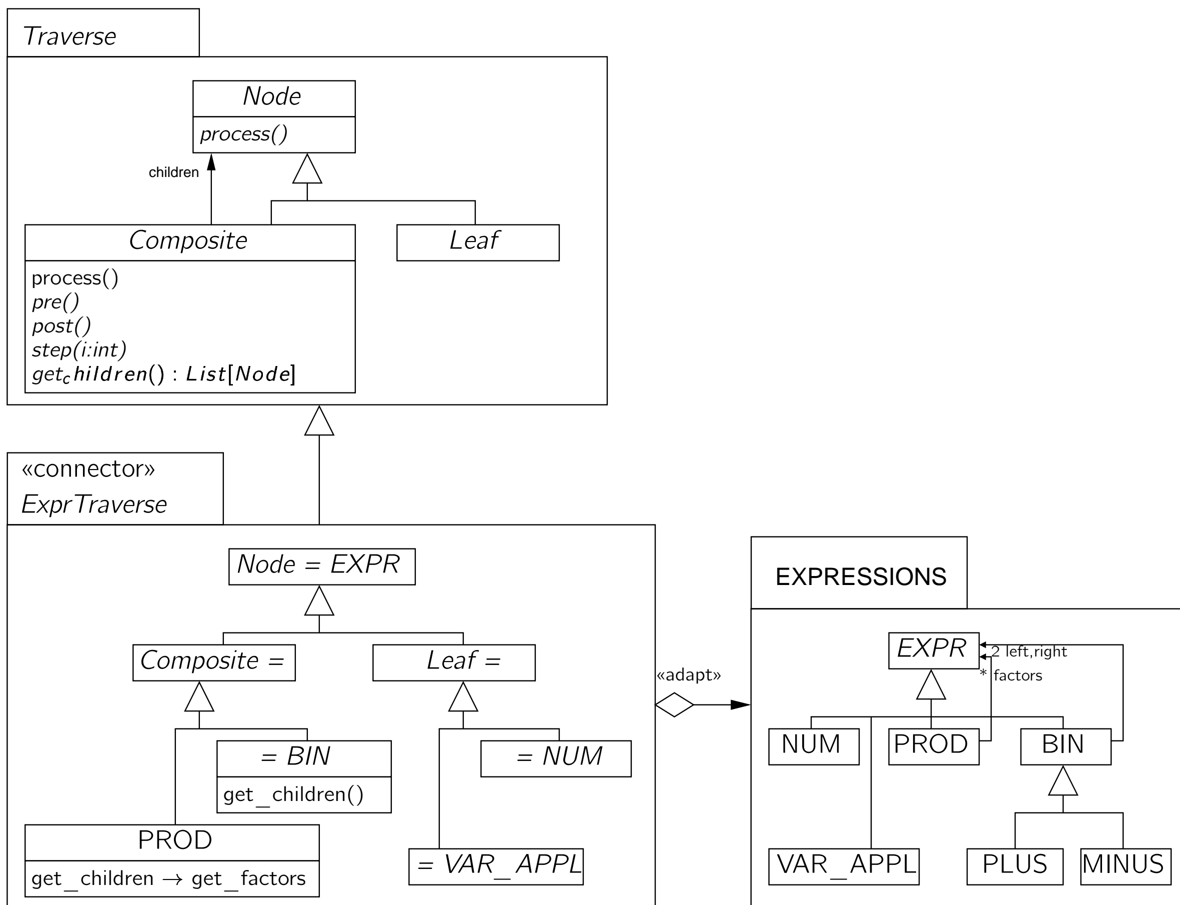


Abbildung 4.3: Beispiel eines UFA-Diagramm

4.3 GEF: Graph Editing Framework

Bei der Entwicklung des UFA-Editors soll auf das GEF Grafikframework (`http://gef.tigris.org`) zurückgegriffen werden. Dieses ging in einer Diplomarbeit von

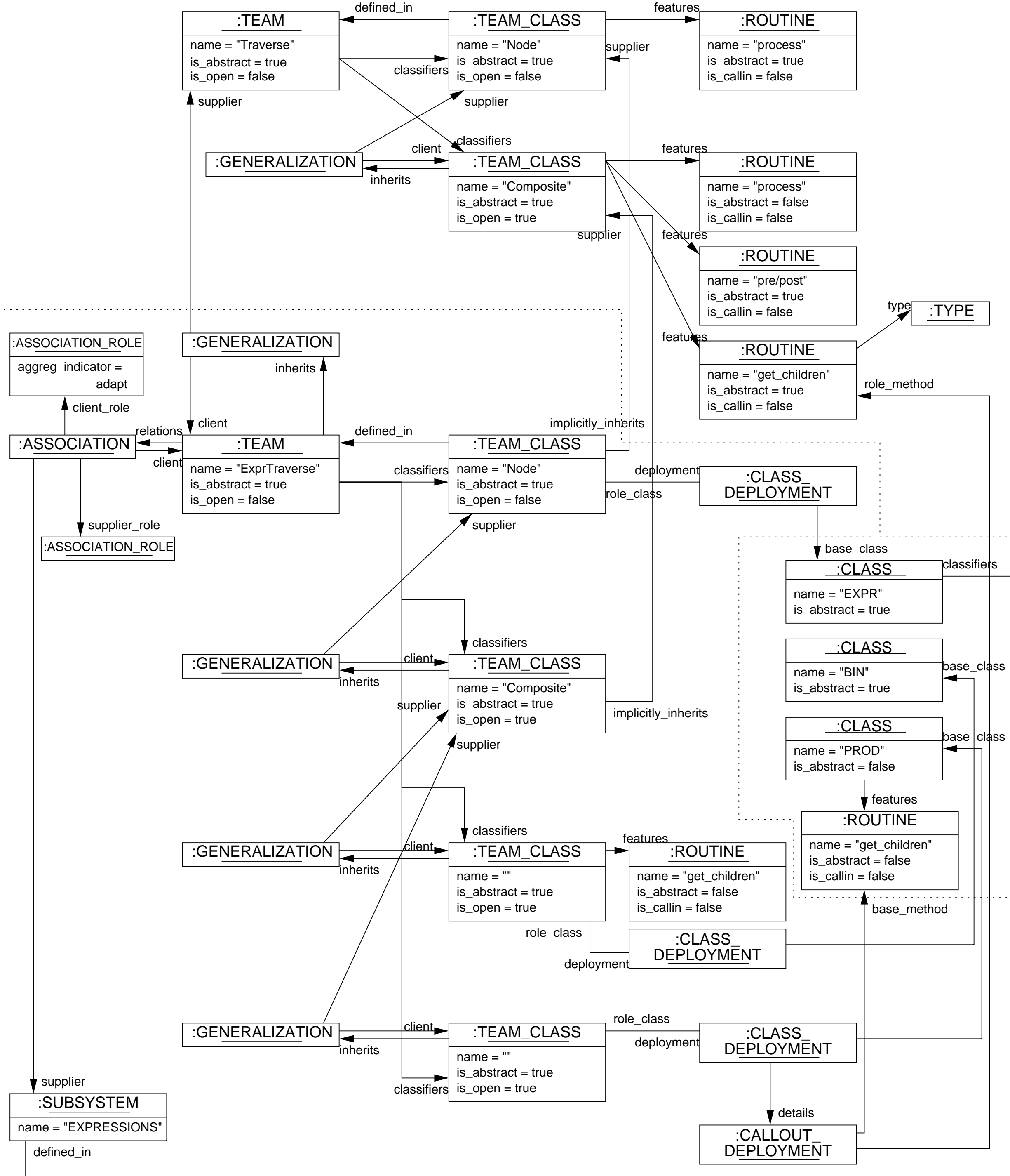


Abbildung 4.4: Objektdiagramm zu einem Ausschnitt aus dem UFA-Diagramm in Abb. 4.3

Burkhard Weber (Weber, 2001) aus einer Gruppe von Grafikframeworks als das Geeignetste hervor und es wurde eine erste Anbindung an PIROL erstellt.

GEF ist ein Java-Framework zur Erstellung von grafischen Editoren für Graphen[1]. Ursprünglich wurde es von Jason Robbins an der University of California entwickelt und fand später Verwendung in ArgoUML (`http://argouml.tigris.org`), einem Java-basierten CASE Werkzeug mit kognitiver Unterstützung für den Entwurf. In ArgoUML beweist GEF seine Leistungsfähigkeit und beeindruckt durch eine für grafische Java-Applikationen nicht selbstverständliche Geschwindigkeit. GEF wird derzeit als Open Source Projekt weiterentwickelt und steht unter der BSD License.

4.3.1 Funktionsweise

Im GEF Framework gibt es zwei zu betrachtende Ebenen: die Graph-Ebene und die Diagramm-Ebene. Die Graph-Ebene enthält Objekte, die die logischen Knoten und Kanten des Graphen repräsentieren und applikations-spezifische Daten und Verhalten besitzen können. Auf der Diagramm-Ebene sind die sichtbaren Grafikelemente enthalten, die die Struktur der Graph-Ebene für den Benutzer durch Rechtecke, Linien, Text usw. visualisieren.

Knoten können Ports besitzen, welche die Andockpunkte für die Kanten darstellen (Abb. 4.5). Ob eine Kante zwischen zwei Knoten (genauer: zwischen den Ports zweier Knoten) zulässig ist, hängt jeweils von der Art der Ports und der Art der Kante ab.

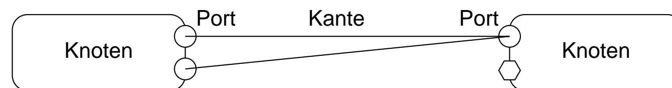


Abbildung 4.5: GEF Node-Port-Edge Modell

Die Visualisierung übernehmen die Klassen der Diagramm-Ebene 4.6. Dazu gehören eine Reihe von Grafik-Primitiven (Linie, Rechteck, Kreis, etc.) sowie mit `FigGroup` eine Komponente, die aus mehreren Grafik-Primitiven aufgebaut sein kann. Die Brücke zur Graph-Ebene bilden die Klassen `FigNode` und `FigEdge`, die ebenfalls aus mehreren Grafik-Primitiven bestehen können und zur Darstellung der applikations-spezifischen Knoten und Kanten des Graph-Modells dienen.

4.3.2 Architektur

Im Folgenden werden die tragenden Elemente zur Entwicklung eines Grafikeditors mit Hilfe von GEF vorgestellt (siehe auch Abb. 4.7):

[1] Im Folgenden wird mit der Schreibweise "Graph" betont, dass es sich hierbei um Graphen im *mathematischen Sinne* handelt. Im Gegensatz dazu werden alle Wörter, die sich auf die *grafische Darstellung* beziehen von "Graf" abgeleitet.

