Abschlußbericht der Projektgruppe PG-HEU (326)

Christoph Begall
Matthias Dorka
Adil Kassabi
Wilhelm Leibel
Sebastian Linz
Sascha Lüdecke
Andreas Schröder
Jens Schröder
Sebastian Schütte
Thomas Sparenberg
Christian Stücke
Martin Uebing

Klaus Alfert
Alexander Fronk
Ernst-Erich Doberkat

Oktober 1999

Inhaltsverzeichnis

Teil I.

Einführung in die Projektgruppe *HEU*

1. Einleitung

Autoren: *Klaus Alfert*
Alexander Fronk
Ernst-Erich Doberkat

Die Projektgruppe soll sich mit der Problematik verteilter Hypermedia-Systeme (HMS) befassen. Dabei sind die grafische Spezifikation von solchen Systemen und Aspekte ihrer verteilten Konstruktion zu betrachten.

1.1. Aufgabenstellung und Motivation

Hyperdokumente sind solche Dokumente, die nicht-linear gelesen werden können. Ihr Inhalt muß nicht notwendig nur aus Text bestehen, sondern kann auch andere *Medienobjekte* wie Grafik, Animation, Ton(-Sequenzen), Video usw. umfassen. Die Konstruktion solcher Dokumente zu ihrer DV-unterstützten Nutzung sollte nicht durch triviales „Copy-Paste-Editing“ auf HTML-Ebene erfolgen, sondern vielmehr Werkzeug-gestützt ablaufen: große, zusammengehörige Datenmengen, die zum einen konsistent dargestellt, zum anderen in einer möglichst gleichförmig präsentiert werden sollen, erfordern die Wahrung des Überblicks über die Daten selbst, über ihre Präsentation und insbesondere über ihre nicht-lineare Verknüpfung. Zudem sollten diese Datenmengen unterschiedlichen Betrachtern unter verschiedenen Sichten dargeboten werden können, ohne für jede Nutzergruppe ein eigenes Dokument manuell erzeugen zu müssen.

Möchte man das werkzeuggestützte Konstruieren von Hyperdokumenten unterstützen, ist die Formalisierung der Hyperdokumente unbedingt erforderlich.

Eine solche Formalisierung bezieht sich zum einen auf die Präsentationssstruktur jedes Medienobjekts. Sie muß so festgelegt sein, daß Inhalte und Struktur unabhängig voneinander sind. Zum anderen müssen die Verknüpfungen (*Links*) von Inhalten definiert sein. Auch dies sollte auf eine Weise geschehen, die flexibel gegenüber Änderungen am Inhalt – und damit an Veränderungen von Links – ist, aber dennoch einer Formalisierung und maschineller Generierung von Links standhält. Diese Trennung zwischen Struktur und Inhalt folgt einem gängigen softwaretechnischen Prinzip, dem *separation of concerns*.

Man kann die Struktur der Verknüpfung von Inhalten etwas genauer betrachten, indem man diese durch Graphen abstrahiert. Links besitzen einen Start- und Zielpunkt, die *Anker* genannt werden. Jeden Anker interpretiert man als Knoten eines Graphen. Knoten sind durch Links, also Kanten, miteinander verknüpft. Die Ankerknoten können (und müssen) mit Attributen annotiert werden, die beispielsweise aussagen, in welchem

Medienobjekt sich der Anker befindet. Die Spezifikation dieser i.a. nicht zusammenhängenden, gerichteten und attributierten Graphen erlaubt die am Lehrstuhl für Software-Technologie entworfene objekt-orientierte Sprache *DoDL*. Mit ihrer Hilfe werden Struktur und Durchlaufverhalten eines Hyperdokuments spezifiziert; ein Compiler erzeugt daraus den Graphen und mit gegebenen Medienobjekten eine Menge von Seiten, die durch einen Browser interpretierbar sind.

Ein Anwendungsfeld für die Spezifikation großer, nicht-trivialer Hyperdokumente kann das System IMIS (Integriertes Meß- und Informations-System) des Bundesamts für Strahlenschutz liefern. Dieses System dient der Sammlung und Präsentation spezieller länderspezifischer Umweltdaten, die an etwa 26000 Meßpunkten in Deutschland ermittelt, dort jeweils lokal verwaltet und landes- und bundesweit verteilt aufbereitet werden. Jedes Bundesland ist dabei autonom, kann also die Darstellung seiner Daten frei wählen. Der Bund wiederum kann Landesdaten auf seine spezifische Art und Weise präsentieren. Eine Entwicklungsumgebung zur Spezifikation eines solchen verteilten Systems könnte die individuellen Gegebenheiten der Länder berücksichtigen und gleichzeitig den Zugriff auf die Länderdaten seitens des Bundes steuern: als Schnittstelle zum Datenaustausch sollen *DoDL*-Klassen von den Ländern definiert werden, auf die der Bund benutzend zugreifen kann. Das erhält die Autonomie der Länder, erspart dem Bund jedoch ein erneutes Spezifizieren der Datenverknüpfung, falls Teile oder die gesamte Landespräsentation übernommen werden sollen.

In solche Entwicklungsprojekte sind normalerweise viele Personen involviert, die zudem räumlich verteilt sind. Bei Projekten der hier vorgestellten Größe muß eine Vielzahl von Mitarbeitern der betroffenen Institutionen hinsichtlich der Anforderungen befragt werden, ferner müssen mehrere Systemanalytiker parallel an der Spezifikation arbeiten. Durch die parallele Arbeit können redundante oder inkonsistente Anforderungen auftreten. Idealerweise sollten daher die Analytiker selbst dann, wenn sie räumlich verteilt sind, auf einer einzigen logisch konsistenten Menge von Dokumenten operieren sowie untereinander und mit Vertretern der Länder und des Bundes konferieren können. Ein Ansatz zur Lösung dieser Probleme besteht in einer Entwicklungsumgebung mit netzwerkfähigen Editierwerkzeugen und der Einbindung von Konferenz-Systemen.

1.1.1. Ziele der PG

Betrachtet man realistische Anwendungsszenarien wie etwa das oben vorgestellte IMIS-Projekt, dann stellt man fest, daß die textuelle Spezifikation von HMS mit Hilfe einer formalen Sprache wenig angemessen ist. Hier wäre eine grafische Notation hilfreich, die die Struktur der HMS geeigneter widerspiegelt und automatisch in eine *DoDL*-Spezifikation übersetzt werden kann.

Im Rahmen der PG soll eine derartige grafische Notation zur Spezifikation komplexer Hypermediasysteme und eine dazu passende verteilte Entwicklungsumgebung (EU) erarbeitet werden. Das zentrale Tool der EU soll der grafische Editor für *DoDL* sein. Die Verwaltung der hypermedial zu präsentierenden Daten soll durch eine persistente Schicht der EU realisiert werden. Diese soll über eine transparente Verteilungsebene wie Java RMI erreicht werden. Eine Java-Schnittstelle zu H-PCTE-Datenbanken wird in Siegen

gerade realisiert, d.h. in Java geschriebene Werkzeuge können direkt auf der Objektbank arbeiten. Diese Trennung von Persistenz und Editor bietet eine leichte Erweiterbarkeit der EU für andere Tools, die auf den persistenten Daten arbeiten.

Der verteilten Spezifikation von Hyperdokumenten soll Rechnung getragen werden durch die Netzwerkfähigkeit des Editors auf Basis eines „aktiven“ OODBMS mit verteilter Notifizierung und der Einbindung von White-Board-Systemen zur Tele-Zusammenarbeit. Die prototypische Realisierung einer derart verteilten Umgebung für die ausgewählte grafische Notation wird als Teilziel angesehen.

In einem weiteren Schritt kann man sowohl die grafische Notation als auch die EU am IMIS-Beispiel auf ihre Praxistauglichkeit überprüfen. Die Verteilung der PG auf die beiden Standorte macht das Entwicklungszenario noch realistischer und erfordert einen angemessenen Entwurf der Kommunikationselemente.

1.2. Minimalziel

Das Minimalziel bildet die Konzeption der grafischen Notation, die darauf aufbauende Implementierung des Editors sowie die Realisierung einer Spezifikation eines verteilten Hypermediasystems im IMIS-Szenario. Während des ersten Semesters konzipiert ein Teil der PG die grafische Notation, die zweite Gruppe entwirft die hierfür nötigen Editierwerkzeuge. Die Implementierung dieser Werkzeuge geschieht gemeinsam. Für das zweite Semester ist die Validierung der grafischen Notation anhand der Spezifikation eines bereits oben erwähnten verteilten Hyperdokuments im IMIS-Szenario vorgesehen.

2. Geplante PG-Realisierung

Autoren: *Klaus Alfert*
Alexander Fronk

Die Projektgruppenarbeit kann grob in folgende Phasen aufgeteilt werden: *Einarbeitungsphase*, *Anforderungsanalysephase*, *Entwurfsphase*, *Spezifikationsphase*, *Implementierungsphase* und, als mögliche Erweiterung, eine *Anwendungsphase*. Diese Phasen laufen teilweise parallel. Folgende Abschnitte erläutern diese Phasen im Detail.

2.1. Einarbeitung

In Form von Seminarvorträgen durch die Projektgruppenteilnehmer wird die Projektgruppe an die zu lösende Aufgabe herangeführt. Dies dient der Aneignung des nötigen Fachwissens. Die Einarbeitung erfolgt in folgende Themenbereiche und Problemfelder:

- Hypermediasysteme
- (verteilte) Spezifikation von HMS
- *DoDL*
- Grafische Notationen (UML, etc.)
- White-Board-Systeme, Tele-Zusammenarbeit
- Verteilungsaspekte (CORBA, Java, RMI etc.)
- IMIS als Anwendungskontext

Weiterhin werden bestehende HMS und EUs betrachtet.

2.2. Anforderungsanalyse

In dieser Phase stellen die Teilnehmer die Anforderungen an die grafische Notation und den grafischen Editor auf. Dabei sind bestehende Systeme zur (verteilten) grafischen Spezifikation zu untersuchen und daraus Anforderungen an die Notation und den Editor abzuleiten.

2.3. Entwurf

Diese Phase bearbeiten die Teilnehmer in getrennten Gruppen. Eine Gruppe entwirft die grafische Notation und die andere Gruppe stellt den Entwurf für den grafischen Editor auf. Diese Tätigkeiten können allerdings nicht unabhängig voneinander durchgeführt werden, da sie sich teilweise gegenseitig beeinflussen. Aus diesem Grund ist eine gewisse Zusammenarbeit der Teilgruppen notwendig.

2.4. HMS-Spezifikation

Nach Abschluß des Entwurfs der grafischen Notation ist es die Aufgabe der entsprechenden Teilgruppe, mit der Spezifikation des HMS für das IMIS-Szenario zu beginnen. Für diese Spezifikation ist die entworfene grafische Notation zu verwenden, die dadurch validiert wird. Diese Phase der Spezifikation wird im zweiten Semester von allen Teilnehmern fortgeführt.

2.5. Implementierung

Diese Phase läuft für den Rest des ersten Semesters parallel zur Spezifikationsphase ab: die Teilnehmer, die den grafischen Editor entworfen haben, implementieren diesen.

2.6. Anwendung

In der Anwendungsphase wird die Validierung des implementierten Editors durchgeführt, indem das spezifizierte HMS für IMIS mit dem grafischen Editor verteilt konstruiert wird. Ein weiteres wichtiges Ziel dieser Phase ist die Sammlung von Ergebnissen, die für die Weiterentwicklung von *DoDL* von Nutzen sein können.

Die gesamten Arbeiten werden jeweils durch einen Zwischen- und einen Abschlussbericht dokumentiert.

2.7. Abschlußbericht

Der Abschlußbericht wird den gesamten Projektverlauf festhalten. Die Ergebnisse der einzelnen Phasen werden vorgestellt und bewertet.

2.8. Exkursion und Fachgespräch

Den Abschluß der Projektgruppe bildet eine Exkursion zu einem dem Projektgruppenthema angemessenen Ziel, der Universität-Gesamthochschule Siegen, um hier mit einer Expertengruppe über die PG-Erfahrungen zu diskutieren. Daneben wird ein Fachgespräch stattfinden, in dem die Projektgruppenteilnehmer den Fachbereich über den Ablauf und

die Ergebnisse der Projektgruppe informieren. Dieses Fachgespräch wird im Rahmen des Diplomanden- und Doktorandenseminars des LS 10 stattfinden.

2.9. Zeitlicher Ablauf

Folgender zeitlicher Ablauf ist geplant:

1. Einarbeitungsphase (2 Wochen)
2. Editor und grafische Notation
   - Anforderungsanalysephase (5 Wochen)
   - Entwurfsphase (4 Wochen)
   - Implementierung (5 Wochen)
3. HMS-Spezifikation (teilweise parallel zur Phase Implementierung, den Rest im zweiten Semester)
   - Einarbeitung in IMIS-Kontext (parallel zur Phase Implementierung) (2 Wochen)
   - Spezifikation des Hypermediasystems im IMIS-Kontext (teilweise parallel zur Phase Implementierung) (7 Wochen)
   - Validierung der grafischen Notation (4 Wochen)
4. Abschlußbericht (3 Wochen)

2.10. Erweiterungsmöglichkeiten

Zusätzlich kann über das Minimalziel hinaus in der Anwendungsphase die Validierung des Editors durch die Umsetzung der Spezifikation des Hyperdokuments erfolgen; aus der Validierung der grafischen Notation können Rückschlüsse auf *DoDL* gezogen werden, die zur Verbesserung der Sprache beitragen können.

2.11. Vorgehensmodell

Autor: *Wilhelm Leibel*

Bei der Vorgehensweise der Projektgruppe lag das Spiralmodell zugrunde (siehe Kapitel 14.2.1 auf Seite 120). Es wurde zunächst ein Prototyp erstellt, um Erfahrungen mit dem Umgang von grafischen Editoren zu sammeln. Außerdem wurden Erkenntnisse mit der Datenbank JHPcte erworben. Gleichzeitig analysierte eine Teilgruppe die Sprache *DoDL* und entwickelte daraus eine grafische Notation. Mit den gewonnenen Erkenntnissen wurde der Zyklus erneut durchlaufen (siehe Abbildung 2.1 auf der nächsten Seite). Es begann wieder eine Seminarphase, um die notwendigen Kenntnisse zu erwerben. In der

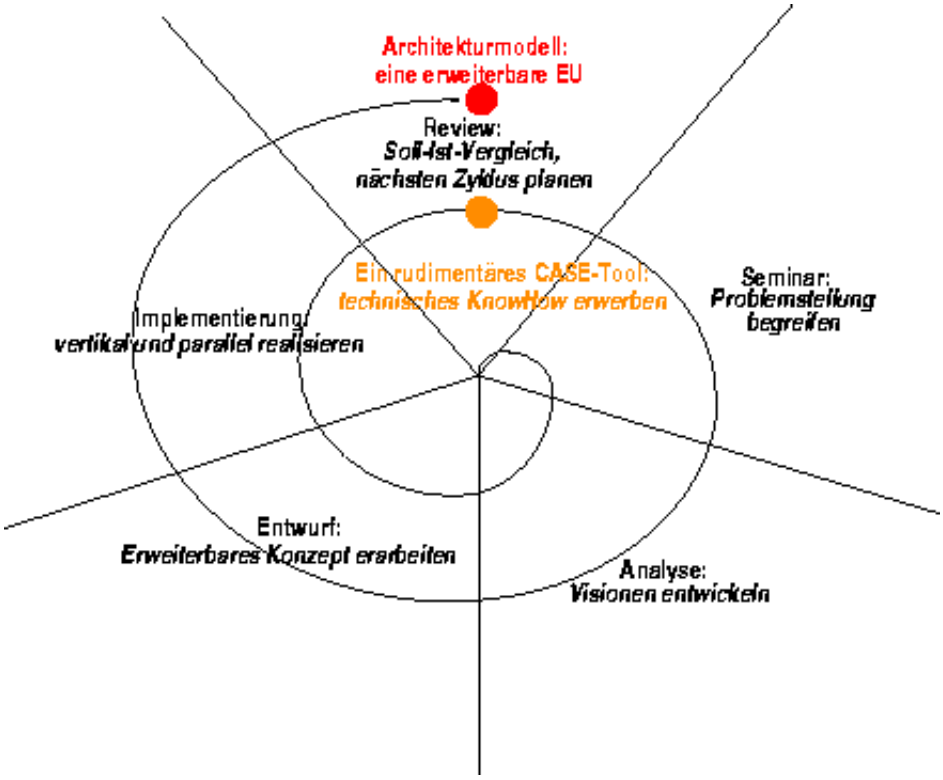


Abbildung 2.1.: Spiralmodell

anschließenden Analyse- und Entwurfsphase wurde ein erweiterbares Konzept erarbeitet und es entstand ein Entwurf der $\mathcal{HEU}$. Die nachfolgende vertikale Implementierung realisierte das erarbeitete Konzept und es entstand eine Entwicklungsumgebung für *DoDL*.

Teil II.

Seminar in Nordhelle (WS 1998/99)

3. Einführung in objektorientierte Konzepte

Autor: *Martin Uebing*

Im ersten Abschnitt dieses Textes wird in die Terminologie der Objektorientierung eingeführt und die Grundlagen von Objekten, Klassen, Vererbung usw. erläutert. Der zweite Teil behandelt die objektorientierte Softwareentwicklung und stellt einige wesentliche Konzepte wie Klassenbibliotheken und Frameworks vor.

3.1. Motivation

Objektorientierte Konzepte sind keine evolutionäre Weiterentwicklung früherer Analyse-, Entwurfs-, Design- und Programmiermodelle, sondern eine grundlegend andere Herangehensweise an die Problemlösung am Rechner.

Die Vorteile der Objektorientierung machen sich inbesondere bei der Entwicklung großer und komplexer Softwareprojekte bemerkbar bzw. machen solche Projekte überhaupt erst beherrschbar. Objektorientierung ist eine Modellbildung der Realität. Statt Anweisungsfolgen stehen die Daten und die zu ihrer speziellen Manipulation nötigen Operationen im Mittelpunkt. Objektorientierung ist dabei kein ausschließliches Implementierungskonzept, sondern bietet die Möglichkeit eines durchgängigen Prinzips von der Analyse über die Implementation bis zur Wartung und erleichtert somit die Wiederverwendbarkeit, Erweiterbarkeit und inkrementelle Entwicklung (Software-Evolution, siehe [Weg90]).

3.2. Grundlagen

3.2.1. Objekte

Objekte sind idealerweise Modelle realer Gegenstände, also in Software umgesetzte physische oder logische Einheiten mit einem inneren Zustand (*state*) und einem nach außen sichtbaren Verhalten (*behaviour*). Die Kommunikation von Objekten untereinander erfolgt über das Versenden von Nachrichten. Eine Zustandsänderung ist nur über fest definierte Schnittstellen möglich und tritt als Reaktion auf Nachrichten bzw. Ereignisse auf.

Die nicht mögliche direkte Manipulation von Daten folgt dem *Geheimnisprinzip* und ist eine wesentliche Vorraussetzung zur Erstellung korrekter, erweiterbarer und sicherer Programme.

3.2.2. Klassen

Durch Abstraktion, d.h. Erkennen grundlegender Gemeinsamkeiten verschiedener Objekte, entstehen abstrakte Objekt-Beschreibungen, die die Objekte zu Instanzen von *Klassen* zusammenfassen.

Klassen sind die Implementierung eines Objektes. Sie legen die Objektstruktur mit internen Daten, Attributen und Methoden fest. *Methoden* oder *Operationen* sind der einem Objekt zugeordnete Programmcode. *Attribute* oder *Instanzenvariablen* sind Daten, auf die von außen durch die Methoden zugegriffen werden darf.

Objekte sind dann quasi lebende Einheiten, die eigene Daten in Form von Werten beinhalten, auf die nur durch einige fest vorgegebene Methoden der Klasse zugegriffen werden kann. Dieser Satz öffentlich zugänglicher Methoden heißt *Objekt- oder Klassenschnittstelle*, ihre Festlegung - meist innerhalb der Klassendefinition - heißt *Schnittstellendefinition*.

Eine Klasse bildet somit eine Vorlage (*template*), nach der Objekte gebildet (*instanziiert*) werden. Die Erzeugungsoperation, meist eine besondere Methode, die *Konstruktor* genannt wird und u.a. auch der Initialisierung des Objekts dienen kann, erstellt dann eine Kopie der Klasse (*Instanz*) mit Kopien der Instanzenvariablen. Weiteres zu Konstruktoren und eine detaillierte, allerdings nicht allgemein übliche Einteilung der Methoden findet sich in [Rog98].

3.2.3. Kapselung

Die Trennung von Zustand und Verhalten des Objektes wird als *Kapselung* bezeichnet. Die Daten eines Objektes werden eingekapselt, sie sind nur durch die vorgesehenen Methoden von außen zu verändern, die damit vom internen Datenformat abstrahieren und es austauschbar machen.

3.2.4. Vererbung

Neben der Klassifizierung 1. Ordnung von Objekten durch Klassen ermöglichen objektorientierte Programmiersprachen (OOPs) auch eine Klassifikation 2. Ordnung von Klassen durch *Vererbung*. Dabei wird das Verhalten einer Klasse zur Definition neuer Klassen wiederverwendet. *Unterklassen* oder *Subklassen* erben dabei die Operationen der *Elternklassen* bzw. *Superklassen* und fügen neue Operationen und Attribute hinzu. Sie können je nach Sprache auch geerbte Methoden löschen oder überschreiben, also neu implementieren, und neue interne Daten hinzufügen.

Gegenüber der Klassifizierung von Werten durch Klassen ist die Vererbung eine stärkere Klassifizierung und verfügt über eine größere konzeptionelle Modellierungskraft. Während Klassen eine *Daten-Abstration* bilden, spricht man bei Vererbung von *Super-Abstraktion* (siehe [Weg90]).

Die Struktur eines Vererbungsbaumes kann je nach Intention Klassifizierung, Spezialisierung, Generalisierung, Approximierung oder Evolution ausdrücken. Diese Formen der Beziehungen von erbenden Klassen zueinander umfaßt häufig mehr als nur eine Ebene, so daß sich eine Verschachtelung von Unterklasse-Oberklasse-Beziehungen ergibt. Dabei sind zwei Kardinalitäten zu beachten. Zum einen koennen mehrere Subklassen von einer Superklasse erben, andererseits aber auch eine Subklasse von mehreren Superklassen. Während erstere Form, die eigentliche *Mehrfachvererbung* (häufig aber fälschlicherweise als Bezeichnung für Mehrfacherbung gebraucht) allgemein üblich ist und überhaupt erst zum Entstehen eines Vererbungsbaumes führt, ist die *Mehrfacherbung* in einigen objektorientierten Programmiersprachen nicht erlaubt, um die Probleme von zyklischer Vererbung und der Kollision gleicher Methoden- und Attributnamen von vornherein auszuschließen. Beispiele zur Behandlung dieser Probleme sind in [Weg90] aufgeführt.

3.2.5. Virtuelle Klassen

Als *virtuell* oder *abstrakt* bezeichnet man Klassen, deren Methoden zum Teil oder ausschließlich aus Schnittstellendefinitionen ohne Implementation bestehen. Die konkrete Implementation bleibt dann den erbenden Subklassen vorbehalten. Virtuelle Klassen bilden häufig die Wurzel bzw. die oberen Ebenen in Vererbungsbäumen. Sie können selbst nicht instanziiert werden.

3.2.6. Polymorphie

Polymorphie oder *Polymorphismus* heißt übersetzt Vielgestaltigkeit und steht für die Eigenschaft, daß der gleiche Methodenaufruf je nach Implementierung des Objekts ein ganz anderes Verhalten zur Folge haben kann. Polymorphie tritt insbesondere im Zusammenhang mit virtuellen Klassen und Methoden auf. Je nach Typ des aufrufenden Objekts wird eine andere Methode aus der entsprechenden implementierenden Subklasse aufgerufen.

Jedes Objekt einer Subklasse ist gleichzeitig Objekt der Superklasse. Überlädt nun die Subklasse eine Methode der Superklasse, so beantwortet das Binden die Frage, welche Komponente gemeint ist, wenn man das Objekt als Objekt der Superklasse benutzt. *Dynamisches* oder *spätes Binden* liefert weiterhin die Komponente der Subklasse, *statisches Binden* die Komponente der Superklasse. Bei Sprachen, die spätes Binden (*late binding*) unterstützen, wird der Typ des aufrufenden Objekts erst zur Laufzeit ermittelt und auch dann erst die entsprechende Methode ausgewählt.

3.2.7. Objektorientierte Systeme

Objekte gehen mit den sie benutzenden Klienten eine *Client/Server-Beziehung* ein. Der Vertrag zwischen einem Objekt und seinen Klienten spezifiziert dabei sowohl die Verantwortlichkeiten des Klienten als auch die des Objekts. Erstere werden durch Vorbedingungen (*preconditions*) ausgedrückt, letztere durch Nachbedingungen (*postconditions*).

Objekte haben globale Software-Management-Verantwortlichkeiten um flexible Objekt-Komposition und System-Evolution zu unterstützen. Dieses Objekt-Management wird

realisiert durch Klassen, die Objekte als Werte erster Ordnung behandeln, und durch Vererbung, die die Wiederverwendung von Schnittstellen-Spezifikationen durch inkrementelle Modifizierung und Erweiterung erleichtert.

Das objektorientierte Paradigma unterstützt drei Arten der Abstraktion: *Daten-Abstraktion* zur Objekt-Kommunikation, *Super-Abstraktion* durch Vererbung zur Objekt-Verwaltung und Verhaltenserweiterung und *Meta-Abstraktion* als eine Basis zur Selbstbeschreibung. Details hierzu findet man in [Weg90].

3.3. Objektorientierte Softwareentwicklung

3.3.1. Software-Komponenten

Während in prozedurorientierten Sprachen Programme aus einer langen Folge von Aktionssequenzen bestehen, sind objektorientierte Programme eine Sammlung untereinander abhängiger *Komponenten*, die jeweils durch ihre Schnittstelle spezifizierte Dienste anbieten. Die objektorientierte Programmstruktur modelliert somit direkt die Interaktion zwischen Objekten eines Anwendungsbereiches.

Ein großes Projekt in Komponenten aufzuteilen ist ein zeitsparendes Verfahren um die Komplexität der Aufgabe zu beherrschen, das auch mit *"devide and conquer"* (Aufteilung in Teilprobleme, die gemeinsam das vollständige Problem lösen) und *"separation of concerns"* (Trennung der Zuständigkeiten) bezeichnet wird. Software-Komponenten umfassen dabei Funktionen, Prozeduren und Objekte sowie nebenläufig ausführbare Komponenten (Prozesse, Akteure, Agenten). Objektorientierte Programmierung führte systematische Techniken zum Management von Software-Komponenten ein: Objekte bieten eine hohe Modularität zur direkten Anwendungsmodellierung, Klassen verwalten Objekte als Werte erster Ordnung und Vererbung organisiert Klassen in Hierarchien. Objekte haben einen höheren Abstraktionsgrad als Prozeduren und Funktionen: Die klar definierte Schnittstelle zu ihren Klienten abstrahiert von der internen Struktur und macht diese austauschbar, die Kommunikation erfolgt über Aufruf- und Rückgabe-Nachrichten, nebenläufige Objekte verfügen dabei über eine Vielzahl an Kommunikations- und Synchronisationsprotokollen.

3.3.2. Klassenbibliotheken

Bibliotheken sind Behälter für Software-Komponenten, die als wiederverwendbare Konstruktionseinheiten in der Software-Entwicklung dienen. Innerhalb einer Klassenbibliothek besteht meist eine fest definierte Klassenhierarchie, die durch Vererbung entsteht. Eine ausgezeichnete Klasse bildet hierbei die Basisklasse (*Wurzelklasse*), von der alle anderen Klassen direkt oder indirekt abgeleitet sind. Ziel bei der Verwendung von Klassenbibliotheken ist eine Programmkonstruktion fast ausschließlich aus vordefinierten Komponenten bei minimaler Verwendung von verbindendem Zwischencode (im englischen bezeichnenderweise *glue* - Klebstoff genannt). Dies verlangt im allgemeinen umgebungsspezifische Komponenten, die auf eine bestimmte Art von Projekt oder Anwendung zugeschnitten sind und führt nicht zu total universellen Bibliotheken. Das Ideal besteht

darin, für jede Anwendung diejenigen Verbindungsstellen zu finden, um ein Produkt fast ausschließlich aus Komponenten konstruieren zu können.

Bibliotheken in prozeduralen Sprachen haben Aktionen (Prozeduren) als Komponenten. Die nahtlose Kombination von Prozeduren mit Befehlen und Schlüsselwörtern der Programmiersprache blendet den verbindenden Code transparent ein und reduziert den Anreiz zur ausschließlichen Komposition von prozeduralen Komponenten.

Komponenten objektorientierter Bibliotheken sind Klassen, aus denen Objekte erzeugt werden. Dabei wird zwischen zwei Arten der Komposition unterschieden: *Deklarative Komposition* von Klassen (z.B. durch Vererbung) und *Laufzeit-Komposition* von Objekten.

Flachen Behältern unabhängiger und unverbundener Komponenten bei prozedurorientierten Bibliotheken stehen hierarchisch organisierte Klassenkollektionen auf objektorientierter Seite gegenüber. Diese können Spezialisierung, Abstraktion, Approximation und Evolution ausdrücken, umfassen Metaobjekte zur Spezifizierung von Operationen auf Klassen und unterstützen virtuelle Klassen mit unvollständigem Verhalten, deren undefinierte Attribute und Methoden in Subklassen geliefert werden müssen, bevor Objekte instanziiert werden können (siehe [Weg90]).

3.3.3. Frameworks

Frameworks erleichtern bzw. ermöglichen erst die Kombination verschiedener Komponenten zu vollständigen Anwendungen. Sie liefern die für die Komponenten nötige Semantik auf der Applikationsebene und legen somit die Regeln zur Zusammenarbeit der Komponenten auf Anwendungsniveau fest. Sie ermöglichen die dynamische Erzeugung von Komponentennetzen, ohne daß deren Kombination von den Komponentenentwicklern ursprünglich vorhergesehen sein muß. Die Komplexität des Frameworks entsteht durch die Organisation, sie steckt nicht in den Komponenten selbst.

Ein Framework ist eine Anzahl von vorgefertigten Softwarebausteinen, die Entwickler für ein spezifisches Softwareprojekt verwenden, erweitern oder anpassen können, also eine Art Softwaresubsystem für eigene Applikationen. Objektorientierte Frameworks sind im Prinzip eine erweiterte, "umgedrehte" Klassenbibliothek. Im Vergleich bietet ein Framework zusätzlich ein hohes Maß an Funktionalität und bereits bestehende Verbindungen zwischen den Objektklassen, die eine Infrastruktur für den Anwendungsentwickler liefern.

Zwar bietet auch ein Framework wie eine Klassenbibliothek Methoden, die aufgerufen werden können, der wesentliche Unterschied besteht aber darin, daß der eigentliche Kontrollfluß (*"thread of control"*) im Framework abläuft. Die speziellen, vom Entwickler geschriebenen Methoden der Anwendung werden also aus einer im Framework integrierten allgemeinen Programmroutine aufgerufen, im Gegensatz zur Klassenbibliothek, bei der eigener Code die Bibliothekskomponenten aufruft.

Als Design für eine Menge von Objekten, die gemeinsam eine Menge von Verantwortlichkeiten aufstellen, sind Frameworks somit ein Gegenstück zu abstrakten Klassen als Design für ein einzelnes Objekt. Beispiele für Frameworks und die genaue Art ihrer Anwendung ist ausführlich in [OHE95] beschrieben.

3.4. Zusammenfassung und Ausblick

Da wir mit Java in der Projektgruppe eine objektorientierte Programmiersprache verwenden, die die hier genannten Konzepte unterstützt, ist es wichtig diese verstanden zu haben und zu unserem Vorteil zu nutzen. Beim genaueren Kennenlernen von Java und seinen Klassenbibliotheken - wie z.B. Swing, aber u.a. auch in der Datenbank, gilt es, objektorientierte Konzepte zu erkennen und umzusetzen. Die Entwicklung des Prototypen Ampelstadt bietet an mehreren Stellen Möglichkeiten zur Wiederverwendung und Erweiterung von Komponenten für das eigentliche Projekt HEU. Die Absprache von sauber trennenden Schnittstellen und Verteilung von Zuständigkeiten zur Implementation der einzelnen Komponenten ist bei der Vielzahl der Entwickler ein wesentlicher Faktor für den Erfolg des Projekts.

4. Einführung in Java

Autor: *Jens Schröder*

Diese Arbeit soll einen Überblick über die Konzepte der Sprache Java vermitteln. Ich werde nur wenig auf die Syntax der Sprache eingehen, sondern vielmehr die grundlegenden Ideen von Java vorstellen.

Zunächst gebe ich einen historischen Überblick über Java's Entwicklungsgeschichte, dann werde ich die Systemarchtiektur der Sprache erläutern und allgemeine Merkmale vorstellen.

Im folgenden Abschnitt stelle ich ausgewählte Sprachkonstrukte vor. Nachdem ich zuerst auf die Packetstruktur eingehe, werde ich die Unterschiede zwischen Klassen und Schnittstellen erläutern. Wie Exceptions und Threads in Java realisiert sind, bilden das Ende dieses Abschnitts.

Zum Schluß gebe ich einen groben Überblick über die Bibliotheken von Java. Ich werde erst auf die Java-API, dann auf Swing, eine Bibliothek zur Erstellung von Benutzerschnittstellen, und zum Abschluß auf eine Auswahl von zusätzlichen API's eingehen.

4.1. Historischer Überblick

Nach [Ban95] und [Sun98b] beginnt Java's Geschichte 1990 mit dem Wunsch von Sun, endlich wieder effektiv ein innovatives Produkt zu entwickeln. Zu diesem Zweck wird mit einer Handvoll Entwicklern ein Projekt zur Erstellung einer Entwicklungsumgebung für eine gemeinsame Fernsteuerung von verschiedenen Alltagsgegenständen (Codename *Green*) ins Leben gerufen.

Um den speziellen Anforderungen einer Fernsteuerung verschiedener Geräte (Beschränkung der Bandbreite und des Speichers, Plattformunabhängigkeit) gerecht zu werden, entwickelt James Gosling 1992 eine neue Programmiersprache, *Oak* genannt. Sie kann als der Vorläufer von Java betrachtet werden.

Da das Projekt *Green* (später FirstPerson Inc.) auch nach mehreren Richtungskorrekturen kein marktfähiges Produkt hervorbringt und das Internet explosionsartig expandiert, beschließt 1994 Sun, daß *Oak* an die Begebenheiten des Internets angepaßt werden soll. Die fertige Sprache wird Java genannt und ist bekanntlich auf dem Markt ein großer Erfolg.

4.2. Systemarchitektur

Die Systemarchitektur besteht hauptsächlich aus zwei Komponenten, dem Javacompiler (*javac*) und der virtuellen Maschine (*java*).

Der Compiler geniert aus dem Quellcode Javabytecode (die Klassen `.class`). Die virtuelle Maschine (VM) interpretiert den Bytecode bzw. benutz einen Just-In-Time-Compiler zum Erzeugen nativen Maschinencodes (je nach Implementierung der VM).

Werden weitere Klassen zum Ausführen eines Programms benötig, lädt der Klassenlader sie nach einem Typentest und einer Verifizierung des Bytecodes dynamisch in die VM. Das Kernstück der VM besteht neben dem Interpreter (bzw. dem JIT-Compiler) aus der Laufzeitumgebung. In ihr laufen die Java-Applets und -Applikationen.

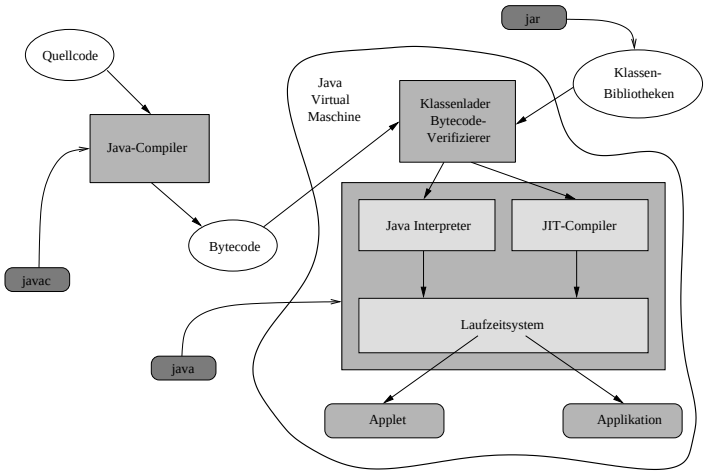


Abbildung 4.1.: Systemarchitektur

Da Java viel im Bereich des Internets eingesetzt wird, müssen hohe Anforderungen an die Sicherheit gestellt werden. Gerade Applets haben deshalb einen strengen Sicherheitsmodus, der ihnen vom Webbrowser zugewiesen wird.

Sie laufen in einer abgesicherten Umgebung (Sandkasten), die ihnen den Zugriff auf das Dateisystem verweigert, es sei denn, man gewährt ihnen explizit das Recht dazu. Vor dem Ausführen von sensitiven Operationen (etwa das Schreiben im Dateisystem) wird immer die `SecurityManager`-Klasse um Erlaubnis gefragt, die dann gegebenenfalls Exception wirft. Durch digitale Signaturen kann man Code als vertrauenswürdig kennzeichnen (*javakey*) und ihm somit mehr Rechte zuweisen.

4.3. Allgemeine Merkmale

Das herausragenste Merkmal von Java ist seine Plattformunabhängigkeit. Der Bytecode, den der Compiler generiert, ist plattformunabhängig. Mittlerweile gibts es auch ein Zertifikat für diese Plattformunabhängigkeit ("100 % pure Java"). Nur die VM muß auf allen Plattformen portiert werden, der Bytecode nicht. Gerade diese Eigenschaft macht

Java auch so geeignet für den Gebrauch im Internet.

Java unterstützt verteilte Anwendungen direkt, da mit `java.net` und `RMI` (Remote Method Invocation) zwei Bibliotheken zur Verfügung stehen, die speziell für das Erstellen von verteilte Anwendungen entwickelt wurden. Auch gibt es mittlerweile CORBA-Implementierungen (siehe Kapitel 7 auf Seite 47), die mit Java arbeiten.

Weiterhin ist im Gegensatz zu C/C++ ist das explizite Freigeben von Speicherplatz bei Java nicht mehr nötig. Der Speicherplatz von Objekten, auf die keine anderen Objekte mehr referenzieren, wird automatisch wieder freigegeben. Dieser Mechanismus wird *Garbage Collection* genannt.

Bei Klassen ist in Java nur Einfacherbung möglich. Eine Klasse darf nur eine Superklasse haben. Um aber dennoch polymorphe Strukturen zu ermöglichen, darf eine Klasse beliebig viele Schnittstellen (`interface`) implementieren. Bei Schnittstellen ist Mehrfacherbung jedoch erlaubt. Eine Schnittstelle darf von mehreren anderen Schnittstellen erben.

Java ist eine streng typisierte Sprache. Während des Compilierens werden schon fehlerhafte Datentypnutzengen erkannt. Der Compiler erkennt auch Fehler beim Aufruf von Methoden, da diese (im Gegensatz zu C) immer explizit deklariert werden müssen.[AHM98]

4.4. Sprachkonstrukte

Nach dem ich einen groben Überblick über Java gegeben habe, gehe ich jetzt auf die wichtigsten Sprachkonstrukte ein.

4.4.1. Packet-Struktur

Die Verwaltung von Klassen und Schnittstellen erfolgt mit Hilfe von Paketen (`package`). Die Pakete sind global eindeutig, d.h. auch mit Standardbibliotheken treten keine Doppeldeutigkeiten auf. Auf Klassen wird über den vollständig qualifizierten Namen zugegriffen, damit die Klassen für den Klassenlader eindeutig bestimmbar sind.

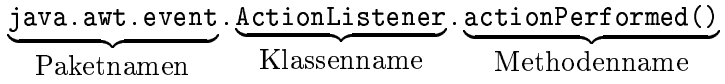


Abbildung 4.2.: vollständig qualifizierter Name

Mit Hilfe der `import`-Anweisung ist auch ein Zugriff nur über den Klassennamen erlaubt, der Paketname kann dann weggelassen werden. Das Basispaket `java.lang` ist standardmäßig importiert.

In der Packetstruktur gibt es zwei mögliche Sichtbarkeitsbereichen (Scopes) von Klassen und Schnittstellen. Eine mit dem Modifikator `public` definierte Klasse bwz. Schnittstelle ist überall sichtbar. Wird kein Modifikator gesetzt, ist die Sichtbarkeit auf das eigene Paket beschränkt (`*package*`, kann nicht explizit angegeben werden).

4.4.2. Klassen

Klassen (Schlüsselwort `class`) kann man als eine Sammlung von Daten (Attributen) und Methoden (auf diesen Daten) ansehen. Ein Objekt ist eine instanziierte Klasse. Die Instanziierung erfolgt durch einen Konstruktoraufruf (Schlüsselwort `new`).

Will man in einem Programm eine Methode oder Variable verwenden, ohne vorher die Klasse zu instanziieren, muß in Java das Schlüsselwort `static` verwendet werden. Solche Methoden und Variablen, auf die unabhängig von der Existenz eines Objekt zugegriffen werden kann, nennt man Klassenmethoden und Klassenvariablen. Von Klassenvariablen existieren nur eine Kopie, sie sind globalen Variablen ähnlich. Analog sind Klassenmethoden globalen Methoden ähnlich, sie werden nur über den Klassennamen und nicht über den Instanznamen angesprochen.

In Java darf eine Klasse nur von einer anderen Klasse erben (Schlüsselwort `extends`), aber beliebig viele Schnittstellen implementieren (Schlüsselwort `implements`). So kann man die Probleme, die bei Mehrfacherbung auftreten, umgehen, ohne daß die Vorteile der Mehrfacherbung verlorengehen.

Die Sichtbarkeit von Methoden einer Klasse kann mit folgenden Modifikatoren festgelegt werden:

- `public`: Die Methode ist überall sichtbar.
- `protected`: Die Methode ist im eigenen Paket und in Subklassen sichtbar.
- `*package*`: Die Methode ist nur im eigenen Paket sichtbar (kann nicht explizit gesetzt werden; wird gesetzt, wenn kein Modifikator angegeben wird).
- `private`: Die Methode ist nur innerhalb der eigenen Klasse sichtbar, auch nicht in Subklassen.

Definiert man in einer Klasse nur die Signatur einer Methode, aber nicht deren Implementierung, so nennt man sie abstrakt. Ist in einer Klasse eine Methode abstrakt, so muß die Klasse auch als abstrakt definiert werden (Schlüsselwort `abstract`). Eine abstrakte Klasse kann nicht instanziiert werden, sie darf aber Attribute, Konstruktoren und nichtabstrakte Methoden besitzen.

4.4.3. Schnittstellen

Schnittstellen (Schlüsselwort `interface`) dürfen im Gegensatz zu abstrakten Klassen ausschließlich abstrakte Methoden (also nur Signaturen) und Konstanten enthalten. Die Datenfelder dürfen also nicht änderbar sein (Schlüsselwort `final`). Alle Methoden und Konstanten müssen, anders als bei abstrakten Klassen, immer `public` deklariert werden.

Da Schnittstellen nur die Signatur festlegen, erlauben sie abstrakte Datentypen zu definieren. Man hat die Möglichkeit, heterogene Schnittstellen zu einer zusammenführen, da Schnittstellen mehrfacherben dürfen.

Schnittstellen werden durch Klassen implementiert, wobei eine Klasse mehrere Schnittstellen implementieren kann. Im Gegensatz zu abstrakten Klassen müssen immer alle Methoden implementiert werden, wenn eine Klasse eine Schnittstelle implementiert.

Es kann vorkommen, daß man leere Schnittstellen definiert, um Strukturen aufbauen zu können. Solche Schnittstellen werden "Marker-Interfaces" genannt. Ein Beispiel dafür ist die `serializable`-Schnittstelle.

4.4.4. Exceptions

Exceptions zeigen Ausnahmesituationen (Fehler) an. Sie sind in Java Objekte (mit der Superklasse `java.lang.Exception`), die am Ort des Auftretens instanziiert (`throw new Exception()`) werden. Sie können entweder vom Programmierer oder vom Laufzeitsystem erzeugt werden. Fügt man einer Methodensignatur das Schlüsselwort `throws` hinzu, wird man bei dem Aufruf dieser Methode automatisch zu einer Ausnahmebehandlung gezwungen.

Exception werden durch eine `try/catch/finally`-Anweisung behandelt. Im `try`-Block erfolgt der Aufruf der kritischen Methode. Im `catch`-Block wird die Exception aufgefangen und behandelt. Es sind mehrere `catch`-Blöcke möglich. Zum Schluß kommt der optionale `finally`-Block, dessen Inhalt garantiert ausgeführt wird, egal ob eine Exception ausgelöst wurde oder nicht.

Dieser Aufbau erlaubt eine strukturierte Behandlung von Exceptions. Durch den strengen Formalismus ist man als Programmierer gezwungen, Ausnahmesituationen abzufangen.

4.4.5. Threads

Ein Thread ist ein leichtgewichtiger Prozeß. Java unterstützt Nebenläufigkeit, in dem sie die Klasse `java.lang.Thread` anbietet und multithreading-fähig ist. Es können also mehrere Threads gleichzeitig ablaufen, was besonders häufig z.B. bei Benutzerschnittstellen von Vorteil ist.

Bei mehreren parallelen Prozessen liegt es am Programmierer, daß er die Konsitenz wahrt. Zu diesem Zweck bietet Java ihm den Mechanismus der Synchronisation an, der zur Wahrung der Konsistenz bei nebenläufigen Programmen eingesetzt werden kann.

Synchronisation verhindert den gleichzeitigen Zugriff verschiedener Threads auf die selbe Resource. Objekte haben eine Sperre ("lock"), die der gerade aktive Thread bekommt. Wenn eine Sperre vergeben ist, kann solange kein anderer Thread auf das Objekt zugriffen, bis die Sperre wieder freigegeben wird.

In Java kann man nicht nur Objekte, sondern auch Methoden synchronisieren. Ein Thread kann nur dann eine sysnchronisierte Methode ausführen, wenn er auch die Sperre des Objekts besitzt, zu dem die Methode gehört.

4.5. Bibliotheken

Ich werde jetzt ausgewählte Bibliotheken vorstellen, die Java bereitstellt.

4.5.1. Java-API

Die Java-API (Application Programming Interface) ist die Standard-Klassenbibliothek von Java. Sie unterteilt sich in verschieden Pakete, von denen ich nur die wichtigsten näher erläutere. Für weitere Informationen sei auf [Fla98] und die mit dem JDK mitgelieferte Dokumentation verwiesen.

- `java.lang`
  Das Basispaket der Java-Api. In ihm wird z.B. die Klasse `Object` definiert, von der jedes andere Java-Objekt erbt.

  Eine weiter bemerkenswerte Klasse ist die Klasse `Class`. Sie repräsentiert eine Java-Klasse zur Laufzeit (`.class`-Datei). Die Klasse `Class` hat eine statische Methode, die das dynamische Nachladen einer Klasse in den Interpreter ermöglicht.

- `java.io`
  Das Paket für I/O-Anwendungen. Es werden eine Vielzahl von Strömen und sonstigen Objekten für I/O-Anwendungen (z.B. ein systemunabhängiges `File`-Objekt) bereitgestellt. Die mächtigsten Stromobjekte sind `ObjectInputStream` und `ObjectOutputStream`, die das Schreiben von Objekten an und das Lesen von Objekten aus einem Strom ermöglichen.

  Diese wird durch Serialisierung ermöglicht, d.h. Java hat die Fähigkeit, komplette Zustände von Objekten in einen Ausgabestrom zu schreiben (zu serialisieren) und durch Lesen der serialisierten Objekte an einen Eingabestrom wieder herzustellen. Die Serialisierung ist rekursiv, Zyklen werden erkannt.

- `java.awt`
  Das `java.awt`-Paket (Abstract Window Toolkit) ist für die Erstellung einer graphische Benutzerschnittstelle gedacht. Die einzelnen Komponenten kommunizieren nach dem Eventmodell.

  Das Eventmodell in Java gliedert sich in drei Komponenten. Man hat Quellobjekte, auf denen ein Event ausgelöst wird (z.B. auf eine Komponente der Benutzungsschnittstelle), ein Eventobjekt, das die Art und die Quelle des Events beinhaltet, und den Eventlistener, der Eventobjekte von den Quellobjekten per Methodenaufruf übergeben bekommt.

  Jedes Quellobjekt holt sich ein entsprechenden Listener. Tritt ein Event auf, schickt es durch einen Methodenaufruf ein Eventobjekt an den Listener. Der Listener bestimmt aus dem Eventobjekt die Art des Events und das Quellobjekt und reagiert dann entsprechend auf den Event. Dadurch braucht der Listener nicht direkt seine Quellobjekt kennen, sie sind durch die Eventobjekte gekapselt.

- `java.util`
  Paket, das Standarddatentypen wie etwa Vector, Hashtable und Stack, beinhaltet. Andere Pakete nutzen diese Datentypen.

- `java.beans`
  Paket zur Erstellung einer wiederverwendbaren, einbettbaren, modularen Softwartekomponente. Beans wurden in der JavaBeans-Spezifikation wie folgt beschrieben: "Eine wiederverwendbare Softwarekomponente, die visuell mit einem Generierungswerkzeug arbeiten kann."

- `java.net`, `java.text`, `java.applet` und `java.math`
  Pakete, die die Erstellung von Software mit speziellen Aufgaben (entsprechend den Namen der Pakete) unterstützen.

4.5.2. Swing

Swing [Sun98a], [Mey98] ist eine Klassenbibliothek zur Erstellung von graphischen Benutzerschnittstellen. Sie setzt auf dem AWT auf und erweitert es. Mit dem JDK 1.2 wird sie standardmäßig mitgeliefert und wird zukünftig das Aussehen der Java-Applikationen bestimmen.

4.5.3. Zusätzliche API's

Es existieren noch eine Vielzahl von weiteren API's, die zur Erstellung spezieller Software nützlich sind. Die Bereiche umfassen Verteilung (Unterstützung für Methodenaufrufe bei entferneten Objekten), Datenbankschnittstellen, plattformunabhängige Kommunikationsanwendungen und vieles mehr. An dieser Stelle sei auf [Sun] verwiesen.

5. Unified Modeling Language

Autor: *Adil Kassabi*

Dieses Kapitel soll dem Leser grundlegende UML-Kenntnisse vermitteln. Anhand eines konkreten Beispiels werde ich auf die wichtigsten Sprachkonstrukte eingehen, diese näher beschreiben und die Einsatzbereiche einiger UML-Diagramme erläutern. Diese Arbeit soll als kurze Einführung in die UML dienen, der Leser, der diese Sprache näher kennenlernen will, sei auf [SF98] verwiesen.

5.1. Einführung

In den späten 80er und frühen 90er Jahren kam eine Welle von objektorientierten Analyse- und Entwurfsmethoden auf. Die Unified Modeling Language (UML) ist der Nachfolger einer Reihe dieser Methoden. Sie vereinheitlicht und erweitert die Methoden von Booch, Rumbaugh, und Jacobson. Es ist keine Methode, sondern eine Modellierungssprache mit graphischer Notation, die von Methoden dazu verwendet wird, Entwürfe auszudrücken.

5.2. Motivation

Die Idee, die Booch, Rumbaugh und Jacobson bei der Entwicklung von UML realisieren wollten, war eine Standardmodellierungssprache in die Softwareentwicklung einzuführen. Als Modellierungssprache hat UML inzwischen einen breiten Einsatz gefunden und dient damit als Kommunikationsbasis zwischen den Beteiligten an der Software-Entwicklung. UML ist nach einer Reihe von oo-Anlyse und -Entwurfsmethoden entstanden und ist damit eine Modellierungssprache der zweiten Generation, in der eine Menge von Schwächen, die man bereits in den vorherigen Modellierungssprachen kennengelernt hat, beseitigt wurden.

5.3. Entwicklung eines Softwaresystems

Das Endziel der Softwareentwicklung ist es, ein lauffähiges Programm zu erhalten. Dieses Ziel wird über verschiedenen Phasen wie Analyse, Entwurf, Implementierung u.a. erreicht. Einige Endergebnisse dieser Phasen sind Dokumente, die mit UML modelliert werden können. Wie UML in einigen Phasen der Entwicklung des Systems zur Erstellung von Dokumenten eingesetzt werden kann, wird in den nächsten Teilen des Artikels anhand

des folgenden Beispiels angegeben. Dabei beschränken wir uns auf die Analyse- und Entwurfsphase.

5.3.1. Beispielproblem

Als illustrierendes Beispiel wollen wir ein Dozenten/Studenten-Verwaltungssystem entwickeln, an das folgende Anforderungen bestehen:

- Dozenten eines bestimmten Fachbereichs auflisten
- Studenten eines bestimmten Fachbereichs auflisten
- Veranstaltungsplan eines Semesters erstellen
- Neue Dozenten einstellen
- Neue Studenten anmelden
- Studenten exmatrikulieren
- Prüfungen verwalten

5.3.2. Analysephase

Das Ergebnisdokument der Analysephase besteht aus einem Klassendiagramm, das eine statische Modellierung des Anwendungunsbereiches darstellt, und einem Use-Case-Diagramm, das zwar nicht formell ist, aber die funktionale Anforderungen an das zu entwicklende System als Anwednungsfall wiedergibt.

Warum verwendet man Use Cases in der Analysephase? Use Cases werden in der Analysephase gebildet, um eine grobe Übersicht über die Aufgaben des Systems zu ermitteln. Mit ihrer Hilfe kann auch die inkrementelle Softwareentwicklung angewandt werden, in dem man die von einander getrennten Komponenten, die einen sinnvollen Aufgabenbereich darstellen, Stück für Stück erstellt.

Die Use-Case-Diagramme dienen später in der Testphase als Testbasis. Dabei wird geprüft ob alle Aufgaben, die im Use-Case-Diagramm modelliert sind, vom Endprodukt realisiert werden können. Use Cases zusammen mit einem erläuternden Text erfassen die funktionalen Anforderungen an das System.

Warum verwendet man Klassendiagramme in der Analysephase? Im Klassendiagramm der Analysephase spiegeln sich die Anforderungen an das System wieder, und der Auftraggeber und Auftragnehmer können sich anhand dessen verständigen.

In der Analysephase werden im Klassendiagramm hauptsächlich die Objekte festgelegt, mit denen der Endbenutzer arbeitet. Objekte, die dem Benutzer transparent bleiben, sind aber nicht ausgeschlossen.

5.3.2.1. Use Cases

Definition: Ein Use-Case, auch Anwendungsfall, ist eine typische Interaktion zwischen dem Benutzer und dem Computersystem. Zwei typische Anwendungsfälle für eine Textverarbeitungssoftware wären: »zeichne eine Textstelle fett« und »erstelle einen Index« [SF98].

Wie kann ein Anwendungsfall erstellt werden? Ein Anwendungsfall kann in seiner einfachsten Form so erfaßt werden, daß man die mit dem System in Verbindung kommende Dinge abgrenzt, jede gewünschte Funktion nennt, ihr einen Namen gibt und dazu einen kurzen erläuternden Text schreibt.

Zu vermeiden in Anwendungsfällen: Man darf nicht versuchen, alle Details gleich zu Beginn zu ermitteln, sonst wird das Use-Case-Diagramm unübersichtlich. Manchmal ist es sinnvoll, Anwendungsziele von Systeminteraktionen zu unterscheiden. In einem Textverarbeitungssystem könnte eine Systeminteraktion z.B. so aussehen wie »übertrage ein Format eines Dokument in ein anderes« und »definiere ein Format«. Die wirklichen Anwenderziele dagegen könnten mit Begriffen wie »stelle eine konsistente Formatierung für ein Dokument sicher« umrissen werden.

»Erweitert« und »Benutzt«-Beziehungen: Man benutzt die »Erweitert«-Beziehung bei Anwendungsfällen, die einem anderen Fall ähnlich sind, aber ein wenig mehr bewerkstelligen. Die »Benutzt«-Beziehung wird verwendet, wenn ein Verhaltensanteil in verschiedenen Anwendungsfällen gleich ist und man sich das Kopieren/Einfügen der Beschreibung des Verhaltens ersparen möchte.

Elemente eines Use Case-Diagramms: Ein Use Case-Diagramm besteht aus den Akteuren und den Use Cases.

Akteure: Ein Akteur ist eine Rolle, die vom Anwender des Systems eingenommen wird: Manager, Händler, Verkäufer ...

Abb 5.1 auf der nächsten Seite stellt das Use-Case-Diagramm vom Dozenten/Studenten-Verwaltungssystem dar, das die Aufgaben, die unser System leisten soll, erfaßt, und die Anwender, die mit dem System arbeiten, anzeigt.

5.3.2.2. Klassendiagramme

Definition: Ein Klassendiagramm beschreibt die Typen von Objekten im System und die verschiedenen Arten von statischen Beziehungen zwischen diesen, im besonderen Assoziationen und Untertypen. Klassendiagramme zeigen weiterhin die Attribute und Operationen einer Klasse sowie die Bedingungsregeln der Verbindung ihrer Objekte [SF98].

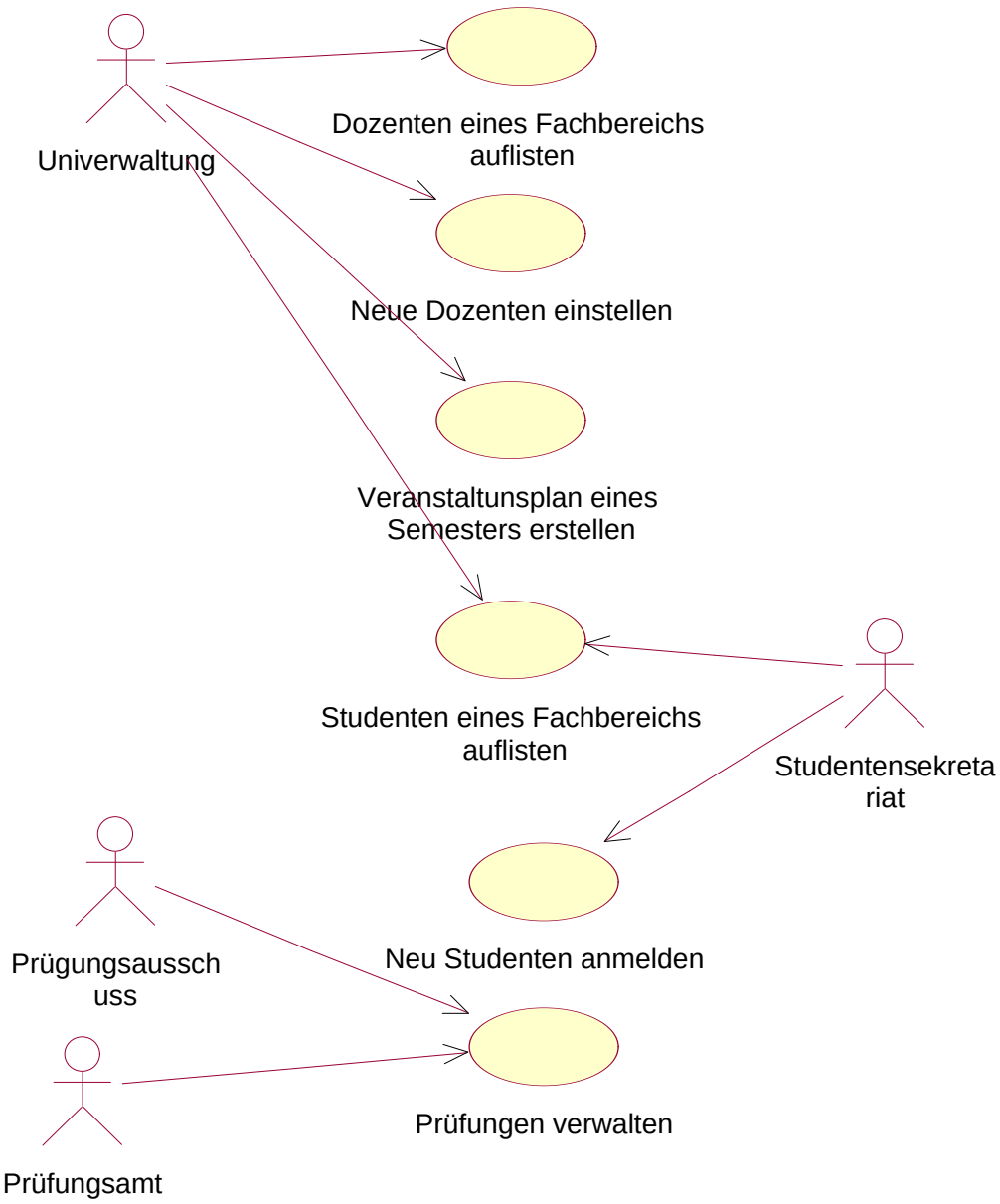


Abbildung 5.1.: Analysephase: Use-Case-Diagramm für das Dozenten/Studenten-Verwaltungssystem

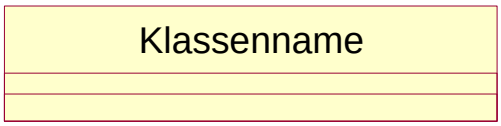


Abbildung 5.2.: UML Notation einer Klasse

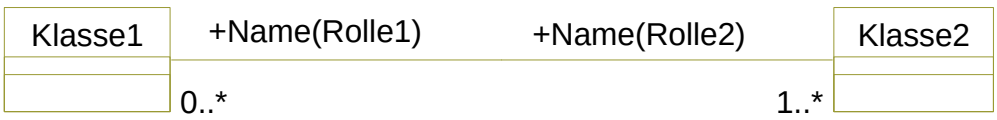


Abbildung 5.3.: UML Notation einer Assoziation

Auf den Abbildungen 5.2 auf der vorherigen Seite und 5.3 auf der vorherigen Seite sind UML Notationen von Klassen und Klassenbeziehungen zu sehen. Aus der Abbildung 5.3 auf der vorherigen Seite läßt sich z.B. ablesen, daß die Klasse **Klasse1** ein Attribut **Rolle2** hat, das ein oder mehrere Objekte vom Typ **Klasse2** enthält

Abbildung 5.4.: UML Notation einer Aggregation Klasse1 aggregiert Klasse2

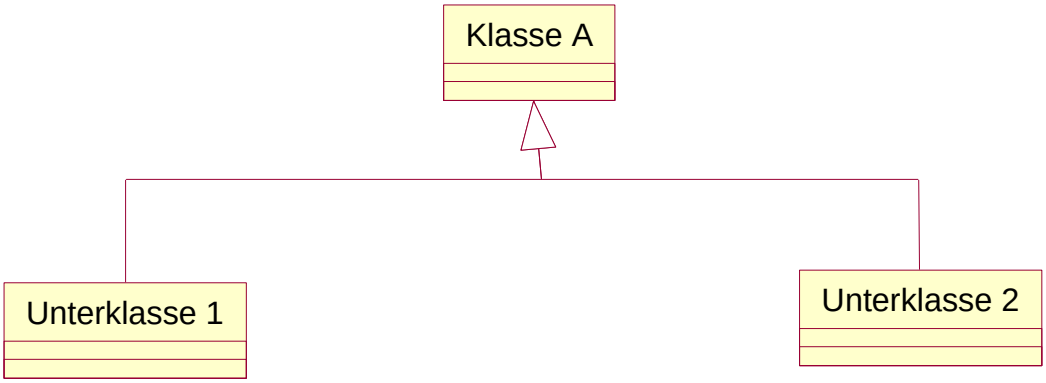


Abbildung 5.5.: UML Notation der Generalisierung

Bedingungsregeln: Die grundlegenden Konstrukte Assoziation, Attribut und Generalisierung können einge wichtige Bedingungen ausdrücken. Diese Bedingungen, die beim Entwurf aufgestellt werden, müssen natürlich bei der Implementierung berücksichtigt werden. Syntax: UML definiert keine strikte Syntax für die Beschreibung von Bedingungen. Sie müssen nur in geschweifte Klammern gesetzt werden. Ein Beispiel für die

Beschreibung von Bedingungen in UML ist in der Abbildung 5.6 gegeben.

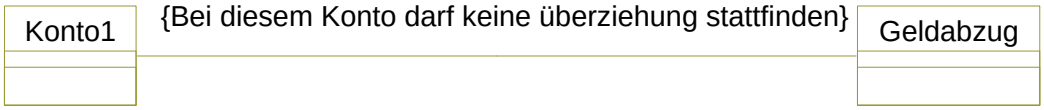


Abbildung 5.6.: Bedingungsregeln in der UML

Stereotypen: Man kann sich Stereotypen als Untertypen der Metamodelltypen Klasse, Assoziation und Generalisierung vorstellen. UML hat sie zu einem allgemeinen Erweiterungsmechanismus für die Sprache selbst gemacht. Stereotypen werden gewöhnlich als Text zwischen französichen Anführungszeichen »control object« dargestellt. Man kann selbst Stereotypen und die Regeln für die Benutzung definieren.

Abb 5.7 auf der nächsten Seite zeigt das Klassendiagramm für das Studenten/Dozenten-Verwaltungssystem. Das Diagramm enthält die wesentlichen Klassen und Beziehungen, die bereits in der Analysephase erkannt werden können.

5.3.3. Entwurfsphase

5.3.3.1. Sichtweisen auf ein Klassendiagramm

Eine wichtige Feinheit, die vor dem Entwurf des Klassendiagramms eines Systems zu berücksichtigen ist, ist die Sicht, mit der das System mittels des Klassendiagramms beschrieben wird. Es gibt drei Sichtweisen:

konzeptionell: Wenn man eine konzeptionelle Sichtweise einnimmt, erstellt man ein Diagramm, das die Konzepte im untersuchten Problembereich wiedergibt. Diese Konzepte haben natürlicherweise eine Beziehung zu den implementierenden Klassen, oft gibt es aber keine direkte Abbildung. Ein konzeptionelles Modell soll mit wenig oder ohne Berücksichtigung der implementierenden Software entworfen werden, so daß es als sprachunabhängig betrachtet werden kann.

spezifizierend: Diese Sichtweise betrachtet die Schnittstellen der Software. Die Implementationsgrundlage wird nicht festgelegt. Es werden möglichst wenige sprachabhängige Konstrukte verwendet.

implementierend: In dieser Sicht wird die Implementierunggrundlage festgelegt und das Modell wird mehr und mehr sprachabhängig. Das Verstehen der Sichtweise ist sowohl beim Zeichnen als auch beim Lesen wichtig.

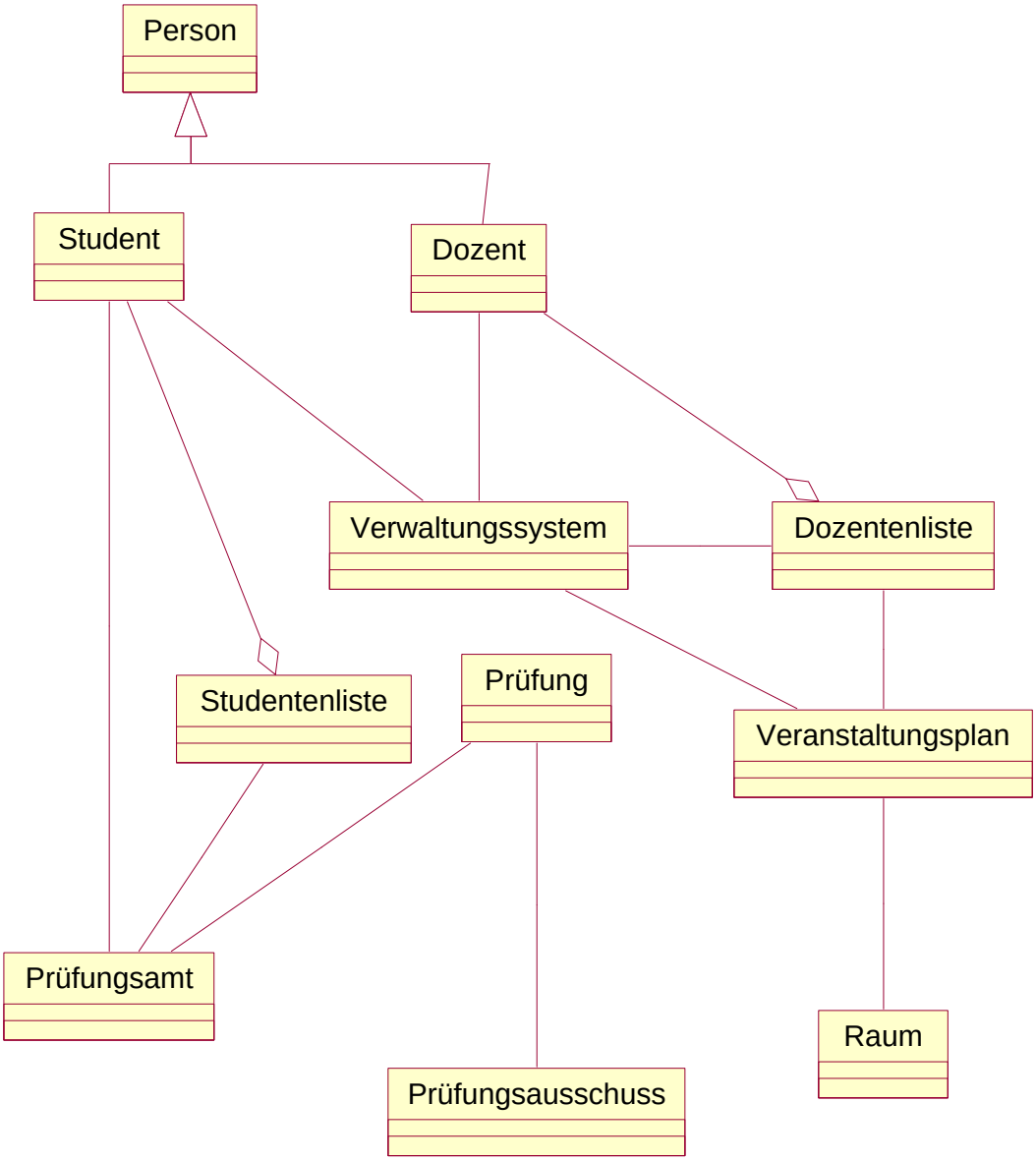


Abbildung 5.7.: Analysephase: Klassendiagramm für das Dozenten/Studenten-Verwaltungssystem

Ein Klassendiagramm für die Entwurfsphase: Abb 5.8 zeigt das Klassendiagramm für die Entwurfsphase. Im Normalfall wird auf den ersten Blick auf Grund der vielen vorgenommenen Änderungen nicht erkannt, daß das Klassendiagramm für die Entwurfsphase aus dem von der Analysephase entstanden ist. In der Entwurfsphase wird versucht, alle zu verwendende Klassen und Methoden vorzusehen und im Klassendiagramm mit einzubeziehen. Die Implementierungsphase ist der nächste Schritt im Softwareentwicklungsprozeß.

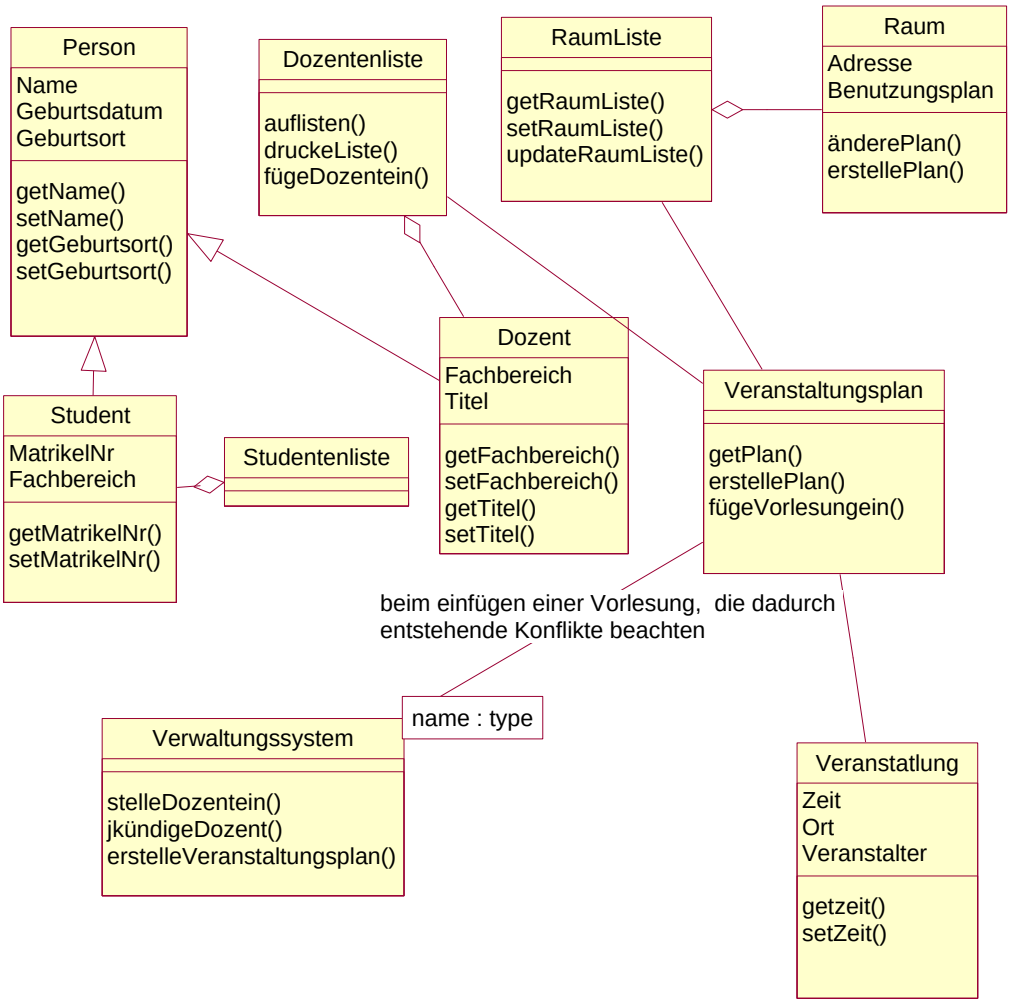


Abbildung 5.8.: Entwurfsphase: Klassendiagramm für das Dozenten/Studenten-Verwaltungssystem

5.3.3.2. Interaktionsdiagramme

Definition: Interaktionsdiagramme beschreiben, mit welchem Verhalten Gruppen von Objekten zusammenarbeiten. Ein Interaktionsdiagramm erfaßt üblicherweise das Verhalten eines einzelnen Anwendungsfalls. Das Diagramm zeigt exemplarisch eine Anzahl von Objekten und die Nachrichten, die zwischen diesen Objekten innerhalb des Anwendungsfalls ausgetauscht werden. Ein Sequenzdiagramm ist ein spezielles Interaktionsdiagramm [SF98].

Beispiel: Abb 5.9 Seite 38 zeigt ein Sequenzdiagramm für eine Beispielsituation aus unserem Dozenten/Studenten-Verwaltungssystem.

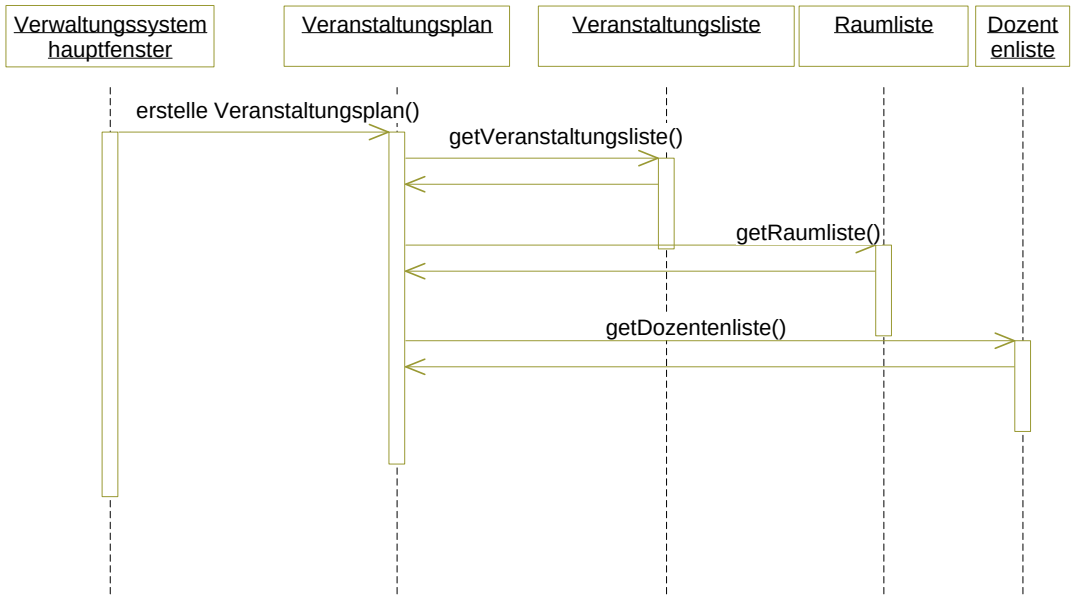


Abbildung 5.9.: Sequenzdiagramm einer Beispielsituation aus dem Dozenten/Studenten-Verwaltungssystem

5.3.4. Zusammenfassung

In diesem Abschnitt wurden die wesentlichen UML-Diagramme präsentiert. In der Analysephase werden hauptsächlich Use-Case-Diagramme und Klassendiagramme benutzt, in der Entwurfsphase ebenfalls Klassendiagramme aber auch Sequenzdiagramme zum Verdeutlichen des, nicht aus Klassendiagrammen unmittelbar verständlichem Nachrichtenaustauschs zwischen Klassen. Es gibt noch andere UML-Diagramme, die in diesem Abschnitt nicht betrachtet wurden und für uns nicht näher von Interesse waren:

- Kollaborationsdiagramm
- Komponentendiagramm
- Verteilungsdiagramm
- Zustandsdiagramm
- Aktivitätendiagramm u.a.

6. *DoDL* - Document Description Language

Autor: *Matthias Dorka*

Diese Ausarbeitung des Vortrags: *DoDL - Document Description Language*, gehalten im Rahmen der Projektgruppenfahrt der PG $\mathcal{HEU}$ führt in die Grundlagen dieser objekt-orientierten Dokumentenbeschreibungssprache ein. Es werden die zugrundeliegenden Konzepte erläutert sowie wesentliche Sprachkonstrukte einführend beschrieben.

6.1. Motivation

Wozu brauchen wir eine Beschreibungssprache für Hyperdokumente, wie sie in [Dob96] vorgeschlagen wird? Wir alle kennen Ausprägungen von Hyperdokumenten, also Dokumenten, die Verknüpfungen enthalten, aus unserem täglichen Umgang mit dem Computer: Online-Hilfen, Texte im Portable Document Format und nicht zuletzt natürlich Dokumente im World Wide Web, die in HTML vorliegen.

Die Erstellung und Wartung solchermaßen vorliegender Dokumente geschieht i. d. R. mittels textueller oder bestenfalls grafischer Editoren, die allesamt gemein haben, daß Hyperlinks explizit angegeben werden müssen, was bei umfassender Verlinkung in großen Dokumenten unpraktikabel wird.

Stellen wir uns ein Hyperdokument als gerichteten Graphen vor, so entsprechen die Knoten den Stellen, von denen Links ausgehen und wo sie enden (Wörter, Buttons, Grafiken), und die Kanten repräsentieren die Hyperlinks dazwischen. Dieser Graph ist ist jedoch nur als Visualisierung verwendbar, wenn es um den Entwurf oder die Wartung privater Homepages oder ähnlich kleiner Dokumente geht. In Anbetracht der enormen Größe, die Hyperdokumente haben können, sind enge Grenzen gesetzt bezüglich Darstellbarkeit und Übersichtlichkeit z. B. auf Bildschirmen. Man denke nur an das Online-Angebot eines Unternehmens, das im Web Informationen über sich selbst, seine Produkte und Dienstleistungen veröffentlicht oder eine Institution, die umfangreiche statistische Daten, Gesetzestexte oder ähnliches bereitstellt. Diese Daten füllen rasch einige tausend Internetseiten im herkömmlichen Sinne, die Anzahl der Links dürfte ein vielfaches betragen.

Trotz der inhaltlichen Struktur des Ganzen ist das zugehörige Hyperdokument kaum überschaubar, geschweige denn komfortabel modifizier- oder wartbar.

Dies ähnelt einer uns bekannten Situation, nämlich dem Erstellen großer Softwareprojekte. Die Probleme sind vergleichbar: die Übersicht geht verloren, kleine Änderungen

verursachen hohen Aufwand und die Fehlersuche und -behebung wird erschwert. So ergab sich die Idee, die aus der Software-Technologie bekannten Ansätze zur Lösung dieser Probleme auf Hyperdokumente zu übertragen: formale Spezifikation, systematisches Design, z. B. objekt-orientiert, also allgemein Methoden der Abstraktion, vereint in der Sprache *DoDL* (siehe [Dob96]).

6.1.1. Die Sichten von *DoDL*

Zur Übertragung der o. g. Idee auf Hyperdokumente führt *DoDL* ein Betrachtungsmodell mit drei Sichten ein, der Datensicht, der Hypersicht und der Durchlaufsicht.

6.1.1.1. Die Datensicht

beschreibt den Aufbau der Spezifikation des Hyperdokuments, hier die objekt-orientierte Spezifikation des Graphen der Hypersicht.

6.1.1.2. Die Hypersicht

beschreibt einen gerichteten attributierten Graphen, der das Hyperdokument im ganzen repräsentiert. Zu beachten ist hierbei, daß dieser Graph im allgemeinen nicht zusammenhängend ist. Dieser Graph, im folgenden Hypergraph genannt, entspricht der im Kapitel Motivation genannten Vorstellung eines Hyperdokumentes als Knoten- und Kantenmenge, bringt jedoch als zusätzliche Merkmale mit, daß seine Knoten und Kanten attributiert sind, wobei wir auf diese Attribute später noch eingehen werden.

6.1.1.3. Die Durchlaufsicht

hat als grundliegende Idee, daß es bei einem Hyperdokument sinnvoll sein kann, dem Benutzer nur Teile hiervon freizugeben, die er traversieren kann. Während bei heute gängigen Hyperdokumentsystemen nur eine Art von Link zur Verfügung steht, die vom Betrachter gleichberechtigt und nach Belieben durchlaufen werden können, gibt es hier die Überlegung, von allen möglichen Links nur eine Teilmenge freizugeben. Dabei soll das Freigeben von Links von der Auswertung einiger der bereits erwähnten zusätzlichen Attribute abhängen. Diese können zum einen an den Kanten des Hypergraphen stehen, beispielweise im Sinne eines "Wegweisers", der verrät, wohin der Link führt. Zum anderen könnte man dem Benutzer Attribute zuweisen, die z. B. seine Interessen oder Zugriffsrechte repräsentieren. Statisch betrachtet ist diese Durchlaufsicht also ein Teilgraph des Graphen der Hypersicht, aus dem gemäß Auswertung einer Auswahlregel Knoten und Kanten herausgefallen sind.

Eleganter und in den Grundzügen so beabsichtigt ist jedoch eine dynamische Auswertung der Auswahlregel zur Laufzeit, d. h. beim Betrachten des Hyperdokuments durch den Benutzer. Auf diese Weise sind viele interessante Mechanismen denkbar, z. B. ein Creditsystem, das dem Benutzer das Traversieren bestimmter Links erst nach Erreichen

einer gewissen persönlichen Punktzahl erlaubt, die sich durch das Besuchen oder Bearbeiten anderer Hyperdokumentseiten ergibt. Eine mögliche Anwendung hierfür wären beispielsweise interaktive Lernsysteme.

6.1.2. Realisierung der Sichten in *DoDL*

Um zu verstehen, wie *DoDL* die o. g. Sichten realisiert, betrachten wir zunächst das Grundgerüst einer Klasse in *DoDL*. Dieses sieht folgendermaßen aus:

```
class <name> is
   <other name> with
     declare ...
     documents ...
     construct ...
     browsing ...
end <name>
```

Die Bedeutung der einzelnen Schlüsselwörter und eine Zuordnung zu den verschiedenen Sichten wird im folgenden erläutert.

6.1.2.1. Die Realisierung der Datensicht

Die Datensicht, als Beschreibung des Aufbaus der Spezifikation des Hyperdokuments, findet sich in den ersten vier Zeilen des Klassengerüstes wieder. Das Schlüsselwort `class` zur Einleitung einer *DoDL*-Klasse wird gefolgt von dem Namen der Klasse. Die in *DoDL* propagierte Einfacherbung wird durch `is <other name>` beschrieben, wobei die Klasse `<name>` die gesamte Definition der Klasse `<other name>` additiv erbt.

Dem Schlüsselwort `declare` folgt die Definition lokaler Klassen mit Hilfe der gleichen Sprachkonstrukte, hinter `documents` werden Variablen von vordefinierten Typen deklariert, wobei die Deklaration einer Variablen vom Typ einer lokalen Klasse zur Instanziierung dieser Klasse führt. Als Typ steht zunächst der alles umfassende Grundtyp `DBUnit` zur Verfügung, eine Erweiterung um einige Standarddatentypen wie `Integer` oder `Text` im Sinne eines String erscheint jedoch sinnvoll. Um auch mengenwertige Attribute bequem handhaben zu können, steht die Datenstruktur Liste zur Verfügung.

An dieser Stelle muß erwähnt werden, daß an diese Variablen noch die konkreten Dokumentinhalte in Form von (Text-, Bild-, Ton- oder Video-) Dateien aus der Datenbank angebunden werden müssen. Dies geschieht durch die sogenannte `binding section`. Sie steht hinter der Klassenbeschreibung und weist, separat für jede Klasse, den Variablen Werte zu. Diese Konstruktion erlaubt die Wiederverwertung von Klassen für andere Daten bzw. das leichte Austauschen von Daten, ohne daß sich die Hyperdokumentspezifikation ändern muß.

6.1.2.2. Die Realisierung der Hypersicht

Der Graph der Hypersicht, die oben eingeführt wurde, wird mit Regeln beschrieben, die hinter `construct` stehen. Da im Prinzip frei programmiert werden kann, können Konstruktionsregeln verwendet werden, die beim Erstellen der zukünftigen Links z. B. Bedingungen berücksichtigen oder in Schleifen ablaufen.

So kann man sich Befehle denken, die zwischen jedem Vorkommen des Strings "Auto" in einem Text-File einen Link zu einer Grafik spannen. Damit sind Anker und Anzahl der Links nur noch vom Inhalt des Text-Files abhängig, wird es ausgetauscht oder verändert, ergibt sich kein zusätzlicher Wartungsaufwand in der *DoDL*-Spezifikation des Hyperdokuments.

Um dies zu erreichen, wurde in der urspünglichen Fassung von *DoDL* die Programmiersprache PROLOG innerhalb der `construct`-Sektion verwendet. Da viele Sprache für diesen Zweck geeignet sind, macht man sich in der Praxis die "Slot-Eigenschaft" von *DoDL* zunutze und verwendet mittlerweile C an dieser Stelle.

"Slot-Eigenschaft" bedeutet im übrigen, daß *DoDL* eine Klammer um bestehende Beschreibungsformalismen spannt, d. h. diese in gewisser Weise auch austauschbar macht. Man kann sich auch die Verwendung von Java innerhalb der `construct`-Sektion vorstellen.

6.1.2.3. Die Realisierung der Durchlaufsicht

Hier wird Smolka's *feature logic* (siehe [Smo92]) benutzt, um mögliche Traversionen durch den eben beschriebenen Hypergraphen zu definieren. Dabei handelt es sich um eine Subsprache der Prädikatenlogik erster Ordnung, die bisher auch zur Versionskontrolle verwendet wurde.

Sie ermöglichen, eine Auswahlregel aufzustellen, nach der Links des Hypergraphen zum Durchlaufen freigegeben werden. Dies kann in Abhängigkeit von Attributen an Knoten und Kanten geschehen, und auch dem Betrachter des Hyperdokuments könnten Attribute zugeschrieben werden. Dabei wird zur Laufzeit die feature-Formel unter Berücksichtigung der Wertbelegungen dieser Attribute ausgewertet und als Ergebnis ein von dieser Stelle ausgehender Link freigeschaltet oder nicht.