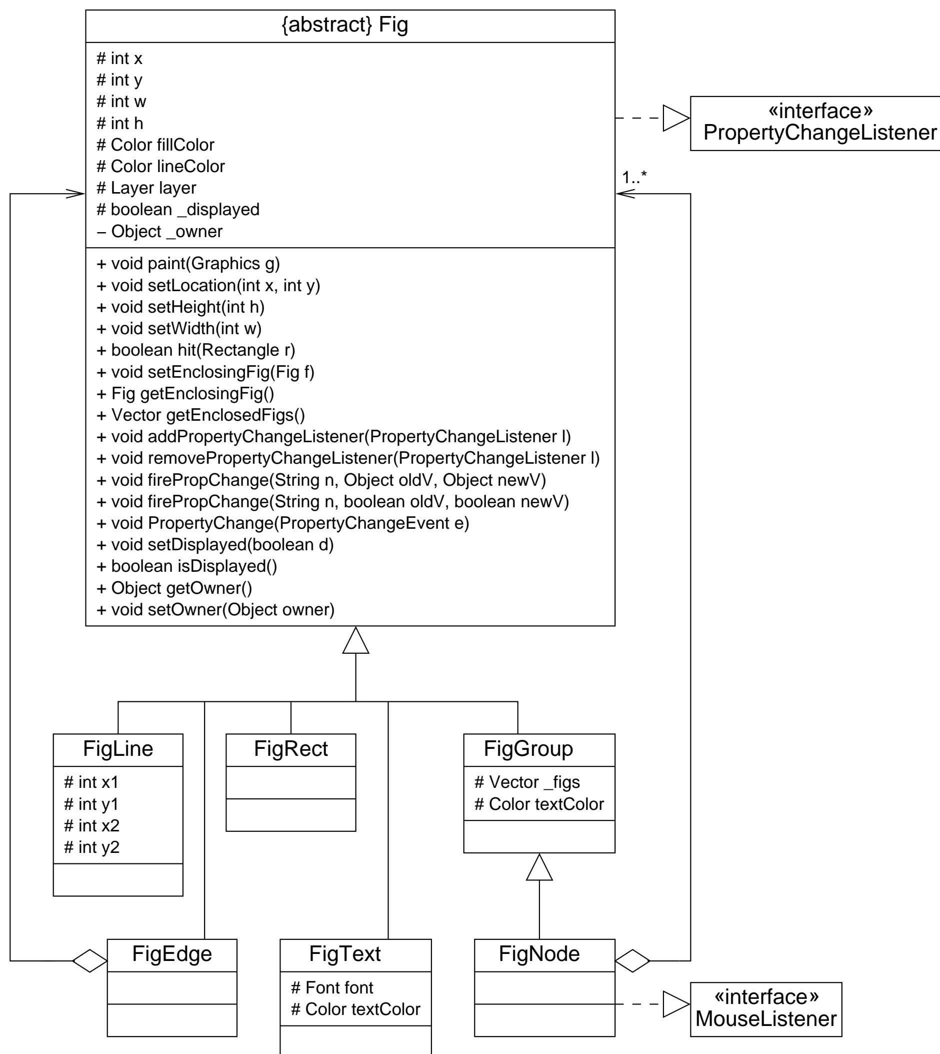


Abbildung 4.6: Ausschnitt aus dem GEF-Paket presentation

JGraph ist eine Swing-Komponente, die die Darstellung eines Editors übernimmt.

Editor ist die zentrale Klasse eines mit GEF erstellten grafischen Editors von der für jedes angezeigte Diagramm genau eine Instanz existiert. Die Hauptaufgabe von Editor ist es, als Container für die anderen benötigten Elemente zu dienen und den Nachrichtenaustausch zu organisieren indem die Nachrichten an die richtigen Objekte weitergeleitet werden. Dazu gehören insbesondere die Benutzereingaben über die Maus.

Modes sind Klassen, die verschiedene Eingabezustände des Editors darstellen. Sie definieren den Kontext, in dem Eingabenachrichten interpretiert werden. So existieren z.B. Modi zur Objekterzeugung, zum Selektieren oder zum Verschieben von Elementen.

ModeManager verwaltet die verschiedenen Eingabe-Modi und leitet die vom Editor empfangenen Nachrichten an die einzelnen Modi in entsprechender Reihenfolge weiter, bis ein Modus die zugestellte Nachricht bearbeitet.

Selection ist eine Klasse, die die selektierten Zeichenobjekte eines Editors repräsentiert. Sie sorgt für die Visualisierung der Selektion (z.B. mit einem Rahmen und Handles) und implementiert die Reaktionen auf deren Manipulation (z.B. Vergrößern durch Ziehen an einem Handle).

Cmds sind Java Actions und implementieren einzelne Befehle für den Editor, wie "Ausschneiden", "Alles selektieren" oder "Knoten erstellen". Idealerweise sollten alle Aktionen durch Cmds implementiert werden, da sie dann einfach in Menüs oder Werkzeugleisten integriert werden können. Eine Undo-Funktionalität ist zwar vorgesehen, aber noch nicht implementiert worden.

Figs stellen die Zeichenobjekte dar mit denen im Editor gearbeitet werden kann. In FigGroups können mehrere primitive Zeichenobjekte zu einem einzelnen zusammengefasst werden.

Layers erlauben es verschiedene Ebenen für die Darstellung der Editor-Zeichenfläche zu verwenden. Die Ebenen werden dann in einer bestimmten Reihenfolge transparent übereinander gelegt. Ebenen können Figures enthalten oder ihre Darstellung auch selbst berechnen, wie beispielsweise LayerGrid, das ein Raster zeichnet und als unterste Ebene verwendet werden sollte.

LayerPerspective ist eine spezielle Ebene, die darauf vorbereitet ist, applikationsspezifische Objekte der Graph-Ebene als Modell für darzustellende Zeichenobjekte zu verwenden.

LayerManager fasst die unterschiedlichen Layers zusammen, organisiert deren Reihenfolge und verwaltet die Information, welche Ebene gerade die aktive Ebene ist.

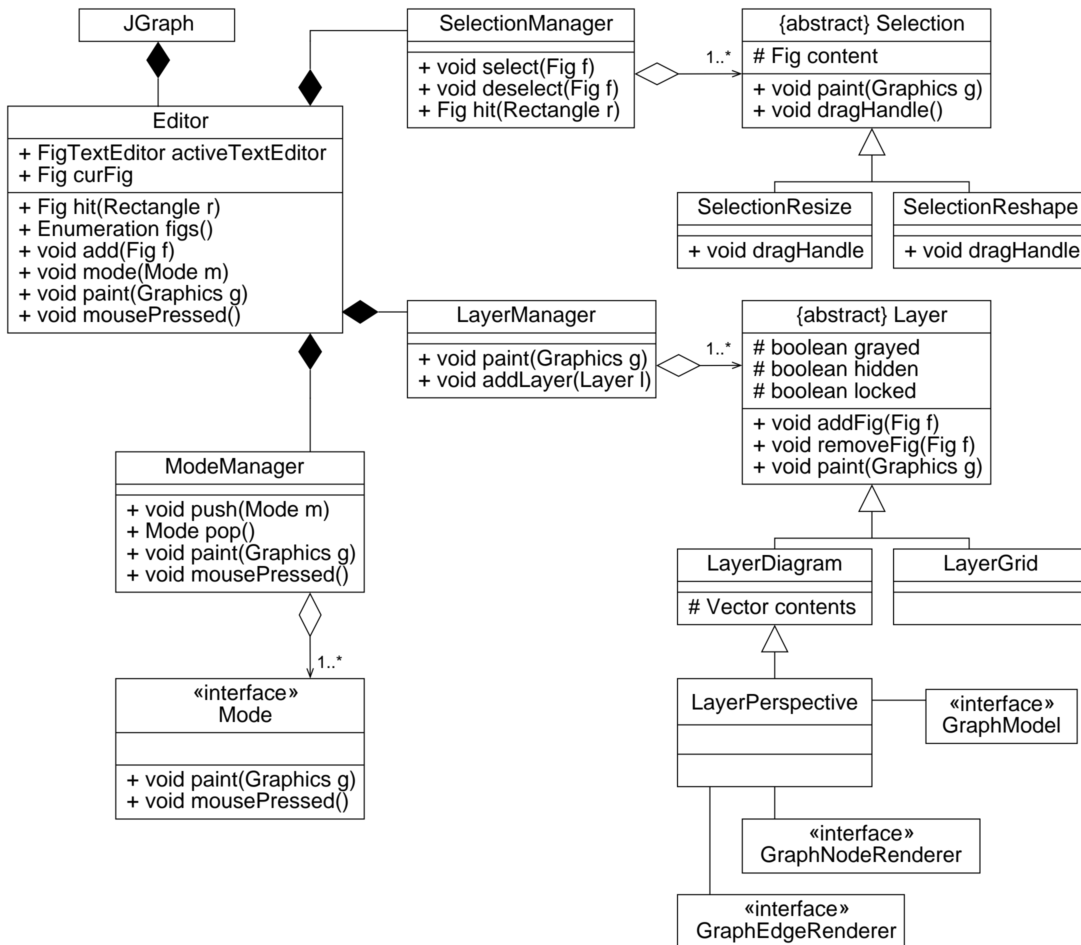


Abbildung 4.7: Hauptstruktur von GEF

4.3.3 GraphModel und DefaultGraphModel

Bei der Realisierung wurde auf das in Abb. 4.8 und Abb. 4.9 dargestellte Default-GraphModel zurückgegriffen. Das Interface *GraphModel* deklariert zunächst alle Methoden, um die Knoten, Kanten und Ports des Modells erfragen zu können. GraphListener können sich beim GraphModel anmelden, um über Veränderungen im Graphen informiert zu werden. Dazu wird das Java-Beans Nachrichtenmodell verwendet.

GraphModel

Das abgeleitete Interface *MutableGraphModel* deklariert alle zur Manipulation des

MutableGraph-Model

Modells nötigen Methoden wie zum Beispiel addNode(Object node). In der abstrakten Klasse MutableGraphSupport werden die geerbten Interfaces erstmals implementiert.

DefaultGraph-Model

Das *DefaultGraphModel* spezialisiert die Klasse MutableGraphSupport und ist so ausgelegt, dass sie mit den Klassen NetPrimitive, NetList, NetNode, und NetEdge (Abb. 4.9) zusammenarbeitet.

NetPrimitive

Alle im DefaultGraphModel enthaltenen Elemente des Graphen leiten von der abstrakten Klasse *NetPrimitive* ab, welche ebenfalls das Java-Beans Nachrichtenmodell implementiert. Damit können auch den Graphelementen Listener hinzugefügt werden, die über die gefeuerten Nachrichten informiert werden. Die wichtigste von den Knoten, Kanten und Ports des Graphen gesandte Nachricht ist "disposed". Sie besagt, dass das entsprechende Element aus dem Graphen entfernt wurde.

Die Klassen NetNode, NetEdge und NetPort implementieren die nötigen Verbindungen untereinander und Methoden zum Abfragen und Setzen ihrer Eigenschaften. Die implementierten Interfaces GraphNodeHooks und GraphPortHooks erlauben es unter anderem, die Graph-Elemente nach der Aufnahme oder vor dem Verbinden mit oder dem Trennen von einer Kante zu informieren. Da die Graph-Elemente meist durch ein Cmd über newInstance() und nicht über ihren Konstruktor erzeugt werden, dient die Methode initialize(Hashtable properties) dazu, die zur Initialisierung nötigen Argumente in allgemeiner Form nachzuliefern.

DefaultGraph-NodeRenderer

DefaultEdge-Renderer

Zur Default-Implementierung gehört auch der *DefaultGraphNodeRenderer*. Er implementiert das GraphNodeRenderer Interface und wird später die Anfragen zur konkreten Visualisierung von Knoten erhalten, welche die Klasse LayerPerspective *DefaultEdgeRenderer* an ihn richten wird (Abb. 4.7). Der DefaultGraphNodeRenderer beantwortet diese Anfragen indem er sie an den entsprechenden NetNode delegiert. Der *DefaultEdgeRenderer* verfährt für Kanten in analoger Weise.

Ports

Ports besitzen keinen expliziten Renderer. Da sie stets einem Knoten oder einer Kante angehören, deren von FigNode (bzw. FigEdge) abgeleitete Repräsentation aus einer Gruppe von Figs aufgebaut ist, kann über FigNode::bindPort(Object port, Fig fig) eine der enthaltenen Figs an den Port gebunden werden. Wird nun beispielsweise eine Kante zu dieser Fig gezogen, kann der Port dazu ermittelt und in die Aktion einbezogen werden.

4.4 Erstellung des UFA-Editors mit Hilfe von GEF

Die für die Repräsentation von UFA-Entwürfen auf PIROL-Seite nötigen Erweiterungen wurden in Abschnitt 4.2.2 vorgestellt. Um diese Repository-Objekte mit Hilfe von GEF auch grafisch editierbar zu machen, müssen sie zum einen Gegenstücke auf der GEF-Seite bekommen und zum anderen muss eine Anbindung erstellt werden, die Veränderungen an Objekten auf beiden Seiten auf der jeweils anderen nachzieht.

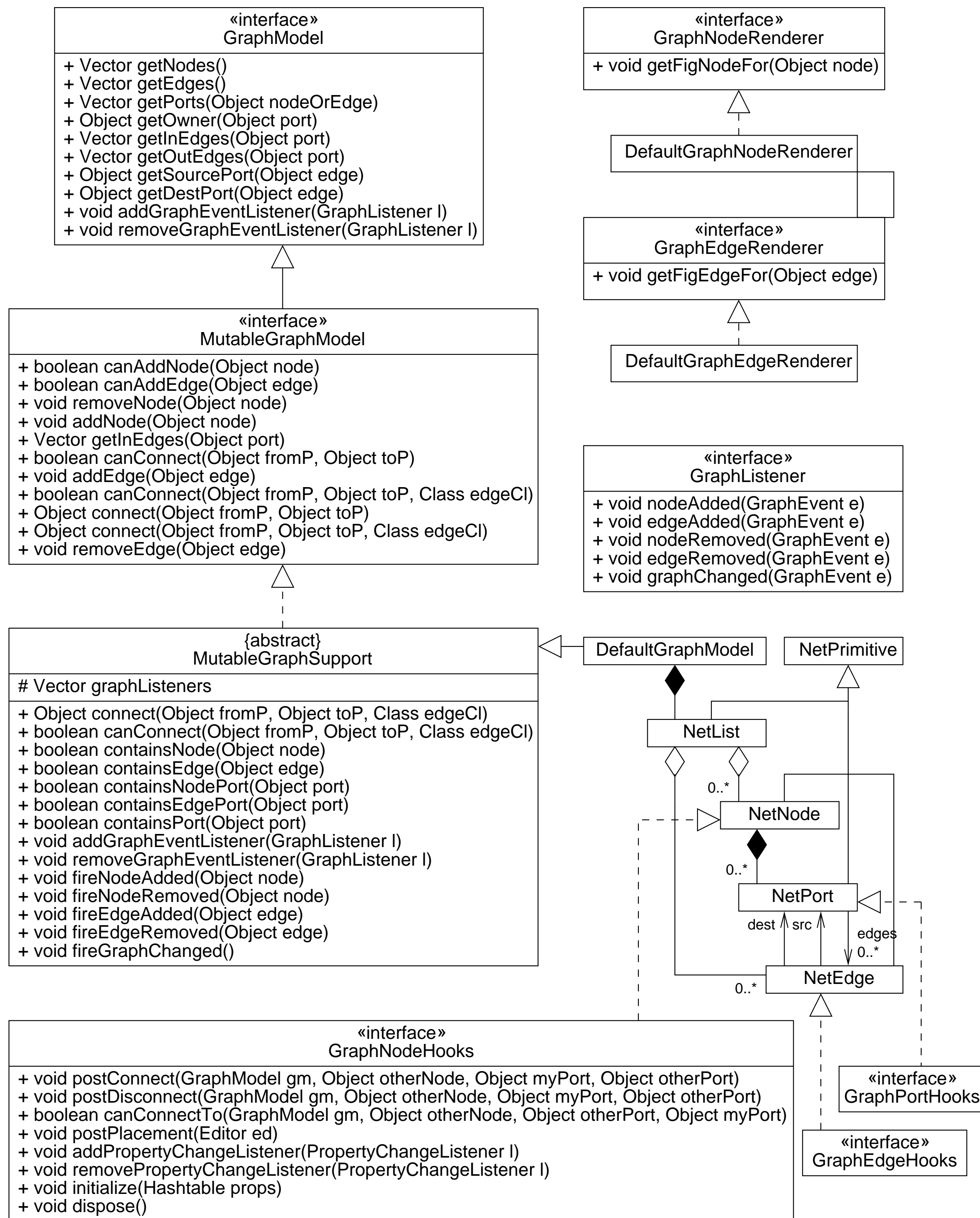


Abbildung 4.8: Ausschnitt aus dem GEF-Graphmodell

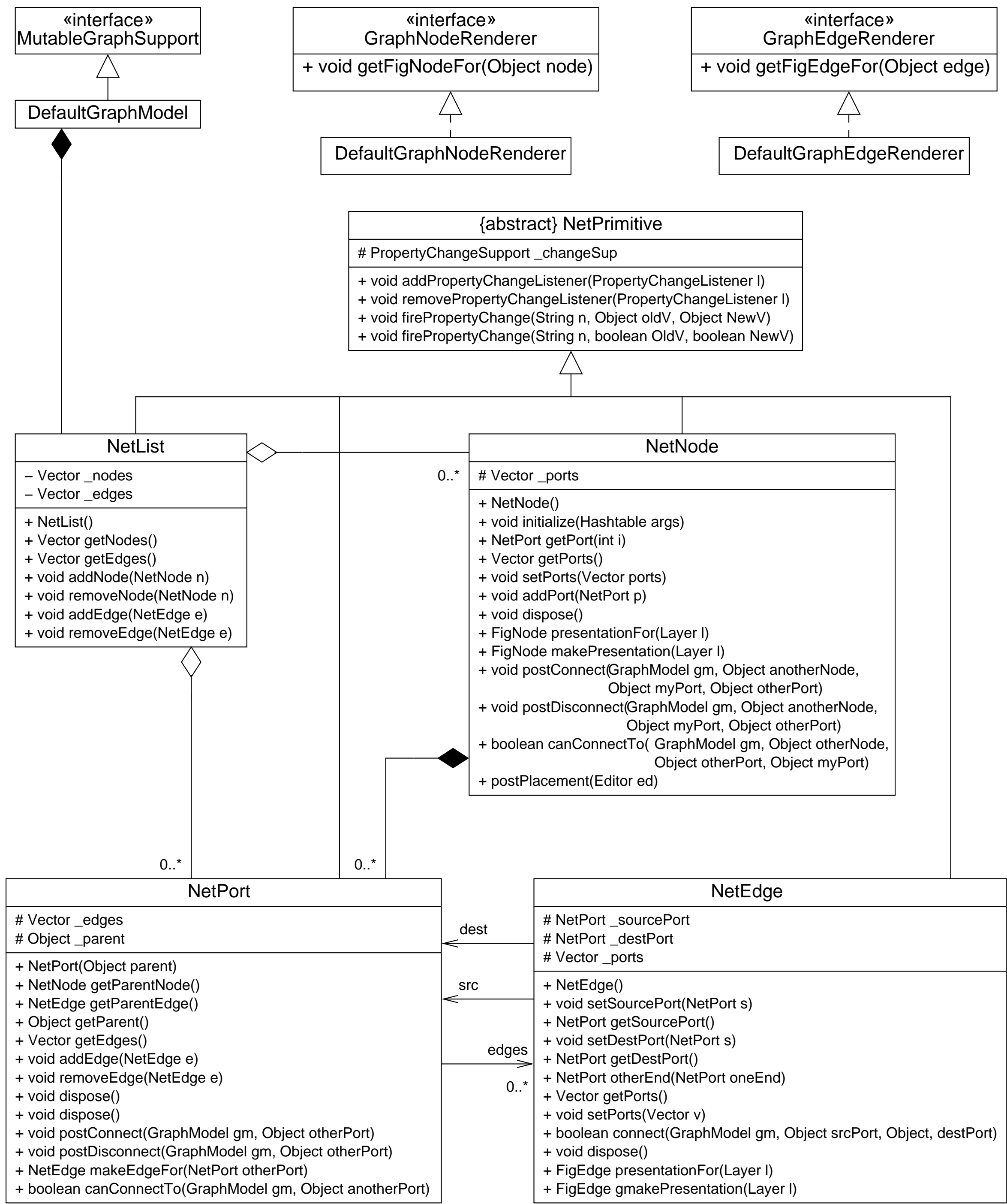


Abbildung 4.9: Ausschnitt aus dem GEF DefaultGraphModel

Außerdem ging es bei der Metamodell-Erweiterung bisher nur um die Repräsentation der in einem UFA-Entwurf verwendeten Elemente wie Klassen, Teamklassen und Teams und noch nicht um deren *grafische* Repräsentation.

Bei den im PIROL-Paket OBJECT_TEAMS bereits modellierten Eigenschaften fehlen z.B. noch die für Diagramm-Elemente benötigten Darstellungsinformationen wie Position und Größe. Da diese zusätzlichen Informationen jedoch nur für eine ganz spezielle Sicht auf das Repository – nämlich die des grafischen Editors – sinnvoll und erforderlich sind, werden sie in einem Konnektor (vgl. Abschnitt 3.2.3) definiert.

4.4.1 UFA-Diagramm Konnektor

Der Konnektor OT_DIAG_CONNECTOR versieht die verwendeten ROs mit den Attributen und Methoden, die nur für die Diagramm-Sicht auf Object Team Entwürfe relevant sind. Neue Eigenschaften sind z.B. die Größe und die Position und die Information, ob die Attribute und Methoden einer Klasse bei der Darstellung im Editor angezeigt oder versteckt werden.

Als nützlich erwies sich an dieser Stelle die Möglichkeit, neue Attribute in die Konnektorklassen einzuführen, die eine gefilterte Sicht auf bestehende Listenattribute erlauben. So besitzt die Metamodell-Klasse SUBSYSTEM z.B. ein Listen-Attribut "classifiers", auf welches über die drei gefilterten Attribute "interfaces", "classes" und "teams" zugegriffen werden kann, die nur die enthaltenen Elemente des entsprechenden Typs zurückgeben.

Wenn im Folgenden von den VOs gesprochen wird, ist immer die durch diesen Konnektor angepasste Sicht auf die ROs gemeint, deren Proxys im Java Paket rocmufa definiert wurden.

4.4.2 Repräsentation auf GEF-Seite

Knoten

Kanten

Folgende Elemente eines UFA-Diagramms wurden als *Knoten* des Graphen identifiziert: Klassen, Pakete, Teams und Teamklassen[1]. Die zwischen diesen Elementen möglichen Assoziationen und Generalisierungen[2] stellen die *Kanten* des Graphen dar.

Bei der Realisierung wurde auf das in Abb. 4.8 und Abb. 4.9 dargestellte DefaultGraphModel zurückgegriffen.

Graph-Ebene

In Abbildung 4.10 sind die Klassen dargestellt, die zur Repräsentation des Graphen implementiert wurden. Sie leiten von den für die Zusammenarbeit mit dem Default-

[1] Entsprechung im Metamodell: CLASS, PACKAGE, TEAM und TEAM_CLASS

[2] Entsprechung im Metamodell: ASSOCIATION und GENERALIZATION

GraphModel vorgesehenen Klassen NetNode und NetEdge ab und spezialisieren diese.

Ein großer Teil der Implementierung stellt Methoden bereit, die über die Daten des Graph-Elementes Auskunft geben und von den zur Visualisierung verwendeten Klassen benötigt werden.

Außerdem wurden die Methoden implementiert, die für die im folgenden Abschnitt beschriebene Anbindung an das PIROL-Repository benötigt werden.