Intention des Ganzen ist, im Gegensatz zu bekannten Hypertextsystemen, die statisch fest für jeden Betrachter in jeder Situation sind, Varianten des Dokuments zu schaffen, die fortgeschrittenen Benutzern beispielsweise mehr Betrachtungsmöglichkeiten bieten als Gastnutzern, oder das oben geschilderte interaktive Lernsystem zu realisieren.

6.1.3. Zusammenfassung und Ausblick

Wir erhalten mit *DoDL* also eine Möglichkeit, Hyperdokumente mit formalen Methoden zu beschreiben. Weitreichend erforschte Formalismen wie Hornklauseln oder feature logic sind ein stabiles Gerüst zur Spezifikation und obendrein mit Rechnern faßbar. Durch die Trennung der Sichten erhalten wir Variabilität, neben der bereits beschriebenen leichten Austauschbarkeit einzelner Daten ist das Wiederverwenden einer *DoDL*-Spezifikation für

komplett andere Datenbankinhalte möglich, und schließlich kann ein und dasselbe Hyperdokument wiederum verschiedene Durchlaufsichten zeigen.

Der große Gewinn liegt sicherlich in der Möglichkeit, ein Hyperdokument mit Methoden des objekt-orientierten Entwurfs sauber zu strukturieren (separation of concerns), sowie die mächtigen Designmöglichkeiten zu nutzen. Hierdurch ergibt sich wiederum eine wesentlich bessere Änder- und Wartbarkeit des Systems in Analogie zur Problematik großer Softwareprojekte. Als Nachteile von *DoDL* sind zu nennen, daß sich diese Chancen nur dann erschließen, wenn das Hyperdokumentsystem groß ist und bereits die Inhalte eine gewisse Grundstruktur mitbringen bzw. eine Strukturierung erlauben. Sicherlich ist die Verwendung von *DoDL* nicht sinnvoll, wenn es um das bequeme und rasche Erstellen einer privaten Homepage geht.

Des weiteren ist der hohe Grad der Abstraktion in *DoDL* ein Erschwernis für Hyperdokumentautoren. Er erlaubt zunächst nur wenig intuitives Arbeiten mit möglichen Entwicklungstools. So hat z. B. die Klassenstruktur der *DoDL*-Beschreibung prinzipiell keine äußerliche Ähnlichkeit mit dem Aussehen des Hypergraphen oder Durchlaufgraphen.

Schließlich bleibt zu nennen, daß *DoDL* wissenschaftliches Neuland darstellt, in dem noch einige Fragen, insbesondere bezüglich der Pragmatik, offen sind, auf die an dieser Stelle aber nicht weiter eingegangen werden soll. Genau hier liegen aber auch noch Chancen für uns, die Projektgruppe $\mathcal{HEU}$ *DoDL* weiterzuentwickeln und reifen zu lassen, um zumindest eine "wissenschaftliche Spielwiese" aufrecht zu erhalten, die Platz bietet für Diplomarbeiten und weitere Projektgruppen.

7. Verteilte Systeme

Autor: *Sascha Lüdecke*

Verteilte Systeme sind ein Paradigma, um fehlertolerante, skalierbare und flexible Systeme zu implementieren. In diesem Abschnitt werden neben einer Einführung der wesentlichen Begriffe und zentralen Gesichtspunkte des Entwurfs, die typischen Problemstellungen Verteiler Systeme vorgestellt. Grundlage dieser Ausarbeitung ist der zweite Teil von [Tan92].

7.1. Einleitung

Bis Anfang der 80er Jahre dominierten Mainframes die Computerlandschaft. Dementsprechend wurden große Applikationen entwickelt, die nur auf einem Computer liefen. Mitte der 80er Jahre kamen dann die Mikrocomputer auf den Markt und bereits nach kurzer Zeit konnte man einen Computer erwerben, der die Rechenleistung eines Mainframe hatte und nur einen Bruchteil kostete. Dazu kam die Entwicklung der Netzwerktechnologie, die einen „schnellen“ Datenaustausch zwischen benachbarten Computern ermöglichte (Local Area Network).

Die so entstandenen Computernetzwerke setzen sich aus einer großen Anzahl von CPU's zusammen, verbunden durch ein schnelles Netzwerk. Damit war der Weg für verteilte Anwendungen geebnet. Während an den Mainframes noch viele 'dumme' Terminals hingen und die Mainframe die gesammte Arbeit verrichtete, war die Rechenkraft in den neuen Netzwerken auf viele Computer verteilt. Diese neue „Form“ der Hardware erforderte ein radikales Umdenken im Bereich der Software weg von zentralistischen Programmen hin zu verteilten Systemen.

Eine der treibenden Kräfte ist die Ökonomie. Man kann ausrechnen, das viele kleine Computer mehr Rechenleistung haben, als ein großer je habe könnte, die Grenzen setzt die Physik. Eine andere ist die Ausfallsicherheit des Systems. Wenn ein Teil im Verteilen System ausfällt, muß nicht gleich das ganze System zum Stillstand kommen.

7.2. Stark und lose gekoppelte Systeme

Ein Verteiltes System besteht im allgemeinen aus verschiedenen Hard- und Softwarekomponenten, die weiter in kleinere Teile zerfallen. Den Zusammenhang zwischen den einzelnen Teilen bezeichnet man je nach Art als „lose“ oder „starke“ Kopplung. Idealer-

weise hat der Benutzer dennoch eine einheitliche Sicht auf das System, ein „single system image“. Beide Begriffe sollen an Hard- und Software erläutert werden.

7.2.1. Hardware

Das erste Beispiel ist ein Multiprozessorensystem. Hier gibt es viele Prozessoren, die sich einen gemeinsamen Speicher teilen und über ein schnelles und zuverlässiges Bussystem miteinander verbunden sind. Der Datenaustausch zwischen den Prozessoren ist einfach und es läuft ein einziges Betriebssystem für alle Prozessoren. Den Zusammenhang in einem solchen System bezeichnet man als *starke Kopplung*.

Das andere Beispiel sind Multicomputersysteme. Hier gibt es ebenfalls viele Prozessoren, jedoch sind sie an unterschiedlichen Orten. Jeder Prozessor hat seinen eigenen Speicherbereich und seine eigene Kopie eines Betriebssystems. Die einzelnen Prozessoren sind über ein vergleichsweise langsames und unzuverlässiges Netzwerk miteinander verbunden. Man spricht von einer *losen Kopplung*.

7.2.2. Software

Die gleiche Unterteilung mach man bei Software oder einer Applikation. *Starke Kopplung* bedeutet dabei, daß die einzelnen Komponenten nicht eigenständig sind und der Ausfall einer Komponente das ganze System bedroht. Ein Beispiel wäre die stark parallelisierbare Suche in Spielbäumen wie z.B. beim Schach. Fällt ein Teil aus, so kann evtl. der richtige Zug nicht gefunden wird, und das System verliert.

Der Gegensatz dazu ist ein *lose gekoppeltes* System. Die sehr eigenständigen Teile interagieren nur bis zu einem gewissen Grad miteinander. Fällt eine Komponente aus, so sind die übrigen häufig noch brauchbar. Beispiele wären Druck-, Web- oder Fileserver; fallen diese aus, kann man ja meist seinen Text noch fertigstellen oder auf einen alternativen Server ausweichen.

7.2.3. Beispielkonstellation

Die Arten des Zusammenhangs bei Hard- und Software sollen zwei weitere Beispiele erhellen:

Kopplung SW	HW	Beispiel
lose	lose	Lokales Netzwerk mit Lastverteilung durch rlogin oder das NFS von SUN
stark	lose	Nach außen eine Applikation, innen verteiltes System (“single system image”) Typisch: Client/Server-Systeme wie Flugbuchungen

Tabelle 7.1.: Beispielkonstellationen von Kopplungen

7.3. Designaspekte und Architekturen

Nach der Erläuterung der Begriffe 'starke' und 'lose' Kopplung konzentrieren wir uns in diesem Abschnitt auf die typischen Aspekte unter denen man Verteilte Systeme entwirft und bewerten kann. Anschließend werden typische Architekturen vorgestellt.

7.3.1. Designaspekte Verteilter Systeme

Transparenz Ein Verteiltes System sollte sowohl für den Benutzer als auch für den Programmierer transparent bzw. als „single system" sichtbar sein. Das betrifft unter anderem den konkreten Ort von Daten, eine eventuell parallele Ausführung von gleichem Code und Bewegung derselben im System.

Flexibilität Das Verteilte System sollte flexibel sein. Dies ist normalerweise eine immanente Eigenschaft, denn einzelne Teile sind gerade bei einer losen Kopplung meist im laufenden Betrieb austauschbar. Ein gutes Beispiel ist sicher das Konzept des Mikrokernels bei Betriebssystemen oder der schon erwähnte Druckserver.

Verläßlichkeit Dieser Aspekt wurde bereits in der Einleitung erwähnt und war bzw. ist eine der treibenden Kräfte bei der Entwicklung von Verteilten Systemen. Er bezieht sich auf mehrere Eigenschaften, wesentlich ist hier die Verfügbarkeit des Systems, dazu ein Zitat über Verteilte Systeme:

> A distributed system is ...one on which I cannot get any work done because some machine I have never heard of has crashed.
> Leslie Lamport, aus [Tan92]

Andere Eigenschaften sind die Verfügbarkeit und Konsistenz der vorgehaltenen Daten, die Absicherung des Systems gegen unbefugte Zugriffe und Fehlertoleranz bei Teilausfällen des System.

Performanz Diesen Aspekt kann man unter verschiedenen Gesichtspunkten betrachten. Diese sind die Antwortzeit des Systems, Netz bzw. CPU-Last und Datendurchsatz. Beim Entwurf des Systems sollte man auf ein ausgewogenes Verhältnis zwischen Nachrichtenlaufzeit und Rechenzeit achten und dementsprechend die Granularität der Teilprobleme wählen.

Skalierbarkeit Es stellt sich die Frage, ob Systeme, die für wenige hundert Knoten entworfen wurden, auch bei einer wesentlich höheren Anzahl noch vernünftig arbeiten, so zum Beispiel eine Adressdatenbank, die zum nationalen Telefonverzeichnis ausgebaut wird. Die Daten an einer Stelle zu halten wäre nicht nur fehlerträchtig, sondern auch ineffizient. Alle Anfragen müßten einen weiten Weg bis dorthin zurücklegen und ein einziger Stromausfall könnte das gesamte System lahmlegen. Die Konsequenz dieser Erkenntnis ist die Vermeidung von zentralen Punkten („single point of failure") und Flaschenhälsen. Bei dezentralen Algorithmen kann ein Teilausfall das gesammte System nicht mehr stoppen, das weltumspannende Internet

mit dem Transport von Datenpacketen, deren Weg nich an einer Stelle berechnet wird, ist ein gutes Beispiel dafür. Solcher Algorithmen habe folgende Eigenschaften:

- Keine globale Sicht auf den Gesamtzustand
- Entscheidungen nur auf Basis lokal verfügbarer Informationen
- Teilausfälle stoppen den Algorithmus nicht
- Es gibt keine globale Zeit

7.3.2. Typische Architekturen verteilter Systeme

Unter den inzwischen vielen bestehenden verteilten Systemen haben sich drei verschieden Architekturen herauskristalisiert:

Client/Server-Modell Ein oder mehrere Teile arbeiten als Server und bieten Funktionalität an, die von den Clients genutzt wird. Beispiel: Webserver, Druckserver

Zentraler Datenspeicher Ein Teil arbeitet als zentraler Datenspeicher mit hoher Ausfallsicherheit, die anderen Teile greifen darauf zu und lösen verteilt ein Aufgabe, z.B. Flugbuchungssystem

Verteilte Applikation Hier gibt es keine ausgezeichneten Punkte. Die Applikationslogik ist samt den Daten über die Knoten verteilt, z.B. ein System aus Jini-Komponenten. Einzelne Teile stellen hier ihre Dienste im Netz zur Verfügung, was sowohl Rechenleistung, spezielle Hardware, Kommunikationsmöglichkeiten als auch sekundärer Speicher sein kann. Andere nutzen diese Dienste als Client und können ihrerseits wieder Dienste anbieten, so daß eine verteilte Applikation entsteht. Der Zusammenschluß von Komponenten ist dynamisch, es besteht eine lose Kopplung sowohl auf Hard- als auch aus Softwareseite. Die Mechanismen zur Kommunikation werden im wesentlichen von Jini bereitgestellt (siehe auch [Dör98]).

Für den Entwurf eines Verteilten Systems gibt es keine einheitliche Vorgehensweise, da dieser Ansatz noch sehr jung ist. Er ist leistungsfähiger als monolithische Ansätze, aber auch um so komplexer. Er bietet die Möglichkeit, ein System ausfallsicher, skalierbar und flexiebel zu einem günstigen Preis-/Leistungsverhältnis zu entwickeln.

Um ein Problem zu lösen, muß man es sorgfältig in eingeständige Komponenten bzw. Prozesse unterteilen, wobei von den in Abschnitt 7.3.1 auf der vorherigen Seite genannten Punkte Transparenz und Flexibilität im Vordergrund stehen sollten. Zu vermeiden sind zentrale Komponenten, Daten und Algorithmen, „single point of failure“ eben.

Da man auf neuen Gebieten überlicherweise Fehler macht, ist die Erweiterbarkeit bzw. Flexibilität des Entwurfs an sich ein nicht zu vernachlässigender Punkt.

7.4. Kommunikation in verteilten Systemen

Betrachten wir nun die Kommunikation der einzelnen Komponenten eines Verteilten Systems. Wir sehen die Komponenten dabei als Prozesse, die miteinander Informationen

austauschen und konzentrieren uns auf potentiell lose gekoppelte Systeme. Gemäß philosophischer Untersuchungen über Sprachhandlungen, kann man die Kommunikation, je nach Zielsetzung, in verschiedene Akte unterteilen (siehe Teil II in [Fer98]):

Kooperation Zwei Prozesse sprechen sich ab, um eine Aufgabe gemeinsam zu bearbeiten.

Datenaustausch Zwei Prozesse tauschen Daten, z.B. Berechnungsergebnisse, miteinander aus (auch Synchronisation von Daten).

Benachrichtigung Ein Prozess informiert einen anderen über das Ende einer Operation, die Änderung eines Zustandes oder ein bestimmtes Ereignis.

Anfragen Ein Prozess fragt einen anderen nach Daten oder stößt eine Handlung an, die ein Ergebnis liefert.

Diese Kommunikationsakte müßen nicht immer nur zwischen zwei Prozessen ablaufen. Man unterscheidet die Begriffe „unicast“, 1:1 Kommunikation, „multicast“, 1:n Kommunikation und „broadcast“, Nachricht an alle Beteiligten.

Das Fehlen von gemeinsamen Speicher macht einen neuen Ansatz zur Kommunikation erforderlich. Eine Möglichkeit ist über „Message Passing“, was allerdings sehr primitiv und alles andere als transparent ist. Der Programmierer wird direkt mit den Problemen der Kommunikation, wie Verpackung der Daten und Aufbau der Verbindung, konfrontiert.

Komfortabler ist der RPC-Mechanismus („remote procedure call“) von SUN. Damit ist ein transparenter Aufruf von Methoden auf fremden Maschinen und so eine Verteilung der Funktionalität möglich. Noch weiter gehen Ansätze wie CORBA oder RMI von Java: entfernte Objekte können transparent und mit geringem Mehraufwand lokal eingesetzt werden. Notwendig sind hier nur noch Schnittstellendefinitionen und die Initiierung der Verbindung.

All diese Ansätze befreien den Programmierer jedoch nicht von dem grundlegenden Problem der Kommunikation: Ist eine Nachricht wirklich angekommen? Typische Lösungsansätze für dieses Problem sind:

- Timeout beim Aufruf entfernter Methoden
- Empfangsbestätigung des Empfängers
- Ergebnis der Methode als Bestätigung für Empfang

An dieser Stelle soll die Kommunikation nicht weiter vertieft werden, da sie mit RMI oder CORBA für den Programmierer weitestgehend transparent ist.

7.5. Synchronisation in verteilten Systemen

Abschließend soll ein wichtiger Bereich der Kommunikation zwischen zwei Prozessen genauer erläutert werden: die Synchronisation. Sie dient dazu, Datenbestände oder auch Prozesszustände abzugleichen. Üblicherweise wartet dabei ein Prozess auf den anderen.

Wichtig ist die Synchronisation vor allem bei dem Zugriff auf seltene Resourcen und kritische Bereiche, wie zum Beispiel besondere Hardware oder eine Brücke zu einem anderen Teil des Systems. In Einprozessorsystemen verwendet man dazu Semaphore, die immer nur an einen Prozess vergeben werden und vom Betriebssystem einfach verwaltet werden können. Verteilte Systeme sind an dieser Stelle komplexer: wie schon im Abschnitt 7.3.1 auf Seite 49 erwähnt, gibt es gerade bei dezentralen Algorithmen keine globale Zeit bzw. diese ist nur aufwendig zu ermitteln, was eine einfache Zuteilung oder Abstimmung von Prozessen verhindert.

Drei Methoden zur Zugriffsteuerung (Ausschlußverfahren) auf kritischen Bereiche haben sich etabliert:

Koordinator Es gibt einen Prozess, der die Zugriffe auf den kritischen Bereich verwaltet und alle eingehenden Anforderungen in eine Warteschlange einreiht. Problematisch ist der Absturz des Koordinators (siehe dazu Bully-Algorithmus, Abschnitt 7.5.1).

Broadcast Ein Prozess verbreitet an alle eine Nachricht, daß er gleich auf den kritischen Bereich zugreifen wird, auf die die anderen Prozesse unterschiedlich reagieren:

- Prozesse, die nicht interessiert sind, geben ihr OK
- Der Prozess, der im kritischen Bereich ist, speichert die eingehende Nachricht für spätere Rückmeldung.
- Prozesse, die selbst auf den kritischen Bereich zugreifen wollen, vergleichen die Nachricht mit der selbst verschickten Anforderung, wobei der früheste Absender gewinnt.

Der Prozess betritt den kritischen Bereich, sobald alle anderen ihre Zustimmung gegeben haben. Problematisch ist bei diesem Verfahren das Fehlen einer globalen Zeit. Man kann zwar eine zeitliche Ordnung von Ereignissen im System herstellen, das ist jedoch nicht trival und würde den Rahmen dieses Abschnittes sprengen.

Tokenring Bei diesem Verfahren bilden alle Prozesse einen Ring und reichen ein Token herum, dessen Besitzer auf den kritischen Breich zugreifen darf. Nach einer gewissen Zeitspanne muß das Token weitergegeben werden. Problematisch ist hier der Absturz eines Prozesses, der Verlust des Tokens und die Wartezeit auf das Token. Dieses Verfahren stammt aus der Netzwerktechnologie und wird auch hauptsächlich dort eingesetzt.

Tabelle 7.2 auf der nächsten Seite stellt die Ausschlussverfahren einander gegenüber (siehe auch [Tan92], Kapitel 11.2.4, Seite 482).

7.5.1. Bully Algorithmus

Abschließend zur Synchronisation soll ein Lösungsansatz angeführt werden, der beim Absturz eines Koordinators einen neuen bestimmt:

Bully Algorithmus (Garcia Molina 1982) aus [Tan92]:
Wenn ein Koordinator nicht mehr antwortet, soll folgendes getan werden:

Verfahren	Anzahl Nachrichten	Wartezeiten (in Nachrichten)	Probleme
Koordinator	3	2	Absturz des Koordinators
Broadcast	$2(n-1)$	$2(n-1)$	Absturz irgendeines Prozesses
Tokenring	1 bis unendl.	0 bis $n-1$	Token verloren, Prozessabsturz

Tabelle 7.2.: Vergleich von Ausschlussverfahren

1. Der Prozess P sendet eine „Election"-Nachricht an alle anderen in Frage kommenden Prozesse
2. bekommt er keine Rückmeldung, wird er neuer Koordinator
3. sonst fährt der Beste, der sich meldet mit dem Algorithmus fort

Am Ende gilt: „best in town wins"

Leider gibt es wie bei der Kommunikation ein grundlegendes Problem bei der Synchronisation, die Deadlocks. Sie treten auf, wenn zwei Prozesse je auf die Reaktion des anderen warten. Zwar gibt es bekannte Vorbedingungen, Deadlocks zu erkennen ist aber sehr schwer. Daher gilt auch hier: der Programmierer muß sich selbst darum kümmern.

7.6. Weiterführende Aspekte

Im Laufe dieser Einführung sind einige Themen aus dem Bereich der Verteilten Systeme nicht oder nur kurz angesprochen worden.

- Entwurf verteilter Algorithmen
- Beweglicher Code
- Vermeidung von Deadlocks

8. Integrierte Entwicklungsumgebungen

Autor: *Thomas Sparenberg*

Die Quellen, die dieser Ausarbeitung zu Grunde liegen, sind vom Anfang der 1990 Jahre. Das war ein Zeitraum, in dem die Probleme mit komplexer Software immer häufiger auftraten und in der der Ruf nach (integrierten) Entwicklungsumgebungen aufkam. Seiner Zeit haben sich bestimmte Persönlichkeiten aus Politik und Wirtschaft zusammengesetzt, und dafür einen Standard, PCTE, entworfen. Dieser Standard stellt eine ganze Reihe von Vorgaben auf, die heute teilweise als unrealisierbar eingestuft werden. Zur Erfüllung dieser Vorgaben wird in der Zusammenfassung noch einiges gesagt werden, der folgende Text beschreibt aber das damalige Wunschdenken der Entwickler dieses Standards.

8.1. Motivation

Wie die Vergangenheit zeigt, halten die Computer immer mehr Einzug in alle möglichen Bereiche der Geschäftswelt und des Privatbereiches. Aber nicht nur die Computer sind auf dem Vormarsch, zusätzlich wird die Hardware wird immer billiger und leistungsfähiger. Dadurch stieg in den vergangenen Jahren auch die Leistungsfähigkeit und die Komplexität der Software an. Und Softwareentwicklung ist seit langem teuer und wird eben nicht billiger. Deshalb ist es sehr wichtig, daß diese Software schnell erstellt werden kann, zuverlässig und flexibel ist, leicht modifiziert werden kann und termingerecht ausgeliefert wird. Durch jahrelange Entwicklungstätigkeiten wurde von den verschiedensten Gruppen, wie z.B. Akademikern, Anwendern oder Hardware-Herstellern, ein gutes allgemeines Verständnis dafür entwickelt, wie man Softwareentwicklung effizient und vorausplanbar macht. Und das nicht nur durch Entwicklungserfahrungen, sondern auch durch Analogien zu anderen Ingenieurtätigkeiten, durch Qualitäts- und Informationsmanagementstandards. Man konzentrierte sich neben den Bereichen Prozeß-Modelle und Methoden/Techniken auch auf den Bereich der Entwicklung von CASE-Tools (Computer aided software engineering) [Tho93].

Die Steigerungen an Effizienz und Genauigkeit durch diese CASE-Tools führten seiner Zeit zu Übererwartungen - und den daraus resultierenden Enttäuschungen. Man hat gelernt, daß die automatischen Anteile am Entwicklungsprozeß nicht alleine die Kontrolle und die Effizienz des Ganzen verbessern können. Das eigentlich zu lösende Problem ist die Verteilung und die Kontrolle der Informationen und der Produkte, auf denen die CASE-Tools arbeiten, damit die komplexen Beziehungen zwischen z.B. dem Design, dem

dem Design entsprechenden Code, die Arbeitsverteilung im Projektteam und dem ausgelieferten System durch den gesamten Lebenszyklus nutzbar gemacht und unterhalten werden können. Fehler bei der Lösung dieses Problems führen u.a. zu Verdopplungen und Inkompabilität von Informationen. Diese Computer-unterstützte Softwareentwicklung durch alle Lebenszyklen führete dann zum Konzept der integrierten Softwareentwicklung.

8.2. Softwareentwicklung

Wenn neue Software entwickelt werden soll, muß man eine Reihe grundlegender Vorgaben einhalten. Zuerst einmal muß die Software korrekt spezifiziert werden und auch genau diese Spezifikationen einhalten. Weiterhin sollte sie rechtzeitig fertiggestellt werden (d.h. wenn man einen Termin ausmacht, an dem die Software auszuliefern ist, sollte sie dann auch fertig sein) und sich gegenüber der Konkurenz auf dem Markt behaupten können (gutes Preis-Leistungs-Verhältnis). Gegebenenfalls muß die Software auf anderer Hardware lauffähig oder mit der Zeit veränderbar oder erweiterbar sein. Das kann sie nur, wenn sie entsprechend flexibel ist. Außerdem kann es vorkommen, daß man Teile der Software (z.B. Softwaremodule, Dokumentationen, Spezifikationen oder das Design) in weiteren Projekten nocheinmal benutzen möchte, also muß Software wiederverwendbar sein. Und damit man nicht von einem einzigen Hersteller abhängig ist (Hardware, Betriebsystem), und damit die Instandhaltung und die Weiterentwicklung sichergestellt werden kann, sollte Software offen sein (s.u.).

Aber allein das Vorhandensein einer Entwicklungsumgebung stellt nicht sicher, daß man alle Ziele erreicht, die man sich für die Software gesteckt hat. Damit so eine Umgebung wirklich gut genutzt werden kann, muß es einen wohldefinierten Lebenszyklus geben; außerdem sollte ein Qualitätsmanagement vohanden sein. Dazu kommt allerdings noch, daß alle Mitwirkenden (Management, Projektmitarbeiter) die Entwicklungsumgebung verstanden haben und benutzen können (und wollen). Das bedeutet für die Entwickler integrierter Entwicklungsumgebungen (also auch für die PG $\mathcal{HEU}$), daß die von ihnen entwickelten Umgebungen leicht verständlich und benutzbar sind, und daß sie vor allem dem Benutzer das Gefühl geben, ihn bei seiner Arbeit zu unterstüten, und daß er nicht lange nach dem Sinn einer solchen Umgebung suchen muß.

8.3. Entwicklungsumgebungen

In einer Enwicklungsumgebung gibt es verschiedene Dienste, die sich das Entwicklerteam zu Nutze machen kann, um die gewünschte Software zu erstellen.

8.3.1. Dienste

Es müssen alle Dienste angeboten werden, die zur Unterstützung des Entwicklungsprozesses nötig sind. Das sind solche Dienste, die die Entwicklungsaktivitäten (Modellieren,

Programmieren), die Enwicklungskontrolle (Zugriffsrechte, Fortschritt) und die Administration der Umgebung (Position eines Servers/einer Datenbank, zu generierende Programmierspache, Bildschirmfarben usw.) unterstützen.

Eine Auswahl der wichtigsten Dienste:

1. Spezifikations- und Designdienste: (front-end oder upper-case) unterstützen Techniken zur Systemanalyse, zur Spezifikation der Systemvoraussetzungen und zum Programm- und Datendesign.

2. Softwareentwicklungsdienste: (back-end oder lower-case) unterstützten die Softwareentwicklung selbst. Das können syntax-spezifische Editoren, Compiler, Debug-Tools usw. sein. Die Werkzeugwahl wird hierbei durch die Programmiersprache, die Programmierrichlinien und die Zielhardware bestimmt.

3. Dokumentationsmanagement- und Veröffentlichungsdienste: Enwicklungsprojekte produzieren viele verschiedene Dokumententypen. Diese enthalten Managementdokumente (Angebote, Vorgehensweisen, Standards), Dokumentationen als Teil des Entwicklungsprozesses (Spezifikationen, Designdokumente) und Benutzerdokumente, die zur Anwendung gehören (Handbücher). Sie alle müssen über den gesamten Lebenszyklus erreichbar sein und dementsprechend gespeichert, versioniert und verwaltet werden.

4. Datenmanagement- und Anfragedienste: Dienste zur Verwaltung der verschiedensten Datentypen. Benötigt werden Dienste zur Datenmodellierung, zur Verwaltung eines geteilten Repository und für den Datenzugriff durch Nutzer und Tools. Datenmanagementtools haben sicherzustellen, daß die Datenkonsistenz, -geheimhaltung und -unversehrtheit unterstützt wird. Außerdem müssen sie für die Elastizität im Fehlerfalle sorgen (damit diese Fehler abgefangen werden, und nicht die gesamte Umgebung mitsamt ungespeicherter Daten abstürtzt).

5. Systemadministration: (ähnlich konventionellen Betriebsystemen) beinhaltet Verwaltung der physikalischen Umgebung, Anlegen und Verwalten der Benutzer und Recourcen der Umgebung, Datensicherung und ähnliche Aktivitäten.

8.3.2. Offenheit

Eine Entwicklungsumgebung ist wertvoller, wenn sie auf verschiedenen Plattformen läuft und Werkzeuge unterschiedlichster Herkunft (Programmiersprache, Hersteller) verbindet. Weiterhin ist es wünschenswert, verschiedene Werkzeuge zwischen Entwicklungsumgebungen austauschen zu können. Um das zu erreichen, muß die Umgebung offen sein. Diese Offenheit erlaubt dann den Austausch oder Wechsel einer Entwicklungsumgebung ohne Informations- oder Werkzeugverlust. Man kann auch die Entwicklung auf verschiedene Entwicklerteams verteilen, die ihre eigenen Umgebungen benutzen, oder man kann sich selbst die besten Werkzeuge (in Bezug auf Qualität, Preis, Techniken/Methoden) aus allen zur Verfügung stehenden aussuchen. Später kann die Umgebung um neue, verbesserte Werkzeuge erweitert werden.

8.3.3. Integration

Die Umgebung soll nahtlos die Entwicklung der Software unterstützen, unzwar dadurch, daß die Werkzeuge zusammenarbeiten können, sich untereinander Daten austauschen und auf einfachste Weise mit dem Benutzer interargieren. Weiterhin soll sie ein Repository enthalten, welches eine Reihe verschiedener Aufgaben wahrnimmt. Es speichert die gesamten Produkte und Informationen des Projektes und ermöglicht den Datenaustausch zwischen den Werkzeugen. Weil alle Daten in einer einzigen Datenbank gehalten werden, verhindert man Datenredundanz und -inkonsistenz. Außerdem sollen alle Entwicklungsarbeiten in der selben Umgebung ausführbar sein, damit die Komplexität der Softwareentwicklung minimiert wird. Dadurch würden dann auch die Benutzerschnittstellen der Werkzeuge vereinheitlicht, was dazu führe, daß wenige verschiedene solcher Schnittstellen vom Entwickler beherrscht werden müßten. Vor allem soll das manuelle Transferieren von Daten von einem Werkzeug ins andere entfallen und auf automatischem Wege geschehen.

Es gibt drei verschiedene Dimensionen der Integration [Tho93], unzwar die Daten-, Kontroll- und Präsentationsintegration, die auch als "drei Achsen der Integration" bezeichnet werden.

Die Datenintegration sorgt dafür, daß die von den Werkzeugen benutzen Daten öffentlich bleiben und von allen anderen Werkzeugen verstanden und verwendet werden können. Ebenso unterstützt sie den Austausch von Daten innerhalb und zwischen verschiedenen Umgebungen. Diese Integration der Daten kann nur durch Verwendung eines völlig offenen Repository erreicht werden.

Die Kontrollintegration deckt den Grad an kooperativer Zusammenarbeit der Dienste ab, d.h. gegenseitiger Aufruf der Dienste untereinander, Verfügbarmachung relevanter Werkzeuge zu bestimmten Zeiten (Compiler erst nach Erstellung des Souce-Codes, Debugger erst danach usw.) und die automatische forschrittsabhängige Ausführung von Funktionen (Archivierung bestimmter Daten, Benachrichtigungen, Fortschrittsanzeigen und -analysen).

Um dem Benutzer/Softwareentwickler ein intuitives Verständnis für seine Umgebung zu geben, wird die Präsentation integriert. D.h. die Präsentationsintegration gleicht die Benutzung aller Werkzeuge der Umgebung aneinander an, z.B. durch die einheitliche Darstellung der Datentypen, gleichartige Benutzung der Tastatur (Funktionstasten usw.) und der Maus, durch Bereitstellung von Hilfedateien usw.

Durch die Benutzung einer integrieren Entwicklungsumgebung erwartet man sich sicherlich einen Vorteil gegenüber Entwicklungen ohne eine solche Umgebung. Diese Vorteile sind am größten für [Tho93]:

1. komplexe Software
2. sicherheitskritische Software
3. ein großes Entwicklungsteam
4. gesplittete Entwicklung in verschiedenen Teams

5. langfristige Instandhaltung
6. die Herstellung wiederzuverwendender Software
7. Forderungen nach detaillierter Aufzeichnung der Entwicklungsgeschichte.

8.3.4. Aufbau einer offenen integrierten Entwicklungsumgebung

Der ideale Hardwaretyp für eine Umgebung versorgt jeden Benutzer mit einer sehr leistungsfähigen Arbeitsstation inklusive grafischer Benutzerschnittstelle. Verbunden sind diese Stationen alle über ein Netzwerk mit einem Server und mit der dazugehörigen Ausstattung, wie z.B. Drucker und Massenspeicher. Wie schon oben erwähnt, gehört in die Entwicklungsumgebung auch ein Repository. Diese Datenbank benutzt die von der Plattform / vom Betriebsystem bereitgestellten Speicherungsmöglichkeiten und Datenzugriffsmechanismen. Das Repository kann als zentrales Element oder verteilt auf den verschiedenen Massenspeichern des Netzwerkes gehalten werden. Die Verteilung über das Netzwerk ist zwar toleranter gegenüber Ausfall eines Rechners und bringt auch schnellere Zugriffszeiten, setzt aber leistungsfähigere Datenverwaltungsmöglichkeiten als bei der zentralen Datenbank voraus. Die Schnittstellen zwischen den Werkzeugen sind standardisiert und öffentlich (PTI, public tool interfaces). Darauf aufgebaute Werkzeuge und Dienste haben Zugriff aus das Repository und auf die Systemresoucen, unabhängig von der darunterliegenden Plattformarchitektur.

8.3.5. Umgebungsaufbau (environment building)

Der Aufbau einer Entwicklungsumgebung wird unterteilt in Komplettumgebung und einer Art Grundskelett, dem Gerüst (framework). Dieses Gerüst stellt Grundsätzliches (s.u) zur Entwicklung von Software bereit. Auf so ein Gerüst kann dann eine komplette Umgebung aufgebaut werden, wobei dann die Arbeit zum Aufbau des Gerüstes nur einmal gemacht werden muß, und die äußerlich individuell und ggf. verschieden aussehenden Entwicklungsumgebungen können auf dem selben Gerüst aufgebaut sein. Das Grundsätzliche, was ein Gerüst bieten muß, ist die öffentliche Werkzeugschnittstelle (PTI), das Repository und alle Software und Daten, die zur Unterstützung der Integration benötigt werden (cross-lifecycle-tools, Umgebungsadministrationswerkzeuge, Bibliotheken und Werkzeugschnittstellen).

8.4. Toaster-Modell

Aufgrund der Idee dieses Gerüstes hat man seiner Zeit ein Referenzmodell geschaffen, welches die Unterteilung in Gesamtumgebung und Gerüst beschreibt. Da dieses Referenzmodell sehr umfangreich und nicht so leicht modellierbar ist, hat man außerdem ein Diagramm geschaffen, welches anschaulich die grundsätzliche Wirkungsweise und die Ideen des Gerüstbaus vermitteln soll. Dieses Diagramm wird aufgrund seines Aufbaus / Aussehens auch als Toaster-Diagramm oder Toaster-Modell bezeichnet. Dieses Diagramm zeigt bei weitem nicht das volle Ausmaß des Referenzmodells und kann auf keinen Fall

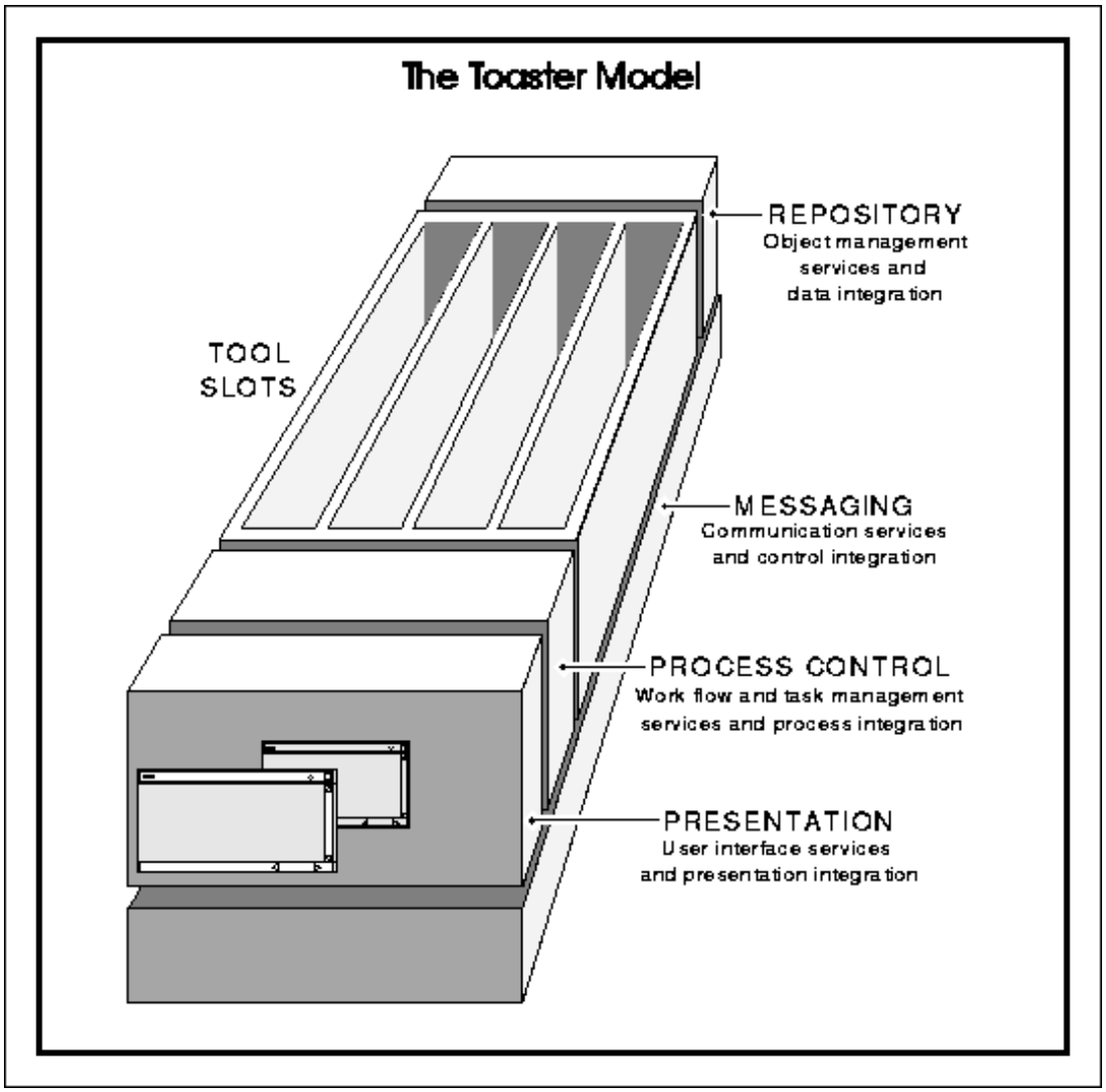


Abbildung 8.1.: Toaster-Modell

buchstäblich als Systemarchitektur interpretiert werden. Es soll nur ein Gefühl dafür vermitteln, wie man (neue) Werkzeuge in eine Entwicklungsumgebung integrieren kann, damit die Gesamtumgebung aufgebaut bzw. erweitert werden kann. Natürlich müssen die Werkzeuge auch in die "Slots" des Gerüsts passen, d.h. sie müssen auf denjenigen Schnittstellen aufsetzen, die das Gerüst bietet, z.B. um die Präsentations-, Daten- und Kontrollintegration in der Umgebung zu verwirklichen. Wenn man auf ein solches Gerüst aufbauen kann, benötigt man bei der Entwicklung neuer Tools nur wenig Aufwand für die Benutzer- und die Werkzeugschnittstellen oder für die Datenverwaltung, und man kann sich hauptsächlich auf die Funktionalität konzentrieren, kann entweder das neue Werkzeug eher in Betrieb nehmen (zeitlich, wegen geringeren Aufwands), oder, bei gleicher Entwicklungszeit, die Funktionalität erweitern. Die Interaktion mit anderen Werkzeugen und der Restumgebung ist dann Sache des Gerüstes (der Gerüstdienste).

8.5. PCTE

PCTE (Portable Common Tool Environment) nennt sich der Standard, den seiner Zeit Abteilungen des Verteidigungsministeriums der vereinigten Staaten und eine Vereinigung europäischer Comoputerhersteller (ECMA - European Computer Manufacturer Association) aufgestellt haben. Es faßt das zusammen, was eine integriere Entwicklungsumgebung bieten soll. Dabei wurde mit Standards und Festlegungen nicht gegeizt. Auch die technischen Anforderungen dieser Vorgaben waren recht hoch angesetzt, da es integrierte Entwicklungsumgebungen vorschlug, die den gesamten Lifecycle unterstützen, also von der Aufnahme der Kundenwünsche über Anforderungsanalyse, Design, Implementierung, Vertrieb bis hin zur Wartung und Erweiterung der Software. Die wichtigsten Punke sind oben schon zusammengefaßt, hier noch folgender Textabschnitt [WJ93] zur Verdeutlichung dieser sehr hoch angesetzten Erwartungen: "In einer vollständig ausgestatteten Umgebung werden die Dienste des Gerüstes den Benutzern durch eine Anzahl CASE-Tools geboten. Die Tools unterstützen sowohl generelle Bedürfnisse (Browser, Dokumentenverwaltung, Zeichentools usw.) als auch die Bedürfnisse der Entwicklung (Designtools, syntaxorientierte Editoren, Compiler). Solche Tools können in modularer Weise durch Wiederbenutzung einfacherer Tools und Verwaltung ihrer Zusammenarbeit konstruiert werden. Aus Sicht des Benutzers bedeutet die Integration von Tools, daß sie einen gewissen Grad an Uniformität besitzen und daß sie öffentliche Daten austauschen und untereinander kommunizieren können ... PCTEs Ziel ist die Schaffung eines industrieweiten Standards durch die Definition einer Schnittstelle und einer Reihe von Funktionen für die Dienste des Gerüsts und für die Datenintegration auf Basis einer Repository. Um die Akzeptanz, die Verbreitung und die universelle Benutzung zu unterstützen, was grundsätzlich wichtig ist für das Erreichen eines reelen Grades an Zusammenarbeit in der Industrie, sind die PCTE-Spezifikationen standardisiert und sowohl maschinen- als auch sprachunabhängig ... PCTE ist ein Standard für eine Reihe öffentlicher Tool-Schnittstellen und einer öffentlichen Repository, der sich an die Bedürfnisse der Produktion offener integrierter Entwicklungsumgebungen richtet. Er bietet eine Reihe von Einrichtungen an, die eine Infrastruktur für diejenigen Tools aufbauen, die Systementwicklungsprojekte unterstüt-

zen. Diese Einrichtungen sind unabhängig von den Maschinen und Betriebsystemen, auf denen sie laufen. Außerdem definiert PCTE eine PTI, die Daten- und Kontrolleinrichtungen bietet." Zur Vollständigkeit noch einige Gebiete, die selbst dieser umfangreiche Standard ausgelassen hat: "PCTE hat sich nicht um Standards aus den Gebieten Benutzerschnittstelle, allgemeine Datenmodelle oder Kommunikation zwischen Umgebungen gekümmert. Das wird Inhalt anderer Initiativen und Körperschaften sein" [WJ93].

8.6. Zusammenfassung

Wie schon am Anfang erwähnt, ist PCTE ein Standard, der vor fünf bis acht Jahren erstellt wurde und eigentlich einem Wunschzettel ähnelt, indem viele, gewiß auch erstrebenswerte Empfehlungen niedergelegt wurden. Die wichtigsten Ziele waren dabei wohl, daß Entwicklungsumgebungen offen und integriert arbeiten, plattformunabhängig bleiben und einfach zu bedienen, zu administrieren und zu modifizieren sind. Dabei verwalten sie alle Daten und unterstützen zweckmäßige Werkzeuge, und nicht nur für den Programmierer, sondern für alle Beteiligten (Designer, Administratoren). Heute, d.h. eigentlich schon im Jahre 1995 [Wil95], weiß/wußte man, daß nur einige Aspekte von PCTE in die Realität umzusetzen sind. Es ist bis heute nicht gelungen, eine Entwicklungsumgebung zu entwickeln, die von der Anforderungsanalyse bis hin zur Auslieferung und Wartung alle Lebenszyklen erfaßt und unterstützt. Ein Problem war seiner Zeit, daß es schon viele fertige Werkzeuge gab. Sie hätten alle umgeschrieben oder gekapselt werden müssen, damit sie mit den Schnittstellen einer PCTE-Umgebung zusammenarbeiten können, also integriert werden. Nach dieser Hürde war es dann auch soweit, daß viele Werkzeuge zusammenarbeiteten, sich Mitteilungen über alle möglichen Geschehnisse sandten. Leider hat niemand es jemals geschaft, ein Repository zu schaffen, daß mit dieser PCTE-artigen Umgebung zusammenarbeiten konnte. Außerdem ist das PCTE-Repository-Modell, da es objektorientiert ist, nur bis zur Dateiebene brauchbar. Es definierte keine feingranulareren Strukturen. Obwohl einige kommerzielle Werkzeug-integrierende Umgebungen geschaffen wurden, führte das Fehlen von öffentlichen Repositories und die Schwierigkeiten bei der Integration der Werkzeuge dazu, daß die Hersteller sich andere Wege ausdachten [Wil95].

Heute hat jeder Hersteller sein eigenes Repository und die darauf arbeitenden Werkzeuge. Nachteilig ist hierbei, daß man eventuell auf einen einzigen Hersteller angewiesen ist, und daß für die Entwicklung neuer Werkzeuge immer wieder integrieraufwand geleistet werden muß, der anderswo eingesetzt der Funktionalität des Werkzeugs zu gute käme [Wil95]. Und gerade das wollte der PCTE-Standard verhindern.

Jedoch sind die guten und realisierbaren Ideen von PCTE in die Tat umgesetzt worden. Es gibt mittlerweilen zahlreiche Entwicklungsumgebungen, die mehrere Lebensabschnitte der Software unterstützen, wie z.B. Rational-Rose, wie wir es auch in der PG $\mathcal{HEU}$ benutzen, bei dem man von der Analyse über Design bis Code-Erzeugung dieses Werkzeug benutzen kann. Weiterhin gibt es einen neuen und erfolgversprechenden Versuch eines Repository, genannt HPCTE, von der Universität Siegen. Dieses ebenfalls in der PG $\mathcal{HEU}$ eingesetzte Repository bietet die genügende Feingranularität und hat dabei eine ausgezeichnete Zugriffsgeschwindigkeit. Es bleibt abzuwarten, was sich daraus

entwickelt.

9. Das Objektmanagementsystem H-PCTE

Autor: *Christian Stücke*

H-PCTE steht für **H**ighspeed-**P**ortable **C**ommon **T**ool **E**nvironment und ist eine spezielle Ausprägung des PCTE-Standards, der Mitte der 80er Jahre von der ECMA erstellt wurde ([ESP85], [ECM]). Bei H-PCTE handelt es sich um ein objektorientiertes Datenbankmanagementsystem (DBMS), das speziell zur Speicherung und Bereitstellung von Dokumenten bei der Software-Entwicklung konzipiert wurde. Es erfüllt somit alle Anforderungen von Software-Entwicklungsumgebungen (SEU) an ein DBMS, wie Unterstützung rekursiver Strukturen, Verteilung und langer Transaktionszeiten, und wird somit auch als Objektmanagementsystem (OMS) bezeichnet. Die Grundlagen dieses Textes bilden [Kel98] und die Dokumentation von H-PCTE 3.0 ([PCT96], [Pla96]).

9.1. Grundlegende Konzepte von H-PCTE

Die Basis jeder H-PCTE-Objektbank bildet das Entity-Relationship-Modell (E-R-Modell). Dieses besteht aus drei grundlegenden Bestandteilen (Objekte, Beziehungen, Attribute), die zunächst in den folgenden Abschnitten behandelt werden.

9.1.1. Objekte (entities)

Objekte bezeichnen in DBMS immer etwas Seiendes. Ein **Objekttyp** besteht aus

- einer Menge von direkten **Attributen**
- einer Menge von zulässigen ausgehenden **Linktypen**
- den **Elterntypen**, wobei Mehrfacherbung (multiple inheritance) möglich ist.

Die Objekttypen bilden infolge des mehrfachen Erbens zunächst eine Halbordnung. Diese wird weiter dahingehend eingeschränkt, daß nur genau eine Wurzel zulässig ist. Dieser Wurzeltyp heißt `object`, d.h. alle anderen Objekttypen sind immer Subtyp von `object`.

9.1.2. Beziehungen (relationships), Links

In H-PCTE sind immer nur binäre Beziehungen möglich, wodurch Beziehungen immer durch zwei Links, ein Link und einen entsprechenden Umkehrlink, repräsentiert werden. Ein **Linktyp** besteht aus

- einer Folge von **Schlüsselattributen**, die einen Link eindeutig identifiziert
- einer Menge von **Nicht-Schlüsselattributen**, die der Beziehung Eigenschaften verleiht
- einer Menge zulässiger **Zielobjekttypen**, die bei Anlegen eines Links überprüft wird
- einer **Kategorie**, die die semantischen Eigenschaften eines Links darstellt
- einer **Duplikationseigenschaft**, die angibt, ob der jeweilige Link beim Kopieren des Ausgangsobjekts mitkopiert werden soll oder nicht
- einem **Umkehrlinktyp**

Durch eine Beziehung kann immer der Zusammenhang zwischen zwei Objekten ausgedrückt werden.

9.1.3. Attribute (Eigenschaften)

Attribute stellen die Eigenschaften der Objekte oder Links dar. Sie enthalten die eigentlichen Daten und dienen oft der Identifizierung von Objekten oder Links. Ein **Attribut** besteht aus

- einem **Attributnamen**
- einem **Attributtyp**, der den zulässigen Wertebereich des Attributs bezeichnet, wobei in H-PCTE nur elementare Datentypen erlaubt sind
- einem **Initialwert**, der den Startwert bei der Instanziierung angibt
- einer **Duplikationseigenschaft**, die angibt, ob das jeweilige Attribut beim Kopieren des Objekts oder des Links mitkopiert werden soll oder nicht.

9.2. Datendefinition und Datensicht in H-PCTE

Jedes DBMS gliedert sich in **drei Ebenen:**

Auf der mittleren Ebene (**konzeptuelles Schema**) werden alle Datenstrukturen und somit eine Sicht auf alle Typdefinitionen realisiert. In H-PCTE enthalten die Metadaten die Typdefinitionen und Datenstrukturen für die Objektbank. Diese Daten werden in einer Metadatenbank in sogenannten **Schema Definition Sets (SDS)** gespeichert. Diese SDS werden in der **Data Definition Language (DDL)** von H-PCTE geschrieben. Anschließend können die SDS mit dem DDL-Compiler übersetzt und somit in der Objektbank bekannt gemacht werden.

Auf der darüberliegenden Ebene (**externes Schema**) werden die verschiedenen Sichten auf die Typdefinitionen für die verschiedenen Applikationen festgelegt. In H-PCTE wird

dies durch sogenannte **Arbeitsschemata (Working Schemas)** gemacht. Eine Applikation kann eines oder mehrere SDS in ihr Arbeitsschema aufnehmen, um dadurch die enthaltenen Strukturen verwenden zu können. Die Applikation benötigt dazu allerdings die benötigten Rechte an den SDS.

Die untere Ebene (**internes Schema**) legt fest, wie die Daten intern auf dem physikalischen Datenträger abgelegt werden. Sie enthält Mechanismen zur Optimierung der Datenspeicherung.

9.3. Indentifizierung von Objekten

Hat man Sicht auf ein oder mehrere compilierte SDS, so kann man Instanzen der Objekte und Links zwischen diesen anlegen. Anschließend muß man natürlich in der Lage sein, durch die existierende Objektbank zu navigieren. Dazu bietet H-PCTE zwei **Referenzobjekte** (Ausgangsobjekte):

Langname	*Kurzname*	*Bedeutung*
`$common_root`	`_`	allgemeines Wurzelobjekt einer Installation (analog zum Verzeichnis '/' in UNIX)
`$home_object`	`~`	Heim-Objekt des Benutzers des laufenden Prozesses

Ausgehend von einem dieser Referenzobjekte kann ein Benutzer nun über die Links durch die Objektbank zu den verschiedenen Objekten navigieren. Um zu einem Objekt zu gelangen, benutzt man einen dahinführenden **Pfadnamen**. Ein Pfadname hat folgende Syntax:

```
Pfadname = Referenzobjektname [ '/'   relativer Pfadname  ]
                                  Linkname {'/' Linkname}
```

Dabei hat ein Linkname in H-PCTE folgende Gestalt: `Schlüsselattribut(e).Linktyp`

Beispiel für einen Pfadnamen:

```
~/UniDo.x/schmidt.ist_uni_angehoeriger/meier.betreut
```

9.4. Link-Kategorien

In den Entwicklungsdaten einer SEU treten häufig verschiedene Typen von Beziehungen auf. Aus diesem Grund können Links in H-PCTE vier verschiedene **semantische Eigenschaften** besitzen:

1. **Referentielle Integrität:**
   Bei einem Link mit referentieller Integrität wird dem Benutzer immer garantiert,

daß das Zielobjekt dieses Objekts existiert. Daraus folgt natürlich, daß alle Links mit referentieller Integrität bei Löschung des Zielobjekts mitgelöscht werden.

2. **Existenzeigenschaft:**
   Für die Existenz eines Objekts ist mindestens ein Link mit der Existenzeigenschaft erforderlich. D. h. ein Objekt kann nur zusammen mit einem Link mit Existenzeigenschaft angelegt werden. Außerdem wird ein Objekt automatisch implizit gelöscht, wenn der letzte Link mit Existenzeigenschaft auf dieses Objekt gelöscht wird.

3. **Komponenteneigenschaft:**
   Die Komponenteneigenschaft besagt, daß das Zielobjekt eines Links mit Komponenteneigenschaft als eine Komponente des Ausgangsobjekt zu behandeln ist. Links mit dieser Eigenschaft werden benötigt, um sogenannte komplexe Objekte (s. u.) zu bilden.

4. **Relevanz für das Ausgangsobjekt:**
   Wenn ein Link mit Relevanz für das Ausgangsobjekt angelegt oder gelöscht wird, so gilt das Ausgangsobjekt dieses Links als verändert im Sinne der Datumszeitstempel.

Die Links in H-PCTE können nun einen oder mehrere dieser semantischen Eigenschaften besitzen. Es sind aber nicht alle Kombinationen dieser Eigenschaften sinnvoll. Deshalb werden in H-PCTE fünf **Link-Kategorien** mit sinnvollen Kombinationen von Eigenschaften vorgegeben:

Kategorie des Linktyps	*zugehörige semantische Merkmale*			
	Komponente	Existenz	Refer. Int.	Relevanz
(c) composition	+	+	+	+
(e) existence	−	+	+	+
(r) reference	−	−	+	+
(i) implicit	−	−	+	−
(d) designation	−	−	−	+

Weiterhin sind nicht alle Kombinationen der Kategorien zwischen Link und Umkehrlink sinnvoll. Es ergeben sich folgende mögliche Kombinationen:

Kategorie des Linktyps	*zulässige Kategorien des Umkehrlinktyps*				
	composition	existence	reference	implicit	designation
composition	−	−	+	+	−
existence	−	−	+	+	−
reference	+	+	+	+	−
implicit	+	+	+	−	−
designation	−	−	−	−	−

Beispiel:
Die Kombination zweier composition-Links ist nicht sinnvoll und sogar untersagt. Würde man einen composition-Link von einem Objekt A zu einem anderen Objekt B fest-

legen und den zugehörigen Umkehrlink auch als composition-Link festsetzen, so hieße das folgendes: Objekt B ist Komponente von Objekt B, gleichzeitig ist Objekt A auch Komponente von Objekt A. Dieses ist nicht sinnvoll und man erhält einen verbotenen Zyklus.

9.5. Komplexe Objekte

In SEUs möchte man oft Objekte der Objektbank zu größeren Einheiten zusammenfassen, um sie als Ganzes weiterverarbeiten zu können. Dies ist in H-PCTE mit den im letzten Abschnitt beschriebenen composition-Links möglich. Diese Art von Links verbinden gerade mehrere zusammengehörende Objekte. Ein daraus entstehendes sogenanntes **komplexes Objekt** besteht aus

- einem **Ausgangsobjekt**
- allen über **composition-Links** erreichbaren anderen Objekte
- allen von diesen Objekten **ausgehenden Links**

Links, die von einem Teilobjekt des komplexen Objekts ausgehen können zu Objekten innerhalb oder außerhalb des komplexen Objekts führen. Dementsprechend unterscheidet man **interne** und **externe** Links.
Es ist möglich, daß Objekte zu mehreren komplexen Objekten gehören. Solche Objekte nennt man **gemeinsame Komponenten (shared objects)**.
Der Vorteil von komplexen Objekten ist der, daß Operationen wie Kopieren, Löschen, Verschieben, Sperren oder Rechtevergabe auf dem gesamten komplexen Objekt ausgeführt werden können und nicht auf jedem einzelnen Teilobjekt ausgeführt werden müssen.

9.6. Referenzen

Wie schon in den vorigen Kapiteln gesehen, wird zur Angabe eines speziellen Objekts ein Pfadname angegeben, der von einem Refrenzobjekt zu diesem Objekt führt. Dies ist nicht sehr effizient, wenn man ein Objekt mehrfach verwendet und an verschiedene Methoden übergibt, denn H-PCTE muß jedesmal einen solchen Pfad neu auswerten. Außerdem können die Pfadnamen durch eine komplexere Datenstruktur sehr lang werden und somit viel Tiparbeit für einen Benutzer erzeugen. Aus diesen Gründen gibt es in H-PCTE **Referenzen**. Eine Referenz in H-PCTE besteht aus

- einem **textuellen Bezeichener**, wobei es sich im Allgemeinen um den entsprechenden Pfad zum gewünschten Objekt handelt
- einem **Handle**, was sowas wie einen direkten Zugriff auf das Zielobjekt durch Auswertung des textuellen Bezeichners darstellt
- einem **Auswertungszeitpunkt**, der angibt, wann das Handle den textuellen Bezeichner auswerten soll. Hier unterscheidet man drei Zeitpunkte: Beim Zeitpunkt

`PCTE_NOW` wird der Bezeichner sofort beim Anlegen der Referenz ausgewertet. Bei `PCTE_FIRST_USE` wird der Bezeichner beim ersten Verwenden der Referenz ausgewertet und bei `PCTE_EVERY_USE` wird der Bezeichner bei jedem Verwenden der Referenz ausgewertet.

- einem **Auswertungszustand**, der angibt, ob der textuelle Bezeichner ausgewertet ist oder nicht.

Eine Referenz bezeichnet also immer das Objekt, das durch den textuellen Bezeichner zum Auswertungszeitpunkt beschrieben wird.

9.7. Segmentierung und Verteilung

Eine weitere Möglichkeit der Performance-Steigerung ist durch die Segmentierung und Verteilung gegeben. Die gesamte H-PCTE-Objektbank kann in sogenannte **Segmente** geteilt werden. Ein Segment ist also eine nichtleere Menge aller Objekte. Zu einem Segment gehören auch alle Links, die von diesen Objekten ausgehen. Der Sinn dieser Segementierung ist, daß die einzelnen Segmente **verteilt** werden können. Die Segmente werden aber nicht auf mehreren Datenträgern gespeichert, da eine H-PCTE-Objektbank nur auf einem Datenträger liegen darf. Vielmehr können sich einzelne Client-Prozesse einzelne Segmente in den Speicher laden, um dort mit einer wesentlich höheren Performance auf die darin enthaltenen Elemente zugreifen zu können. Lediglich das Segment 0 wird immer zentral im Serverspeicher geladen.
Ein SEU-Prozeß kann also mit hoher Performance auf alle lokal geladenen Segmente und mit geringer Performance auf alle zentral im Server geladenen Segemente zugreifen. Allerdings kann ein SEU-Prozeß **NICHT** auf lokal geladene Segmente anderer Prozesse zugreifen! Deshalb müssen alle Daten, auf denen Prozesse mehrerer Entwickler arbeiten immer zentral geladen werden.
Um die Performancesteigerung durch die Verteilung auch effektiv zu nutzen sollten beim Umgang mit Segmenten einige Regeln eingehalten werden:

1. Segmente sollten nicht zu groß werden (5-10 MB) (größere Segmente sollten sinnvoll geteilt werden)
2. Eine Applikation sollte auf nicht mehr als ca. 10 Segmenten arbeiten
3. Zentrale Daten, auf die viele Entwickler zugreifen, sollten immer zentral geladen werden
4. Private Daten sollten immer lokal geladen werden

9.8. Transaktionen

Bereits durch kleinste Operationen auf den Elementen der Objektbank können die gespeicherten Daten inkonsistent werden. Um bestimmte Ressourcen für andere Benutzer

zu Sperren und somit Sichten auf inkonsistente Daten zu vermeiden bietet H-PCTE die **Transaktionen** an. Verschiedene Prozesse konkurrieren also um die Ressourcen, weshalb man Transaktionen auch als einen **Concurrency-Control-Mechanismus** bezeichnet. Transaktionen bilden somit einen Rahmen um einzelne Operationen und führen einen konsistenten Zustand in einen anderen konsistenten Zustand über. Das folgende Bild gibt das allgemeine Schema einer Transaktion wider:

Innerhalb von Transaktionen können beliebig viele **Sicherungspunkte** angelegt werden. Zu diesen kann während der Transaktion immer zurückgesprungen werden, um alle nach dem Setzen des Sicherungspunktes gemachten Operationen rückgängig zu machen.
Außerdem können alle Transaktionen abgebrochen und dadurch der ursprüngliche Zustand wiederhergestellt werden.
In H-PCTE werden zwei **Sperrmodi** unterschieden:
Der Modus `TRANSACTIONED` bietet den klassischen Transaktionsschutz für die modifizierten Daten bezüglich fremder Prozesse. Fremde Prozesse dürfen auf die von einem anderen Prozeß gesperrten Daten nur lesend zugreifen. Dagegen kann es manchmal sinnvoll und erwünscht sein, daß auch andere Prozesse auf Daten innerhalb einer Transaktion zugreifen. Dazu bietet H-PCTE den Transaktionsmodus `UNPROTECTED` an.

9.9. Der Notifikationsmechanismus von H-PCTE

Wenn mehrere Prozesse auf den gleichen Daten arbeiten (z. B. in mehreren Fenstern) und die Daten in allen Prozessen änderbar sein sollen, so kann man die entsprechenden Daten nicht durch die Transaktion eines Prozesses sperren. Deshalb müssen bei Änderung der Daten durch einen Prozeß alle anderen betroffenen Prozesse benachrichtigt werden, damit diese ihre aktuelle Sicht auf die Daten aktualisieren können. Aus diesem Grund gibt es in H-PCTE den **Notifikationsmechanismus**. Ein Prozeß ist dadurch in der Lage, einen **Notifizierer** an einer Ressource der Objektbank anzubringen und sich von diesem über eventuelle Änderungen an dieser Ressource informieren zu lassen. Man unterscheidet 5 verschiedene **Ressourcen** und 10 verschiedene **Zugriffsereignisse** auf diesen Ressourcen:

Überwachbare Ressourcen	*mögliche Zugriffsereignisse*
ein bestimmtes Objekt	HPCTE_OBJECT_DELETE_EVENT HPCTE_OBJECT_MOVE_EVENT HPCTE_OBJECT_MODIFY_ACL_EVENT
Attribut eines Objekts	HPCTE_OBJECT_MODIFY_EVENT
Linkmenge eines Objekts	HPCTE_DELETE_*_LINK_EVENT HPCTE_DELETE_LINK_OF_TYPE_EVENT HPCTE_APPEND_*_LINK_EVENT HPCTE_APPEND_LINK_OF_TYPE_EVENT
ein bestimmter Link	HPCTE_LINK_DELETE_EVENT
Attribut eines Links	HPCTE_LINK_MODIFY_EVENT

Die Notifikation läuft in 4 Schritten ab:

1. Ein Prozeß, der über bestimmte Ereignisse an einer oder mehrerer Ressourcen benachrichtigt werden will, muß ein sogenannter **NotificationListener** sein.

2. Dieser NotificationListener legt dann einen **Notifizierer (Notifier)** für die zu überwachende Ressource an.

3. Der NotificationListener meldet sich beim Notifizierer als zu benachrichtigende Instanz an.

4. Im Falle des entsprechenden Zugriffsereignisses meldet der Notifizierer allen bei ihm angemeldeten NotificationListenern dieses Zugriffsereignis als **NotificationEvent**. Darin sind genaue Details über das Zugriffsereignis (z.B. ein neuer Attributwert) enthalten. Der NotificationListener muß nun dieses Event auswerten und den Anforderungen entsprechend darauf reagieren.

9.10. Zugriffskontrollen

H-PCTE kennt wie UNIX nur **diskretionäre Zugriffskontrollen**, d. h. nur der Besitzer (owner) einer Ressource kann die Rechte für diese vergeben. H-PCTE unterstützt desweiteren **gruppenorientierte Zugriffskontrollen** bei denen Mitgliedern verschiedener Gruppen Rechte an den Ressourcen eingeräumt werden können. Die Vergabe der Rechte ist aber im Vergleich zu UNIX wesentlich flexibler. Der Besitzer einer Ressource verwaltet zu dieser eine sogenannte **Access Control List (ACL)**, in die er beliebig viele Benutzer bzw. Arbeitsgruppen mit Rechten versehen kann. Es stehen einem Besitzer ca. 20 verschiedene Zugriffsrechte zur Verfügung. Die wichtigsten werden durch die Tabelle der Zugriffsereignisse aus dem vorherigen Kapitel abgedeckt. Zusätzlich gibt es natürlich Rechte, die das Lesen der Ressourcen erlauben.

9.11. Ausblick: Was kann H-PCTE noch?

Neben den oben beschriebenen Eigenschaften und Mechanismen gibt es bei H-PCTE noch weitere Features:

- Alle oben beschriebenen Operationen und Mechanismen werden durch Funktionen bzw. Methoden in einem **C- bzw. Java-API** realisiert.
- **Threads** werden unterstützt.
- Es gibt mit "ntt" und "poql" zwei **Abfragesprachen** für H-PCTE.
- H-PCTE enthält diverse **Administrationswerkzeuge**.
- "HBrowse" ist ein **Browser**, mit dem man durch die angelegte Objektbank navigieren kann.

10. Toolframe

Autor: *Sebastian Linz*

Zunächst möchte ich versuchen zu motivieren, daß das Thema Toolframe für unsere Projektarbeit von Bedeutung sein könnte. Darauffolgend stelle ich Toolframe vor. Anschliessend möchte ich das Konzept vorstellen, das Toolframe zugrundeliegt und diskutiere es in Hinblick auf unsere Projektarbeit.

10.1. Motivation

Es ist das Ziel unserer Arbeit, eine verteilte Hypermedia-Entwicklungs-Umgebung ($\mathcal{HEU}$) zu implementieren, die eine graphische Notation für DoDL integriert. Dafür müssen wir eine graphische Notation für DoDL erarbeiten und eine verteilte Hypermedia-Entwicklungs-Umgebung entwerfen und implementieren.

Toolframe ist eine Software-Entwicklungs-Umgebung, die Werkzeuge enthält, mit denen man Dokumente bearbeiten kann, die beim Software-Enwicklungs-Prozeß anfallende Aufgaben beschreiben. In den Dokumenten findet man Diagramme oder Modelle, die gewissen Regeln nachkommen müssen. Diese Regeln beschreiben die graphische Notation des entsprechenden Modells syntaktisch.

Man stellt fest, daß die grundlegende Problemstellungen von Toolframe und unserem Projekt $\mathcal{HEU}$ ähnlich sind. Toolframe und $\mathcal{HEU}$ realisieren eine graphische Entwicklungsoberfläche für Dokumente, denen eine graphische Notation zugrundeliegt.

Aufgrund dessen, daß man bei beiden Projekten eine ähnliche Problemstellung wiederfindet, ist es selbstverständlich, Toolframe und sein Konzept zu betrachten. Man kann Entwurfsideen übernehmen, programmiertechnische Lösungen abkupfern oder Konzepte wiederverwenden, wenn sie bei der eigenen Arbeit hilfreich sein können.

10.2. Toolframe

Wie einleitend erwähnt, ist Toolframe eine Software-Entwicklungs-Umgebung, die eine Menge von Werkzeugen zusammenfaßt. Mit diesen Werkzeugen lassen sich Dokumente bearbeiten, die beim Software-Entwicklungs-Prozeß anfallende Aufgaben erschlagen.

Der Entwurf von Entity-Relationship-Diagrammen, von Modellen der objektorientierten Analyse und Modellen des objektorientierten Entwurfs sind beispielhafte, typische Aufgaben des Software-Entwicklungs-Prozesses, für den Entwurfswerkzeuge zur Verfügung stehen. Diese Entwurfswerkzeuge sind Editoren, die die zugrundeliegende graphi-

sche Notation des entprechenden Dokumenttyps kennen. Mit ihnen können neue Modelle erzeugt oder bestehende Modelle visuell modifiziert werden, wobei die durchzuführenden Aktionen ständig auf Ausführbarkeit überprüft werden. Wie bei syntaxgesteuerten Editor ist auch diesen Editoren die grundlegende Syntax und graphische Notation bekannt, so daß Interaktionen des Benutzers auf syntaktische Korrektheit und Konsistenz überprüft werden können.

Toolframe ist jedoch nicht nur eine lose Sammlung von syntaxgesteuerten, graphischen Editoren, sondern es bietet die Möglichkeit, die bearbeiteten Dokumente zu verwalten. Dabei kann man für jeden Software-Entwicklungs-Prozeß ein Projekt anlegen. Jedes dieser Projekte enthält dann die Dokumente des zugehörigen Software-Entwicklungs-Prozesses.

Neben den Editoren umfaßt Toolframe weitere Werkzeuge. Es gibt Anfragenwerkzeuge, die Auskünfte über Daten der Objektbank liefern und Analysatoren, die Konsistenzbedingungen der Objektbank überprüfen können. An diesen Werkzeugen kann das wesentliche Konzept, das Toolframe zugrundeliegt, nicht beschrieben werden, so daß diese nicht näher beschrieben werden.

Toolframe bietet die Möglichkeit, Dokumente verteilt zu bearbeiten.

10.3. Das zugrundeliegende Konzept

Software-Entwicklungs-Umgebungen setzen sich aus mehreren Werkzeugen zusammen, die den Entwickler beim Software-Entwicklungs-Prozeß unterstützen sollen. Herkömmliche Verfahren der Entwicklung von Software-Entwicklungs-Umgebungen gestalten sich so, daß man die diversen Werkzeuge entwirft und implementiert, um diese anschliessend zu einer Einheit zu verschmelzen ([DK94]). Die Entwickler von Toolframe haben sich für die Entwicklung von graphischen Editoren in Software-Entwicklungs-Umgebungen ein neues Konzept ausgedacht. Sie haben erkannt, daß allen Dokumenten, die man mit den Werkzeugen der Software-Entwicklungs-Umgebung bearbeiten will, eine netzartige Struktur zugrundeliegt. Diese Erkenntnis wurde in den Entwurf umgesetzt, so daß man die Integration neuer Werkzeuge vereinfachen und den Implementierungsaufwand minimieren konnte.

Den Dokumenten, die mit Hilfe der Editoren der Software-Entwicklungs-Umgebung bearbeitet werden sollen, liegen netzartige Strukturen zugrunde. Diese Strukturen setzen sich aus Elementen zusammen, die über Verbindungen miteinander in Relation stehen. Erst bei der Darstellung eines einzelnen Dokuments gewinnen die Relationen und das Aussehen der Objekte des Dokuments an Bedeutung. Systemintern ist die Darstellung der Objekte nicht relevant, sondern nur an Bezeichnern und Attributen festgemacht. Aus diesem Blickwinkel erkennt man, daß man abstrahieren kann, indem man die Editoren nach Funktionalität in Komponenten aufteilt. Nach herkömmlichen Vorgehensweisen wird jedes Werkzeug bzw. jeder Editor zu einer Komponente gemacht und unabhängig von den anderen Werkzeugen entwickelt. Im Gegensatz dazu setzen sich Toolframe-Editoren aus dem allgemeinen, generischen Editor, einem speziellen Objektbankschema und den Spezialisierungen des entsprechenden Dokuments zusammen. Der generische Editor realisiert

die grundlegende Funktionalität der Editoren und den Zugriff auf die Objektbank. Ein Dokument mit seinen Elementen, Verbindungen und Attributen wird feingranular auf Objekte, Verbindungen und Attribute abgebildet, die die Objektbank verwalten kann.

Diese Vorgehensweise erleichtert es, die Werkzeuge zu einer Entwicklungsumgebung zusammenzuführen, wohingegen es sehr aufwändig, ist eine Menge heterogener Editoren zusammenzuführen. Das vorgestellte Konzept bietet jedoch noch weitere Vorteile. Dadurch, daß man einen generischen Editor programmiert, spart man Programmierarbeit. Denn die generische Komponente wird nur einmal implementiert und für alle Editoren verwendet. Würde man jeden Editor für sich und von Grund auf implementieren, so wird grundlegende Funktionalität mehrmals erzeugt. Dies bedeutet, daß man redundanten Quelltext erzeugt. Durch den Einsatz objektorientierter Methoden konnte die Effizienz bei der Entwicklung von Toolframe weiter gesteigert werden. Vererbung und Polymorphismus erlauben die Benutzung von Klassenbibliotheken, wodurch Funktionalität wiederverwendet werden kann, ohne diese selbst entwickeln zu müssen. Sowohl die verwendete Objektbank H-PCTE ([Kel98]) als auch das Benutzerschnittstellenmanagementsystem ET++ ([Wei92][Gam92]) bieten Klassenbibliotheken für ihre Benutzung. Diese mächtigen Systeme steigern die Produktivität, wohingegen sie viel Zeit bei der Einarbeitung verschlingen.

Mit der Verwendung von H-PCTE ([Kel98]) als Objektbank, ist es möglich, die oben angesprochenen, externen Datenbankschemata zu definieren. Mit Hilfe von externen Datenbankschemata kann jedem Dokumenttyp die eigene Sicht auf die Daten der Objektbank zugeordnet werden. Dies geschieht dadurch, daß man bei einem Schema Attribute neu hinzufügen und vorhandene Elemente verstecken oder umbenennen kann. Somit können alle Dokumente auf eine Objektbankstruktur abgebildet werden, weil dokumentspezifische Eigenschaften auf Attribute der Objekte abgebildet werden, die durch das spezifische Schema definiert sind.

Eine spezialisierende Komponente muß für jeden Editor implementiert werden. Da der generische Editor so konzipiert ist, daß er die grundlegende Funktionalität beinhaltet, müssen die spezialisierenden Komponenten nur noch die graphische Darstellung des Editors und des darzustellenden Dokuments konkretisieren. Die graphische Darstellung basiert auf der graphischen Notation des Dokumenttyps und ist für jeden Dokumenttyp unterschiedlich. Die Implementation dieser verschiedenen Spezialisierungen ist somit notwendig und erzeugt keinen redundanten Quelltext.

Jeder Editor setzt sich aus den drei vorgestellten Komponenten zusammen. Möchte man die Entwicklungsumgebung um weitere Dokumenttypen und die entsprechenden Editoren erweitern, so muß der Objektbank ein weiteres Schema zugefügt und das System um spezialisierende Komponenten erweitert werden.

10.4. Relevanz für die $\mathcal{HEU}$

Toolframe ist so konzipiert, daß es ohne wenig Aufwand um Werkzeuge erweitert werden kann. Dabei ist jedoch zu beachten, daß die Vorzüge des Konzepts nur zum Tragen kommen, wenn dem Dokumenttyp, der mit dem Werkzeug bearbeitet werden soll, eine

Graphenstruktur zugrundeliegt. Kann man vom vorliegenden Dokument nicht so weit abstrahieren, daß es auf die bestehende Objektbank abgebildet werden kann, so ist der Aufwand für die Integration des neuen Werkzeugs ebenso groß, als wenn man nach herkömmlichen Methoden vorgeht.

Es wird hervorgehoben, daß der Aufwand für die Implementation neuer Werkzeuge sehr gering ist. Im Vergleich zu herkömmlichen Vorgehensweisen soll ein neues Werkzeug durch 200 Zeilen Quelltext spezifiziert werden können, wohingegen der Aufwand bei herkömmlichen Methoden 10 bis 100 mal größer ist ([DK94]). Doch man darf den Aufwand für die Analyse und den Entwurf des neuen Werkzeugs und die Einarbeitung in die umfangreiche Klassenbibliothek nicht vernachlässigen.

Wenn bei $\mathcal{HEU}$ Dokumenten eine gemeinsame Struktur wiedergefunden werden kann, so sollte man das Konzept des generischen Editors übernehmen, um die Integration neuer Werkzeuge zu vereinfachen. Ich würde vorschlagen, daß man im Gegensatz zu Toolframe, den generischen Editor als Superklasse aller konkreten Editoren implementiert. Und für den Zugriff auf die Objektbank sollten abstrakte Datentypen realisiert werden, die Objekte der zu bearbeitenden Dokumente darstellen und sich auf die Objektbank abbilden. Die Entwickler von Toolframe haben diesen Ansatz bewußt vermieden, weil sie die Implementation redundanten Quelltextes verhindern wollten ([DK94]). Doch dadurch haben sie objektorientierte Konzepte übergangen und Vorteile aussen vor gelassen, die man durch Datenkapselung und Vererbung für sich gewonnen hätte.

11. Konzepte und Realisierung grafischer Editoren

Autor: *Christoph Begall*

Bei dem Projekt HEU ist die Erstellung grafischer Editoren ein zentrales Thema. Im Folgenden wird auf einige Aspekte solcher Editoren eingegangen, wobei besonderes Augenmerk auf eventuell auftretende Probleme gelegt wird. Zunächst wird allgemein in das Gebiet eingeführt, um klar herauszustellen, womit sich diese Arbeit beschäftigt und die grundlegenden Begriffe zu klären. Dann werden grafische Benutzungsoberflächen als grundlegende Technologielieferanten betrachtet. Schließlich wird zu den auftretenden Problemen übergegangen, wobei aus der großen Menge nur wenige, repräsentative ausgesucht wurden.

11.1. Einführung

Zuerst sollen einige grundlegende Fragen geklärt werden wie „Was ist ein grafischer Editor, was nicht?“ und „Wozu braucht man ihn überhaupt?“.

11.1.1. Motivation

Grafische Editoren sind ein Paradebeispiel für die Umsetzung der Ideen von grafischen Benutzungsoberflächen. Die einmal erlernten Methoden (z.B. zum Verschieben von Fenstern) können vom Benutzer wiederverwendet werden, um den Editor zu bedienen (z.B. Objekte im Editor verschieben). Ihre intuitive Bedienung ermöglicht auch technisch nicht versierten Benutzern, komplizierte Datenstrukturen zu bearbeiten. Sie werden nach [Hen88] z.B. gerne in der künstlichen Intelligenz eingesetzt, um Expertensysteme mit Wissen zu füllen: Die Experten haben oft andere Spezialgebiete als den Umgang mit Computern, so daß ihnen die Wissenseingabe möglichst leicht gemacht bzw. der Lernaufwand für sie niedrig gehalten werden soll.

Die visuelle Manipulation von Strukturen führt dazu, daß grafische Benutzungsoberflächen nicht nur zum Öffnen mehrerer Terminal-Fenster dienen. Stattdessen können *alle* Möglichkeiten der zugrundeliegenden grafischen Oberfläche voll genutzt werden. Dies kann z.B. dazu führen, daß Programme weniger Benutzerfehler ermöglichen bzw. robuster werden.

Viele Diagrammformen lassen sich gut durch solche Editoren bearbeiten: allgemeine Graphen, Flußdiagramme, Entscheidungsbäume, UML. Oft ist deren Bearbeitung mit

der *Papiermethode* gar nicht mehr möglich, bzw. der Aufwand nicht vertretbar.

11.1.2. Definitionen

Wenn im Folgenden einige Begriffe wiederholt auftauchen, so muß klar sein, was in diesem Kontext damit gemeint ist.

Editor Ein Editor ist ein Werkzeug zur Eingabe, Veränderung und Darstellung von Daten.

Syntaxgesteuerter Editor Ein Editor, der nur Daten akzeptiert, die einer bestimmten Grammatik entsprechen, und deren Eingabe besonders unterstützt.

Grafischer Editor Ein syntaxgesteuerter Editor zur Manipulation von Daten mit *grafischen* Mitteln.

Bei einem Editor unterliegen die Daten also keinen festen Regeln; Beispiele sind Emacs, hexedit etc. . Sie werden für vielfältige Zwecke benutzt, z.B. für Programmierung, Dokumentation oder zur Bearbeitung von Konfigurationsdateien.

Mit dem Verzicht auf Freiheit zugunsten implizit *richtiger* Daten (Programme, Dokumente ...) kommt man zu syntaxgesteuerten Editoren: Texte werden durch den Editor möglichst nur von einem in den anderen syntaktisch korrekten Zustand transformiert. Näheres hierzu findet man in [RT89].

Schränkt man die Möglichkeiten zur Transformation der Daten auf grafische Mittel ein, so erhält man grafische Editoren. Dabei ist natürlich zu beachten, daß einem grafischen Editor immer eine geeignte grafische Notation zugrunde liegen muß, die natürlich auch wieder gewissen Regeln unterliegt. Die Konsistenz der Daten innerhalb dieser Regeln entspricht weitgehend der syntaktischen Korrektheit in syntaxgesteuerten Editoren. Umgekehrt ist aber nicht jede Art von strukturierten Daten gut grafisch darstellbar oder manipulierbar.

Die erwähnten grafischen Mittel sind in diesem Fall weniger die Bedienung mit der Maus, sondern das sogenannte *visuelle Feedback*. Dahinter verbirgt sich einerseits eine sinnvolle Sicht auf die zugrundeliegenden Daten, andererseits eine optische Rückmeldung an den Benutzer, wenn Aktionen auf den Daten durchgeführt werden oder nicht durchgeführt werden können.

11.1.3. Verwandte Gebiete

Betrachtet man grafische Editoren auf einer etwas abstrakteren Ebene, so fallen die Verbindungen zu anderen Gebieten auf (Abbildung 11.1 auf der nächsten Seite). Einerseits sind grafische Editoren eine Art von Verfeinerung syntaxgesteuerter Editoren (siehe Abschnitt 11.1.2). Zu diesen besteht wohl die stärkste „Verwandtschaft“. Daraus ergibt sich natürlich auch eine Verbindung zu Themen des Compilerbaus, auf die aber in Abschnitt 11.3.1 auf Seite 82 noch näher eingegangen wird. Andererseits sind bei grafischen Editoren (im Gegensatz zu syntaxgesteuerten) Algorithmen zur Darstellung der Daten notwendig. Fast jeder grafische Editor setzt auf einer grafischen Benutzungsoberfläche auf und benutzt deren Bedienschemata (siehe Abschnitt 11.2.1 auf der nächsten Seite).

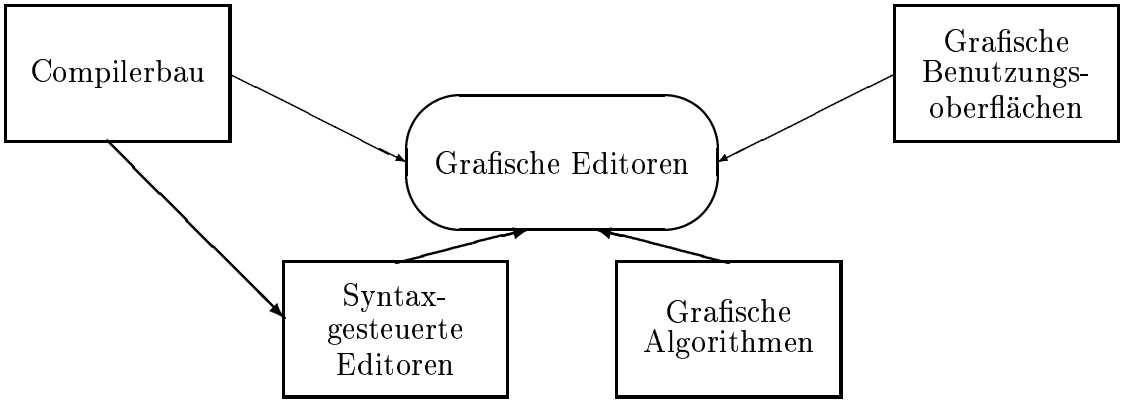


Abbildung 11.1.: Das Umfeld grafischer Editoren.

11.2. Bearbeitung der Daten

Wir betrachten im weiteren grafische Benutzungsoberflächen als Grundlage, auf der wir aufbauen. Außerdem stellen wir uns die Frage, wie man die dort gegebenen Möglichkeiten für visuelle Datenmanipulation benutzen kann.

11.2.1. Grafische Benutzungsoberflächen

Das wohl wichtigste für die Erstellung guter grafischer Editoren ist *die konsequente Umsetzung des zugrundeliegenden Bedienschemas*. Programme, die auf dem Macintosh laufen, müssen sich auch so *anfühlen*, wie Programme, die auf dem Macintosh laufen. Es gibt nun verschiedene Ansätze und Möglichkeiten, gute Benutzungsoberflächen zu erstellen. Hier werden ein paar dieser Ansätze aufgelistet, die Ideen sind Abstraktionen nach [Ebe94]:

- Experimentieren: Der Endbenutzer ist dabei die Testperson. Meist stehen keine guten oder nicht genügend Testpersonen zur Verfügung. Außerdem ist das Testen, ob die entworfene Benutzungsoberfläche intutiv bedienbar ist, sehr aufwendig. Es gibt aber gute Quellen [SIG]: z.B. Hat sich die SIGCHI der ACM schon mit vielen Problemen des *Human-Computer-Interface* auseinandergesetzt.

- Metaphern finden: Meist wurden die Daten, die der grafische Editor bearbeiten soll, schon vorher (computerlos) benutzt. Solche Umsetzungen alter (computerloser) Techniken sind aber nicht immer gut übertragbar. Außerdem sind alte Techniken nicht immer gut.

- Logisch denken: „Was erwartet der Benutzer als nächstes?“ ist die Frage, die man sich beim Erstellen grafischer Benutzungsoberflächen stellen sollte (siehe [Shn87]). Dieser Ansatz beruht auf der Idee von der kleinstmöglichen Überraschung. Letztendlich ist er aber wieder stark abhängig von der Modellbildung des Denkenden.

Verschiedene Techniken zur Visualisierung bieten sich in grafischen Editoren an, besonders weil sie bereits bekannt und damit leichter zu erlernen sind. Konzepte wie `Objekt`, `Verbindung`, `Behälter`, `Attribut` sind in fast allen grafischen Editoren zu finden, auch wenn sie im Einzelnen andere Namen tragen. Auch die Ikonifizierung kann zur Qualität eines grafischen Editors beitragen, denn einprägsame Bilder haben einen mnemonischen Effekt.

Ein anderes Thema, das man immer wieder antrifft, ist Modus-basiertes arbeiten. Das Programm `xfig` arbeitet nach diesem Schema, näheres findet man auch in [Ebe94]: z.B. Verschieben, Zusammenfassen, Verbinden als Modi. Dies ist ähnlich wie in Malprogrammen. Drücken-Ziehen-Loslassen oder Mehrfach-Klicken (Selektieren) und Befehl-Anwählen sind oft genutzte Mechanismen zur Argumentauswahl für die jeweiligen Aktionen. Ersteres hat zur Folge, daß man Rahmen oder Verbindung **ziehen** kann, während man sonst die beteiligten Objekte einzeln auswählt. Schwierigkeiten gibt es dann bei der Auswahl mehrerer entfernter Objekte, da hier eventuell kein einfacher bzw. rechteckiger Rahmen möglich ist.

11.2.2. Manipulation der Daten

Wie kann man nun die Aktionen umsetzen und worauf muß man achten? Zunächst einmal dürfen nur syntaktisch korrekte Datenstrukturen erzeugt werden. Ein Objekt darf vielleicht nur zu einem bzw. in einen Behälter gehören. Dann darf der Editor auch nichts anderes zulassen. Außerdem soll der Benutzer ein visuelles Feedback bekommen, z.B. die Veränderung des Mauszeigers. Wenn man etwas an einer bestimmten Stelle nicht fallen lassen darf, wird der Mauszeiger zu einem Halteverbot-Zeichen oder einem Totenkopf, wenn man etwas anfassen darf, wird er zur Hand. Und wieder kommt man auf die Metaphern zurück.

Eine andere sinnvolle Einschränkung der Aktionsmöglichkeiten des Benutzers ist ein kontextsensitiver Befehlsvorrat. Nur sinnvolle/erlaubte Befehle werden überhaupt angeboten. Kontext-Menüs sind ja auch von anderen Programmen bekannt. Wenn man dennoch in die Situation kommt, einen nicht korrekten Zustand dulden zu müssen, so sollte dies angezeigt werden, z.B. rot und/oder blinkend. Fehlende Informationen sind deutlich hervorzuheben.

11.3. Probleme bei der Realisierung

Bei all diesen Anforderungen gibt es natürlich Schwierigkeiten, besonders bei der Wahrung der *strukurellen Integrität*. Damit ist die Korrektheit bezüglich der zugrundeliegenden Regeln gemeint. In diesem Abschnitt wollen wir die Probleme ein wenig näher betrachten und auf mögliche Lösungsansätze eingehen.

11.3.1. Sichten und Schichten

Aus den schon beschriebenen Aspekten ergeben sich sehr verschiedene Anforderungen an die benutzten Datenstrukturen. Beim Entwurf grafischer Editoren ergeben sich einige

Muster zur Lösung auftretender Probleme:

- Die Datenstrukturen, die andere Werkzeuge als der grafische Editor benutzen, müssen nicht unbedingt für die Arbeit in dem Editor sinnvoll sein. Die zu bearbeitenden Daten liegen vielleicht auch schon in einem definierten Format vor.
- Fehler und Inkonsistenzen müssen schnell und einfach erkannt werden; das erfordert eine andere Sicht der Daten als z.B. deren Abspeicherung. Die interne Darstellung kann große Mengen von redundanten Zusatzinformationen enthalten, um schnelle Bearbeitung der Eingaben des Benutzers zu gewährleisten. Diese Information soll vielleicht für andere gar nicht sichtbar sein.
- Eventuell gibt es mehrere Visualisierungen derselben Daten. Diese müssen von den benutzten Datenstrukturen unterstützt werden.

Daher unterscheidet man zumindest zwei Sichten auf die Daten: *Intern* und *Extern.*

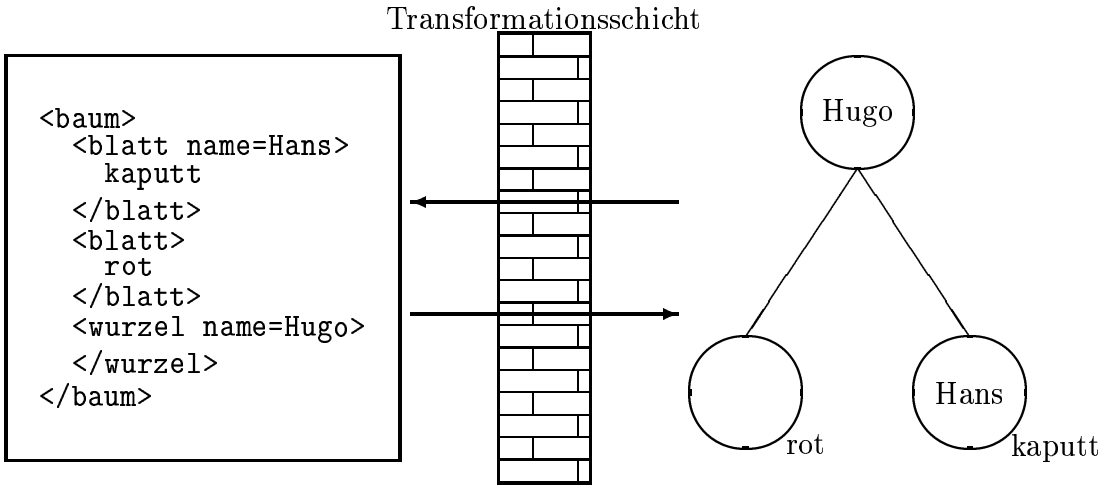


Abbildung 11.2.: Abbildung zwischen interner und externer Sicht.

Nun ergeben sich Probleme bei der Transformation der Daten. Zusätzliche Informationen sollen/dürfen nicht verloren gehen. Solche Zusatzattribute können z.B. die grafischen Koordinaten einzelner Objekte sein oder Informationen zur schnellen Überprüfung der strukturellen Integrität. Die Abbildung zwischen den beiden Sichten muß also **eindeutig** sein. Außerdem muß diese *Umwandlungsschicht* davon ausgehen, daß externe Daten fehlerhaft sein können: Hier können Techniken aus dem Compilerbau zum Einsatz kommen, z.B. um Fehler zu erkennen und entsprechende Meldungen vernünftig an den Benutzer weiterzugeben.

11.3.2. Darstellung

Wie im letzten Abschnitt schon erläutert, kann es zu einem Verlust von Informationen kommen. Unter anderem kann der Benutzer eventuell die genaue Position einzelner gra-

fischer Objekte festlegen. Wird diese Information nun durch die Umwandlungsschicht zerstört, so ist dies ein für den Benutzer nicht durchsichtiges Verhalten.

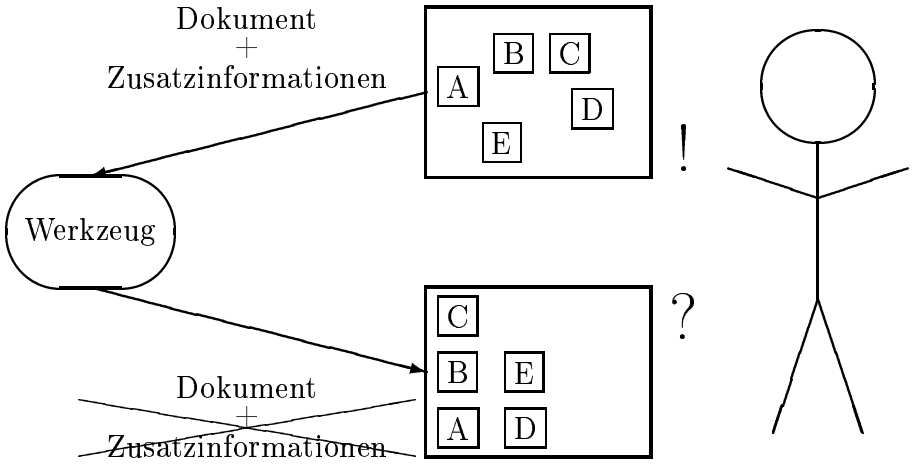


Abbildung 11.3.: (Nicht-)Schutz grafischer Zusatzinformationen.

Es ergeben sich aber noch viele andere Probleme bei der Darstellung. Das Anordnen der Einzelteile der grafischen Notation in der Fläche ist ein nicht zu unterschätzendes Problem. Es gibt sehr gute Algorithmen aus dem Bereich *Grafische Systeme*. Rechteckige und einheitlich große Objekte erleichtern diese Aufgabe oft. Die Beachtung von Benutzerwünschen kann aber mitunter das Problem sehr verschärfen. Eine einfache Lösung (und oft die einzige) ist, dem Benutzer nur wenige oder gar keine Möglichkeit zu geben, die Darstellung der Daten zu beeinflußen. Oft wird das Problem auch leichter, wenn alle Objekte an einem Raster ausgerichtet werden.

11.3.3. Strukturelles Editieren

Ein anderes, bereits erwähntes Problem ist die Konsistenz der Struktur der bearbeiteten Daten. Um beim Verändern der Daten deren Struktur zu erhalten, werden verschiedene Strategien verfolgt:
Das Einfügen von Schablonen, einerseits zusammengesetzter Objekte, die in sich strukturell konsistent sind, andererseits Rahmen mit Platzhaltern, ist eine solche. Beispielsweise können Behälter, die nicht leer sein dürfen, in Form einer Schablone mit einem *Dummy* als Inhalt erstellt werden. Eine andere Strategie ist die strukturbedingte Einschränkung der Aktionsmöglichkeiten des Benutzers. So lassen sich vielleicht Objekte nur zusammen mit allen dazugehörenden Objekten auswählen. Ein Behälter kann also nur mit seinem Inhalt ausgewählt werden, so daß bei einem Verschieben- oder Löschen-Befehl das Gesamtbild konsistent bleibt.

Bei Schablonen mit Platzhaltern ergeben sich andere Probleme, da wir einen temporär inkonsistenten Zustand zulassen. Aus Performanz-Gründen kann man eine Liste mit nicht ausgefüllten Platzhaltern führen, die z.B. beim Speichern auf Leerheit geprüft wird. Sollen inkonsistente Zustände (nicht ersetzte Platzhalter) auch beim Speichern erlaubt

sein, so führt dies zu besonderen Anforderungen beim Einlesen und Schreiben der Daten (eventuell wird ein Parser notwendig).

11.3.4. Sofortige Überprüfung

Nach jeder Aktion wird sofort die Konsistenz der Daten sichergestellt. Um die Antwortzeiten des Systems klein zu halten, soll die Überprüfung möglichst *lokal* sein. Dazu werden an den Knoten bzw. Objekten für die interne Sicht zusätzliche Attribute gespeichert, die helfen, die Zulässigkeit möglichst schnell festzustellen. Am angenehmsten sind natürlich Überprüfungen, deren Antwortzeiten so kurz sind, daß man eine Aktion simulieren, prüfen und bewerten kann, bevor sie beendet ist.
Beispiel: Beim Loslassen des rechten Mausknopfes, der ein Icon gezogen hat, wird festgestellt, daß die Aktion unzulässig ist, und anstatt das Icon zu verlieren, wird der Mauszeiger zu diesem Icon.

11.4. Fazit

Schon beim Entwurf einer grafischen Notation sind viele Dinge zu ihrer *grafischen Editierbarkeit* zu beachten. Die entstehenden Probleme sind keineswegs trivial, meist gibt es jedoch bereits gute Lösungen aus den Bereichen Compilerbau, Grafische Algorithmen und UI-Design. An einigen Stellen ist es besser, sich auf Erfahrungen anderer zu verlassen, anstatt eigene Experimente zu starten, gerade im Bereich GUI-Design. Denn besonders in diesem Bereich ist es nicht sinnvoll, ständig das Rad neu zu erfinden, da so der Wiedererkennungseffekt ausbleibt. Einfache, aber nicht so komfortable Lösungen sind mitunter *gute* Zwischenlösungen, sofern man plant, sie später zu verwerfen.

Die Komplexität der Materie macht eine gute Kommunikation zwischen den beteiligten Entwicklern notwendig. Bestes Beispiel ist das Transformationsproblem, bei dem sich die zuständigen Programmierer frühzeitig auf geeignete Datenstrukturen einigen müssen.

12. Visuelle Programmierung

Autor: *Andreas Schröder*

Das Problem ist, daß Software meist "unverständlich" ist. Gemeint ist damit, daß man nur die Ergebnisse des Softwareprogrammes sieht, nicht aber das Zusammenwirken der Elemente, die zu diesem Ergebnis führen.

Daraus gewinnt man die Erkenntis, daß der Mensch Anschauungsmaterial braucht, damit ihm ein "Licht aufgeht" . Das bedeutet, daß der Mensch bildliche Ausdrücke besonders effizient verarbeitet, da er konkrete Objekte besser erfaßt als abstrakte Beschreibungen.

Damit ergibt sich das Ziel, Programme so zu gestalten, daß sie durch "Zusammenstecken und Ausprobieren" entstehen. Damit soll eine Erhöhung der Verständlichkeit von Programmen und die Erleichterung der Programmierung gewährleistet werden.

12.1. Begriffe

Um das Thema "visuelle Programmierung" vorzustellen, bedarf es einer Einführung einiger Begriffe zu diesem Thema. Dazu wird der Begriff "visuell", dann "visuelle Sprache" näher betrachtet, um dann zum Begriff "visuelle Programmiersprache" zu kommen um dann schließlich den Begriff "visuelle Programmierung" darzulegen.
Der Begriff "visuelle Programierung" besteht aus "visuell" und "Programmierung". Zur Programmierung wird eine "Programmiersprache" benötigt (hier eine visuelle). Sind beide Teilbegriffe erläutert, so kann man den Begriff "visuelle Programmierung" leichter zugänglich machen.

12.1.1. visuell

visuell ist die Eigenschaft eines Objektes (Gegenstand, d.h. alles Abstraktes und Konkretes), mit dessen Hilfe Information (d.h. Wichtiges für das Erreichen eines Handlungsziels) gewonnen werden kann durch das visuelle Wahrnehmungssystem des Menschen! [SCH7]

Beispiel für den Begriff visuell:

Das Beispiel soll kurz darstellen, daß für sehende und blinde Personen der "Programmcode" verschiedene Bedeutungen haben kann, da er visuell (durch Fettschrift) unterschiedliche Bedeutung enthält.

Schlüsselwörter in Fettschrift!

If while **then** do **else** end	Eindeutig!	1. Bedeutung für sehende Person
If **while** then **do** else **end**	Eindeutig!	2. Bedeutung für sehende Person
If while then do else end	Mehrdeutig!	Einer blinden Person vorgelesen

12.1.2. visuelle Sprache

"visuelle Sprache" ist eine formale Sprache mit visueller Syntax und/oder visueller Semantik und dynamischer oder statischer Zeichengebung

Zunächst einmal soll der Begriff *formale Sprache* betrachtet werden. Aufbauend auf diesem Begriff wird dann *statische/dynamische Zeichengebung* erläutert, um dann schließlich zu *visuell syntaktischen/semantischen Konstrukten* zu kommen.

- Eine *formale Sprache*: besteht aus Grundzeichen (dem Alphabet) und einem Regelwerk zur Bildung von Ausdrücken aus Grundzeichen (sog. **Syntax**). Desweiteren besteht sie aus Interpretationsregeln zur Ermittlung der Bedeutung von Ausdrücken (**Semantik**).
- *statische Zeichengebung*: Senden und Empfangen der Nachricht geschehen zeitlich getrennt (Bsp.: STOP-Schild (bleibend!))
- *dynamische Zeichengebung*: Senden und Empfangen geschehen gleichzeitig (Bsp.: Ampel (einmalig))
- *visuelles syntaktisches Konstrukt*: Bsp.: Ein Pfeil zwischen 2 Elementen einer Grafik
- *visuelles semantisches Konstrukt*: Bsp.: Färbung dieses Pfeiles, die den Zustand dieser Beziehung ausgedrückt, etwa Grün für aktiv und Grau für inaktiv.

Darstellen soll dies, daß eine visuelle Sprache eine formale Sprache ist, die aber zusätzlich in der Lage ist, syntaktische und semantische Bedeutungen visuell hervorzuheben bzw. hinzuzufügen. Sie beinhaltet zusätzlich auch noch statische und dynamische Zeichengebung.

12.1.3. visuelle Programmiersprache

"visuelle Programmiersprache" ist eine visuelle Sprache (s.o.) zur vollständigen Beschreibung der Eigenschaft von Software. Sie ist entweder eine *Universal- programmier- oder Spezialprogrammiersprache.*

- *visuelle Universalprogrammiersprache*: Damit kann jedes Berechnungsverfahren formuliert werden, das auf einer Turingmaschine ausführbar ist. (auf TM sind alle berechenbaren Probleme lösbar)
- *visuelle Spezialprogrammiersprache*: Nicht berechnungsuniversell, da wesentliche Sprachkonstrukte fehlen. (z.B.: Programmierung mit Makrorekordern)

Mit dem folgenden Beispiel soll der Unterschied zwischen verbaler und visueller Programmiersprache gezeigt werden. Dies ist ein ganz entscheidender Unterschied bei der Betrachtung des Begriffes “visuelle Programmiersprache”.

1. VP-Systeme mit visueller (grafischer) Programmiersprache:
   - Programmcode enthält einen hohen Anteil grafischer Elemente.
   - Text ist dagegen nur wenig vorhangen!
   - Bsp.: LabVIEW (National Instruments)

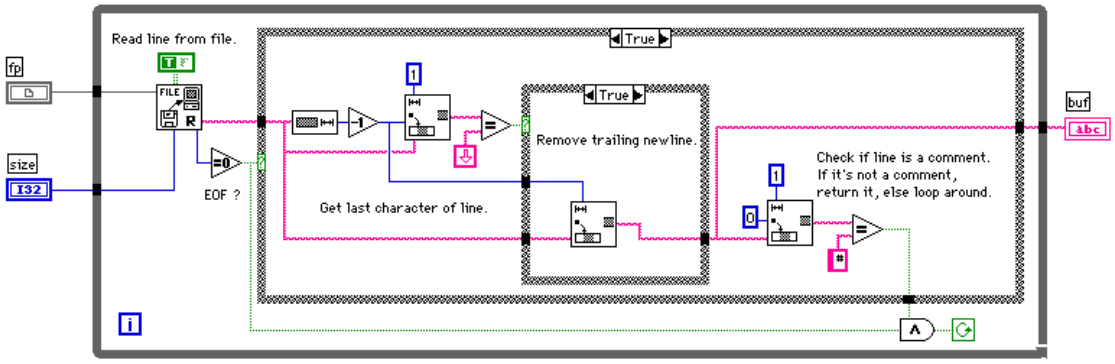


Abbildung 12.1.: LabVIEW Beispiel

2. VP-Systeme mit verbaler (textueller) Programmiersprache
   - Grafische Elemente primär beim Bau der Benutzungsschnittstelle.
   - Programmlogik wird textuell programmiert.
   - Bsp.: Visual C++ (Microsoft)

```
char *getsline (buf, size, fp)
char *buf;
unsigned int size;
FILE *fp;
{
register int len;
while (fgets(buf, (int) size, fp) != NULL)
{
/* Remove trailing newline. */
len = strlen (buf);
if (len > 0 AND buf[len - 1] == '/n')
buf[len - 1] = '/0';
if (buf[0] != '#')
return (buf);
```

```
}
return (NULL)
}
```

12.1.4. visuelle Programmierung

Die visuelle Programmierung ist nun die Erstellung von Software mit einer visuellen Programmiersprache.

VP-Systeme	Menge intergrierter Werkzeuge
VP: visuelle Programmiersysteme	zur visuellen Programmierung

12.2. Pro und Contra

Um die Vor- und Nachteile von visueller Programmierung zu verstehen, möchte ich zunächst einmal einen kurzen Einblick in die Kognitionspsychologie geben, wo genau ergründet wurde, warum Bilder effizienter verarbeitet werden als Text. Bei der Aufzählung der Vor- und Nachteile beziehen ich mich auf alle in den Literaturangaben (7.Kapitel) gemachten Einträgen. Im vierten Abschnitt "fünf Thesen zu VP" geht es darum Pro und Contra anhand von fünf Thesen in Frage zu stellen. In den Thesen wird diskutiert ob visuelle Programme die Kommunikation erleichtern (1.These), ob visuelle Programme leicht verständlich sind (2.These), ob visuelle Programmierung leicht erlernbar ist (3.These), ob visuelle Programmierung Sprachbarrieren überwindet (4.These) und ob visuelle Programmierung optimale Kognition bedeutet.

12.2.1. Kognitionspsychologie

Abbildung 12.2.: Kognitionspsychologie

12.2.2. Pro

- Bilder sind mächtige Kommunikationsmittel
- Diagramme und andere visuelle Repräsentationen unterstützen das Verstehen und die Ideenübermittlung
- Piktogramme als international verständliche Sprachkonstrukte
- Menschen bevorzugen im allgemeinen Bilder gegenüber Worten

- Im Bezug auf die Kommunikation sind Bilder mächtiger als Worte, da sie eher ihrer Bedeutung, mit wenig Ausdrücken übermitteln können
- Bilder haben nicht die Sprachbarriere, die übliche Sprachen haben
- Die sinkenden Kosten von grafisch zusammenhängender Hardware und Software haben es möglich gemacht, Bilder als ein Mittel zur Kommunikation zu verwenden
- Das menschliche Wahrnehmungssystem und das menschliche Informationsverarbeitungssystem sind deutlich optimiert für vielschichtig dimensionierte Daten
- Visueller Ausdruck der Programmierung kann von Menschen leicher verstanden und erstellt werden
- Menschen fällt es leicher, mit exakten Beispielen zu handeln als mit abstrakten Ideen

12.2.3. Contra

- Grafische Wiedergabe eines Softwaresystems können keinen Gesamteindruck des Systems vermitteln
- Es gibt keine befriedigende grafische Notation für Rekursionen und Selbstbezüglichkeiten
- Text ist hinsichtlich des Platzbedarfs optimiert, was für Bilder nicht gilt
- Einsatz rein visueller Programmiersprache stößt bei der Konstruktion großer Systeme auf unüberwindliche Hindernisse (z.B.: schlechte Skalierbarkeit, mangelnde Ausdruckskraft, umständliche Benutzung)
- Schwäche von Bildern bei der Beweisführung von bildlichen Gleichheiten (Abbildung 3.2)

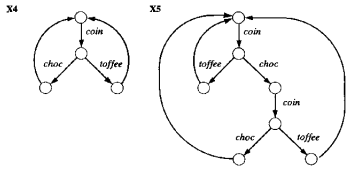


Abbildung 12.3.: zwei äquivalente Zustandsdiagramme eines Süßigkeitsautomaten

12.2.4. fünf Thesen zu VP

1. These: Visuelle Programme erleichtern Kommunikation

Wie in der Abbildung 3.3 deutlich erkennen kann, haben die vier Beobachter alle unterschiedliche Eindrücke (im Bezug auf die Verständnis). Daher ist eine effektive Kommunikation mit bildlichen Illustrationen nur dann möglich, wenn verbale Erläuterungen eventuelle Zweideutigkeiten beseitigen und sich die grafische Repräsentation auf das Wesentliche beschränkt.

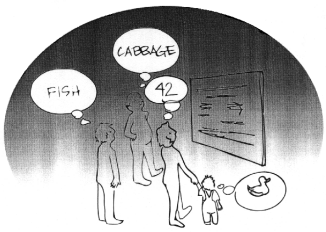


Abbildung 12.4.: Dasselbe Bild, vier Betrachter, vier Interpretationen

2. These: Visuelle Programme sind leicht verständlich

Auch hier ist in der Abbildung 3.4 deutlich zu erkennen, warum visuelle Programme nicht unbedingt leichter verständlich sein müssen (können). Die Darstellung von Programmcode in grafischer Form ist eher verständnishemmend als verständnisfördernd. Dies gilt insbesondere für Diagramme, wo Beziehungen zwischen einzelnen Elementen durch Linien dargestellt sind. In solchen Diagrammen führen komplexere Strukturen unausweichlich zu unentwirrbaren Liniengeflechten.

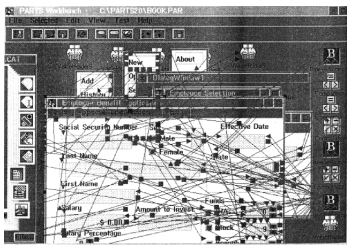


Abbildung 12.5.: ein einfaches PARTS-Programm

3. These Visuelle Programmierung ist leicht erlernbar

Bei diesem (Abbildung 3.5) und sicher einigen anderen Beispiel tritt schnell eine Art Frustration hervor, wenn geeignete Sprachkonzepte zur Bewältigung komplexer Problemstellungen fehlen oder umständliche Interaktionsmechanismen das schnelle Editieren größerer Programme verhindern.

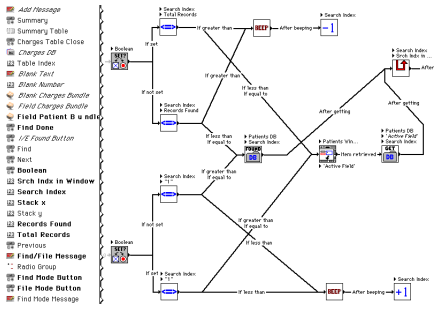


Abbildung 12.6.: Ausschnitt aus einer Serius-Anwendung

4. These Visuelle Programmierung überwindet Sprachbarrieren

Das Problem ist, den gesamten Begriffsraum der Programmierung in Bildern zu fassen. Dies stellt sich als ein "unlösbares" Problem dar. Es würde ein Bilder-Esparanto entstehen mit unüberwindlichen Kommunikationsbarrieren, die jede Verständigung zum Erliegen brächten (Abbildung 3.6) .

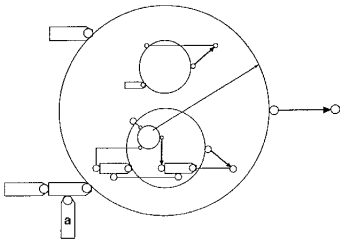


Abbildung 12.7.: Diagramm aus Pictorial Janus

5. These Visuelle Programmierung heißt optimale Kognition

Zu den Dingen, die ein Programmierer im Kopf behalten muß, gehören u.a. der Aufbau und die Funktionalität des Programms, typische Programm- und Datenstrukturen, Entwurfs- und Codierstrategien, Syntax und Semantik von Programmiersprachen, die Bedienung der Werkzeuge sowie Datei- und Verzeichnisnamen. Nur weniges aus diesen Bereichen läßt sich durch einprägsame Bilder darstellen (siehe Abbildung 3.7) . Die Wirksamkeit von visuellem Programmcode ist hinsichtlich des Erinnerungsvermögens als gering einzuschätzen.

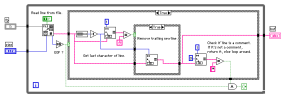


Abbildung 12.8.: LabVIEW-Programm

12.3. Konzepte visueller Programmiersysteme

VP-Systeme können anhand der Klassifikation von Burnett und Baker [Bur3] in drei Gruppen zusammengefaßt werden:

1. VP-Systeme basierend auf verbalen Sprachen:

Umfaßt VP-Systeme, deren Programmiersprachen eng an die Konzepte verbaler Sprachen angelehnt sind !	steuerflußorientiert, funktionsorientiert, datenfluß-, objekt-, constraintorientiert

2. VP-Systeme basierend auf visuellen Sprachen:

VP-Systeme, deren visuelle Sprachen kein Gegenstück in der verbalen Programmierung haben !	regelorientiert, beispielorientiert, formularorientiert

3. VP-Systeme für spezielle Bereiche:

VP-Systeme, die spezielle Ansätze für die parallele Programmierung realisieren oder unterschiedliche Sprachkonzepte vereinen	Parallelitätsorientiert, multiparadigmenorientiert

Die hier gewählte Einteilung nach Spracheigenschaften ist aus zwei Gründen zielführend:

- Es gibt viele VP-Systeme, die keiner prinzipiellen Einschränkung hinsichtlich ihrer Anwendungsgebiete unterliegen.
- Forschung und Industrie entwickeln laufend neue Ideen zur Unterstützung der Softwareentwicklung mit visuellen Elementen.

Visuelle Programmiersprachen sind fast immer auf ein bestimmtes VP-System abgestimmt und mit diesem eng verflochten . Die klassische Trennung zwischen Sprache und Werkzeug ist deshalb nicht möglich, und eine Beschreibung von Konzepten der visuellen Programmierung muß sinnvollerweise auf VP-Systeme Bezug nehmen.
Bei der nun folgenden Beschreibung verschiedener VP-Systeme beziehe ich mich zum größten Teil auf [Sch7], wobei ich aber auch aus [Gli6] Systembeschreibungen genommen habe, die zwar sinngemäß gleich sind, aber textuell leichter verstädlich erklärt wurden.

12.3.1. Steuerflußorientierte VP-System

Befehlsaktivierung wird durch Programmsteuerung ausgelöst !!! (beruht auf dem imperativen Programmierparadigma)
Festlegung des Steuerflusses erfolgt auf 3 Arten:

1. *Anweisungssequenzen*:
   Folge von Operationen, die in der Reihenfolge der Aufschreibung ausgeführt werden (mit Ausnahem von Sprüngen, Schleifen, ...)

2. *Komponentennetze*:
   Komponentennetze funktionieren nach folgendem Prinzip:

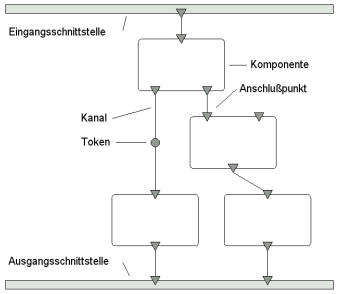


Abbildung 12.9.: Schematische Darstellung eines Komponentennetzes

Erreicht ein Token eine Komponente, so wird diese aktiviert. Daraus folgt, daß der Verarbeitungsprozeß beginnt, der zur Folge eine Emission weitere Token hat (sequentiell oder parallel).
Dies hat dann zur Folge, daß mehrere Token gleichzeitig im Komponentennetz vorhanden sind.
Ein *Programm* in steuerflußorientierten VP-Systemen besteht meist aus mehreren Komponentennetzen, die entweder für Reaktionen auf externe Ereignisse zuständig sind oder innerhalb anderer Komponentennetze wie Unterprogramme verwendet werden.
Der Steuerfluß eines Programms ergibt sich aus dem Fluß der Tokens durch diese Netze !!!
Um ein bestimmtes Berechnungsverfahren durchzuführen, definiert man, welche Programmkomponenten auf welche Ereignisse reagieren sollen, woraus ein bestimmtes Programmverhalten realisiert wird !!!

3. Transitionsnetze:
   Transitionsnetze beschreiben Automaten (Zustandsdiagramme) und Petrinetze (gerichtete Graphen). Sie werden für reaktive Systeme benutzt, die kontinuierlich auf Ereignisse reagieren und deren Verhalten zeitlichen Bedingungen unterworfen ist.
   Anwendungsgebiet: Bereich der Systemtechnik und Simulation.

Transitionsnetze ermöglichen die exakte Spezifikation von Systemeigenschaften, erlauben beweisbare Aussagen über das Systemverhalten und sind ein zweckmäßiges Mittel, um den Ablauf und das Zusammenwirken von Prozessen zu definieren.

12.3.2. Datenflußorientierte VP-Systeme

Hier bestimmt die Verfügbarkeit von Daten (d.h. der Fluß der Daten durch das System) die Ausführungsreihenfolge von Operationen.(und nicht der Wert des Befehlszählers s.o.)
Das Datenflußmodell kann nicht nur für die Implementierung von Software, sondern auch zur Systemanalyse und zum Systementwurf dienen.

1. *Datenflußprogramm-Darstellung*:
   Ist ein gerichteter Graph, mit Datenquellen, Datensenken und Operationen als Knoten und Datenkanälen als Kanten.
   $\longrightarrow$ sog. Datenflußdiagramm
   Bsp.:

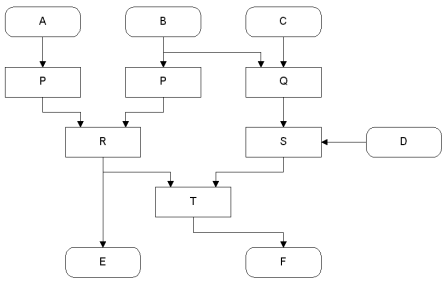


Abbildung 12.10.: Schematische Darstellung eines Datenflußprogramms

2. *Datenflußprogramm-Ausführung*:
   Daten sind aktive Einheiten. Sie bestimmen, welche Operationen wann zur Ausführung gelangen.
   Eine Operation wird dann ausgeführt, wenn alle Eingabedaten (stammend von Datenquellen oder von anderen Operationen) zur Verfügung stehen.
   Für die Berechnung der Ausgabedaten konsumiert die Operation alle Eingabedaten.
   Eine *Programm* wird nun so erstellt, daß der Programmierer nur dieDatenabhängigkeiten zwischen den Operationen definiert, statt die Ausführungsreihenfolge der Operationen festzulegen.
   Operationen, die Daten produzieren sollen, müssen dann vor Operationen ausgeführt werden, die diese Daten konsumieren.

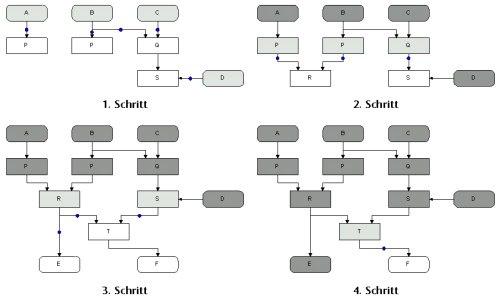


Abbildung 12.11.: Ausführung eines Datenflußprogramms

Da grundsätzlich alle Operationen parallel ausgeführt werden können, zwischen denen keine Datenabhängigkeit existiert (d.h. zwischen denen weder direkt noch indirekt ein Datenaustausch stattfindet), gibt es in einigen VP-Systemen spezielle Sprachkonstrukte, die eine sequentielle Ausführung erzwingen!

- *Keine Nebeneffekte/Variablen*:
  Das reine Datenflußmodell ist frei von Nebeneffekten.
  Der Nachteil der sich aus Variablen ergibt, ist in vielen Fällen die Unterbindung der Nutzung globaler Systemressourcen (z.B. für Interprozeßkommunikation).
  Deshalb is es nicht notwendig, die Daten in Variablen abzulegen, weil die Daten konzeptionell auf den Datenkanälen gespeichert werden (durch das Weiterleiten).
  Das Fehlen von Variablen wird daher allgemein als großer Vorteil datenflußorientierter Systeme angesehen, da der Programmierer nicht darüber nachdenken muß, welche Auswirkungen die Zuweisung an eine Variable an jenen Programmstellen hat, wo diese Variable verwendet wird.

- *Strukturierte Daten*:
  Elementare Daten sind prinzipiell nicht modifizierbar, daher kann Nebeneffektfreiheit garantiert werden!
  Das Problem bei strukturierten Daten ist aber, daß eine Modifikation jedoch einen Nebeneffekt bei allen Operationen auslöst, die auf dasselbe Datum zugreifen.
  Die Lösungmöglichkeit die sich stellt, ist die, das Laufzeitsystem vor Ausführung einer Operation, die strukturierte Daten modifiziert, zuerst zu kopieren. Dies halt allerdings Latenzzeiten und Speicherplatzprobleme zur Folge.
  Nun gibt es aber leider auch Fälle bei manchen "Variablenarten", wo die Nebeneffektfeiheit nicht garantiert werden kann (z.B.: globale Systemressourcen, wie Dateien).

Wie man sieht, ist also abhängig vom Problem die Garantie der Nebeneffektheit möglich oder auch nicht.

- *Spezialkonstrukte für Ablaufstrukturen*:
  Es werden spezielle Operationnen, die Einführung von Zyklen im Datenflußdiagramm, die Erweiterung der Schaltregeln und das Konzept der Vorbelegung von Datenkanälen mit bestimmten Werten erforderlich.
  Die Folge daraus ist Unübersichtlichkeit, die wiederum zur Folge hat, daß besondere Strukturierungsmechanismen notwendig sind (siehe LabVIEW).

12.3.3. Funktionsorientierte VP-Systeme

Problem: Zuweisungsoperationen sind die Ursache struktureller Defizite von imperativen Programmen
Alternative: FP-Systeme (funktionale Programmiersysteme)
Ein FP-System besteht aus einer Menge von Objekten, einer Menge von Funktionen, die Objekte auf Objekte abbilden, einer Menge von funktionalen Formen zur Kombination von Funktionen oder Objekten, um daraus neue Funktionen zu erzeugen, und einer Menge von Definitionen, die Symbole an Funktionen binden.
Variablen und Wertzuweisungen gibt es nicht!!!
Ein funktionales Programm kann visuell als Funktionsdiagramm dargestellt werden:
Rein funktionale Sprachen leiden an denselben Schwächen wie reine Datenflußsprachen:

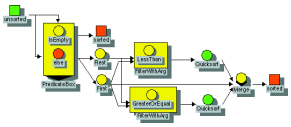


Abbildung 12.12.: Funktionsdiagramm in VisaVis

Sie sind ineffizient und für die Programmierung interaktiver Systeme unbrauchbar.
Die Stärken funktionaler Sprachen liegen vor allem im algorithmischen Bereich, der jedoch durch konventionelle Funktionsbibliotheken ebenfalls gut abgedeckt ist.
Daher ist die Verwendung von FP-Systemen eher selten.

12.3.4. Objektorientierte VP-Systeme

Software wird meist auf zwei eng verbundenen Ebenen erstellt:

1. Entwicklung von Programmbausteinen mit verbalen Programmiersprachen und Klassenbibliotheken.
2. Entwicklung von Applikationen durch Verwendung der verbal erstellten Bausteine und durch die visuelle Verbindung zu komplexeren Einheiten.

Der Term *Programmbaustein* stellt eine spezielle Form eines Objektes dar. Seine Struktur und Teile des Verhaltens sind in einer Klasse beschrieben. Ein Baustein verfügt über eine Schnittstelle zur Kommunikation mit anderen Bausteinen, die Attribute, Ereignisse und Nachrichten umfaßt.
Der Term *Verbindung* meint die gerichteten Daten- und Kommunikationskanäle.
Das objektorientierte Denkmodell leistet durch seine methodischen und technischen Konzepte einen "bedeutenden" Beitrag zur Produktion von Qualitätssoftware.
Es verbessert insbesondere die Erweiterbarkeit, Wiederverwendbarkeit und Wartbarkeit von Software.
Die problemgerechte Art der Modellierung, die imaginäre Sachverhalte oder Ausschnitte der realen Welt in Form von kooperierenden Objekten erfaßt, erlaubt eine bessere Umsetzung der Aufgabenstellung in eine softwaretechnische Realisierung, als dies etwa mit funktionalen Ansätzen möglich ist.
Die bei OOP gegebene enge Verbindung von Problem und Lösung legt eine visuelle Repräsentation von Objekten nahe. Geeignete grafische Darstellungen und visuelle Interaktionsmechanismen können die Problemmodellierung unterstützen und die Softwarekonstruktion erleichtern, indem sie eine metaphorische Korrespondenz mit der realen oder imaginären Problemwelt herstellen.
Nachteile:

- Tendenz zur Unübersichtlichkeit
- Keine Methoden und keine Vererbung
- Probleme mit Objekterzeugung und Polymorphismus

12.3.5. Constraintorientierte VP-Systeme

Hier besteht ein *Programm* aus einer Menge von Objekten (Werten, Variablen) und eine Menge von Beziehungen zwischen den Objekten, die ständig gelten müssen. Diese Beziehungen nennt man Constraints. Ein automatischer Mechanismus, der Constraint-Satisfier, sorgt dafür, daß die Constraints bestehen bleiben, sobald sich Objektattribute ändern.
Constraints in ThingLab:
($f = c * 1,8 + 32$ und $c = (f - 32)/1,8$)
Constraints entsprechen den Relationen und Regeln der logischen Programmierung.

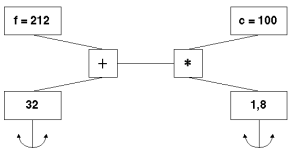


Abbildung 12.13.: Constraints in ThingLab

Man definiert nur, was gemacht werden soll, aber nicht, wie es gemacht werden soll.
$\longrightarrow$ sog. Deklarative Programmierung
Durch dies Art des Vorgehens ergeben sich nun einige Probleme, für die es aber auch Lösungen gibt:
Problem: Unter- und Überspezifikationen, die zu wenige bzw. zu viele Constraints umfassen.
Lösung: Constraint-Hierarchien mit absteigender Priorität von einer Ebene zur nächsten.
Problem: Erfassung zeitlicher Bedingungen.
Lösung: Erweiterung um temporale Constraints.
Deklarative Programmierung mit Constraints stößt jedoch schnell an Grenzen (z.B. Probleme nur zu spezifizieren, aber nicht algorithmisch zu lösen) .

12.3.6. Regelorientierte VP-Systeme

Programme werden als eine Folge von Bildtransformationen beschrieben.
Ein regelorientiertes Programm besteht dann aus einer Menge von grafischen Transformationsregeln mit optionalen Bedingungen und Aktionen. Eingabe für das Programm ist eine umzuformende Grafik. Eine Transformationsregel besteht aus einer linken und rechten Grafik. Links wird ein Bildausschnitt vor der Transformation dargestellt (Muster), rechts der umgeformte Ausschnitt (Resultat). Bei der Ausführung eines Programms versucht das VP-System durch Mustervergleich (unter Berücksichtigung der Bedingungen), eine der linken Regelseiten in der umzuformenden Grafik zu finden. Gefundene Ausschnitte werden durch die Darstellung der rechte Seite ersetzt, wobei eventuell zusätzliche Aktionen ausgeführt werden. Der Mustervergleich wird ständig wiederholt und kontinuierlich visualisiert, wodurch gewissermaßen ein Film abläuft. Das Programm endet, wenn das System keinen Ausschnitt mehr findet, der zu einer Regel paßt, oder der Benutzer das Programm abbricht.
Bsp.:

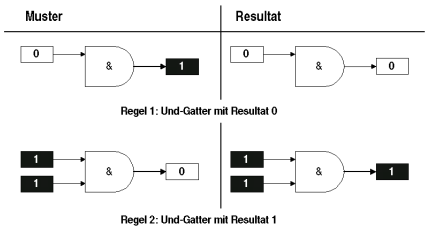


Abbildung 12.14.: Transformationsregeln

12.3.7. Beispielorientierte VP-Systeme

Beruhen auf dem Gedanken, daß in manchen Fällen die beispielhafte Druchführung einer Aufgabe am Computer genügt, um daraus ein Programm abzuleiten.
Grundidee:
Der Benutzer führt dem System in einzelnen Schritten vor, wie eine Aufgabe zu lösen ist, das System beobachtet den Benutzer, zieht Schlußfolgerungen und generiert ein Programm, das diese Aufgabe wiederholt ausführen kann.

1. *Programmierung mit Beispielen*:
   Das System zeichnet die einzelnen Benutzeraktionen auf und wiederholt sie später. Das VP-System versucht nicht, die Absicht des Benutzers zu erkennen, sondern dient im wesentlichen als Übersetzer, der Aktionen am Bildschirm in internen Programmcode überführt.

2. *Programmierung durch Beispiele*:
   Der Benutzer demonstriert anhand typischer Szenarien, wie aus gegebenen Daten das gewünschte Resultat zu erzeugen ist. Das VP-System versucht die Aktionen des Benutzers zu verallgemeinern und leitet aus den gezeigten Beispielen ein Programm ab, das nicht nur auf die vorgezeigten Fälle, sondern auch auf ähnliche Situationen anwendbar ist.
   Der Nachteil der sich dabei ergibt, ist die Diskrepanz zwischen der Interaktion des Benutzers und dem vom System generierten Pogramm.

12.3.8. Formularorientierte VP-Systeme

Diese beruhen auf dem Konzept von Tabellenkalkulation.
Der große Erfolg von Tabellenkalkulationsprogrammen erweckte die Hoffnung, daß formularbasierte Ansätze auch die Programmierung erleichtern könnten. Diese Erwartungen haben sich nicht erfüllt.[Sch7]
In formularorientierten VP-Systemen werden Programme durch Ausfüllen von Programmblättern (Formularen bzw. Tabellen) erstellt. Zellen in den Programmblättern enthalten entweder Daten oder Formeln. Daten können skalar oder strukturiert sein. Die Programmierung geschieht deklarativ. Der Programmierer definiert keine Algorithmen, sondern gibt in den Formeln an, welche Daten zur Berechnung anderer Daten benötigt werden und mit welchen Operationen die Daten zu verknüpfen sind. Ein eingebauter Mechanismus sorgt für die automatische Nachberechnung von Daten, die von anderen Daten abhängig sind. Variablen werden nicht benötigt.
Der Unterschied solcher Systeme zu herkömmlichen Tabellenkalkulationsprogrammen liegt primär darin, daß keine Unterscheidung zwischen konstanten und variablen Zellen notwendig ist. Jede Zelle kann Werte liefern und Werte berechnen.
Formularorientierte VP-Systeme haben sich vor allem für Datenbanksysteme bewährt.

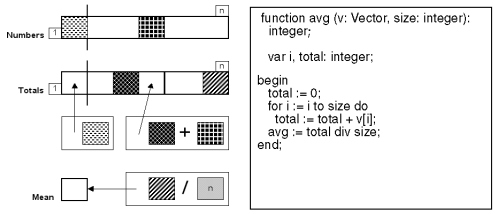


Abbildung 12.15.: Formularorientierte Programmierung

12.3.9. Multiparadigmenorientierte VP-Systeme

Fassen mehrere der zuvor genannten Programmierparadigmen zusammen.
Der Einsatzbereich multiparadigmenorientierter VP-Systeme ergibt sich aus den Anwendungsgebieten der unterstützten Paradigmen und dem Bedarf an einer engen Integration von unterschiedlichen Konstruktionsmethoden auf der Entwurfs- und Implementierungsebene.
Nachteil: Tendenz zu konzeptioneller Überlagerung.

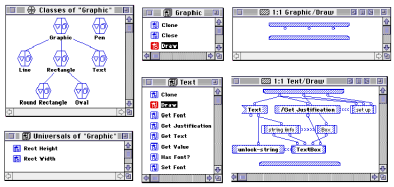


Abbildung 12.16.: Multiparadigmenorientierte Programmierung

12.4. Beispiele für visuelle Programmiersysteme

12.4.1. C^2

C^2 (nach [Gli8])ist ein steuerflußorientiertes VP-System auf Basis von Anweisungssequenzen, das den Versuch zeigt, eine verbale Programmiersprache in eine visuelle Programmiersprache zu transformieren, ohne an den Sprachkonzepten etwas zu ändern. Wie in "vielen" Berichten (siehe Literaturangaben) gezeigt wird, bringt eine solche Transformation keine Vorteile.

12.4.2. SERIUS

SERIUS (von Joe Firmage, Referenz [Sch7])ist ein steuerflußorientiertes VP-System auf Basis von Komponentennetzen, das visuelle Programmierung auf einer mit 4GL-

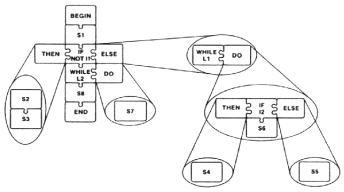


Abbildung 12.17.: Schematische Darstellung eines Programms in der Proc-BLOX-Notation

Werkzeugen vergleichbaren Ebene erlaubt. Der Anwender soll damit in der Lage sein, maßgeschneiderte Applikationen für seinen Aufgabenbereich schnell zu entwickeln.

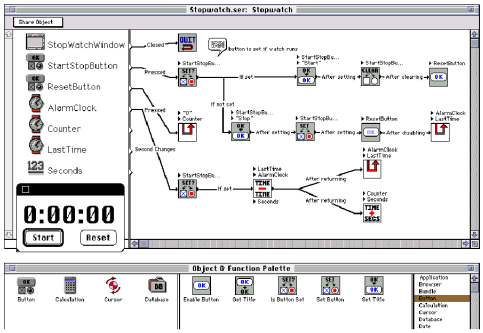


Abbildung 12.18.: Das VP-System Serius

12.4.3. LabVIEW

LabVIEW (Jamal und Pichlik [JUP9]) ist ein datenflußorientiertes VP-System, das für die Programmierung virtueller Instrumente eingesetzt wird. Die zugrundeliegende Metapher ist der Entwurf elektronischer Schaltkreise. Das System ist kommerziell erfolgreich und wurde bereits in vielen Anwendungsbereichen eingesetzt.

12.4.4. VisaVis

VisaVis (an der Universität Dortmund (BRD) von Poswig [POS10]) ist ein funktionsorientiertes VP-System aus dem akademischen Bereich, das als Musterbeispiel für die gelungene Abbildung eines schwierigen Formalismus in eine visuelle Programmiersprache gesehen werden kann. VisaVis zeigt besonders gut, welche Möglichkeiten und Grenzen sich für die Programmierung mit Funktionsdiagrammen ergeben.

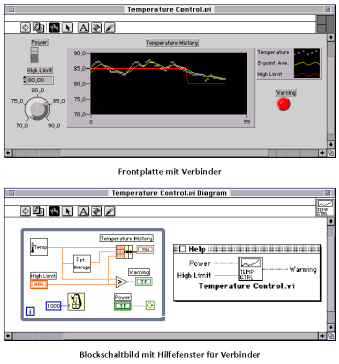


Abbildung 12.19.: Frontplatte und Blockschaltbild eines LabVIEW-Proramms zur Temperaturmessung

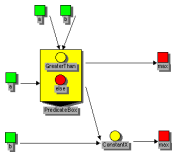


Abbildung 12.20.: Maximumberechnung mit Verzweigugn in VisaVis

12.4.5. PARTS

PARTS (Referenz [Sch7]) ist ein objektorientiertes VP-System für die Programmierung mit Softwarekomponenten. Es war das erste VP-System, das eine objektorientierte Programmiersprache (Smalltalk) und einen Benutzungsschnittstelleneditor auf visueller Ebene nahtlos integrierte.

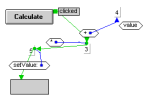


Abbildung 12.21.: Resultat- und Argumentverbindungen in PARTS

12.5. Fazit

Visuelle Programmierung stellt für die Entwicklung allgemeiner Softwaresysteme eher eine Sackgasse dar. Und zwar aufgrund der bei komplexen Problemen entstehenden Unübersichtlichkeit der visuellen Darstellungen und einiger weiterer Probleme (s.o.)

Visuelle Programmierung offenbart seine Stärke für spezielle Anwendungsgebiete (d.h. überschaubare und abgegrenzte Problemstellung), da hier die Komplexität viel geringer ist als bei großen Softwarentwicklungen.

13. Einführung in die computerunterstützte Gruppenarbeit

Autor: *Wilhelm Leibel*

Ob Ziele in Organisationen erreicht werden, hängt in vielen Fällen von kooperativer Aufgabenerfüllung ab. Die computerunterstützte Gruppenarbeit (engl.: Computer Supported Cooperative Work - CSCW) unterstützt diese Kooperation durch Überbrückung räumlicher und zeitlicher Distanzen. CSCW ist zudem ein Hilfsmittel für das Management arbeitsteiliger Prozesse. Das primäre Ziel ist dabei immer die Erhöhung der Effektivität und Effizienz betrieblicher Abläufe und damit die Steigerung der Leistungsfähigkeit der gesamten Organisation. CSCW gewinnt in zunehmendem Maße an Bedeutung, da in vielen Fällen die Hardware-Infrastruktur für die Unterstützung der Gruppenarbeit bereits vorhanden ist.
Im folgenden Kapitel wird zunächst in die Grundlagen der computerunterstützten Gruppenarbeit eingeführt. Das dritte Kapitel beschäftigt sich anschließend mit den Informations- und Kommunikationstechnologien. Im vierten Kapitel werden dann die Anwendungen der computerunterstützten Gruppenarbeit vorgestellt, und das letzte Kapitel gibt eine Zusammenfassung der relevanten Fakten wieder.

13.1. Grundlagen

Um ein Verständnis für die computerunterstützte Gruppenarbeit zu bekommen, werden zunächst einige wichtige Begriffe eingeführt. Ausgangspunkt der folgenden Darstellungen ist der Begriff der Gruppe. Darauf aufbauend wird der Begriff der Gruppenarbeit erläutert. Anschließend wird CSCW definiert, um danach Kriterien herzuleiten, nach denen CSCW-Applikationen klassifiziert werden können.

13.1.1. Definitionen

13.1.1.1. Gruppe

Es gibt viele unterschiedliche Definitionen des Begriffes Gruppe. Es können grundsätzlich Gruppen, Arbeitsgruppen und Teams unterschieden werden. In Anlehnung an Zimbardo wird der Begriff Gruppe folgendermaßen definiert (vgl.: [TSMB95]):

Gruppe Von einer Gruppe spricht man, wenn zwei oder mehrere Personen interagieren und dabei eine gegenseitige Beeinflussung stattfindet.

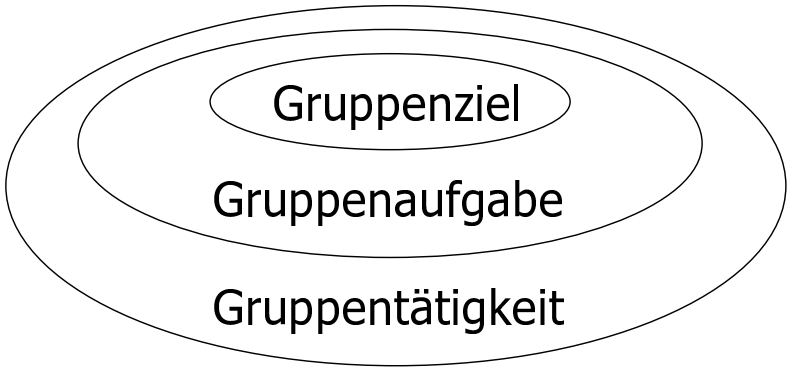


Abbildung 13.1.: Elemente der Gruppenarbeit

Arbeitsgruppe Eine Arbeitsgruppe ist eine Gruppe mit einer gemeinsamen Aufgabe.

Team Ein Team ist eine Arbeitsgruppe, deren Mitglieder den Willen haben, ein gemeinsames Ziel zu erreichen.

13.1.1.2. Gruppenarbeit

Innerhalb von Teams findet Gruppenarbeit statt. Sie dient dem Zweck der Zielerreichung. Dazu müssen zielbezogene Aufgaben erfüllt werden. Dies geschieht, indem aufgabenbezogene Tätigkeiten ausgeführt werden. Die Elemente der Gruppenarbeit sind also Gruppenziele, Gruppenaufgabe und Gruppentätigkeit (vgl. Abbildung 13.1). Gruppenarbeit kann demnach folgendermaßen definiert werden:

Gruppenarbeit ist die Summe aller aufgabenbezogenen Tätigkeiten, die von Gruppenmitgliedern ausgeführt werden, um zielbezogene Aufgaben zu erfüllen und somit Gruppenziele zu erreichen. Die Elemente der Gruppenarbeit sind Gruppenziele, Gruppenaufgabe und Gruppentätigkeit.

13.1.1.3. Gruppenprozesse

Um einzelne Gruppentätigkeiten aufgabenbezogen und damit zielorientiert ausführen zu können, sind bestimmte Gruppenprozesse notwendig. Gruppenprozesse können in drei Kategorien differenziert werden (vgl. Abbildung 13.2 auf der nächsten Seite).

Kommunikation ist die Verständigung mehrerer Personen untereinander.

Koordination bezeichnet jene Kommunikation, die zur Abstimmung aufgabenbezogener Tätigkeiten, die im Rahmen von Gruppenarbeit ausgeführt werden, notwendig ist.

Kooperation bezeichnet jene Kommunikation, die zur Koordination und Vereinbarung gemeinsamer Ziele notwendig ist.

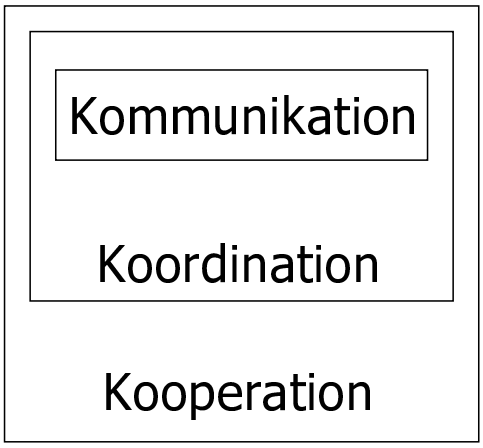


Abbildung 13.2.: Gruppenprozesse

An dieser Stelle muß darauf hingewiesen werden, daß die bisher zur Definition der Elemente der Gruppenarbeit und zur Definition der Gruppenprozesse notwendigerweise gemachten Abstraktionen, dem komplexen Phänomen Gruppenarbeit natürlich nicht vollumfänglich gerecht werden.

13.1.1.4. CSCW

Computer Supported Cooperative Work (CSCW) gilt als junges, aber bereits etabliertes Forschungsgebiet innerhalb der Informatik. Eine allgemein anerkannte Definition von CSCW existiert aber bisher noch nicht. Greenberg schreibt:

> *CSCW is not particularly well defined, a consequence of its youth and its multi-disciplinary nature.*

Die meisten Autoren definieren CSCW in einer informellen Art und Weise. Einig ist man sich darüber, daß CSCW als Forschungsgebiet verstanden wird, welches auf interdisziplinärer Basis untersucht, wie Personen in Arbeitsgruppen oder Teams zusammenarbeiten und wie sie dabei durch Informations- und Kommunikationstechnologie unterstützt werden können. Ziel aller Bemühungen ist es, unter Verwendung aller zur Verfügung stehenden Mittel der Informations- und Kommunikationstechnologie, Gruppenprozesse zu unterstützen und dabei die Effektivität und Effizienz der Gruppenarbeit zu erhöhen.
CSCW ist ein ausgeprägt interdisziplinäres Forschungsgebiet. Demzufolge muß es immer im Zusammenhang mit den entsprechenden relevanten Forschungsgebieten betrachtet werden (vgl. Abbildung 13.3 auf der nächsten Seite).

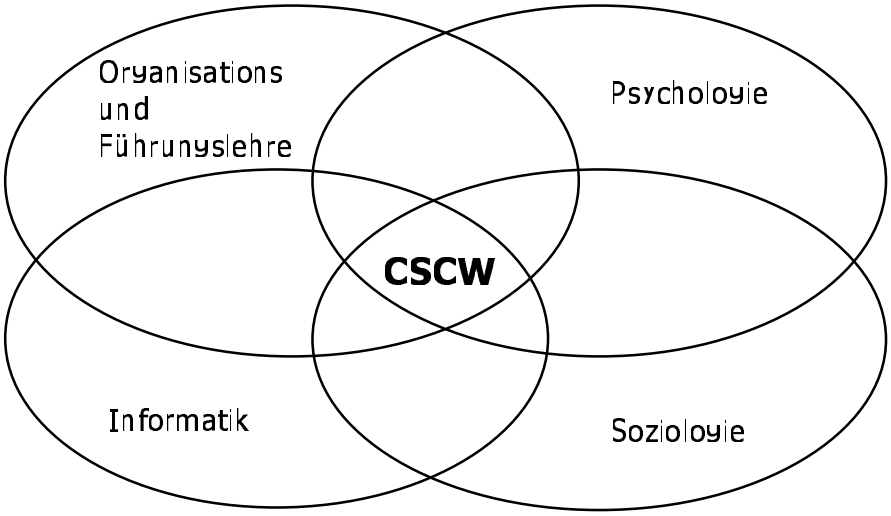


Abbildung 13.3.: Interdisziplinarität von CSCW

13.1.1.5. Groupware

Software, die im Rahmen von CSCW entsteht, wird als Groupware bzw. CSCW-Applikation verstanden[1]. Ursprünglich verstanden die Schöpfer dieses Begriffes darunter nicht nur die Software, sondern auch die betroffenen sozialen Prozesse zusammen als ein soziotechnisches System. Heute wird der Begriff Groupware folgendermaßen definiert (vgl.: [Teu95]):

Groupware bzw. CSCW-Applikationen sind aus Software und evtl. spezifischer Hardware bestehende Systeme, durch die Gruppenarbeit unterstützt oder ermöglicht wird.

13.1.2. Klassifikation von CSCW

13.1.2.1. Ausgewählte Klassifikationskriterien

CSCW-Applikationen lassen sich nach verschiedenen Kriterien klassifizieren:

- *Die verwendeten Medientypen* Medientypen sind Seh- und Gehörsinn ansprechende Typen wie Text, Grafik, Bild und Audio- bzw. Videosequenzen.
- *Die örtliche Verteilung* Dieses Kriterium gliedert sich in räumlich benachbarte (lokale) und räumlich entfernte (verteilte) Anwendungen.
- *Die zeitliche Verteilung* Hierbei unterscheidet man zwischen zeitgleicher (synchroner) und zeitverschiedener (asynchroner) Kommunikation.

[1] Groupware ist vor allem im kommerzielen Bereich zu einem Schlagwort avanciert. Viele angebotene Produkte sind herkömmliche Mehrbenutzer-Applikationen, obwohl sie als Groupware bezeichnet werden.

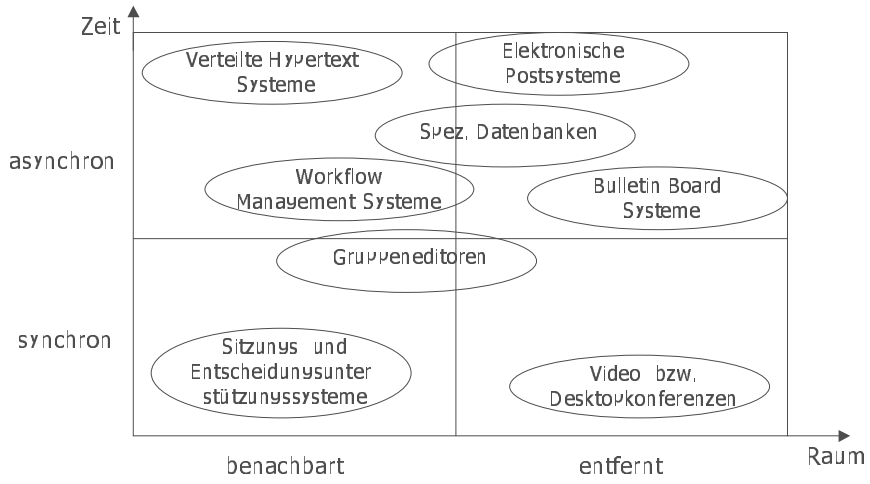


Abbildung 13.4.: Raum-Zeit-Matrix

- *Die Anzahl der Kommunikationspartner* Dieses Kriterium bestimmt im wesentlichen die zum Einsatz kommenden Technologien. Es können Kommunikationssituationen mit einem, zwei oder mehreren Teilnehmern unterschieden werden. Diese erfordern jeweils spezifische Unterstützungstechniken.

- *Die Art der Kommunikation* Hier unterscheidet man zwischen expliziter und impliziter Kommunikation. Explizite Kommunikation liegt dann vor, wenn die Kommunikationsteilnehmer aktiv Informationen austauschen, beispielsweise in einem Gespräch von Angesicht zu Angesicht (engl.: face to face) oder über elektronische Post. Dagegen ist die Kommunikation implizit, wenn sie über das Medium gemeinsamer Informationsobjekte stattfindet, z.B. über Dokumente, die mit Hilfe einer gemeinsam nutzbaren Datenbank verwaltet werden.

CSCW-Applikationen integrieren meist mehrere Funktionen zur Unterstützung von Gruppenarbeit. Daher ist es oft schwierig, einzelne Applikationen einem einzigen Klassifikationskriterium zuzuordnen.

13.1.2.2. Klassifikationsschema nach Raum und Zeit

Die Unterscheidung bezüglich der örtlichen und zeitlichen Verteilung ist die bekannteste Klassifikation, die sogenannte *Raum-Zeit-Matrix* (siehe Abbildung 13.4). Eine eindeutige Zuordnung ist hierbei oft nicht möglich. Die Raum-Zeit-Matrix bleibt somit nicht ohne Kritik. Die Klassifizierung ist für eine genauere Untersuchung von CSCW-Applikationen zu grob und nicht umfassend genug.

13.2. Informations- und Kommunikationstechnik

CSCW-Applikationen stellen einige spezielle Anforderungen an die Informations- und Kommunikationstechnologie. Aus Sicht der technischen Infrastruktur müssen geeignete Netzwerke vorhanden sein und entsprechende Standards etabliert werden, die den effizienten Informationsaustausch zwischen Personen ermöglichen, die räumlich oder zeitlich getrennt sind. Ein weiterer Entwicklungsbereich neben den Netzwerken und Standards ist die mobile Kommunikation, die zu neuen Möglichkeiten für das mobile Arbeiten führt. Sicherheitsüberlegungen spielen im Bereich der Kommunikationstechnik ebenfalls eine wichtige Rolle.

13.2.1. Netzwerktechnik

Ein Netzwerk ist ein Verbund von räumlich getrennten Computern, die zum Zweck des Datenaustausches verbunden sind. Dabei unterscheidet man zwei Arten von Netzwerken:

Lokale Netze (Local Area Network - LAN) Prinzipiell wird unter lokalen Netzen ein auf Privatgelände beschränktes Netz zur Datenübertragung verstanden, das mehrere hundert Rechnersysteme umfaßt und eine Ausdehnung von maximal einigen Kilometern hat. Die bekanntesten Vertreter für lokale Netze sind Ethernet und das FDDI[2].

Weitverkehrsnetze (Wide Area Network - WAN) Weitverkehrsnetze verbinden Tausende von Computern und können sich über Tausende von Kilometern erstrecken. In den meisten Fällen werden Punkt-zu-Punkt-Verbindungen zwischen angeschlossenen Gruppen von Computern benutzt. Der weltweit größte Verbund mit mehreren Millionen Computern ist das Internet.

13.2.2. Sicherheit

13.2.2.1. Sicherheitsbedrohungen

Ausgangspunkt für Sicherheitsüberlegungen in Kommunikationssystemen sind die folgenden Bedrohungen.

Verlust der Vertraulichkeit Bei der Vervielfältigung von Daten muß die Vertraulichkeit sichergestellt werden, d.h. daß die Daten nur denjenigen bekannt werden dürfen, die dazu berechtigt sind. Die Verbreitung von personenbezogenen Daten muß verhindert werden. Dies kann durch Abhören, dem Vorspiegeln einer falschen Identität oder der Umgehung von Sicherheitsmechanismen geschehen.

Verlust der Integrität Ein Verlust der Integrität bedeutet, daß die Daten aufgrund von Fehleingaben, fehlerhaften Programmen oder anderen Störungen nicht mehr der Realität entsprechen.

[2]Fiber Distributed Data Initiative

Verlust der Verfügbarkeit Ein Verlust der Verfügbarkeit liegt vor, wenn Daten nicht mehr erfaßt und verarbeitet werden können oder wenn die Funktionalität eingeschränkt ist, beispielsweise durch den Ausfall eines PCs. Weitere Gefahren sind Viren oder Würmer.

Verlust der Verbindlichkeit Ein Verlust der Verbindlichkeit kann beispielsweise auftreten, wenn jemand abstreitet, an einer Kommunikation teilgenommen zu haben, d.h. das Absenden oder Empfangen von Daten leugnet. Sobald rechtliche Aspekte wichtig werden, ist die Wahrung der Verbindlichkeit Voraussetzung für die Vermeidung von Schäden.

13.2.2.2. Sicherheitsmethoden

Es existieren eine Vielzahl von Methoden zum Schutz vor den aufgezeigten Bedrohungen. Einige dieser Methoden sind nachfolgend beschrieben.

Verschlüsselung Die Verschlüsselung dient dem Schutz der Vertraulichkeit und der Verbindlichkeit. Sie kann auf zwei Arten angewendet werden. Zum einem für die Verschlüsselung von Meldungen an einen bestimmten Empfänger und zum anderen für die Erzeugung von elektronischen Unterschriften.

Zugriffskontrolle Zugriffskontrollverfahren dienen dazu, nur autorisierten Benutzern Nutzungsrechte zu gewähren, z.B. Lese- und Schreibrechte auf Datenbestände oder Ausführungsrechte für Applikationen.

Integritätssicherung Die Integritätssicherung muß z.B. in Konferenzsystemen gewährleisten, daß Audio- und Videoübertragung auch wirklich zusammengehören und nicht manipuliert wurden.

Verkehrserzeugung Bei der Verkehrserzeugung handelt es sich um Übertragung von Fülldaten, um die Analyse des Verkehrsflusses zu verhindern.

13.2.3. Nutzung von gemeinsamen Informationsobjekten

In diesem Abschnitt geht es um den gemeinsamen (shared) Zugriff auf Informationen, die entweder in Informationsobjekten oder auf Arbeitsoberflächen vorhanden sein können. Eine Arbeitsoberfläche kann von verschiedenen, räumlich getrennt arbeitenden Personen nach dem WYSIWIS[3]-Prinzip synchron betrachtet bzw. manipuliert werden. Moderne Fenstersysteme werden für die Ausgabe einer Applikation auf mehreren Bildschirmen (window multiplexing) und für die Weiterleitung von Eingaben an die Applikationen eingesetzt. Die Werkzeuge zur Unterstützung des Zugangs zu gemeinsamen Objekten können in drei Klassen eingeteilt werden. Sie umfassen Werkzeuge für

- die gleichzeitige Darstellung einer Applikation auf mehreren Bildschirmen,

[3] What you see is what I see

- die synchrone Bearbeitung von Multimedia-Dokumenten durch mehrere Benutzer sowie
- die unstrukturierte aufeinanderfolgende Bearbeitung des Multimedia-Dokumentes in einer Gruppe.

13.3. CSCW-Applikationen

13.3.1. Elektronische Postsysteme

Elektronische Postsysteme unterstützen den asynchronen Austausch von Informationen. Sie sind grundsätzlich asynchrone Kommunikationshilfsmittel. Dies hat gegenüber synchroner Kommunikation den Vorteil, daß keine gleichzeitige Anwesenheit der Kommunikationspartner erforderlich ist. Der größte Netzverbund, der elektronische Postdienste anbietet, ist das Internet. Hier hat sich der Standard SMTP[4] vorwiegend etabliert. Mit Hilfe des MIME[5]-Standards ist es zudem möglich, auch nicht-textuelle Meldungen, z.B. Bilder und Töne, weltweit auszutauschen. Elektronische Postsysteme erlauben darüberhinaus das gleichzeitige Verschicken einer Nachricht an mehrere Empfänger.

13.3.2. Bulletin Board-Systeme

Bulletin Board-Systeme sind spezielle Datenbanken, bei denen Meldungen verschiedener Autoren nach Themenschwerpunkten gespeichert und einer Vielzahl von Lesern zur Verfügung gestellt werden. Die Meldungen eines Themas bilden eine sogenannte Interessensgruppe (engl.: News Group). Sie besitzen ein semi-strukturiertes Format, ähnlich dem einer eMail. Bulletin Board-Systeme ermöglichen eine schnelle Verfügbarkeit von Informationen für räumlich und zeitlich verteilte Gruppen und werden auch für verteilte asynchrone, offene Diskussionen verwendet. In Bulletin Board-Systemen findet eine implizite Kommunikation statt, d.h. der Autor einer Meldung stellt diese grundsätzlich einer anonymen Leserschaft zur Verfügung. Deswegen handelt es sich hierbei um eine 1:n-Kommunikationsbeziehung. Bekannte Beispiele für Bulletin Board-Systeme sind das Usenet oder das Fidonet.

13.3.3. Konferenzsysteme

Im Gegensatz zu den bisher beschriebenen asynchronen Kommunikationssystemen stehen die synchronen Konferenzsysteme. Sie unterstützen eine genau adressierbare Menge von Kommunikationspartnern, die zur gleichen Zeit miteinander kommunizieren wollen. Ein Konferenzsystem schaltet Kommunikationskanäle zu Teilnehmern zusammen, so daß alle die gesamte Kommunikation mitverfolgen können. Konferenzsysteme können mit verschiedenen Medien realisiert werden. Es gibt rein textuelle, audiobasierte oder videobasierte Konferenzsysteme sowie Mischformen.

[4] Simple Mail Transfer Protocol
[5] Multipurpose Internet Mail Extension

13.3.3.1. Textbasierte Konferenzsysteme

Relativ früh gab es die Möglichkeit, auf einem Großrechner zwischen zwei Benutzern eine Verbindung herzustellen, so daß textuell synchron kommuniziert werden konnte. Beispiele dafür sind *talk* oder *chat* auf UNIX-Systemen. In *talk* wird der Bildschirm horizontal in mehrere Bereiche unterteilt, in denen jeweils ein Konferenzteilnehmer seinen Beitrag hineinschreiben kann. Alle anderen können gleichzeitig mitverfolgen, was geschrieben wird.

13.3.3.2. Videokonferenzsysteme

Die Videotechnologie unterstützt die synchrone Kommunikation in verteilten Gruppen durch Ton- und Bildwiedergabe mehrerer Konferenzteilnehmer. Das sogenannte Bildtelefon (erstmals 1964 von AT&T vorgestellt) ist der Vorläufer von Videokonferenzsystemen. Sie weisen demgegenüber einen erhöhten Funktionsumfang auf und erlauben eine bessere Bildqualität. Es wird dazu aber eine Übertragungskapazität von 1-10 Mbps und eine räumlich feste Installation benötigt, die im allgemeinen in einem speziellen Raum eingerichtet ist. Videokonferenzsysteme konnten sich bisher nicht stark durchsetzen. Dies liegt weniger an der Aktzeptanz bei den Benutzern, als an den hohen Kosten und den zu geringen Übertragungskapazitäten.

13.3.3.3. Desktopkonferenzsysteme

Desktopkonferenzsysteme sind eine Weiterentwicklung der Videokonferenzsysteme. Durch die Verfügbarkeit von höheren Bandbreiten und verbesserten Kompressions- und Verarbeitungsmöglichkeiten ist eine synchrone Videoverbindung heute vom Arbeitsplatz aus möglich. Die Kommunikationspartner müssen somit nicht mehr in ein speziell eingerichtetes Videokonferenzstudio wechseln. Es können neben mehreren Videoverbindungen gleichzeitig andere CSCW-Applikationen benutzt und andere Informationsobjekte verteilt eingesehen oder manipuliert werden.

13.3.4. Gruppeneditoren

Gruppeneditoren unterstützen die gemeinsame Erstellung von Multimedia-Dokumenten. Es lassen sich drei Typen unterscheiden.

Annotationssysteme erlauben es, an ein Dokument Kommentare und Korrekturen anzufügen. Hierbei gibt es drei Methoden.

1. Im einfachsten Fall verwenden die Autoren das gleiche Textverarbeitungssystem und fügen ihre Kommentare direkt in das Dokument ein, evtl. mit einer anderen Farbe oder einem anderen Schriftsatz. Dies verändert aber unmittelbar das Originaldokument.

2. Eine andere Methode ist die Verwendung der Overlay-Technik. Hierbei werden die Kommentare auf einer gesonderten Ebene geschrieben, wie beim Auflegen einer Klarsichtfolie auf ein beschriebenes Blatt.
3. Eine dritte Möglichkeit ist das Anbringen von sogenannten elektronischen Notizzetteln an bestimmte Stellen im Originaldokument.

Koautorensysteme unterstützen die Dokumentenerstellung durch einen oder mehrere Autoren. Dabei können je nach System mehrere Benutzer gleichzeitig an einem Dokument an verschiedenen Stellen Veränderungen vornehmen, während diese den anderen Autoren unmittelbar auf dem Bildschirm angezeigt werden.

Gemeinsam verwendbare Zeichnungswerkzeuge erlauben ähnlich wie bei den Textdokumenten die simultane Erstellung von Bildern und Zeichnungen.

13.3.5. Planungssysteme

Die Koordination von Ressourcen und Kapazitäten ist seit jeher ein Optimierungsproblem der industriellen Fertigung. Durch den Zwang zur Kosteneinsparung wird dieses Problem zunehmend auch in anderen Bereichen relevant. Planungssysteme legen das Schwergewicht auf die Verwaltung und Koordination von Ressourcen.

13.3.5.1. Ressourcenplanung

Im industriellen Bereich versucht man, das Ressourcen-Koordinations-Problem durch den Einsatz von Produktionsplanungs- und -steuerungssystemen zu lösen. Die Schwächen dieser Systeme, die vor allem bei schlecht strukturierten und unvorhersehbaren Abläufen auftreten, lassen zunehmend alternative Konzepte zum Zuge kommen, die eine zentrale Planung in Frage stellen und Koordinationsfunktionen an unterste Ebenen, an Arbeitsgruppen und Teams delegieren.

13.3.5.2. Terminplanung

Ein Terminverwaltungs- und -vereinbarungssystem kann zur Verwaltung der eigenen Termine und zur Terminvereinbarung innerhalb von Gruppen genutzt werden. Es erleichtert somit die Projekt- und Terminplanung einer Arbeitsgruppe. Voraussetzung für das Funktionieren solcher Systeme ist, daß alle Gruppenmitglieder ihre Termine in ihren elektronischen Kalender eintragen. Das System zeigt dann sämtliche Termine der Gruppenmitglieder auf einem Übersichtsplan an und generiert Vorschläge für einen möglichen Termin.

13.3.6. Workflow Management-Systeme

Workflow Management-Systeme (WFMS) dienen der Optimierung von Geschäftsprozessen durch Erhöhung der Produktivität und Flexibilität von Abläufen. Workflow Management-Systeme erfüllen dabei folgende Aufgaben:

- *Modellierung* Die Modellierung von Workflows erfolgt bei den meisten WFMS mit Hilfe semi-formaler Sprachen. Solche Sprachen umfassen eine Menge von Konstruktionsoperatoren, die die Syntax des Modells bilden. Kommerziell verfügbare WFMS verwenden z.B. Datenflußdiagramme, Petri-Netze, Zustandsübergangsdiagramme oder PERT[6]-Diagramme. Alle Techniken haben spezifische Vorteile (wie z.B. Petri-Netze, mit deren Hilfe sehr einfach Simulationen durchführbar sind).

- *Simulation* Mit Hilfe von Simulationen kann die inhaltliche und formale Korrektheit des Workflow-Schemas überprüft werden. Außerdem werden quantitative Aussagen über den Workflow möglich, beispielsweise Durchlaufzeiten, Kosten oder die Auslastung von Ressourcen.

- *Ausführung und Steuerung* Die Ausführung und Steuerung von Workflows umfaßt das Starten und Beenden von Aktivitäten, die Zuordnung von Akteuren zu Aktivitäten, die Allokation von Ressourcen und die Überwachung definierter Ablaufregeln.

Nach Heilman läuft ein Workflow Management idealtypisch in einem Zyklus ab (siehe Abbildung 13.5 auf der nächsten Seite). Einstiegspunkt in den Workflow Management-Zyklus ist die Modellierung eines bereits existierenden Workflows (*Ist-Modellierung*). Im Anschluß an die Ist-Modellierung erfolgt eine Analyse hinsichtlich der Stärken und Schwächen dieses Workflows. Durch *Simulationen* erhält man Hinweise auf weitere Analyseschritte. Auf Basis der Analyseergebnisse erfolgt anschließend die *Soll-Modellierung*. Der Teilzyklus (1) muß unter Umständen mehrmals durchlaufen werden, und zwar solange, bis ein hinreichend optimierter Workflow vorliegt. Das Schema des so modellierten neuen Workflows dient der *Steuerung* des real ablaufenden Workflows (2). Innerhalb von Teilzyklus (2) erfolgt die *Protokollierung* des Ablaufes des Workflows. Das Protokoll kann für *Revisionszwecke* und als Eingabe (3) in eine erneute Analyse Verwendung finden.

13.4. Fazit

CSCW unterstützt und ermöglicht in vielen Bereichen die Gruppenarbeit und sorgt für eine effiziente Zusammenarbeit. Gerade bei verteilten Gruppen lassen sich die Kommunikationshilfsmittel wie eMail oder News Groups sinnvoll einsetzen. Aber auch Konferenzsysteme ersparen den Gruppen weite Wege und tragen dadurch zur Kostenreduktion bei. Die Gruppeneditoren erlauben es zudem, Dokumente simultan zu erstellen, um ein paralleles Arbeiten zu ermöglichen. Die genaue Projekt- und Terminplanung ist notwendig, um das Gruppenziel rechtzeitig zu erreichen. Workflow Management-Systeme sollen außerdem dazu beitragen, daß Arbeitsabläufe effizienter gestaltet werden können. Alles in allem sorgen die CSCW-Systeme für mehr Effektivität und Effizienz bei der Gruppenarbeit und sollten daher bei der Zusammenarbeit von Arbeitsgrupen und Teams nicht fehlen. Da die benötigte Infrastruktur heutzutage an vielen Arbeitsplätzen schon vorhanden ist, sind auch keine großen Investitionen mehr notwendig. Allerdings muß man

[6]Program Evaluation and Review Technique

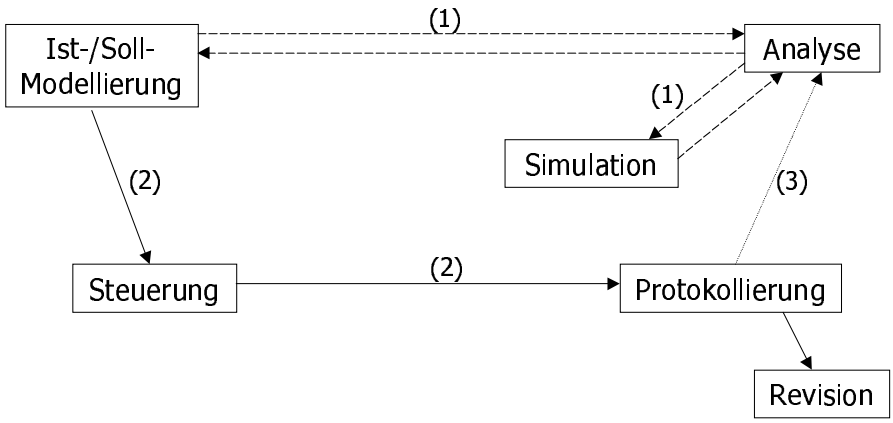


Abbildung 13.5.: Workflow Management-Zyklus

an Maßnahmen gegen die Sicherheitsbedrohungen denken, um Schäden und Verluste zu vermeiden.

14. Projektmanagement

Autor: *Sebastian Schütte*

Es werden ausgewählte Themen und Methoden aus dem Projektmanagement vorgestellt, die nach Meinung des Autors für die erfolgreiche Realisierung des Projektgruppenzieles von Relevanz sind.
"Egoless Programming" bezeichnet eine Form der Zusammenarbeit, die die Bereitschaft des Einzelnen zur Kritik erhöhen soll.
Ein prototypisches Vorgehen erlaubt den Entwicklern, Erfahrungen zu sammeln und Fehler zu machen.
Die Verwendung eines Prozeßmodelles erlaubt die Strukturierung der Entwicklung und gliedert den Entwicklungsprozeß in definierte Phasen. Die Entwicklung wird transparent und steuerbar.
Die Arbeit im Team bedarf einer reflektierten Lenkung, um die Effektivität der Vorgaben verbessern zu können und um auf Veränderungen in der Gruppensituation geeignet reagieren zu können.

14.1. Motivation

Die Bedeutung des Themas Projektmanagement für eine universitäre Projektgruppe könnte in Frage gestellt werden, da die im folgenden beschriebenen Methoden im Kontext kommerzieller Software-Entwicklung entwickelt wurden. Hauptargument könnte das Fehlen eines Budgets sein, das für eine wirtschaftlich erfolgreiche Entwicklung eingehalten werden muß. Bei näherem Hinsehen entkräftet sich dieses Argument jedoch von selbst. Der Autor ist der Meinung, daß die im folgenden genannten Probleme klassische Probleme der Software-Entwicklung sind und somit auch mit Methoden angegangen werden dürfen, die sich im Kontext kommerzieller Entwicklung bewährt haben:

- Für die Entwicklung steht zwar kein begrenztes Budget in Form eines Geldbetrages zur Verfügung, jedoch ist die verfügbare Entwicklungszeit beschränkt.
- Das Team kann im Laufe der Zeit kleiner werden, neue Mitglieder kommen auf keinen Fall hinzu.
- Die Entwickler besitzen wenig Erfahrung in der Anwendungsdomäne.
- Das geforderte Produkt stellt nach Meinung des Autors ein ehrgeiziges Ziel dar.
- Es werden fremde Teilkomponenten verwendet.

14.2. Prozeßmodelle

Ein Prozeßmodell beschreibt die Form eines strukturierten Vorgehens in der Softwareentwicklung. Der Entwicklung wird in Phasen untergliedert zwischen denen Übergänge definiert werden. Durch die Verwendung eines Prozeßmodelles soll die Softwareentwicklung verläßlich, vorhersehbar und effizient werden [GJM91].
Das Prozeßmodell strukturiert den Entwicklungsprozeß grob. Die Aufteilung in Phasen wie beispielsweise Analyse, Implementation oder Test gestaltet die Entwicklung transparent und überwachbar. Im Allgemeinen steht als Abschluß einer Phase die Erstellung eines Dokumentes, das die Ergebnisse der vorangegangenen Phase festhält. Dies kann beispielsweise ein Testplan sein oder auch ein Entwurfsdokument. Solche Dokumente eigenen sich als Meilensteine (siehe Abschnitt 14.4.5.1 auf Seite 126) zur Überprüfung des Zeitplans.

14.2.1. Spiralmodell

Das Spiralmodell ist im Gegensatz zum bekannten Wasserfallmodell ein iteratives Modell. In der Regel werden die Phasen des Spiralmodelles mehrfach durchlaufen, wobei die Ergebnisse der vorangegangenen Phasen verwendet werden, um das weitere Vorgehen zu planen [GJM91].
Abbildung 14.1 auf der nächsten Seite visualisiert die Softwareentwicklung gemäß des Spiralmodelles. Das Diagramm ist in vier Quadranten aufgeteilt, die den vier Phasen der Entwicklung gemäß dieses Modelles entsprechen. Phase eins beginnt oben links. In Phase eins werden das Ziel der Entwicklung festgelegt sowie Alternativen in der Entwicklung aufgezeigt. In Phase zwei werden die aufgezeigten Alternativen evaluiert und deren Risiken bestimmt (evtl. durch die Erstellung von Prototypen). In Phase drei wird das zu entwickelnde Produkt entworfen bzw. implementiert. Phase vier ist eine Besonderheit dieses Modells. Die Erfahrungen der vorangegangenen Entwicklungsphasen werden verwendet, um die folgende Iteration der Spirale zu planen. Man begibt sich auf die nächst höhere Abstraktionsebene und plant den weiteren Entwicklungsprozeß selbst, während in den vorangegangenen Phasen an dem konkreten Produkt gearbeitet wurde. Solche Spiralumläufe werden solange wiederholt, bis das Produkt die gewünschte Qualität erreicht hat.