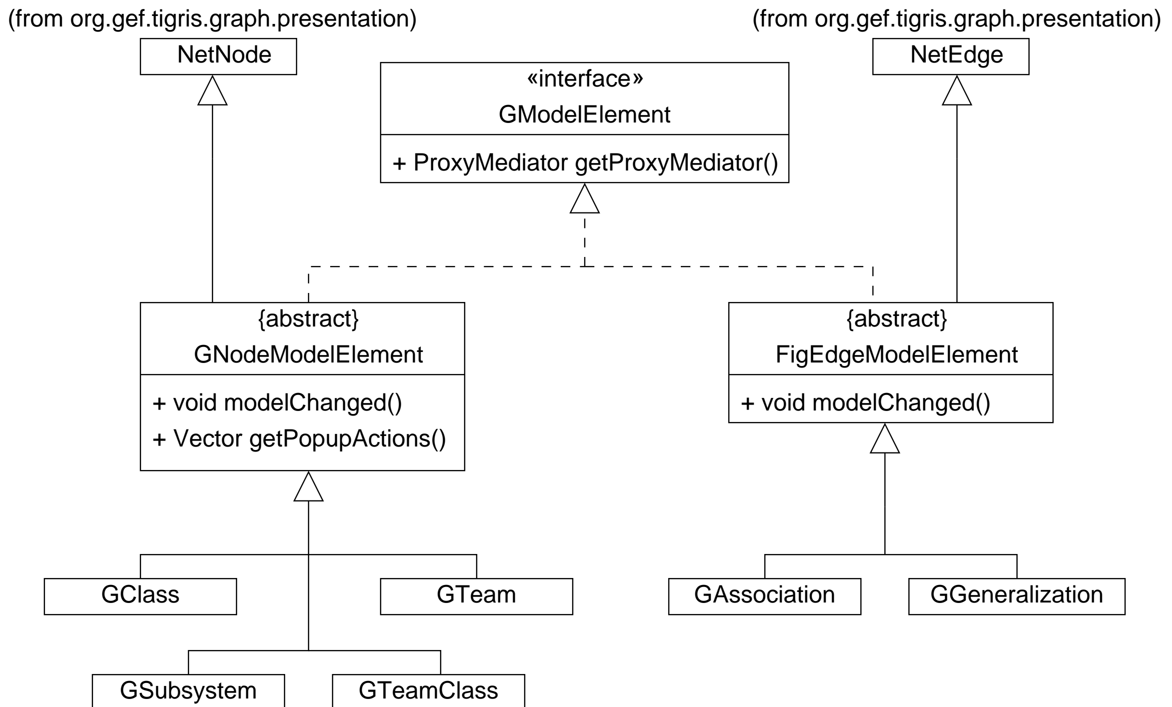


Abbildung 4.10: Ausschnitt aus dem UFAEd-Paket graph

Diagramm-Ebene

Die Darstellung der Diagramm-Elemente übernehmen die in Abbildung 4.11 gezeigten Klassen. Sie greifen dazu auf die Klassen FigNode und FigEdgePoly des Frameworks zurück, die zum Beispiel die Verwaltung der Ports implementieren und sich aus dem sie anzeigenden Layer entfernen lassen, wenn sie ein PropertyChangeEvent vom Typ "disposed" erhalten, das von ihrem _owner-Objekt des Graphen stammt.

Die zur Darstellung benötigten Informationen erfragen die Klassen des Paketes figs von ihren _owner-Objekten der Graph-Ebene.

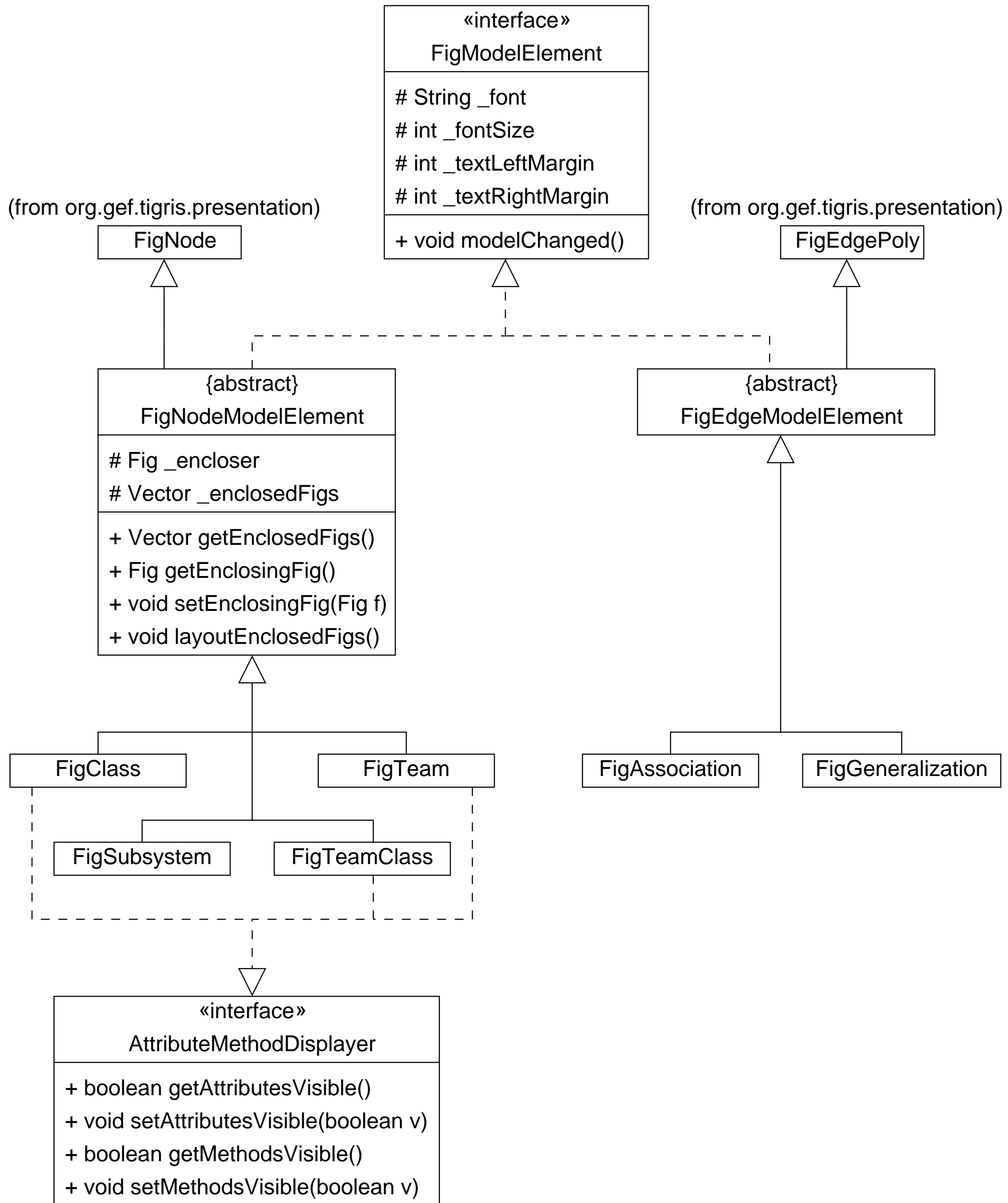


Abbildung 4.11: Ausschnitt aus dem UFAEd-Paket figs

4.4.3 Anbindung von GEF an PIROL

Die statischen Eigenschaften der beiden zu verbindenden Seiten wurden beschrieben. In Tabelle 4.1 sind alle Repräsentationsebenen zusammenfassend dargestellt, angefangen beim RO, über das vom Konnektor zum VO (view object) angereicherte RO, über die Java-Proxyklasse bis zur Repräsentation auf GEF-Seite in dem dem Editor zugrundeliegenden Graphmodell und der eigentlichen Visualisierungsklasse.

	RO	**VO**	**Java-Proxy**	**Graph-Element**	**Diagramm-Element**
Beispiel-Klasse	CLASS	CLASS	CLASS	GClass	FigClass
aus Paket	PRODUCT	OBJECT_ TEAMS	rocmufa	ufaed.graph	ufaed.figs
Implementierungssprache	Lua/P	Lua/P	Java	Java	Java

Tabelle 4.1: Repräsentationsebenen

Die Graph-Elemente stellen nun nicht mehr, wie von GEF eigentlich vorgesehen, die unterste Repräsentationsebene dar. Die Aufgabe, die Daten der Graph-Elemente zu speichern, wird nun vom PIROL-Repository übernommen. Die Daten werden nicht mehr im Graph-Element selbst gehalten, sondern in dem dem Graph-Element entsprechenden PIROL-Proxy Objekt. Es verbleibt den Graph-Elementen aber die Aufgabe, mit den restlichen Komponenten von GEF zusammenzuarbeiten.

4.4.4 Mediatoren

Um die PIROL Java-Proxys der VOs und deren Repräsentation im Graphmodell zueinander zu bringen, werden ProxyMediatoren eingeführt, die zwischen diesen beiden Seiten vermitteln (Abb. 4.12 und 4.13). Für jede als Knoten oder Kante im Graphen repräsentierte PIROL Proxy-Klasse existiert ein entsprechender Mediator.

Es gibt von PIROL-Seite aus drei Gruppen von Ereignissen (Fälle 1-3), die durch den Mediator behandelt und geeignet auf die GEF-Seite übertragen werden müssen. Auf der GEF-Seite gibt es drei entsprechende Klassen von Ereignissen (Fälle 4-6).

Fall 1: Objektveränderung auf PIROL-Seite

Damit die Darstellung im UFA-Editor immer den aktuellen Daten der zugrunde liegenden VOs im Repository entspricht müssen die Mediatoren Veränderungen an den VOs