14.3. Evolutionäre Software-Entwicklung

Am Anfang der Software-Entwicklung stehen Entwurfsentscheidungen, die den weiteren Verlauf der Entwicklung nachhaltig prägen. Ein schlechter Entwurf kann die Entwicklung drastisch verlängern. Oft erweisen sich Entwurfsentscheidungen erst am Ende in Nachhinein als ungünstig, da die Anforderungen an das Produkt erst im Laufe der Zeit konkret werden. Vor allem beim Entwurf von Systemen, für deren Anwendungsdomäne wenig Erfahrung bei den Entwicklern vorhanden ist, ist ein sinnvoller Entwurf des Produktes kaum möglich, da das dazu notwendige Wissen erst am Ende der Entwicklung zur Verfügung steht. (Anmerkung des Autors: Dies trifft auf die Projektgruppe $\mathcal{HEU}$ mit

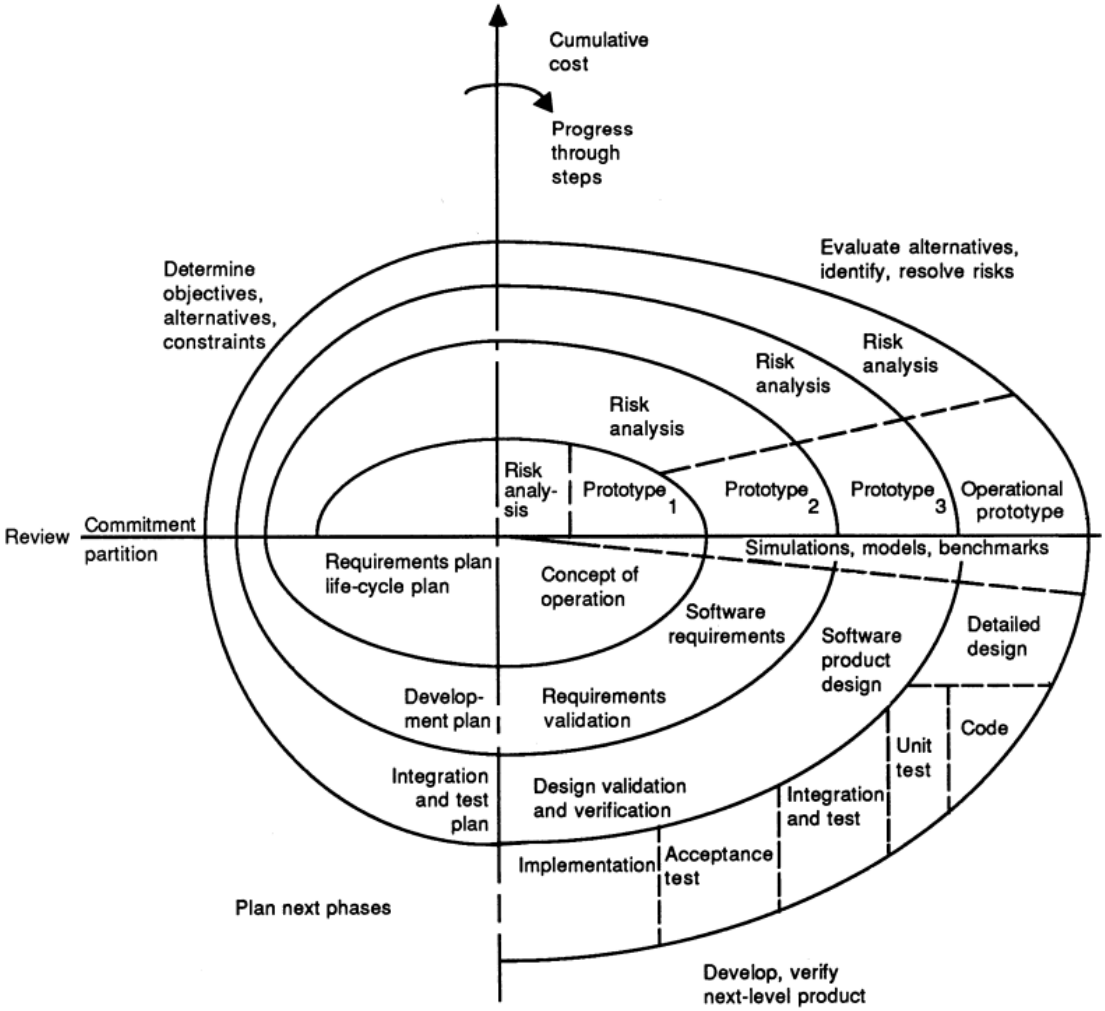


Abbildung 14.1.: Spiralmodell

Sicherheit zu, da wenig Erfahrung in der Erstellung von graphischen Editoren sowie der Sprache DoDL existiert.)

14.3.1. Prototyping

Ein Ansatz zur Lösung des oben genannten Problems ist die Erstellung eines Prototypen, der wesentliche Grundzüge der zu entwickelnden Anwendung realisiert. Die Zeit, die zur Erstellung dieses Prototypen notwendig ist, geht natürlich für die Entwicklung der eigentlichen Anwendung verloren. Es gibt zahlreiche Gründe, die für die Erstellung eines Prototypen sprechen:

- Die Durchfürbarkeit der gestellten Anforderung wird verifiziert.

- Grundlegende Funktionen können zu einem frühen Zeitpunkt der Entwicklung getestet und optimiert werden.
- Es findet eine frühe Rückkopplung der Entwurfsentscheidungen statt.
- Das Verständnis für die Anwendung wächst mit der Erstellung des Prototypen.
- Die Entwickler sammeln Erfahrung aus der Anwendungsdomäne.
- Die Anforderungen an das Produkt werden konkretisiert.

Man kann Prototypen in "quick and dirty"-Manier erstellen, so daß Wartbarkeit und Erweiterbarkeit keine Rolle bei der Implementierung spielen. Der erzeugte Code kann somit im Allgemeinen nicht weiter verwendet werden. Die Entwicklung der geforderten Anwendung allerdings steht nach Erstellung des Prototypen auf sichererem Fundament. Die gewonnene Erfahrung sowie die sehr viel konkreteren Anforderungen können verwendet werden, um das endgültige Produkt zu entwerfen.
Wenn bereits bei der Erstellung des Prototypen Wartbarkeit und Erweiterbarkeit eine Rolle spielen bzw. sich der Entwurf als tragfähig erwiesen hat, können möglicherweise Teile des Codes für das endgültige Produkt verwendet werden. Dann geht das Prototyping in die evolutionären Software-Entwicklung über.

14.3.2. Inkrementelle Implementierung

Ein weiterer Ansatz, um das in Abschnitt 14.3 auf Seite 120 skizzierte Problem zu entschärfen, ist die sogenannte "Inkrementelle Implementierung". Die zu entwickelnde Anwendung wird in Komponenten zerlegt, die für sich ein sinnvolles Teilproblem lösen. Jede dieser Komponenten durchläuft ihren eigenen Entwicklungsprozeß. Die Entwicklung derart aufzuteilen, bietet folgende Vorteile:

- Die Teilsysteme sind deutlich weniger komplex als die gesamte Anwendung. Die Erfahrung zeigt, daß die Entwicklung mehrerer kleiner Systeme einfacher ist, als die Realisierung eines großen Systems.
- Die Realisierung von Teilsystemen, die nicht zum Grundgerüst der Anwendung gehören, kann zurückgestellt werden. Somit können knappe Ressourcen konzentriert werden.
- Fertiggestellte Teilsysteme können den Anwendern präsentiert werden. Die geäußerte Kritik kann verwendet werden, um die Entwicklung der übrigen Teilsysteme besser an die Anforderungen der Anwender anzupassen.
- Ein System, das aus einzelnen Anwendungen besteht, ist leichter zu erweitern und zu warten.

14.4. Projektmanagement

Während im vorangegangenen Teil der Ausarbeitung Methoden der Softwaretechnologie vorgestellt wurden, befaßt sich der nun folgende Teil mit den Aufgaben die bei der Abwicklung einer Softwareentwicklung zu bewältigen sind. Das sogenannte "Projektmanagement" umfaßt Aufgaben der Planung, Organisation, Lenkung, Personalpflege und Kontrolle, die im folgenden im Hinblick auf die Projektgruppe $\mathcal{HEU}$ vorgestellt werden.

14.4.1. Planung

Bevor die Entwicklung der Anwendung beginnen kann, müssen Rahmenbedingungen bestimmt werden. Allen an der Entwicklung Beteiligten sollte klar sein, welches **Ziel** die Entwicklung verfolgt. Dies hört sich trivial an, ist es aber keineswegs. Im Laufe der Entwicklung werden Gruppen gebildet, die ein jeweils eigenes Teilproblem lösen. Oftmals ist dazu der Erwerb von Spezialwissen unerläßlich. Durch die Konzentration auf Teilaspekte geht der Überblick über das ganze Projekt verloren. So entwickeln sich innerhalb der Teilgruppe eigene Ziele, die von denen der übrigen Beteiligten erheblich abweichen können (siehe dazu Abschnitt 14.4.4.1 auf Seite 125).
Ein geeignetes **Prozeßmodell** ist zu wählen. Für die Entwicklung der $\mathcal{HEU}$ eignet sich nach Meinung des Autors das in Abschnitt 14.2.1 auf Seite 120 vorgestellte Modell. Wenngleich der im Bereich kommerzieller Software-Entwicklung zentrale Punkt der Risikoanalyse im Fall der Projektgruppe $\mathcal{HEU}$ wenig berücksichtigt werden wird, erscheint dem Autor ein iteratives Vorgehen unerläßlich, um die notwendige Sachkenntnis zu erwerben.
Im Kontext kommerzieller Software-Entwicklung ist eine initiale **Aufwandsschätzung** eine zentrale Aufgabe des Projektmanagements. Eine realistische Aufwandsschätzung ist die Grundlage jedes Zeitplanes. Aber auch im Kontext einer universitären Projektgruppe ist eine realistische Aufwandsschätzung notwendig, da eine strikte Zeitbegrenzung der Entwicklungsdauer vorliegt. Ein realistischer **Zeitplan** für das PG-Projekt $\mathcal{HEU}$ läßt sich nach Meinung des Autors erst aufstellen, wenn die Erfahrungen aus der Implementierung des Prototypen vorliegen. Dieser Zeitplan kann dann als Grundlage für die weitere Kontrolle des Projektes dienen (siehe auch Abschnitt 14.4.5 auf Seite 125).

14.4.2. Personalpflege

Aus dem Bereich Personalpflege sind zwei Punkte nach Meinung des Autors von Relevanz für die Projektgruppe $\mathcal{HEU}$.
Der Aufwand für den Erwerb von notwendiger Sachkenntnis muß bei der Erstellung des Zeitplanes berücksichtigt werden. Bestandteil der Aufgabenstellung ist die Verwendung der Datenbank H-PCTE, der Klassenbibliothek Swing, der Programmiersprache Java sowie der Modellierungssprache UML. Kenntnis dieser Sprachen bzw. Systeme muß von den Mitgliedern der PG erst noch erworben werden und ist Teil des Lernzieles. Aufgabe des Projektmanagements ist es, einen hierfür geeigneten Rahmen zu schaffen.
Um die Motivation der Gruppenmitglieder aufrecht zu erhalten, ist es sinnvoll, Aufgaben

rotieren zu lassen und dafür Sorge zu tragen, daß die Arbeit der einzelnen Gruppenmitglieder derart dokumentiert wird, daß im Falle von Krankheit oder Ausscheiden aus der Gruppe die wahrgenommenen Aufgaben nach möglichst kurzer Zeit von anderen PG-Mitgliedern übernommen werden können.

14.4.3. Organisation

Die an der Entwicklung beteiligten Teams können auf verschiedene Weise organisiert werden, je nachdem, ob Effizienz oder Qualität der in der Gruppe gefundenen Lösungen im Vordergrund stehen. Eine Organisationsstruktur, bei der die Effizienz im Vordergrund steht, ist das sogenannte "chief programmer team", bei dem einem besonders begabter Programmierer von einem Assistenten und einem Bibliothekar zugearbeitet wird (siehe dazu [GJM91]). Eine solche Struktur ist sinnvoll, wenn die Aufgabe aus einem Gebiet stammt, das gut verstanden ist, bzw. die Teilaufgaben überschaubar sind, so daß ein Einzelner sie erfolgreich lösen kann.
Bei Aufgaben, die noch nicht klar verstanden sind, bietet sich eine demokratischere Teamstruktur an, bei der Lösungen gemeinsam gefunden werden. Sofern ein solches Team nicht zu groß wird, wird die Qualität der getroffenen Entscheidungen höher sein, als dies bei Einzelentscheidungen der Fall ist. Ein solches Team arbeitet unter Umständen langsamer, da viel Zeit für Kommunikation notwendig ist.

14.4.3.1. Egoless Programming

Das "Egoless Programming" verfolgt das Ziel, die Offenheit für Kritik zu fördern und somit die Qualität der gefundenen Lösungen zu verbessern. Die Gruppe ist als Ganzes für das Programm verantwortlich, unabhängig, wer Entscheidungen getroffen hat, bzw. einen Fehler zu verantworten hat.

Ausgangspunkt dieser Politik sind Untersuchungen ([GJM91]), die gezeigt haben, daß Menschen, die eine bestimmte Auffassung vertreten, dazu tendieren, Kritik an dieser Auffassung aus dem Weg zu gehen. Ein solches Verhalten konterkariert das Arbeiten in der Gruppe. Anstatt Kritik als Möglichkeit der Verbesserung zu suchen, wird ihr aus dem Weg gegangen. Die Konzentration auf die Gruppe beim "Egoless Programming" soll diesem Effekt entgegen wirken. So soll die Bereitschaft zu Kritik und Reviews erhöht werden.

14.4.3.2. Konfigurationsmanagement

Hat die zu entwickelnde Anwendung ein bestimmtes Stadium erreicht, so wird ein symbolischer Name vergeben und der Quellcode eingefroren. Falls sich spätere Änderungen als nicht sinnvoll erweisen, kann man zu einem definierten Zustand zurückkehren. Auch ist auf diese Weise möglich, parallele Entwicklungslinien zu beginnen, um beispielsweise einem Team grundlegende Änderungen zu ermöglichen, während der Rest weiterentwickeln kann. Später können die Produktlinien dann wieder zusammengeführt werden. Diese Aufgabe kann durch ein Releasemanagementsystem unterstützt werden, daß die

Vielzahl der abzuspeichernden Codefragmente verwaltet. Weitere Ausführung zu diesem Thema würden den Rahmen der vorliegenden Ausarbeitung sprengen.

14.4.4. Lenkung

Damit eine Gruppe von Entwicklern erfolgreich zusammenarbeiten kann, bedarf es sinnvoller Vorgaben, wie die Arbeit zu organisieren ist. Auf welche Weise die Zusammenarbeit zu organisieren ist, dafür läßt sich jedoch kein Patentrezept angeben, da jedes Projekt seine spezifischen Eigenheiten hat. Hinzu kommt, daß sich die Art, wie die Gruppe miteinander umgeht, im Laufe der Zeit ändert (siehe dazu den folgenden Abschnitt). Aus diesem Grund ist eine Rückkopplung der Lenkungsvorgaben unerläßlich. Neben technischen Reviews sollte es auch Besprechungen geben, die der Verbesserung des Arbeitsklimas dienen.

14.4.4.1. Gruppenphasen

Aus der Soziologie ist bekannt, daß Gruppen, die sich neu zusammenfinden, bestimmte Phasen durchleben, die die Kommunikation innerhalb der Gruppe und das Verhalten der Gruppe nach außen stark prägen.

- *Kontakt & Orientierung:* Die Gruppenmitglieder sind stark auf die Leitung fixiert, Kommunikation untereinander findet kaum statt.
- *Machtkampf & Kontrolle:* Zwischen den Gruppenmitgliedern bilden sich Beziehungen heraus. Meinungsführer und Cliquen kristallisieren sich heraus. Die Kommunikation ist durch Konkurrenzkampf geprägt.
- *Vertrautheit:* Zu diesem Zeitpunkt bildet sich ein "Wir-Gefühl" heraus. Die Gruppe akzeptiert die Eigenheiten ihrer Mitglieder, die Kommunikation ist durch hohe Toleranz geprägt. Die Gruppe zeigt sich nach außen sehr geschlossen, neue Impulse werden kaum aufgenommen.
- *Differenzierung & Abgrenzung:* Die Gruppe hat ein klares Profil gewonnen. Die Stärke der Bindung läßt nach und Impulse von außen können aufgenommen werden.

Bei der Lenkung der Gruppe sollten die Veränderungen im Umgang innerhalb der Gruppe gemäß der oben vorgestellten Phasen berücksichtigt werden. Das Durchleben dieser Phasen ist notwendig und sollte nicht unterdrückt werden, jedoch lassen sich die ablaufenden Prozesse unter Umständen kanalisieren. So kann beispielsweise das Risiko vermindert werden, daß ein Mitglied die Gruppe aufgrund großer Spannungen verläßt.

14.4.5. Kontrolle

Der Fortschritt in der Entwicklung eines immateriellen Produktes wie Software kann nur indirekt beobachtet werden. Die Konvergenz zum Zeitplan ist ein Kriterium, anhand dessen Fortschritt gemessen werden kann.

14.4.5.1. Gant-Chart

Ein Gant-Chart ist eine Notation, mit deren Hilfe eine Resourcenzuteilung visualisiert werden kann. In der vorliegenden Variante wird die für eine Aufgabe veranschlagte Zeit dunkel dargestellt, eine eventuelle Verzögerung ohne Auswirkung auf den Zeitplan hell. Aufgaben ohne Zeit für Verzögerungen liegen auf dem sogenannten "Kritischen Pfad" (In der Abbildung bilden die durchgängig schwarz gefärbten Balken den Kritischen Pfad). Verzögerungen in diesen Aufgaben führen zur Verzögerung des Gesamtprojektes. Der Verlauf dieser Teilaufgaben muß also besonders sorgfältig überwacht werden.
Um den Zeitplan transparent zu gestalten, ist es sinnvoll, sogenannte Meilensteine zu

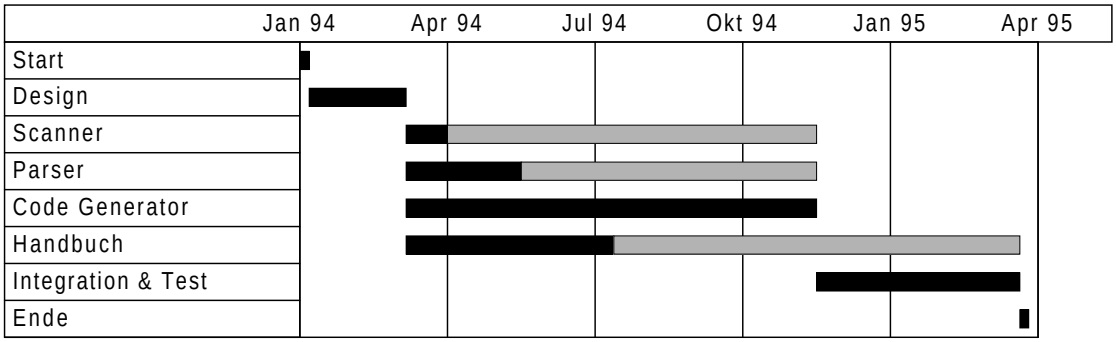


Abbildung 14.2.: Beispiel eines Gant-Charts

setzen. Ein Meilenstein kennzeichnet die Fertigstellung eines leicht zu überprüfenden Teilzieles, wie z.B. die Abgabe eines bestimmten Dokumentes. Ein Meilenstein ist immer mit einem konkreten Datum verbunden und ist ein Hinweis auf Konvergenz zum Zeitplan (Beispiel: 1. November 1994, Implementierung abgeschlossen).

14.4.5.2. PERT-Chart

Die Abhängigkeiten der einzelnen Aufgaben sind nur implizit aus dem Gant-Chart zu erkennen. Diese werden unter Verwendung der "Programm Evaluation and Review Technique" deutlich. Abhängigkeiten werden durch Pfeile gekennzeichnet.
Die Teilaufgaben können je nach Notation mit weiteren Daten dekoriert werden. Aufgaben auf dem kritischen Pfad werden fett dargestellt, Meilensteine können als Teilaufgaben ohne Dauer eingeführt werden.

14.5. Zusammenfassung

Die hier vorgestellten Methoden können nach Meinung des Autors dazu beitragen, daß die Entwicklung der $\mathcal{HEU}$ erfolgreich verläuft.
Die Benennung der Aufgaben des Projektmanagements macht darauf aufmerksam, daß neben den zahlreichen technischen Aufgaben auch Aufgaben aus anderen Bereichen zu

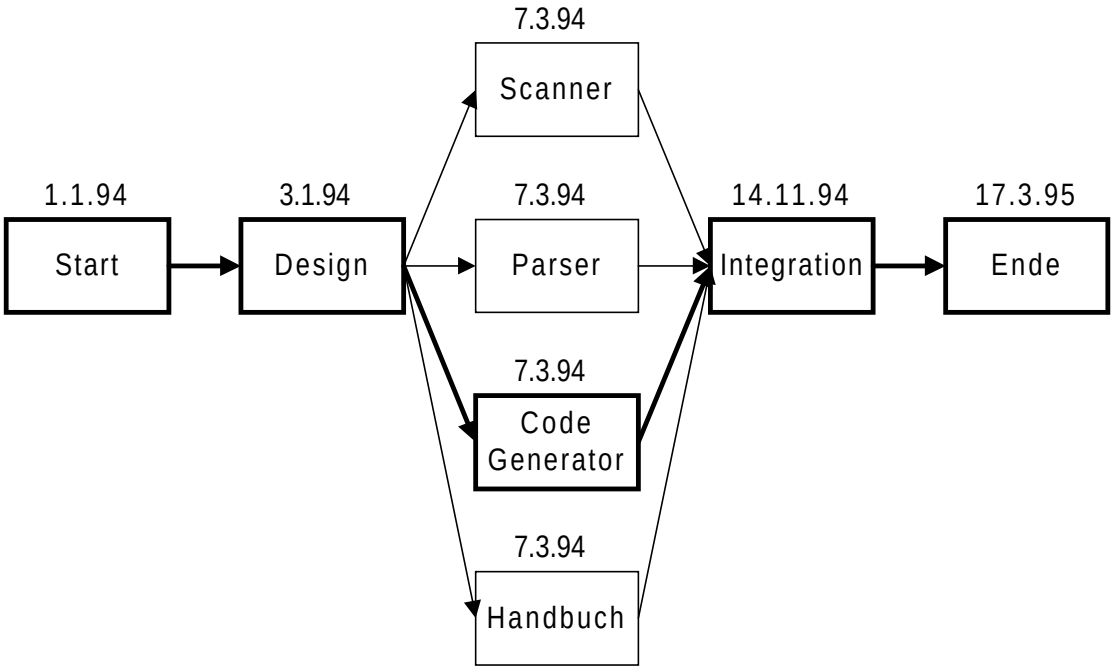


Abbildung 14.3.: Beispiel eines PERT-Charts

bewältigen sind. Dies sind vor allem die Koordination von Arbeit in der Gruppe, die Erhaltung der Motivation und die Sicherstellung der Qualität des erzeugten Produktes. Die Implementierung eines Prototypen, wie in der Aufgabenstellung der PG vorgesehen, bietet die Möglichkeit, die geschilderten Probleme zu erfahren. Das Ziel dieser Ausarbeitung soll sein, die Reflektion der getätigten Erfahrungen zu ermöglichen, indem hier beschriebene Schwierigkeiten und deren Lösungsansätze in der alltäglichen PG-Arbeit wiedererkannt werden. Verläuft die Entwicklung bewußt, kontrolliert und reflektiert, kann dies nach Meinung des Autors erheblich dazu beitragen, daß die Erfahrungen aus der PG für spätere Projekte genutzt werden können.

Teil III.

Prototyp Ampelstadt

15. Problembeschreibung der Ampelstadt

Autor: *Alexander Fronk*

Über die Ampelstadt und ihren Sinn und Zweck ist hier zu berichten. Es wird eine grafische Notation grob vorgestellt und einige Eigenschaften eines unterstützenden Editors skizziert. Die Ananlyse des hier gegebenen Problems erfolgt im nächsten Abschnitt.

15.1. Warum eine Ampelstadt?

Die Ampelstadt ist ein erdachtes Szenarium, das die Entwicklung der $\mathcal{HEU}$ vereinfachen soll. Sie ist an die später zu realisierenden Editoren und deren Funktionalität angelehnt, erreicht jedoch bei weitem nicht deren Komplexität. Das ist beabsichtigt. Die zu entwerfende grafische Notation ist extrem einfach gehalten, die Funktionalitäten des zugehörigen Editors ebenfalls. Die Ampelstadt ist darüberhinaus oder gerade deswegen als „Wegwerfprodukt"geplant, an dem lediglich aber dennoch im besonderen technisches Wissen und handwerkliches Knowhow ausprobiert und erworben werden soll.

15.2. Die grafische Notation

Städte, in denen man nicht wirklich leben möchte, weil es keine Häuser und Parks gibt, bestehen lediglich aus Kreuzungen, an denen der Verkehr durch Ampeln geregelt wird. Kreuzungen werden über Strassen erreicht, die entweder fußgängerfreundliche Einbahnstrassen sind oder radfahrerunfreundlich zweispurig ausfallen können. Radfahrwege und Fußgängerüberwege sind nicht geplant.

Man sieht recht schnell, daß sich diese Städte als gerichtete attributierte Graphen entpuppen, wie Bild 15.1 auf der nächsten Seite deutlich zeigt. Kreuzungen sollen als Knoten modelliert werden, Ampeln als Attribute der Knoten. Eine Kante zwischen zwei Knoten ist gerichtet und steht damit für eine Einbahnstrasse. Doppelte Einbahnstrassen, in verschiedenen Richtungen, bilden in Konsequenz zweispurige Strassen.

15.3. Eigenschaften des Editors

Das grafische und interaktive Konstruieren von Ampelstädten soll durch einen syntaxgesteuerten Editor unterstützt werden. Der Editor soll verteilt arbeiten, d.h. mehrere Benutzer sollen an verschiedenen Arbeitsplätzen diegleichen Städte gleichzeitig bearbeiten können. Ein Notifikationsmechanismus soll dabei die benötigten Informationen über

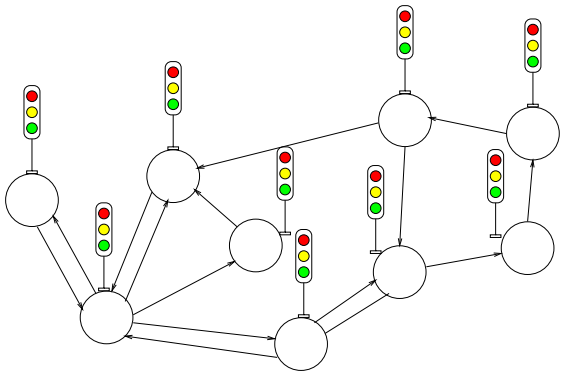


Abbildung 15.1.: Eine Ampelstadt

Veränderungen an einer Stadt den anderen Benutzern mitteilen. Es sollen Rechte vergeben werden können, die Benutzern bestimmte Aktionen ermöglichen oder verweigern. Die erbauten Städte sollen in einer Datenbank gespeichert und von dort auch wieder geladen werden können. Eine Reportbox soll über den Zustand der gerade neu angelegten Städte berichten, als auch den Besuch persistent gemachter Siedlungen ermöglichen, die dann nicht weiter verändert werden dürfen.

16. Analyse der Ampelstadt

Die Analyse umfaßt die funktionalen Anforderungen an die Ampelstadt und einen Entwurf für die Benutzerschnittstelle.

16.1. Allgemeiner Teil des Editors

Autor: *Jens Schröder*
Sebastian Schütte
Martin Uebing

In diesem Abschnitt beschreiben wir die im Editor zur Verfügung stehenden Aktionen und die Anforderungen, die er an die zugrundeliegende Datenbank stellt.

16.1.1. Funktionalität

Um uns mit den neuen Technologien vertraut zu machen, entwickeln wir zuerst einen Prototypen eines graphischen Editors. Er soll eine Ampelstadt modellieren können.

Die Welt der Ampelstadt besteht aus Kreuzungen und Straßen. Die Kreuzungen sind attributiert mit einen Ampelzustand und einer Eigenschaft 'Stadttor'. Kreuzungen werden Straßen verbunden. Kreuzungen und Straßen werden zu Städten zusammengefaßt, deren Stadttore ebenfalls durch Straßen verbunden sind.

Man hat zwei Sichten auf die Ampelstadt. Einmal eine Stadtsicht, in der eine Stadt mit ihren Kreuzungen und Straßen angezeigt wird, und eine Weltsicht, in der nur die Städte als ganzes und die Straßen zwischen den Städten zu sehen ist.

Der Editor soll folgende Funktionalität besitzen:

- Der Benutzer muß im einem lokalen Arbeitsbereich eine neue Stadt anlegen und sie aus Kreuzungen, Stadttoren und Straßen zusammensetzen können. Er muß die Möglichkeit haben, eine unvollständige Stadt abspeichern zu können.
- Eine konsistente Stadt, also eine Stadt deren Graph aus Kreuzungen und Straßen stark zusammenhängend ist, kann in die Welt eingecheckt werden.
- Eine einmal eingecheckte Stadt kann nie mehr editiert werden, sie kann nur als ganzes gelöscht werden.
- Auf der Weltsicht können die Stadttore der Städte durch Straßen miteinander verbunden werden. Dabei muß eine Straße entweder von einer eigenen Stadt ausgehen oder in einer eigenen Stadt enden.

16.1.2. Anforderungen an die Datenbank

Alle Änderungen, die der Benutzer vornimmt, werden direkt in der Datenbank vorgenommen. Werden Änderungen in der Datenbank wirksam und andere Benutzer sind davon betroffen, so werden sie mit dem Notifikation von H-PCTE darüber informiert. Dem liegt das Model-View-Controller-Konzept zugrunde.

Im folgenden werden die Anforderungen an die Schnittstellen zur Datenbank beschrieben.

- Methoden zum Anlegen und zum Löschen von Objekten
  - create/delete City
  - create/delete Crossing
  - create/delete Gate
  - create/delete Road
  - create/delete OneWay
- Methoden für das erzeugen einer Sicht. Es werden Listen von Objekten zurückgegeben, um die graphische Visualisierung zu ermöglichen.
  - getCityList
  - getGateList
  - getCrossingList
  - getRoadList
  - getOneWayList
- Methoden zur Zustandsänderung.
  - checkIn
    Die Methode checkt eine Stadt in die Weltsicht ein. H-PCTE ist hierbei für den Konsitenzcheck verantwortlich.
  - setLights Die Methode setzt Ampelattribute.
- Abfragemethoden
  - allowedRoad
    Eine Straße ist nur dann erlaubt, wenn die Kreuzungen bisher noch nicht verbunden sind.
  - allowedOneWay
    Eine Einbahnstraße ist nur dann erlaubt, wenn die Kreuzungen bisher noch nicht in Richtung der Einbahnstraße verbunden sind.
  - getOwner
    Liefert den Erbauer einer Stadt.

  Benachrichtigungen von der Datenbank, auf die der Editor reagieren muß, werden hier noch nicht berücksichtigt, da zum Zeitpunkt der Textlegung der Vortag über den Notifikationmechanismus in H-PCTE noch nicht gehalten wurde.

16.2. Analyse der Benutzungsoberfläche (Spezielle Editorengruppe)

Autoren: *Adil Kassabi*
Sascha Lüdecke
Andreas Schröder

Dieser Abschnitt soll verdeutlichen, wie die Benutzungsoberfläche entstanden ist. Dabei wurde analysiert, was der Bentuzer tun muß, um mit dem Tool zu arbeiten. Während dieser Analyse wurden dann die verschiedenen Fentstertypen entwickelt und graphisch festgehalten.

16.2.1. Entwurf der grafischen Oberfläche des Editors

Bei der Analyse des Problems haben wir im wesentlichen die notwendige Funktionaltät der Benutzeroberfläche herausgearbeitet. Diese ergibt fünf verschiedene Fenstertypen für die einzelnen Arbeitsabläufe. Der Benutzer meldet sich zunächst beim System an und erhält dann eine Sicht der Welt. Von dort aus kann er Städte ansehen und seinen privaten Bereich betreten, der noch nicht fertiggestellte Städte enthält.

1. **LoginWindow** für das Einloggen als Benutzer
2. **WorldView** für die globale Sicht aller Städte
3. **OwnCities** für das Anzeigen noch nicht eingecheckter Städte
4. **ViewCity** für das Anzeigen eingecheckter Städte zur Ansicht
5. **EditCity** für das Editieren von neuen Städten

16.2.1.1. Wege in der GUI

Zuerst erscheint der Login-Screen. Nach erfolgreichem Anmelden dann die Weltansicht. In dieser können durch Click auf "Eigenen Städte" ein Fenster "OwnCities" zur Auswahl der Bearbeitung von noch nicht eingecheckten Städten geöffnet werden. Von diesem Fenster kommt man nach der Auswahl einer Stadt zu dem Fenster "EditCity" in dem man eine Stadt editieren kann. Möchte man weitere Städte editieren so kann man dies durch Auswahl eines Städtenamens im Fenster "OwnCities" tun. Innerhalb dieses Fenstes kann man die Stadt über Close im möglicherweise inkonsistenten Zustand in die Liste des Fensters "EigenenStädte" einfügen. Befindet man sich innerhalb der Weltsicht, so kann man durch Auswahl einer Stadt diese in einem Fenster "ViewCity" zur Ansicht öffnen.

Die Fenstertypen "ViewCity" und "EditCity" können beliebig oft geöffnet werden.

16.2.2. Das Fenster "LoginWindow"

Der erste Fenstertyp soll schlicht aus Login-Zeile und Password-Zeile bestehen. Er ermöglicht dem Benutzer das Einloggen in das System, um mit dem Editor auf der H-PCTE-Datenbank zu arbeiten. Damit wissen wir welche Städte unsere Städte sind.

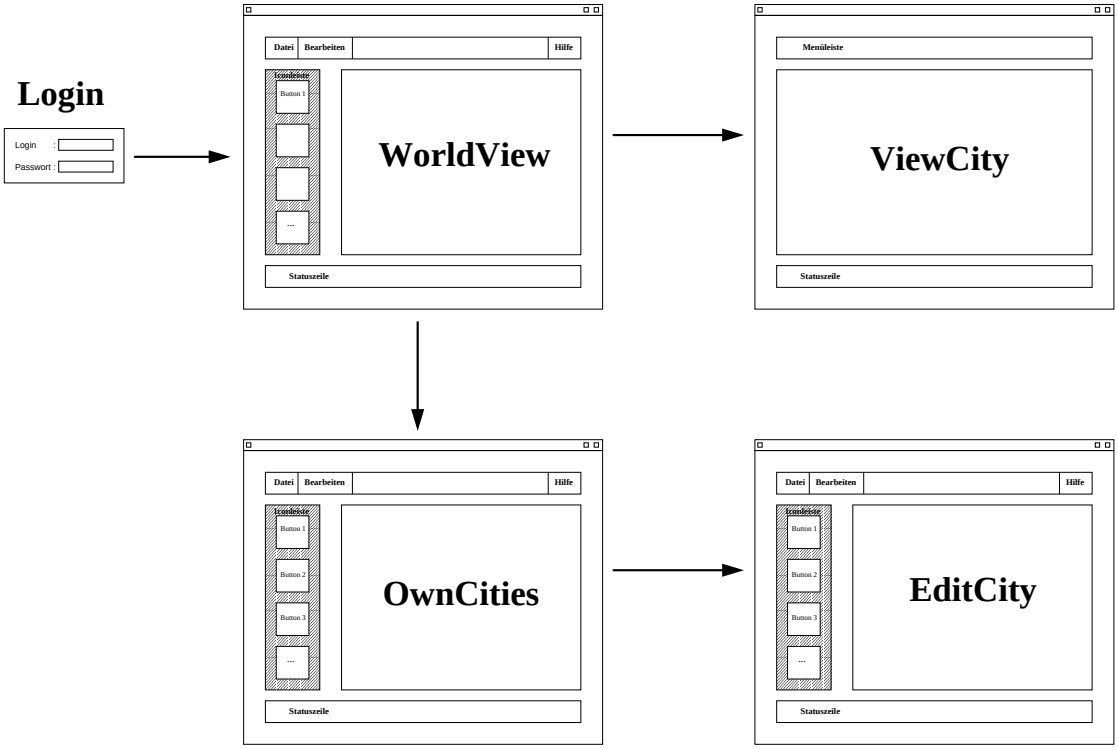


Abbildung 16.1.: GUI Übersicht

16.2.3. Das Fenster "WorldView"

Der zweite Fenstertyp soll direkt nach dem erfolgreichen Einloggen erscheinen und die Welt anzeigen. Hier kann der Benutzer neue Städteverbindungen anlegen, Städte ansehen, eigene Städte löschen und ein Fenster mit seinem privaten Bereich öffnen (Siehe Abbildung 16.2).

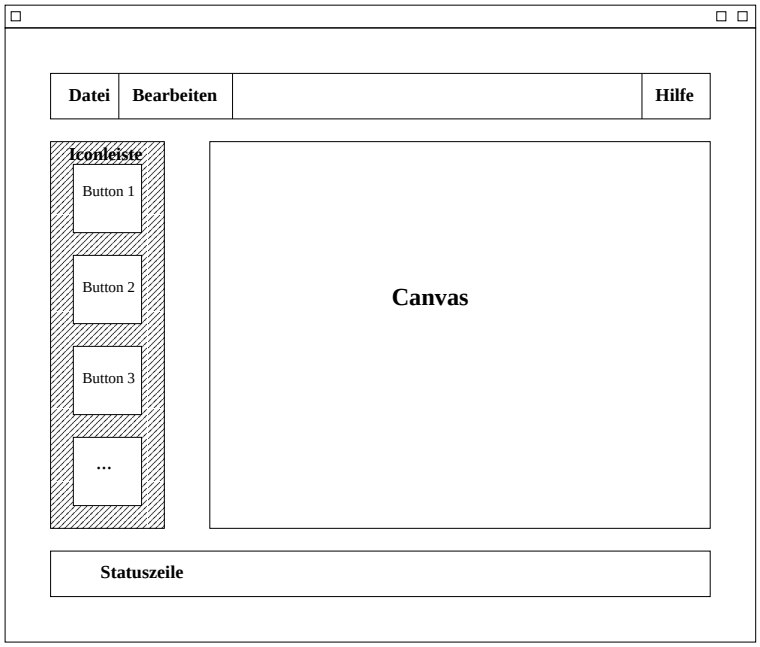


Abbildung 16.2.: Weltsicht "WorldView"

Bestandteile:

- Menüleiste: (oben)
  - Datei (Beenden)
  - Bearbeiten (Stadt löschen, Städteverbindung, Stadt ansehen, Städte in Arbeit ansehen)
  - Hilfe (Über, ...)
- Statuszeile: (unten)
  Gibt Hinweise an den Benutzer über den Status von Operationen (z.B. Quelle oder Ziel anwählen beim Anlegen einer neuen Straße, Konsistenz wird überprüft, ...)
- Iconleiste: (links)
  Menüpunkte von Bearbeiten (partiell) als Buttons (mit Icon) erfaßt für schnellen Zugriff auf die Funktionen

- Canvas: (rechts)
  Für die Weltansicht

16.2.4. Das Fenster "OwnCities"

Dieser Fenstertyp stellt den privaten Bereich des Benuzters dar. Hier befinden sich noch nicht veröffentlichte und unfertige Städte. Der Benutzer kann neue Städte anlegen, eine Stadt bearbeiten, löschen und veröffentlichen (evtl. per Drag and Drop in das Fenster "WorldView", die Stadt bewegt sich von "OwnCities" nach "WorldView"). Das Fenster hat das gleiche Layout wie das der Weltsicht (Abbildung 16.2 auf der vorherigen Seite), unterscheidet sich aber in der Buttonleiste und den Menüeinträgen unter "Bearbeiten". Der Benutzer kommt von hier in Fenster des Typs "EditCity" (Siehe Abschnitt 16.2.6).

16.2.5. Das Fenster "ViewCity"

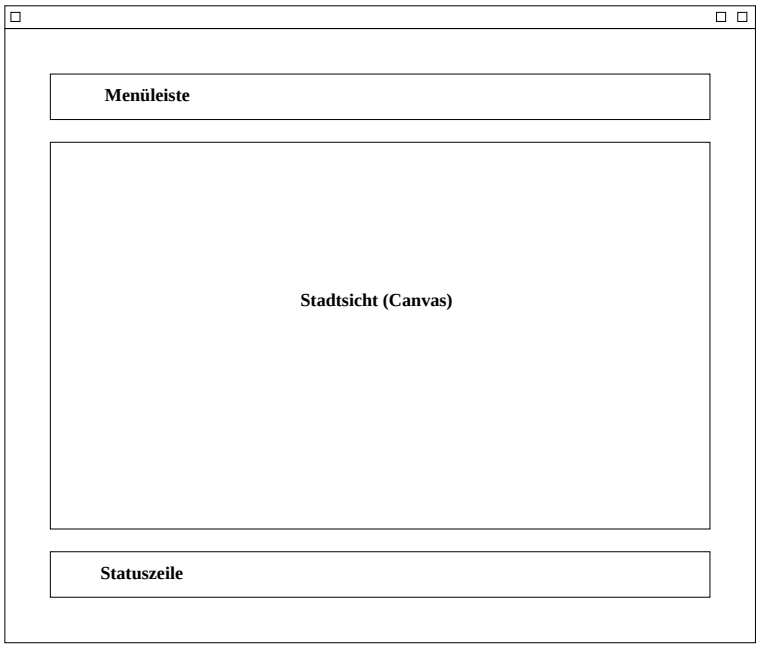


Abbildung 16.3.: Stadt ansehen "ViewCity"

Dieser Fenstertyp soll ausschließlich der Darstellung einer Stadt dienen um sich als Benutzer innerhalb der Weltsicht einzelne Städte anzeigen zu lassen. Er kann mehrere Fenster von diesem Typ öffnen (siehe auch Abbildung 16.3).

16.2.6. Das Fenster "EditCity"

Dieser Fenstertyp erscheint sobald eine neue Stadt angelegt wird oder fertiggestellt werden soll (erreichbar von Fenster "OwnCities"). Dieses Fenster hat das gleiche Layout wie

die Weltsicht, unterscheidet sich jedoch im Menü "Bearbeiten" und in der Iconleiste auf der linken Seite (siehe Abbildung 16.4). An dieser Stelle sollen folgende Operationen möglich sein:

- Neue Kreuzung
- Neues Stadttor
- Neue Straße
- Element löschen
- Änderungen rückgängig machen
- Schließen

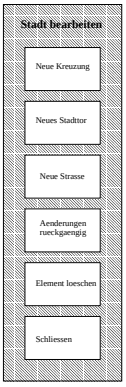


Abbildung 16.4.: Iconleiste beim Bearbeiten von Städten

Dieser Fenstertyp kann beliebig oft geöffnet werden, falls der Benutzer mehrere Städte gleichzeitig erzeugen bzw. bearbeiten will. Er kann das Fenster auch dann schließen, wenn die Stadt noch nicht konsistent ist, um seine Arbeit später fortzusetzen. Dem Benutzer wird im "OwnCities" die Möglichkeit gegeben aus der Menge seiner noch nicht eingecheckten Städten eine Stadt auszuwählen und diese zu veröffentlichen (einchecken).

16.2.7. Sonstige Anmerkungen und Gedanken

An dieser Stelle führen wir einige Eingenschaften der Oberfläche an, die uns aufgefallen sind und implementiert werden sollten:

- die Objekte in dem Canvas dienen als Repräsentanten für die Objekte der H-PCTE-Datenbank (MVC-Konzept)
- Beim Anwählen eines Objektes auf dem Canvas kann mit der rechten Maustaste die Eigenschaften des Objektes verändert werden (z.B.: bei einer Kreuzung den Ampelstatus, ...)

- Das Fenster mit "Worldview" und mit "OwnCities" sollen in den Vordergrund geholt werden beim Schliessen von "EditCity"
- kein CheckIn für:
  - inkonsistente Städte
  - Städte in Bearbeitung, diese über Drag und Drop von "OwnCities" nach "WorldView".
- Strassen in Canvas als gerade Verbindungen darstellen (bis jetzt). Erweiterung auf Polygone in stäteren Versionen
- D'n'D - Button für neue Kreuzung/Stadttor
- Konsistenz einer Stadt: Es muß mindestens 1 Stadttor geben

17. Entwurfsdokument für die Ampelstadt

Autoren: *Jens Schröder*
Sebastian Schütte
Sascha Lüdecke

Die Autoren gehen im folgenden davon aus, daß die Problemstellung der Ampelstadt bekannt ist. Ansonsten sei auf den Abschnitt 16 auf Seite 133, Analyse des Prototyps, verwiesen.

Die Anwendung zerfällt in zwei Teile, einen Datenbank-Teil, der das Datenmodell realisiert und einen Editor, der die graphische Repräsentation und Benutzeroberflächer zur Ampelstadt übernimmt.

Die Datenbank ist in zwei Bereiche unterteilt, einen öffentlichen und mehrere private. Sowohl die Stadt als auch die Welt stellen einen Graphen dar. Hierbei entsprechen Städte und Kreuzungen den Knoten, Straßen den Kanten des Graphen.

Im öffentlichen Bereich wird der Graph abgelegt, der von allen Benutzern gemeinsam bearbeitet wird. Er stellt im wesentlichen die Weltsicht dar. Jeder Benutzer kann seine eigenen Städte erstellen. Die Graphen dieser Städte werden in privaten Bereichen für jeden Benutzer getrennt abgelegt. Werden sie in der Weltsicht bekannt gemacht, wird jeweils der Stadtgraph in den gemeinsamen Weltgraphen eingehängt.

Bei näherer Betrachtung erweist sich das Editieren in der Welt- und in der Stadtsicht als sehr ähnlich. Aus diesen Grund haben wir uns entschieden, einen generischen Editor mit Spezialisierungen zu entwerfen. Durch diese Einteilung soll auch die Wiederverwendbarkeit des Editor für *DoDL* erhöht werden.

17.1. Die Schnittstelle zur Objektbank

Autoren: *Sebastian Linz*
Thomas Sparenberg
Christian Stücke

Die Bausteine von Ampelstädten sind Städte, Kreuzungen und Strassen. Diese Bausteine können als komplexe Datentypen auf benutzerdefinierte Objekte des *Object-Management-System* (OMS) HPcte abgebildet und gespeichert werden. Der Entwurf sieht

eine Schnittstelle zum OMS vor, damit Editoren nicht direkt auf die Objektbank zugreifen müssen. Desweiteren ermöglicht dieser Entwurf die Integration des *Modell-View-Controller*-Konzepts 33 auf Seite 345.

Die Schnittstelle zur Objektbank setzt sich aus Klassen zusammen, die die Verbindung und den Zugriff auf die Daten im OMS ermöglichen und für Editorklassen die notwendigen Operationen zur Verfügung stellen. Städte, Kreuzungen und Strassen sind beispielhaft für elementare Objekte einer Ampelstadt. Sie werden in einer Klasse als komplexer Datentyp realisiert, der die Operationen integriert, um an den Attributen Veränderungen vornehmen zu können oder ein solches Objekt neu anlegen zu können. In der GUI findet man Editoren, die, je nach Blickwinkel, die Ampelstadtwelt oder eine detaillierte Ampelstadt darstellen können. Für die Editoren muß die OMS-Schnittstelle Operationen und Datentypen bereitstellen, um Mengen von Ampelstadtbausteinen verwalten zu können. Dies wird durch die OMS-Container-Klassen realisiert.

Das *Modell-View-Controller*-Konzept wird in der OMS-Schnittstelle implementiert. Klassen, die Daten des OMS darstellen, melden sich bei Notifizierern an, um bei Änderungen benachrichtigt zu werden. Diese Änderungen werden dem entsprechenden Objekt der Benutzeroberfläche mitgeteilt. Notifizierer werden hier als Objekte verstanden, die eine Ressource des OMS überwachen und alle Klassen benachrichtigen, die sich für die Überwachung dieser Ressource angemeldet haben.

17.1.1. `OMSMain`

Jede Ampelstadt-Applikation sollte genau eine Instanz dieser Klasse haben. Der Konstruktoraufruf erfolgt mit einem Benutzernamen und einer Kennung, sofern Name und Kennung korrekt sind. Instanzen dieser Klasse stellen die Verbindung zum OMS her und ermöglichen es `PcteProcess`-Objekte zu erzeugen. Erzeugt die zugrundeliegende Objektbankschnittstelle `OMSExceptions`, so werden diese an die aufrufende Instanz weitergereicht.

17.1.2. `OMSContainer`

Der `OMSContainer` ist das Pendant zu einem `DrContainer` und hält die Verbindung zur Datenbank. Dieser `OMSContainer` wird mit der Anlage eines neuen Editorfensters konstruiert. Er wird als Listener an der entsprechenden Stelle in der Datenbank angemeldet und horcht auf die dort getätigten Veränderungen. Er ist insbesondere für das Anlegen neuer Knoten/Kanten zuständig, weil diese ihrerseits noch nirgendwo als Listener angemeldet sein können. Hierbei sorgt er für die nötigen Verbindungen und Anmeldevorgänge. Und das nicht nur, wenn er den direkten Methodenaufruf zum Erstellen bekommt, sondern auch, wenn er durch Horchen davon erfährt, daß ein neues Element von einem anderen OMSContainer angelegt wurde. Dadurch werden alle Fenster, die mit der Datenbank arbeiten und diesen Fensterausschnitt darstellen, auf einmal aktualisiert. Des weiteren kann er eine Liste aller Kanten und Knoten liefern, die zu einer "Sicht" gehören (Stadtsicht, Weltsicht). Das Löschen, Positionieren oder Verändern der `OMSNode` und `OMSEdge` ist für den `OMSContainer` eher uninteressant.

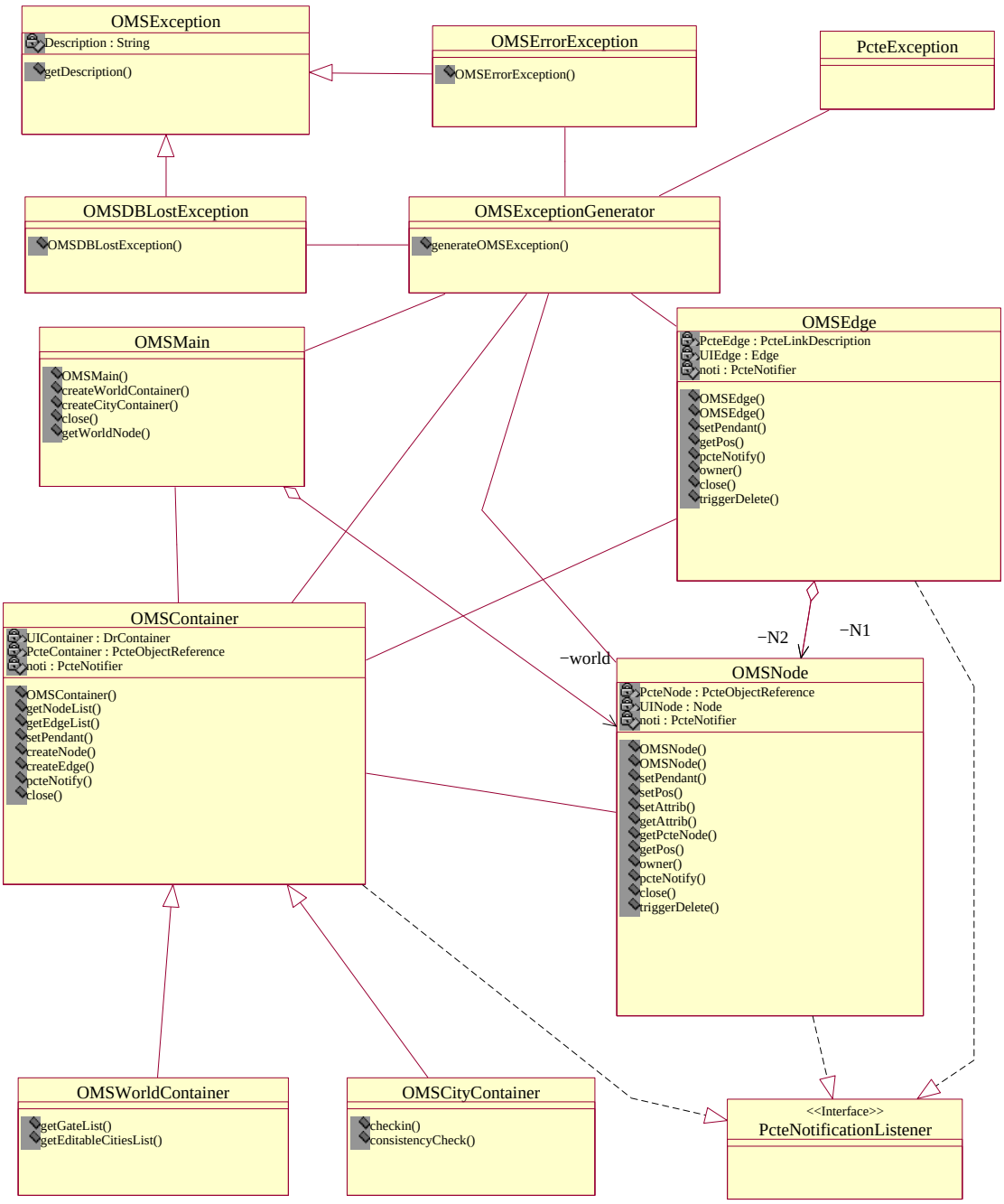


Abbildung 17.1.: Klassendiagramm-OMS

17.1.3. OMSWorldContainer

Der OMSWorldContainer ist eine Subklasse von OMSContainer. Er spezialisiert die Fähigkeiten des OMSContainer auf die Repräsentation einer "Weltansicht". Des weiteren kann er Listen von Stadttoren aller eingecheckten Städte und Listen aller editierbaren Städte liefern.

17.1.4. OMSCityContainer

Der OMSCityContainer ist eine Subklasse von OMSContainer. Er spezialisiert die Fähigkeiten des OMSContainer auf die Repräsentation der "Stadtansicht". Er übernimmt die Konsistenzprüfung einer neu angelegten Stadt und das einchecken einer neu angelegten, konsistenten Stadt.

17.1.5. OMSNode

Über Instanzen dieser Klasse werden Welten, Städte und Kreuzungen auf die Objektbank abgebildet. Jede Instanz dieser Klasse besitzt ein Pendant der Editorenschicht, dem dadurch der Zugriff auf das Objekt-Management-System ermöglicht wird. Treten bei Zugriffen Fehler auf, so werden entsprechende OMSExceptions erzeugt. Die Klasse implementiert die Schnittstelle PcteNotificationListener mit den zugehörigen Methoden. Die Klasse enthält ein Attribut, das ein Notifizierer-Objekt ist. Bei diesem Notifizierer kann sich die Instanz dieser Klasse selbst anmelden, so daß Manipulationen an einer bewachteten Ressource den Notifikationsmechanismus auslösen.

17.1.6. OMSEdge

Die Klasse OMSEdge bildet das Datenbank-Gegenstück zu einer Kante, also einer Straße im Editor. Jede Instanz dieser Klasse hat somit genau einer Instanz der Klasse DrEdge auf der Editor- seite, durch die sie im Editor repräsentiert wird. Die Klasse OMSEdge besitzt zwei Konstruktoren: Der erste Konstruktor wird von einem OMSContainer aufgerufen, wenn er die Nachricht bekommt, daß eine neue Straße angelegt weden soll. Dem Konstruktor von OMSEdge werden dann die beiden Endpunkte der Straße (in Form von OMSNodes) und der Typ der anzulegenden Kante (Einbahn- oder Zweibahnstraße) übergeben. Für den Typen einer Straße sind in der Klasse zwei Konstanten namens EINBAHN und ZWEIBAHN definiert. Der Konstruktor legt nun auf Datenbankebene einen Link zwischen den beiden Knoten in der Datenbank an und setzt die Linkattribute entsprechend des Straßen typs. Der zweite Konstruktor wird benötigt, wenn nun alle betroffenen OMSContainer, d.h. alle betroffenen Editorfenster, durch den Notifikationsmechanismus erfahren, daß es einen neuen Link in der Datenbank gibt. Der Notifizierer liefert dem Container durch des mitgeschickte PcteNotificationEvent den neuen Link in Form eines PcteLinkDescriptors. Mit diesem wird nun der zweite Konstruktor von OMSEdge aufgerufen. Zusätzlich wird der aktuelle Editor- prozeß mitübergeben. Der Konstruktor legt seinerseits wieder Notifizierer auf den Link und seine Endknoten in der Datenbank an und über die Methode setPendant() wird die Verbindung zu einer Kante auf der Editorseite, also einer Instanz

der Klasse `DrEdge` aufgebaut. Es entsteht somit für jede Instanz eines Editorfenstes eine entsprechende Instanz einer Kante. Solange eine Kante existiert überwacht sie den eigenen Link in der Datenbank und die Positionen der Endknoten. Zu diesem Zweck wird also das Interface `PcteNotificationListener` implementiert. Die im Konstruktor erzeugten Notifizierer benachrichtigen nun die Instanz der Klasse `OMSEdge`, sobald Änderungen in der Datenbank gemacht werden. Eine Instanz von `OMSEdge` muß dann ensprechend in der Methode `pcteNotify` regieren, z.B. wenn sich die Koordinaten eines Endpunktes ändert muß die entsprechende Verschiebe-Methode beim Editor-Pendant, also der Instanz von `DrEdge` aufrufen. Ähnlich wie beim Erzeugen einer Kante läuft auch das Löschen einer solchen ab: Mit der Methode `triggerDelete()` wird die Kante, also der Link, aus der Datenbank gelöscht. Da alle Instanzen von `OMSEdge`, die diese Kante repräsentieren sich selbst durch Notifizierer überwachen, wird also jede betroffene Instanz durch den Notifizierungsmechanismus von der Löschung des Links benachrichtigt. Jede betroffene Instanz muß nun alle Referenzen auf Datenbankobjekte und alle Notifizierer abmelden. Dann werden sie auto- matisch durch die Garbage-Collection gelöscht.

17.1.7. `OMSExceptions`

Die Klasse `OMSExceptions` bildet die Superklasse für `OMSLostDBExceptioni` und `OMSErrorException`. Sie enthält eine Methode `getDescription`, die das Attribut `description` ausgibt. `description` enthält eine Beschribung der Exception und kann in einem Dialogfenster des Editors ausgegeben werden.

17.1.8. `OMSDBLostException`

`OMSDBLostException` werden erzeugt, wenn die Verbindung zum OMS unterbrochen ist.

17.1.9. `OMSErrorException`

`OMSErrorException` werden erzeugt, wenn Methodenaufrufe des zugrundeliegenden Java-API des OMS `PcteException` erzeugt werden.

17.1.10. `OMSExceptionGenerator`

Der `OMSExceptionGenerator` wird benutzt, sobald auf Datenbankebene eine `PcteException` geworfen wurde. Diese wird dann an den `OMSExceptionGenerator` durch die Methode `generateOMSException` weitergeleitet. In dieser Methode wird eine `OMSException` vom richtigen Typ erzeugt und an die aufrufende Klasse zurückgegeben. Diese kann dann diese `OMSException` werfen.

17.2. Entwurf des Editors

In diesem Abschnitt wird unser Entwurf des Editors und seine Einteilung in generischen und speziellen Teil beschrieben. Wir geben zunächst eine Übersicht und gehen dann de-

tailiiert auf die einzelnen Stichpunkte generischer Teil, Interaktion zwischen Objektbank und Editor sowie Verfeinerung des generischen Editors zu einem speziellen Editor, ein.

Autoren: *Sascha Lüdecke*
Jens Schröder
Sebastian Schütte

17.2.1. Gesamtsicht auf den Editor

Der Entwurf des Editors teilt sich in zwei Teile. Einmal einen generischen Teil, der den Großteil des Verhaltens des Editors festlegt, sowie einen speziellen Teil, der die konkreten Ausprägungen des Editor beschreibt.

Im generischen Editor werden nur Graphen betrachtet, die im allgemeinen aus Kanten und Knoten bestehen. Genauso beziehen sich alle angebotenen Manipulationen auf diese allgemeinen Graphen.

Im speziellen Editor hingegen wird die konkrete grafische Repräsentation der Kanten und Knoten bestimmt, sowie konkrete Typen von Knoten und Kanten. In unserer Ampelstadt gibt es die Knotentypen 'Stadt', 'Stadttor' und 'Kreuzung' und die Kantentypen 'Einbahnstraße' und 'Zweibahnstraße'.

Der generische Editor läßt sich in vier Pakete gliedern. Dabei faßt ein Paket Klassen zusammen, die von ihrer Funktionalität zusammengehören.

Eine generelle Eigenschaft des generischen Editors läßt sich aus dem Klassendiagramm 17.2 auf der nächsten Seite erahnen: Klassen, die mit der Repräsentation des Graphen zu tun haben, sind paarweise vorhanden, jeweils eine Klasse mit dem Präfix OMS und eine Klasse mit dem Präfix `Dr`. Dies läßt sich verstehen, wenn sich der Leser an das Model-View Konzept erinnert. Dabei wird ein Objekt aufgeteilt in eine Klasse, die die Repräsentation der inneren Zustände übernimmt, das Modell, und ein oder mehrerer Klassen, die ein solches Modell visualisieren, die sogenannten Beobachter.

Im Fall der Ampelstadt wird das Modell in der Datenbank H-PCTE verwaltet. Die `Dr`-Objekte sind die Beobachter dieses Modells. Objekte mit dem Präfix `OMS` (`OMSEdge`, `OMSContainer`) dienen als Schnittstelle für die Benutzung dieses Modells.

17.2.2. Generischer Editor

Der generische Editor zerfällt in die Teile Applikation, Eingabe und Statusanzeige, Kontrolle des Editors und Darstellung.

Der Applikationsteil stellt die Verbindung zur Datenbank sicher und umfaßt das allgemeine Editorobjekt, in dem alle generischen Editorelemente gehalten werden. Von hier aus startet das Objektsystem.

Der Teil Eingabe und Statusanzeige beinhaltet alle generischen Elemente die eine Interaktion des Benuzers mit dem Editor erlauben.

Im kontrollierenden Teil des Editors befindet sich die Zustandssteuerung des Editors und ein Controller, der alle Ereignisse an die Zustandssteuerung weiterleitet.

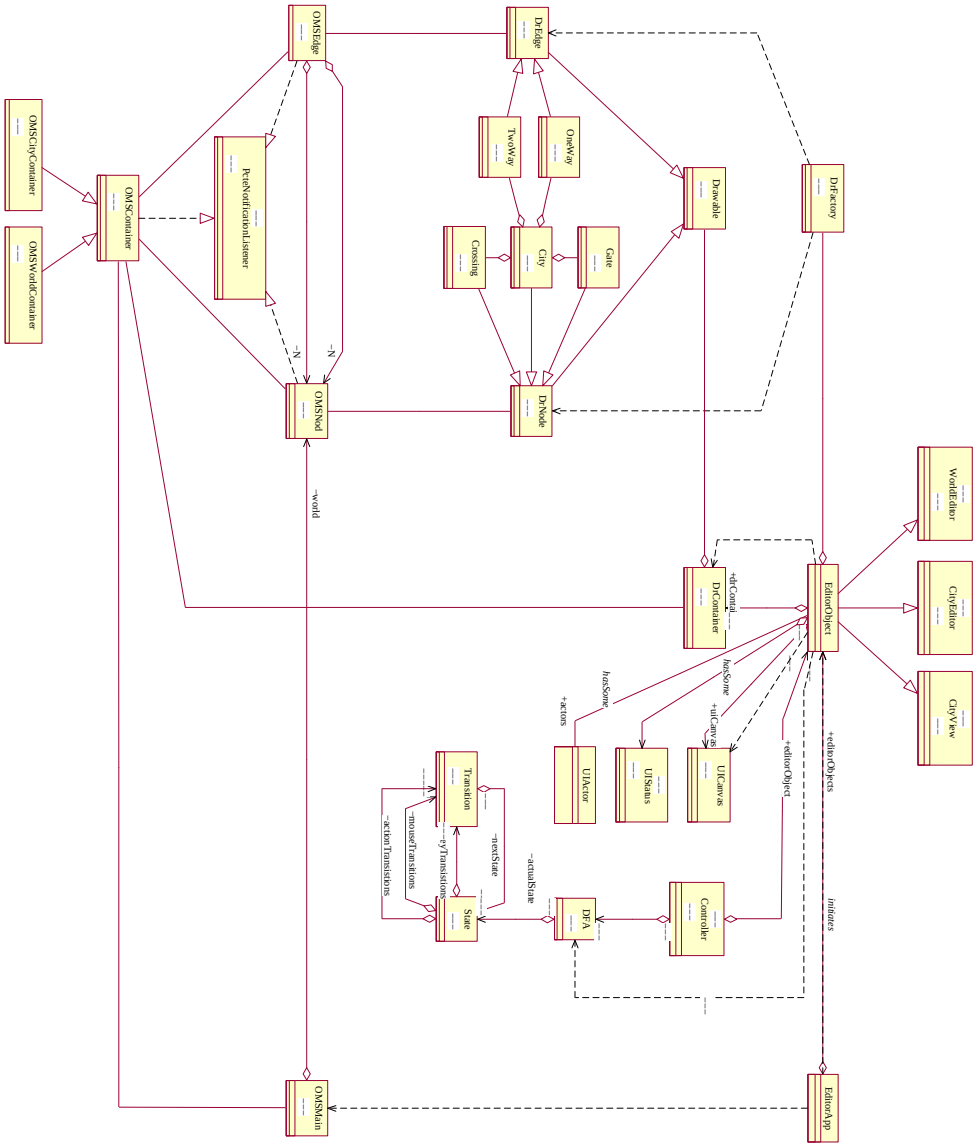


Abbildung 17.2.: Klassendiagramm des gesamten Editors

Alle Klassen, die zur Visualisierung des allgemeinen Graphen im Editor benötigt werden, sind im darstellenden Teil des Editors zusammengefaßt. Sie enthalten auch die Anbindung der darstellenden Komponenten an die Datenbank.

17.2.2.1. Applikationsteil

In Abbildung 17.3 befindet sich das Klassendiagramm mit den zentralen Objekten `EditorObject` und `EditorApp`, die hier kurz erläutert werden. Der Stereotyp `VisaJ` kennzeichnet eine mit VisaJ erstellte Klasse. `FactoryMethod` ist eine Methode nach dem Entwurfsmuster *Factory*.

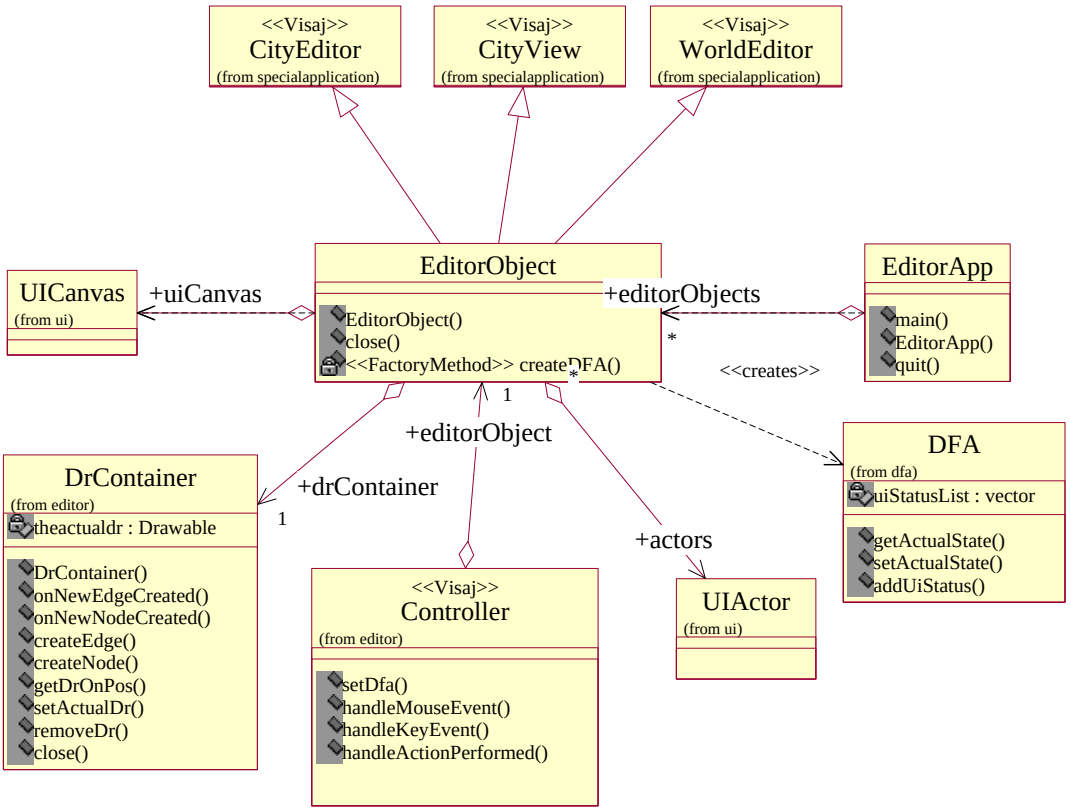


Abbildung 17.3.: Klassendiagrams des Applikationspakets

EditorObject Betrachtet man das Klassendiagramm 17.3, so fällt ins Auge, daß die Klasse `EditorObject` eine zentrale Rolle einnimmt. Ein `EditorObject` repräsentiert einen

Editor für einen Teilgraphen (in der späteren Spezialisierung also einen Welt- oder Stadteditor). Die Klasse `EditorObject` aggregiert zahlreiche weitere Klassen, die die eigentliche Programmlogik beinhalten und ist für die korrekte Erzeugung der untergeordneten Klassen zuständig.

`EditorObject` ist eine abstrakte Klasse, von der der Welteditor und der Stadteditor erben.

EditorApp Um dem Benutzer die Verwendung von mehreren Editoren gleichzeitig zu erleichtern, besitzt die Klasse `EditorApp` die Fähigkeit, mehrere EditorObjekte zu verwalten. Die Klasse `EditorApp` ist der "Applikationsrahmen" für die graphischen Editoren. Diese Klasse übernimmt die Verwaltung der Editorfenster und das Erzeugen eines neuen Editors auf Verlangen des Benutzers. Startet der Benutzer die Ampelstadt, wird eine Instanz der Klasse `EditorApp` erzeugt.

17.2.2.2. Eingabe und Statusanzeige

Interface UI-Status In Abhängigkeit vom aktuellen Zustand des Editors (siehe dazu auch 17.2.2.3 auf der nächsten Seite) soll sich der Mauszeiger verändern bzw. ein Text in der Statuszeile ausgegeben werden. Um dies zu verwirklichen, wurde das Interface `UI-Status` eingeführt. Die Klasse `DFA` teilt bei einem Zustandswechsel, durch Aufruf der showStatus-Methode mit dem Namen des Zustandes als Parameter, den angemeldeten UI-Status-Objekten mit, welcher Zustand der aktuelle ist. Beispielsweise eine Statuszeile kann `UI-Status` implementieren und in der Methode showStatus den passenden Text für die Statuszeile aus dem Namen des Zustandes bestimmen und anzeigen. Eine andere Klasse kann analog die Form des Mauszeigers ändern.

UI Action Der generische Editor wird die Klasse `Action` der Swing-Bibliothek verwenden, um aktvierte bzw. deaktivierte Befehle auf der Oberfläche sichtbar zu machen. Das `EditorObject` verwaltet eine Menge von `Action`-Objekten, die je einen Befehl, repräsentieren. Ein solcher Befehl kann aktiviert oder deaktiviert sein. Toolbars und Menuzeilen, die bei diesem Befehl als "ChangePropertyListener" registriert sind, werden über die Änderung des Zustands informiert und wissen, wie die graphische Entsprechung des `Action`-Objektes auf der Oberfläche zu zeigen ist.

Hier sind noch einige Details offen, die erst während der Implementation geklärt werden, da wenig Erfahrung im Umgang mit Java-Action Objekten besteht.

Verwaltung der Zustandsmengen Offen ist die Frage, wie die Menge der Action-Objekte verwaltet wird. Zum einen muß die Menge der verfügbaren Befehle insgesamt verwaltet werden. Darüber hinaus jedoch ist, abhängig vom aktuellen Zustand des Automaten, nur eine Teilmenge dieser Befehle aktiv, der Rest deaktiviert. Diese Teilmengen können sowohl bei den Zuständen, als auch zum Teil bei den Drawables gespeichert werden. Eine mögliche Realisierung ist die Speicherung der Action-Objekte in einer Hashtable mit den Teilmengen als Bitvektoren, wobei jedem Action-Objekt eine Position auf dem Bitvektor zugewiesen wird.

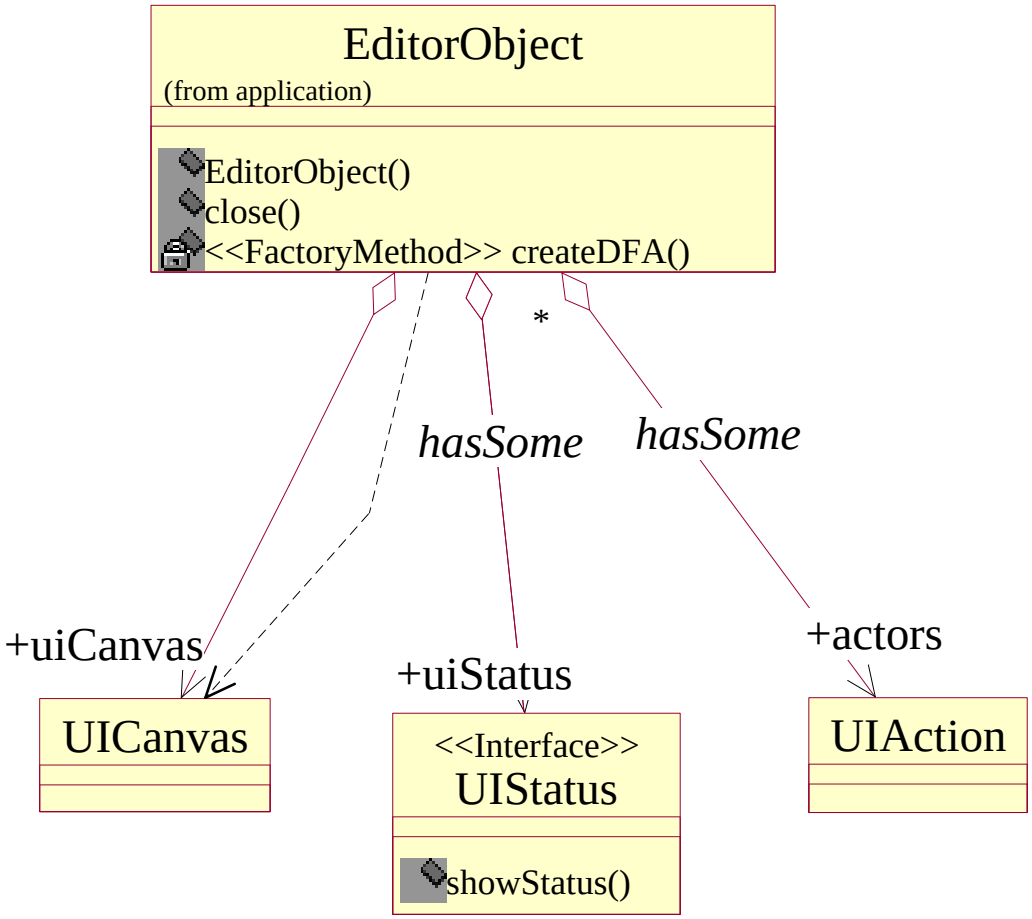


Abbildung 17.4.: Eingabe und Statusainzeige

17.2.2.3. Steuernder Teil des Editors

Das Zustandsdiagramm des generischen Editors Im folgenden wird häufig das Wort *Ereignis* verwendet. Aus der Sicht der Objekte des Editors sind Benutzereingaben Ereignisobjekte (Mausklicks, Tastatureingaben, ...), die zur Verarbeitung übergeben werden.

Der generische Editor besitzt Zustände, die sein Verhalten auf Benutzereingaben bestimmen. Ein Benutzer erwartet, daß nichts passiert, wenn er in einen freien Bereich der Zeichenfläche klickt. Möchte er jedoch einen neuen Knoten erzeugen, wählt er einen entsprechenden Menupunkt o.ä. aus und klickt in die freie Zeichenfläche, worauf an dieser Stelle ein neuer Knoten angezeigt wird.

Ein naiver Ansatz, um dieses Verhalten des Editors zu erhalten, wäre, nach der Wahl des Menupunktes auf das nächste Mausereignis zu warten. Es liegt jedoch in der Natur

ereignisoriertierter Oberflächen, daß ein solcher Ansatz nicht funktionieren kann: Damit die Oberfläche wieder auf den Benutzer reagieren kann, müssen Methoden, die aufgrund bestimmter Ereignisse ausgeführt werden, terminieren. Folglich muß an geeigneter Stelle vermerkt werden, daß das nächste Mausereignis zur Neuerzeugung eines Knotens führen soll.

Um die Vielzahl der möglichen Zustände mit ihren Übergängen zu verwalten, bietet sich ein endlicher Automat an. Die Zustände dieses Automaten bezeichnen Zustände des Editors (z.B. "Knoten selektiert", "Neuer Knoten", ...), während das Eingabealphabet aus der Menge der möglichen Ereignisse besteht.

Als Notation für den Zustandsautomaten, der dem generischen Editor zugrunde liegt, wurde ein UML-Zustandsdiagramm gewählt. Mit diesen Diagrammen lassen sich endliche Automaten beschreiben. Zusätzlich bietet diese Notation zwei Erweiterungen, die die Zahl der gezeichneten Zustandsübergänge verringert.

- Zustände können andere Zustände beinhalten. Wird von einem solchen umschließenden Zustand eine ausgehende Kante gezeichnet, bedeutet dies, daß alle enthaltenen Zustände einen entsprechenden Zustandsübergang besitzen.
- Zustandsübergänge können Bedingungen enthalten. Ein Übergang findet nur statt, wenn die Bedingung als *wahr* ausgewertet wird.

Die Zustandsübergänge sind wie folgt gekennzeichnet:

```
Ereignis [Bedingung] / Aktion
```

Realisierung des Zustandsdiagrammes Während der Entwicklung des Entwurfes wurden verschiedene Möglichkeiten gefunden, wie obiger Zustandsautomat zu realisieren ist.

1. *Matrixdarstellung* Eine $m \times n$ Matrix M mit m = Anzahl Zustände und n = Anzahl Ereignisse kann als Repräsentation gewählt werden. Im Eintrag $M_{Ereignis,Zustand}$ kann ein Objekt gespeichert sein, das mögliche Bedingungen überprüft und die geeignete Aktion ausführt.
2. *State-Pattern* In Anlehnung an das State-Pattern aus [GHJV96] können die Zustände als Objekte verwaltet werden, die eine gemeinsame Superklasse haben, die abstrakte Methoden für die Verarbeitung von bestimmten Ereignistypen besitzt. Die konkreten Zustände implementieren diese Methoden, innerhalb einer solchen Methode befindet sich eine große Fallunterscheidung. Für jeden Zustandsübergang muß ein Fall vorgesehen sein.
3. *Zustände & Übergänge als Objekte* Ähnlich wie in Punkt 2 werden die Zustände als Objekte realisiert. Diese Zustandsobjekte verwalten eine Menge von Übergangsobjekten, im folgenden Transistionen genannt. Das dem Zustand übergebene Ereignis wird der Reihe nach jeder dieser Transitionen übergeben, die es entweder verarbeiten oder ablehenen (siehe auch "Chain of Responsibility" [GHJV96]).

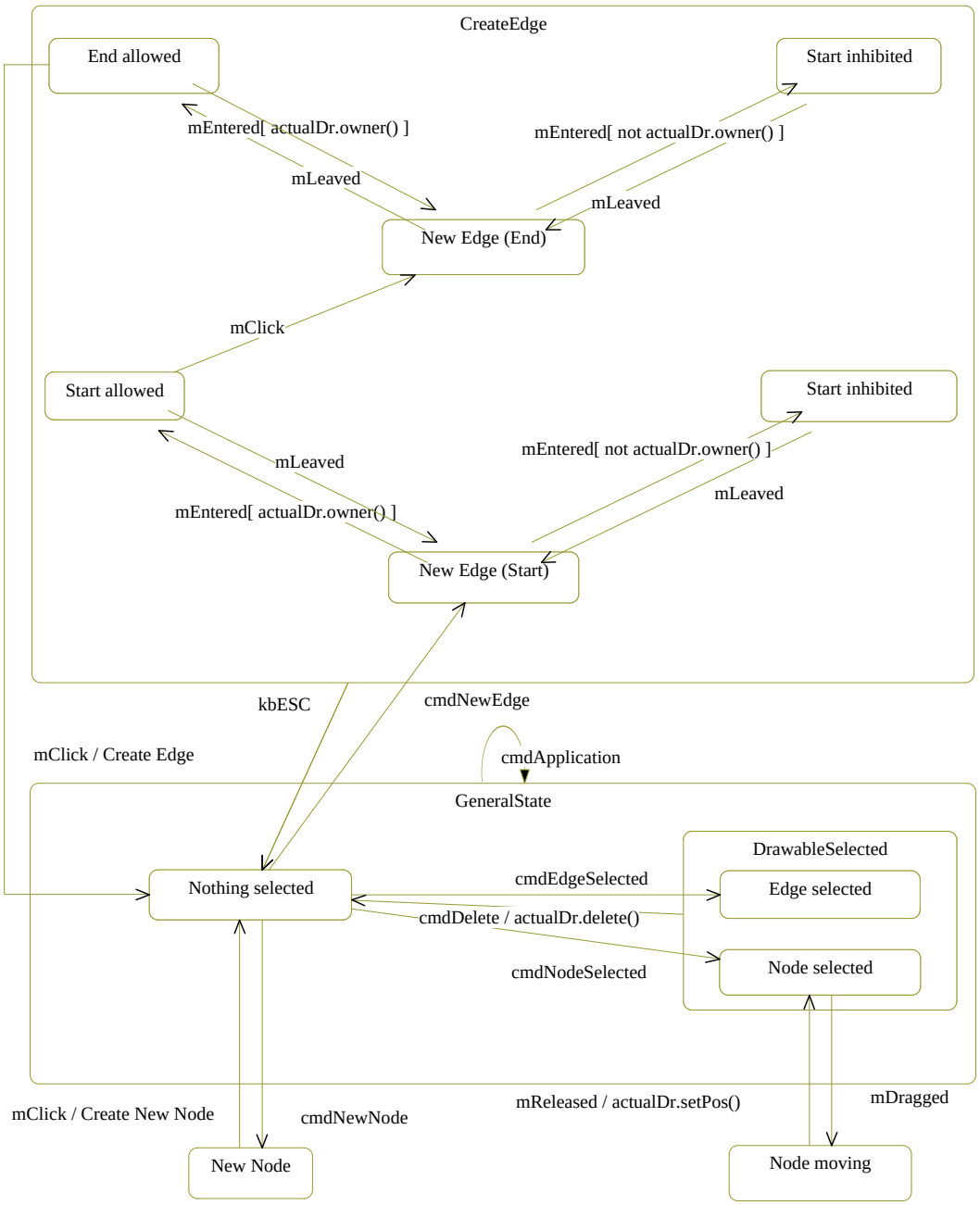


Abbildung 17.5.: Zustandsdiagramm

Variante 1 auf Seite 151
: wurde verworfen, da die große Zahl an Ereignissen eine große, evtl. spärlich besetzte Matrix zur Folge hätte. Darüberhinaus wurde diese Variante als sehr unübersichtlich empfunden.

Variante 3 auf Seite 151
: wurde Variante 2 auf Seite 151 vorgezogen, da eine große Fallunterscheidung unübersichtlich und unhandlich ist. Soll dem Zustandsautomaten ein neuer Übergang hinzugefügt werden, so wird in Variante 3 auf Seite 151 eine neue Transistionsklasse erzeugt, der übrige Code kann unverändert bleiben, während bei Variante 2 auf Seite 151 dem existierenden Code neue Zeilen hinzugefügt werden müssen.
Zusätzlich bietet Variante 3 auf Seite 151 den Vorteil, daß einem existierenden Automaten (evtl. aus einer serialisierten Datei eingelesen) sogar zur Laufzeit neue Übergänge hinzugefügt werden können.

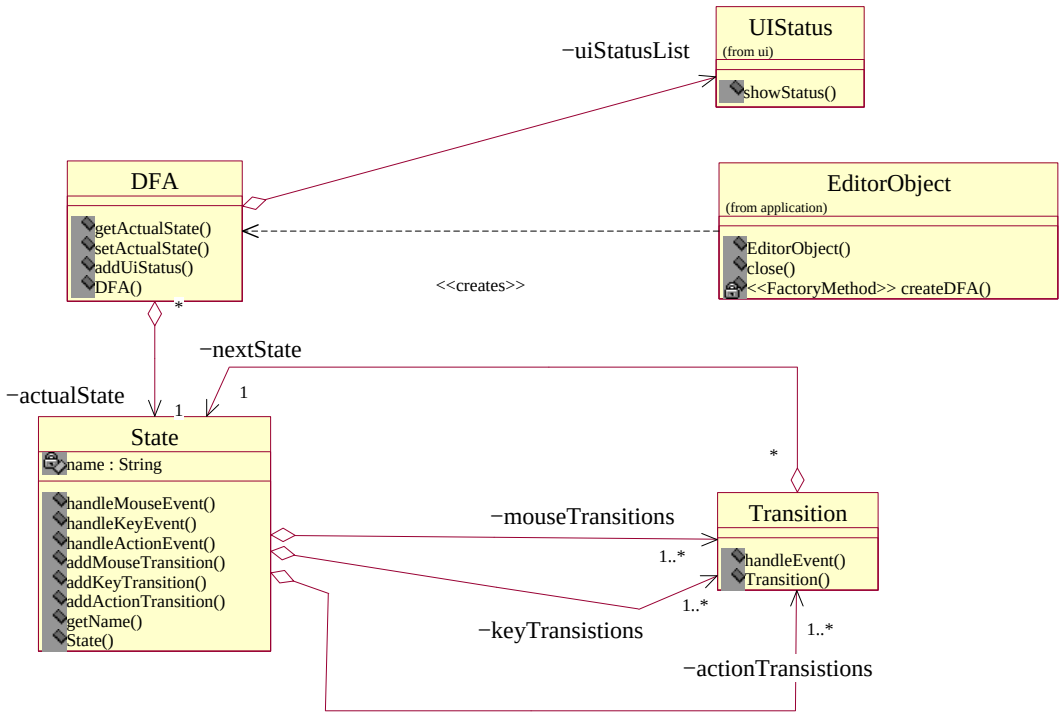


Abbildung 17.6.: Klassendiagramm des DFA-paketes

State Eine State-Klasse verwaltet eine Menge von Transitionen. Wird einem Zustand ein Ereignis zur Verarbeitung übergeben, wird der Reihe nach die handleEvent-Methode der registrierten Transitionen aufgerufen. Sofern eine Transition ein Ereignis verarbeitet,

gibt sie den neuen Zustand zurück, ansonsten einen Null-Zeiger. Es ist sichergestellt, daß entweder eine oder keine, aber niemals zwei Transitionen ein Ereignis verarbeiten.

Aus Performanzgründen werden drei Listen verwaltet, für Maus-, Tastatur- und Befehlsereignisse.

Transition Eine Transition besitzt einen Nachfolgezustand sowie eine abstrakte Methode handleEvent. Diese Methode ist in den Klassen, die von Transition erben, zu realisieren. Innerhalb dieser Methode muß entschieden werden, ob das übergebene Ereignis zu verarbeiten ist oder nicht, sowie ob die Bedingung erfüllt ist. Darüberhinaus steht innerhalb dieser Methode der Code, der ausgeführt werden muß, um beispielsweise einen neuen Knoten zu erzeugen.

Das Transitionsobjekt könnte auch drei Schablonenmethoden vorsehen: Event, Condition, Action, die dann alle drei gefüllt werden müssen. Die Trennung ist jedoch nicht immer vorteilhaft, da in dem vorliegenden Zustandsdiagramm oftmals, abhängig, ob die Bedingung als Wahr oder Falsch ausgewertet wird, eine von zwei Alternativen gewählt werden muß. Verwendet man Schablonenmethoden, müßte man in einem solchen Fall zwei Transitionen vorsehen, was sonst in einer handleEvent-Methode erledigt werden könnte.

Da die handleEvent-Methoden i.d.R. nur aus wenigen Methodenaufrufen anderer Objekte des generischen Editors besteht, ist die Aufteilung auf drei Schablonenmethoden nicht notwendig.

DFA Über den DFA ist der aktuelle Zustand zugreifbar. Man könnte erwarten, daß der DFA eine Menge von Zuständen speichert. Dies ist jedoch bei näherer Betrachtung nicht erforderlich. Der Zustandsgraph ist zusammenhängend, über Zustandsübergänge sind alle Zustände zu erreichen.

17.2.2.4. Darstellender Teil des Editors

Drawable, DrNode, DrEdge Der generische Editor erlaubt die Erstellung und Manipulation eines Graphen. Die Klasse Drawable ist die gemeinsame Oberklasse für Node und Edge. Wie die Namen suggerieren, repräsentieren diese Klassen einzelne Elemente des zu bearbeitenden Graphen (die Ausprägungen des Editors werden von diesen Klassen erben, um die Visualisierung anpassen zu können).

Für das Verständnis der Methodensignaturen dieser Klassen ist es hilfreich, sich an das MVC-Konzept zu erinnern. Die Klasse Drawable stellt einen Beobachter dar. Sie verwendet ein Objekt vom Typ OMSNode bzw. OMSEdge als Schnittstelle, um auf das zugehörgie Modell in der Datenbank zuzugreifen.

Beim Entwurf der Ampelstadt erschien es sinnvoll, die Existenz dieser Schnittstellenobjekte nur der Klasse Drawable bekannt zu machen, um die Datenbank weitestgehend zu verstecken. Alle übrigen Klassen des Editors rufen Methoden des Beobachters auf, um bestimmte Änderungen vorzunehmen oder ein Objekt zu löschen. Diese Methoden stoßen über den Aufruf geeigneter Methoden der Schnittstellenobjekte die Änderung im Modell in der Datenbank an. Auf diese Weise läßt sich verstehen, warum viele Metho-

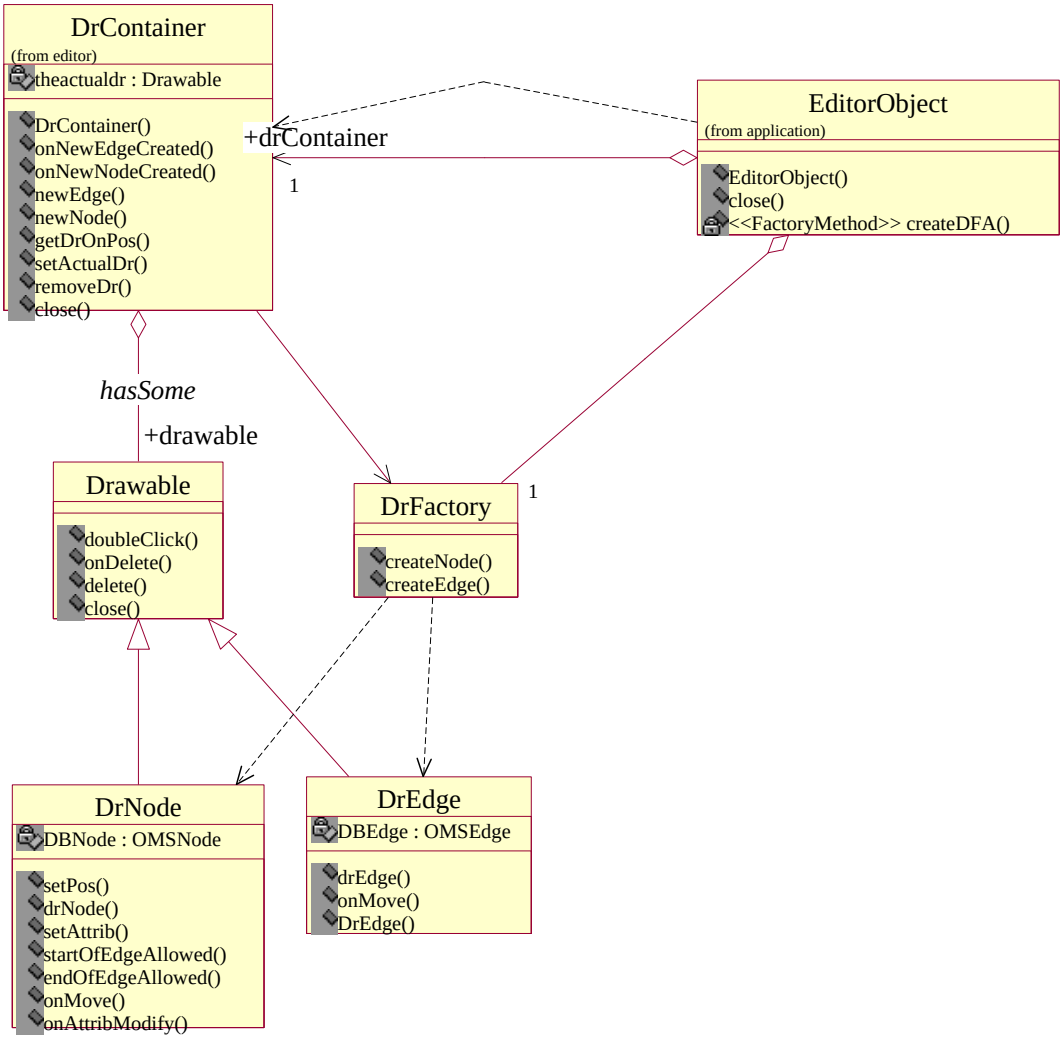


Abbildung 17.7.: Darstellende Komponenten

den scheinbar doppelt vorhanden sind. Dies soll am Beispiel der Neupositionierung eines Knotens erklärt werden.

Beispiel: Neupositionierung eines Knotens Der Benutzer wählt einen Knoten aus und bestimmt dessen neue Position. Die Bildschirmkoordinaten eines Drawables sind persistent und werden somit im Modell in der Datenbank gespeichert. Wir wollen den Weg des Koordinatentupels durch die Klassenhirarchie verfolgen.

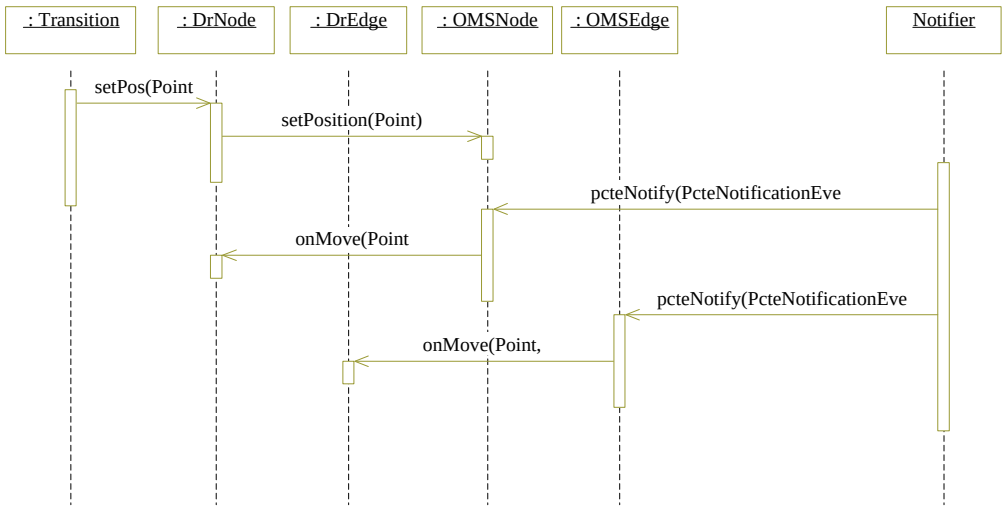


Abbildung 17.8.: Knoten verschieben

Es liegt ein Mausereignis mit einem Koordinatentupel vor, das als neue Position des Knotens interpretiert werden soll. Das aktuell ausgewählte Drawable ist über eine Methode der Klasse DrContainer zugreifbar. Das Koordinatentupel wird der Methode setPos des aktuellen Knotens übergeben. Hier wird nun *nicht* die Positions dieses Objektes auf der Oberfläche geändert. Stattdessen veranlaßt das aktuelle Drawable die Änderung der Position im Modell. Jeder Knoten hat ein korrespondierendes OMSNode-Objekt. Über dieses Objekt kann ein Drawable auf sein Modell in der Datenbank zugreifen. Unser Koordinatentupel wird nun der setPos-Methode des OMSNode-Objektes übergeben, wo die Änderung der Koordinaten des Knotenmodelles veranlaßt wird.

Alle angemeldeten Beobachter eines Knotens werden nun über die Veränderung der Koordinaten informiert. Das OMSNode-Objekt unseres Knotens erhält eine entsprechende Notifikation. Daraufhin wird die onMove-Methode unseres Knotens mit dem neuen Koordiantentupel aufgerufen. Erst jetzt wird die Positions des Knotens auf der Oberfläche aktualisiert.

DrContainer Die Klasse DrContainer verwaltet eine Menge von Drawables. Diese können hinzugefügt und entfernt werden. Desweiteren bietet die Klasse die Methode getDrOnPos an, die zu einer Bildschirmkoordinate das zugehörige Drawable heraussucht, sofern eines existiert.

Neben der Verwaltung der Drawables besitzt DrContainer eine weitere zentrale Aufgabe. Die Klasse wird benachrichtigt, wenn ein neues Modell eines Knotens oder einer Kante in der Datenbank angelegt wurde. Nach dem Modell-View Konzept werden angemeldete Beobachter über Änderungen am Modell benachrichtigt. Wird jedoch ein neues Modell erzeugt, kann es noch keinen Beobachter für dieses Modell geben. Folglich muß dies an anderer Stelle bekanntgegeben werden. Dazu bietet DrContainer zwei Methoden an, `onNewEdgeCreated` und `onNewNodeCreated`. Diese sorgen dann für die Erzeugung der passenden Beobachter.

Ähnlich wie bei den Drawables existiert auch zu DrContainer eine Schnittstellenklasse zur Datenbank, namentlich OMSContainer. Die Klasse OMSContainer erhält die Notifikation aus der Datenbank, daß ein neuer Knoten (bzw. Kante) erstellt wurde und ruft die onNewNodeCreated (onNewEdgeCreated)-Methode des zugeordneten DrContainer-Objektes auf.

17.2.3. Interaktion Objektbank/Editor

Im folgenden wird die Interaktion zwischen Objektbank und Editor näher betrachtet.

17.2.3.1. Node & Edge

Wie im Abschnitt 17.2.2.4 auf der vorherigen Seite erwähnt interagieren immer Paare von Objekten, wenn der Benutzer Veränderungen vornimmt. Über ein Objekt vom Typ `OMSNode` ist das Modell eines Knotens zugreifbar. Der Knoten kann gelöscht werden, sowie seine Attribute können verändert werden.
Das Entwurfsmuster Modell-View schlägt vor, bei einem Modellobjekt alle vorhandenen Beobachter anzumelden, sprich in einer Liste zu verwalten. Der Entwurf der Ampelstadt sieht hier eine leichte Ergänzung vor. Jedes Objekt vom Typ `OMSNode` besitzt genau ein Objekt vom Typ `DrNode` als Pendant. Genaugenommen dient `OMSNode` nur als Schnittstelle für das eigentliche Modell, ein PCTE-Objekt in der Datenbank. So bleiben PCTE-Interna vor den Editorobjekten verborgen.
Die `OMSNode`-Objekte werden bei ihrem zugeordneten PCTE-Objekt als Beobachter angemeldet. Werden Sie über Änderungen notifiziert, reichen sie die Notifikation an ihr Pendant weiter.
Die hier nicht behandelten Kanten ineragieren analog.

17.2.3.2. Ampelstadt starten

Startet der Benutzer die Ampelstadt, so wird als erstes eine Instanz der Klasse `EditorApp` erzeugt (siehe auch 17.2.2.1 auf Seite 149). Der Benutzer meldet sich ein einem Login-Dialog an. Mit den Angaben zu Passwort und Benutzernamen wird nun versucht eine Verbindung zur Datenbank aufzubauen. Ausgangspunkt hierfür ist das Objekt `OMSMain`.

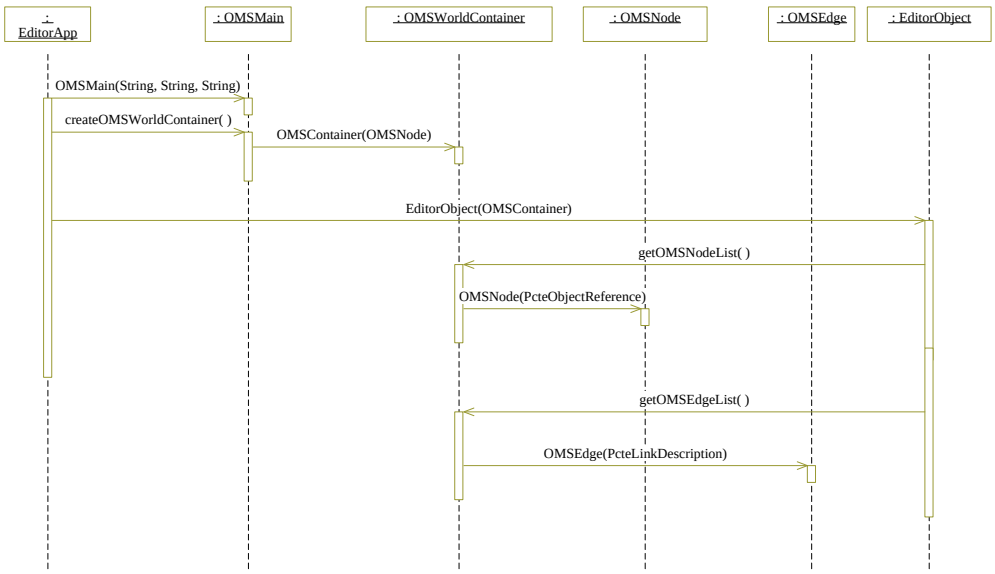


Abbildung 17.9.: Ampelstadt starten

`EditorApp` versucht eine Instanz dieser Klasse anzulegen. `OMSMain` besitzt nur einen öffentlichen Konstruktor, der als Parameter Benutzernamen, Passwort (und Datenbankposition; hier nicht weiter von Belang). Schlägt die Anmeldung bei der Objektbank fehl, bricht der Kontruktor die Erzeugung des Objektes ab, es wird eine Exception geworfen. Die Instanziierung von `OMSMain` ist nur mit gültigem Benutzernamen und Passwort möglich.

Die Container

Erzeugung der Beobachter Ein Editorfenster stellt einen Teil des gesammten Ampelstadtgraphen dar. Wird ein Editor neu gestartet, sind die Modelle der anzuzeigenden Knoten und Kanten in der Datenbank gespeichert. Die entsprechenden Beobachter eines EditorObjektes werden durch einen zugeordneten DrContainer verwaltet. Dieser erhält eine Liste der in der Objektbank vorhandenen Modelle und erzeugt zughörige Beobachterobjekte. Diese Liste ist über eine Methode der Klasse `OMSContainer` verfügbar. Das paarweise Vorhandensein von Klassen der Objektbank und des Editors setzt sich auch an dieser Stelle fort.
Eine Instanz von `DrContainer` verwendet eine Instanz von `OMSContainer` um eine Liste der anzuzeigenden Objekte zu erhalten. Gemäß dieser Liste werden Beobachterobjekte erzeugt und mit ihrem Modell verbunden.

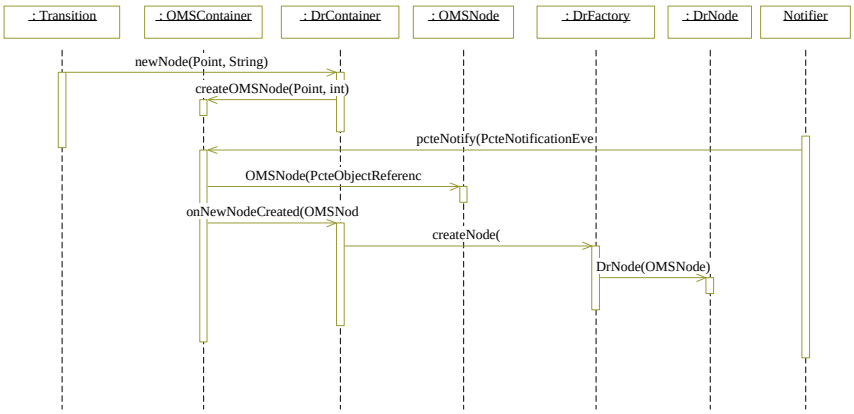


Abbildung 17.10.: Erzeugen eines neuen Knotens

Neue Knoten/Kanten Soll ein neuer Knoten (Stadt oder Kreuzung) erzeugt werden, so wird in der entsprechenden Transition des Automaten die Methode `newNode` des `DrContainers` mit den Parametern Position und Typ (Stadt, Kreuzung oder Stadttor) aufgerufen. Der `DrContainer` leitet diesen Aufruf mit `createOMSNode` an seinen zugehörigen `OMSContainer` weiter. Dieser instanziiert einen neuen `OMSNode`. Erhält der `OMSContainer` nun von der Datenbank die Notifizierung, daß der Knoten erfolgreich angelegt wurde, wird diese vom `OMSContainer` durch Aufruf der Methode `onNewNodeCreated` des `DrContainers` weitergeleitet. Der `DrContainer` läßt sich von der `DrFactory` mittels `createNode` einen neuen `DrNode` instanziieren und fügt diesen dem Zeichenpanel hinzu.

Welteditor starten In Abschnitt 17.2.3.2 auf Seite 157 wurde die Instanziierung eines `OMSMain`-Objktes beschrieben. Anforderung an die Ampelstadt ist, daß der Benutzer nach dem erfolgreichen Anmeldevorgang den Welteditor auf dem Bildschirm sieht. Im Anschluß an den Anmeldevorgang wird ein `WorldEditor` erzeugt. Über `OMSMain.getWorldContainer()` läßt sich ein `OMSContainter` erzeugen. Dieser Container macht alle in der Weltsicht vorhandenen Knoten und Kanten zugreifbar. Mittels dieses Containers kann der Welteditor auf die Modelle der Knoten und Kanten der Weltsicht zugreifen.

17.2.4. Spezieller Editor

Allgemeine Uebersicht ueber die Konkretisierung der generischen Teile im speziellen Editor.

- Diagramm der angezeigten Klassen
- Kurzbeschreibung: Sichten, Akteure, konkreten Fabriken

Teil IV.

Grundlagen für die $\mathcal{HEU}$

18. Die grafische Notation von *DoDL*

Dieser Abschnitt enthält die komplette Beschreibung der grafischen Notation zu *DoDL*. Es wird zunächst ein kurzer Überblick gegeben, dann werden die einzelnen Diagramme beschrieben. Dabei wird zu jedem Diagramm seine Aufgabe beschrieben, ein Beispiel angegeben und die Abbildung in textuelles *DoDL* gezeigt.

18.1. Einleitung

Autor: *Christoph Begall*

Bei den grafischen Elementen der Notation wurden weitestgehend bestehende Notationen wie UML und Symbole der visuellen Programmierung zum Vorbild genommen, und mit der Einfachheit und Eleganz von *DoDL* gepaart. Auf die (textuelle) Sprache *DoDL* wird im weiteren nur insofern eingegangen, daß die Abbildung der einzelnen Diagramme nach *DoDL* beschrieben wird.

Die grafische Notation zu *DoDL* besteht aus vier verschiedenen Diagrammformen, die etwas später noch genauer betrachtet werden. Diese vier Diagrammformen beschreiben bestimmte Aspekte eines mit *DoDL* erstellten Hyperdokuments auf verschiedenen Verfeinerungsebenen.

Auf der obersten Ebene wird das Hyperdokument als Verkettung von Medienobjekten betrachtet. Diese Objekte werden durch Klassen beschrieben, die Beziehungen zwischen ihnen durch Aggregationen und Generalisierungen. Die Objekte können Teile eines größeren Hyperdokuments sein, oder umgekehrt auch mehrere Teildokumente zu einem größeren zusammenfassen. Auf ähnliche Weise können abstrakte und konkrete Formen eines solchen Dokumentes beschrieben werden. Diese Beziehungen zwischen den Klassen werden im *Klassendiagramm* beschrieben.

Betrachtet man solche Teile des gesamten Hyperdokuments genauer, so begibt man sich auf die Ebene der *Klassenansicht*, in der man die in der Klasse existierenden Attribute und Methoden sehen kann. Attribute sind dabei die Daten und Teildokumente einer Klasse, Methoden schaffen die abstrakten Verbindungen zwischen diesen.

Solche Methoden lassen sich in der grafischen Notation zu *DoDL* auf drei verschiedene Arten spezifizieren, von denen die naheliegenste wohl das *Hyperdokumentdiagramm* ist. Dieses zeigt die Verbindungen der einzelnen Dokumente durch Pfeile zwischen den Dokumenten.

Um eine größere Modellierungsfreiheit zu gewährleisten und die Möglichkeiten der Implementation von *DoDL* besser auszunutzen, kann man Methoden auch über *Kontrollflußdigramme* spezifizieren. Diese geben nicht direkt die Verbindungen zwischen den

Dokumenten an, sondern beschreiben, wie die entsprechenden Verbindungen konstruiert werden.

Als letzte Möglichkeit der Methodenspezifikation sei hier die Freitext-Methode erwähnt, die einfach nur Quellcode in der Programmiersprache darstellt, die die Gastsprache zu *DoDL* darstellt. Da dies zum einen kein Diagramm darstellt, also nicht Teil der grafischen Notation ist, zum anderen stark von der Implementation des textuellen *DoDL* abhängt, wird es nicht weiter beschrieben und hier nur auf die prinzipielle Möglichkeit einer solchen Spezifikation verwiesen. Für näheres sei hier auf [Dob96] und Kapitel 6 auf Seite 41 verwiesen.

18.2. Klassendiagramme

Autor: *Christoph Begall*

Im folgenden werden die *DoDL*-Klassendiagramme beschrieben. Dazu wird nach einer kurzen Einleitung ein Beispiel gegeben, dann die Regeln für korrekte Diagramme beschrieben. Anschließend wird gezeigt, wie die Diagramme in korrektes *DoDL* überführt werden. Schließlich wird noch ein kurzer Abriß darüber gegeben, warum diese und keine andere Notation gewählt wurde.

18.2.1. Allgemeine Beschreibung

Das Klassendiagramm stellt die *DoDL*-Klassen und ihre Beziehungen untereinander dar. Die Klassen werden als Rechtecke dargestellt, in deren Mitte sich der Name der Klasse befindet. Zwischen den Klassen gibt es zwei Arten von Beziehungen: Generalisierung und Aggregation. Generalisierung, oder auch Erbung, wird durch eine durchgezogene Linie und ein Dreieck in der Mitte dieser Linie dargestellt. Das nicht ausgefüllte Dreieck zeigt auf die allgemeinere Klasse, eine Generaliesierung ist also eine gerichtete Kante von Sub- zu Superklasse. Eine beispielhafte Genaralisierung ist in Abbildung 18.1 links zu sehen, wobei die erbende Klasse `Sohn` ist.

Abbildung 18.1.: Generalisierung und Aggregation in *DoDL*

Aggregation wird ebenfalls durch eine durchgezogene Linie dargestellt, die am Ende der aggregierenden Klasse eine leere Raute hat. Aggregation bedeutet in diesem Zusammenhang, daß die Klasse, an der sich die Raute befindet, ein Attribut hat, das Objekt der Klasse ist, die sich am anderen Ende der Kante befindet. Auch dies kann man in Abbildung 18.1 beispielhaft sehen (rechts).

Über solchen Aggregations-Kanten kann optional ein Sternchen (`*`) stehen, das anzeigt, daß die aggregierende Klasse eine Liste von Objekten enthält.

18.2.2. Ein Beispiel

Um einen Überblick über die vorgestellte Notation zu geben, wird hier in Abbildung 18.2 ein beispielhaftes Klassendiagramm gezeigt. Es wurde hierbei versucht, die Dokumentation der Projektgruppe als großes Hyperdokument darzustellen.

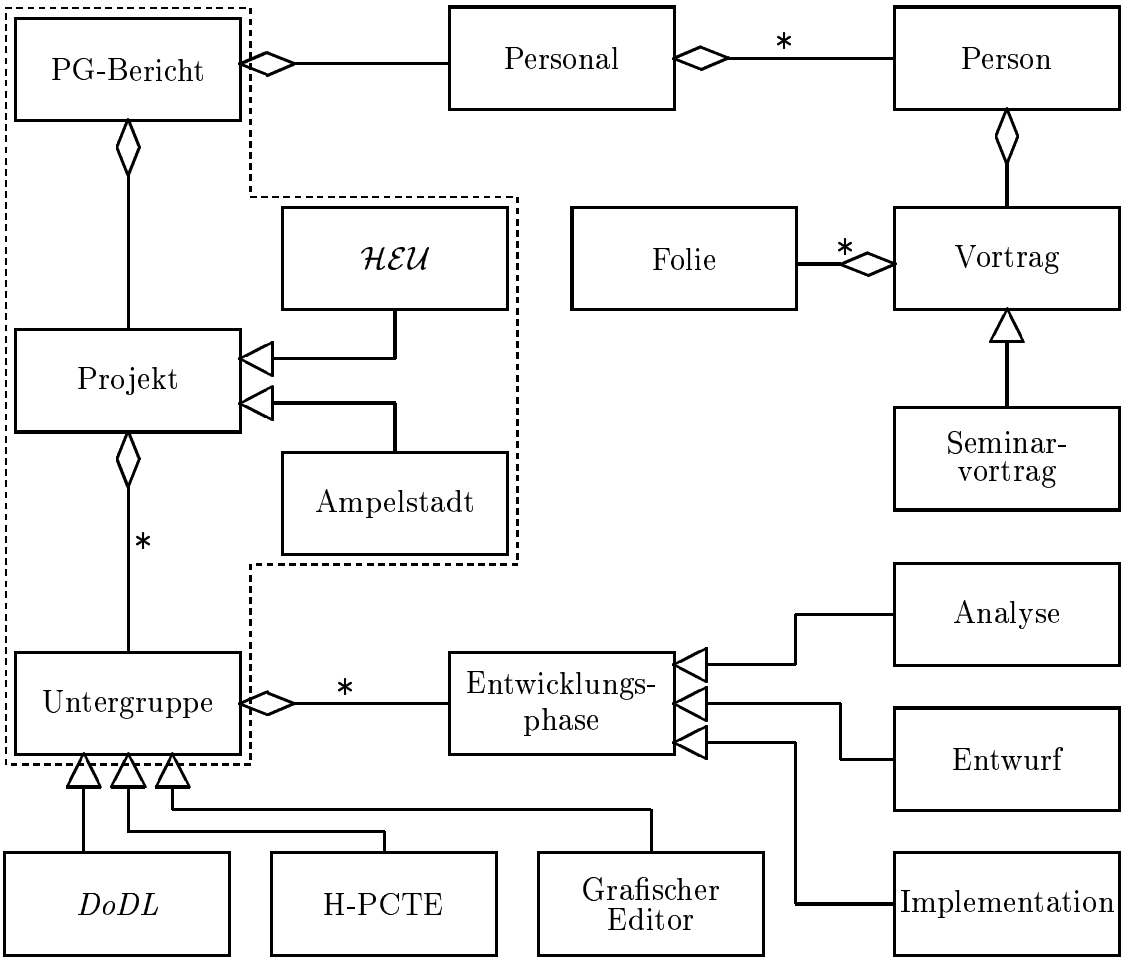


Abbildung 18.2.: Ein etwas größeres Beispiel in *DoDL*

18.2.3. Regeln für korrekte Klassendiagramme

Da die *DoDL*-Klassendiagramme in korrektes textuelles *DoDL* übersetzt werden sollen, müssen einige Regeln beachtet werden.

1. In den Diagrammen dürfen keine zyklischen Erbungs- oder Aggregationsbeziehungen auftreten.

2. Auch gemischte Zyklen sind nicht zulässig.

Um dies zu verdeutlichen, wurden in Abbildung 18.3 alle Kombinationen aus erlaubten und nicht erlaubten Beziehungen zwischen drei *DoDL*-Klassen dargestellt.

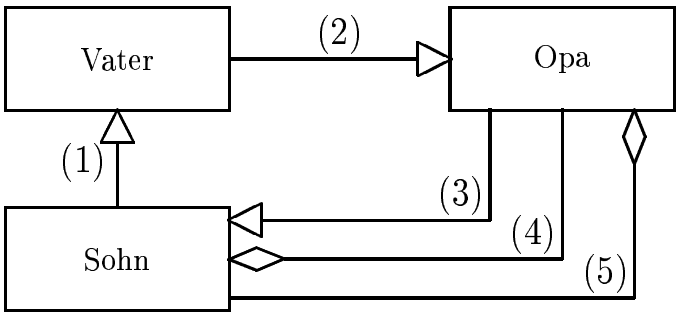


Abbildung 18.3.: Nicht zulässige Beziehungen (3,4,5) zwischen *DoDL*-Klassen

Wenn z.B. eine Klasse `Sohn` von einer Klasse `Vater` erbt (1) und diese wiederum von einer Klasse `Opa` (2), so darf weder `Opa` von `Sohn` erben (3), noch darf `Sohn` ein Attribut vom Typ `Opa` haben (4). Außerdem ist es natürlich nicht zulässig, daß in einer Kette von Erbungsbeziehungen zwei Klassen Teil der gleichen Aggregation sind. So darf auch `Opa` kein Attribut vom Typ `Sohn` haben (5).

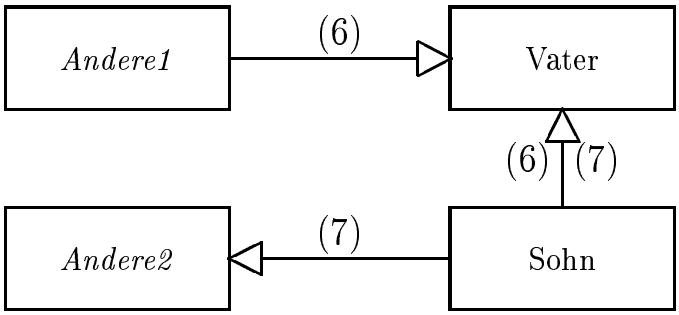


Abbildung 18.4.: Mehrfacherbung in *DoDL*: Nicht erlaubt (7)

Desweiteren ist in *DoDL* Mehrfacherbung nicht erlaubt, wie in Abbildung 18.4 dargestellt ist. Das bedeutet für die *DoDL*-Klassendiagramme, daß eine Klasse zwar mehrere eingehende Pfeilspitzenhaben darf (6), jedoch nicht mehrere ausgehende Pfeile (7). Im eben angeführten Beispiel darf `Sohn` also auch nicht von weiteren Klassen erben (einschließlich Klassen, von denen er ohnehin direkt – im Beispiel `Vater` – oder indirekt – im Beispiel `Opa` – erbt).

18.2.4. Umsetzung in *DoDL*

Die textuelle Umsetzung nach *DoDL* kann auf verschiedene Arten vor sich gehen, es wurde jedoch aus den bestehenden Möglichkeiten diejenige ausgewählt, die sich am leichtesten umsetzen läßt. Hier wird beschrieben, wie sich das gesamte Klassendiagramm in eine

Rahmenklasse einbetten läßt, und wie darin die einzelnen Klassen spezifiziert werden können.

Bei den im folgenden aufgeführten textuellen *DoDL*-Klassen handelt es sich um Schablonen, bei denen einige allgemein übliche Techniken benutzt wurden:

- Platzhalter werden durch spitze Klammen (`< >`) hervorgehoben.
- Sternchen (`*`) zeigen eine beliebig lange Liste des vorangehenden Platzhalters an, analog zu regulären Ausdrücken.
- Eckige Klammern (`[ ]`) werden für optionale Teile benutzt.

18.2.4.1. Die Rahmenklasse

Das gesamte Szenario, das im Klassendiagramm festgelegt wird, wird in eine große *DoDL*-Klasse umgesetzt, in die die einzelnen Klassenspezikationen eingefügt werden. Die folgende Schablone einer *DoDL*-Klasse kann für einen solchen Rahmen benutzt werden:

```
class HEUDoDLRahmenKlasse is
  declare
    <Klassenspezifikationen>*

  documents
    m: <Startklasse>;

  constructs
    void main() {
      m.<StartMethode>();
    }
end HEUDoDLRahmenKlasse;
```

Die `<Klassenspezifikationen>*` müssen dann durch die im nächsten Abschnitt beschriebenen *DoDL*-Klassen ersetzt werden. Die Namen für die `<Startklasse>` und die `<Startmethode>` müssen vor der Umsetzung des Diagramms festgelegt werden.

18.2.4.2. Einzelne Klassen

Die Klassenspezifikationen werden im `declare`-Teil einfach hintereinander geschrieben. Dabei werden die Beziehungen, also Generalisierungen (zwischen `<Klassenname>` und `<SuperklassenName>`) und Aggregationen, zwischen den Klassen wie folgt abgebildet:

```
class <KlassenName> is [<SuperklassenName> with]
  documents
    <VariablenName 1>: <AggregierterKlassenName 1>;
    ⋮
```

```
      <VariablenName n>: <AggregierterKlassenName n>;

    constructs
      <Methodenspezifikationen>*
  end <KlassenName>;
```

Es handelt sich hierbei um eine Schablone, die für jedes einzelne Rechteck in einem Diagramm ausgefüllt werden muß. Der `<Klassenname>` (auch der `<SuperklassenName>`, falls existent) ist in den Diagrammen eindeutig festgelegt durch den Namen, der in den Rechtecken steht. Die Superklasse ist dabei diejenige, auf die ein eventuell vorhandener Generalisierungspfeil zeigt.

Alle Aggregationen, deren Rauten-Enden bei der abzubildenen Klasse liegen, müssen im `documents`-Teil der Klasse erscheinen, wobei die Namen der Klassen am anderen Ende der Aggregation dabei die Platzhalter `<AggregierterKlassenName 1>` ... `<AggregierterKlassenName n>` füllen. Die dazugehörenden `<VariablenName i>`-Platzhalter müssen vor der Code-Erzeugung festgelegt werden, ebenso die `<Methodenspezifikationen>*`, die auch durch andere Diagrammformen beschrieben werden können.

18.2.4.3. Beispielhafte Umsetzung

Um die in den vorangehenden Abschnitten erläuterten Hinweise zur Umsetzung der Klassendiagramme in textuelles *DoDL* noch deutlicher zu machen, wird hier abschließend noch einmal ein Teil des Beispiels aus Abschnitt 18.2.2 auf Seite 165 betrachtet. Es wird dabei davon ausgegangen, daß in der Klasse `PG-Bericht` eine Methode `start()` existiert und als Startmethode markiert wurde. Als Startklasse wurde folglich diese Klasse ausgewählt.

Auf Methodenspezifikationen und auf Platzhalter für diese wurde hier verzichtet, da aus dem letzten Abschnitt klar sein sollte, wo diese einzufügen sind, und ihre Erzeugung in den Abschnitten 18.4 auf Seite 172 und 18.5 auf Seite 173 beschrieben wird. Der Code, der so bei der Bearbeitung des gestrichelt umrandeten Bereichs in dieser Abbildung entsteht, sieht wie folgt aus:

```
class HEUDoDLRahmenKlasse is
  declare
    class Untergruppe is
    end Untergruppe;

    class Projekt is
      documents
        list1: list of Untergruppe;
    end Projekt;

    class HEU is Projekt with
    end HEU;

    class Ampelstadt is Projekt with
```

```
    end Ampelstadt;

    class PGBericht is
      documents
        name1: Projekt;
    end PGBericht;

  documents
    m: PGBericht;

  constructs
    void main() {
      m.start();
    }
end HEUDoDLRahmenKlasse;
```

18.2.5. Rationale

Während der Erarbeitung einer grafischen Notation für die Beschreibung von Beziehungen zwischen *DoDL*-Klassen wurden mehrere Ansätze verfolgt. Der letztendlich benutzte beruht, wie schon erwähnt, z.T. auf den Klassendiagrammen von UML (siehe auch 5.3.2.2 auf Seite 31).

Bei der Erforschung von UML findet man dort jedoch keine Konzepte, die denen der inneren Klassen von *DoDL* ähnlich sind. Einer unserer dahingehenden Ansätze war, Dokumente und Klassen als ineinander geschachtelte Rechtecke darzustellen. Bei der Weiterverfolgung und Ausarbeitung dieser Idee wurde jedoch klar, daß eine solche Notation für einen grafischen Editor nur sehr schwer bearbeitbar ist, und die Komplexität der Aufgabe der Projektgruppe unnötig steigert.

Es ist auch nur schwer vorstellbar, wie sich die Aufgabe von inneren Klassen, nämlich unter anderem die Datenkapselung und Abstraktion mit einer solchen Darstellung übereinbringen läßt, da dem Benutzer immer ein Blick auf das Ganze, also sämtliche vorhandenen Klassen gegeben wird. Alle Ideen zur Umsetzung vom „Verstecken der inneren Klassen“ bei einer solchen Diagrammform erwiesen sich als nicht umsetzbar.

18.3. Die Klassenansicht

In diesem Abschnitt wird die grafische Darstellung einzelner *DoDL*-Klassen beschrieben, die in einer kompakten Darstellung die Eigenschaften einer Klasse aufzeigt. Sie soll ein leichteres und somit schnelleres Verständnis der Struktur der *DoDL*-Klasse bieten.

18.3.1. Beschreibung der Klassenansicht

Autor: *Matthias Dorka*

Die Klassenansicht ist ein Teil der grafischen Notation von *DoDL*, der eine einzelne *DoDL*-Klasse mit ihren Bestandteilen visualisiert. Dabei wird die Klasse in Anlehnung an UML als rechteckiges Kästchen mit spitzen Ecken dargestellt (siehe Abb. 18.5 auf der nächsten Seite). Dieses Kästchen wird grundsätzlich durch drei horizontale Trennstriche in vier Bereiche unterteilt, den Klassenkopf, den *complex-documents*-Teil, den *single-documents*-Teil und den *method*-Teil.

Im Klassenkopf steht der Name der Klasse, so wie er in textuellem *DoDL* hinter den Schlüsselwort `class` steht. Aus dem Klassenkopf geht nicht hervor, ob die Klasse von einer anderen erbt oder abstrakt ist.

Im *complex-documents*-Teil werden alle Attribute dargestellt, die in dieser Klasse instanziiert werden und selbst vom Typ *class* sind. Dabei wird jede Instanz einer aggregierten Klasse wiederum als Icon dargestellt, das einem horizontal viergeteilten Rechteck entspricht. Der Name und Typ der Variable steht dabei unter dem Icon in der Form `<Name>:<Typ>`.

Im *single-documents*-Teil finden sich in gleicher Weise Variablen wieder, die vom allgemeinen Grundtyp `dbUnit` sind bzw. vom Typ einer einfachen Datenstruktur wie beispielsweise `text` oder `graphic`. Dabei sollen Icons verwendet werden, die der Verkleinerung eines Blattes gleichen: rechteckig, hochkant stehend und mit eingeklappter oberer rechter Ecke (siehe Abb. 18.5 auf der nächsten Seite). Zur Veranschaulichung der vordefinierten primitiven Datentypen beinhalten diese Icons eventuell weitere Symbole, im Falle `text` drei geschlängelte Linien und für `graphic` z. B. einen Smiley (ikonifiziertes lächelndes Gesicht bestehend aus einem Kreis mit zwei Punkten und einem Bogen). Je nach aktueller Version der *DoDL*-Spezifikation sind hier in Zukunft Symbole für weitere, noch zu definierende primitive Datentypen denkbar. Ist die Variable vom Grundtyp `dbUnit`, so bleibt das Icon leer.

Da *DoDL* auch mengenwertige Variablen in Form von Listen unterstützt, werden diese für komplexe als auch für einfache Dokumente dadurch veranschaulicht, daß sich hinter dem Icon horizontal und vertikal leicht versetzt die Rahmen mehrerer gleichartiger Icons befinden. Dies ist ebenfalls aus Abb. 18.5 auf der nächsten Seite ersichtlich. Die Unterschrift einer solchen Variable ist `<Name>: List of <Typ>`.

Der unterste Teil der grafischen Notation einer Klasse in der Klassenansicht beinhaltet die Methoden dieser Klasse. Jede Methode wird dabei als rechteckiges Kästchen mit abgerundeten Ecken dargestellt, analog zur Darstellung in der Methodenansicht. Das genaue Aussehen dieser Icons wird in Abb. 18.5 auf der nächsten Seite unten spezifiziert. Die Methode `main` ist dabei als einzige mit doppelter Umrandung darzustellen. Der Name der Methode ist unter dem jeweiligen Symbol angegeben. Dabei kann die Art der Methode durch einen führenden Buchstaben vorangestellt werden, beispielsweise `H:<Name>` für Methoden, die in der Abb. 18.5 auf der nächsten Seite genauer betrachtet werden können. Als weitere Buchstaben bezeichnen `T` und `C` Methoden, die über die Text- bzw. Kontrollflußansicht im einzelnen dargestellt werden können.

Sollte eine Klasse so unspezifiziert sein, daß sie einen oder mehrere der hier beschriebenen vier Teile nicht enthält, so bleibt in der Klassenansicht dieser grafischen Notation das entsprechende Teilkästchen leer.

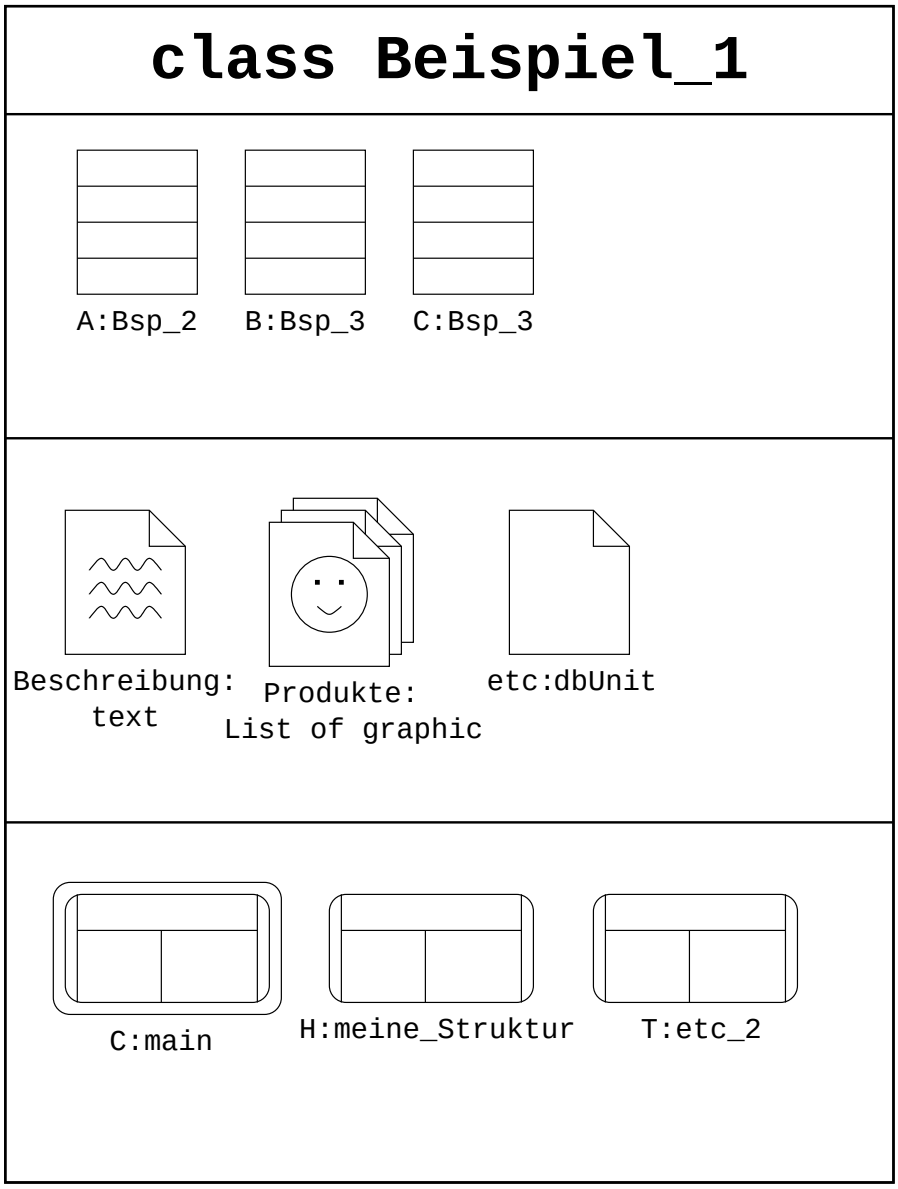


Abbildung 18.5.: Eine einfache Klasse in der Klassenansicht

18.4. Die Hyperlinkansicht

Autor: *Matthias Dorka*

18.4.1. Allgemeines zur Hyperlinkansicht

In der Hyperlinkansicht der grafischen Notation von *DoDL* werden generische Hyperlinks des späteren Hyperdokuments und damit eine bestimmte Art von Methoden der Dokumentspezifikation visualisiert. Diese Methoden haben die Eigenschaft, daß sie Hyperlinks erzeugen, deren Ankerpositionen nicht sehr speziell berechnet werden müssen, sondern die immer gleichen Anforderungen entsprechen und deshalb als vordefinierte Linktypen in der grafischen Notation zur Verfügung gestellt werden sollen.

18.4.2. Anker in der Hyperlinkansicht

Dazu wird zunächst vereinbart, daß für jedes in *DoDL* benutzte Dokument der Anfang und das Ende dergestalt definiert sind, daß sich diesen Dokumentstellen immer Positionen für Anker zuweisen lassen. Diese Anker werden *begin* und *end* genannt und sollen sicherstellen, daß jedes Dokument (Textdatei, Grafik, ...) mindestens eine Stelle hat, über die es an das gesamte Hyperdokument angebunden werden kann. Es ist vorstellbar, daß *begin* und *end* zusammenfallen, z. B. bei einer leeren Textdatei oder einer sehr kleinen Grafik.

Der dritte an dieser Stelle vordefinierte Ankertyp ist der *occurrence*-Anker. Er kann nur in Dokumenten vorkommen, die vom Typ `text` sind, da er Positionen in einem solchen Dokument bezeichnet, an denen eine angegebene Zeichenkette vorkommt.

Der Zweck der Hyperlinkansicht ist nun, Anker in Dokumenten und die sie verbindenden Links sichtbar zu machen. Dazu werden einfache Dokumente als Icons wie im Kapitel 18.3.1 auf Seite 169 dargestellt: als hochkant stehendes Rechteck mit eingeklappter oberer rechter Ecke und eventuell einem weiteren Symbol darin, das den Typ des Dokuments angibt. Zu jedem Dokument gehören gemäß obiger Vereinbarung implizit zwei Anker, der *begin*- und der *end*-Anker. Diese sind seitlich an der linken oberen und rechten unteren Ecke des Dokument-Icons darzustellen. Alle Anker werden durch ein Fünfeck symbolisiert, das einem nach unten zeigenden Blockpfeil entspricht (siehe Abb. 18.6). Die Darstellungen der ausgezeichneten Anker *begin* und *end* enthalten auch als Bezeichnungen `begin` und `end`, der an beliebiger Stelle des Dokument-Icons seitlich befindliche *occurrence*-Anker enthält den Schriftzug `occ`.

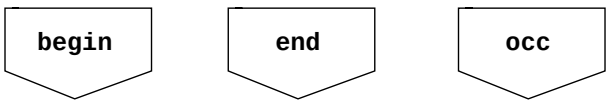


Abbildung 18.6.: *begin*-, *end*- und *occurence*-Anker in der Hyperdiagrammansicht

Man beachte, daß ein *occurrence*-Anker nicht eine bestimmte Position im zugrundeliegenden Dokument bezeichnet, sondern alle Positionen, an denen der spezifizierte

Suchstring dieses Ankers vorkommt. Sollte dieser im Text nicht vorkommen, so liefert der Anker keine Position zurück, und eventuell von diesem Anker ausgehende Hyperlinks (s. u.) würden bei der Umwandlung der *DoDL*-Spezifikation in ein Hyperdokument nicht erzeugt.

An dieser Stelle sei angemerkt, daß die grafische Notation von *DoDL* hier Raum läßt, um in Zukunft weitere vordefinierte Ankertypen einzuführen, beispielsweise ein *first-occurrence*, `1-occ`, der das erste Vorkommen einer Zeichenkette in einem Textdokument bezeichnet, oder spezielle Anker für bestimmte Inhalte von Grafiken.

18.4.3. Links in der Hyperlinkansicht

Darüberhinaus sind in dieser Ansicht auch die Hyperlinks visualisiert, die zwischen Ankern liegen und die entsprechenden Dokumentpositionen verbinden. Diese werden als dünne durchgehende Pfeile mit geschlossener Spitze dargestellt, ausgehend vom Startanker zeigen sie auf das Sprungziel.

Aus Einschränkungen, die sich durch die Praxis des Durchlaufens von Hyperdokumenten ergeben, darf von jedem Anker nur ein Link ausgehen, während beliebig viele Links an einem Anker enden dürfen. Andernfalls wäre das Sprungziel des Links nicht eindeutig definiert. Deshalb dürfen bei *occurrence*-Ankern nie Pfeile enden, da sie potentiell mehrere Positionen in einem Dokument bezeichnen.

Ein beispielhaftes Aussehen einer Spezifikation eines kleinen Hyperdokuments in der Hyperlinkansicht ist abschließend in Abb. 18.7 gezeigt.

18.5. Kontrollflußdiagramme

Ein Kontrollflußdigramm ermöglicht eine einfache visuelle Programmierung. Es wird zunächst die Syntax und die Umsetzung in DoDL erläutert. Anschliessend wird ein kleines Beispiel für eine Kontrollflussmethode vorgestellt.

18.5.1. Beschreibung der Kontrollflußmethode

Die Kontrollflußmethode ruft bereits existierende Methoden auf und legt dabei den Kontrollfluß fest. Das Kontrollflußdiagramm veranschaulicht also grafisch die Reihenfolge der Abarbeitung anderer Methoden. Es sind dabei Verzweigungen erlaubt, die eine Programmierung von Schleifen ermöglichen.

18.5.2. Syntax der Kontrollflußmethoden

Kontrollflußmethoden (siehe Abbildung 18.8) sind in drei Bereiche eingeteilt. Im obersten Bereich steht der Name der Methode. Im mittleren Bereich sieht man die zur Verfügung stehenden Methoden, und im unteren Bereich wird schließlich der Kontrollflußgraph dargestellt. Pfeile geben die Flußrichtung der Abarbeitung von Methoden und Verzweigungen an. Die Startmethode wird mit einem Doppelrahmen gekennzeichnet. Jede Methode

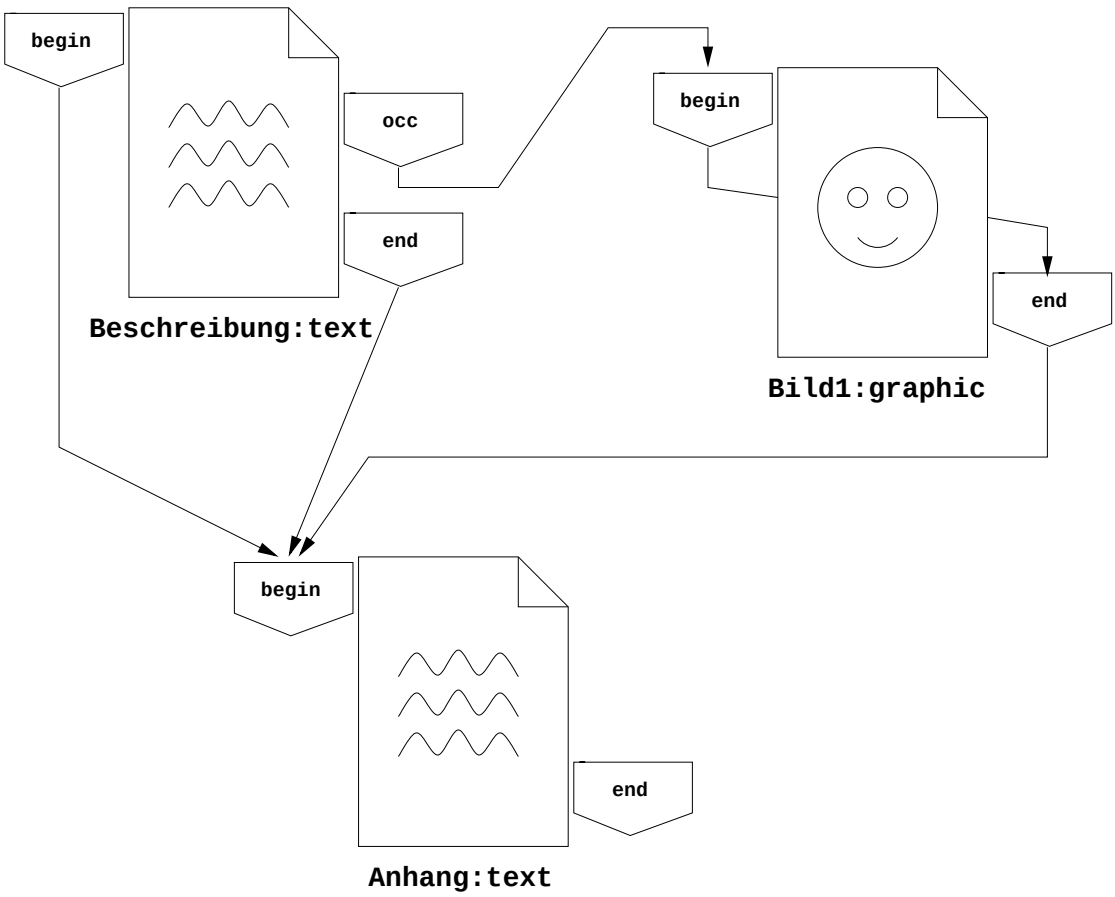


Abbildung 18.7.: Die Hyperdiagrammansicht mit Dokumenten, Ankern und Links

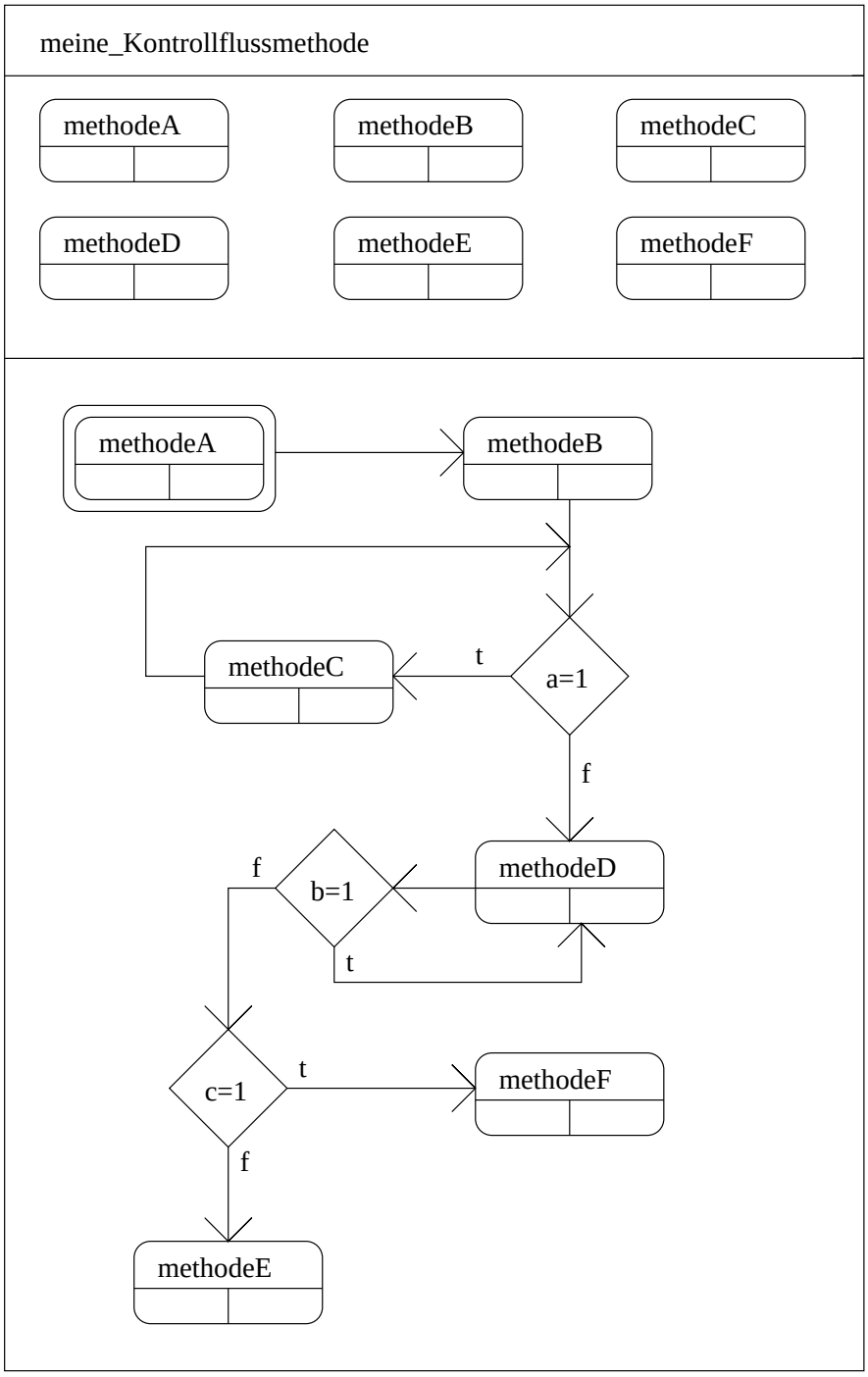
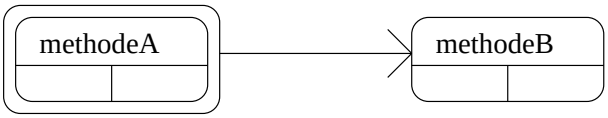


Abbildung 18.8.: Beispiel für eine Kontrollflußmethode

kann höchstens einen ausgehenden und beliebig viele eingehende Pfeile haben. Verzweigungen werden durch eine Raute dargestellt. Sie haben genau zwei ausgehende Pfeile, die durch `true` und `false` gekennzeichnet werden. Die Verzweigungsbedingung steht in de Raute selbst und beinhaltet einen boolschen Ausdruck. Ist der boolsche Ausdruck wahr, wird nach 'true' verzweigt, ansonsten nach 'false'. Die Verzweigung kann nun auch für Schleifen eingesetzt werden. Wird die Verzweigung am Anfang der Schleife benutzt, realisiert man eine 'while'-Schleife. Will man eine 'do-while'-Schleife programmieren, so setzt man die Schleifenabbruchbedingung am Ende der Schleife ein.

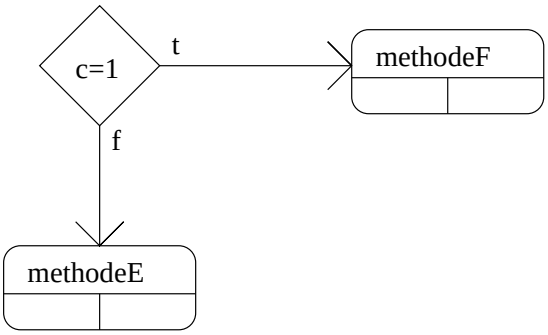
18.5.3. Umsetzung in *DoDL*

Die Umsetzung in *DoDL* wird anhand des Beispiels erklärt, indem die syntaktischen Elemente der Kontrollflußmethoden in die Gastsprache C übersetzt werden.



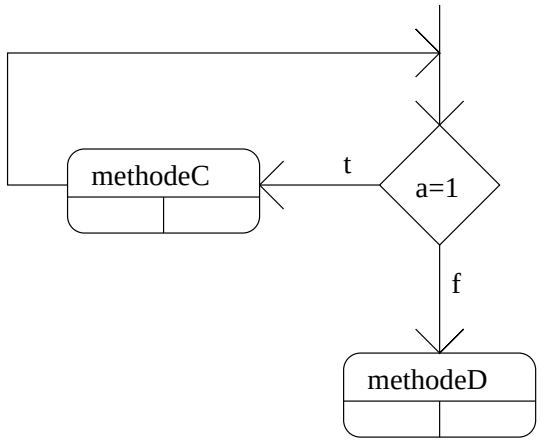
Die Pfeile zu einer Methode legen die Reihenfolge der Methodenaufrufe fest:

```
methodeA();
methodeB();
```



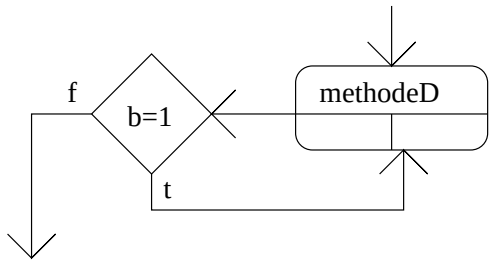
Einfache Verzweigungen stellen eine if-Anweisung dar:

```
if (c=1) {
  methodeF();
}
else {
  methodeE();
}
```



While-Schleifen werden wie folgt in der Gastsprache C umgesetzt:

```
while (a=1) {
  methodeC();
}
methodeD();
```



Steht die Abbruchbedingung am Ende der Schleife, so wird eine Do-while-Schleife realisiert:

```
do {
  methodeD();
} while (b=1);
```

Das gesamte Beispiel sieht in der Gastsprache C folgendermaßen aus:

```
void meine_Kontrollflussmethode(void) {
   methodeA();
   methodeB();
   while (a=1) {
      methodeC();
   }
   do {
      methodeD();
   } while (b=1);
   if (c=1) {
```

```
      methodeF();
   } else {
      methodeE();
   }
}
```

Für Kontrollflußmethoden sind bisher noch keine Parameter vorgesehen. Daher ist der Typ der Eingabe- und Rückgabeparameter `void`.

Teil V.

Anforderungsanalyse der $\mathcal{HEU}$

19. Planung für das zweite Semester

Autoren: *Klaus Alfert*
Alexander Fronk

Nach einem Semester intensiver Auseinandersetzung mit den Strukturen einer $\mathcal{HEU}$ hat sich herausgestellt, daß die Spezifikation mit *DoDL* nur dann sinnvoll erfolgen kann, wenn die $\mathcal{HEU}$ ihren Ansprüchen gerecht wird.

Daher sind wir von dem Minimalziel abgewichen und stellen die Architektur und Realisierung der $\mathcal{HEU}$ in den Vordergrund der Arbeit im zweiten Semester. Grundlage für die Planungen war das folgende Papier, das nahezu unverändert mit den Teilnehmern der PG diskutiert wurde.

19.1. Disskussionsvorschlag für die Planung des zweiten Semesters

Das Vorgehen im ersten Projektsemester hat sich als sehr hilfreich erwiesen. Der Prototyp *Ampelstadt* ist mehr als ein bloßes Übungsobjekt, an dem wir technisches Wissen gesammelt haben. Es haben sich darüberhinaus viele strukturelle Gemeinsamkeiten mit der für *DoDL* geplanten Editorlandschaft ergeben. Um den erfolgreichen Abschluß ein wenig zu steuern, haben wir einen Schlachtplan entwickelt, den wir zur Diskussion stellen wollen.

19.2. Idee

Eine $\mathcal{HEU}$ im eigentlichen Sinne zu implementieren wird in dieser Projektgruppe nicht möglich sein. Die für *DoDL* benötigten Editoren stellen lediglich einen Teil der $\mathcal{HEU}$ dar, sind aber ihrerseits in der Entwicklung sehr zeit- und arbeitsintensiv. Wir haben uns eine Architektur (s. Abb. 19.1) für $\mathcal{HEU}$ überlegt, in die wir die Editorlandschaft einbetten wollen. Ziel des zweiten Projektsemesters ist somit die Entwicklung der Editoren und deren Einbettung in $\mathcal{HEU}$. Das verwendete Komponentenkonzept zeigt, wie Informationen ggf. sinnvoll aufgeteilt werden können, um den Gedanken der verteilten Umgebung zu untermauern (s. Abb. 19.2).

Die verbleibende Zeit bis zum Ende des SoSe 99 ist erschreckend kurz. Wir möchten und sollten den Hauptteil der Arbeit auf die Implementierungsphase legen. Damit müßen wir die klassischen Phasen Analyse und Entwurf möglichst effizient in möglichst geringer Zeit durchlaufen. Diesen vermeindlichen Zielkonflikt wollen wir wie folgt lösen.

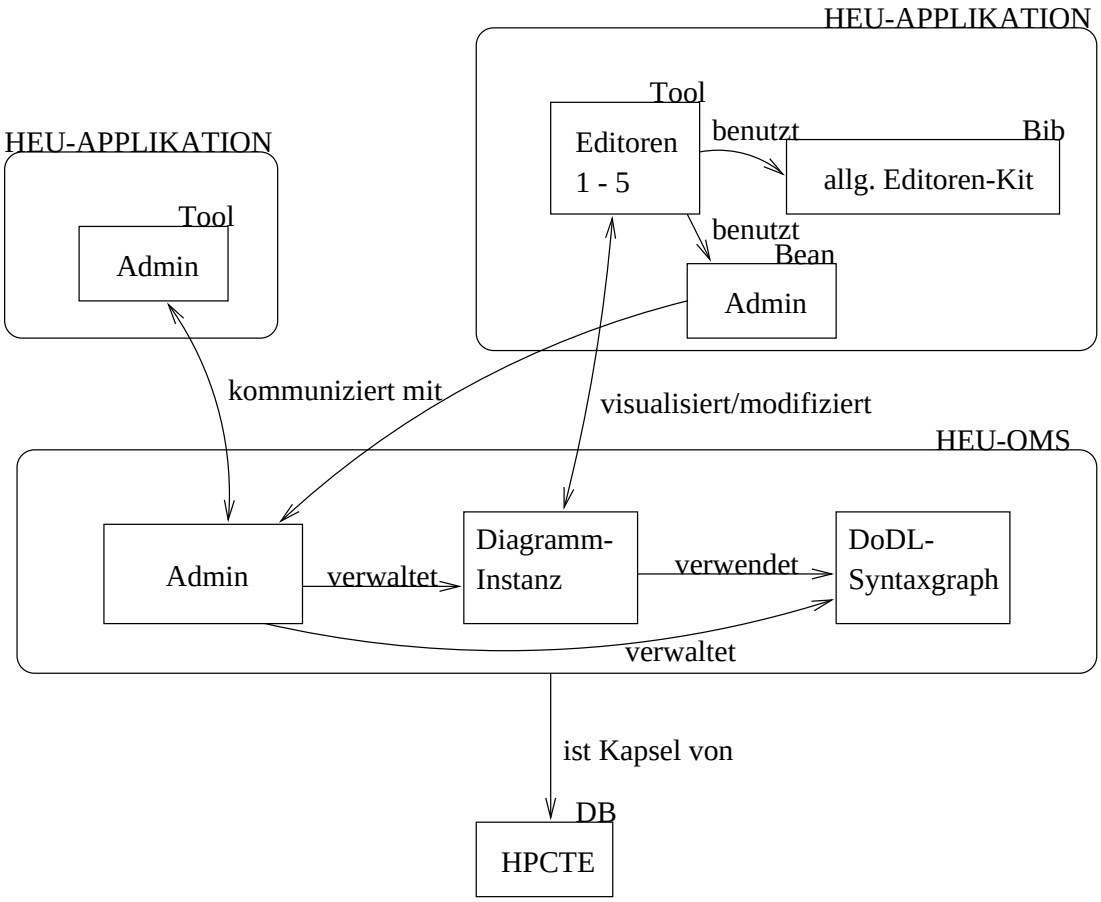


Abbildung 19.1.: Ein erster $\mathcal{HEU}$-Architekturvorschlag

Die Analysephase soll *DoDL* grafisch und textuell betrachten. Wir wollen *Use Cases* bauen, die die Verwendung des grafischen *DoDL* zeigen. Die Editorlandschaft soll ebenfalls in der Analysephase aufgegriffen werden. Jeweils einige Use Cases können grob auf die Editoren abgebildet werden (welcher Editor macht was). Daraus ergeben sich für jeden Use Case ein kleiner und recht einfacher fallbasierter DFA. Insbesondere sollen Multi-User-Aspekte berücksichtigt werden. Weiterhin wird in der Analyse ein Anwendungsszenario erstellt und die hierfür nötigen Use Cases identifiziert. Damit wollen wir eine Entwicklung der Editoren erreichen, die stets den Erstellungsprozeß von Hyperdokumenten vollständig verfolgen kann. Die Hinzunahme weiterer Funktionalität geschieht dann zur Erweiterung der Anwendbarkeit.

Die Editorlandschaft wird auf folgende Editoren festgelegt:

- Ein Klassendiagramm-Editor ermöglicht die Darstellung von Klassenzusammenhängen Aggregation, Vererbung und lokale Klassen. Dabei soll eine UML-ähnliche Notation verwendet werden, in der Attribute und Methoden ebenfalls aufgenom-

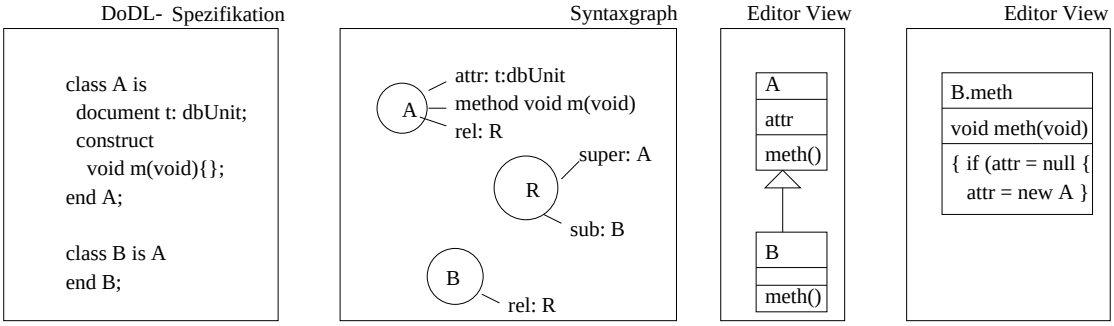


Abbildung 19.2.: Skizze der *DoDL*-Views

men werden.

- Ein Text-basierter Editor ermöglicht die Spezifikation einzelner Klassenmethoden.

Die Editoren arbeiten zusammen, d. h. im Klassendiagramm-Editor kann eine Methode einer Klasse ausgewählt werden, die dann im zweiten Editor „ausprogrammiert“ werden kann. Hier können alle Attribute resp. Dokumente bearbeitet werden, die sich im Sichtbarkeitsbereich der entsprechenden Klasse befinden. Um diesen Sichtbarkeitsbereich bestimmen zu können, zeigen wir in Abbildung 19.3 ein Konturmodell.

Unsere Architektur zeigt die HEU-OMS-Schicht. Hierin soll u. a. ein abstrakter Syntaxgraph die für beide Editoren notwendigen Daten der *DoDL*-Spezifikation aufnehmen. Daraus resultiert die Möglichkeit, die in den jeweiligen Editoren gezeigte Information als Sicht auf diese Spezifikation zu verstehen. Weitere Editoren können dann in die $\mathcal{HEU}$ aufgenommen werden, die ebenfalls auf diesem Syntaxgraphen arbeiten sollen. Ein Klassendiagramm der Datenstruktur abstrakter Syntaxgraph zeigt Abbildung 19.4.

Die Entwurfsphase betrachtet die identifizierten Use Cases genauer. Ihre feingranulare Struktur soll hier aufgedeckt werden. Die Architektur der $\mathcal{HEU}$ muß erarbeitet werden, damit die Editorlandschaft integriert werden kann. Genau hier greifen wir die Seminarphase wieder auf, diesmal aber modifiziert. Wir möchten einen zielgerichteten Workshop veranstalten, auf dem wir die Architektur von $\mathcal{HEU}$ erarbeiten. Das Analysedokument dient dabei als Grundlage des Workshops. Ergebnisse sollen sein 1) die Grundlagen eines Entwurfsdokuments für die $\mathcal{HEU}$ 2) die Integration der Editoren in diese und 3) deren eigene Feinstruktur. Damit bildet der Workshop eine Brücke zwischen Analyse- und Entwurfsphase, was im Hinblick auf eine Zeitersparnis und den Erfolg des ersten Seminars möglich und sinnvoll erscheint.

19.3. Themenvorschläge

Die Seminarthemen sind so ausgewählt, daß sie die Entwicklung der Architektur möglichst effizient unterstützen. Es werden Themen vergeben, die von mehreren HEUlern

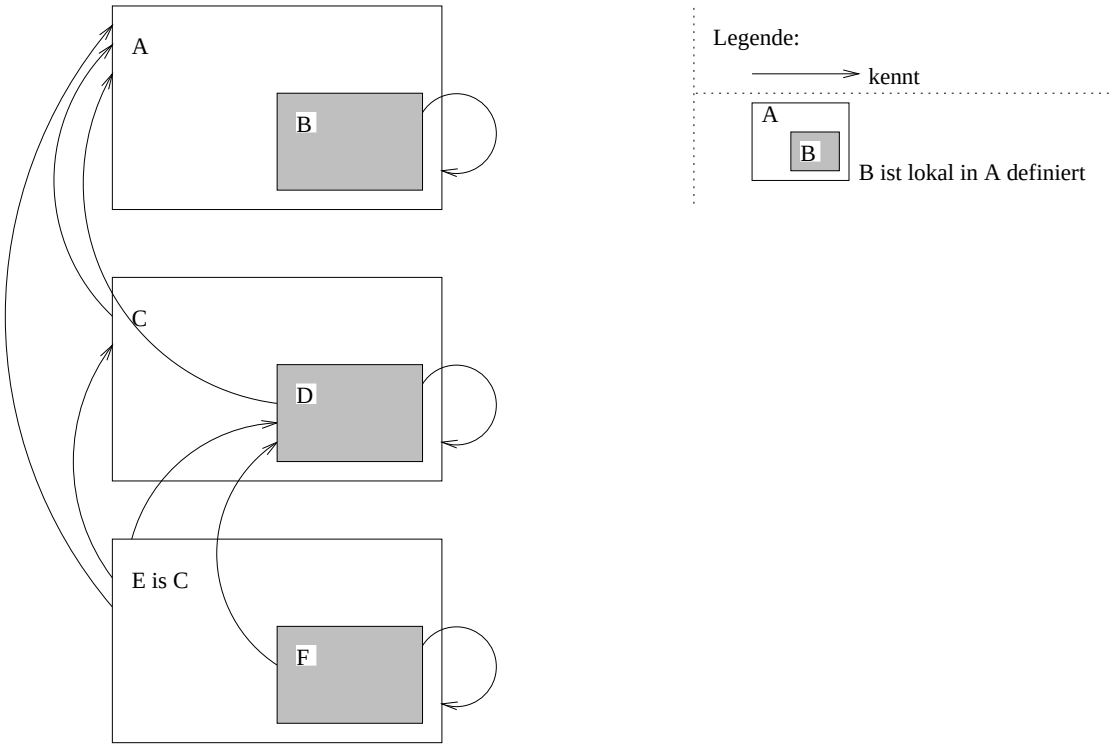


Abbildung 19.3.: Die Scoping-Regeln

gemeinsam bearbeitet werden sollen. Diese dienen dann der Ideenfindung auf dem Workshop. Andere Themen sind technischer Natur und helfen bei der Umsetzung der gewonnenen Ideen. Zu jedem Thema wird hier relevante Literatur angegeben.

1. Architektur von Shared Informations Systems (2 Personen)
   - Kap. 4 aus Shaw/Garlan: Software Architecture
2. Komponenten und multi-tier Architekturen (2 Personen)
   - ACM Computing Surveys, Heft 1, 1998
   - Analysis Patterns von Martin Fowler
   - Distr. Obj. Guide von Orfali et al.
3. JAVA Beans (2 Person)
   - JDK-Dokumentation von Sun zu Java 2
   - JAVA Beans (O'Reilly)
   - Enterprise-Beans (White Paper bei SUN im Web)
   - Distr. Obj. Guide von Orfali et al.

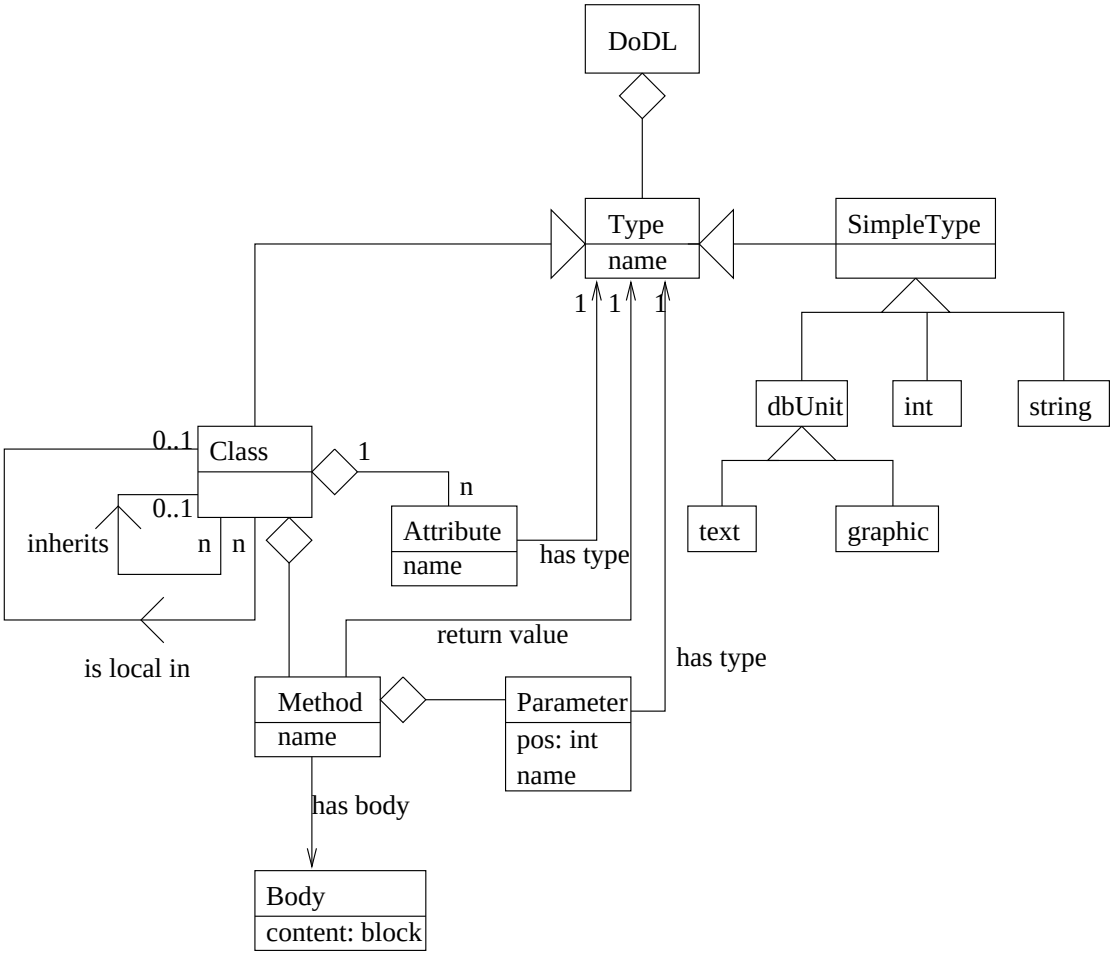


Abbildung 19.4.: Das Klassendiagramm für den abstrakten Syntaxgraphen der HEU-OMS

4. Rechte-Konzepte von H-PCTE (2 Personen)
   - Dokumentation von H-PCTE
5. Transaktionsmanagement für non-standard DBMS (2 Personen)
   - ein jedes (weitergehende) DB-Buch
   - Artikel aus der Merlin-Gruppe
6. APIs von JAVA 2: (2 Personen)
   - Dokumenation von Sun zu JAVA 2
   - Design Patterns

19.4. Workshop-Aufgaben der Teilgruppen

1. Shared Informationssystem
   a) Welche Informationen werden von verschiedenen Anwendungen gemeinsam genutzt?
   b) Prinzipielle Überlegungen zur Schnittstelle für den Zugriff auf diese Informationen (vgl. JDBC als allg. Schnittstelle für relationale DBMS).
   c) Welche Funktionalität braucht man für den Zugriff?
2. Komponenten
   a) Bildung von funktionalen Blöcken, die soweit autonom sind, daß sie (auch) auf unterschiedlichen Rechnern laufen könnten: grobgranulare Sichtweise ist wichtig.
3. Beans
   a) Feingranulare Komponentenbildung, z. B. GUI-Komponenten, aber auch Syntaxgraph-Elemente (wenn's denn geht).
4. Rechte-Konzepte
   a) Was kann H-PCTE?
   b) Was wird davon sinnvoll realisiert?
5. Transaktionsmanagement
   a) Umsetzung von Transaktionskonzepten in H-PCTE: Multiuser-Zugriff ermöglichen.
   b) verschläge, welche Architekturebene für das Transaktionmanagement verantwortlich ist.
6. API's
   a) Kochbuch Programmiersupport

19.5. Zeitplan

Den Zeitplan ermitteln wir rückwärts, ausgehend vom Datum des Campus Festes.

ab 12. Juni 99: primär sind dann nur noch Berichte und Dokumentationen zu verfassen. Für kleinere Reparatur- und Vervollständigungssimplementierungen wird sicher auch noch Zeit bleiben.

am 12. Juni 99: Campus Fest. Das Anwendungsszenario sollte lauffähig sein, die Demos vorbereitet.

ab 26. April 99: Implementierung (ca. 8 Wochen).

6. - 23. April 99: Entwurfsdokumente fertigstellen

18. März - 5. April: Ferien. Wenn die Gruppe einverstanden ist, sollten wir die 12. KW (22. - 26. März) nutzen, um an den Entwurfsdokumenten zu arbeiten. Je früher wir damit fertig sind, umso mehr Zeit bleibt für die Implementierung.

15. - 17. März 99: Workshop in Nordhelle (geplant und möglich!).

12. Februar - 15. März 99: Vorbereitung des Workshops: Vorträge und Ideen, zielgerichtet auf den Architekturentwurf.

28. Januar - 12. Februar 99: Analysedokument fertigstellen (Use Cases erstellen).

bis 28. Januar: Analyseteil *DoDL* und Editorlandschaft fertigstellen. Gleichzeitig werden die wichtigsten Dinge im Prototyp fertiggestellt. Eine Review-Sitzung ist ebenfalls geplant.

20. Die Fassadenschicht der Datenbank

Autor: *Sebastian Schütte*

Das folgende Kapitel basiert auf Überlegungen die im Verlauf des zweiten PG-Seminarwochenendes angestellt wurde. Ziel der Überlegungen war es, die wesentlichen Prinzipien der Datenbankschicht zu dokumentieren. Dazu wurden Schnittstellen definiert, die den Zugriff auf die Datenstruktur der $\mathcal{HEU}$ ermöglichen. Ausgehend von diesen Schnittstellen wurde im Verlauf der weiteren Entwicklung überlegt, wie die angebotenen Dienste zu realisieren sind. Die hier beschriebenen Grundlagen waren die Vorraussetzung zur Planung der höheren Schichten der $\mathcal{HEU}$.

20.1. Fassade der Datenbank Architektur

Es wurde vier Klassen/Schnittstellen identifiziert, die die zu speichernde Information strukturieren sollen. Von diesen Klassen werden Erweiterungen erstellt, die dann die konkreten Daten wie *DoDL*-Klassen, Syntaxgraph, Diagramminformationen, persönliche Arbeitsbereiche, usw. aufnehmen.

Durch die Verwendung gemeinsamer Oberklassen, besteht die Hoffnung bestimmte Code-Teile (wie beispielsweise die Versionsverwaltung) bereits in einer der Oberklassen ansiedeln zu können.

Durch die im folgenden vorgestellten vier Klassen/Interfaces wird eine für alle Spezialisierungen gemeinsam zu verwendende Fassade geschaffen, die für jeden neuen Informationstyp, der im Informationssystem verwaltet werden soll, um spezifische Funktionen erweitert wird.

20.1.1. OMSObject

`OMSObject` ist die Wurzel der Objekthierarchie der Datenbankfassade. `OMSObjekt` soll folgende Funktionalität zur Verfügung stellen:

- Zugriffsrechteverwaltung, Eigentümer
- Versionierung
- Erstellung von Arbeitskopien
- Sicherstellung der Unveränderbarkeit einer eingecheckten Version
- Kommentar, Erstellungsdatum, u.ä.

- Eindeutige Namensvergebung
- Möglichkeit Nutzdaten aufzunehmen

`OMSObjekt` ist abstrakt.

20.1.2. OMSSimple

`OMSSimple` erweitert `OMSObject` um die Funktionalität ein konkretes Datum abspeichern zu können. `OMSSimple` stellt die feinste Granulierung dar, in der im OMS Daten gespeichert werden können. I.d.R. wird `OMSSimple` nicht direkt verwendet werden, sondern eine weitere Erweiterung.

Das Objekt, das in einem `OMSSimple` gespeichert werden soll, kann (und sollte) selbst eine Struktur besitzen. Hier sollen nicht atomare Daten wie Punkte, Texte oder Zahlen gespeichert werden, sondern strukturierte Daten wie eine *DoDL*-Klasse oder Methode, ein komplettes Diagramm o.ä.. Welche Granulierung der Informationsverwaltung für die $\mathcal{HEU}$ gewählt wird, bleibt weiteren Entwurfsüberlegungen vorbehalten. Als Richtlinie sollte jedoch in einem `OMSSimple` gespeichert werden, was in einer dateibasierten Entwicklungsumgebung als eine Datei gespeichert wird bzw. eine Informationseinheit bezeichnet, für die eine Versionierung und länger dauernde Bearbeitung Sinn macht.

20.1.3. OMSCollection

`OMSCollection` erweitert `OMSObject` um die Funktionalität, eine ungeordnete Menge von `OMSObjects` zu erstellen. Dabei werden nur Verweise auf die entsprechenden Objekte bzw. auf eine spezielle Version eines Objektes gespeichert. Auch `OMSCollection` wird in der Regel vor der Verwendung erweiter werden. Der Zugriff auf die Elemente der Menge wird über geeignete Methoden möglich gemacht.

Offen blieb die Frage inwieweit eine geordnete Menge durch ein Objekt des Typs `OMSAggregation` repräsentiert wird oder eine entsprechnde Erweiterung von `OMSCollection`.

20.1.4. OMSAggregation

`OMSAggregation` erweitert `OMSObject` (oder `OMSCollection` (?)) um die Funktionalität eine Menge sowie eine Struktur innerhalb dieser Menge zu verwalten . Zusätzlich zu der Menge können noch getypte, gerichtete, zweistellige Relationen verwaltet werden. Auch füer `OMSAggregation` gilt, es werden keine `OMSObjects` gespeichert, sondern nur Verweise auf diese (`OMSAggregation` läßt sich auch als gerichteter Graph mit getypten Kanten vorstellen).

20.1.4.1. Realisierung der Relation

Ein `OMSAggregation` verwendet getypte Relationen um die Beziehungen zwischen OMS-Objekten zu speichern (siehe 20.1 auf der nächsten Seite). Diese Beziehungsinformation muß nach außen gebracht werden können. Dazu soll ein weiteres Interface (bzw. eine

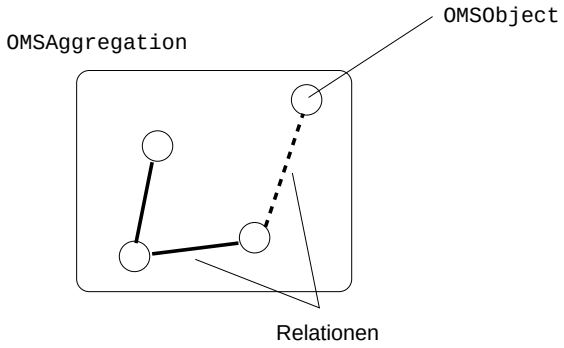


Abbildung 20.1.: Aggregation enthält Objekte und Relationen

Hierarchie von Interfaces) erstellt werden, die den Zugriff auf eine Relation erlauben. Eine innere Klasse des `OMSAggregation`-Objektes kann dann dieses Interface implementieren und die innere Klasse über eine geeignete Methode zur Verfügung stellen (siehe 20.2; vergleiche auch `Iterator`-Interface, [Jav]).