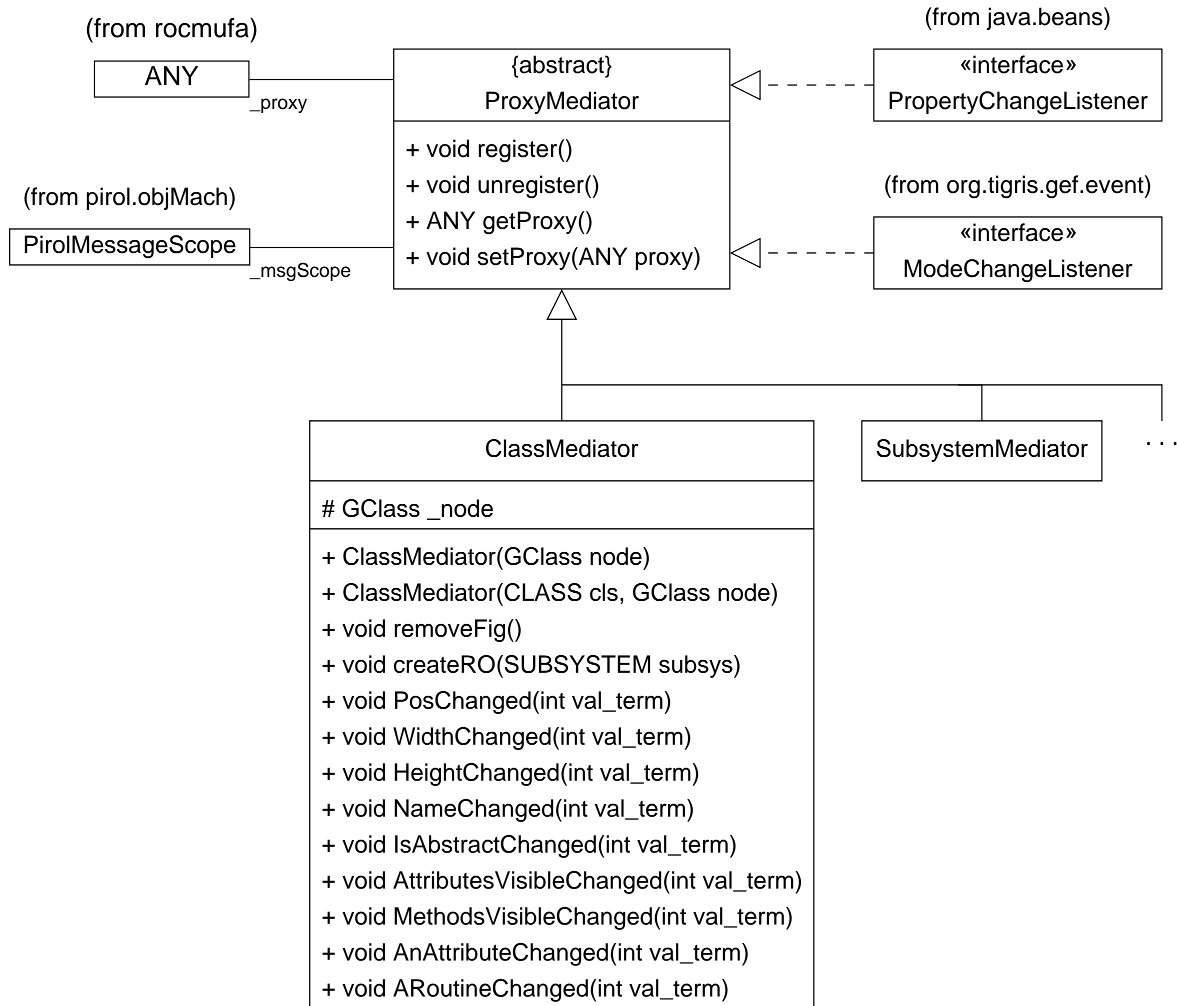


Abbildung 4.12: Ausschnitt aus dem Paket ufaed.mediators

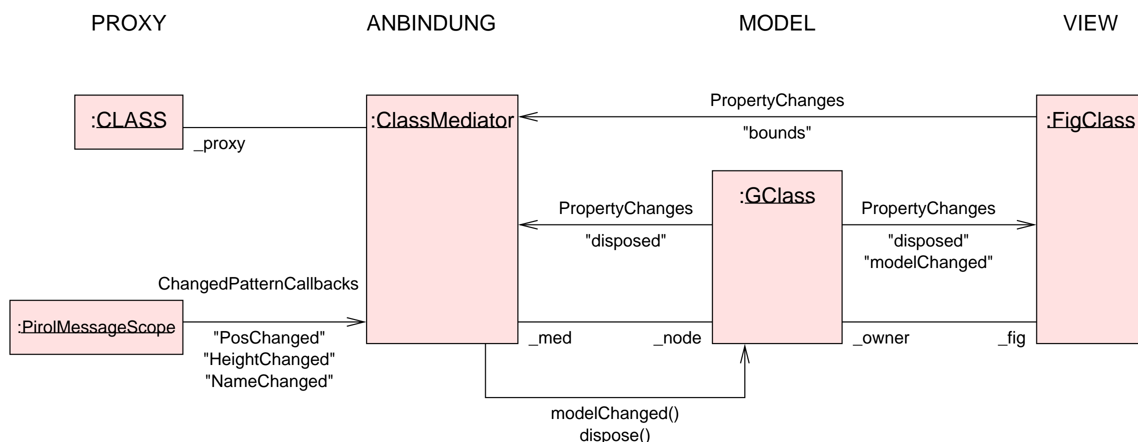


Abbildung 4.13: Übersicht über die Anbindung am Beispiel von CLASS

verfolgen. Dazu meldet jeder Mediator für alle relevanten Attribute ChangedPatterns und ListChangeObserver beim PirolMsgScope bzw. der entsprechenden PirolList an.

Relevant sind alle Attribute, die im UFA-Editor auch angezeigt werden. Für das VO CLASS gehört dazu z.B. sein Name und die Liste seiner Attribute. Da aber nicht nur auf das Hinzufügen und Löschen von Attributen in die bzw. aus der Liste reagiert werden soll, sondern auch auf Veränderungen an den Attributen selbst (Name, Sichtbarkeit usw.), müssen auch für diese Attribute ChangedPatterns bzw. ListChangeObserver angemeldet werden.

Werden die ListChangeObserver Callbacks aktiviert weil ein Element der Liste hinzugefügt oder entfernt wurde, müssen die entsprechenden ChangedPatterns für dieses Element an- bzw. abgemeldet werden.

Außerdem informiert der Mediator sein entsprechendes Gegenstück auf der Graph-Seite darüber, dass sich die ihm zugrundeliegenden Daten geändert haben. Das Graph-Element wiederum gibt diese Änderung allen seinen PropertyChangeListenern (und damit der darstellenden Fig) über die Nachricht “modelChanged” bekannt.

Im Sequenzdiagramm in Abbildung 4.14 ist der Nachrichtenfluss ausschnittsweise dargestellt, der beim Ändern des Attributes “name” des VOs CLASS stattfindet. Dies ist zunächst der einfachste Fall, da er lediglich Änderungen in bereits bestehenden Objekten erfordert.

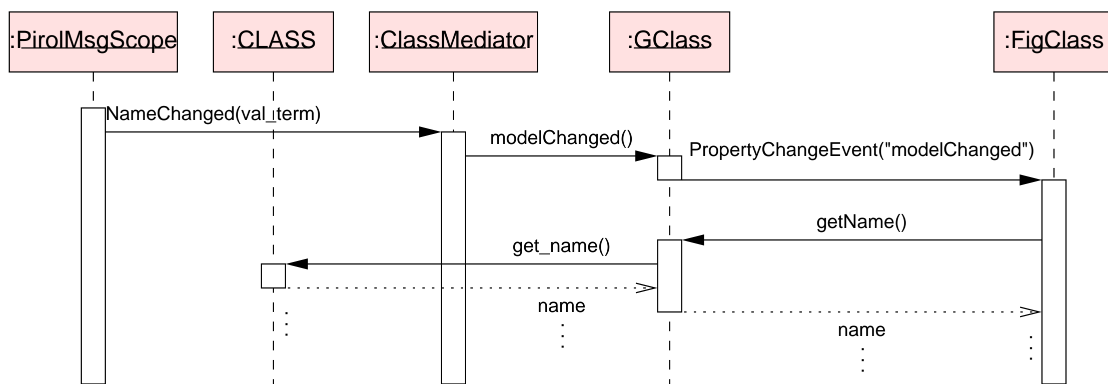


Abbildung 4.14: Nachrichtenfluss beim Ändern des Attributes “name” einer CLASS

Fall 2: Objekterzeugung auf PIROL-Seite

Wenn auf der PIROL-Seite ein im Editor durch einen Knoten oder eine Kante repräsentiertes Element neu erstellt wird, ist der Ablauf komplizierter, da die Gegenstücke der GEF-Seite erst erzeugt werden müssen. Der im Folgenden beschriebene Ablauf ist dem Sequenzdiagramm in Abb. 4.15 zu entnehmen.

Wird eine neue Klasse in ein Subsystem eingefügt (also ein Objekt der Klasse CLASS in die Liste SUBSYSTEM.classifiers eingefügt), so wird der zu dieser Liste gehörige ListChangeObserver aufgerufen: ListChangeObserver.do_append(Object o).

Dort wird entschieden, um welche Art von Classifier es sich handelt und in diesem Fall entsprechend eine Factory-Methode des ClassMediators aufgerufen: ClassMediator.create(CLASS c). Diese erstellt einen neuen Knoten vom Typ GClass, einen Mediator vom Typ ClassMediator, initialisiert den Knoten mit dem neuen ClassMediator und fügt anschließend den Knoten zum GraphModel hinzu.

Das UFAGraphModel informiert die als GraphEventListener registrierte LayerPerspective über das Hinzufügen des Knotens. Diese fragt daraufhin beim DefaultGraphNodeRenderer nach einer Repräsentation für diesen Knoten. Die Anfrage wird an den Knoten GClass delegiert, welcher die entsprechende FigClass erzeugt und zurückliefert.

Gleich nach der Instanziierung der FigClass erneuert diese über einen Aufruf von modelChanged() ihre Darstellung. Dazu fragt sie den sie besitzenden Knoten des Graphen nach den benötigten Daten wie Name, Attribute, Methoden, Größe und Position.

Abschließend wird die neu erstellt FigClass vom LayerPerspective aufgefordert, sich im entsprechenden Graphics-Kontext zu zeichnen.

Fall 3: Objektentfernung auf PIROL-Seite

Wird auf PIROL-Seite eine Klasse aus der Liste der Classifier eines Subsystems entfernt, erhält der angemeldete ListChangeObserver die Nachricht do_delete(Object o) gesandt und ruft daraufhin die dispose() Methode des entsprechenden ClassMediators auf. Dieser feuert ein "disposed" Event und benachrichtigt somit die darstellende FigClass. Diese ruft ihre delete-Methode auf, die sich an die sie darstellende Ebene wendet, um sich daraus zu entfernen. Am Ende des Nachrichtenflusses wird der Knoten, dessen Visualisierung gerade aus der LayerPerspective entfernt wurde, auch aus dem UFAGraphModel gelöscht.

Fall 4: Objektveränderung auf GEF-Seite

Im GEF-Framework ist die Manipulation von Graph-Elementen nur insoweit vorbereitet, als es sich um die Änderung der grafischen Eigenschaften des Elementes handelt, wie z.B. deren Position. Alle applikations-spezifischen Veränderungen der Graph-Elemente (wie das Ändern des Namens einer Klasse) wurden so implementiert, dass das entsprechende Attribut des zugrundeliegenden Proxy-Objektes geändert wird. Die Aktualisierung der Repräsentation auf der GEF-Seite erfolgt dann wie in Fall 1 beschrieben.

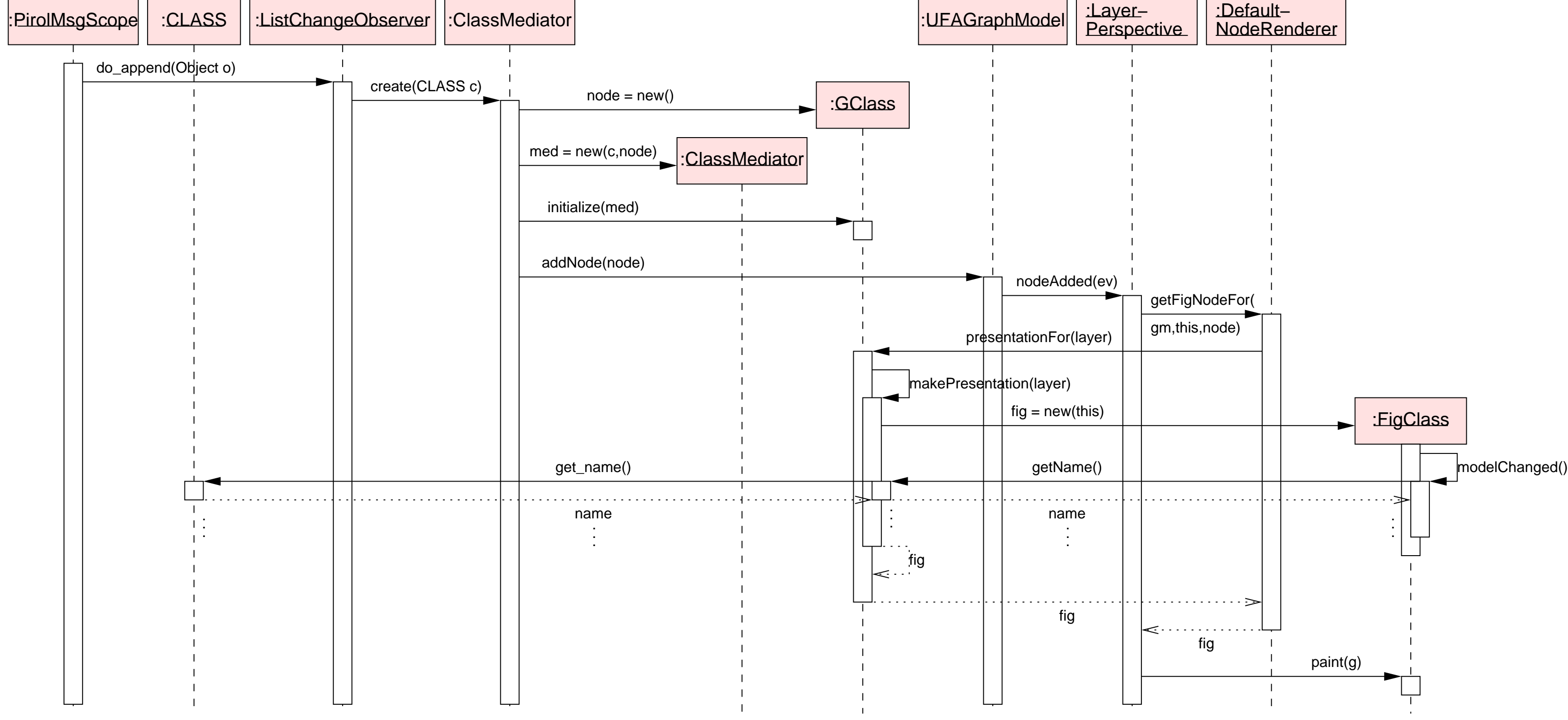


Abbildung 4.15: Nachrichtenfluss beim Erstellen einer neuen CLASS auf PIROL-Seite