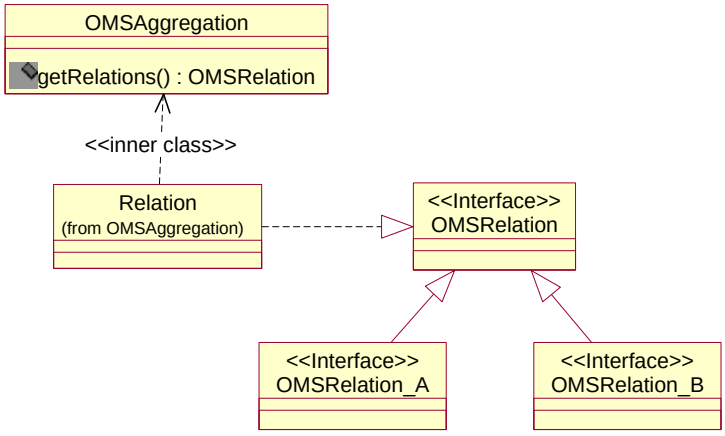


Abbildung 20.2.: Realisierung in Java

20.2. Entwurfsideen

20.2.1. Interfaces statt Objekte

Die oben dargestellte Vererbungshierarchie wurde anfangs als Hierarchie von Klassen diskutiert, später wurde jedoch überlegt, die oben genannten Klassen als Interfaces zu verstehen. Dann kann die Fassade zur OMS-Schicht geordnet sein, ohne das die Ordnung

der Fassade die Struktur der implementierenden Objekte vorgibt.

20.2.2. Versionsverwaltung

Als Grundlage füer die Versionierung wurde folgendes Konzept vorgeschlagen: Von einer Version eines `OMSObjects` können beliebg viele Kopien zur Bearbeitung ausgecheckt werden, es kann allerdings nur eine als Nachfolgeversion wieder eingetragen werden. Die verbleibenden Kopien werden entweder verworfen oder als neuer Zweig (branch) abgelegt. Ein verschmelzen (merge) von zwei Versionen ist nicht vorgesehen.

Um den Versionsbaum abzuspeichern, kann ein Mechanismus eingesetzt werden, der sich seinerseits auf die Erweiterung von `OMSAggregationOMSCollection` abstützt.

20.2.3. TransactionObject

Für die Transaktonskontrolle wird ein extra Interface/Klasse geschaffen, daß die notwendigen Methoden für die Transaktionskontrolle (Beginn, Ende, Abbruch) zur Verfügung stellt. Dazu wird bei `OMSObject` eine Methode zur Verfügung gestellt, die das entsprechende Transaktionsobjekt zurückgibt.

20.2.4. NewObjectManager

Um einen geeigneten Adressaten für die Benachrichtigung über die Erzeugung von neuen Objekten zu besitzen, wurde überlegt, ob ein `NewObjectManager` eingeführt wird, der über neu erstellte Objekte informiert wird und die Erzeugung geeigneter GUI-Repräsentanten übernimmt.

21. Umgebung und Tools der $\mathcal{HEU}$

Autor: *Wilhelm Leibel*

Es wird zwei Tools geben, die man von der UNIX-Shell aufrufen kann. Zum einem die integrierte Entwicklungsumgebung $\mathcal{HEU}$, zum anderen das Administrationstool $\mathcal{HEU}$Admin zum Verwalten der Rechte. In diesem Kapitel wird ein prototypischer Entwurf der Benutzungsoberfläche beschrieben, der aber nicht realisiert wurde.

21.1. $\mathcal{HEU}$

Nach dem Aufruf der Entwicklungsumgebung $\mathcal{HEU}$ muß sich der Benutzer mit seinem Benutzernamen und seinem Kennwort am Entwicklungssystem anmelden (siehe Abbildung 21.1 auf der nächsten Seite). Man gelangt in ein Projektauswahlfenster, in dem der Benutzer das Projekt und die Benutzergruppe auswählen kann. Eventuell bekommt der Benutzer noch die Möglichkeit, die Rechte der Benutzergruppe einzusehen (siehe Abbildung 21.2 auf der nächsten Seite). Nach dem erfolgreichen Auswählen des Projektes öffnet sich der Klassendiagrammeditor mit dem ausgewählten Projekt. Dort hat der Benutzer die Möglichkeit, im Rahmen seiner Benutzerrechte das Klassendiagram zu editieren. Die Menüleiste hat folgende Struktur:

- Projekt
  - Clear (löscht den Arbeitsbereich)
  - Checkin (checkt den aktuellen Arbeitsbereich ins Repository ein)
  - Checkout (kopiert die Version des Repositorys in den Arbeitsbereich)
  - Versionierung (erstellt eine Version des Arbeitsbereiches)
  - Release (erstellt eine Release aller Klassen im Arbeitsbereiches)
  - Dokumentation (erlaubt eine Dokumentation des Projektes)
  - Close (schließt das aktuelle Fenster und öffnet das Projektauswahlfenster)
  - Exit (verläßt die Entwicklungsumgebung)
- Edit
  - Undo (macht die letzte Aktion rückgängig)
  - Redo (macht die letzte Undo-Aktion rückgängig)

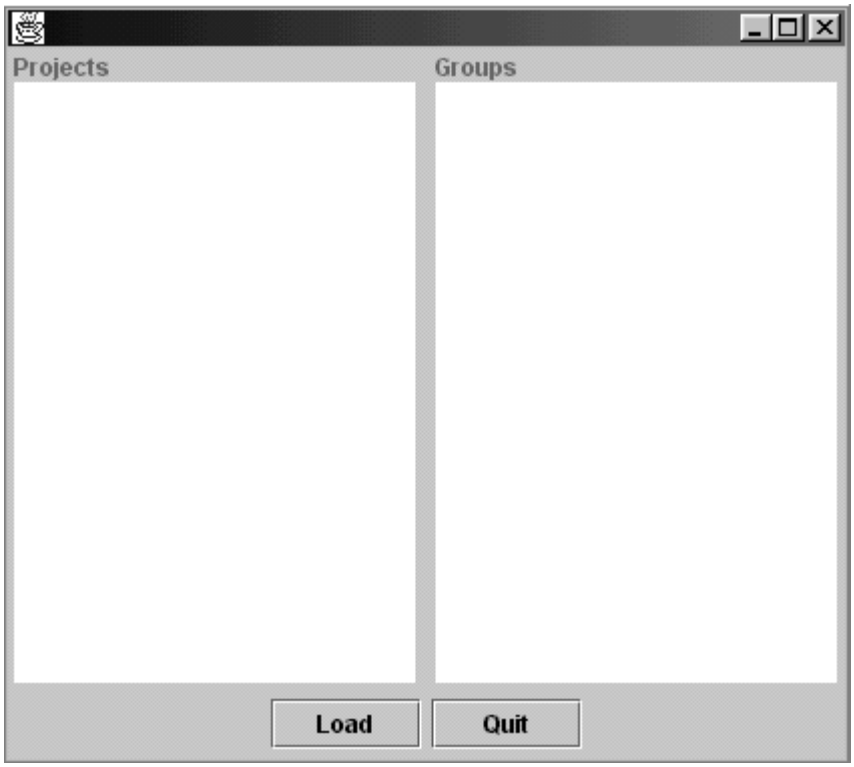


Abbildung 21.2.: ProjectSelection

  - Delete (löscht das selektierte Element)
  - Properties... (öffnet das Properties-Fenster)

- Insert
  - New class (erstellt eine neue Klasse)
  - Clone class (klont eine bestehende Klasse)
  - Used class (referenziert eine bestehende Klasse)
  - Aggregation (erstellt eine neue Aggregation)
  - Inherit (erstellt eine Vererbungsbeziehung)

Am linken Rand des Fensters befindet sich eine Tool-Bar mit Buttons für die wichtigsten Befehlen, die man auch in der Menüleiste wiederfindet:

- New class
- Clone class
- Used class
- Aggregation
- Inherit
- Redo
- Undo
- Del

Enthält eine Klasse lokale Methoden, so erkennt man dies an einem Sternchen in der rechten oberen Ecke der Klasse. Bei einem Doppelklick auf dieses Sternchen öffnet sich erneut ein Klassendiagrammeditor mit den lokalen Klassen.

21.2. Properties

Nach einem Doppelklick auf eine Klasse öffnet sich das Properties-Fenster mit einem Karteikartenreiter und den folgenden Kategorien (siehe Abbildungen 21.3 auf der nächsten Seite und 21.4 auf der nächsten Seite):

Class: Diese Karteikarte ermöglicht das Ändern des Klassennamens.

Attributes: In dieser Eingabemaske kann man Atrribute ändern, hinzufügen und löschen.

Methods: In dieser Karteikarte kann man Methoden ändern, hinzufügen, löschen und editieren. Bei einem Mausklick auf Editieren öffnet sich ein Texteditor, in dem man die Methode programmieren kann.

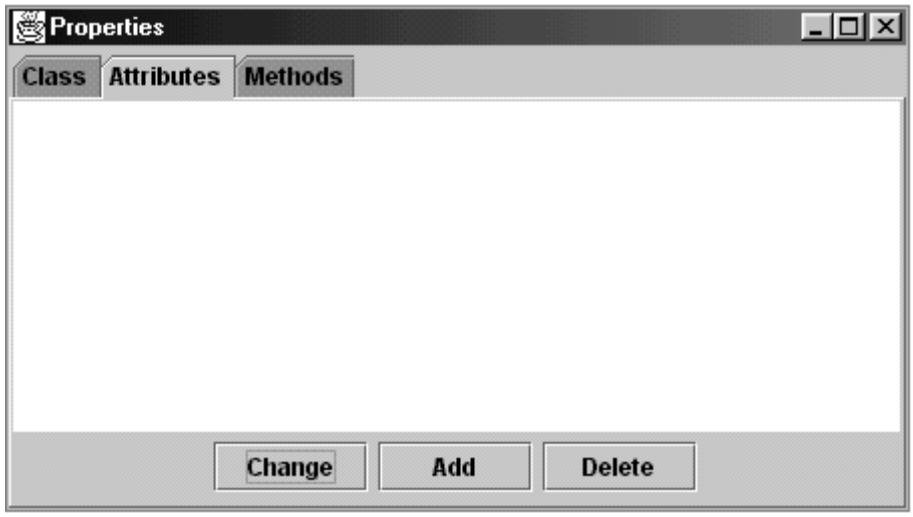


Abbildung 21.3.: Properties: Attributes

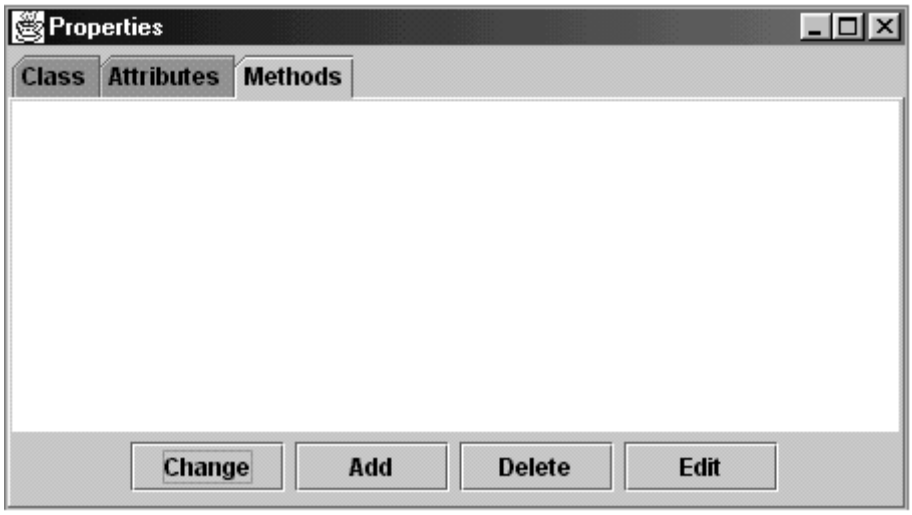


Abbildung 21.4.: Properties: Methods

22. Der Projektmanager

Autor: *Adil Kassabi*

Das Tool "Projekt Manager" dient als Eintrittswerkzeug in die $\mathcal{HEU}$. Dieses Tool ist für eine Reihe von Aktivitäten zuständig. Dazu zählt das Anmelden des Benutzers, Anzeigen von Projekten und das Starten von Tools wie Klassendiagrammeditor, Klasseneditor und Methodeneditor.

22.1. Projektmanager-Gruppe

Die Projektmanager-Gruppe hat sich die groben Komponenten und das ungefähre Aussehen des Projekt- und Werkzeug-Managers ausgedacht. Die graphische Benutzerschnittstelle sollte über drei Listen verfügen. Die erste Liste dient zum Anzeigen des Projektbaumes. Da werden alle Projekte aufgeführt, auf die der Benutzer Zugriff hat. Die zweite Liste zeigt Gruppen an, zu denen der Benutzer bezüglich eines konkreten Projektes gehört. Diese Gruppen werden erst angezeigt, wenn ein bestimmtes Projekt selektiert ist.

Wenn ein Projekt und die Rolle, mit der der Benutzer das Projekt öffnen möchte, bereits selektiert ist, dann werden die Tools angezeigt, mit denen der Benutzer das gewählte Projekt mit der selektierten Rolle öffnen kann.

Zur Unterstützung dieses Ansatzes hat sich die Gruppe auf ein Komponentenmodell geeinigt.

Das Komponentenmodell ist dreischichtig:

- Auf der obersten Schicht gibt es eine Projekt-Toolchooser-Komponente. Diese Komponente ruft eine Rechtemanager-Komponente auf, um die Projekte, Projektbaum, Klassendiagramme und Klassenobjekte aus der Datenbank zu ermitteln, auf die der Benutzer Zugriffsrechte hat.

- Um die Projekte aus der Datenbank herauszuholen, gibt es eine Projektmanagerkomponente.

- Ein Toolmanager informiert den Projekt-Toolchooser über die zur Verfügung stehenden Tools durch Lieferung eines ToolDescription-Objekts für jedes Tool. Jedes ToolDescription-Objekt enthält eine Start-Methode zum Starten des entsprechenden Tools.

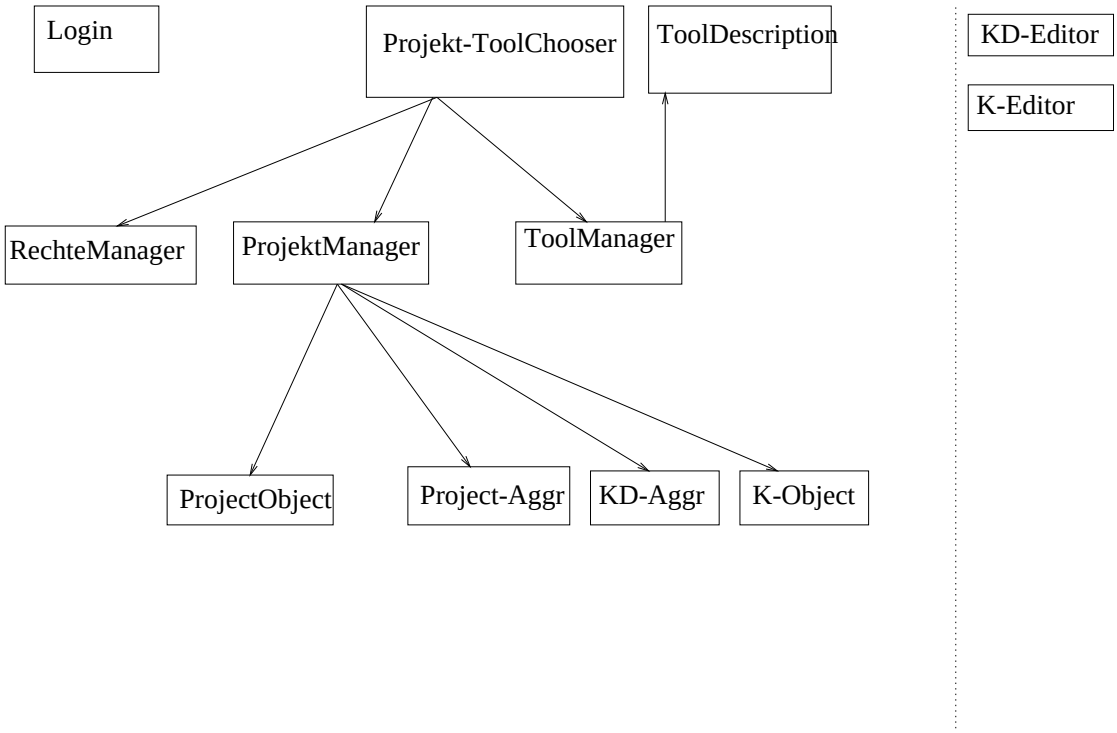


Abbildung 22.1.: Das grobe Komponentenmodell

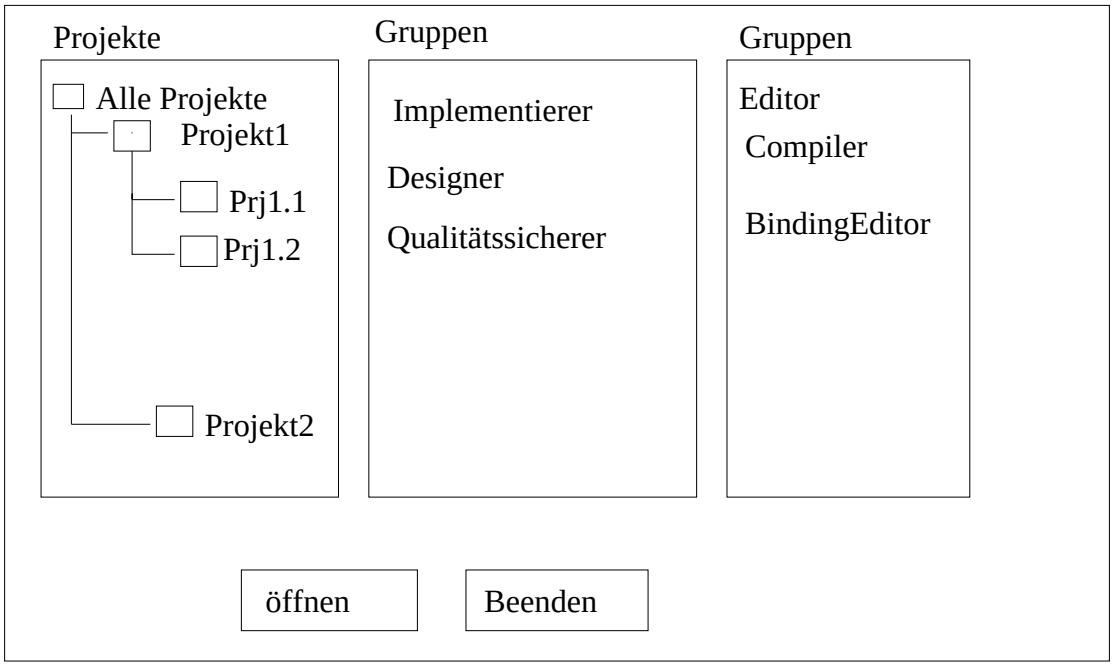


Abbildung 22.2.: Projekt- und Werkzeug-Manager

23. Der Klassendiagrammeditor

Autor: *Sascha Lüdecke*

Dieses Kapitel beinhaltet ein ausformuliertes Protokoll unserer Überlegungen im zweiten Seminar auf Nordhelle zum Klassendiagrammeditor. Die drei Abschnitte behandeln die identifizierte Funktionalität, einen ersten Versuch eines Entwurfes bzw. einer Architektur und weitere Bemerkungen und Fragen, die noch genauer zu klären sind.

23.1. Funktionalität

Als erstes haben wir über die grundlegenden Funktionen nachgedacht, die der Klassendiagrammeditor beherrschen sollte. Die so gefundene Menge wird uns bei weiteren Überlegungen sehr nützlich sein. Wir identifizierten folgende Tätigkeiten, die ein Benutzer mit Hilfe des Editors wenigstens ausführen können sollte:

- Neues Diagramm anlegen oder löschen
- Neues/Anderes Diagramm anzeigen
- Relationen erzeugen/löschen/bearbeiten
- Klassen erzeugen/löschen/bearbeiten

23.2. Erster Versuch einer Architektur

Nach der Aufstellung der funktionalen Anforderungen konnten wir eine Systemarchitektur skizzieren, die vier Schichten enthält, von denen die oberen drei in Bild 23.1 auf der nächsten Seite dargestellt sind. Diese Architektur soll eine erste Näherung an einen endgültigen Entwurf sein und erhebt weder Anspruch auf Vollständigkeit noch auf Korrektheit; eine feinere Ausarbeitung ist Gegenstand weiterer Untersuchungen.

Die unterste Schicht, die konkrete Datenbank, ist nicht aufgezeichnet, da sie für den Klassendiagrammeditor keine Rolle spielen wird, seine Objekte verwenden die Dienstleistungen, die durch die OMS-Schicht bereitgestellt werden. In der OMS-Schicht sollen die Daten möglichst günstig für die Speicherung modelliert werden. Darüber liegt die Modellschicht, die das Datenmodell für die Applikation repräsentiert. Die Abbildung OMS↔Modell ist nicht 1:1; Modelle können aus verschiedenen OMS-Objekten zusammengesetzt sein, z.B. aus `DoDLKlasse` und Dokumentation. Die GUI wiederum bildet das

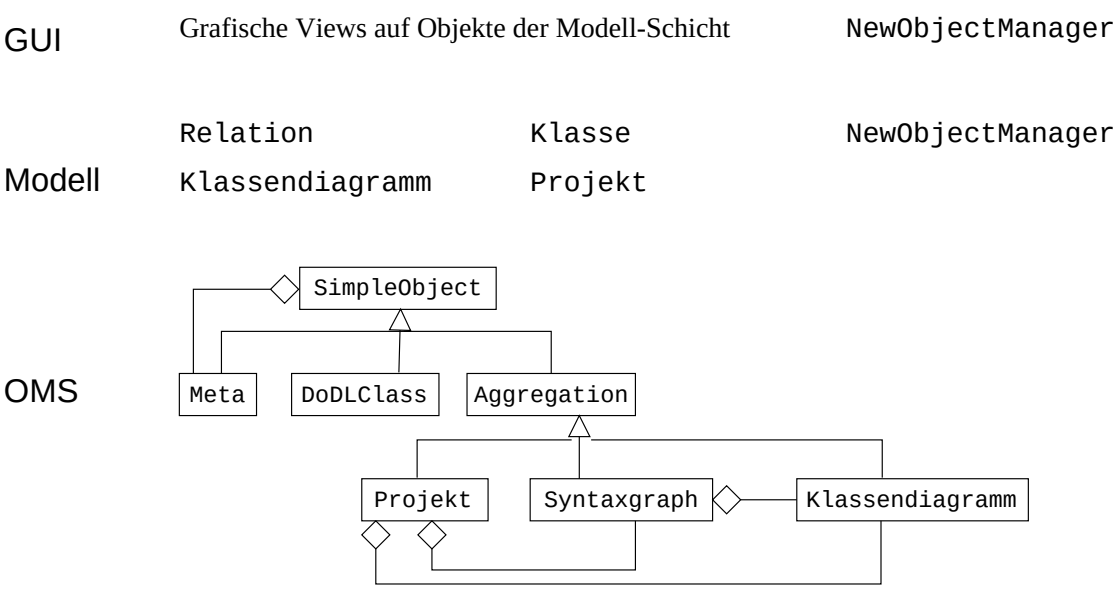


Abbildung 23.1.: Erste Architektur

Datenmodell aus der Modellschicht 1:1 auf dem Bildschirm ab. Eine Beschreibung des endgültigen Entwurfs findet sich in Teil VI auf Seite 209.

23.3. Fragen/Bemerkungen

Nachdem wir eine grobe Architektur gefunden und die wesentliche Funktionalität identifiziert haben, ergab sich ein eher allgemeines Gespräch über Gesichtspunkte des Editierens von Klassen. Die wichtigsten Punkte daraus sollen hier erwähnt werden, da wir sie bei dem weiteren Entwurf berücksichtigen müssen, sie sich jetzt aber noch nicht in der Architektur wiederfinden:

- Gibt es mehrere Diagramme in einem Syntaxgraphen?
  ⇒ Das haben wir bejaht.
- Das Vorhandensein wenigstens einer Main-Methode bei einer der Klassen im Diagramm muß gewährleistet sein.
- Das Modell einer Relation kommuniziert mit dem Modell eines Klassendiagramms, nicht mit der OMS-Schicht, da Relationen dort nicht als eigenständige Objekte modelliert werden. Relationen sind jedoch integraler Bestandteil von Aggregationen.
- Klassen enthalten lokale Klassen. Diese können beliebig geschachtelt werden.
  ⇒ Ein Syntaxgraph enthält mehrere Klassendiagramme.
  ⇒ Klassen mit lokalen Klassen sollen visuell markiert werden.

⇒ Sichtbare Klassen sollen in ein Klassendiagramm kopier- und darin referenzierbar sein, um sie im Editor wiederverwenden zu können.

24. Der Klasseneditor

Autor: *Thomas Sparenberg*

Dieses Kapitel faßt die Überlegungen zusammen, die in Nordhelle von einer Arbeitsgruppe erarbeitet wurden. Es beschreibt den Entwurf des Klasseneditors zum Zeitpunkt des Seminars. Nachträgliche Änderungen im Entwurf bilden dann den Unterschied, den man zum Kapitel 31.2 auf Seite 321 sehen kann. Da der Zeitrahmen in Nordhelle recht eng gesteckt war und es noch einige Verständnisschwierigkeiten zum Thema *DoDL* gab, beschreibt das hier gesagte nicht das volle Ausmaß des Klasseneditors. Es gibt nur einen groben Überblick.

24.1. Funktionaliät

Der Klasseneditor kann das "innere" einer Klasse, also atomare Datentypen, editieren. Dazu zählen Klassenname, atomare Attribute, Methoden und die Dokumentation. Die benutzten und die inneren Klassen (used und inner Classes) können hier nur namentlich angezeigt werden, aber keinesfalls editiert. Dafüer gibt es den Klassendiagrammeditor.

Da man Attribute einer *DoDL*-Klasse editieren können muß, diese aber ggf. als `DBUnit` eine Grafik beinhalten können, werden diese Grafiken etc. genauso wie in *DoDL* nur als Dateiname, also als String, gespeichert. Es wird nur die Information, die man dem *DoDL*-Compiler zu übergeben hat, gesichert. Es macht keinen Sinn, die einzelnen Pixel in der Objektbank zu speichern, denn dann bräuchte man außerdem noch einen Datenbanktreiber für das Dateisystem, damit der Compiler diese Grafiken erreichen könnte, oder müßte sie temporär auslagern und dem Compiler diesen neuen Pfad übergeben. Dann kann man aber die benutzte Grafik nicht einfach gegen eine andere austauschen.

"Normale" atomare Attribute, wie z.B. `int`, werden ebenfalls als Strings gespeichert, und zwar sowohl der Datentyp (hier int) als auch der Name des Attributs. Da diese Daten in der Objektbank sowieso als Strings gespeichert werden, wird hier kein Datenverlust erwartet. Das Parsen obliegt dann dem jeweiligen Editor.

Da es keinen gesonderten Methodeneditor gibt, werden Methoden in einem Frei-Text-Editor bearbeitet und als reiner Text gespeichert. Das betrifft alle Methoden einer Klasse, also auch die ausgezeichneten Methoden Begin und End.

(Anmerkung: im entgültigen Entwurf und in der Implementierung ist ein eigenständiger Methodeneditor entwickelt worden, der ähnlich dem Klasseneditor die Attribute einer Methode bearbeiten kann: Parameter, Returntyp, Returntypkardinalität. Weitere Informationen im Kapitel 31.4 auf Seite 329)

Außerdem wird es zu jeder Klasse und Methode eine textuelle Beschreibung geben, die Dokumentation. Diese Dokumentation ist offensichtlich auch nur Text und wird ebenfalls als String gespeichert werden.

Es läßt sich zusammenfassen, daß der Klasseneditor nur mit String-Einträgen und atomaren Datentypen arbeiten muß. Deshalb werden hier keine besonders großen Schwierigkeiten im Umgang mit den Daten und beim Abspeichern der Daten in der Objektbank erwartet.

24.2. Erste Spezifikation

Im ersten Versuch einer Spezifikation wurden folgende Methoden identifiziert, die der Klasseneditor haben sollte:

1. um den Namen einer Klasse lesen und setzten zu können, gibt es folgende Methoden:
   a) getName
   b) setName
2. um Informationen über die dargestellte Klasse zu erfahren, kann man folgende Methoden benutzen:
   a) getUsedClasses (Namen der) benutzte Klassen lesen
   b) getAggregatedClasses (Namen der) aggregierten Klassen lesen
3. zum Editieren von Klassenattributen:
   a) getAttributes auslesen aller Klassenattribute
   b) setAttribute einzelnes Attribut verändern
   c) createAttribute ein Attribut hinzufügen
   d) deleteAttribute ein Attribut löschen
4. zum Editieren der Klassenmethoden: (teilweise veraltet)
   a) getMethods Liste aller Methoden auslesen
   b) setMethod Methode verändern
   c) createMethod Methode anlegen
   d) deleteMethod Methode löschen
   e) getBegin-/EndMethod die ausgezeichneten Methoden `Begin` und `End` auslesen

   Die besondere Behandlung von `Begin` und `End` sind später entfallen, da man einfach Methoden selben Namens anlegen kann, und schon ist die ganze Sonderbehandlung überflüssig.
5. Methoden für die Klassendokumentation:
   a) getDoku
   b) setDoku

Teil VI.

Allgemeine $\mathcal{HEU}$-Architektur

25. Die Drei-Schichten-Architektur

Autor: *Christoph Begall*

In diesem Abschnitt wird die grobe Untergliederung der $\mathcal{HEU}$ beschrieben, wobei auf die Gründe für diese Unterteilung und ihre Vor- und Nachteile eingegangen wird. Es wird nicht die Aufgabe von einzelnen Klassen oder Packeten erläutert, sondern die Klassen und Packete werden zu Gruppen bzw. Schichten zusammengefaßt, so daß Konzepte, die für die gesamte Architektur gelten, einheitlich beschrieben werden können.

25.1. Intention von Multi-Tier-Architekturen

Bei der Entwicklung der Ampelstadt haben wir viele Erfahrungen mit dem Model-View-Controller Entwurfsmuster gesammelt. Die Datenbank bzw. die direkt darauf aufsetzenden Java-Klassen stellten das Modell für die `Dr`-Klassen, die *Views*, dar (siehe Abschnitt 17.2.1 auf Seite 146). Eines der Probleme, die sich aus dieser Architektur ergaben, war, daß diese *Model*-Klassen zwei, zum Teil widersprüchliche Aufgaben erfüllen mußten:

1. Die zugrundeliegende Datenorganisation mußte abgebildet, also das Datenbankschema mit einer Java-Schnittstelle versehen werden.

2. Den *Views* mußte ein für sie passendes *Model* geboten werden, in denen sie, quasi als eine Dienstleistung, ihre Daten abspeichern konnten.

Um dieses Problem zu entschärfen, wurde in der $\mathcal{HEU}$ eine Drei-Schichten-Architektur eingeführt: OMS, Modell und GUI, die strikt getrennt sind, und nur über den JavaBeans Event-Mechanismus miteinander kommunizieren sollen. Dies soll zu einer Entkopplung der Aufgabenbereiche und so zu einer klaren Verteilung der Zuständigkeiten führen.

Neben der schon erwähnten Entlastung der untersten Schicht, gab es noch weitere Intentionen hinter dieser Einteilung. Zunächst einmal sollen die klaren Zuständigkeiten den Entwurf und die Implementierung vereinfachen und Redundanzen wie auch „Löcher“, also Probleme, für die sich niemand zuständig fühlt, vermeiden. Dadurch wird eine stark verteilte Entwicklung möglich, die die Effizienz bei der Entwicklung erhöhen soll.

Desweiteren war in einem frühen Architektur-Entwurf die Benutzung von CORBA zur Kommunikation der einzelnen Komponenten untereinander vorgesehen. Da sich dies in der vorliegenden Implementierung der $\mathcal{HEU}$ aus Zeitgründen nicht verwirklichen ließ, mußte eine Möglichkeit geschaffen werden, dieses Konzept auch später noch einzubauen. Die vorliegende Schichteneinteilung und die damit verbundene Entkopplung erleichtern

es, auch nachträglich noch zwischen den einzelnen Schichten eine Kommunikation via CORBA einzuführen. Auf ähnliche Art und Weise sollte jede der einzelnen Schichten ohne Auswirkungen auf die anderen ersetzt oder erweitert werden können.

25.2. Die drei Schichten

In folgenden Kapiteln wird bei der Beschreibung der Kommunikation bzw. Interaktion der von Objekten der einzelnen Schichten immer wieder eine Richtung („nach oben", „nach unten") erwähnt. Dies wird einsichtiger wenn man sich bei der Einteilung der Schichten eine horizontale Teilung wie in Bild 25.1 vorstellt. Im weiteren werden die Aufgaben der einzelnen Schichten betrachtet, wobei „von unten nach oben" vorgegangen wird.

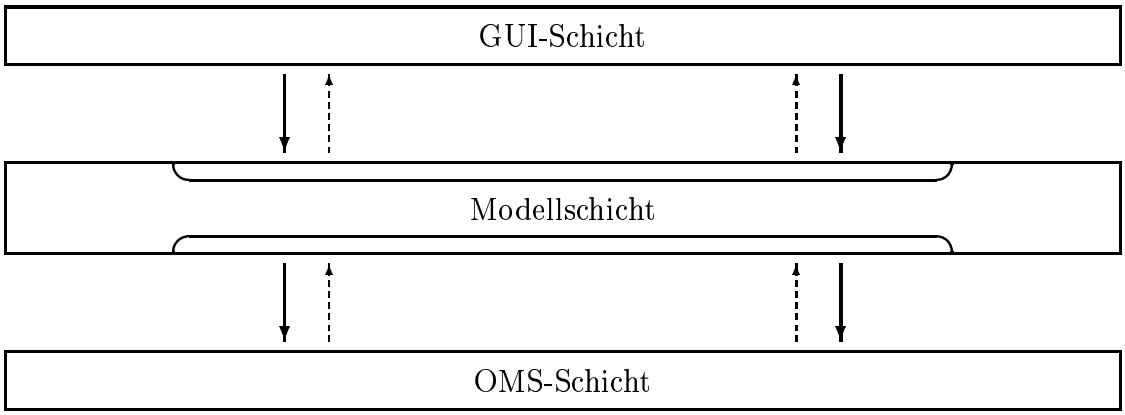


Abbildung 25.1.: Die drei Schichten der $\mathcal{HEU}$: OMS, Modell und GUI

25.2.1. Die OMS-Schicht

Die Aufgabe der OMS-Schicht ist es dabei, vollen Zugriff auf die Datenbank zu geben, also die Struktur der Datenhaltung zugänglich zu machen. Desweiteren soll die OMS-Schicht jedoch von spezifischen Datenbank-Management Systemen (DBMS) abstrahieren, so daß ein solches zu einem späteren Zeitpunkt mit möglichst geringem Aufwand auszutauschen ist.

Bei der Implementierung der $\mathcal{HEU}$ wird wie schon bei der Ampelstadt das Datenbank-Management-System H-PCTE benutzt, wobei die Einzelheiten dieses Systems vor dem Benutzer weitgehend verborgen bleiben sollen.

25.2.2. Die Modellschicht

Die Modellschicht ist dafür verantwortlich, die Daten der OMS-Schicht für die darüberliegenden Views optimal zu präsentieren, Algorithmen auf den Daten zu implementieren, und, wo immer möglich, die Komplexität der OMS-Schicht zu verbergen.

Diese Schicht sollte insgesamt vier Gesichtspunkten gerecht werden:

1. Sie soll möglichst 1:1 die tatsächlichen Entitäten – also die *DoDL*-Klassen – und ihre Struktur abbilden.

2. Sie muß „nach unten“ als eine Menge von Views zur Kommunikation mit den OMS-Interfaces, die sich an der Struktur der Datenbank orientierten, agieren.

3. Sie muß „nach oben“ möglichst direkt für *Controller* und *Model* der GUI (Swing) verwendbare Funktionalität zur Verfügung stellen, die über die eigentlich modellierte Struktur hinausgeht bzw. von ihr abweicht.

4. Sie ist für die Realisierung von Algorithmen auf den Daten zuständig, z.B. die Ermittlung aller für eine bestimmte *DoDL*-Klasse sichtbaren Typen.

25.2.3. Die GUI-Schicht

Die GUI-Schicht enthält sowohl die Views als auch die Controller des klassischen MVC-Konzepts, sofern diese nicht ohnehin aufgrund der Benutzung von Swing zusammenfallen. Hier werden einerseits die Benutzereingaben verarbeitet und die entsprechenden Befehle zur Datenänderung nach unten in die Modellschicht gesendet, andererseits werden auch die Daten und deren Änderungen für den Benutzer dargestellt. Die Benutzung der Swing-Bibliothek zur Programmierung der grafischen Benutzungsschnittstelle ist besonders für die Umsetzung des Model-View-Controller-Entwurfsmusters geeignet.

25.3. Weitere Untergliederung der Schichten

Neben dieser globalen Trennung in die drei bisher identifizierten Schichten GUI, Modell und OMS wurde die $\mathcal{HEU}$ in noch feinere Strukturen aufgeteilt.

25.3.1. Allgemeine, *DoDL*- und Werkzeug-spezifische Klassen

Da die $\mathcal{HEU}$ einerseits nicht auf die Entwicklung in bzw. mit *DoDL* beschränkt ist, andererseits die einzelnen Werkzeuge, die mit *DoDL* arbeiten, sehr unterschiedliche Sichten auf den von ihnen bearbeiteten Teil haben, wurden die im Bild 25.1 auf der vorherigen Seite horizontal veranschaulichten Schichten noch einmal vertikal geteilt. Diese Trennung findet sich in der Hierarchie der Klassen und Packete wieder: Jedes Werkzeug hat seine eigenen OMS-, Modell- und GUI-Packete. Auf der allgemeineren Ebene der *DoDL*-Modellierung, wie auch auf oberster Ebene, die die grundlegende, für alle Teile der $\mathcal{HEU}$ wichtigen Teile umfaßt, gibt es eine ebensolche Unterteilung, wobei sich hier keine eigenen GUI-Packete befinden.

Diese speziellere Untergliederung soll in der Abbildung 25.2 auf der nächsten Seite verdeutlicht werden.

Auf dieser allgemeinen Ebene ist auch der generische Editor anzusiedeln, der als Service für alle $\mathcal{HEU}$-Werkzeuge zur Verfügung steht, von seiner Aufgabe her aber nicht klar

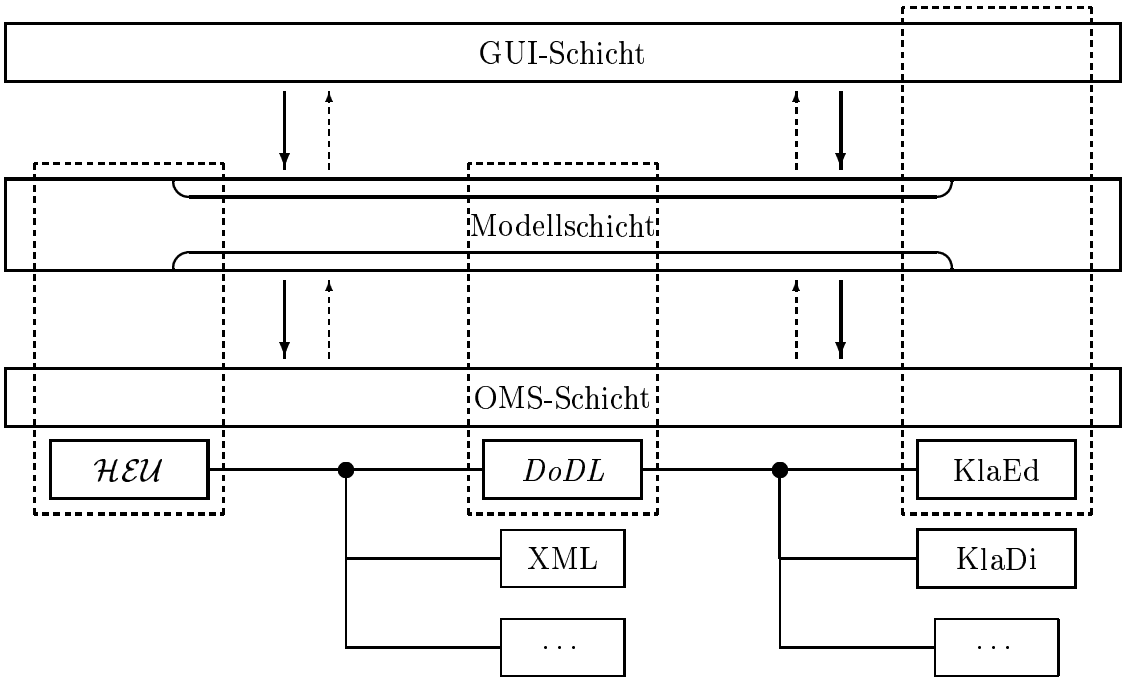


Abbildung 25.2.: Hierarchische, vertikale Unterteilung der Schichten

einer allgemeinen GUI- oder Modellschicht zuzuordnen ist. Ähnliche allgemeine Funktionsmodule auf GUI-Ebene, etwa zur Erleichterung der Kommunikation mit Swing, könnten auf ähnliche Weise in der allgemeinen $\mathcal{HEU}$-Ebene angesiedelt werden.

25.3.2. OMS-Fassade und JHpcte-Implementierung

Um die Datenbank möglichst ohne Änderungen in Modell- und GUI-Schicht austauschbar zu machen, bilden die OMS-Interfaces zusammen mit wenigen zur Instanziierung notwendigen Klassen eine Fassade, hinter der die implementierenden JHPcte-Klassen vollständig versteckt sind. Dies wirkt zwar zunächst so, als sei sogar die Bezeichnung „Vier-Schichten-Architektur“ gerechtfertigt, erweist sich aber bei genauerem Hinsehen als herausragende Eigenschaft der OMS-Schicht per se. Denn die schon beschriebenen Aufgaben dieser Schicht können nur durch die Auftrennung in OMS- und JHPcte-Teil erfüllt werden. Als Funktionsblock wirkt sie weiterhin homogen.

25.3.3. Adapter zwischen GUI und Modell

Auf GUI-Ebene wurden insbesondere für aufwendigere Containerstrukturen wie Tabellen und Bäume zusätzliche Datenstrukturen benötigt. Hier haben wir Adapterklassen verwendet, um die Objekte aus der Modell- bzw. OMS-Schicht in Instanzen der Klassen `Table`- und `TreeModel` verwalten zu können.

Diese Zwischenstücke, die zwischen der eigentlichen Oberfläche und der Modellschicht

vermitteln, sind notwendig, um die Komplexität und Funktionalität aus den mit VisaJ erstellten GUI-Klassen herauszuhalten.

25.4. Die Umsetzung

Bei der Umsetzung der hier beschriebenen Architektur gab es einige Schwierigkeiten, und manche Entwurfsziele konnten nicht in der in diesem Abschnitt vorgestellten Klarheit umgesetzt werden. Dies soll im weiteren hier erläutert werden, insofern es sich nicht um speziellere Aspekte der Implementierung handelt, die in den jeweiligen Kapiteln (z.B. bei der Beschreibung der Modellschicht in Kapitel 28 auf Seite 285 noch behandelt werden.

25.4.1. Probleme bei der Aufteilung der Zuständigkeiten

Wie schon in der Einführung 25.1 auf Seite 209 beschrieben, war die Intention der hier beschriebenen Architektur unter anderem, daß eine verteilte, möglichst unabhängige Entwicklung stattfinden sollte. Dazu ist eine sehr genaue Beschreibung der Aufgaben der einzelnen Schichten und Packete notwendig, die aus Zeitgründen so nicht erstellt werden konnte. So mußten solche Aufgabenzuteilungen während der Implementierung geklärt werden, was natürlich zusätzlichen Kommunikationsaufwand zur Folge hatte.

25.4.2. Versteckte Komplexität der Modellschicht

Bei der Definition der einzelnen Schichten wurde die Modellschicht als relativ einfach zu implementierender Bereich betrachtet. Die im Abschnitt 25.2.2 auf Seite 210 beschriebenen Aufgaben implizieren jedoch eine Aufgabenfülle, die über eine reine Proxy-Funktionalität weit hinausgeht. Durch eine bessere Planung wäre es möglich gewesen, gemeinsame Funktionalität zu erkennen und auszulagern, um die Implementierung dieser Schicht zu vereinfachen.

25.4.3. Fehlende Service-Klassen und -Packete für GUI-Schicht

Ähnlich wie in der Modellschicht wurde auch in der GUI-Schicht aus Zeitgründen auf eine genaue Analyse der gemeinsamen Funktionalität verzichtet, so daß sich die Komplexität auf Werkzeugebene *staute*. Auch hier wäre es sicherlich möglich gewesen, gewisse Serviceleistungen, die für andere Werkzeuge von Nutzen sein können, zu identifizieren, und in entsprechenden Packeten auszulagern. Dies trüge sicherlich zur leichteren Verständlichkeit und Übersichtlichkeit der Architektur bei.

25.5. Dynamik zwischen den Schichten

Autor: *Matthias Dorka*

Dieses Kapitel beschreibt die dynamischen Abläufe innerhalb der $\mathcal{HEU}$. Es erklärt die verwendeten Kommunikationsmechanismen innerhalb und zwischen den Schichten

der Architektur und zeigt insbesondere die Stellen in der Implementierung auf, an denen sie verwendet werden.

25.5.1. Notwendige Kommunikation zwischen den Schichten

Im vorangegangenen Kapitel 25 auf Seite 209 wurde ausführlich erläutert, welche Vorteile Mehrschichtarchitekturen mit sich bringen und welche Möglichkeiten sie eröffnen. Dies erfordert natürlich bei der Implementierung die Beachtung einiger Regeln, um die durch die Architektur gewonnene Flexibilität und Anpaßbarkeit nicht wieder zu verlieren.

Das trifft insbesondere auf die Kommunikation zwischen den in der Analysephase gefundenen Schichten zu. Eine der Grundregeln lautet hier, die Schnittstellen der Schichten und damit ihrer Komponenten klein zu halten und zum anderen die zwischen diesen Komponenten ausgetauschten Informationen auf ein Mindestmaß zu beschränken. Dieser Anspruch ergibt sich beispielsweise aus der Tatsache, daß sich zwischen zwei Schichten ein Netzwerk befinden kann, dessen physikalischer Verkehr natürlich klein gehalten werden soll.

Zum anderen deutet eine zu starke Verflechtung zwischen Programmpaketen auf einen nicht gekonnt durchgeführten Komponentenentwurf hin, bei dem die interne Kohäsion groß gegenüber der externen Kopplung sein soll (Zweite Seminarphase Nordhelle: Vortrag über Komponenten-Technologien). Einzelne Komponenten, die eine abgeschlossene Funktionalität über klar definierte Schnittstellen anbieten, können i. d. R. leichter wiederverwendet werden, als Teile aus monolithischen Programmblöcken. Diese Grundidee ist es, die durch den objektorientierten Ansatz in Analyse und Entwurf von Software ja auch verfolgt wird und die sich bei Java im *Bean*-Konzept widerspiegelt, welches als Orientierungsleitfaden für den Entwurf der $\mathcal{HEU}$ diente.

Nachteil des Java-eigenen *Bean*-Konzeptes ist die Tatsache, daß es sich dabei lediglich um eine Programmierrichtlinie handelt, die einige konzeptionelle Anforderungen stellt, diese aber kaum weiter technisch unterstützt. So gibt es beispielsweise keine abstrakte *Bean*-Klasse, von der alle *Beans* erben müssen, um eine minimale einheitliche Schnittstelle zu besitzen (Zweite Seminarphase Nordhelle: Vortrag über JavaBeans). In der PG $\mathcal{HEU}$ wurde der Fehler gemacht, sich zu sehr auf das schlecht verstandene *Bean*-Konzept zu verlassen, ohne dies weiter zu erarbeiten.

Besser verstanden und in hohem Maße genutzt (bzw. Nutzung geplant) wurden dagegen die Java-Mechanismen *Event* und *Exception*. So besteht die Kopplung der Komponenten zwischen den Schichten fast überall aus Methodenaufrufen in Richtung der jeweils zugrundeliegenden Schicht und *Events* bzw. *Exceptions* in der Gegenrichtung. Letztgenannte machen diese Kopplung zu einer flexiblen und ausbaufähigen Verbindung, da *Events* als universelle Benachrichtigungen gesehen werden können, die von jedem Programmstück "abgefangen„ werden können, das sich als *Eventlistener* bei diesem Objekt anmeldet. Dies erleichtert beispielsweise das Hinzufügen neuer Tools auf GUI-Ebene, da diese sich an den gleichen Objekten der Modellschicht anmelden und auf deren Änderungen reagieren könnten, ohne das dort eine Sourcecode-Änderung notwendig wird.

Aus diesem Grunde folgt die Implementierung der $\mathcal{HEU}$ im Wesentlichen diesem Grundsatz, was in den nachfolgenden Abschnitten gezeigt werden soll. Dabei gehen wir

zunächst allgemein auf die verwendenten Methoden sowie *Events* und *Exceptions* und deren Zweck ein, ehe abschließend die Abfolge von Kommunikationsprozessen an zwei Beispielen detailliert beschrieben wird.

25.5.1.1. Kommunikation innerhalb der OMS-Schicht

Der Kommunikation innerhalb er OMS-Schicht wird an dieser Stelle ein eigenes Unterkapitel gewidmet, da es sich hierbei nicht um eine homogene Schicht im oben beschriebenen Sinne handelt, sondern vielmehr um zwei einzelne Schichten, die zwar beide mit der Datenhaltung in der Datenbank zu tun haben, dies jedoch auf unterschiedliche Weise. Man könnte die Architektur der $\mathcal{HEU}$ also durchaus als Vier-Schicht-Architektur bezeichnen, wir haben sie jedoch stets als Drei-Schicht-Architektur mit geteilter OMS-Schicht verstanden. Diese Einteilung ist in Abb. 25.3 dargestellt.

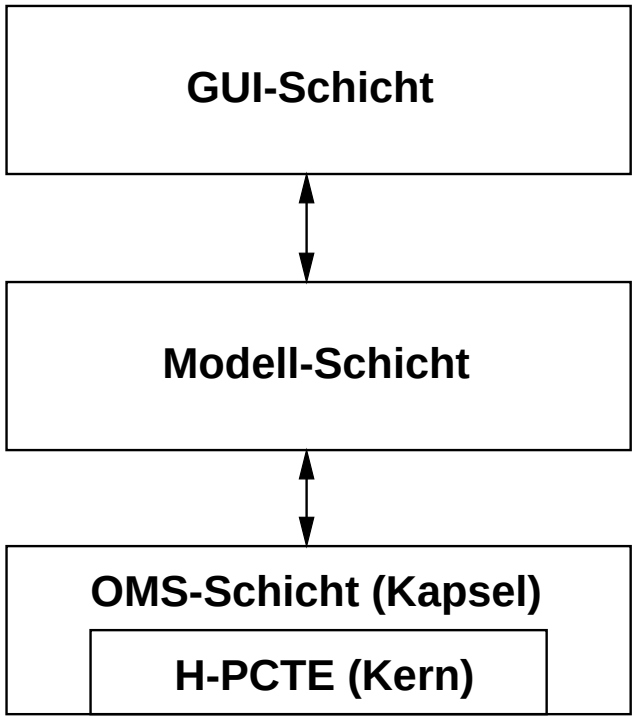


Abbildung 25.3.: Die Schichtenarchitektur der $\mathcal{HEU}$

Die Notwendigkeit hierzu ergibt sich aus der Tatsache, daß die verwendete Datenbank H-PCTE von uns über ihr Java-API angesprochen wird, eine Sammlung von Java-Klassen, die eine Vielzahl grundlegender Funktionen für die Datenhaltung und das Datenmanagement beinhalten. Diese Klassen, deren Namen i. A. mit `HPCTE` beginnen, stellen jedoch nur allgemeine Funktionalität auf unterster Datenbankebene zur Verfügung. Funktionen, die darauf aufbauend eine direkte Manipulation gemäß des von uns verwendeten Datenmodells erlauben, befinden sich in den `JHPcte`-Klassen, die die das Datenmodell

abbildenden OMS-Interfaces implementieren. Dieser Sachverhalt ist im Kapitel 20 auf Seite 189 detailliert beschrieben und die Kenntnis der dort erklärten Zusammenhänge wird für den Rest dieses Abschnitts vorausgesetzt.

Die wesentlichen Methoden des OMS-Kerns, die von der OMS-Kapsel aufgerufen werden, betreffen das Neuanlegen und Löschen von Datenbankobjekten und der dazwischenliegenden Links zur persistenten Speicherung der grundlegenden Objekte (`JHPcteObjekt` und alle Kinder) sowie das Navigieren durch diese Objektwelt und sind nachfolgend aufgezählt:

- `Pcte_object_create` - Erzeugt ein neues Pcte-Objekt
- `HPcte_object_create_and_set_several_attributes` - Erzeugt ein neues Pcte-Objekt, wobei gleichzeitig Attribute gesetzt werden können
- `Pcte_object_delete` - Löscht ein Pcte-Objekt
- `Pcte_object_set_attribute` - Setzt das Attribut eines Pcte-Objekts auf einen Wert
- `Pcte_object_get_attribute` - Liefert zu einem Pcte-Objekt den Wert eines bestimmten Attributs zurück
- `Pcte_object_get_type` - Liefert den Typ eines Pcte-Objekts als String zurück
- `Pcte_link_create` - Erzeugt einen neuen Link zwischen Pcte-Objekten
- `Pcte_link_delete` - Löscht einen Link zwischen Pcte-Objekten
- `Pcte_object_list_links_of_types` - Liefert die von einem Pcte-Objekt ausgehenden Links eines bestimmten Typs
- `Pcte_link_get_reverse` - Liefert den Link in Gegenrichtung zu einem Link, sofern vorhanden
- `Pcte_object_reference_set_relative` - Erzeugt einen Pcte-Link auf ein Objekt von einem bestehenden Link aus
- `Pcte_object_reference_set_absolute` - Erzugt einen absoluten Pcte-Link auf ein Objekt
- `HPcte_object_references_are_equal` - Prüft, ob zwei Referenzen auf das gleiche Objekt in der Datenbank zeigen
- `HPcte_lwp_login_create_and_start` - Erzeugt einen Leichtgewichtprozess auf Datenbank-Ebene
- `HPcte_process_set_ws_via_string` - Setzt das Arbeitsschema des aufrufenden Prozesses auf das Arbeitschema, dessen Name übergeben wird

- `Pcte_activity_start` - Startet einen in sich abgeschlossenen Zeitraum für Transaktionen
- `Pcte_activity_abort` - Beendet den Transaktionszeitraum so, daß Transaktionen, die noch nicht per COMMIT abgeschlossen wurden, verworfen werden (entspricht klassischem ROLLBACK)
- `Pcte_activity_end` - Beendet den Transaktionszeitraum regulär (unter Ausführung eines COMMIT)
- `Pcte_transaction_set_savepoint` - Setzt einen Sicherungspunkt innerhalb einer Transaktion
- `Pcte_transaction_get_savepoints_for_undo` - Liefert die für ein "Undo" möglichen Sicherungspunkte zurück
- `Pcte_transaction_undo_to_last_savepoint` - Macht alle Operationen seit dem Setzen des letzten Sicherungspunktes rückgängig
- `Pcte_transaction_undo_to_previous_savepoint` - Macht alle Operationen seit dem Setzen des vorherigen Sicherungspunktes rückgängig
- `Pcte_transaction_redo_possible` - Aktiviert die Mögllichkeit eines "Redo"
- `Pcte_transaction_redo_to_next_savepoint` - Wiederhölt rückgängig gemachte Aktionen bis zum nachfolgenden Sicherungspunkt

In Gegenrichtung agieren von Pcte geworfene Exceptions als Benachrichtigungen in erster Linie für den Nichterfolg der oben aufgeführten Datenbankaktionen. Zur Kategorisierung und Weiterreichung dieser Exceptions dient eine Hierarchie von OMS-Exceptions, die alle von der Java-eigenen `RuntimeException` erben, und die in Abb. 27.4 auf Seite 249 unten dargestellt ist.

Die allgemeine `OMSErrorException`, die als einzige eigene Methode `getPcteErrorException` zum zurückliefern der Nummer der aufgetretenen `PcteException` besitzt, wird erweitert zu den spezielleren Ausnahmefällen `OMSDBLostException`, `OMSTransactionException` und `OMSInvalidObjectException`.

- `OMSDBLostException` wird ausgelöst, wenn ein schwerwiegender Fehler innerhalb der Datenbank auftritt, der jeglichen weiteren Zugriff darauf unmöglich macht.
- `OMSTransactionException` zeigt an, daß ein Fehler im Zusammenhang mit Transaktionen aufgetreten ist.
- `OMSInvalidObjectException` dient als Indikator für den Zugriff auf ungültige Objekte, beispielsweise der Versuch des Auslesens von Objektattributen eines bereits gelöschten Objekts.

Daneben existiert noch eine Hierarchie von `OMSVeteExceptions`, die für das Weiterreichen weniger schwerwiegender Ausnahmefälle vorgesehen ist und bei denen es sich um Erweiterungen der Java-eigenen `PropertyVetoException` handelt. Diese Hierarchie ist in Abb. 27.4 auf Seite 249 oben dargestellt.

- `OMSDBRightsException` zeigt eine Rechteverletzung an, beispielsweise ist der nachfragende Benutzer nicht berechtigt, auf das angeforderte Objekt lesend zuzugreifen.
- `OMSVersioningException` indiziert einen Fehler im Zusammenhang mit der Versionskontrolle, beispielsweise den Zugriff auf ein Objekt, das mittlerweile in einer neueren Version vorliegt.
- `OMSLockingException` ist dafür gedacht, den Zugriff auf ein z. B. zum Schreiben gesperrtes Objekt anzuzeigen.

Eine genaue Beschreibung der in der $\mathcal{HEU}$ geplanten Features Versionskontrolle und Rechteverwaltung ist in den entsprechenden Kapiteln zu finden. Es sei an dieser Stelle jedoch angemerkt, daß sich oben beschriebene Exceptionmechanismen nur ansatzweise realisiert in der $\mathcal{HEU}$ wiederfinden, da einerseits Versionierung und Zugriffsverwaltung als sekundäre Implementierungsziele verstanden wurden und andererseits einer der Schwachpunkte der Implementierungsphase in der mangelhaften Absprache der Entwickler bezüglich Werfen und Abfangen von Exceptions und Events lag.

So befinden sich die zu den oben genannten Exceptions gehörenden Java-Klassen zwar im Paket `heu.oms` und werden größtenteils auch geworfen, jedoch in diesen Fällen von den zu benachrichtigenden Klassen nicht immer abgefangen bzw. interpretiert und ausgewertet.

Zur regulären Kommunikation während der Laufzeit des Systems, die nicht einer eher ungewollten Ausnahmebehandlung entsprechen soll, verwendet die $\mathcal{HEU}$ den H-PCTE-eigenen Mechanismus der *Notifier*. Diese Möglichkeit, in der Datenbank gespeicherte Objekte mit Notifzierern zu versehen, die beispielsweise die Modifikation eines Attributes propagieren, war ein wesentlicher Grund für die Verwendung von H-Pcte als objektorientierter Datenbank. Ein wesentlicher Vorteil, der sich daraus ergibt, ist die gute Eignung von H-PCTE als Basis einer verteilten Entwicklungsumgebung, da Notifikationen automatisch an alle das Objekt beobachtenden anderen Objekte geschickt werden.

Zur Realisierung bietet die H-PCTE-API Methoden wie `HPcte_notifier_create` an, die einen Notifizierer zu einem Datenbankobjekt erzeugt, der bei Veränderungen an Attributwerten des Objekts in Analogie zum Java-Event-Mechanismus Notifizierungen aussendet, die von den beobachtenden Objekten ausgewertet werden können.

Ein typisches Beispiel für die Verwendung der Notifier ist etwa die Änderung eines *DoDL*-Klassenattributs, das gerade in zwei Editor-Fenstern zur gleichen Zeit zu sehen ist. Sollte ein Benutzer schreibend auf das Attribut zugreifen, so wird dieser Änderungswunsch als Methodenaufruf von der Benutzungsoberfläche an die korrespondierenden `ModelDoDLClass` weitergereicht, von wo aus ein weiterer Aufruf an das entsprechende OMS-Objekt stattfindet, das dann via JHPcte-Objekt das Datenbankobjekt modifiziert. Diese Veränderung wird vom Notifier des Objekts propagiert und den umgekehrten Weg

als `PropertyChangeEvent` nach oben weitergereicht, woraufhin sich die Objekte der Modellschicht und schließlich die auf dem Bildschirm dargestellten Swing-Elemente ändern, und dies natürlich an allen Clients. Eine ausführlichere Beschreibung dieser Abläufe mit veranschaulichendem Sequenzdiagramm befindet sich am Ende dieses Kapitels.

25.5.1.2. Kommunikation zwischen OMS- und Modell-Schicht

Bei der Kommunikation zwischen OMS- und Modell-Schicht handelt es sich im Gegensatz zum vorherigen Abschnitt um wirklich schichtenübergreifende Kommunikation im Sinne softwaretechnologischer Architekturschichten. Wärend die OMS-Schicht, wie schon erwähnt, die Aufgabe hat, für eine persistente Datenspeicherung zu sorgen, soll die Modell-Schicht den Großteil der Applikationslogik bezüglich der modellierten Objekte beherbergen, d. h. in unserem Falle, *DoDL*-Klassen und ihre Beziehungen abzubilden und die möglichen Operationen darauf zu ermöglichen.

Dies bedeutet für die Kommunikation zwischen diesen Schichten einen Bruch in Hinsicht der verwendeten Modelle. Änderungen an Objekten z. B. vom Typ `ModelDoDLClass`, die den texuellen *DoDL*-Klassen am ehesten entsprechen, müssen abgebildet werden auf die zur Verfügung stehenden Manipulationsmethoden der Objekte der OMS-Schicht, die sich am gewählten Datenmodell orientieren und damit i. A. nicht direkt denen der Modell-Schicht entsprechen. Somit ist hier eine Art Mapping von Operationen notwendig, das dieser Aufgabe gerecht wird.

Der Aufbau der OMS-Schicht ist an dieser Stelle jedoch genauer zu betrachten. Er ist nicht so generisch, daß mit wenigen Objekttypen alle Konstrukte der zu speichernden Sprachen gehandhabt werden können, sondern besteht vielmehr aus einem generischen Teil sehr allgemeiner Klassen (z. B. `OMSObject` und `OMSSet` in `heu.oms`), die in einer Vererbungshierarchie verfeinert wurden in Richtung einer den Anforderungen von DoDL genügenden Klassenmenge, die insbesondere `OMSDoDLClass`, `OMSDoDLMethod`, `OMSDoDLClassAggregation`, `OMSDoDLClassGeneralization` und weitere beinhaltet (Vgl. Kapitel 27.2 auf Seite 241). So sind es auch die Methoden dieser Klassen, die von den Modellobjekten aufgerufen werden, und die nachfolgend aufgeführt sind:

- In `OMSDoDLClass`:
  - `isInnerClass` - Liefert zurück, ob es sich bei der *DoDL*-Klasse um eine innere Klasse handelt
  - `getActualOuterClass` - Liefert die umgebende *DoDL*-Klasse
- In `OMSDoDLMethod`:
  - `getContainingClasses` - Liefert die Klassen zurück, die diese Methode enthalten
  - `getBody` - Liefert den Methodenquelltext als Zeichenkette
  - `setBody` - Weist einen Methodenquelltext einer Methode zu

In `OMSDoDLClassAggregation(WorkingCopy)`:

  - `getDoDLAggragations` - Liefert alle Aggregationsbeziehungen der *DoDL*-Klasse
  - `createDoDLAggragations` - Erstellt eine Aggregationsbeziehung zu einer anderen *DoDL*-Klasse
  - `getDoDLInnerClassRelations` - Liefert alle Beziehungen zu enthaltenen *DoDL*-Klassen
  - `createDoDLInnerClassRelations` - Liefert alle Beziehungen zu enthaltenen *DoDL*-Klassen
  - `getDoDLGeneralizations` - Liefert alle Vererbungsbeziehungen der *DoDL*-Klasse
  - `createDoDLGeneralizations` - Liefert alle Vererbungsbeziehungen der *DoDL*-Klasse

  In `OMSDoDLAggregation(WorkingCopy)`:

  - `getName` - Liefert den Namen der Aggregationsbeziehung
  - `setName` - Setzt den Namen der Aggregationsbeziehung
  - `getCardinality` - Liefert die Kardinalität der Aggregationsbeziehung
  - `setCardinality` - Setzt die Kardinalität der Aggregationsbeziehung

Darüber hinaus stehen folgende Methoden zur Verfügung, mit denen sich Objekte der Modellschicht als *EventListener* bei `OMSDoDL`-Objekten anmelden können. Mittels dieser Events wird die Kommunikation in Gegenrichtung realisiert.

- In `OMSDoDLClassAggregation`:
  - `<add|remove>DoDLAggregationListener` - An- und Abmelden als *EventListener* für Veränderungen an den Aggregationen einer `OMSDoDLClass`
  - `<add|remove>DoDLGeneralizationListener` - An- und Abmelden als *EventListener* für Veränderungen an den Generalisierungen einer `OMSDoDLClass`
  - `<add|remove>DoDLInnerClassListener` - An- und Abmelden als *EventListener* für Veränderungen bezüglich innerer Klassen einer `OMSDoDLClass`
- In `OMSDoDLClassAggregation`:
  - `<add|remove>BodyListener` - An- und Abmelden als *EventListener* für Veränderungen an den Methodeninhalten einer `OMSDoDLMethod`

Die sich bei diesen OMS-Objekten anmeldenden Modellobjekte sind im Wesentlichen `ModelDoDLClass` und `ModelDoDLMethod` aus dem Paket `heu.dodl.model`, die als *DoDL*-spezifische Spezialisierungen des allgemeinen `ModelObject` aus `heu.model` insbesondere dessen Eigenschaft als `PropertyChangeEventListener` erben. Hier müssen alle Events der OMS-Schicht abgefangen und verarbeitet werden, die für die Modellobjekte

von Bedeutung sind, was im Rahmen der Projektgruppe jedoch nur unvollständig und teilweise unausgereift implementiert wurde. Ein nicht geringer Teil der fehlenden Funktionaliät des Gesamtsystems beruht auf den hier und und in der nächsthöheren Schicht offen gelassenen Lücken in der Eventverarbeitung.

25.5.1.3. Kommunikation zwischen Modell-Schicht und Benutzungsschnittstelle

Die Methoden und Mechanismen zur Kommunikation zwischen Modell-Schicht und Benutzungsschnittstelle bilden die komplexeste aller Kommunikationsebenen der $\mathcal{HEU}$. Dies liegt zum einen an der Aufspaltung des Informationsflusses von einem Objekt auf Modellebene hin zu potentiell mehreren Werkzeugen auf GUI-Ebene, die es beobachten, zum anderen kann das von uns verwendete *Model-View-Controller*-Konzept (siehe Kapitel 33 auf Seite 345) an dieser Stelle nicht so geradlinig angewendet werden, wie beispielsweise zwischen OMS- und Modellschicht oder in der Ampelstadt, da die hier zum Einsatz kommenden Swing-Elemente von Java zur Visualisierung auf der Oberfläche selbst nach dem *MVC*-Konzept arbeiten, dabei jedoch ihre ganz spezifischen Modellobjekte benötigen.

Deshalb findet sich bereits in der statischen Architektur der $\mathcal{HEU}$ die etwas ungewöhnliche Lösung eines asymmetrisch geschachtelten *MVC*-Musters, die sich gleichermaßen in den Wegen der ausgetauschten Nachrichten niederschlägt. Während Änderungen an den Bildschirmobjekten zu Methodenaufrufen bei den korrespondierenden Objekten auf Modellebene führen, geschieht die rückwärtige Benachrichtigung der Oberflächenelemente über den Umweg der von uns GUI-Adapter genannten Klassen, die einerseits *View*-Objekte der Modellschichtobjekte sind, andererseits aber auch selbst als "Zwischen"-Modelle für die angezeigten Oberflächenelemente dienen. Dieser Zusammenhang wird im Kapitel 29.4 auf Seite 306 näher erläutert.

Nachfolgend werden die wesentlichen Methoden der Klassen der Modellschicht aufgeführt, die von den Werkzeugen der GUI-Schicht aufgerufen werden, um die Eigenschaften der modellierten Objekte abzurufen und Manipulationen daran durchzuführen. Die generischen Modellobjekte wie `ModelObject` und `ModelException` befinden sich im Paket `heu.model`, die davon abgeleiteten *DoDL*-spezifischen Spezialisierungen wie `ModelDodlClass` und `ModelDodlMethod` in `heu.dodl.model`.

- In `ModelObject`:
  - `checkin` - Schreibt das auf Modellebene bearbeitete Objekt in die Datenbank. Damit wird es persistent und für andere Editoren sichtbar.
  - `checkout` - Macht das Objekt editierbar, indem eine veränderbare Arbeitskopie angelegt wird
  - `close` - Gibt das Objekt frei, d. h. es kann von anderen Editoren bearbeitet werden
  - `delete` - Löscht das Objekt (auch auf Datenbank-Ebene!)
  - `equals` - Liefert "true" zurück, falls zwei Objekte gleich sind
  - `register` - Führt eine Anmeldung des Modellobjekts als `EventListener` bei den korrespondierenden Objekten auf OMS-Ebene durch

  - `deregister` - Meldet das Modellobjekt in obigem Sinne bei den OMS-Objekten ab, sofern keine *Listener* der GUI-Schicht mehr registriert sind
  - `dismiss` - Verwirft das aktuelle Objekt und alle Änderungen daran
  - `lock` - Sperrt das Objekt
  - `unlock` - Hebt die Sperre auf dem Objekt auf
  - `getCreationDate` - Liefert das Erzeugungsdatum des Objekts zurück
  - `getDBObject` - Liefert das das Modellobejkt repräsentierende OMS-Objekt zurück
  - `setDBObject` - ordnet dem Modellobjekt ein OMS-Objekt zu
  - `getDBProject` - Liefert das Datanbankprojekt zurück, zu dem das aktuelle Objekt gehört
  - `getDocumentation` - Liefert die Dokumentation des Objekts als Zeichenkette zurück
  - `setDocumentation` - Fügt eine Dokumentationszeichenkette zu einem Objekt hinzu
  - `getID` - Liefert die eindeutige Datenbank-ID eines Objekts als Zeichenkette zurück
  - `getLocker` - Liefert den Namen des Benutzers, der das Objekt gesperrt hält
  - `getName` - Liefert den Namen des Modellobjekts zurück
  - `setName` - Setzt den Namen des Modellobjekts
  - `getNumberOfListeners` - Liefert die Anzahl der an diesem Modellobjekt angemeldeten *Listener* zurück
  - `getVersionNumber` - Liefert die Versionsnummer des Objektes zurück
  - `isDeprecated` - Liefert "true" zurück, falls der Objektzustand nicht mehr dem neusten Stand entspricht
  - `isDirty` - Liefert "true" zurück, falls der Objektzustand verändert wurde
  - `isEditable` - Liefert "true" zurück, falls das Objekt als Arbeitskopie und damit editierbar vorliegt

- In `ModelDoDLClass`:
  - `getAggregation` - Liefert eine Aggregationsbeziehung, also ein Attribut der *DoDL*-Klasse mit einem bestimmten Namen zurück
  - `getAggregations` - Liefert alle Attribute dieser Klasse zurück
  - `insertAggregation` - Fügt ein Attribut zur Klasse hinzu
  - `removeAggregation` - Löscht ein Attribut einer Klasse
  - `replaceAggregation` - Ersetzt ein Attribut durch ein anderes
  - `getMethod` - Liefert eine Methode mit einem bestimmten Namen zurück

- `getMethods` - Liefert alle Methoden dieser Klasse zurück
- `insertMethod` - Fügt eine Methode zur Klasse hinzu
- `removeMethod` - Löscht eine Methode dieser Klasse
- `replaceMethod` - Ersetzt eine Methode
- `getInnerClasses` - Liefert alle inneren Klassen dieser Klasse zurück
- `addInnerClass` - Fügt eine neue innere Klasse in eine Klasse ein
- `removeInnerClass` - Löscht eine innere Klasse
- `isInnerClass` - Liefert zurück, ob diese Klasse die innere Klasse einer anderen Klasse ist
- `getOuterClass` - Liefert die umgebende Klasse zurück, falls vorhanden
- `getRelationWithDest` - Liefert die Relationsbeziehung zurück, die das angegebene Objekt als Ziel hat
- `getSuperClass` - Liefert die direkte Superklasse der vorliegenden Klasse zurück
- `setSuperClass` - Setzt die Superklasse dieser Klasse
- `removeSuperClass` - Löscht die Superklasse dieser Klasse
- `getSuperClassOfClass` - Liefert die Superklasse der übergebenen Klasse zurück
- `getVisibleClasses` - Liefert alle hier sichtbaren Klassen zurück

In `ModelDoDLMethod`:

- `getBody` - Liefert den Rumpf einer Methode als Zeichenkette
- `setBody` - Setzt den Rumpf einer Methode als Zeichenkette, die übergeben wird
- `getContainingClass` - Liefert die umgebende Klasse zu einer Methode zurück
- `getParameters` - Liefert ein Feld mit allen Parametern der Methode zurück
- `insertParameter` - Erzeugt einen Parameter dieser Methode
- `removeParameter` - Löscht einen bestimmten Parameter dieser Methode
- `replaceParameter` - Ueberschreibt einen bestimmten Parameter dieser Methode
- `getReturnType` - Liefert den Rückgabetyp einer Methode
- `setReturnType` - Hiermit setzt man den Typ des Rückgabewertes dieser Methode

Neben diesen Methoden zur Manipulation der Modell-*DoDL*-Objekte existieren in jeder dieser Klassen Methoden, die der Eventverarbeitung mit den benachbarten Schichten dienen. Hierbei handelt es sich neben den `add-` und `removeListener`-Methoden, die von den Objekten der GUI-Schicht aufgerufen werden, um sich als zu Benachrichtigender an-

und abzumelden, auch um `fire`-Methoden, die innerhalb der Modellschicht selbst aufgerufen werden, um Veränderungen in Richtung Benutzungsoberfläche propagieren zu können.

Die Notwendigkeit, eine Veränderung an den zugrundeliegenden Daten weiterkommunizieren zu müssen, sollte dabei im Allgemeinen von der Methode `propertyChange` erkannt werden, die es in allen Klassen der Modellschicht gibt, die Modellobjekte repräsentieren, und die einen `propertyChangeListener` zu implementieren hat.

Alle die Eventverarbeitung betreffenden Methoden der oben erwähnten Klassen sind nachfolgend aufgeführt, wobei in `ModelObject` wieder jene Methoden stecken, die allen Modellobjekten zumindest im Prinzip gemein sind, teilweise werden sie jedoch in den konkreten Modellobjektklassen dem Kontext entsprechend überschrieben.

- In `ModelObject`:
  - `<add|remove>DeprecatedListener` - An- und Abmelden als *EventListener* für die Veralterung eines Objekts
  - `<add|remove>DocumentationListener` - An- und Abmelden als *EventListener* für die Veränderungen an der Dokumentation des Objekts
  - `<add|remove>LockerListener` - An- und Abmelden als *EventListener* für Veränderungen desjenigen Benutzers, der dieses Objekt gesperrt hat
  - `<add|remove>NameListener` - An- und Abmelden als *EventListener* für Veränderungen des Namens des Objekts
  - `<add|remove>VersionListener` - An- und Abmelden als *EventListener* für Veränderungen der Version des Objekts
  - `fireDeprecated` - Sendet allen Listenern, die sich für die Veralterung angemeldet haben, eine Notifikation als Event verpackt
  - `fireDocumentationChanged` - Sendet allen *Listenern*, die sich fuer die Dokumentationsänderung angemeldet haben, eine Notifikation als Event verpackt
  - `fireLockerChanged` - Sendet allen *Listenern*, die sich fuer die Sperrendenveränderung angemeldet haben, eine Notifikation als Event verpackt
  - `fireNameChanged` - Sendet allen *Listenern*, die sich fuer die Namensveränderung angemeldet haben, eine Notifikation als Event verpackt
  - `fireVersionChanged` - Sendet allen *Listenern*, die sich fuer die Versionsänderung angemeldet haben, eine Notifikation als Event verpackt
- In `ModelDoDLClass`:
  - `<add|remove>AggregationsListener` - An- und Abmelden als *EventListener* für die Veränderung an Attributen
  - `<add|remove>InnerClassesListener` - An- und Abmelden als *EventListener* für die Veränderung an inneren Klassen

- `<add|remove>MethodsListener` - An- und Abmelden als *EventListener* für die Veränderung an Methoden
- `<add|remove>SuperClassListener` - An- und Abmelden als *EventListener* für die Veränderung an der Superklasse
- `<add|remove>VisibleClassesListener` - An- und Abmelden als *EventListener* für die Veränderung an den sichtbaren Klassen
- `fireAggregationsChanged` - Sendet allen *Listenern*, die sich fuer die Attributveränderung angemeldet haben, eine Notifikation als Event verpackt
- `fireInnerClassesChanged` - Sendet allen *Listenern*, die sich fuer die Veränderung an inneren Klassen angemeldet haben, eine Notifikation als Event verpackt
- `fireMethodsChanged` - Sendet allen *Listenern*, die sich fuer die Methodenveränderung angemeldet haben, eine Notifikation als Event verpackt
- `fireSuperClassChanged` - Sendet allen *Listenern*, die sich fuer die Veränderung an der Superklasse angemeldet haben, eine Notifikation als Event verpackt
- `fireVisibleClassesChanged` - Sendet allen *Listenern*, die sich fuer die Veränderung an den sichtbaren Klassen angemeldet haben, eine Notifikation als Event verpackt

In `ModelDoDLMethod`:

- `<add|remove>BodyListener` - An- und Abmelden als *EventListener* für die Veränderung an Methodenquelltext
- `<add|remove>ParametersListener` - An- und Abmelden als *EventListener* für die Veränderung an den Methodenparametern
- `<add|remove>ReturnTypeListener` - An- und Abmelden als *EventListener* für die Veränderung am Rückgabewert
- `fireBodyChanged` - Sendet allen *Listenern*, die sich fuer die Veränderung am Methodenquelltext angemeldet haben, eine Notifikation als Event verpackt
- `fireParametersChanged` - Sendet allen *Listenern*, die sich fuer Parameterveränderungen angemeldet haben, eine Notifikation als Event verpackt
- `fireReturnTypeChanged` - Sendet allen *Listenern*, die sich fuer die Rückgabewertveränderung angemeldet haben, eine Notifikation als Event verpackt

Darüber hinaus besitzen die konkreten Modellobjekte `ModelDoDLClass` und `ModelDoLDMethod` noch jeweils eine Methode `listen()`, deren Aufruf für die Anmeldung dieser Modellobjekte selbst als *Listener* bei allen anderen Objekten sorgt, deren Veränderung von Interesse ist.

Neben der regulären Kommunikation, die gewollte Veränderungen an Objekten bekannt macht, gibt es auch hier wieder die Kommunikationsform der *Exception*, die im Fehlerfall Anwendung findet.

Hierfür befinden sich in `heu.model` die generische `ModelException` und die beiden davon erbenden Spezialisierungen `ModelRightsException` und `ModelVersioningException`.

- `ModelRightsException` tritt auf, wenn eine ändernde Operation nicht durchgeführt werden konnte.

- `ModelRightsException` zeigt an, das keine neue Version oder Arbeitskopie eines Objektes erzeugt werden konnte.

Diese *Exceptions* dienen dazu, um im Fehlerfall die Werkzeuge der GUI-Schicht zu benachrichtigen, damit diese auch in Ausnahemsituationen zumindest mit einer Statusmeldung reagieren können.

25.5.2. Realisierter Informationsaustausch zwischen den Schichten

In diesem Abschnitt wird der realisierte Teil der Kommunikationsmechanismen der $\mathcal{HEU}$ beschrieben, welcher nicht all dem entspricht, was in den vorangegangenen Abschnitten vorgestellt wurde. Zwar bestehen alle oben erwähnten Klassen, Methoden, *Events* und *Exceptions*, die Differenz zum Entwurf ergibt sich in der Realisierung im Wesentlichen dadurch, daß nicht alles vollständig implementiert wurde. An verschiedenen Stellen fehlen beispielsweise Methodenrümpfe oder bestehende *Events* werden an keiner anderen Stelle benutzt. Dieser Umstand ist im Quelltext i. A. durch entsprechende Kommentare deutlich gemacht.

Darüberhinaus sind einzelne Aspekte der Kommunikation noch nicht einmal ansatzweise verwirklicht, da die Implementierung gar nicht bis in den Bereich dieser nur grob angedachten Funktionalitäten reichte (z. B. Zugriffsrechteverwaltung auf Objekte). Deshalb wird nachfolgend an zwei Beispielen protokollhaft aufgeführt, welche Dynamik-Aspekte im lauffähigen Teil der $\mathcal{HEU}$zum Tragen kommen.

25.5.2.1. Start des Systems

Der Start der $\mathcal{HEU}$ geschieht mittels Projektmanager wie im Kapitel 30 auf Seite 315 beschrieben. Hierbei wirkt dieses Werkzeug als universeller Starter der anderen Werkzeuge. Die Einzelheiten der Geschehnisse sind nachfolgend aufgeführt, wobei in diesem Beispielfall die Datenbank bereits mit zwei Projekten, die jeweils zwei Klassen enthalten, gefüllt ist:

- Über die Klasse `OMSRootSessionSingleton` erzeugt der Projektmanager eine initaile Datenbankverbindung `RootSession`, die mittels dieser Implementierung des *Singleton*-Entwurfsmusters nur einmal erzeugt werden kann.

- Der Projektmanager meldet sich selbst als *Listener* für Veränderungen an Projekten bei dieser `RootSession` an.

- Es werden Objekte des Typs `DefaultMutableTreeNode` für Projekte, Klassen und Methoden erzeugt, die später an Objekte der Datenbank gebunden werden und als Elemente der Baumdarstellung auf der linken Seite des Projektmanagers dienen.

- Mittels der Methode `getProjects` der `RootSession` werden alle Projekte ermittelt, die z. Z. in der Datenbank gespeichert sind. Dabei werden den gefundenen Objekten entsprechende Java-Objekte erzeugt und gegebenenfalls die hierfür notwendigen Notifizierer auf Datenbankebene.

- Der Projektmanager meldet sich daraufhin bei allen gefundenen Projekten als `ContentDifferenceListener` an.

- Zu jedem Projekt wird die Menge aller Klassen durchlaufen, die dazu gehört, und als Klassenknoten in eine Baumstruktur `JTree` eingebunden.

- Zu jeder Klasse wird die Menge aller Methoden dieser Klasse durchlaufen, und jede gefundene Methode mit ihrem Namen als Methodenknoten in obige Baumstruktur `JTree` eingebunden.

- Nun werden Instanzen der $\mathcal{HEU}$-Tools Klassendiagrammeditor, Klasseneditor und Methodeneditor erzeugt, wobei das Erzeugen des Klassendiagrammeditors mangels Implementierung ohne sichtbare Auswirkung bleibt. Die Werkzeuge werden in die Liste der verfügbaren Werkzeuge des Projektmanagers eingetragen und ihre textuelle Beschreibung in das dafür vorgesehene Beschreibungsfenster der Oberfläche kopiert.

- Schließlich wird das Fenster des Projektmanagers erzeugt und der bestehende `JTree` als Modell für den auf der Oberfläche des Projektmanagers angezeigten Baum eingesetzt, womit die tatsächlich vorhanden *DoDL*-Objekte für den Benutzer sichtbar werden.

25.5.2.2. Benutzung des Systems am Beispiel des Anlegens einer neuen Methode

Nachdem nun der Projektmanager gestartet ist, gemäß obiger Beschreibung die Verbindung zur Datenbank aufgebaut und entsprechende Laufzeit-Objekte zur Repräsentation der vorhandenen Daten erzeugt hat, kann mittels Selektion eines angezeigten *DoDL*-Objektes und Click des Buttons "Open" dieses im dazugehörigen Editor weiter bearbeitet werden. Die Abläufe der Kommunikation, die zu dieser Handlung notwendig sind, werden nachfolgend beschrieben. Als Beispiel dient, ausgehend von obiger Situation, das Öffnen eines Projekts per Doppelklick, das ebensolche Öffnen einer darin vorhandenen *DoDL*-Klasse und das Selektieren der Methode dieser Klasse. Der Druck auf den Button "Open" öffnet den Methodeneditor der $\mathcal{HEU}$.

- Die Methode `open` des Projektmagers wird aufgerufen.

- Es wird ein `ModelObjekt` erzeugt, das sich mittels `_register` als *Listener* für verschiedene Eigenschaften bei den Datenquellen des aktuellen Objekts in der OMS-Schicht anmeldet. Dabei werden auch die notwendigen Datenstrukturen geschaffen, damit sich weitere Objekte hier als *Listener* anmelden können.
- Das dazugehörige OMS-Objekt wird aus der Datenbank ausgelesen.
- Gemäß Typ des vorliegenden Objekts wird der zugehörige Editor gestartet, hier also der Methodeneditor. Dazu wird zunächst von der vorliegenden `RootSession` zur Datenbank eine eigene `ToolSession` abgespalten. Dies geschieht über die Methode `createToolSession` der `RootSession`. Damit wird ein neuer H-PCTE-Prozeß auf dem Server gestartet!
- Die neue `ToolSession` übernimmt mit der Methode `createOMSObjectCopyFromForeignSession` per Kopie das zugrundeliegende OMS-Projekt und die OMS-Methode.
- Jetzt wird die Instanz des neuen Methodeneditors erzeugt und davon die Methode `start` aufgerufen, der die neue `ToolSession` zur Benutzung übergeben wird.
- Der Methodeneditor erzeugt seine grafische Benutzungsoberfläche, die nun sichtbar wird. Dabei wird ein Objekt vom Typ `GenericVariableTableAdapter` erzeugt, jener Adapter, der benötigt wird, um die Modellobjekte der $\mathcal{HEU}$, in diesem Fall in Tabellenform, auf der Benutzungsoberfläche anzuzeigen. Der nähere Zusammenhang ist in Kapitel 29.4 auf Seite 306 dargestellt.
- Bespielhaft wird der besteheneden Methode ein neuer Parameter hinzugefügt. Dies führt zum Aufruf der Methode `addParameter` der GUI-Klasse `MethodEditorFrame`.
- Die aktuelle Implementierung erzeugt nun zunächst eine leere *DoDL*-Klasse `ModelDoDLClass` auf Modellschicht als umgebende Klasse. Bei dieser Aktion werden wiederum die notwendigen Methoden zum Anmelden als *Listener* und Erzeugen von Referenzen aufgerufen.
- Schließlich wird die Erzeugung des Methodenparameters mittels `insertParameter` von `ModelDoDLMethod` angestoßen, sie findet gemäß *MVC*-Konzept jedoch erst statt, wenn das Objekt in OMS erzeugt wurde und seine Entstehung dort bekannt gemacht wird.
- Dies führt zur Erzeugung des neuen Methodenparameters auf Datenbankebene durch `createRelation` von `JHPcteAggregationWorkingCopy`, da die Zugehörigkeit eines Parameters zu einer Methode als Relation modelliert wurde.
- Auf OMS-Ebene wird durch diese Objekterzeugung eine Notifikation `JHPcteLinkChangeNotifier.pcteNotify` ausgelöst.
- Diese führt zur Erzeugung eines entsprechenden `PropertyChangeEvent` durch die Methode `JHPcteLinkChangeOfTypeNotifier.fireChangeEvent` in der OMS-Kapsel.

- Das *Event* wird propagiert und es beginnt die *Event*-Verarbeitung auf Modellebene. Die `ModelDoDLMethod` reagiert in ihrer Eigenschaft als `EventListener` mit Ausführung der `propertyChange`-Methode. Dabei trägt das *Event* den alten Wert *null*, d. h. der Methodenparameter bestand vorher gar nicht, genauso mit sich, wie die exakte Datenbank-ID und Beschreibung der neuen Relation, nämlich daß es sich um den Parameter einer *DoDL*-Methode handelt.

- Nun entsteht der Methodenparameter auf Modellschichtebene mit entsprechender Erzeugung aller benötigten *Listener* und Referenzen.

- Der nachfolgende Ablauf ist folgendermaßen zu beschreiben: Die `ModelDoDLMethod` regiert ihrerseits mit der Abfeuerung eines *Events*, das das Vorhandensein eines neuen *DoDL*-Methodenparameters in Richtung Benutzungsoberfläche bekannt macht.

- Der oben erwähnte `TableAdapter` reagiert darauf mit der Neuanlage des Parameters zur Darstellung in dem Tabellenfenster des Methodeneditors. Damit würde das neu erzeugte Sprachelement auch dem Benutzer an seiner Schnittstelle sichtbar gemacht.

In der vorliegenden Implementierung ist dieser letzte Schritt jedoch noch nicht vollständig realisiert. Genau wie an vielen anderen Stellen der $\mathcal{HEU}$ bricht der Programmablauf ungewollt ab. Ursache ist in aller Regel eine fehlende oder fehlerhafte Implementierung, bedingt durch den Zeitmangel gegen Ende der Projektgruppe. Insbesondere das vollständige Reagieren auf und Abarbeiten von *Events* und *Exceptions* ist oftmals nicht geschehen.

Die obige Ablaufbeschreibung macht deutlich, wie vielschichtig und komplex die Kommunikationsmechanismen in der $\mathcal{HEU}$ sind und welchen grundsätzlichen Prinzipien sie folgen. Es läßt sich sicherlich sagen, daß das verwendete Konzept an sich tragfähig ist und eine weitere Vervollständigung der Implementierung zwar eine zeitaufwendige, jedoch lösbare Aufgabe wäre.

26. Verteilung und Versionierung

Autor: *Sebastian Linz*

Die folgenden Abschnitte befassen sich mit den Konzepten der Verteilung und Versionierung.

26.1. Ansprüche an die $\mathcal{HEU}$

Zunächst werden die Konzepte Verteilung und Versionierung erläutert, die in der $\mathcal{HEU}$ integriert werden sollen. Während der Entwicklungsphase ergaben sich Schwierigkeiten bei der Integration der Konzepte, die nachfolgend beschrieben werden.

26.1.1. Verteiltes Arbeiten

Wenn wir im Rahmen einer Entwicklungsumgebung von verteiltem Arbeiten sprechen, haben wir die Vorstellung, daß mehrere Entwickler gleichzeitig zusammen an einem Entwicklungsprojekt arbeiten können. Das bedeutet, daß die Entwicklungsumgebung verteiltes Arbeiten unterstützen muß.

Ein Szenario wäre, daß ein Entwickler an einem Klassendiagramm arbeitet und ein anderer eine Klasse dieses Diagramms modifiziert. Man kann sich nun leicht vorstellen, daß die Ansicht des Klassendiagramms aktualisiert werden muß, sobald der Entwickler beispielsweise den Namen der Klasse verändert.

An diesem Beispiel erkennt man, daß eine Entwicklungsumgebung, die verteiltes Arbeiten unterstützen soll, im wesentlichen zwei Dinge realisieren muß. Zum einen muß gewährleistet sein, daß mehrere Benutzer gleichzeitig ein Datum bearbeiten können. Desweiteren muß es einen geeigneten Mechanismus geben, der Datumsänderungen unterstützt und diese anschließend an alle Werkzeuge propagiert, die die geänderte Ressource benutzen.

26.1.2. Versionierung

Versionierung soll gewährleisten, daß Entwicklungsstände registriert werden. Dadurch ist es einerseits möglich, bei Fehlentwicklungen zu einem alten korrekten Stand zurückzugehen. Nachdem man zu einem älteren Stand zurückgegangen ist, muß es andererseits aber auch möglich sein, von diesem Stand aus einen neuen Entwicklungsweg einzuschlagen

und seine neuen Entwicklungsstände registrieren zu können. Versionszweige sollten jedoch auch ohne Fehlentwicklungen zur Verfügung stehen, um zum Beispiel parallel an Entwicklungsständen weiterarbeiten zu können.

26.1.3. Schwierigkeiten bei der Integration der Konzepte

Bei $\mathcal{HEU}$-Dokumenten handelt es sich nicht um Textdateien, sondern um Dokumente, deren Elemente feingranular modelliert sind. So werden die einzelnen Klassen eines Klassendiagramms und das Klassendiagramm selbst auf Objekte mit ihren spezifischen Ausprägungen abgebildet. Zwischen den Objekten besteht eine gewisse Hierarchie, wie man sie auch von einem Verzeichnisbaum her kennt, da es Objekte gibt, die andere beinhalten.

Möchte man verteiltes Arbeiten und Versionierung in einer Entwicklungsumgebung integrieren, so fordert die Objekthierarchie, daß man parallele Objekte im Rahmen der Versionierung anlegen können muß. Hierbei spricht man von parallelen Objekten, wenn man Versionsäste erzeugt und ein Objekt nebenläufig versioniert. Im Rahmen einer verteilten Arbeitsumgebung sollte es dann jedoch auch möglich sein, parallele Objekte vermischen zu können. Ansonsten hätte man nicht mehr verteilt und parallel an einem Objekte gearbeitet, sondern man hätte unterschiedliche Objekte erzeugt. Aber gerade das Vermischen von Objekten stellt sich als besonders problematisch heraus. Das folgende Beispiel soll die Komplexität veranschaulichen. Wenn zwei Entwickler ein Objekt bearbeiten und dieses Objekt andere Objekte enthalten kann, so sollte es auch möglich sein, enthaltene Objekte zu löschen. Daraus resultiert jedoch ein komplexes Problem für den Mischvorgang. Denn es ist nicht sofort eindeutig klar, welcher Zustand derjenige ist, der ein korrektes Objekt definiert. Es müssten Konsistenzbedingungen und Mischregeln definiert werden, sodaß der Mischvorgang korrekte Ergebnisse erzeugt.

Läßt man es nicht zu, parallele Objekte zu erzeugen, um dem Mischproblem zu entgehen, steht man vor einer anderen Problematik. Innerhalb der Objekthierarchie findet man stets ein Wurzelobjekt, das eine Objekthierarchie enthält. Wenn man keine parallelen Objekte erzeugen kann, so kann von einem Wurzelobjekt nur eine Arbeitskopie angelegt werden. Jede weitere Arbeitskopie wäre ein paralleles Objekt zur ersten Arbeitskopie. Dies bedeutet aber auch, daß nur ein Werkzeug oder nur ein Entwickler an diesem Objekt arbeiten kann. Im Rahmen einer verteilten Entwicklungsumgebung ist dies fatal. Wäre die Arbeitskopie ein Projekt, so würde dies unweigerlich nach sich ziehen, daß kein weiterer Entwickler im vollen Umfang an diesem Projekt arbeiten könnte. Bestehende Elemente könnten zwar bearbeitet werden, doch es könnten keine neuen Elemente eingefügt oder existierende Elemente entfernt werden.

26.1.3.1. Aggregationen

Auch die Manipulation existierender Objekte hat Konsequenzen für den Versionierungsvorgang. Wenn ein Objekt (im folgenden Element genannt) Teil eines anderen Objekts (im folgenden Container genannt) ist, so stellt sich die Frage, was mit dem Container geschieht, wenn es eine neue Version des Elements gibt, wenn man parallele Objekte

nicht zuläßt. Wenn man Container grundsätzlich automatisch aktualisiert, kann es zu Situationen kommen, die zu einem Zyklus führen. Eine solche Situation tritt zum Beispiel dann ein, wenn ein Container selbst Element eines anderen Elements ist. Dieser Fall tritt schon auf, wenn eine Methode Element einer Klasse ist und die Klasse Parameter der Methodensignatur ist. Hier kann kein offensichtliches Stopkriterium definiert werden, das den Prozess der automatischen Aktualisierung unterbrechen könnte. Mit einer neuen Methode gibt es eine neue Klasse und mit ihr wiederum eine neue Methode.

26.1.3.2. Relationen

Durch Relationen soll die Beziehung von Objekten zueinander modelliert werden können. Im konkreten Fall beschreibt eine Relation immer genau die Beziehung von zwei versionierten Objekten zueinander. Dies könnte man auch so ausdrücken:

Objekt A_i in Relation mit B_i

Dieser Ausdruck beschreibt, daß die i-te Version von Objekt A in Relation mit der i-ten Version von Objekt B steht.

Hierbei ist die Relation für das Objekt B irrelevant. Zum Beispiel könnte hier beschrieben werden, daß eine Klasse A von einer Klasse B erbt. In diesem Beispiel ist die Relation für das Objekt B nicht von Bedeutung. Trotzdem könnte es im konkreten Fall wichtig sein, daß Objekt A genau von dieser Version von Objekt B erbt. Der Nachteil dieser Art der Modellierung der Relation besteht darin, daß man keine allgemeine Regel definieren kann, die beschreiben würde, was mit der Relation geschehen soll, wenn man eine neue Version von Objekt B erzeugt. Es ist unklar, ob Objekt A dann weiterhin in Beziehung mit Objekt B steht.

Eine andere Möglichkeit, die Relation zu modellieren, könnte so beschrieben werden:

Objekt A in Relation mit Objekt B

Diese Beschreibung ist unproblematisch für den Versionierungsvorgang. In diesem Fall geht man von einer statischen Relation zwischen den beiden Objekten aus. Wenn unter diesen Umständen neue Objekte erzeugt werden, stehen die stets in Relation zueinander. Diese Art, eine Relation zwischen zwei Objekten zu modellieren, ist sicherlich ungenügend. Wenn die Relation zwischen den beiden Objekten im Laufe der Entwicklung entstanden ist, kann nicht mehr nachvollzogen werden, wann die Relation eingefügt worden ist. Desweiteren ist es nicht mehr möglich eine ältere Version von einem der beiden Objekte zu erhalten, die nicht in Relation zu dem anderen Objekt steht. Relationen sind für den restlichen Entwicklungsprozeß dargestellt oder entfernt.

Folglich muß ein Konzept erarbeitet werden, daß die wichtigsten Versionierungsinformationen speichert.

26.2. Das Versionierungskonzept der $\mathcal{HEU}$

Das Versionierungskonzept der $\mathcal{HEU}$ ist eine Kompromisslösung. Einerseits soll es die Anforderungen einer verteilten Entwicklungsumgebung, wie Verteilung und Versionierung integrieren. Andererseits ist nur ein Umfang an Funktionalität berücksichtigt, der im Rahmen der Projektgruppenarbeit bewältigt werden konnte.

26.2.1. Parallele und lineare Objekte

Wie oben beschrieben, müssen parallele Objekte wieder vermischt werden können, wenn man nicht aneinander vorbei, sondern zusammen entwickeln möchte. Da sich der Vorgang des Vermischens von Objektversionen als sehr komplex herausgestellt hat, läßt das $\mathcal{HEU}$-Versionierungskonzept keine parallelen Objekte zu. Dies wird dadurch verhindert, daß von Objekten, die man bearbeiten möchte, eine Arbeitskopie (Workingcopy) erzeugt werden muß. Sobald eine Arbeitskopie erzeugt wurde, kann von anderen Benutzern keine Arbeitskopie von dem entsprechenden Objekt erzeugt werden. Aufgrund der Tatsache, daß der Entwickler, der als erster eine Arbeitskopie von einem Objekt erzeugt hat, das Objekt für sich reserviert hat, sprechen wir auch vom *Locking*-Mechanismus. Sobald die Arbeitskopie versioniert wurde und nun aktuelle Version ist, können andere Benutzer das Objekt bearbeiten, sofern sie eine Arbeitskopie erzeugt haben.

Desweiteren läßt es das $\mathcal{HEU}$-Versionierungskonzept nur zu, daß von der letzten aktuellen Version eines Objekts eine Arbeitskopie erzeugt werden darf. Dies verursacht, daß das $\mathcal{HEU}$-Versionierungskonzept keine Versionsäste zuläßt.

26.2.2. Projekte

Da das $\mathcal{HEU}$-Versionierungskonzept keine Versionsäste zuläßt, ist der *Locking*-Mechanismus für Projekte nicht konsequent durchführbar. Wenn ein Benutzer ein Projekt als Arbeitskopie reserviert hätte, könnten andere Benutzer nicht mit ihm zusammen an dem Projekt arbeiten. Aus diesem Grund sind Projekte ständige Arbeitskopien, die verteilt bearbeitet werden können. Projekte können nicht versioniert werden. Aber man kann von Projekten *Releases* erzeugen, die dann den aktuellen Stand eines Projekts und der enthaltenen Objekte als Versionen konservieren. Alle anderen Objekte werden im Rahmen des Versionierungskonzepts gleich behandelt, wie im folgenden beschrieben wird.

26.2.3. Die Wurzel des Versionierungsstrangs - die Base-Version

Abgesehen von Projekten hat jedes Objekt eine Wurzel seines Versionierungsstrangs die *Versionswurzeln*.

Im Rahmen der Objekthierarchie werden die Versionswurzeln von den Projekten verwaltet. Wenn von einem Projekt ein Release erzeugt werden soll, so werden die jeweils aktuellen Versionen der Versionswurzeln als Projektstand konserviert. Bei allen anderen Objekten, die andere Objekte beinhalten können, wird stets die entsprechende Version eines Objekts verwaltet.

Die wesentliche Aufgabe der Versionswurzeln steckt in der Modellierung von Relationen. Das Versionierungskonzept stellt eine Relation als Eigenschaft einer Objektversion zu einer Versionswurzel dar.

26.2.4. Relationen

Relationen werden zwischen einem versionierten Objekt und der Versionswurzel eines anderen Objekts modelliert:

Objekt A_i in Relation mit Objekt B

Somit wird die Relation als Eigenschaft von Objekt A modelliert. Dies entspricht eigentlich auch der Verwendung von Relationen. Wenn die Objekte A und B Klassen wären und Klasse A von B erben würde, so wäre diese Information für Klasse B nicht relevant. Auf diese Weise lassen sich alle Relationstypen durch das Versionierungskonzept darstellen.

Dadurch, daß man die Relation und das Objekt B mit einem Zeitstempel versieht, ist sogar möglich, festzuhalten, zwischen welchen versionierten Objekten von A und B die Relation gültig ist.

26.2.5. Aggregationen

Objekte, die eine beliebige Anzahl anderer Objekte aggregieren, wie zum Beispiel Klassen werden nicht automatisch aktualisiert. Wenn man von ihnen eine Arbeitskopie erzeugen will, muß die Aggregation selbst, wie auch alle enthaltenen Elemente, als aktuelle Version enthalten sein.

26.2.6. Objekte löschen

Prinzipiell dürfen Objekte nicht einfach gelöscht werden, denn sie könnten Ziel einer Relation sein. Daraus folgt, daß Objekte nur als ungültig oder gelöscht markiert werden dürfen. Man könnte Objekte löschen, wenn die Versionswurzel nicht mehr Ziel von Relationen ist. Doch dies würde auch bedeuten, daß der gesamte Versionsstrang entfernt werden müsste. Da dies dem Entwicklungsprozeß eines Objekts nicht entspricht, den man ja eigentlich festhalten wollte, läßt es das Versionierungskonzept nur zu, Objekte als ungültig (*isDeleted*) zu markieren.

Teil VII.

Umsetzung der $\mathcal{HEU}$-Architektur

27. Objekt-Modellierung für die Datenbank (OMS)

Dieses Kapitel beinhaltet die Modellierung unserer Daten in Hinsicht auf die Datenbank. Wir beschreiben hier, warum wir Interfaces genommen haben, wie wir modelliert haben, welche Auswirkungen Transaktionen auf den Entwurf haben und auf welche Art wir ihn implementiert haben.

27.1. Interfaces

Autor: *Christian Stücke*

Wir benutzen H-PCTE (siehe [Kel98]) als Objektmanagementsystem, um die von uns verwendeten Daten und Objekte permanent zu speichern. Außerdem werden durch die Speicherung die Daten für die Zugriffe mehrerer Benutzer im verteilten System bereitgestellt.

Die OMS-Interfaces erfüllen dabei mehrere Aufgaben, die im folgenden beschrieben werden:

27.1.1. Bereitstellung einer Schnittstelle zur Datenbank

Wir können unsere speziellen Daten und Objekte der $\mathcal{HEU}$ nicht direkt auf einzelne Objekte in H-PCTE abbilden, da sie sich aus komplexen Strukturen zusammensetzen, Objekte in H-PCTE jedoch nur einfache Datenbank-Entitäten sind. Somit können auch die von H-PCTE zur Verfügung gestellten Operationen nicht direkt von der Modell-Schicht verwendet werden. Aus diesen Gründen benötigt die Modell-Schicht die OMS-Schicht, die dann die transienten Objekte explizit in persistente Objekte umsetzt, die dem Datenbankschema in H-PCTE entsprechen und umgekehrt. Objekte müssen mit all ihren Eigenschaften aus der Objektbank ausgelesen und nach der Benutzung in bearbeiteter Form wieder in die Objektbank geschrieben werden können.

Genau diese Funktionalität müssen die Interfaces der OMS-Schicht nach oben hin anbieten. In den Interfaces sind entsprechende Methoden definiert, die das Arbeiten mit den Objekten der $\mathcal{HEU}$ ermöglichen. Somit haben die OMS-Interfaces die Aufgabe eines Dienstanbieters gegenüber der Modell-Schicht.

27.1.2. Kapselung der Datenbank

Wie schon im letzten Abschnitt angesprochen arbeiten wir nicht direkt auf den Operationen von H-PCTE. Dadurch erreichen wir auch eine Kapselung der eigentlichen Datenbank. Ein dadurch entstehender Vorteil ist die Flexibilität bei der Wahl der zugrundeliegenden Daten- bzw. Objektbank.

Da die eigentlichen Datenbank-Zugriffe nur in den implementierenden Klassen erfolgen, könnten wir ohne größere Konzeptänderungen auf andere Datenbanken wie O2 oder Oracle wechseln. Natürlich müßten dann die bestehenden Interfaces der neuen Datenbank entsprechend neu implementiert werden, die einmal definierten Methoden in den OMS-Interfaces müßten dahingegen nicht mehr modifiziert werden.

Gründe für das Wechseln auf ein anderes Objektmanagementsystem könnten Performance- oder Stabilitätsprobleme etc. sein.

27.1.3. Read-only-Objekte und Workingcopies

Wir unterstützen in unserer $\mathcal{HEU}$ das Konzept der Versionierung. Aus diesem Grunde müssen wir in der OMS-Schicht der Modell-Schicht zwei verschiedene Arten aller in der $\mathcal{HEU}$ vorkommenden Objekte anbieten.

Zum einen gibt es Read-only-Objekte, die stabile, eingecheckte Versionen von Objekten darstellen. Diese Objekte dürfen natürlich weder verändert noch eingecheckt werden (sie sind ja bereits eingecheckt). Deshalb dürfen wir der Modell-Schicht in den Read-only-Interfaces nur lesende Operationen auf diese Objekte anbieten.

Auf der anderen Seite gibt es für alle Objekte in der $\mathcal{HEU}$ ein WorkingCopy-Interface. Dieses bietet genau die schreibenden Operationen, um Attribute bzw. Beziehungen zu erstellen, zu ändern oder zu löschen. Da das Workingcopy-Interface vom Read-only-Interface erbt, stehen natürlich auch bei den implementierenden Workingcopy-Objekten die lesenden Operationen zur Verfügung.

Die Implementierung eines Read-only-Objekts implementiert somit natürlich nur das Read-only-Interface, wohingegen das WorkingCopy-Objekt beide Interfaces implementiert.

Die Aufteilung in Read-only- und Workingcopy-Interfaces bringt aber auch einen Nachteil: Die Anzahl der Interfaces verdoppelt sich, was zur Verminderung der Übersichtlichkeit beiträgt. Als Alternative hätten wir uns auch auf nur ein Interface pro Objekt beschränken können. Die dort definierten Methoden für die Workingcopy-Objekte müßten dann eine spezielle Exception werfen, wenn ein Read-only-Objekt auf diese Methoden zugreifen will. Der Nachteil dieser Variante liegt darin, daß dies nicht streng dem Bean-Konzept entspricht, da in der Definition lediglich `PropertyVetoExceptions` erlaubt sind.

Außerdem handelt es sich bei Read-only-Objekten und Workingcopies um zwei wirklich verschiedene Objekte, und somit sind zwei getrennte Interfaces wohl eine vertretbare Realisierung.

27.1.4. Generische und *DoDL*-spezifische Interfaces

Alle *DoDL*-Objekte in unserer $\mathcal{HEU}$ haben viele gemeinsame Eigenschaften und damit verbunden viele gemeinsame Operationen. Zum Beispiel haben alle Objekte einen Namen und eine Versionsnummer und somit benötigen alle *DoDL*-Objekte entsprechende Methoden zum Setzen bzw. Auslesen der entsprechenden Attribute. Deshalb bietet es sich an, möglichst viele solch allgemeiner, für alle Objekte geltenden Methoden in generischen Super-Interfaces zu definieren, um diese nur einmal definieren - und somit auch nur einmal implementieren - zu müssen. Von diesen generischen Interfaces erben dann die speziellen *DoDL*-Interfaces, und es müssen dort schließlich nur die für dieses Objekt speziellen Methoden definiert werden. Wir erreichen dadurch eine große Wiederverwendung von Quellcode.

Auf die generischen bzw. *DoDL*-Interfaces wird in den nächsten Abschnitten noch genauer eingegangen.

27.2. Generische Objekte

Autoren: *Andreas Schröder, Sascha Lüdecke*

Die "generischen Objekte" sollen die generell behandelten Daten und Aspekte wie Datenbanksitzung, Administration sowie Ausnahmebehandlung abdecken. Wir werden in dieser Reihenfolge die OMS-Interfaces Schritt für Schritt einführen und die konkreten Aufgaben und die Funktionsweise näher erläutern. Beginnen wir mit der allgemeinen Struktur.

27.2.1. Allgemeine Struktur

Bei der Modellierung der OMS-Interfaces stand die Persistenz unserer Daten im Vordergrund. Es ging dabei nicht um eine Semantik, sondern um die Struktur in unseren Daten und wie wir diese für eine vernünftige Speicherung ausnutzen können. Weitere Anforderungen waren die Fähigkeit zur Versionierung und die Möglichkeit, Informationseinheiten bzw. Entitäten durch Metatdaten genauer zu beschreiben, gleich welche Struktur sie besitzen. Metadaten sind dabei nicht fester Bestandteil jeder Entität, sondern sind eher spärlich vorhanden und unterschiedlicher Natur; nicht jede Art Metadatum hängt an jeder Entität. Ihre Aufgabe ist es, für verschiedene Tools zusätzliche Informationen abzulegen, wie zum Beispiel Positionsinformation oder Syntaxinformation für einen Compiler.

Eine erste Untersuchung der Struktur ergab zunächst eine Aufteilung in atomare und in zusammengesetzte Entitäten, die atomare Entitäten enthalten (in Anlehnung an [KLLMM93, Kapitel 10]). Eine nähere Betrachtung zeigte gewisse Gemeinsamkeiten zwischen diesen Strukturen, wie Versionierung und Dokumentation bzw. Metadaten zu unseren Daten, die gerade aus den Anforderungen hervorgingen. Die zusammengesetzten Entitäten konnten wir nocheinmal in reine Sammlungen und in Aggregationen aufteilen. Letztere stellen eine Erweiterung der Sammlungen dar, die Beziehungen zwischen ihren

Bestandteilen beinhaltet. Diese Beziehungen sind aus unserer Sicht atomar, unterliegen aber nicht der Versionierung und assoziieren zunächst keine Metadaten. Sie sind reine Verbindungselemente.

Aus diesen Überlegungen ergab sich die in Bild 27.1 dargestellte Struktur. Funktionalität zu Versionierung und Metadatenverwaltung stellt das Objekt zur Verfügung. Es zerfällt in atomare Objekt (`Simple`) und Mengen (`Set`). Aggregationen sind eine Erweiterung letzterer, die zusätzlich eine Menge von Relationen zwischen Objekten verwalten. Relationen habe aus dieser Ebende lediglich die Aufgabe, zwei Objekte zueinander in Beziehung zu setzen. Welche Bedeutung diese Beziehung hat (Ober-Unter-Klasse, Teilprojekt-Von, ...) geht später aus entsprechenden Spezialisierungen hervor.

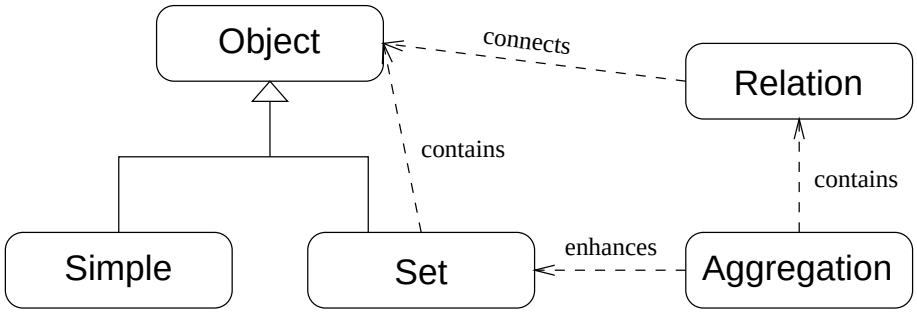


Abbildung 27.1.: Strukturierung der Information

Im weiteren haben wir die gefundene, noch sehr abtrakte, Struktur durch Unterklassen erweitert und an die Gegebenheiten der Datenbank bzw. die Aufgabenstellung, nämlich die Verwaltung von Klassen, Projekten, Diagrammen und mehr, angepasst. Dies umfaßt die vier in der Einleitung zum Abschnitt 27.2 auf der vorherigen Seite genannten Aspekte, die im folgenden behandelt werden.

27.2.2. Datenhaltung

In diesem Abschnitt soll erklärt werden, wie die Datenhaltung funktioniert.

Abbildung 27.2 auf Seite 244 verdeutlicht die Vererbungshierarchie. Diese Skizze soll einen Überblick darüber geben, wie genau die einzelnen Interfaces in Beziehung stehen.

`OMSObject` stellt dabei die oberste Klasse dar. Sie bietet Funtkionalitäten wie den `HistoryController` für die Versionierung und den `RightsDescriptor` für die Rechtevergabe. Desweitern gibt es für jedes `OMSObject` eine Dokumentation, die man sich über `getDocumentation()` holen kann, und es gibt für jedes `OMSObject` die Möglichkeit, über getSession(), herauszufinden, in welcher Session es erzeugt wurde.

`OMSSimple` erbt von `OMSObject` und stellt die einfachste Form der Datenhaltung da (d.h. es kann nicht viel mehr als `OMSObject`). Es dient eigentlich nur für die *DoDL* - Methoden.

`OMSSet` dagegen verfügt über etwas mehr Funktionalität. Es ermöglicht über das geerbet Interface Set (von Java) die Verwaltung von mehreren Objekten (hier

`OMSObjects`) sog. Kollektion. Ändert sich etwas am Inhalt der Kollektion wird ein `Property-Change-Event` ausgelöst. Dafür sind die `ContentDifferenceListener` zuständig. Kurze Zusammenfassung:

- `OMSObject`:
  Abstrakte Superklasse aller OMSKlassen, die grundliegende Konzept wie die Versionierung, Namensvergabe, Notifizierung ... unterstützt.
- `OMSSimple`:
  Stellt ein einfaches OMSObject dar, zur Handhabung der Methoden (*DoDL*-Methoden)
- `OMSSet`:
  Verwaltet eine Ansammlung von OMSObjects.

Nun kommen wir zu den Arbeitskopie. Diese sind aufgrund der Versionierung notwendig. Denn nur auf diesen kann gearbeitet werden. D.h. hier kann geändert werden und anschließend eingecheckt werden. Dies geht über den `WorkingCopyController` der *dismiss* und *checkin* unterstützt. `OMSSetWorkingCopy` und `OMSSimpleWorkingCopy` erben von `OMSWorkingCopy` und ermöglichen dementsprechend das Arbeiten mit `OMSSet` und `OMSSimple`. Kurze Zusammenfassung:

- `OMSObjectWorkingCopy`:
  Stellt das Interface zur WorkingCopy dar, die Superklasse von OMSSetWorkingCopy und OMSSimpleWorkingCopy.
- `OMSSetWorkingCopy`:
  Hier kann diese Ansammlung verändert werden.
- `OMSSimpleWorkingCopy`:
  Hier kann dieses Object dann verändert werden.

Nun sind wir also in der Lage, *DoDL*Methoden über `OMSSimple` und *DoDL*Klassen über `OMSSet` zu halten. Es fehlt jetzt nur noch die Möglichkeit, die Beziehungen zwischen den Klassen festzuhalten. Dazu gibt es `OMSAggregation`. Dieses Interface ermöglicht es Verbindungen zu anderen Objekten aufzunehmen. Diese Beziehungen zwischen Objekten können ebenfalls bearbeitet werden. Dazu dient die `OMSAggregationWorkingCopy`. `OMSAggregation` bietet nun die Möglichkeit, eine Menge von `OMSObjects` und zusätzlich Beziehungen zwischen diesen Objekte zu speichern. Kurze Zusammenfassung:

- `OMSRelation`:
  Interface, das eine Relation zugänglich macht. Diese Verbindungen werden behandelt wie Attribute des Objektes. D.h. sie werden beim Versionieren mitkopiert und beim Editieren gesperrt. Eine Relation kann nur vom ausgehenden Objekt erzeugt

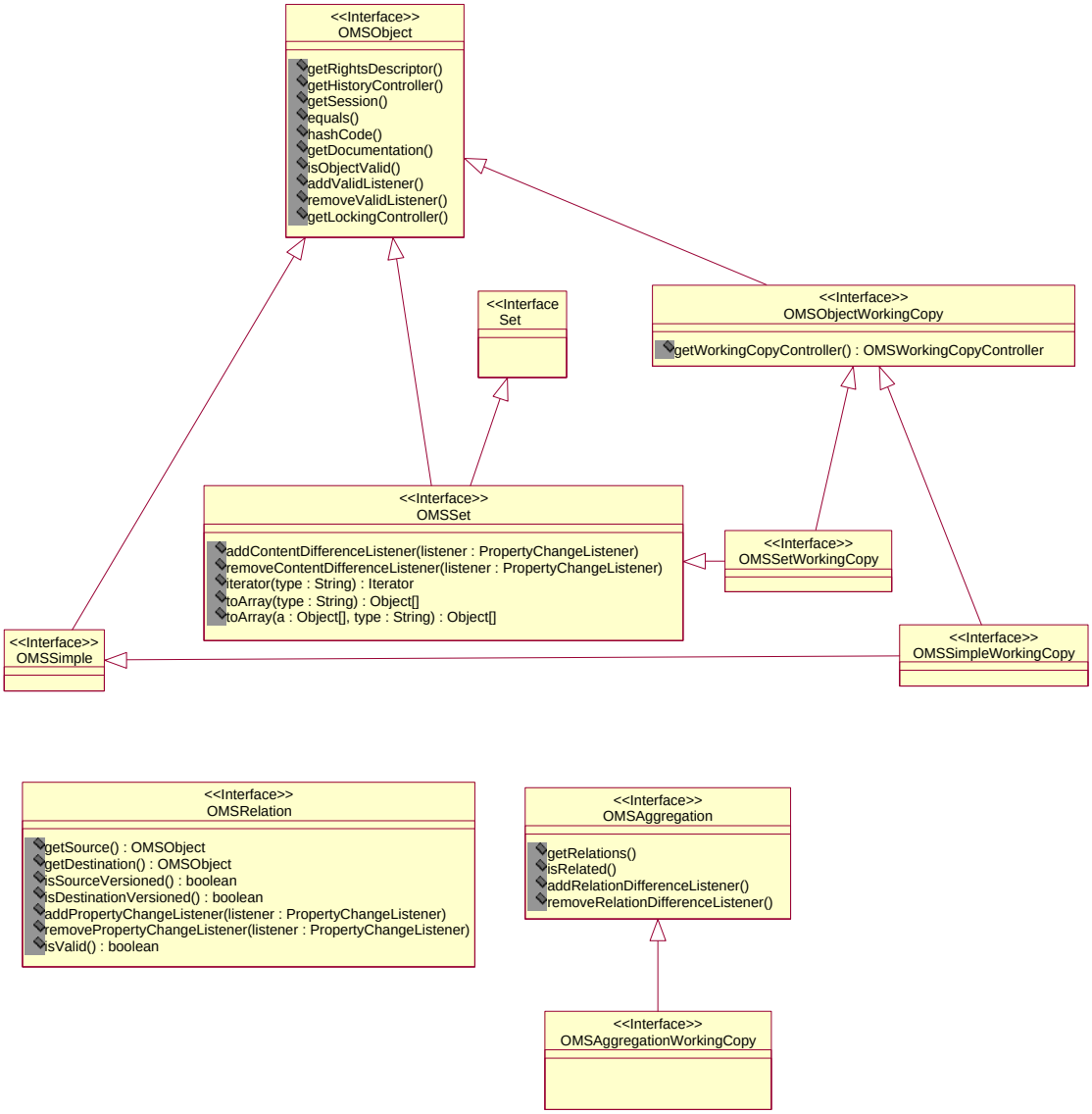


Abbildung 27.2.: OMSObjectInterfaces

werden. Das Zielobjekt kann eine stabile Version sein und muß demzufolge auch nicht ausgecheckt sein. Das bedeutet umgekehrt aber, daß sich die ankommenden Relationen einer stabilen Version ändern können.

- `OMSAggregation`:
  Interface, das ein `OMSObject` zusätzlich implementieren kann, damit es Verbindungen zu anderen Objekten aufnehmen kann. Diese Verbindungen werden behandelt wie Attribute des Objektes. D.h. sie werden beim Versionieren mitkopiert und beim Editieren gesperrt. Eine Relation kann nur vom ausgehenden Objekt erzeugt werden. Das Zielobjekt kann eine stabile Version sein und muß demzufolge auch nicht ausgecheckt sein. Das bedeutet umgekehrt aber, daß sich die ankommenden Relationen einer stabilen Version ändern können.

- `OMSAggregationWorkingCopy`:
  Stellt die WorkingCopy zu `OMSAggregation` dar, auf der gearbeitet werden kann.

27.2.3. Datenbanksitzung

Um auf der H-Pcte Datenbank zu arbeiten muß man sich zunächst einmal mit Benutzername und Passwort anmelden. Diese Anmeldung ordnet dann einem seine Rechte zu. D.h. man wird einer bestimmten Benutzergruppe, die bestimmte Rechte hat (Rechte-Konzept), zugeordnet.

Die Abbildung 27.3 auf der nächsten Seite soll die Vererbungshierarchie der verwendeten Interfaces zeigen. `OMSRootSession` ermöglicht nun eine solche Anmeldung. Desweiteren kann aus ihr heraus eine *ToolSession* gestartet werden (diese wird von allen Tools der $\mathcal{HEU}$ benötigt). Aus der *RootSession* heraus können dann auch alle vorhandenen Projekte geholt werden, als auch neue Projekte erzeugt werden.

In jeder Session gibt es nun die Möglichkeit, sich einen TransactionsController zu besorgen. Dieser übernimmt die Aufgabe, die Transaktionen zu verwalten. Er kann sie Starten, Savepoints setzen, redo und undo durchführen und ein Commit machen.
`OMSUser` und `OMSRole` sind für die Rechtevergabe zuständig. `OMSUser` stellt den Benutzer dar, der einen Namen, Arbeitskopien und gelockte-Dateien besitzen kann. `OMSRole` dient dazu, die verschiedenen Rollen zu verwalten. Kurze Zusammenfassung:

- `OMSSession`:
  Interface zur Steuerung von Session-Objekten. Ein Session-Objekt stellt im wesentlichen den Transkationsmechanismus zur Verfügung. Zu beachten ist, daß ein `OMSObject` immer im Kontext eines Prozesses erzeugt wird. D.h. die Transkationen einer Session beziehen sich auch nur auf OMSObjekte, die innerhalb dieser Session erzeugt worden sind.

- `OMSRootSession`:
  Basis-Session, die beim Start des Systems automatisch existiert. Über die Factorymethode `createToolSession()` werden die eigentlichen Prozesse für die Arbeit mit den Werkzeugen erzeugt.

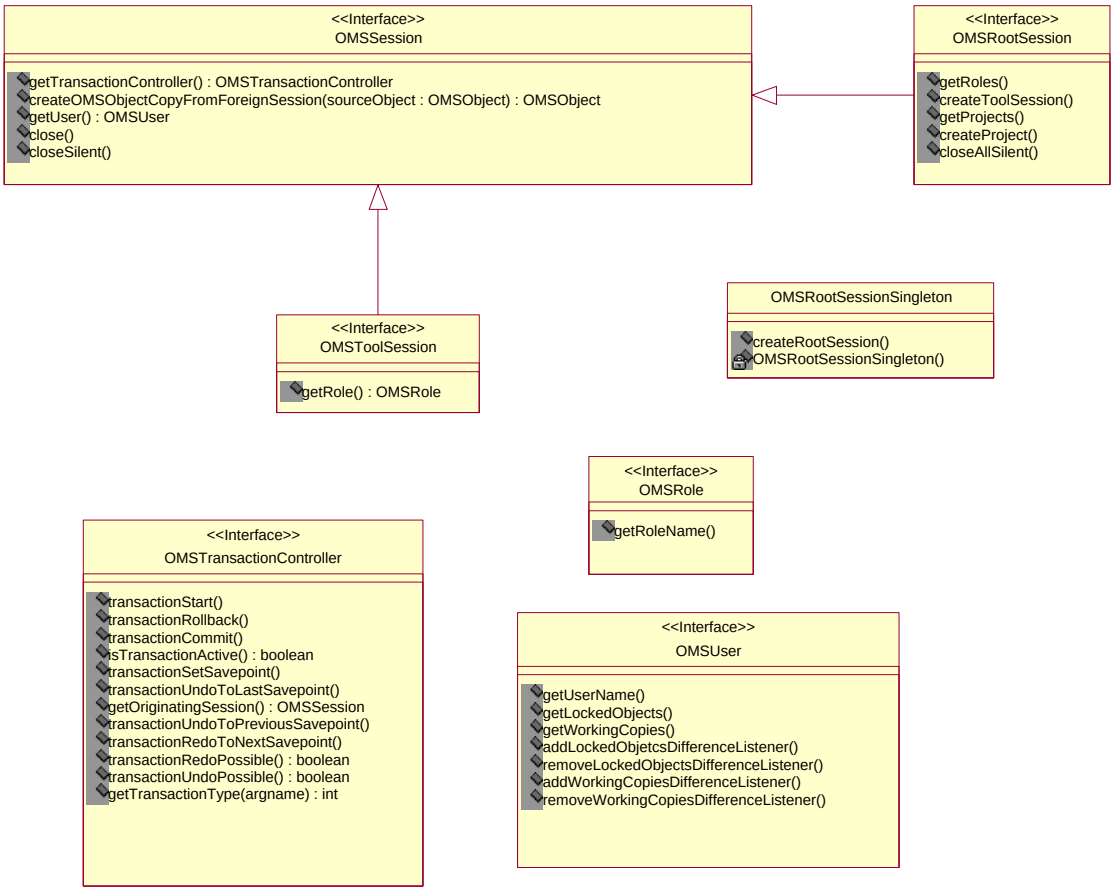


Abbildung 27.3.: OMSSession

- `OMSToolSession`:
  Interface für die eigentlichen Werkzeug-Sessions.
- `OMSTransactionController`:
  Ist für Start, Rollback, Commit, SetSavepoint, Redo, Undo usw. von Transaktionen zuständig.
- `OMSUser`:
  Stellt den Benutzer dar.
- `OMSRole`:
  Stellt die Rolle da, die ein Benutzer einnehmen kann.

27.2.4. Datenadministration

Im Abschnitt Datenhaltung wurde beschrieben, wie die Daten gehalten werden. In diesem Abschnitt wird nun darauf eingegangen, wie die Daten administriert werden.

Dazu dient der `OMSAspectDescriptor`. Dieser liefert über `getOrigin()` das `OMSObject`, zu dem es gehört. `OMSVersioningController` administriert die Versionierung und `OMSRightsDescriptor` ist dabei für die Rechte-Verwaltung zuständig.

- `OMSAspectDescriptor`:
  Abstraktes Interface, das einen Aspekt eines OMSObjects näher beschreibt. Subtypen sind `OMSVersioningController`, `OMSRightsDescriptor`.

27.2.5. Ausnahmebehandlung

Es gibt zwei Arten von Fehlern. Einmal der Benutzerfehler (siehe Abbildung 27.4 auf Seite 249 unterer Teil), d.h. ein unerlaubter Befehl wird wissentlich oder unwissentlich ausgeführt. Oder die Datenbank hat Verbindungsprobleme (siehe Abbildung 27.4 auf Seite 249 oberer Teil) oder ähnliches und es tritt ein Fehler auf.

Um nun einmal die Datenbankfehler abzufangen, dient das Interface `OMSErrorException`. Dies ist die Oberklasse dieser Fehlergruppe. Interfaces, die davon erben: `OMSInvalidObjectException`, `OMSDBLostException` und `OMSTransactionException`. Diese Interfaces befinden sich in der Abbildung 27.4 auf Seite 249 im oberen Teil.

Die einzelnen Kind-Interfaces sind nun spezielle Ausprägungen, die für bestimmte Fehlerbehandlugen zuständig sind. Kurze Zusammenfassung:

- `OMSErrorException`: Die Oberklasse dieser Art von Fehlern. Sie erbt von RuntimeException.
- `OMSInvalidObjectException`: Verantwortlich für die Behandlung von Fehlern die auftreten, wenn versucht wird, nicht mehr gültige Objekte zu bearbeiten.
- `OMSDBLostException`: Wenn die Datenbankverbindung abbricht.

- `OMSTransactionException`: Wenn Transaktionen fehlschlagen.

Die andere Gruppe von Fehlern beläuft sich auf VetoExceptions. Diese werden in der Abbildung 27.4 auf der nächsten Seite im unteren Teil dargestellt. Die Oberklasse hierbei ist `OMSVetoException`. Hier geht es darum, benutzerbedingte Fehler abzufangen. Diese können sich auf Rechteverletzungen (`OMSRightsException`), Versionierungsverletzungen (`OMSVersioningException`) oder Locking-Verletzungen (`OMSLockingException`) beziehen.
Auch hier eine kurze Zusammenfassung:

- `OMSVetoException`: Die Oberklasse dieser Art von Fehlern. Sie erbt von *PropertyVetoException*.
- `OMSRightsException`: Ist für die korrekte Rechtebehandlung zuständig.
- `OMSVersioningException`: Diese für die Versionierung.
- `OMSLockingException`: Und diese für die lock-Eigenschaft.

27.3. Metadaten

Autor: *Andreas Schröder*

Bei Metadaten handelt es sich um Daten für Daten. Damit ist gemeint, daß man für die Daten, mit denen man arbeiten will, nochmal Daten braucht, um diese zu verwalten. Als Beispiel könnte man eine (objektorientierte) Klasse betrachten, die Methoden und Variablen enthält. Würde man nun diese Klasse graphisch anordnen, so benötigt man dazu Koordinatenpaare. Dieses sind dann die Daten zu den Daten, mit denen man arbeiten will (sog. Metadaten).

27.3.1. Einleitung

Der Klassendiagrammeditor siehe (Abbildung 27.5 auf Seite 250) bietet die Möglichkeit, Klassen graphisch anzuordnen und zwischen Klassen Beziehungen darzustellen, die beliebig verlaufen können. Damit haben sowohl Klassen als auch Beziehungen Koordinatenpaare bzw. eine Liste von Koordinatenpaaren bei Beziehungen, da hier der graphische Beziehungsverlauf verändert werden kann.

27.3.2. Überlegung

Um nun die in der Einleitung vorgestellten Eigenschaften zu verwirklichen, muß man sich überlegen, wo man diese Daten (Koordinaten, etc. sogenannte Metadaten) am besten speichert.

Ein Klassendiagramm kann mehrere Klassen beinhalten. Diese Klassen haben nun in diesem Klassendiagramm eine bestimmte Position. Da aber eine solche Klasse auch

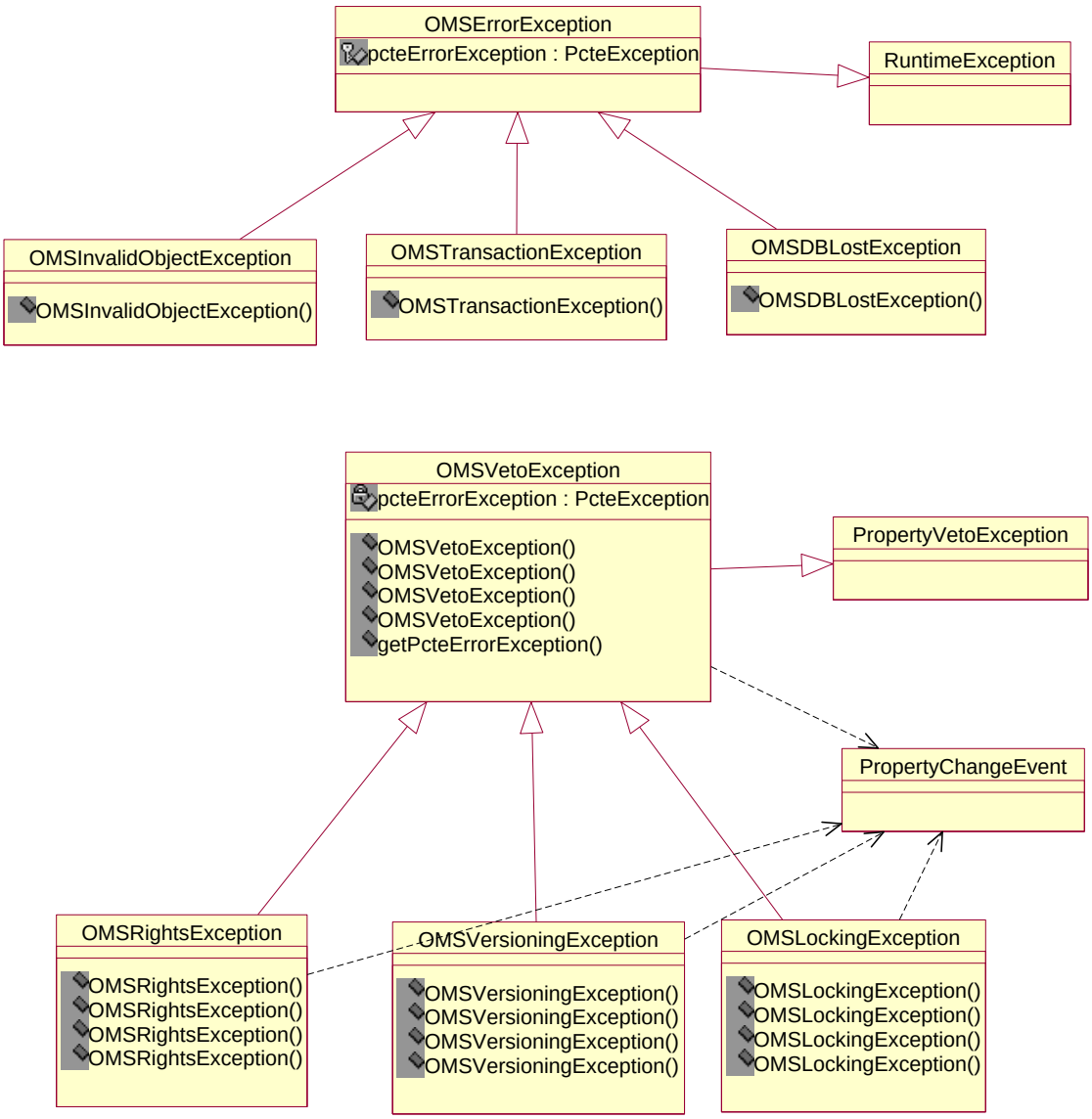


Abbildung 27.4.: OMSException

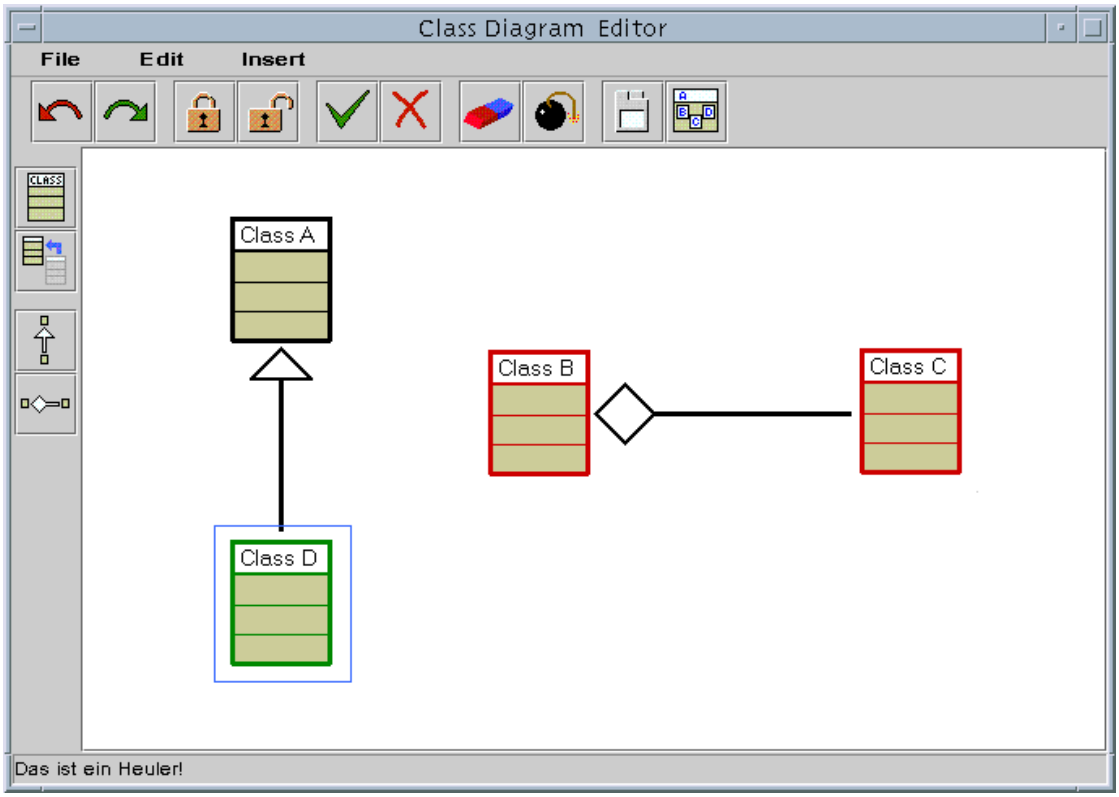


Abbildung 27.5.: ein Klassendiagramm

in einem anderen Klassendiagramm verwendet werden kann, kann sie dort folglich auch eine andere Position einnehmen. Damit kann man die Koordinaten einer Klasse nicht an der Klasse selber anheften.

27.3.3. Vorgehen

Man könnte nun zu jedem Klassendiagramm, das graphisch gesehen die Aufteilung beinhaltet, noch die Koordinaten der Klassen mit dem Klassenname als mögliche Referenz und den Koordinaten der Beziehungen ebenfalls mit entsprechender Namens-Referenz hinzufügen.

Damit wäre man dann in der Lage, in jedem Klassendiagramm mehrere Klassen mehrfach zu verwenden (an unterschiedlichen Positionen).

D.h.

- Klassendiagramm K1
  - Klasse C1 Koord (1,2)
  - Klasse C2 Koord (20,8)
- Klassendiagramm K2
  - Klasse C1 Koord (30,4)
  - Klasse C5 Koord (50,50)

Über den Namen hat man dann die Möglichkeit, an die Klassen zu gelangen, um spezifische Daten derselben zu erfragen.

27.3.4. Weitere Überlegungen

Desweiteren können Metadaten als Syntaxinformationen für mögliche Compiler dienen. Wobei auch hier Überlegungen angestellt werden müssen, wo genau diese Daten festgehalten werden müßten. So zum Beispiel (binding section), wenn bestimmte Dateinamen in den Klassen festgelegt werden sollen. Diese Festlegung könnte ja nun auch von Klassendiagramm zu Klassendiagramm wechseln, die dieselbe Klasse beinhalten.

27.4. *DoDL*-Objekte

Autor: *Christian Stücke*

Dieser Abschnitt soll die konkreten Objekte des *DoDL*-Syntaxgraphen und die Modellierung ihrer Schnittstellen erläutern. Die einzelnen *DoDL*-Interfaces erben viele Methoden von den generischen OMS-Interfaces, sodaß sie jeweils nur wenige "individuelle" Methoden besitzen. Dadurch bleiben diese Interfaces schlank und übersichtlich.

27.4.1. Methoden

Im Rahmen der Versionierung müssen wir uns irgendwann entscheiden, welches Objekt des *DoDL*-Syntaxgraphen die kleinste von uns zu versionierende Einheit darstellt. Dieses bis auf ein einzelnes *DoDL*-Attribut herunterzubrechen, erschien uns zu fein, andererseits erschien uns die *DoDL*-Klasse als zu grob, denn jede Änderung eines Attributs oder einer Methode zöge das Auschecken der kompletten Klasse mit sich. Während der Änderungen wäre die komplette Klasse für andere Benutzer zum Editieren gelockt und beim Einchecken würde eine neue Version der gesamten Klasse erzeugt, selbst wenn die Änderungen nur minimal waren oder nur eine Methode betrifft. Deshalb haben wir die *DoDL*-Methode als kleinste zu versionierende Einheit innerhalb der $\mathcal{HEU}$ gewählt.

Eine solche *DoDL*-Methode wird bei uns durch das Interface `OMSDoDLMethod` bzw. `OMSDoDLMethodWorkingCopy` repräsentiert. Da eine Methode keine kleinere Einheiten enthält, also keine Containerfunktion für andere Objekte besitzt, erben die oben genannten Interfaces nicht von `OMSSet` bzw. `OMSSetWorkingCopy` sondern von `OMSSimple` bzw. `OMSSimpleWorkingCopy`, da keine Mengenoperationen zur Verwaltung von kleineren Objekten benötigt werden. Die Vererbungshierarchie ist in Abbildung 27.6 auf der nächsten Seite auf der rechten Seite zu erkennen.

Man könnte meinen, eine Methode verwaltet verschiedene Parameter, dieses realisieren wir aber nicht über Mengenoperationen, sondern über Beziehungen. Die Handhabung von Parametern und Rückgabetyp wird daher in Abschnitt 27.4.4 auf Seite 254 und 27.4.5 auf Seite 257 näher erläutert.

Neben den Parametern und dem Rückgabetyp besteht eine *DoDL*-Methode noch aus einem Namen und natürlich aus dem eigentlichen Methoden-Body. Der Name wird über die entsprechenden get- und set-Methoden aus `OMSSimple` bzw. `OMSSimpleWorkingCopy` gelesen bzw. gesetzt. Der Methoden-Body wird über die Methoden `getBody()` bzw. `setBody()` gelesen bzw. geschrieben.

Für Änderungen an diesem Body kann sich ein View mit den Methoden `addBodyListener()` bzw. `removeBodyListener()` bei der Datenbank an- bzw. abmelden. Die Angemeldeten Views werden dann via Notifikationsmechanismus des MVC-Konzepts über die Änderungen benachrichtigt.

Als letzte individuelle Methode in `OMSDoDLMethod` gibt es noch die Methode `getContainingClasses()`. Wir können in der OMS-Schicht nicht verhindern, daß eine Methode in mehreren Klassen enthalten ist. Die oben genannte Methode liefert dem Aufrufenden daher alle *DoDL*-Klassen zurück, in denen die entsprechende Methode enthalten ist.

27.4.2. Klassen

Im Gegensatz zur *DoDL*-Methode ist die *DoDL*-Klasse ein Container. Sie muß nämlich Attribute und Methoden verwalten. Also erben die repräsentierenden Interfaces `OMSDoDLClass` und `OMSDoDLClassWorkingCopy` von `OMSSet` und `OMSSetWorkingCopy`. Diese Vererbungshierarchie ist in Abbildung 27.6 auf der nächsten Seite auf der linken Seite zu erkennen. Allerdings realisieren wir lediglich die Verwaltung der enhaltenen Metho-

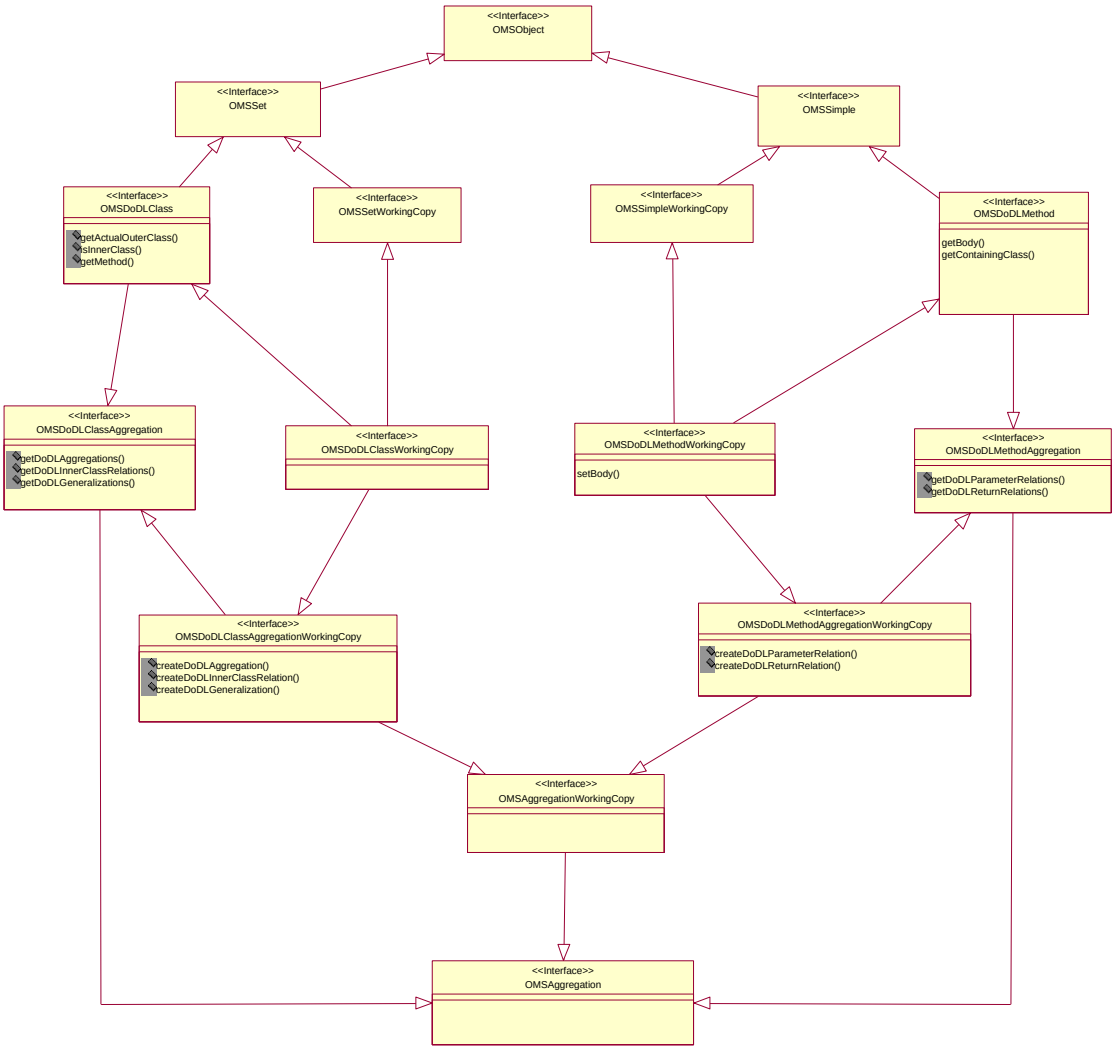


Abbildung 27.6.: `DoDLClass`, `DoDLMethod` und die Verwaltung der Beziehungen

den über die so geerbten Mengen-Methoden. Die Verwaltung der Attribute handhaben wir wie die Parameter und den Rückgabetyp von Methoden durch Beziehungen (siehe Abschnitte 27.4.4 und 27.4.5 auf Seite 257).

Um eine *DoDL*-Methode in einer bestimmten Klasse zu erzeugen, wird zunächst die entsprechende Instanz der Methode erzeugt und diese wird dann der Methode `add()` übergeben. Natürlich muß es sich bei der entsprechenden Klasse um eine Workingcopy handeln.

Änderungen an der Menge der enthaltenen Methoden bekommt ein View auf eine Klasse wieder über den Notifizierungsmechanismus mit. Dazu kann sich ein View mit den Methoden `addContentDifferenceListener()` bzw. `removeContentDifferenceListener()` aus `OMSSet` an- bzw. abmelden.

Eine Klasse kann innere Klasse einer anderen Klasse sein. Deshalb gibt es im Klassen-Interface der OMS-Schicht zwei weitere Methoden: Mit `isInnerClass()` läßt sich herausfinden, ob die aktuelle Klasse innere Klasse einer anderen ist. Ist dies der Fall, so kann man sich mit der Methode `getActualOuterClass()` die Referenz auf die umfaßende äußere Klasse geben lassen.

27.4.3. Systemtypen

Systemtypen sollen die von der $\mathcal{HEU}$ grundsätzlich zur Verfügung gestellten Datentypen für Attribute, Parameter und Rückgabewerte darstellen. Auf der OMS-Seite gibt es dafür das Interface `OMSDoDLSystemType`. Das Interface erbt vom Interface `OMSDoDLClass`, besitzt aber keine eigene Funktionalität. Momentan werden von uns die Systemtypen `Int`, `String`, `void`, `noname`, `Text`, `DBUnit` und `Graphic` unterstützt, die Anzahl der Systemtypen ist aber jederzeit erweiterbar.

Diese Typen werden durch eine spezielle Initialisierung in der Datenbank geschrieben und stehen der Modell-Schicht immer zur Verfügung. Es existiert natürlich vom Systemtypen-Interface keine WorkingCopy-Variante, da die Systemtypen nicht editierbar sind. Dadurch werden für Systemtypen auch keine Listener benötigt.

Die Modell-Schicht kann durch die Klasse `OMSDoDLSystemTypeBroker` auf die Systemtypen zugreifen: Die Methode `getDoDLSystemType()` liefert zu einem speziellen, übergebenen Typnamen eine Referenz auf ein entsprechendes Systemtyp-Objekt zurück. Entsprechend liefert `getOMSDoDLSystemTypes()` Referenzen auf alle unterstützen Systemtypen zurück. Der Grund, warum wir für Systemtypen eigene Objekte eingeführt und sie nicht über simple Strings realisiert haben, wird in Abschnitt 27.4.4 erläutert.

27.4.4. Beziehungen

Schaut man sich Klassen näher an, so stellt man fest, daß es eigentlich zwei verschiedene Arten von Attributen gibt. Zum einen gibt es Attribute, die als Datentyp einen einfachen Datentyp wie `Int` oder `String` besitzen, zum anderen können Attribute auch aggregierte Klassen sein. Der Datentyp eines solchen Attributs ist dann eine andere *DoDL*-Klasse. Ähnlich verhält es sich auch bei Methoden. Parameter und Rückgabewert einer Methode können entweder von einem einfachen Datentyp oder eine DoDL-Klasse sein.

Zur Handhabung dieses Problems bieten sich zwei Lösungen an: Die eine Möglichkeit ist die, für die beiden verschiedenen Arten von Datentypen auch verschiedene Methoden anzubieten. Im Falle der Attribute könnte das folgendermaßen aussehen: Die Methode `getAttributes()` liefert alle Attribute mit elementaren Datentypen und die Methode `getAggregations()` liefert alle Attribute, die durch Aggregation anderer Klassen enstanden sind, zurück. Da wir aber zwischen den verschiedenen Arten von Attributen und Datentypen nicht unterscheiden wollen, haben wir diese Variante wieder verworfen und uns stattdessen für folgendes Vorgehen entschieden: Wir fügen in der OMS-Schicht ein Beziehungs-Objekt `OMSRelation` bzw. `OMSRelationWorkingCopy` ein. Von diesen erben dann fünf verschiedene Subinterfaces, die die speziellen Beziehungen von Klassen und Methoden realisieren. Dies ist in Abbildung 27.7 auf der nächsten Seite zu sehen, die einzelnen Beziehungstypen werden in den folgenden beiden Abschnitten erläutert.

27.4.4.1. Beziehungen von Klassen

Wir ordnen Beziehungen zwischen *DoDL*-Klassen immer einer Ursprungsklasse zu. Deshalb können wir Attribute vom Typ `OMSDoDLClass` als eine Beziehung vom Typ "Aggregation" zu einer anderen Klasse sehen. Durch die Einführung der *DoDL*-Systemtypen (siehe Abschnitt 27.4.3 auf der vorherigen Seite), die ja Subobjekte von `OMSDODLClass` sind können wir auf diese Weise aber auch Attribute von elementaren Datentypen, den Systemtypen, so behandeln. Man erstellt einfach eine Beziehung zu einem Systemtypen. Neben der Aggregation werden auch Vererbungsbeziehungen und Beziehungen zu inneren Klassen über dieses "Relation-Modell" realisiert. Für eine Klasse stehen somit folgende konkrete Beziehungstypen zur Verfügung:

- Aggregation:

  Diese Beziehung wird durch die Interfaces `OMSDoDLAggregation` und `OMSDoDL-AggregationWorkingCopy` realisiert. Die Beziehung besteht immer von der aggregierenden zur aggregierten Klasse. In der Read-only-Variante hat man die Möglichkeit Namen und Kardinalität der Aggregation mit den Methoden `getName()` bzw. `getCardinality()` zu erfragen. In der Workingcopy-Variante können diese beiden Attribute natürlich mit den entsprechenden set-Methoden gesetzt werden.

- Vererbung:

  Die Vererbungsbeziehung wird bei uns durch `OMSDoDLGeneralization` und der entsprechenden WorkingCopy-Variante `OMSDoDLGeneralizationWorkingCopy` dargestellt. Die Beziehung wird hier immer von der Subklasse zur Superklasse aufgebaut. Es gibt keine Attribute, die gelesen werden könnten, deshalb ist die WorkingcopyVariante auch nur der Vollständigkeit halber realisiert.

- Beziehung zu inneren Klassen:

  Beziehungen zu inneren Klassen werden in der $\mathcal{HEU}$ über `OMSDoDLInnerClass-Relation` und `OMSDoDLInnerClassRelationWorkingCopy` realisiert. Die Beziehung ist dabei immer von der äußeren zur inneren Klasse gerichtet. Auch bei dieser Beziehung gibt es keine Attribute, die gelesen oder geschrieben werden könnten.

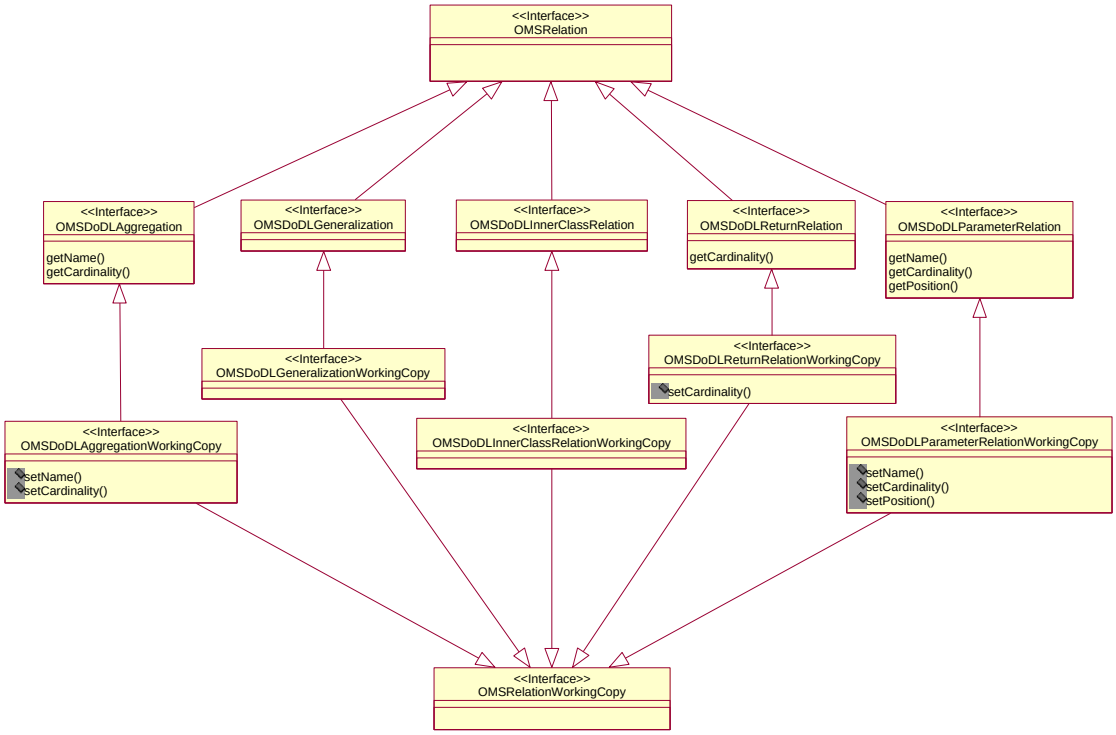


Abbildung 27.7.: Alle Relationstypen für Klassen und Methoden

27.4.4.2. Beziehungen von Methoden

Nach dem Relationen-Prinzip gibt es für Methoden nun genau zwei verschiedene Beziehungsarten zu *DoDL*-Klassen oder zumindest zu *DoDL*-Systemtypen:

- Parameter-Beziehung:

  Dieser Beziehungstyp wird durch `OMSDoDLParameterRelation` bzw. `OMSDoDLParameterRelationWorkingCopy` umgesetzt. Die Beziehungsrichtung ist hier natürlich von der Methode zur entsprechenden Klasse oder zum entsprechenden Systemtypen. Bei der Parameter-Beziehung kann neben dem Namen und der Kardinalität die Position über die entsprechende get-Methode gelesen werden, an der der Parameter innerhalb der Parameter-Liste der DoDL-Methode stehen soll. Im Workingcopy-Interface kann dieser Wert entsprechend gesetzt werden.

- Rückgabe-Typ:

  Der Rückgabe-Typ wird ebenfalls über eine Beziehung behandelt. Dafür stehen den oberen Schichten die Interfaces `OMSDoDLReturnRelation` und `OMSDoDLReturnRelationWorkingCopy` zur Verfügung. Als einziges Attribut kann beim Rückgabe-Typ die Kardinalität gelesen und geschrieben werden.

27.4.5. Verwaltung von Beziehungen

Es mag auffallen, daß es in den Interfaces für *DoDL*-Methoden und *DoDL*-Klassen keine Methoden für das Anlegen und Löschen der verschiedenen Beziehungen gibt. Um die Übersichtlichkeit zu wahren, haben wir für die Verwaltung der Beziehungen eigene Objekte eingeführt. Diese Objekte und die Vererbungshierarchie dieser Objekte sind in Abbildung 27.6 auf Seite 253 unten zu sehen.

27.4.5.1. Verwaltung von Klassenbeziehungen

Für die Verwaltung der Beziehungen zwischen *DoDL*-Klassen sind die Interfaces `OMSDoDLClassAggregation` bzw. `OMSDoDLClassAggregationWorkingCopy` zuständig, die nicht zu verwechseln sind mit einem der speziellen Beziehungs-Typen (`OMSDoDL-Aggregation`).

Im Interface `OMSDoDLClassAggregation` stehen die Methoden zur Verfügung, die dem Aufrufenden alle Beziehungen eines bestimmten Typs zurückliefern. Diese Methoden heißen `getDoDLAggregations()`, `getDoDLGeneralizations()` und `getDoDLInner-ClassRelations()`. Für jeden der Beziehungs-Typen kann sich ein View als Listener anmelden. Dazu stehen ihm die Methoden `addDoDLAggregationListener()`, `add-DoDLGeneralizationListener()` und `addDoDLInnerClassListener()` zur Verfügung. Bei Änderung an den entsprechenden Beziehungen werden die angemeldeten Views durch das entsprechende Property-Change-Event notifiziert. Natürlich können sich die Views durch entsprchende remove-Methoden als Listener auch wieder abmelden. Im Workingcopy-Interface stehen mit den Methoden `createDoDLAggregation()`, `create-DoDLGeneralization()` und `createDoDLInnerClassRelation()` die notwendigen Methoden zur Erzeugung der gewünschten Beziehungen zur Verfügung.

27.4.5.2. Verwaltung von Methodenbeziehungen

Die Verwaltung von Beziehungen, die von Methoden ausgehen, ist äquivalent zur Verwaltung von Klassenbeziehungen, die im letzten Abschnitt beschrieben wurde. Verantwortlich sind dafür die Interfaces `OMSDoDLMethodAggregation` und `OMS-`*`DoDL`*`Method-AggregationWorkingCopy`. Es stehen hier mit `addParameterListener()`, `remove-ParameterListener()`, `addReturnTypeListener()` und `removeReturnTypeListener` entsprechende Listener-Methoden zur Verfügung. Außerdem können mit `getParameter-Relations()` und `getDoDLReturnRelations()` die entsprechenden Beziehungen erfragt werden. Neue Beziehungen können im Workingcopy-Interface über `createParameter-Relation()` und `createReturnTypeRelation()` angelegt werden.

27.4.6. Klassendiagramme

In *DoDL*-Klassendiagrammen lassen sich die Beziehungen zwischen verschiedenen Klassen darstellen. Somit fungiert ein Klassendiagramm als ein Container für Klassen. Aus Zeitgründen haben wir es nicht geschafft, ein Interface für *DoDL*-Klassendiagramme zu

entwerfen. Wir wollen an dieser Stelle jedoch kurz aufzeigen, wie ein solches Interface aussehen müßte.

Zunächst einmal müßte das Interface von `OMSSet` erben, damit die Methoden für die Verwaltung der enthaltenen Klassen zur Verfügung stehen. Dadurch könnten sich auch Views auf ein Klassendiagramm mit der geerbten Methode `addContentDifference-Listener()` als Listener auf die enthaltene Klassenmenge an- und mit `removeContent-DifferenceListener()` wieder abmelden.

Klassen und deren Beziehungen können in mehreren Klassendiagrammen vorkommen und dort in verschiedenen Weisen, z. B. an verschiedenen Positionen oder in verschiedenen Farben, dargestellt werden. Deshalb benötigt ein Klassendiagramm-Interface in der Read-only-Version Methoden, um diese "Zusatzinformationen" zu lesen und in der Workingcopy-Variante Methoden, um diese auch zu schreiben. Man könnte sich also folgende Methoden denken: `getPosition()` ist eine Methode, der eine Instanz einer *DoDL*-Klasse übergeben wird und eine die entsprechende Bildschirmposition der Klasse zurückliefert. `getColor()` liefert zu einer *DoDL*-Klasse die entsprechende Darstellungsfarbe. Ebenso sind natürlich Methoden denkbar, die zu einem Beziehungspfeil die Menge der Stützpunkte liefert. Die set-Methoden wären in einer Workingcopy-Version entsprechend. Vermutlich wäre auch noch eine Methode wie `getProject()` notwendig, die das Projekt zurückliefert, zu dem das Klassendiagramm gehört.

27.5. System-Objekte

Autor: *Christian Stücke*

In diesem Abschnitt werden die Objekte beschrieben, die nicht zum *DoDL*-Syntaxgraphen gehören, die aber für die Verwaltung in der $\mathcal{HEU}$ wichtig sind.

27.5.1. Administration

Als Teil der $\mathcal{HEU}$ war ein "Admin-Tool" geplant, in der man Personen bestimmte Rollen zuordnen kann, die dann bestimmte Rechte an verschiedenen $\mathcal{HEU}$ -Objekten haben. Zum Beispiel könnte einer Person die Rolle "Tester" zugeordnet werden, in der die Person dann zwar alle Objekte lesen aber nicht bearbeiten darf. Im Gegensatz dazu gibt es vielleicht die Testberichte, die nur eine Person in der Tester-Rolle bearbeiten darf und die für andere Rollen nur lesbar oder gar nicht zugänglich sind. Ein Interface für diese Art der Administrierung ist aber aufgrund des Zeitmangels nicht spezifiziert worden. Ein solches Interface müßte folgendermaßen aussehen: Zum einen müßte es Methoden geben, die zu einer bestimmten Rolle die entsprechenden Rechte zuordnet. Dafür sind bereits die Interfaces `OMSRole` und `OMSRightsDescriptor` vorhanden. Dann müßte es Methoden geben, die Benutzer der $\mathcal{HEU}$ anlegen und auch wieder löschen. Im "Admin-Tool" kann man dann in einer bestimmten Rolle mit den entsprechenden Rechten ein Projekt öffnen. Das Projekt-Objekt soll im nächsten Abschnitt erläutert werden.

27.5.2. Projekte

Projekte sind in unserer $\mathcal{HEU}$ zunächst einmal eine Ansammlung von Klassendiagrammen, Klassen und Methoden. Es ist also nicht so, daß Klassen nur in Klassendiagrammen "hängen", denn Klassen können ja in mehreren Klassendiagrammen vorkommen. Außerdem versionieren wir Klassendiagramme, Klassen und Methoden. Dadurch sind in den jeweiligen Containern nur die aktuellsten Versionen der beinhalteten Objekte sichtbar. In den Projekten verwalten wir dagegen die sogenannten Base-Versionen der Objekte (siehe Abschnitt 26.2 auf Seite 234). Deshalb fungiert das Projekt als Container für alle *DoDL*-Objekte des Syntaxgraphen in der Base-Version, damit alle Objekte für alle anderen Objekte des selben Projekts zentral sichbar und zugreifbar sind. Aus diesem Grunde erben die entsprechenden Interfaces `OMSProject` und `OMSProjectWorkingCopy` von `OMSSet` bzw. `OMSSetWorkingCopy`. So können alle enthaltenen *DoDL*-Objekte mit den geerbten Methoden verwaltet werden.

Außerdem existieren in der Workingcopy-Variante mit `createDoDLClass()` und `createDoDLMethod()` Methoden zum Erzeugen von *DoDL*-Klassen und *DoDL*-Methoden. Falls das Klassendiagramm realisiert wäre, müßte natürlich noch eine Methode `createDoDLClassDiagram()` existieren.

Wir versionieren komplette Projekte nicht. Das liegt daran, daß wir in der OMS-Schicht kein Container-Objekt modelliert haben, das von `OMSSet` erbt und die Menge der Projekte verwaltet. Ohne solch ein Objekt greifen die Versionierungsmechanismen der $\mathcal{HEU}$ nicht. Deshalb sind alle Projekte in der $\mathcal{HEU}$ nur Workingcopies.

27.6. Implementierung der generischen OMS-Interfaces

Autor: *Sebastian Schütte*

27.6.1. Vorbemerkungen

Dieser Abschnitt dokumentiert, wie die in Abschnitt 27.1 auf Seite 239 beschriebenen Interfaces im Rahmen der $\mathcal{HEU}$ implementiert wurden. Im Folgenden werden diese Interfaces als bekannt vorrausgesetzt. Die wesentlichen Konzepte beim Entwurf der OMS-Schicht werden anhand zweier Grundprinzipen des Entwurfs erläutert. Dazu werden geeignete, meistens abstrakte Klassen vorgestellt, die jedoch den Löwenanteil der Funktionalität bereits anbieten. Sofern es für die Beschreibung der Arbeitsweise einer Klasse notwendig ist, werden Details des Datenbankschemas vorweggenommen. Eine vollständige Beschreibung des Datenbankschemas wird zum Ende dieses Abschnittes geliefert. Insbesondere wird beschrieben, wie das in Abschnitt 26 auf Seite 231 definierte Versionierungskonzept umgesetzt wurde.

Es wird vorrausgesetzt, daß der Leser mit den grundliegenden Konzepten von H-PCTE (Datenbankschema, `Pcte_Object_Reference`, `Pcte_Process`) sowie demJH-PCTE-API vertraut ist.

27.6.2. Terminologie

In diesem Abschnitt wird mit *OMS-Schicht* die Implementierung der OMS-Interfaces bezeichnet. Zur OMS-Schicht gehört nach dieser Definition das Datenbankschema der $\mathcal{HEU}$ und die implementierenden Klassen aller OMS-Interfaces (zu finden in den H-PCTE-Paketen der Paketstruktur der $\mathcal{HEU}$).

Die in Abschnitt 27.2 auf Seite 241 eingeführten Strukturprinzipien des Informationsraumes treten in diesem Abschnitt in verschiedenen Kontexten in Erscheinung.

1. Auf der Ebene der implementatiosnunabhängigen Struktur wird von *Informationseinheiten* oder *Entitäten* gesprochen. Dabei werden drei Strukturprinzipen genannt:
   - Die atomare Entität
   - Das Enthaltensein
   - Die Beziehung zu anderen Entitäten
2. Die OMS-Interfaces stellen eine Realiserung dieser Konzepte dar. Sie fallen mit den hier beschriebenen implementierenden Klassen zusammen. Im Kontext der implementierenden Klassen, bzw. deren Interfaces wird von `OMSSimple`, `OMSSet` und `OMSAggregation` (bzw. `OMSRelation`) gesprochen. Sie bilden eine Realisierung der oben genannten Strukturprinzipen und werden in diesem Abschnitt als Objekte bzw. als *OMSObjekte* bezeichnet. Synonym wird auch der Begriff *Fassadenklasse* verwendet, wenn die Struktur der OMS-Schicht im Vordergrund steht. Alle sogeannten Fassadenklassen sind Subklassen der generischen OMS-Objekte.
3. Der dritte Kontext, in dem die Strukturprinzipien auftreten ist der der Objekte im Datenbankschema. Die dort vorhandenen Objekttypen tragen ebenfalls Namen wie `OMSSimple`, `OMSSet` und `OMSRelation`. In diesem Zusammenhang ist von *Datenbankobjekten* die Rede.

27.6.3. Entwurfsüberlegungen

Die Struktur der OMS-Schicht läßt sich anhand zweier grundsätzlicher Überlegungen nachvollziehen:

1. Wie in Abschnitt 27.1 auf Seite 239 beschrieben, wurde für die Gestaltung der Schnittstelle der OMS-Schicht Mehrfacherbung eingesetzt. Mehrfacherbung ist in Java für Interfaces erlaubt, für Klassen jedoch verboten. Mit der implementierenden Klassenhierarchie der Hierarchie der Interfaces zu folgen war also nicht möglich.
2. Die generischen OMS-Interfaces definieren Funktionalität, die von allen Subklassen benötigt wird. Als Beispiel sei die Fähigkeit zur Verwaltung von Listener oder das Ein-/und Auschecken genannt. Ein naheliegender Lösungsansatz ist, die gemeinsame Funktionalität in einer gemeinsamen Oberklasse anzusiedeln. Es gibt jedoch auch Funktionalität, die an bestimmten Stellen benötigt wird und an anderer nicht. Als Beispiel sei hier die Verwaltung von Beziehungen zu anderen Objekten genannt.

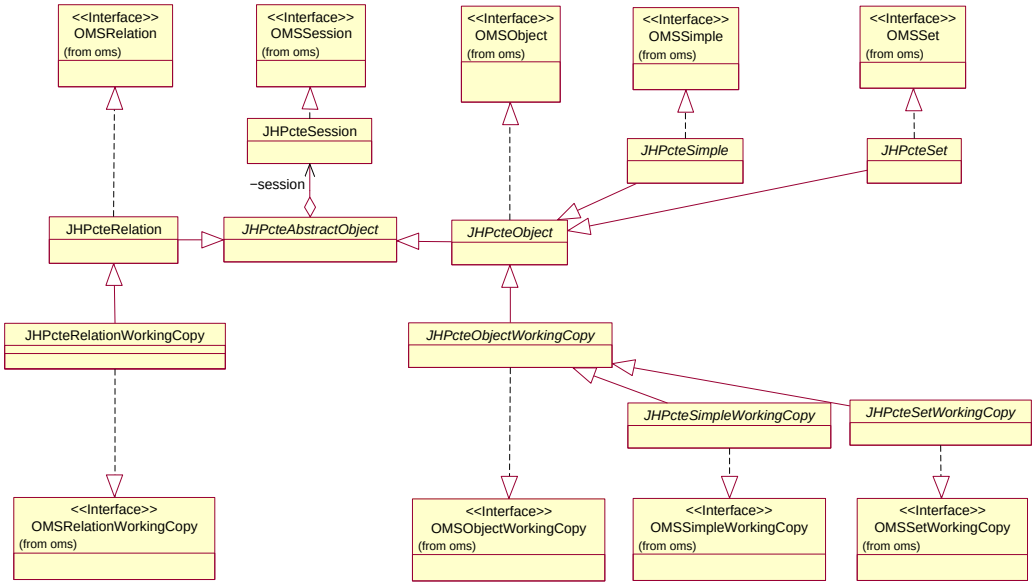


Abbildung 27.8.: Struktur der Fassade

Aus den oben genannten Überlegungen lassen sich zwei Richtlinien für die Implementierung ableiten:

1. Die Implementierung verwendet Fassadenklassen, die die in der Schnittstelle definierten Methoden nach außen anbieten.

2. Eine Sammlung von Klassen, im Folgenden Delegierte genannt, übernimmt die eigentlich Durchführung der Aufgaben.

27.6.4. Delegation der Funktionalität

In der Objektorientierung wird die Vererbung als Mittel beschrieben, um Funktionalität mehrfach zu verwenden. Man übersieht jedoch leicht, daß die Delegation ebenfalls ein Weg ist, Funktionalität einmal zur Verfüfung zu stellen und mehrfach zu verwenden.

Im Fall der OMS-Schicht bietet die Vererbung nicht die notwendige Flexibilität. Stattdessen wird an vielen Stellen Delegation eingesetzt, was ein flexibles Kombinieren von Klassen erlaubt, die gemeinsam die geforderten Aufgaben erfüllen.

Es wurden folgende Aufgabenfelder identifiziert, die jeweils durch einen Delegierten bzw. eine Familie von Delgiertenklassen bewältigt werden:

1. Verwaltung der Dokumentation

2. Verwaltung der Versionsinformation

3. Steuerung des Ein-/Auscheckens von Objekten

4. Verwaltung von Mengenbeziehungen
5. Verwaltung Relationsbeziehungen
6. Verwaltung von Listener und deren Benachrichtigung als Reaktion auf Notifikationen der Datenbank

27.6.4.1. Die Fassade - Große Schnittstelle, kleine Funktionalität

Für höhere Schichten ist es nicht von Belang, wie und von wem die durch die Schnittstelle definierten Dienstleistungen erbracht werden. Die definierte Schnittstelle wird durch eine Fassadenklasse implementiert (siege dazu [GHJV96]). Diese Fassadenklasse verwaltet die zur Bewältigung der Aufgaben nötigen Delgierten und kapselt deren Existenz nach oben.

Die von uns erzeugten Fassadenklassen verwenden für alle Delegierten das Konzept der späten Initialisierung. Die Fassadenklasse erzeugt einen Delegierten dann, wenn er gebraucht wird. Beim ersten Aufruf einer Methode, die sich auf einen spzifischen Delegierten abstützt, wird dieser erzeugt. Für alle weiteren Aufrufe wird der einmal erzeugte Delegierte weiterverwendet. Dieses Vorgehen spart Resourcen.

Späte Initialisierung - Ein Beispiel Der Leser möge sich als Beispiel vorstellen, daß die in einer *DoDL*-Klasse enthaltenen Methoden, bzw. deren Namen angezeigt werden sollen. Dazu wird bei der *DoDL*-Klasse die Liste der enthaltenen Methoden angefordert. Für jede *DoDL*-Methode muß eine geeignete Fassadenklasse erzeugt werden. Die einzige Aufgabe, die die Methodenobjekte in diesem Beispiel zu erfüllen haben, ist ihren Namen auszugeben. Ohne die Verwendung der späten Initialisierung wären jetzt für jedes Methodenobjekt alle Delegierten erzeugt worden, deren Funktionalität jedoch noch gar nicht benötigt wird.

27.6.4.2. JHPcteAbstractObject

Systematische Resourcenverwaltung `JHPcteAbstractObject` ist das Wurzelobjekt für alle Fassadenklassen. Sowohl `JHPcteRelation` als auch `JHPcteObject` sind Subklassen von `JHPcteAbstractObject`. So ist gewährleistet, daß alle Fassadenobjekte, die höheren Schichten zur Verfügung gestellt werden, durch den selben Erzeugungsprozess instantiiert werden können. Dazu mehr in Abschnitt 27.6.5.1 auf Seite 266.