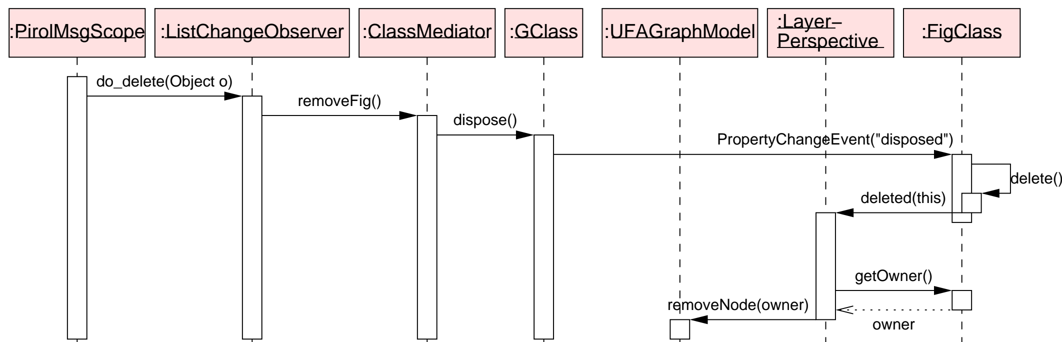


Abbildung 4.16: Nachrichtenfluss beim Entfernen einer CLASS auf PIROL-Seite

Dieses Vorgehen wurde gewählt, um die Implementierung flexibel zu halten und nicht ohne Gewinn aufzublähen. Die Alternative hätte zudem erfordert, dass alle durch das Repository sichergestellten Konsistenzbedingungen auch auf GEF-Seite überprüft werden. Andernfalls könnte eine Änderung auf GEF-Seite vorgenommen und im Editor bereits angezeigt werden, die beim Propagieren an das Repository abgelehnt wird.

Fall 5: Objekterzeugung auf GEF-Seite

Die Objekterzeugung im Editor erfolgt durch die vom Framework bereitgestellte Java Action-Klasse CmdCreateNode (bzw. CmdCreateEdge). Diese kann mit unterschiedlichen zu erzeugenden Elementen instanziiert und zum Beispiel in die Werkzeugleiste eingebettet werden.

Wird die Action aktiviert, beispielsweise für einen zu erzeugenden Knoten vom Typ GClass, so lässt sie den Editor in den Modus ModePlace wechseln.

Dieser wartet zunächst darauf, dass die linke Maustaste heruntergedrückt wird. Dann erfragt ModePlace vom CmdCreateNode eine Instanz des neuen Knotens, die dieses über newInstance() erzeugt. Da an dieser Stelle keine Argumente wie beim Erzeugen durch einen Konstruktor übergeben werden können, werden sie über den Aufruf der initialize-Methode nachgereicht. In dieser Methode erzeugt sich der Knoten GClass unter anderem den für ihn benötigten Mediator vom Typ ClassMediator.

ModePlace erfragt dann vom GraphNodeRenderer des Editors eine Repräsentation für den neuen Knoten und stellt diese dar. Wird die Maus bei gedrückter Maustaste bewegt, ändert ModePlace die Position des angezeigten Grafikelementes. Erst beim Loslassen der Maustaste fügt ModePlace den neuen Knoten dem GraphModel hinzu.

Zuletzt bekommt der Knoten von ModePlace die Nachricht postPlacement() gesandt. Erst an dieser Stelle fordert GClass seinen Mediator abschließend dazu auf, ein neues VO vom Typ CLASS auf der PIROL-Seite zu erzeugen.

Fall 6: Objektentfernung auf GEF-Seite

Das Entfernen von Graph-Elementen geschieht durch die Java Action-Klasse Cmd-Dispose, die u.a. an die Delete-Taste gebunden ist. Wird sie aktiviert, ruft sie die dispose-Methode des SelectionManagers auf. Dieser ruft seinerseits auf allen derzeit selektierten Figs die dispose-Methode auf.

Von FigClass::dispose() wird die Anfrage an den dazugehörigen Knoten GClass im Graphen gesandt. Dieser benachrichtigt alle angemeldeten PropertyChangeListener – und damit auch den dazugehörigen ClassMediator – mit einer "disposed"-Nachricht.

Der ClassMediator seinerseits entfernt das betreffende CLASS-VO aus der Classifier-Liste des umschließenden Subsystems.

4.5 Beschreibung des UFA-Editors UFAEd

In Abbildung 4.17 ist der UFA-Editor UFAEd mit einem aus dem Repository geladenen Diagramm dargestellt. Das Fenster ist zweigeteilt und beinhaltet ganz oben eine Menüleiste und eine Werkzeugleiste. Darunter befindet sich die Zeichenfläche und am unteren Rand ein Bereich, der die Details des gerade selektierten Elements anzeigt.

Mit den grafischen Elementen der Zeichenfläche kann komfortabel gearbeitet werden. Knoten und Kanten können wie von anderen grafischen Editoren gewohnt erstellt, verändert und entfernt werden. Die Bedienung ist erfreulich unkompliziert.

4.5.1 Werkzeugleiste

Viele der benötigten Funktionen sind über die Werkzeugleiste (Abb. 4.18) zu erreichen. Ob die einzelnen Knöpfe der Werkzeugleiste aktiviert oder deaktiviert sind hängt von den aktuell selektierten grafischen Elementen ab.

Befindet sich beispielsweise in der Menge der selektierten Elemente kein einziges Element, das seine Attribute versteckt, ist entsprechend die Schaltfläche "Show Attributes" deaktiviert, da bereits alle Attribute angezeigt werden. Die Funktionen sind von links nach rechts im Einzelnen:

Select wechselt in den Selektionsmodus.

Broom wechselt in den Broom-Modus, der das Herumschieben der Grafikelemente auf der Zeichenfläche erlaubt.

Transfer Layout überträgt die Höhe und die Breite des aktuell gewählten Elementes auf ein anderes.

Subsystem erstellt eine neues Paket.

Class erstellt eine neue Klasse.

Abbildung 4.17: UFAEd mit geladenem Diagramm

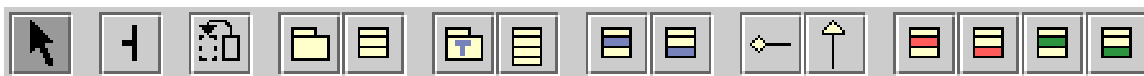


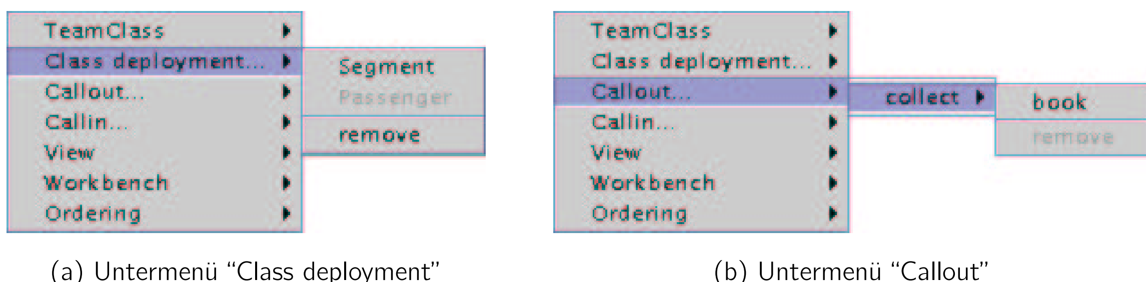
Abbildung 4.18: UFAEd Werkzeugleiste

Team	erstellt ein neues Team.
TeamClass	erstellt eine neue Teamklasse.
Add Attribute	fügt ein Attribut zum selektierten Element hinzu.
Add Method	fügt eine Methode zum selektierten Element hinzu.
Association	legt eine Assoziation durch Ziehen einer Kante zwischen zwei Elementen an.
Generalization	legt eine Generalisierung durch Ziehen einer Kante zwischen zwei Elementen an.
Hide Attributes	blendet die Attribute der selektierten Elemente aus.
Hide Methods	blendet die Methoden der selektierten Elemente aus.
Show Attributes	zeigt die Attribute der selektierten Elemente wieder an.
Show Methods	zeigt die Methoden der selektierten Elemente wieder an.

4.5.2 Kontextmenü

Das Kontextmenü ist über die rechte Maustaste zu erreichen. Es enthält je nach gewähltem Element die dazu passenden Einträge. Das Workbenchmenü (vgl. Abschnitt 3.2.1) wurde in das Kontextmenü aufgenommen und ist auch zugänglich, wenn lediglich auf die leere Zeichenfläche geklickt wird.

Als Beispiel sei hier nur das Kontextmenü der Teamklassen auszugsweise beschrieben (Abb. 4.19). Es enthält einen Eintrag "Class Deployment", der aktiviert ist, sobald das Team in dem sich die Teamklasse befindet an ein Basispaket gebunden wurde. Das über diesen Eintrag zu erreichende Untermenü enthält dann Einträge für alle in den adaptierten Paketen enthaltenen Klassen. Wird einer dieser Einträge ausgewählt, wird die Teamklasse an die gewählte Basisklasse gebunden.



(a) Untermenü "Class deployment" (b) Untermenü "Callout"

Abbildung 4.19: Kontextmenü der Teamklasse Subscriber aus Abb. 4.17

Abbildung 4.19(a) zeigt das Kontextmenü für die Teamklasse Subscriber aus Abbildung 4.17. Da die Teamklasse bereits an die Basisklasse Passenger aus dem Paket FlightBooking gebunden ist, wird dieser Eintrag deaktiviert dargestellt. Durch den Menüpunkt "remove" kann die Bindung von Subscriber an Passenger entfernt werden und durch den Eintrag "Segment" an die gleichnamige Klasse gebunden werden.

In entsprechender Weise funktionieren auch die beiden Einträge "Callin" und "Callout" des Kontextmenüs, die in Abbildung 4.19(b) zu sehen sind. Erst wenn die Teamklasse an eine Basisklasse gebunden wurde sind sie aktiviert. Dann kann die zu erstellende oder zu löschende Bindung komfortabel aus den jeweils möglichen ausgewählt werden.

4.5.3 Details

Im unteren Bereich des Editors werden die Details des aktuell selektierten Elementes angezeigt, in Abbildung 4.17 zum Beispiel die des Teams Bonus. Durch die dargestellten Textfelder lassen sich die jeweiligen Attribute direkt verändern.

Über die angezeigten Listen der im Team enthaltenen Klassen, Attribute usw. kann die Detailsicht zu diesen Elementen navigieren und deren Details anzeigen.

Wird beispielsweise über die Liste der enthaltenen Classifier zur Teamklasse BonusItem navigiert, werden die in Abbildung 4.20 gezeigten Details dargestellt. Nun ist auch der "Back"-Knopf aktiviert, der die Detailsicht wieder auf das Ausgangselement fokussiert.

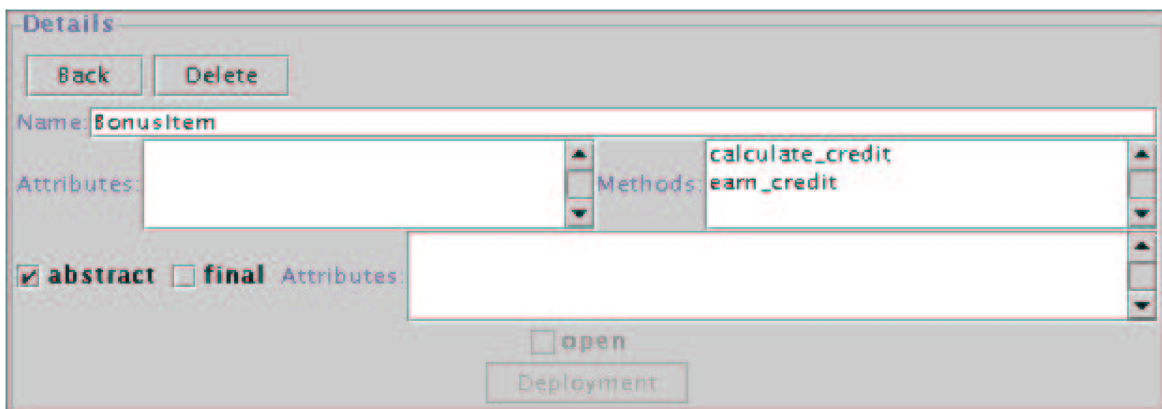


Abbildung 4.20: Details zur Teamklasse BonusItem des Teams Bonus

Das Formular der Details ist so generisch wie möglich realisiert worden und dadurch sehr einfach erweiterbar. Der Klasse DetailPanel kann ein beliebiges Objekt vom Typ ANY oder dessen Spezialisierungen übergeben werden. Das Formular wird dann zeilenweise entlang der Vererbungshierarchie des Metamodells mit geeigneten Bedienelementen gefüllt.

Da jeder übergebene Proxy eine Instanz von ANY ist, wird dem Formular als erstes ein Textfeld zum Editieren des Attributes "name" hinzugefügt. Ist der übergebene Proxy darüber hinaus eine Instanz von CLASSIFIER, so werden dem Detail-Formular u.a. Einträge für dessen Attribute und Methoden angehängt, was in diesem Fall sowohl für CLASS als auch für TEAM_CLASS zutrifft.

Auch die Bedienelemente wurden möglichst generisch ausgelegt. Ihnen wird bei ihrer Erzeugung der Proxy übergeben, auf den sich z.B. die Änderung eines Textfeldes auswirken soll. Außerdem müssen die zum Auslesen und Setzen zu verwendenden Methoden angegeben werden. Das spezialisierte Textfeld kann sich dann anhand dieser Informationen über den Java Reflections Mechanismus initialisieren und ebenso die erfolgten Änderungen weiterleiten.

Entsprechend spezialisierte Bedienelemente existieren in Form von CheckBoxes für Boolesche Attribute, in Form von ListBoxes für Listenattribute und in Form von ComboBoxes für Integer Attribute, die über eine Dropdown-Liste ihrer textuellen Beschreibung ausgewählt werden sollen (z.B. für die Sichtbarkeitsattribute von Methoden).

4.6 Fazit zum Einsatz von GEF

Die Bewertung des Einsatzes von GEF fällt zwiespältig aus. Auf der einen Seite konnte mit dem Framework ein mächtiger grafischer Editor realisiert werden, dessen grundlegende Funktionen auf GEF aufbauen. Auf der anderen Seite gab es jedoch einige Hindernisse, die den Einsatz sehr erschwert haben.

4.6.1 Positives

GEF stellt ein durchdachtes Grundgerüst für grafische Graph-Editoren bereit. Die gewählten Abstraktionen sind durchweg sinnvoll und die Geschwindigkeit ist für eine grafische Java-Applikation bemerkenswert hoch. Die verwirklichten Konzepte, wie beispielsweise die Layer, bieten eine gute Flexibilität und die bereits im Framework realisierte Funktionalität ist groß.

Ohne die einzelnen Funktionen von GEF an dieser Stelle erneut aufzuzählen bleibt die Liste der positiven Erfahrungen relativ kurz. Daher sei ausdrücklich betont, dass dies natürlich in keinem Verhältnis zu dem großen Nutzen steht, der aus der Verwendung von GEF erwachsen ist.

4.6.2 Kritikpunkte

Zwei Punkte haben den Einsatz von GEF am stärksten behindert: die praktisch nicht vorhandene Dokumentation und die unklare Struktur.

Die fehlende Dokumentation erlaubt die Verwendung von GEF nur bei intensivem Studium des Quelltextes. Gerade bei der Realisierung der Anbindung musste der Nachrichtenfluss immer wieder bis ins Detail nachvollzogen werden, um geeignete Andockstellen ausfindig zu machen. Fehlende Kommentare, einige unvollständig implementierte Methoden und Fehler erschwerten die Arbeit zusätzlich.

Damit einher geht eine teils unklare Strukturierung, die daher rührt, dass GEF zunächst mit ArgoUML verschmolzen war und erst später wieder herausgelöst worden ist. Die Grenzen zwischen den unterschiedlichen Schichten sind nicht klar erkennbar, weil sie nicht überall eingehalten worden sind. So werden an einigen Stellen des Frameworks Annahmen gemacht, die zwar in ArgoUML gelten mögen, in dieser Form für das Framework jedoch falsch sind.

Im Laufe der Entwicklung des Editors konnten einige Fehler des Frameworks behoben und in den aktuellen Stand integriert werden. Es stellte sich leider heraus, dass es zu GEF selbst keine aktive Entwicklergemeinde mehr gibt. Zwar werden noch Änderungen vorgenommen, diese gehen jedoch meist auf Anforderungen des ArgoUML-Projektes zurück, mit der Folge, dass die Weiterentwicklung von GEF als unabhängiges Framework stagniert.

Wohl auch aus diesem Grund wurden noch nicht alle Fehlerkorrekturen in den aktuellen Stand von GEF aufgenommen und müssen daher weiterhin explizit umgangen werden.

4.6.3 Fazit

Erst der Einsatz von GEF hat den grafischen Editor in dieser Form möglich gemacht. Es wäre lohnenswert, das Framework gründlich aufzuräumen und von Fehlern zu befreien, denn die enthaltene Substanz ist leistungsfähig und könnte sauber strukturiert die Grundlage für viele grafische Editoren werden.

5 Zusammenfassung und Ausblick

5.1 Zusammenfassung

Das Ziel dieser Arbeit war es, die Bedeutung aspektorientierter Techniken im Softwareentwurf zu erarbeiten und einen grafischen Editor für den aspektorientierten Entwurf zu entwickeln, der sich möglichst nahtlos in die repository-basierte Softwareentwicklungsumgebung PIROL einbettet.

Dazu wurde zunächst das Paradigma der aspektorientierten Programmierung erläutert und anschließend vier verschiedene Umsetzungen desselben dargelegt. Eines davon war das für die weitere Arbeit verwendete Modell der Aspectual Collaborations.

Im zweiten Teil wurde die Bedeutung des Softwareentwurfes für die Softwareentwicklung erarbeitet und die Notwendigkeit begründet, dass auch die zunächst als reine Programmiertechnik aufgekommene Aspektorientierung im Softwareentwurf berücksichtigt werden muss. Hieran anschließend wurden Aspekte im Entwurf genauer ausgelegt und UFA vorgestellt, eine für den aspektorientierten Entwurf nach dem Aspectual Collaborations Modell geeignete Erweiterung der UML.

Das dritte Kapitel behandelte die für den praktischen Teil dieser Arbeit relevante Softwareentwicklungsumgebung PIROL und bewegte sich dabei entlang allgemeiner Integrationskriterien für Softwareentwicklungsumgebungen. Es wurden die Rahmenbedingungen und die zur Verfügung stehenden Mittel zur Einbindung von Werkzeugen analysiert.

Der vierte Teil beschrieb die Entwicklung des grafischen Editors für UFA-Diagramme. Es wurden zunächst die nötigen Erweiterungen des PIROL-Metamodells erarbeitet, um dann das zur Realisierung eingesetzte Graph-Editor Framework GEF und dessen Anbindung an das PIROL Repository zu erläutern. Abschließend wurde ein Fazit zum eingesetzten Framework GEF gezogen und der grafische Editor UFAEd präsentiert.

5.2 Ausblick

Von dieser Arbeit ausgehend lassen sich mindestens vier Aufgabenfelder für weiterführende Untersuchungen mit unterschiedlichen Schwerpunkten identifizieren.

Zunächst müssen Erfahrungen mit dem aspektorientierten Entwurf mit Hilfe von UFA gesammelt werden. Dabei geht es zum einen um die Notation selbst: Hilft sie effektiv, die aspektorientierten Entwürfe kommunizieren zu können? Noch wichtiger ist es, aus den gesammelten Erfahrungen Heuristiken und Richtlinien für den aspektorientierten Entwurf an sich zu formulieren, bis hin zu aspektorientierten Entwurfsmustern.

Wie gut funktioniert der Übergang vom aspektorientierten Entwurf zum aspektorientierten Code? Welche Stellen der aspektorientierten Programme bedürfen einer besonderen Berücksichtigung im Entwurf? Eine notwendige Voraussetzung zur Beantwortung dieser Fragen entsteht derzeit im Rahmen von zwei Diplomarbeiten, die das Object Teams Modell für Java umsetzen (Binder, 2002; Hundt, 2002).

Im Zusammenhang mit PIROL stellen sich Aufgaben hinsichtlich der Integration des UFA Editors mit anderen Werkzeugen. Beispielsweise wurden in einer vor kurzem abgeschlossenen Diplomarbeit von Christian Mattick (2002) die Voraussetzungen für das Editieren von Quelltexten in PIROL geschaffen. Es bleibt daher noch ein Sprachmodul für die Object Teams Erweiterung von Java zu erstellen, um den nahtlosen Übergang vom Entwurf zur Implementierung näher untersuchen und die Werkzeugunterstützung optimieren zu können.

Als viertes sind Verbesserungen des Editors selbst möglich, deren Zielrichtungen sich beim konkreten Einsatz ergeben werden.

A Literaturverzeichnis

Aksit und Bergmans 2001
Aksit, M. ; Bergmans, L.: Composing Crosscutting Concerns Using Composition Filters. In: *Communications of the ACM* 44 (2001), Oktober, Nr. 10, S. 51–57

ArgoUML
ArgoUML: *ArgoUML – A java based cognitive CASE tool.* – URL `http://argouml.tirgris.org`

Bansiya und Davis 2002
Bansiya, J. ; Davis, C. G.: A Hierarchical Model for Object-Oriented Design Quality Assessment. In: *IEEE Transactions on Software Engineering* 28 (2002), Januar, Nr. 1, S. 4–17

Binder 2002
Binder, Christof: *Aspectual Collaborations: Erweiterung des Java-Compilers für verbesserte Modularität durch aspekt-orientierte Techniken*, Technische Universität Berlin, Diplomarbeit, 2002

Boehm und Basili 2001
Boehm, B. ; Basili, V. R.: Software Defect Reduction Top 10 List. In: *IEEE Computer* 34 (2001), Januar, Nr. 1, S. 135–137

Buschmann u. a. 1996
Buschmann, F. ; Meunier, R. ; Rohnert, H. ; Sommerlad, P. ; Stal, M.: *Software Design Patterns*. Bd. 1: *Pattern-Oriented Software Architecture – A System of Patterns*. John Wiley & Sons Ltd., 1996

Constantinides und Skotiniotis 2002
Constantinides, A. C. ; Skotiniotis, T.: Reasoning about a classification of crosscutting concerns in object-oriented systems. In: Constanza, P. (Hrsg.) ; Kniesel, G. (Hrsg.) ; Mehner, K. (Hrsg.) ; Pulvermüller, E. (Hrsg.) ; Speck, A. (Hrsg.): *Second Workshop on Aspect-Oriented Software Development*. Bonn, Germany, Februar 2002, S. 1–6

Dijkstra 1976
Dijkstra, E. W.: *A discipline of Programming*. Prentice-Hall, 1976

ECMA TR/55 1993
ECMA TR/55: Reference Model for Frameworks of Software Engineering Environments / European Computer Manufacturers Association (ECMA). 1993 (TR/55). – Forschungsbericht

Elrad u. a. 2001
Elrad, T. ; Filman, R.E. ; Bader, A.: Aspect Oriented Programming. In: *Communications of the ACM* 44 (2001), Oktober, Nr. 10, S. 28–32

Faulk 1997
Faulk, S. R.: Software Requirements: A Tutorial. In: Dorfman, M. (Hrsg.) ; Thayer, R. H. (Hrsg.): *Software Engineering*. IEEE Computer Society Press, 1997, S. 82–103

GEF
GEF: *Java Graph Editing Framework*. – URL `http://gef.tirgris.org`

Groth u. a. 1995
Groth, B. ; Herrmann, S. ; Jähnichen, S. ; Koch, W.: Project Integrating Reference Object Library (PIROL). In: *Proc. of the 7th Conference on Software Engineering Environments (SEE '95)*, IEEE Computer Society Press, 1995

Harrison und Ossher 1993
Harrison, W. ; Ossher, H.: Subject-Oriented Programming (A Critique of Pure Objects). In: *Proc. of OOPSLA '93*, ACM Press, 1993, S. 411–428

Herrmann 2000
Herrmann, S.: Lua/P – A Repository Language for Flexible Software Engineering Environments. In: *Proc. of The Second International Symposium on Constructing Software Engineering Tools (CoSET 2000)*, 2000, S. 77–86

Herrmann 2002a
Herrmann, S.: *Composable Designs with UFA*. Workshop on Aspect-Oriented Modeling with UML at AOSD 2002. April 2002

Herrmann 2002b
Herrmann, S.: *Object Teams: Improving Modularity for Crosscutting Collaborations*. Submitted to: Net.ObjectDays 2002. 2002

Herrmann und Mezini 2000
Herrmann, S. ; Mezini, M.: PIROL: A Case Study for Multidimensional Separation of Concerns in Software Engineering Environments. In: *Proc. of OOPSLA 2000*, ACM Press, 2000, S. 188–207

Herrmann und Mezini 2001a
Herrmann, S. ; Mezini, M.: *Aspect-Oriented Software Development with Aspectual Collaborations*. Submitted to ECOOP 2001. 2001

Herrmann und Mezini 2001b
Herrmann, S. ; Mezini, M.: Combining Composition Styles in the Evolvable Language LAC. In: *Proc. of ASoC workshop at the 23rd ICSE*, IEEE Computer Society Press, 2001

Herrmann und Mezini 2001c
Herrmann, S. ; Mezini, M.: Connectors for bridging mismatches between the components of a software engineering environment. In: *IEE Proceedings – Software Engineering* 148 (2001), Juni, Nr. 3

Hundt 2002
Hundt, Christine: *Bytecode-Transformation zur Laufzeitunterstützung von aspektorientierter Modularisierung mit Object-Teams/Java*, Technische Universität Berlin, Diplomarbeit, 2002

IEEE Std 1061-1998 1998
IEEE Std 1061-1998: *IEEE Standard for a Software Quality Metrics Methodology*. 1998

IEEE Std 610.12-1990 1990
IEEE Std 610.12-1990: *IEEE Standard Glossary of Software Engineering Terminology*. 1990

ISO/IEC 9126-1:2001 2001
ISO/IEC 9126-1:2001: *Software Engineering – Product quality – Part 1: Quality Model*. 2001

Jacobsen u. a. 1998
Jacobsen, E. E. ; Kristensen, B. B. ; Nowack, P.: Models, Domains and Abstraction in Software Development. In: *Proc. of TOOLS-27'98*, IEEE Computer Society Press, 1998

Kiczales u. a. 2001a
Kiczales, G. ; Hilsdale, E. ; Hugunin, J. ; Kersten, M. ; Palm, J. ; Griswold, W. G.: Getting Started wit AspectJ. In: *Communications of the ACM* 44 (2001), Oktober, Nr. 10, S. 59–73

Kiczales u. a. 2001b
Kiczales, G. ; Hilsdale, E. ; Hugunin, J. ; Kersten, M. ; Palm, J. ; Griswold, W. G.: An Overview of AspectJ. In: *ECOOP*, 2001, S. 327–353

Kiczales u. a. 1997
Kiczales, G. ; Lamping, J. ; Menhdhekar, A. ; Maeda, C. ; Lopes, C. ; Loingtier, J. ; Irwin, J.: Aspect-Oriented Programming. In: Akşit, Mehmet (Hrsg.) ; Matsuoka, Satoshi (Hrsg.): *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland* Bd. 1241. Springer-Verlag, 1997, S. 220–242

Lieberherr u. a. 1999
Lieberherr, K. ; Lorenz, D. ; Mezini, M.: Programming with Aspectual Components / Northeastern University. April 1999. – Forschungsbericht

Lieberherr u. a. 2001
Lieberherr, K. ; Orleans, D. ; Ovlinger, J.: Aspect-Oriented Programming with Adaptive Methods. In: *Communications of the ACM* 44 (2001), Oktober, Nr. 10, S. 39–41

Mattick 2002
Mattick, Christian: *Das Editieren von Quelltexten in einer Softwareentwicklungsumgebung (PIROL) mit einem einheitlichen Repository*, Technische Universität Berlin, Diplomarbeit, Juni 2002

McCall 1977
McCall, J. (Hrsg.): *Factors in Software Quality*. General Electric. 1977. – Forschungsbericht

Meyer 1997
Meyer, B.: *Object-oriented software construction*. Prentice Hall, 1997

Ossher 2002
Ossher, H.: 1st International Conference on Aspect-Oriented Software Development. Enschede, Netherlands : Ossher, H., April 2002. – URL `http://trese.cs.utwente.nl/aosd2002/index.php`. – Zugriffsdatum: 6.3.2002

Pressman 1997
Pressman, R. S.: *Software Engineering: A Practitioner's Approach*. 4th. McGraw-Hill, 1997

Siewczynski 2002
Siewczynski, Sven: *Web-Präsentation von Projektmanagementinformationen aus der repository-basierten Softwareentwicklungsumgebung PIROL*, Technische Universität Berlin, Diplomarbeit, 2002

Tarr und Ossher 2000
Tarr, P. ; Ossher, H.: *Hyper/J User and Installation Manual*. IBM Research. 2000

Tarr u. a. 1999
Tarr, Peri L. ; Ossher, Harold ; Harrison, William H. ; Sutton Jr., Stanley M.: N Degrees of Separation: Multi-Dimensional Separation of Concerns. In: *International Conference on Software Engineering*, 1999, S. 107–119

Wasserman 1989
Wasserman, A. I.: Tool Integration in Software Engineering Environments. In: Long, Fred (Hrsg.): *Software Engineering Environments, International Workshop on Environments Proceedings*, Springer-Verlag, 1989, S. 137–149

Weber 2001
Weber, B.: *Graphische Editoren für die repository-basierte Softwareentwicklungsumgebung PIROL durch Erweiterung eines bestehenden Frameworks*, Technische Universität Berlin, Diplomarbeit, 2001

B Anhang

B.1 QMOOD Definitionen

Quality Attribute	Definition
Reusability	Reflects the presence of object-oriented design characteristics that allow a design to be reapplied to a new problem without significant effort.
Flexibility	Characteristics that allow the incorporation of changes in a design. The ability of a design to be adapted to provide functionally related capabilities.
Understandability	The properties of the design that enable it to be easily learned and comprehended. This directly relates to the complexity of the design structure.
Functionality	The responsibilities assigned to the classes of a design, which are made available by the classes through their public interfaces.
Extendibility	Refers to the presence and usage of properties in an existing design that allow for the incorporation of new requirements in the design.
Effectiveness	This refers to a design's ability to achieve the desired functionality and behavior using object-oriented design concepts and techniques.

Tabelle B.1: QMOOD Qualitätsattribute des Entwurfes

Design Property	Definition
Design Size	A measure of the number of classes used in a design.
Hierarchies	Hierarchies are used to represent different generalization-specialization concepts in a design. It is a count of the number of non-inherited classes that have children in a design.
Abstraction	A measure of the generalization-specialization aspect of the design. Classes in a design which have one or more descendants exhibit this property of abstraction.
Encapsulation	Defined as the enclosing of data and behavior within a single construct. In object-oriented designs the property specifically refers to designing classes that prevent acces to attribute declarations by defining them to be private, thus protecting the internal representation of the objects.
Coupling	Defines the interdependency of an object on other objects in a design. It is a measure of the number of other objects that would have to be accessed by an object in order for that object to function correctly.
Cohesion	Assesses the relatedness of methods and attributes in a class. Strong overlap in the method parameters and attribute types is an indication of string cohesion.
Composition	Measures the "part-of", "has", "consists-of", or "part-whole" relationships, which are aggregation relationships in an object-oriented design.
Inheritance	A measure of the "is-a" relationship between classes. This relationship is related to the level of nesting of classes in an inheritance hierarchy.
Polymorphism	The ability to substitute objects whose interfaces match for one another at run-time. It is a measure of services that are dynamically determined at run-time in an object.
Messaging	A count of the number of public methods that are available as services to other classes. This is a measure of the services that a class provides.
Complexity	A measure of the degree of difficulty in understanding and comprehending the internal and external structure of classes and their relationships.

Tabelle B.2: QMOOD Entwurfs-Eigenschaften

Metrik	Name	Definition
DSC	Direct Size in Classes	This metric is a count of the total number of classes in the design.
NOH	Number of Hierarchies	This metric is a count of the number of class hierarchies in the design.
ANA	Average Number of Ancestors	This metric value signifies the average number of classes from which a class inherits information. Informationstechnik is computed by determining the number of classes along all paths from the “root” class(es) to all classes in an inheritance structure.
DAM	Data Access Metric	This metric is the ratio of the average number of private (protected) attributes to the total number of attributes declared in the class. A high value for DAM is desired (Range 0 to 1).
DCC	Direct Class Coupling	This metric is a count of the different number of classes that a class is directly related to. The metric includes classes that are directly related by attribute declarations and message passing (parameters) in methods.
CAM	Cohesion Among Methods of Class	This metric computes the relatedness among methods of a class based upon the parameter list of the methods. The metric is computed using the summation of the intersection of parameters of a method with the maximum independent set of all parameter types in the class. A metric value close to 1.0 is preferred. (Range 0 to 1)
MOA	Measure of Aggregation	This metric measures the extent of the part-whole relationship, realized by using attributes. The metric is a count of the number of data declarations whose types are user defined classes.
MFA	Measure of Functional Abstraction	This metric is the ratio of the number of methods inherited by a class to the total number of methods accessible by member methods of the class. (Range 0 to 1)
NOP	Number of Polymorphic Methods	This metric is a count of the methods that can exhibit polymorphic behavior. Such methods in C++ are marked as virtual.
CIS	Class Interface Size	This metric is a count of the number of public methods in a class.
NOM	Number of Methods	This metric is a count of all the methods defined in a class.

Tabelle B.3: QMOOD Entwurfs-Metriken