Darüberhinaus kann in `JHPcteAbstractObject` die `finalize()` Methode derart implementiert werden, daß die verwendeten Resourcen wieder freigegeben werden. Zum einen wäre das die Datenbankreferenz, was jedoch im gegenwärtigen Stand der Implementierung noch nicht erfolgt. Zum anderen hält der oben genannte Erzeugungsprozess Daten, die ebenfalls gelöscht werden können, wenn das Objekt verschwindet. Die Freigabe dieser Resourcen ist implementiert (siehe auch Abschnitt 27.6.5.1 auf Seite 266).

Verwaltung Pcte-Objektreferenz und SessionDatenbanksitzung Ein `JHPcteAbstractObject` besitzt eine Referenz auf ein Datenbankobjekt und eine

Referenz auf eine `JHPcteSession`. Dies ist die SessionDatenbanksitzung, in der das Objekt erzeugt worden ist. Für manche Operationen muß die zugehörige Datenbanksitzung bekannt sein (beispielsweise für das Setzen eines H-PCTE-Notifiers), insofern muß diese Referenz bei jedem Objekt gehalten werden. Die OMS-Schicht folgt in diesem Fall dem HPCTE-API, denn auch dort ist es möglich zu jeder Objektreferenz den zugehörigen Prozess zu erhalten.

Häufig benutzte Funktionen `JHPcteAbstractObject` bietet Methoden zum Zugriff auf Attribute des verwalteten Datenbankobjektes. Diese Methoden dienen als Schreibvereinfachung für andere Programmstellen. Häufig verwendet werden Methoden, die die Versions-ID bzw. den exact-identifier der Datenbank herausgeben.

Verwaltung der Notifier Alle Fassadenklassen benutzen für das Erzeugen und Verwenden von Notifiern den in `JHPcteAbstractObject` implementierten Mechanismus, der im wesentliche aus drei Methoden und einer Datenstruktur besteht. Für jedes in der Schnittstelle der OMS-Schicht nach außen definierte Property existiert eine Konstante. Mittels dieser Konstante kann über `get-, set-, createNotifier()` auf den entsprechenden Notifier zugegriffen werden. Auch hier wird die späte Initialisierung verwendet (siehe auch Abschnitt 27.6.4.1 auf der vorherigen Seite). Wenn Subklassen von `JHPcteAbstractObject` weitere Properties besitzen, müssen entsprechende Konstanten eingeführt und die `createNotifer()` Methode erweitert werden.

Ausgangspunkt dieser Realisierung war folgende Anforderung: Zum einen sollten über genau eine Factory-Methode alle Notifier erzeugt werden, was die Verwendung von Typbezeichnern vorraussetzt. Zum anderen sollte es auch möglich sein, daß sich ein Objekt für alle vorhanden Properties als Listener anmelden kann. Dazu wäre ein Mechanismus wünschenswert, der alle möglichen Notifizierer erzeugen und den Listener anmelden kann. Dazu sollte jede Subklasse ihrer Properties beginnend mit der nächsten noch freien Nummer der Superklasse durchnumerieren und die Gesamtzahl aller Properties zur Verfügung stellen.

In der gegenwärtigen Implementierung sind alle Property-Konstanten in `JHPcteAbstractObject` definiert. Ebenso erzeugt die `createNotifier` Methode von `JHPcteAbstractObject` alle Notifizierer, auch die die nur in Subklassen verwendet werden. Dies geschah aus Gründen der Übersichtlichkeit während der Implementierung und ist im Zusammenhang des großen Zeitdrucks zu sehen. Eine solide Implementierung würde den dort implementierten Code auf die Subklassen verteilen, in denen das Property definiert wird.

27.6.4.3. JHPcteAbstractDelegate

`JHPcteAbstractDelegate` ist die gemeinsame Superklasse aller Delegierten. Einzige Methode ist `getOrigin()`, die den Zugriff auf das zugehörige Fassadenobjekt erlaubt. Alle Delegierten verwenden die Datenbankreferenz der Fassade.

27.6.5. Modellierung einer Datenbanksitzung

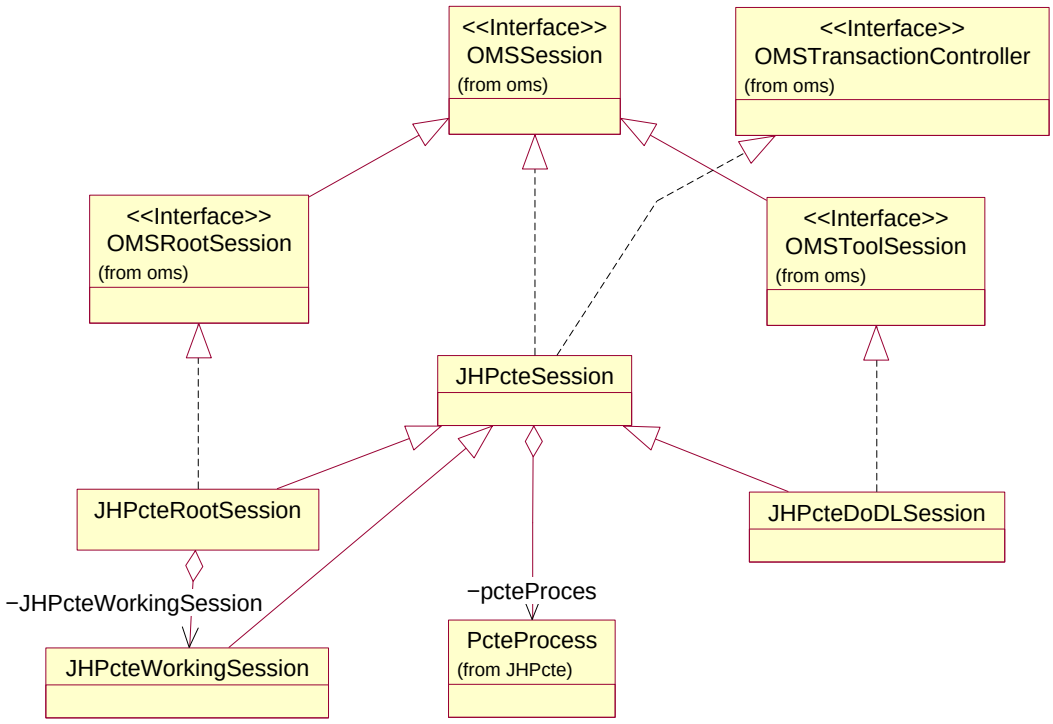


Abbildung 27.9.: Sessions

Die Fassadenobjekte der OMS-Schicht existieren nur innerhalb eines Prozesskontextes. Das ist der Prozess, in dem sie erzeugt wurden. Hier folgt die OMS-Schicht dem API von H-PCTE. Nach oben bietet die Datenbanksitzung `OMSSession` die Möglichkeit Transaktionen zu steuern und Objekte fremder Sessions in die eigene zu übernehmen. Intern übernimmt die Klasse `JHPcteSession` darüberhinaus noch die Aufgabe der Erzeugung der Fassadenobjekten aus einer Datenbankreferenz.

Wechsel des Prozesskontextes Das H-PCTE-API gibt keine generelle Möglichkeit vor, zu einem Objekt eines fremden Prozesses eine Referenz innerhalb des eigenen Prozesses zu erhalten. Wenn jedoch ein Werkzeug W_1, daß innerhalb einer eigenen Datenbanksitzung läuft, einem anderen Werkzeug W_2 ein Objekt O zur Bearbeitung übergeben möchte, muß aus der Objektreferenz für das Objekt O im Kontext von W_1 eine Referenz im Kontext W_2 erzeugt werden.

Es bietet sich folgendes Vorgehen an: Zu dem Objekt O wird ein absoluter Pfad bestimmt. Über diesen Pfad kann im Prozesskontext von W_2 eine neue Pcte-Referenz und damit ein neues Fassadenobjekt erzeugt werden.

Die in Abschnitt 27.6.9 auf Seite 278 beschriebene Struktur des Datenbankschemas erlaubt es, jederzeit einen absoluten Pfad zu bestimmen. Hilfreich ist dabei, daß jedes OMS-Objekt Element eines Projektes ist, unabhängig davon, worum es sich handelt. Im wesentlichen wird ausgehend von der aktuellen Version des zu übergebenden Objektes zum enthaltenden Projekt und von dort aus zum Wurzelobjekt der Datenbank navigiert. Die Umkehrlinks der dabei benutzten Links in umgekehrter Reihenfolge konkateniert ergeben einen Pfad zum Objekt O.

Transaktion Die Klasse `JHPcteSession` implementiert das `TransactionController` Interface und stützt sich dabei auf die im HPCTE-API angebotenen Methoden ab. Es wird lediglich eine Typvariable verwaltet, die Aufschluß über den gestarteten Transaktionstyp gibt, bzw. erlaubt herauszufinden, ob eine Transaktion aktiv ist oder nicht. Eine alternative Möglichkeit diese Information zu erhalten wäre die Verwendung der H-PCTE-Metadatenbank (siehe auch Abschnitt 27.6.10 auf Seite 282).

Im Rahmen des Entwurfes wurde überlegt, ob die änderenden Methoden der OMS-Schicht nicht automatisch ihre Arbeit durch eine Transaktion umschließen sollten und somit die Atomarität der Operation gewährleisten. Da HPCTE keine geschachtelten Transaktionen unterstützt, ist dies jedoch nicht möglich.

Geschachtelte Transaktionen ließen sich durch geschicktes Setzen von Sicherungspunkten simulieren. Dazu müßte ein API oberhalb der Transaktionsoperationen von H-PCTE erstellt werden, was das Setzen und Ansteuren von Sicherungspunkten auf mehreren Ebenen ermöglicht. Soll innerhalb einer Operation, die auf Ebene n läuft, eine geschachtelte Transaktion gestartet werden, werden alle folgenden Operationen auf Ebene $n+1$ durchgeführt, bis die geschachtelte Transaktion beendet ist. Dieses API müßte sicherstellen, daß wenn auf einer Ebene n zu Sicherungspunkten hin bzw. zurückgeblättert werden soll, Sicherungspunkte tieferer Ebenen automatisch mitgeblättert werden. Im Rahmen der $\mathcal{HEU}$ wurde bewußt darauf verzichtet, ein solches API zu erstellen. Vielmehr wurde die Verantwortung für Steuerung von Transaktionen/Sicherungspunkte vollständig in die Modellschicht verlagert. Der Preis hierfür ist die Tatsache, daß die Atomarität der Operationen der OMS-Schicht nicht gewährleistet ist.

Subklassen Die Subklassen von `JHPcteSession` fügen nur wenig Funktionalität hinzu. `JHPcteRootSession` ist in der Lage weitere Sessions zu erzeugen. Hier war ursprünglich geplant, nur leichtgewichtige Pcte-Prozesse zu erzeugen. Im Laufe der Implementierung stellten sich jedoch Probleme heraus, so daß nach Rücksprache mit Siegen jeder neu zu erzeugende Datenbanksitzung einen schwergewichtigen Pcte-Prozess erzeugt.

Wenn man ein sauberers Beenden aller im Rahmen der $\mathcal{HEU}$ erzeugten Datenbanksitzungem implementieren möchte, müßte bei `JHPcteRootSession` die Menge aller erzeugten Sitzungen verwaltet werden, um nach einem entsprechenden Methodenaufruf alle Sitzungen zu beenden. Im Interface `OMSRootSession` ist eine derartige Methode vorhanden, `JHPcteRootSession` implementiert diese jedoch nur leer.

`JHPcteDoDLSession` setzt das Datenbankschema des verwendeten Pcte-Prozesses so, daß die *DoDL*-spezifischen Datenbankobjekte sichtbar sind.

27.6.5.1. Objekt-Factory

Erzeugung der Fassade zu einer Pcte-Referenz An vielen Stellen der OMS-Schicht muß folgende Aufgabe bewältigt werden: Aufrufe an das H-PCTE-API geben ein oder mehrere Pcte-Referenzen zurück. Für höhere Schichten müssen diese Referenzen jedoch geeignet verkapselt werden. Man benötigt einen Mechanismus, der zu einer Pcte-Referenz die Fassadenklasse des richtigen Typs erzeugt.

`JHPcteSession` dient als Factory zur Bewältigung dieser Aufgabe. Es gibt mehrere Factory-Methoden, die sich im Typ der Rückgabe unterscheiden, sich jedoch auf den selben Erzeugungsmechanismus abstützen. Diese Schnittstelle ist im Laufe der Implementierung gewachsen und nicht von Anfang an sauber entworfen worden.

Der Mechanismus Aus Gründen der Übersichtlichkeit und der einfachen Erweiterbarkeit wurde der Erzeugungsmechanismus auf mehrere Methoden aufgeteilt. Die Methode `doCreateObjectFromPcteReference()` bestimmt alle notwendigen Informationen, um die richtige Fassadenklasse auswählen zu können. Die Methode `createObject()` erledigt die Instantiierung der Fassadenklasse aus ihrem Namen. Dies geschieht über den Mechanismus der `Class`-Objekte von Java. Soll die Menge der Fassadenklassen erweitert werden, muß lediglich die Methode `getClassNameForDBEntityTypeName()` ergänzt werden, die die eigentliche Zuordnung von Datenbanktypen und Fassadenklassen übernimmt.

Caching und Wiederverwendung der Objekte Beim Entwurf der OMS-Interfaces wurde Wert darauf gelegt, daß die Objekte der Fassade als Modell für mehrer Beobachter dienen können. Folglich spricht konzeptionell nichts dagegen, das Fassadenobjekt zu einer Informationseinheit genau einmal zu erzeugen und bei nachfolgenden Anforderungen dieser Informationseinheit wiederzuverwenden.

Für die Instanzen aller Fassadenobjekte einer Sitzung gilt, daß sie als Singleton behandelt werden (siehe dazu [GHJV96]). Hinter der Factory-Schnittstelle von `JHPcteSession` wird versteckt, daß jede erzeugte Instanz bei der Datenbanksitzung registriert wird. Erfolgt nach der Registration die Anforderung eine Fassadenklasse zu erzeugen, die bereits existiert, wird keine neue Instanz angelegt, sondern die bereits existierende herausgegeben. Jede Instanz wird durch einen Schlüssel, der sich aus dem exact-identifier der Datenbank und einem Präfix zusammensetzt, das für stabile Versionen und Arbeitskopien unterschiedlich ist, eindeutig identifiziert. Durch die Verwendung von `WeakReferences` wird ermöglicht, daß die Garbage-Collection Objekte abräumt, die zwar nicht mehr verwendet werden, aber noch registriert sind.

Ordentliches Beenden einer Sitzung Wird eine Sitzung ordentlich beendet, sollten alle noch vorhandenen Objekte dieser Sitzung benachrichtigt werden, daß sie nun ungültig geworden sind. Da jedes erzeugte Fassadenobjekt registriert wurde, ist das ohne weiteres möglich.

27.6.6. Die Delegierten

Im Folgenden werden die Klassen vorgestellt, die die in Abschnitt 27.6.4 auf Seite 261 identifizierten Aufgaben realisieren.

27.6.6.1. Dokumentation

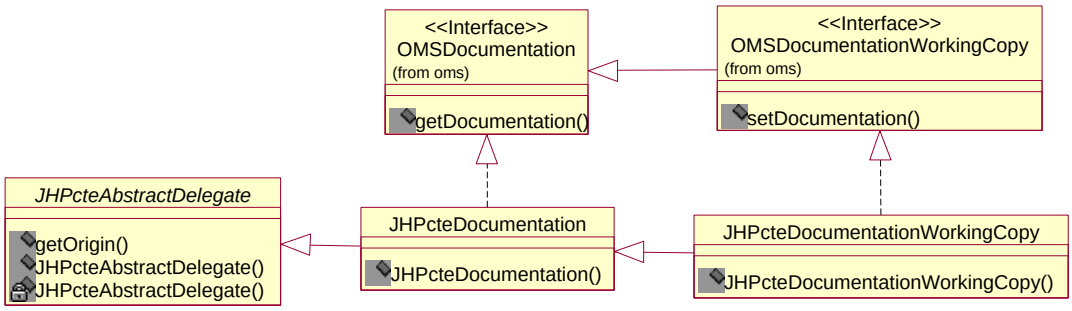


Abbildung 27.10.: Dokumentationsdelegierte

Für jedes OMS-Objekt existiert ein Dokumentationsobjekt. Stand der Implementierung ist, daß der Dokumentationsdelegierte einen String verwaltet. Mögliche Ergänzungen wären Informationen über den Author und verschiedene Zeitstempel.

Ein mächtigeres Dokumentationskonzept würde eigene Subklassen der generischen OMS-Interfaces einführen, so daß beliebig komplexe Dokumentationsstrukturen abgebildet werden können. Über Relationen oder Enthalten-Beziehungen ließe sich die Dokumentation mit den zu dokumentierenden Objekten verbinden. Dieser Weg wurde jedoch aus Zeitgründen nicht eingeschlagen.

27.6.6.2. Versionierung

Die Versionsdelegierten bietet Methoden an, um daß in Abschnitt 26 auf Seite 231 Versionierungskonzept zu verwenden. Dabei wurde bei der Planung sorgfältig darauf geachtet, daß der dort implementierte Mechanismus alle Arten von OMS-Objekten verwalten kann. Werden neue Subklassen der OMS-Objekte eingeführt, sollte es nicht notwendig sein, an den Versionsdelegierten Änderungen vorzunehemen.

Die Operationen zum Ein- und Auschecken der Speicherobjekte sind die kritischsten Operationen, die auf der Datenbank vorgenommen werden, da hier u.U. große Datenmengen verändert werdern. Diese Operationen sollten auf jeden Fall innerhalb einer Transaktion durchgeführt werden. Da H-PCTE keine geschachtelten Transaktionen vorsieht, kann dies jedoch nicht durch die Versioneriungsdelegierten sichergestellt werden. Falls keine Transaktion aktiv ist, könnte der Delegierte automatisch eine starten, die die Operation umschließt. Problematisch ist jedoch der Fall, daß bereits von höheren Schichten eine Transaktion gestartet wurde. Hier müßte vor Beginn und nach Abschluß

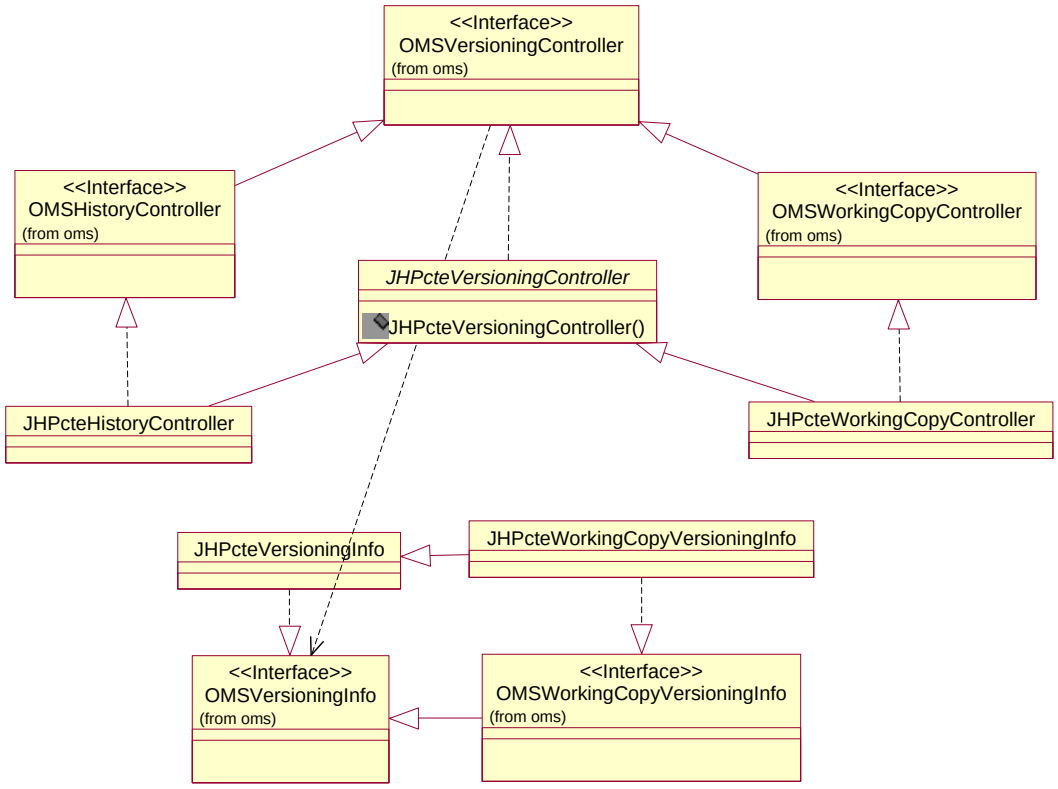


Abbildung 27.11.: Versionierungsdelegierte

der Operation ein Sicherungspunkt gesetzt werden. Höhere Schichten der $\mathcal{HEU}$ setzen Sicherungspunkte nach jedem Arbeitsschritt des Benutzers und erlauben somit einen Undo/Redo-Mechanismus. Würden hier durch den Versionierungsdelegierten als Seiteneffekt weitere Sicherungspunkte eingefügt, müßten höhere Schichten verwalten, welche Sicherungspunkte selbst erzeugt und welche als Seiteneffekt gesetzt wurden. Da im Allgemeinen das Ein-/Auschecken als eine Benutzeraktion angesehen wird, erfolgt diese Operation ohnehin geschützt durch Transaktion/Sicherungspunkte.

Für die Implemtentierung wurde bewußt darauf verzichtet, das Vorhandensein einer Transaktion zu prüfen. Vielmehr ist es Bestandteil der Schnittstellenvereinbarung der OMS-Schicht, daß die Steuerung von Transaktionen den höheren Schichten vorbehalten bleibt.

Identifikation der Datenbankobjekte Die einzelnen Versionen eines Objektes sind über einen besonderen Linktyp miteinander verbunden. Zusätzlich exitsiert zu jedem Versionsstrang ein Wurzelobjekt. Alle Objekte eines Versionsstranges erhalten die gleiche Identifikationsnummer, die beim Erzeugen der Versionswurzel vergeben wird. Diese Nummer

bezeichnet also ein Objekt über seine gesamte Lebensdauer und wird im Attribut "id" der Datenbankentität `OMSObject` gespeichert. Die einzelnen Versionen werden durch den exact-identifier von H-PCTE eindeutig gekennzeichnet.

Erzeugung von Arbeitskopien Für das Erzeugen von Arbeitskopien muß von einem Objekt in der Datenbank eine Kopie erzeugt werden. Hierzu stützen sich die Versionierungsdelegierten auf den Mechanismus des komplexen Kopierens von H-PCTE ab. Besteht ein `OMSObjekt` aus mehreren Datenbankobjekten, sind diese über Composition-Links miteinander verbunden, so daß beim Kopieren der Wurzel alle darunterhängenden Datenbankobjekte mitkopiert werden.

Alle Links, die Information des OMS-Objektes tragen, besitzen die Duplikationseigenschaft, so daß sie ebenfalls mitkopiert werden. Links die der Verwaltung des Versionskonzeptes dienen besitzen die Duplikationseigenschaft nicht.

Für die Erzeugung einer Arbeitskopie wird also zuerst überprüft, ob die Operation überhaupt erlaubt ist. Falls ja wird über den Aufruf der Kopierfunktion von H-PCTE die eignetliche Arbeitskopie erzeugt. Daraufhin werden noch Links erzeugt, die der Verwaltung der Arbeitskopien bzw. stabilen Versionen dienen und Zeitstempel gesetzt.

Einchecken Für das Einchecken eines Objektes müssen einige Verwaltungslinks gesetzt werden und Zeitstempel eingetragen werden. Weitere Veränderungen der Datenbank sind nicht notwendig.

Zugriff Die Versionerungsdelegierten erlauben Zugriff auf Objekte die in der Versionerungshierarchie vor bzw. hinter dem aktuellen Objekt liegen. Darüberhinaus kann auf das aktuell neueste Objekt eines Versionsstranges zugegriffen werden.

Die Versionerungsdelegierten bieten darüberhinaus Funtkionalität an, die in engem Zusammenhang mit dem Versionieren steht und von anderen Klassen der OMS-Schicht benötigt wird.

Weiterentwicklung Der gegenwärtige Stand der Implementierung erlaubt nur Versionsstränge, keine Verzweigungen. Bei der Planung der Realsierung des Versionskonzeptes wurde Wert darauf gelegt, daß eine Erweiterung des Versionierungskonzeptes möglich ist. Im Laufe der Implementierung ist dieser Anspruch jedoch immer weiter in den Hintergrund getreten. Wenn das Versionierungkonzept geändert werden sollte, besteht jedoch nicht nur bei den Versionerungsdelegierten Änderungsbedarf. An der ein oder anderen Stelle der OMS-Schicht wird stillschwiegend davon ausgegangen, daß Versionsäste nicht vorhanden sind. Dies wurde vor allem durch die Verwendung einer einheitlichen Identifikation für alle Objekte eines Versionsstranges forciert, die Teil der Schlüsselattribute der Links sind, die die Objekte im Schema verbinden (Abschnitt 27.6.9 auf Seite 278 geht darauf genauer ein). Rein konzeptionell sollte es jedoch möglich sein, das Versionerungskonzept auf Zweige bzw. das Verschmelzen von Zweigen zu erweitern.

Locking Der Entwurf der $\mathcal{HEU}$ sieht vor, daß OMS-Objekte durch Benutzer explizit gesperrt werden können. Hält ein Benutzer A eine solche Sperre, können andere Benutzer die von A gesperrten Objekte nicht auschecken. In diesen Zusammenhang gehören auch Methoden, die in den OMS-Interfaces vorhanden sind und Information darüber liefern, wer was ausgecheckt bzw. gelockt hat. Die Verwaltung dieser Information sowie ein Sperrmechanismus wurden nicht implementiert. Es wurden nur grobe Überlegungen angestellt, wie ein solcher Mechanismus zu implementieren sei. Hierzu einige Ideen:

1. Durch die Einführung eines einstelligen Linktypen, der als Ziel ein Datenbank-`OMSObject` hat, könnte ein einfacher Sperrmechnismus realisiert werden. Dabei muß bedacht werden, daß Benutzer in mehreren Rollen auftreten (siehe auch Abschnitt 27.6.8 auf Seite 278). Sperren sollten also einem Benutzer in einer Rolle zugeordnet werden.

2. Die Metadatenbank von H-PCTE (siehe auch Abschnitt 27.6.10 auf Seite 282) enthält Objekte, die Benutzer und Benutergruppen (Rollen) repräsentieren. Durch Einführung eines weiterern Datenbankobjektes "Benutzer in Rolle", die mit den Benutzerobjekten und den Rollenobjekten verbunden ist, ließe sich daß oben genannte Konzept des Lock-Links realisieren. Dabei ist zu beachten, daß die Existenz solcher Links die Veränderung von Gruppenzuordnungen erschwert.

3. Ein alternatives Konzept zum Sperren von Objekten wäre die Veränderung der Zugriffsrechte auf dieses Objekt. Dieser Ansatz wird von der H-PCTE-Crew bevorzugt (siehe dazu [Kel89]). Im Rahmen der Projektgruppe haben wir uns mit diesem Ansatz nicht beschäftigt.

JHPcteVersioningInfo Die Klasse `JHPcteVersioningInfo` macht Versionierunginformation nach oben zugänglich. Sie dient als reine Datenstruktur und wird über den `JHPcteVersioningController` erzeugt.

27.6.6.3. JHPcteSetController

Eines der Strukturprinzipien der $\mathcal{HEU}$ ist Enthaltensein. Der `JHPcteSetController` ist der Delegierte, der dieses Strukturprinzip realisiert. Bestandteil von Java 1.2 ist das Collection-Framework, daß ein Interface für eine Menge bereits vordefiniert (siehe [Jav]). `JHPcteSetController` implementiert das Java `Set` Inteface. Seine Implementierung stützt sich jedoch auf H-PCTE ab.

Darüberhinaus bietet `JHPcteSetController` inspizierende Methoden an, die eine Beschränkung der zurückgelieferten Elemente auf einen als Parameter übergebeben Typ erlaubt.

Umsetzung auf H-PCTE-Links Im Datenbankschema (siehe auch Abschnitt 27.6.9 auf Seite 278) existiert ein eigener Linktyp für Mengenbeziehungen. Generell können in einem `OMSSet` alle Subklassen von `OMSObject` enthalten sein. Der entsprechende Link trägt den Typnamen "contains". Während der Entwicklung das Datenbankschemas hat sich

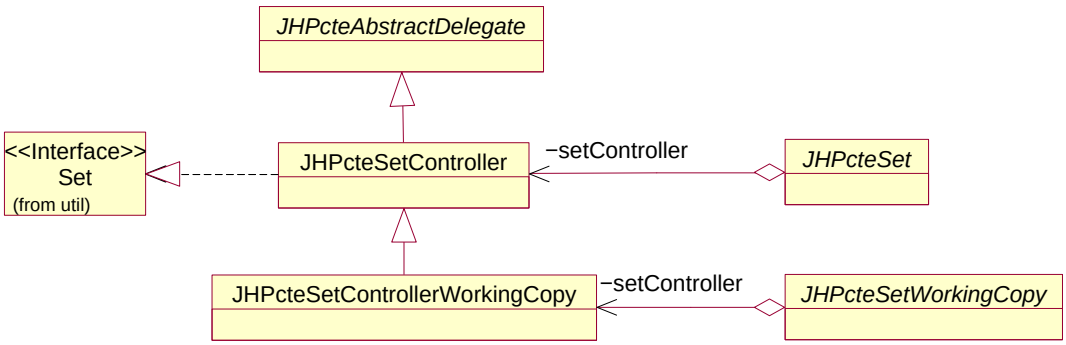


Abbildung 27.12.: JHPcteSetController

herausgestellt, daß es nützlich wäre, wenn H-PCTE Polymorphie für Links unterstützte. Hintergrund dieser Idee sind die im vorigen Absatz beschriebenen Methoden, die nur Elemente eines bestimmten Typs zurückliefern. Dann wäre für jeden Elementtyp ein neuer Subtyp des contains-Links eingeführt worden. Methoden die alle Elemente behandeln arbeiten dann mit dem allgemeinen contains-Link, Methoden die nur einen bestimmten Elementtyp zurückliefern sollen, arbeiten mit dem entsprechenden spezialisierten Linktyp. Dies hätte darüberhinaus ermöglicht in Subklassen von `OMSSet` die Menge der erlaubten Elemente mittels der Schemadefinition einzuschränken.

Es wäre natürlich möglich, den Typ der Zielobjekte aller ausgehender contains-Links zu überprüfen und auf diesem Weg die oben beschriebene Auswahl zu treffen.

Um die Zahl der Zugriffe zu minimieren und um das Navigieren durch die Datenbank mittels der HBrowse übersichtlicher zu gestalten, wurde für die contains-Links folgende Vereinbarung getroffen: Schlüsselattribute sind die Versions-ID und der Typname des Zielobjektes. Dies verhindert, daß ein Element zweimal in der selben Menge enthalten ist. Während in früheren Dokumenten noch von Kollektion die Rede ist (keine Beschränkung der enthaltenen Elemente) wurde der Name aus Konsistenzgründen später in Set (Menge) verändert, um aus dem Namen das Verhalten der Sammlung bezüglich des Einfügens von bereits vorhanden Elementen ablesen zu können.

Iterator Das Java-Collection-Framework definiert das `Iterator` Interface (siehe [Jav]). Jede Implementierung eines `Collection` Interfaces muß über die Methode `iterator()` eine Implementierung dieses Interfaces zur Verfügung stellen. Die gegenwärtige Implementierung des `Iterator` Interfaces des `JHPcteSetController` kopiert den Inhalt der Menge zum Zeitpunkt der Erzeugung des Iterators. Dies entspricht nicht der Vereinbarung des Collection-Frameworks. Ein Itertor stützt sich auf die Implementierung der Collection und definiert eine Durchlaufstrategie, so daß jedes Element genau einmal besucht wird. Der Iterator muß also in einem gewissen Sinne eine Position innerhalb der zugrundeliegenden Menge verwalten. Das H-PCTE-API garantiert jedoch keine Reihenfolge in der Ausgabe der Links, insofern darf die Linkmenge für den Iterator nur ge-

nau einmal abgefragt werden. Eine "Fail-Fast-Implementierung" des Iterators würde eine `ConcurrentModificationException` werfen, wenn sich der Inhalt der zugrundeliegenden Menge ändert. Dazu müßte jeder Iterator die Menge überwachen. Dies ist im gegenwärtigen Stand der Implementierung nicht der Fall.

27.6.6.4. Aggregation

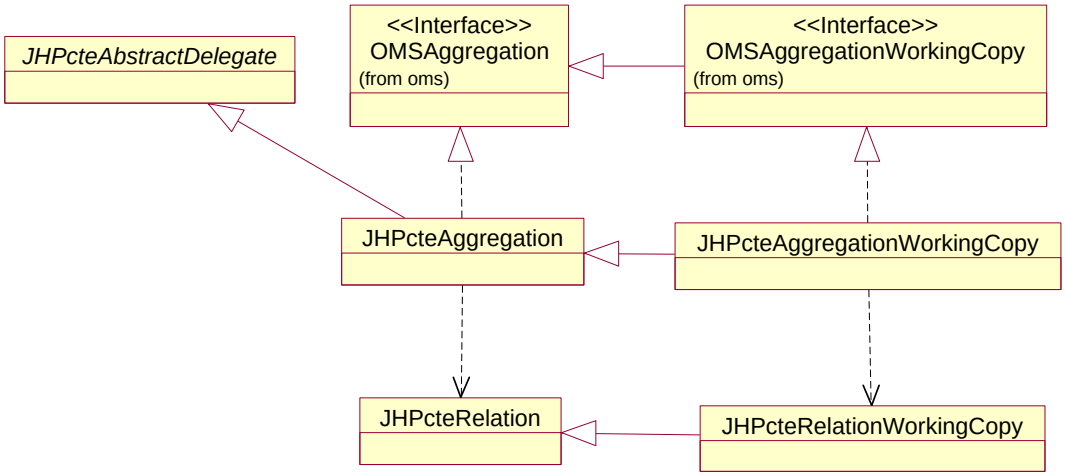


Abbildung 27.13.: JHPcteAggregation und JHPcteRelation

Ein weiteres Strukturprinzp der $\mathcal{HEU}$ ist die Verwaltung von Beziehungen zu anderen Entitäten. Für die Modellierung sind drei Klassen von Bedeutung:

- `JHPcteRelation` modelliert eine Beziehung zwischen zwei Objekten und macht Eigenschaften dieser Beziehung für höhere Schichten zugreifbar. Die Arbeitsvariante der Relation erlaubt ein Löschen.

  `JHPcteRelation` ist eine Fassadenklasse (und erbt somit von `JHPcteAbstractObject`).

- `JHPcteAggregationWorkingCopy` erlaubt die Erstellung einer Beziehung zu einem anderen Objekt. `JHPcteAggregationWorkingCopy` ist eine Delgiertenklasse (und erbt somit von `JHPcteAbstractDelegate`). `JHPcteAggregationWorkingCopy` bietet Methoden an, über die Relationen erzeugt werden können. Dazu wird sowohl der Typ der zu erzeugenden Relation als auch eine (möglicherweise leere) Liste von Attributen als Parameter übergeben. Die vorhandenen Subklassen von `JHPcteAggregationWorkingCopy` stützen sich auf diese Implementierung ab und definieren für jede zu erzeugende Relation eine neue Methode.

- Ähnlich wie bei `JHPcteAggregationWorkingCopy` wird in `JHPcteAggregation` ein Mechanismus angeboten, der die Relationen eines bestimmten Types, bzw. alle

zurückliefert. Auch hier stützen sich die Subklassen von `JHPcteAggregation` auf die Implementierung der Superklasse ab.

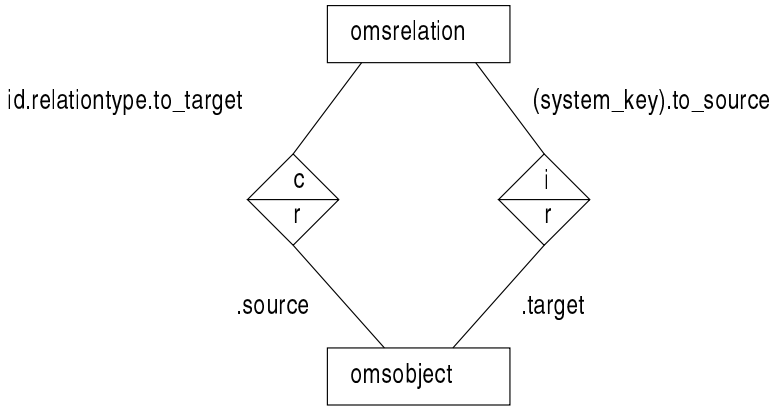


Abbildung 27.14.: Modellierung einer Relation im Datenbankschema

Modellierung in der Datenbank Relationen zwischen Informationseinheiten werden im Schema anders als bei den Mengenbeziehungen über eigene Datenbankobjekte modelliert (siehe Abbildung 27.14). Für die Erstellung von Klassendiagrammen müssen den Relationen Metadaten zugeordent werden können. Eine Modellierung der Relation als Pcte-Link hätte eine Verbindung von Metadatum mit dem Relationslink verhindert, da Links nicht Ziel von Links seien können.

Eine Relation wird im Schema über das Datenbankobjekt `OMSRelation` sowie die beiden Links "source" und "target" modelliert.

Bei der Definition der Relation und ihrer Links wurde reger Gebrauch der Kopiereigenschaften von H-PCTE gemacht. Wird ein Objekt kopiert, daß Quelle von Relationen ist, werden die Relationsobjekte automatisch mitkopiert. Gleiches gilt für die Links der Relation.

Wird ein Objekt kopiert daß Ziel einer Relation ist, werden die eingehenden Relationen nicht mitkopiert.

Source-Link Der Source-Link ist einstellig. Folglich kann jede Relation nur eine Quelle besitzen.

Der Umkehrlink zu "source" ist als Composition-Link definiert. Das hat zur Folge, daß H-PCTE die Relationen, die von einem Objekt ausgehen, als Bestandteil dieses Objektes betrachtet und beim Versionieren automatisch mitkopiert (siehe auch Abschnitt 27.6.6.2 auf Seite 269). Als Schlüsselattribute des Umkehrlinks zu "source" wird die Versions-ID und der Typ des *Zielobjektes* der Relation verwendet. Dies erleichtert das Auswählen der Relationen eines bestimmten Types. Es ist nicht möglich, daß zwischen zwei Objekten mehrere Beziehungen gleichen Typs existieren.

Target-Link Der Target-Link ist einstellig. Folglich kann jede Relation nur ein Ziel besitzen. Er besitzt die Duplikationseigenschaft, wird also beim Kopieren der Relation mitkopiert.

Der Umkehrlink zu "target" ist als n-stelliger Implicit-Link definiert. Folglich kann das System automatisch neue Schlüssel für diesen Link berechnen. Wenn ein Objekt A_1 in Relation zu B steht und von A_1 eine Nachfolgeversion A_2 erzeugt wird, muß bei B ein neuer Link erzeugt werden, da B jetzt Ziel zweier Relationen ist.

Subklassen Im Laufe der Entwicklung hat sich herausgestellt, daß die Subklassen von `JHPcteAggregation` bzw. `JHPcteAggregationWorkingCopy` unnötig sind. Wie bereits oben beschrieben ist die Funktionalität vollständig in den Superklassen angesiedelt. In den Methoden der Subklassen werden nur Methoden der Superklasse mit den richtigen Parametern aufgerufen. Dies könnte man bereits in den Methoden der Fassadenklassen tun, die einen Aggregationsdelegierten verwenden. Aufgrund der knappen Implementierungszeit wurde diese eher kosmetische Änderung der OMS-Schicht nicht vorgenommen.

27.6.6.5. Die Notifier

Adapter zwischen Pcte und Java-Beans Die Architektur der $\mathcal{HEU}$ verwendet das MVC-Konzept (siehe Abschnitt 33 auf Seite 345). Beim MVC-Konzept benachrichtigen die Modelle ihre Beobachter durch Nachrichten, im Folgenden Events genannt. Für die Kommunikation zwischen den Schichten der $\mathcal{HEU}$ werden dazu `PropertyChangeEvents` (siehe [Jav]) verwendet. Das JHPcte-API besitzt einen eigenen Benachrichtigungsmechanismus, der zur Notifikation `PcteNotificationEvents` verwendet. Die Notifizierer der OMS-Schicht dienen als Adapter zwischen diesen beiden Konzepten. Für jede Kategorie von Resourcen, die in der Datenbank überwacht werden kann, existiert eine eigene Notifiziererklasse, die die Übersetzung der übertragenen Information vornimmt. Dabei werden die im folgenden beschriebenen Vereinbarung der Schnittstellendefinition der OMS-Schicht berücksichtigt.

Vereinbarungen bei der Benachrichtigung Durch die OMS-Interfaces werden Properties definiert über deren Änderung benachrichtigt wird. Durch das `PropertyChangeEvent` wird dabei folgende Information übermittelt:

- Name des Property
- Quelle der Benachrichtigung (als Referenz auf das Modellobjekt)
- Wert des Property vor der Änderung als Java `Object` (im Folgenden als "Alter Wert" bezeichnet)
- Aktueller Wert des Property als Java `Object` (im Folgenden als "Neuer Wert" bezeichnet)

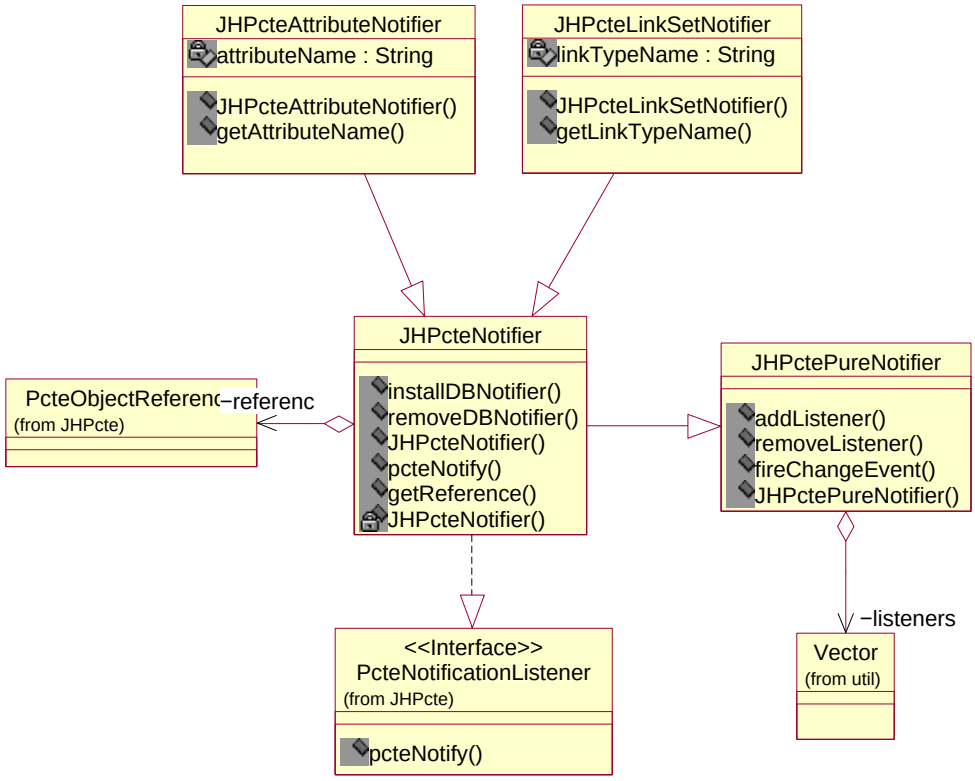


Abbildung 27.15.: Notifier

Der Name des Property wird indirekt durch die Namen der Methoden in der Schnittstelle festgelegt. Was als Wert transportiert wird, bleibt jedoch dem Programmierer des Modellobjektes vorbehalten (siehe [Eng97]). Eine vollständige Dokumentation der bei Änderungen transportierten Werte würde den Rahmen dieses Abschnittes sprengen, stattdessen werden zwei Regeln angegeben, mit deren Hilfe der genaue Inhalt von Altem und Neuem Wert abgeleitet werden kann:

1. `PropertyChangeEvents` die über Änderungen an Attributen von OMS-Objekten informieren, transportieren den Wert des Attributes vor und nach der Änderung.

2. `PropertyChangeEvents` die über Änderungen an Mengen informieren, die von OMS-Objekten verwaltet werden, transportieren nie die gesammte Menge sondern nur das Element, das die Änderung verursacht hat (enthaltene Elemente für `OMSSet`, Relationen für `OMSAggregation`.

   - Wurde ein Element entfernt, wird dieses als Alter Wert geliefert, der Neue Wert bleibt leer.

- Wurde ein Element hinzugefügt, wird dieses als Neuer Wert transportiert, der Alte Wert bleibt leer.

Die Wurzel(n) der Notifier-Hierarchie Die Klasse `JHPctePureNotifier` verwaltet eine Menge von `PropertyChangeListenern` und bietet die Methode `fire()` an, die Alten und Neuen Wert als Parameter übergeben bekommt und alle angemeldeten Listener über ein `PropertyChangeEvent` informiert. Name und Ereignisquelle werden im Konstruktor von `JHPctePureNotifier` übergeben, da sich diese Werte während der Lebensdauer eines Notifizieres nicht ändern.

Die Klasse `JHPcteNotifier` ist die abstrakte Superklasse aller Notifizierer, die eine Datenbankresource überwachen und erbt von `JHPctePureNotifier`. Damit auf Benachrichtigungen der Datenbank reagiert werden kann, wird das `PcteNotificationListener` Interface implementiert. Wird der erste `PropertyChangeListener` angemeldet, wird ein Notifizierer in der Datenbank gesetzt und die `JHPcteNotifier` Instanz als Listener angemeldet. Wird der letzte `PropertyChangeListener` abgemeldet, wird der Datenbanknotifizierer wieder entfernt. Die Subklassen von `JHPcteNotifier` müssen noch zwei Dinge spezifizieren:

1. Resource die in der Datenbank zu überwachen ist
2. Werte für Alten und Neuen Wert des `PropertyChangeEvent` bei Änderungen der überwachten Datenbankresource

Subklassen von JHPcteNotifier

JHPcteAttributeNotifier Die Klasse `JHPcteAttributeNotifier` überwacht ein Attribut einer Datenbankentität. Dazu kann dem Konstruktor als Parameter der Name des Attributes übergeben werden. Für die in 27.6.6.5 auf Seite 274 beschriebene Übersetzung der Ereignisse muß lediglich der alte Attributwert gepuffert werden, damit bei einer Änderung der alte Wert noch zur Verfügung steht, da H-PCTE bei der Benachrichtigung nur den neuen Attributwert liefert.

JHPcteDeletionNotifier Die Klasse `JHPcteDeletionNotifier` wird verwendet, um Fassadenobjekte zu benachrichtigen, das daß zugehörige Datenbankobjekt gelöescht wurde. Nachdem ein Fassadenobjekt diese Benachrichtigung erhalten hat, wird es als ungültig markiert. Wird von einer höheren Schicht ein ungültiges Objekt aufgerufen, wird eine entsprechende Exception geworfen.

Jedes Fassadenobjekt erzeugt bei der Instantiierung einen `JHPcteDeletionNotifier`.

JHPcteLinkChangeNotifier Die Klasse `JHPcteLinkChangeNotifier` überwacht Änderungen der Linkmenge eines Datenbankobjektes. Es kann entweder die Erzeugung von neuen Links oder das Löeschen von alten Links überwacht werden. Diese Eigenschaft von H-PCTE wird vor höheren Schichten versteckt. Demtentsprechend werden für die

Überwachung von Mengen immer zwei `JHPcteLinkChangeNotifier` erzeugt (siehe dazu 27.6.6.5).

JHPcteLinkSetNotifier Objekte, die Mengen verwalten, informieren über die Änderung an diesen Mengen wie in Abschnitt 27.6.6.5 auf Seite 274 beschrieben. Bei Änderungen am Inhalt der Menge, wird ein Element der Menge mit der Benachrichtigung transportiert. Im Datenbankschema werden die Beziehungen zwischen Mengen und Elementen durch Links modelliert. Um über Änderungen an der Menge informiert zu werden, muß die Linkmenge des Mengenobjektes überwacht werden (siehe dazu 27.6.6.5 auf der vorherigen Seite). Um auf das betroffene Element zugreifen zu können, muß der Link aufglöst werden. Wurde jedoch ein Link gelöscht, ist es nicht mehr möglich, auf das Zielobjekt des Links zuzugreifen, da der Link nicht mehr existiert. Von H-PCTE wird lediglich der Name des gelöschten Links übermittelt. Die Klasse `JHPcteLinkSetNotifier` verwaltet aus diesem Grund eine Abbildung von Linknamen auf Fassadenobjekte, die beim Erzeugen der Instanz aufgebaut wird und bei jeder Änderung der zugrundeliegenden Menge aktualisiert wird.

JHPcteLinkSetOfTypeNotifier Die Schnittstelle der OMS-Schicht läßt es zu, daß die Benachrichtigung bei der Änderung an Mengen auf Elemente eines Typs beschränkt wird. Die Klasse `JHPcteLinkSetOfTypeNotifier` arbeitet im wesentlichen genauso wie die Klasse `JHPcteLinkSetNotifier`, jedoch werden alle Benarichtigungen, über Objekte, die nicht den ausgezeichneten Typ besitzen, unterdrückt.

27.6.7. Die Fassadenklassen

27.6.7.1. JHPcteObject

Als wesentliche Dienstleistung der Klasse `JHPcteObject` ist die Erzeugung von Objekten in der Datenbank zu nennen. Soll ein neues `OMSObject` angelegt werden, müssen in der Datenbank drei Objekte erzeugt und verlinkt werden: Die Versionswurzel, das erste versionierte Objekt sowie ein Dokumenatationsobjekt. `JHPcteObject` bietet eine Klassenmethode `create()` an, über die ein neues Objekt in der Datenbank angelegt werden kann. Subklassen von `JHPcteObject` überschreiben diese Methode. Es hat sich gezeigt, daß der Erzeugungsprozess weitgehend parametrisierbar ist, so daß in den Subklassen nur eine Typzeichenkette geändert werden muß. Eine Ausnahme bildet hier `OMSProject`, doch dazu mehr im Abschnitt 27.6.7.2.

27.6.7.2. JHPcteProject

Die Klasse `JHPcteProject` dient als Behälter für alle Objekte eines Projektes. Unabhängig von der durch den Benutzer vorgegebenen Struktur der Objekte innerhalb des Projektes, ist jedes in einem Projekt erzeugte Objekt in diesem Behälter enthalten. Genauer gesagt, ist die Versionswurzel jedes Objektes Element des Projektes. Zum gegenwärtigen Stand der Implementierung, ist `OMSProject` die einzige Subklasse von `OMSSet`, die Versionswurzeln enthält. Bei Zugriffen auf Elemente eines Projektes wurde vereinbart,

die gerade aktuelle Version des enthaltenen Objektes zurückzugeben, da die Existenz der Versionswurzeln eine Hilfskonstruktion der OMS-Schicht ist und für höhere Schichten nicht direkt zugreifbar sein muß.

Zum gegenwärtigen Stand der Implementierung ist eine Versionierung von Projekten nicht möglich. Rein prinzipiell stellt die Versionierung eines Projektes jedoch kein Problem dar. Der in `OMSSet` definerte Mechanismus würde mit folgender Modifikation die Versionerung von Projekten erlauben: Wie oben dargestellt enthalten Projekte keine Objekte in einer Version, sondern nur deren Versionswurzeln. Soll ein Projekt versioniert werden, müßten mit der Kopie des Projektes die Links zu den Versionswurzeln durch Links zur gerade aktuellen Version der enthaltenen Objekte ersetzt werden. Ein Projekt zu versionieren würde demnach dazu führen, daß der aktuelle Projektstand eingeforen wird, daß heist es wird hinterlegt, welche Objekte zu diesem Zeitpunkt Bestandteil des Projektes waren und in welcher Version sie vorlagen.

Bei der gegenwärtigen Implementierung wird bei der Erzeugung eines Projektes jedoch keine Versionswurzel angelegt. Diese Entscheidung erscheint dem Autor im Nachhinein als nicht nachvollziehbar und ist nur mit dem großen Zeitdruck der Implementierung zu erklären. Ist eine Versionierung von Projekten gewünscht, wären weitergehende Modifikationen an der OMS-Schicht notwendig als im vorigen Absatz beschrieben. In diesem Zusammenhang ist ebenfalls zu erwähnen, daß es sinnvoll wäre, die Datenbankentität `HEU` ebenfalls als Subklasse von `OMSSet` zu modellieren. Dann wären Projekte einzige Elemente der Menge `HEU`. Dies würde die Behandlung der im Schema vorhanden Entitäten vereinheitlichen.

27.6.8. Rechte & Rollen

Die Schnittstellen der OMS-Schicht sehen eine Vergabe von Rechten an Objekten sowie die Einteilung von Rollen vor. Ein Benutzer kann eine von möglicherweise mehreren Rollen annehmen. Die Rechte an den einzelnen Objekten können für die verschiedenen Rollen unterschieldich sein.

Das Sicherheitssystem von H-PCTE erlaubt es Benutzer und Gruppen zu verwalten. Eine Realisierung des gewünschten Rollenbegriffs durch die HPCTE-Gruppen sollte möglich sein. Für die gegenwärtige Implementierung wurde der Bereich Rechte & Rollen ausgeklammert, da die Zeit fehlte, die notwendigen H-PCTE-Kenntnisse zu erwerben.

27.6.9. Das Datenbankschema

Im Folgenden wird davon ausgegangen, daß der Leser im Groben mit den Möglichkeiten der SDS (Schema Definition Language) von H-PCTE vertraut ist und ER-Diagramme lesen kann.

Es wird beschrieben, wie die in den vorangegangenen Abschnitten erwähnten Klassen ihre Daten im Schema ablegen. Da Links in H-PCTE immer aus Link und Rücklink bestehen und die Zuordnung eindeutig ist, wird im folgenden immer nur einer der Linknamen verwendet.

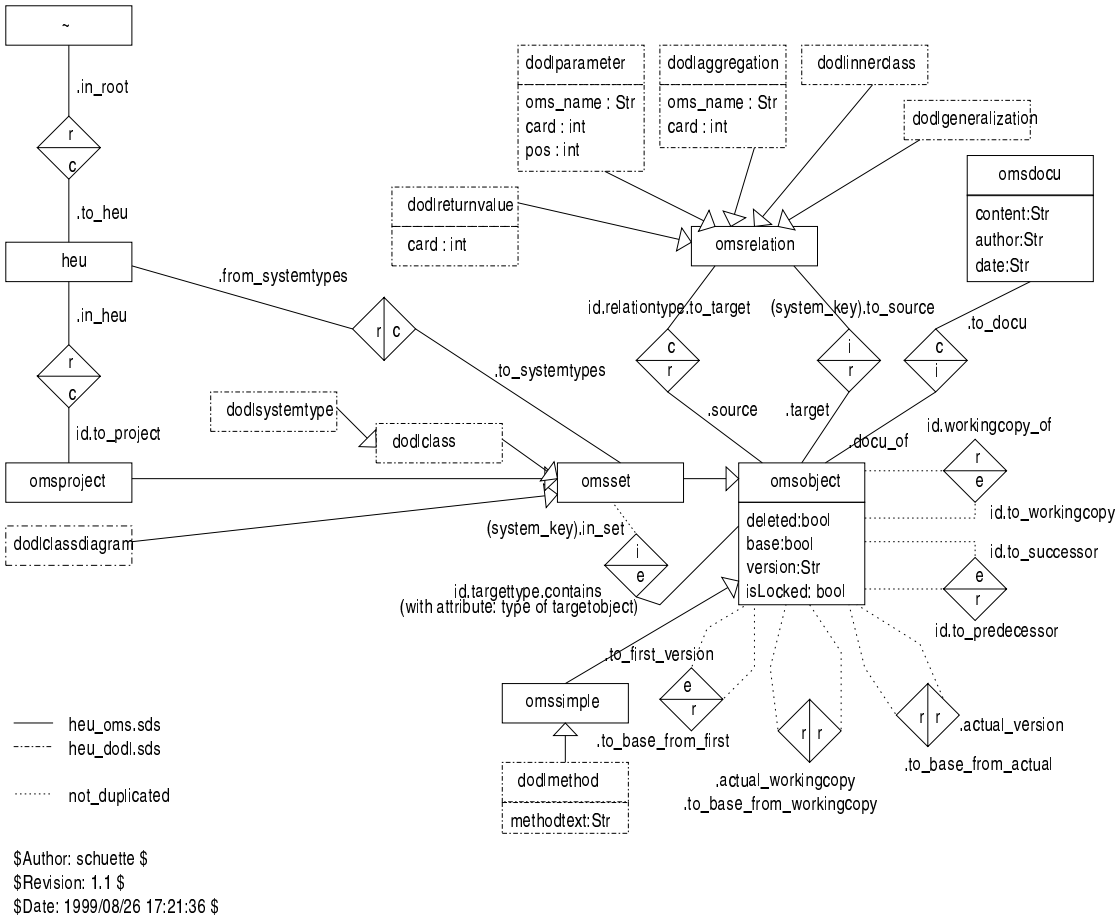


Abbildung 27.16.: Datenbankschema

27.6.9.1. Base und Versioned Object

Beim Blick auf Abbildung 27.16 fällt die Entität `OMSObject` ins Auge. `OMSObject` ist die Superklasse aller Datenbankentitäten. Bei `OMSObject` sind die Links angesiedelt, die für die Verwaltung des Versionskonzeptes notwendig sind. Das Versionskonzept sieht vor, daß einzelene Versionen eines Objektes über entsprechende Links miteinander verbunden sind und somit einen Versionsstrang bilden (siehe auch 26 auf Seite 231). Zusätzlich existiert zu jedem Strang eine Versionswurzel, die den ganzen Strang, nicht jedoch eine bestimmte Version kennzeichnet. `OMSObject` wird verwendet um sowohl versionierte Objektes als auch Versionswurzeln anzulegen. Die an der Entität eingetragenen Links lassen sich eindeutig den beiden Rollen zuordnen, in denen `OMSObject` auftritt. Wird `OMSObject` als Versionswurzel verwendet, ist hat das Attribut "base" den Wert wahr.

Zu Beginn der Entwicklung erschien es sinvoll, `OMSObject` auf diese Weise zu verwenden. Im Nachinein erscheint es günstiger, `OMSObject` aufzuteilen. Dann wäre im Schema eine Entität AbstractObject einzuführen, die gemeinsame Superklasse von BaseObject

und VersionedObject ist. Die bei `OMSObject` angeordneten Links wären dann geeignet aufzuteilen.

Versionslinks Der Link "to_successor" verbindet zwei versionierte Objekte. Existiert zu einem Objekt eine stabile Nachfolgeversion, so ist sie über "to_successor" zu erreichen. Über den Umkehrlink "to_predecessor" ist der Vorgänger zu erreichen, sofern er existiert. Existiert zu einer stabilen Version einer Arbeitskopie, so ist diese über "to_workingcopy" zu erreichen.

Wird `OMSObject` in der Rolle einer Versionswurzel verwendet, führt der Link "to_first_version" zu ersten stabilen Version. "actual_workingcopy" führt zur aktuellen Arbeitskopie, sofern eine existiert. Der Link "actual_version" führt zur neuesten stabilen Version.

Alle Versionslinks sind als nicht-duplizierbar markiert, da diese Links von `JHPcteVersioningController` beim Ein-/Auschcken umgesetzt werden müssen und es somit keinen Sinn macht, sie von H-PCTE automatisch mitkopieren zu lassen.

27.6.9.2. OMSSet

`OMSset` erweitert `OMSObject` um den Link "in_set". Ist ein `OMSObject` Element einer Menge, sind beide Entitäten über den "in_set"-Link miteinander verbunden. Würde man wie in 27.6.9.1 auf der vorherigen Seite beschrieben `OMSObject` aufteilen, müßte der "in_set"-Link zu AbstractObject führen, so daß sowohl Baseobjects als auch VersionedObjects in einer Menge enthalten sein können.

Der "contains"-Link besitzt die Duplikationseigenschaft, so daß beim Kopieren einer Menge die Information über die enthaltenen Elemente automatisch mitkopiert wird. Der Umkehrlink "in_set" besitzt die Duplikationseigenschaft nicht.

27.6.9.3. OMSRelation

`OMSAggregation` existiert nicht als eigenständige Entität. Die zu speichernde Information steckt in den Relationen. Um Relationen zwischen beliebigen `OMSObject` erzeugen zu können, ist die `OMSRelation`-Entität mit `OMSObject` verbunden. Eine genaue Erklärung der Links ist in Abschnitt 27.6.6.4 auf Seite 272 zu finden. Die Schnittstelle der OMS-Schicht sieht gegenwärtig fünf Typen von Relationen vor. Zur jedem dieser Typen existiert eine korrespondierende Entität im Schema, die von `OMSRelation` erbt. Je nach Art der Relation wurden einige Attribute eingeführt, die die verwendeten Daten aufnehmen können.

27.6.9.4. Beziehungen zu anderen Objekten

Im Schema der $\mathcal{HEU}$ werden zwei Sorten von Daten abgelegt. Zum einen die Zustandsbeschreibung der einzelnen Objekte, zum anderen die Beziehungen der Objekte untereinander. In H-PCTE werden Beziehungen zwischen Objekten durch Links gespichert. Während jedoch für die Objekte in der Datenbank Polymorphie existiert, gibts für die Links kein vergleichbares Konzept.

Beim Entwurf der OMS-Schicht wurde versucht, durch die Klassen `OMSObject`, `OMSSet`, `OMSAggregation` und `OMSRelation` geeignete Abstraktionen für die Speicherung der Daten der $\mathcal{HEU}$ zu finden, die den Löwenanteil der Implementierung bereits beinhalten und nur gerinfügig ergänzt werden müssen um konkrete Ableitungen (*DoDL*-Klasse, Klassendigramm,...) zu bilden. Insbesondere die Verwaltung der Beziehungen der Objekte untereinander soll in den abstrakten Klassen `OMSSet` und `OMSAggregation` angesiedelt sein. Das führt jedoch bei der Schemadefinition zu Problemen. Idealerweise sollten in einem Klassendiagramm bespielsweise auch nur *DoDL*-Klassen enthalten sein, nicht jedoch Projekte. Man könnte dies erreichen, indem man im Schema einen entsprechenden Linktyp definiert, der *DoDL*-Klassen und Klassendiagramme verbindet. Im Fall der $\mathcal{HEU}$ gäbe es viele solcher Linktypen, denen jedoch eines gemein ist: Sie alle drücken ein Enthaltensein aus und können auf die gleiche Weise behandelt werden. Bei der Implementierung der OMS-Schicht wurde der umgekehrte Weg eingeschlagen. Es gibt einen allgemeinen Linktyp "contains", der Elemente und Mengen verbindet und im Schema `OMSObject` und `OMSSet` miteinander verbindet. So läßt sich der Code zur Verarbeitung in `OMSSet` ansiedeln. Dies hat jedoch zur Folge, daß die Möglichkeit, die Konsistenz der Daten mittels des Schemas sicherzustellen, verschenkt wird, da es laut Schemadefinition erlaubt ist, daß jeder Subtyp von `OMSSet` jeden Subtyp von `OMSObject` enthalten kann.

Ein ähnliches Problem stellt sich bei der Verwaltung der Relationen. Laut Schema können alle Objekte in der Datenbank zu allen Objekten in jeder Beziehung stehen, da alle Subtypen von `OMSRelation` die "source" und "target" -Links erben.

Metadaten zu Beziehungen Sofern Metadaten zu Datenbankobjekten abzuspeichern sind, sollten sie über geeignete Links mit den Datenbankobjekten verbunden werden. Dies ist im Fall der Relationen kein Problem, da Relationen als Objekte in der Datenbank abgespeichert werden. Die Mengenbeziehung wird nur als Link verwaltet, aus diesem Grund sind Metadaten zur Mengenbeziehung nicht möglich.

Schlüsselattribute der Links Die Links zur Speicherung von Beziehungen zwischen Objekten der OMS-Schicht tragen als Schlüsselattribute zwei Werte:

1. Die Nummer des Verionsstranges des Zieliobjektes im Feld "id" (siehe auch 27.6.6.2 auf Seite 268). Der Link "to_target" trägt also die Versions-ID des Zielobjektes der Relation und nicht etwa die ID der Relation selbst.

2. Den Typ des Zielobjektes im Feld "Targettype" (`OMSSet`) bzw. den Typ der Relation im Feld "Relationtype".

27.6.9.5. Andere Entitäten

`OMSSet` und `OMSObject` sind als abstrakte Datenbankobjekte anzusehen; nur Subtypen werden tatsächlich zur Speicherung von Daten verwendet. Der sehr generische Ansatz der $\mathcal{HEU}$ beschränkt die notwendigen Ergänzungen auf ein Minimum. Um die Daten der konkreten Realisierung aufzunehmen, muß der Obekttyp um entsprechende Attribute

erweitert werden. Sollen komplexere Daten abgelegt werden, können über Composition-Links weitere Datenbankobjekte "angehängt" werden, die die zu speichernde Information aufnehmen. Da beim Versionieren das "Komplexe Kopieren" von H-PCTE zum Einsatz kommt, können solche Strukturen vom Versionierungssystem der OMS-Schicht verarbietet werden, ohne das Änderungen notwendig sind (siehe auch Abschintt 27.6.6.2 auf Seite 267).

DoDLsystemtype *DoDL* bietet eine Reihe von Systemtypen wie z.B. "String" an. Um die Behandlung von Attibuten aller Art einer *DoDL*-Klasse in der OMSSchicht einheitlich zu gestalten, werden mit der Installation der $\mathcal{HEU}$ einige Objekte in der Datenbank angelegt, die die *DoDL*-Systemtypen repräsentieren. So werden einheitlich Attribute aller Art durch eine `DoDLAggregation`-Relation zwischen Klasse und Zielobjekt realisiert. *DoDL*-Systemtypen werden in Datenbankobjekten vom Typ `DoDLSystemType` abgelegt.

In diesem Zusammenhang ist ein Mangel des Schemas zu kritisieren: Der Link "to_systemtypes" wurde zwischen `HEU` und `OMSSet` eingeführt. Der Link wurde notwendig, da die *DoDL*-Systemtypen unabhängig von einzelnen Projekten in der Datenbank abgelegt werden und von jedem Projekt aus zugreifbar sein sollen.Es wäre besser gewesen eine Subklasse von `OMSSet` einzuführen, beispielsweise `DoDLSystemTypeContainer` und den "to_systemtypes"-Link dort anzusiedeln.

27.6.9.6. Metadaten

Aus dem Schema in Abbildung 27.16 auf Seite 279 abgebildeten Schema ist zu entnehmen, daß zum gegenwärtigen Stand der Implementierung der $\mathcal{HEU}$ noch keine Metadaten vorhanden sind (siehe dazu auch 27.3 auf Seite 248). Wie Metadaten im Schema abzulegen wären wurde im Verlauf der Entwicklung nur kurz überlegt, hier die wesentlichen Ideen:

- Metadaten lassen sich als eigenständige Objekte in der Datenbank ablegen und können über Composition-Links mit der Ursprungsobjekt verbunden werden.
- Metadaten können als Subtypen von `OMSObject` bzw. von `OMSSet` realisiert werden. Die Beziehung zu dem Ursprungsobjekt könnte dann mit den Strukturierungsmöglichkeiten der OMS-Schicht dargestellt werden (Relation bzw. Enthaltensein).

27.6.10. Verwendung der Metadatenbank von H-PCTE

Die Metadatenbank von H-PCTE stellt durch den Mechanismus der Selbstreferentialität unter anderem die Möglichkeit zur Verfügung, Information über die Beziehungen der vorhandenen Objekttypen sowie über aktive Transaktionen zu erhalten. Von dieser Möglichkeit wurde in der gegenwärtigen Implementierung jedoch kein Gebrauch gemacht. Im Code sind an den entsprechenden Stellen Kommentare eingetragen, die auf die Metadatenbank von H-PCTE verweisen. Stattdessen wurden Übergangslösungen programmiert, die zügiger zu realisieren waren. Möglicherweise unterstützt dasJH-PCTE-API zukünftiger Versionen den Zugriff auf die Metadatenbank, was eine Korrektur der oben erwähnten Codestellen mit geringem Aufwand erlauben würde.

27.6.11. Ideen zur Verbesserung

Sofern möglich, wurden Ideen zur Verbesserung der Implementierung im jeweiligen Abschnitt aufgeführt. An dieser Stelle sollen Ideen skizziert werden, die sich keinem der in 27.6 auf Seite 259 vorhandenen Abschnitte zuordnen lassen.

- Die Klasse `JHPcteSetController` ließe sich deutlich breiter nutzen, wenn der folgende Teil seiner Funktionalität durch eine separate Klasse übernommen würde: Bei Zugriffen auf die Elemente einer Menge müssen die Linknamen zu Zielobjekten aufgelöst werden. Zu diesen Zielobjekten müssen die entsprechenden Fassadenklassen herausgesucht bzw. instantiiert werden (siehe auch 27.6.5.1 auf Seite 266). Die gegenwärtige Implementierung geht dabei von dem Spezialfall aus, daß in der Menge nur versionierte Objekte enthalten sind. In diesem Fall ist das Zielobjekt des Links bereits das enthaltene Element. Sind jedoch Versionswurzeln in einer Menge enthalten, muß nach einem bestimmten Verfahren eines der möglichen versionierten Objekte ausgewählt werden (siehe auch 27.6.7.2 auf Seite 277). In der gegenwärtigen Implementierung existiert eine Subklasse von `JHPcteSetController`, die Versionswurzeln immer zum aktuellen Objekt auflöst. Es sind jedoch auch andere Strategien denkbar. Wenn ein `JHPcteSetController` bei der Instantiierung ein Objekt übergeben bekommt, das daß Auflösen der Links übernimmt, ließen sich verschiedene Strategien zur Auflösung hinter einer einheitlichen Schnittstelle verbergen und eine Implementierung von `JHPcteSetController` würde ausreichen.

28. Modell-Schicht der $\mathcal{HEU}$-Architektur

Autor: *Sascha Lüdecke, Thomas Sparenberg*

Wir haben die OMS-Schicht in Hinblick auf Struktur bzw. Persistenz, die Modell-Schicht in Hinblick auf Semantik bzw. Verwendung in einer Applikation entworfen. In diesem Kapitel beschreiben wird letzteres aus drei Sichten, nämlich (1) von oben, d.h. die verwendbare Schnittstelle, (2) nach unten, welche OMS-Teile wie verwendet werden und (3) von innen, wie wir implementiert haben. Da wir keine konkrete Soll-Vorgabe entwickelt haben, versuchen wir abschliessend, die Modellschicht kritisch zu betrachten.

28.1. Modellschicht-Objekte

Autor: *Thomas Sparenberg*

Wie bereits in Kapitel 25.2 auf Seite 210 erläutert, ist die Modell-Schicht das Zwischenglied zwischen der OMS-Schicht und der GUI. Sie hat die Aufgabe, die Daten, die die GUI anfordert, aus der OMS-Schicht anzufordern, wobei das keinesfalls nur ein Durchschleifen von Anfragen bedeutet. Die Abbildung von auf der GUI dargestellten Daten auf Objekte in der Objektbank ist relativ komplex, da die beiden anderen Schichten jeweils andere Politiken zur Verwaltung ihrer Objekte verfolgen. Siehe dazu auch das Kapitel 26 auf Seite 231.

28.1.1. Architektur

Die Architektur der Modell-Schicht ist ebenfalls, wie die ganze $\mathcal{HEU}$auch, in drei Schichten aufgeteilt: die generische, die *DoDL*- und die Klassendiagramm-Schicht. Diese Schichten mit den dazugehörigen Klassen wird in Abbildung 28.1 auf der nächsten Seite veranschaulicht.

Durch das Model-View-Controller-Konzept (vgl. Kapitel 33 auf Seite 345) werden Änderungen an Objekten erst durch die entsprechende Notifikation allen Listenern bekanntgegeben. Da eine *DoDL*-Klasse aber mehrere Klassen und Methoden beinhaltet (Aggregationen und innere Klassen sind auch wieder Klassen), muß eine Modell-Klasse (MDC, s.u.) bei allen ihren inneren Klassen, Aggregationen, Methoden und der Superklasse als Listener angemeldet werden, damit solche Änderungen angezeigt werden können. Dadurch ist eine Modell-Klasse häufig Listener bei mehreren OMS-Resourcen.

Andererseits ist es durch das verteilte Arbeiten möglich, daß eine Modell-Klasse mehrmals in verschiedenen GUI-Instanzen (auf welche Art und Weise auch immer) dargestellt

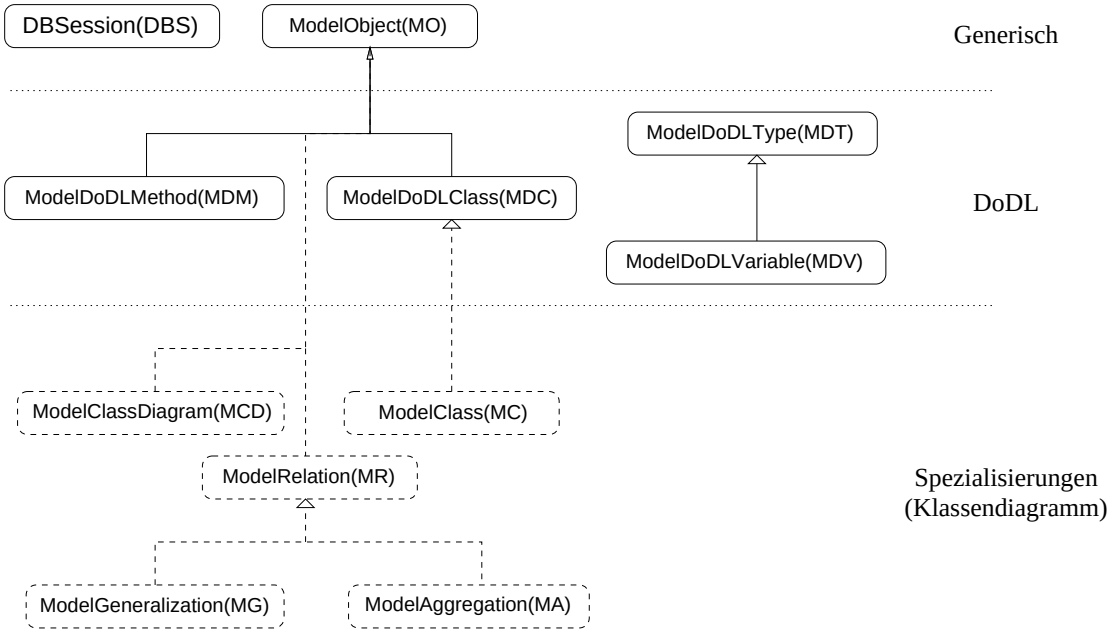


Abbildung 28.1.: Struktur der Modellschicht

wird. Da diese GUIs auch von den Änderungen der Modell-Klasse erfahren wollen, werden sie sich sort als Listener anmelden. Daraus, daß eine Modell-Klasse an mehreren Objekten der OMS-Schicht als Listener angemeldet ist und daß sich wiederum mehrere GUI-Instanzen bei dieser Modell-Klasse als Listener anmelden, ergeben sich die Listener-Abhängigkeiten, wie sie in Abbildung 28.2 auf der nächsten Seite dargestellt sind. Das Gleiche gilt natürlich auch für Modell-Methoden, jedoch in einem etwas geringeren Umfang.

Des weiteren hat die Modell-Schicht die Aufgabe, Methodenaufrufe von GUI-Objekten an die Objektbank zu vermitteln. Sobald eine GUI-Instanz ein Modell-Objekt beauftragt, einen Wert zu lesen oder zu verändern, muß dieses Modell-Objekt sich die anzusprechende(n) OMS-Instanz(en) suchen und diese mit der Ausführung beauftragen. Oft sind mehrere solcher Schritte, in denen das passende OMS-Objekt gesucht und danach mit einer Aufgabe betreut wird, in einem Auftrag von der GUI ans Modell nötig. Die schematische Visualisierung dieser Methodenaufrufe zwischen GUI, Modell und OMS ähnelt dem Listener-Abhängigkeiten-Diagramm und ist in 28.3 auf der nächsten Seite dargestellt.

28.1.2. Aufgaben

Da die Modell-Objekte in drei Modellschicht-Schichten eingelagert sind, werden auch ihre Aufgaben in dem entsprechenden Schichten-Kontext erläutert.

1. Generische Schicht

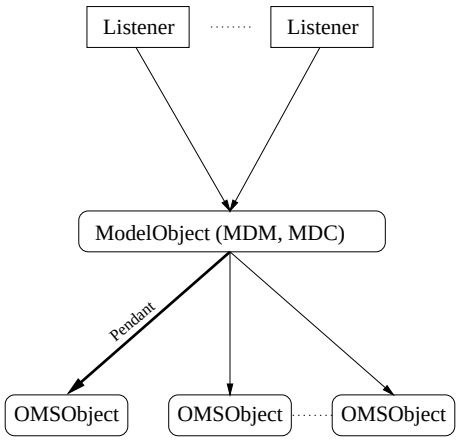


Abbildung 28.2.: Listener-Abhängigkeiten

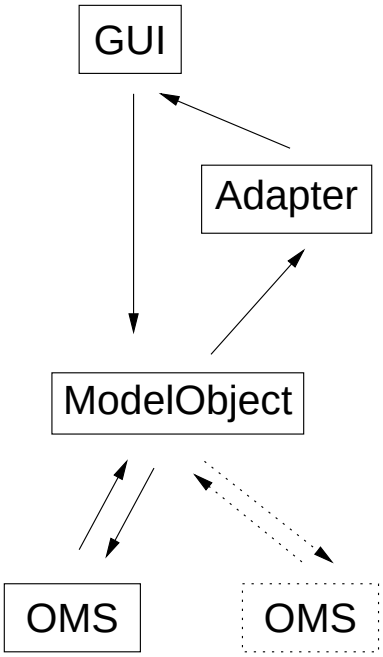


Abbildung 28.3.: Verbindung der Schichten

Das ist die Schicht, in der die generischen Objekte residieren. Es handelt sich dabei eigentlich nur um das `ModelObject` (im weiteren auch mit `MO` bezeichnet), das mit Hilfe von `DBSession` die administrative Verbindung zur Objektbank hält. Das `ModelObject` vereint die Funktionalität in sich, die alle weiteren Modell-Objekte zur Zusammenarbeit mit der OMS-Schicht benötigen. Das umfaßt sowohl das Halten der Verbindung zum Datenbankpendant (s.u.) und das Auslesen und Schreiben von Werten aus dem / in das Pendant-Objekt, als auch administrative Methodik, z.B. das Sperren von Resourcen oder das Ein-/Aus-checken von Versionen. Diese Details werden in diesem Kapitel noch tiefer behandelt. Das `ModelObject` ist Superklasse von zwei Klassen aus der nächsten Schicht, der

2. *DoDL*-Schicht

   Diese Ebene spezialisiert das generische `MO` zu zwei *DoDL*-spezifischen Ausprägungen, und zwar zur `ModelDoDLClass` (`MDC`) und zur `ModelDoDLMethod` (`MDM`). Die `MDC` repräsentiert eine *DoDL*-Klasse, die über eine Superklasse, einige Attribute und Methoden verfügt, eventuell auch noch innere Klasse ihrer äußeren Klasse ist und selbst wiederum innere Klassen haben kann. Da alle diese Attribute entweder wiederum *DoDL*-Klassen sind, was mit der `MDC` schon abgedeckt wird, oder *DoDL*-Methoden sind, haben wir auch noch die `MDM` realisiert: sie besitzt einen Rückgabetyp inklusive Rückgabekardinalität, eine sortierte Liste von Paramtern und natürlich einen Namen sowie eine sie umgebende Klasse. Beide ModelDoDL-Objekte haben sich bei einer Menge von OMS-Objekten als Listener anzumelden, um sie zu überwachen, und können von mehreren GUI-Objekten benutzt werden.

   Zu diesem Zeitpunkt haben wir Klassen und Methoden; wir wissen, daß sie irgendwie in Beziehung stehen, müssen diese Beziehungen aber noch speichern bzw. darstellen können. Da die OMS-Schicht alle diese Relationen als eigene Objekte speichert, die GUI-Schicht aber keine Relationsobjekte benötigt, haben wir hier in der Modell-Schicht einen Bruch geplant. Unsere Modell-Objekte kennen einerseits die Granularität der Relationen der Objektbank, andererseits bietet sie der GUI Modell-eigene Datentypen an, in denen diese Beziehungen nur noch "inhaltlich" dargestellt werden, d.h. in diese Datentypen steht nicht mehr das Start- und Zielobjekt der Relation, wie es auf OMS-Seite der Fall ist, sondern beinhaltet nur eine ModelObject-Instanz (also `MDC` oder `MDM`) des Zeilobjektes und die Werte, die zu einer solchen Relation gehören können, als da wären Typ, Kardinalität, Name und Position. Diese Modell-eigenen Datentypen sind der `ModelDoDLType`(`MDT`) und dessen Subklasse, die `ModelDoDLVariable`(`MDV`). Die Namenswahl dieser Klassen ist eigentlich unglücklich, da im `MDT` nicht einfach nur ein Typ steht, und vor allem, weil eine Variable im normalen Sprachgebrauch nicht von einem Typen erbt, aber diese Namen sind historisch gewachsen und eigentlich nur Gewöhnungssache. Uns selbst ist dieser Namenskonflikt anfangs garnicht aufgefallen.

   Der `MDT` beinhaltet sowohl eine `MDC` als Zielklasse der Relation (also den Typen einer Relation, wobei Relation z.B. eine Aggregation oder eine innere Klasse sein kann) als auch die Kardinalität dieser Relation. Damit reicht der `MDT` zur Speicherung von

Relationen zu inneren Klassen und zu Rückgabewerten von Methoden aus. Für anspruchsvollere Relationen, wie z.B. die zu Aggregationen oder zu Parametern, die noch einen Namen benötigen, schufen wir die `MDV`. Sie kann eben noch diesen Namen aufnehmen.

3. Klassendiagramm

   Eigentlich sollte diese Schicht diejenige sein, die die Standard-*DoDL*-Klassen für die verschiedenen Editoren, sprich GUIs, spezialisiert. Da der Klasseneditor jedoch mit der Ausprägung der Modell-DoDL-Schicht ausreichend bedient war, blieb hier nur die Spezialisierung für den Klassendiagrammeditor übrig. Deshalb hat diese Modell-interne Ebene auch den Namen Klassendiagramme(Editor)-Ebene bekommen. Hier gibt es die Klassen `ModelClassDiagramm`(`MCD`) und `ModelRelation`(`MR`), die beide von `MO` erben, die `ModelClass`(`MC`), die die `MDC` spezialisiert, und sowohl die `ModelAggregation`(`MA`) als auch die `ModelGeneralization`(`MG`), die beide die `MA` erweitern. Wer welche Aufgaben hat, weiß ich doch nicht.

28.2. Benutzung der OMS-Schicht

Autor: *Sascha Lüdecke*

Wir haben versucht, die konkrete Ausprägung der OMS-Schicht vor den Applikationen über der Modellschicht zu verbergen. Dies war notwendig, da wir die unterschiedlichen Ebenen unter verschiedenen Gesichtspunkten entworfen haben (siehe auch Kapitel VI auf Seite 209). Dadurch entstand ein Bruch in der Modellierung, der in der Modell-Schicht zum Tragen kommmt.

In diesem Abschnitt gehen wir auf die Zusammenführung der beiden Schichten über das MVC-Konzept und die Datenbanksitzung bzw. Versionierung ein.

28.2.1. Gegenstücke in der OMS-Schicht

Die Objekte in der Modell-Schicht verwenden mehrere Objekte aus der OMS-Schicht, um ihre Dienste zu realisieren. In der Regel haben sie eine Haupt-Gegenstück (Pendant), von dem die wesentlichen Daten stammen, und mehrere weitere Objekte, die meist Bestandteil des Pendant sind. Ein Beispiel ist `ModelDoDLClass` (MDC), die sich auf ein `OMSDoDLClass`-Objekt und mehrere `OMSDoDLMethod`-Objekte stützt. In der Modellschicht werden, mit Ausnahme einer Identifikationsnummer (ID) keinerlei Daten gespeichert, alle lesenden und schreibenden Anfragen gehen direkt an die OMS-Objekte. Die gespeicherte ID ist die des jeweiligen Versionsstranges.

Das Modell-Objekt meldet sich bei allen OMS-Objekten als Listener für die verschiedenen Aspekte an (siehe Abschnitt 28.3 auf Seite 291). Alle Ereignisse, die lediglich eine Änderung an den Daten, wie Namen oder Dokumentation, darstellen, werden nach oben an die entsprechenden Listener in `nameListeners, lockerListeners` und `documentationListeners` weitergeleitet. Events, die eine Änderung der Gültigkeit der OMS-Objekte bedeuten, beschreiben wir in Abschnitt 28.2.2.2 auf Seite 291.

die Zusammenhänge zwischen Modell- und OMS-Schicht sind in Bild 28.2 auf Seite 287 nocheinmal visualisiert. Die mit abgebildeten Listener über dem Modell-Objekt stellen die Schicht darüber, das heisst eventuelle Applikationen oder benutzeroberflächen, die Modell-Objekte verwendet, dar.

28.2.2. Datenbanksitzung und Versionierung

Unsere Daten werden letztendlich in einer Datenbank abgelegt, was einige Konsequenzen in allen Schichten hat. Dies ist zum einen der Begriff einer Datenbanksitzung, zum anderen Funktionalität zur Versionierung, die sich auch im Modell wiederfindet. Ersteres führte uns zu einer Klasse `DBSession`, da es objektübergreifend ist, letzteres schlug sich der Signatur von `ModelObject` nieder, da jedes einzelne Objekt für sich versioniert werden kann (siehe auch Abschnitt 26 auf Seite 231).

28.2.2.1. Datenbanksitzung

Das Paket `heu.model` enthält die Klasse `DBSession`. Sie hat eine Verbindung zu der aktuellen Datenbanksitzung in der OMS-Schicht und besitzt Methoden, um direkt die aktuelle Transaktion zu steuern, mit der Garantie, daß immer eine Transaktion geöffnet ist. Es sind:

- `save`: Führt ein `commit` aus, sichert alle Änderungen in der Datenbank.
- `rollback`: Nimmt eine Transaktion zurück, alle Änderungen sei Beginn der Sitzung oder dem letzten Neustart der Transaktion gehen verloren.
- `setSavePoint, undo`: Realisierung von Undo, geht jeweils zurück zum letzten `SavePoint`.
- `close`: Beendet die aktuelle Sitzung mit optionalem Sichern aller Änderungen. Eine Transaktion wird hier nicht neu gestartet, da diese Methode bei Programmende zum sauberen Herunterfahren der Datenbank verwendet wird.
- `getToolStarter`: Gibt Objekt zurück, mit dem ein neues Tool, und damit eine neue Sitzung, gestartet werden kann.

Eine neue Sitzung kann von hier aus leider nicht gestartet werden, da dies alles andere als trival ist. Eine Einschränkung von HPCTE war zum Beispiel, das jedes Objekt mit einer Sitzung assoziiert ist, und das jede Sitzung nur eine Transaktion besitzt. Also war es notwendig, für jedes neu geöffnete Tool eine neue Sitzung zu starten und dort ein OMS-Objekt anzufordern. Dies wurde in das Interface `ToolStarter` im Paket `heu.system` verlegt, das von `ProjectManager` implementiert wird. Zu einer näheren Beschreibung des Sitzungs- und Transaktionsmechanismus siehe auch Kapitel 27 auf Seite 239, des Projektmanagers Abschnitt 30 auf Seite 315.

28.2.2.2. Versionierung

Jedes einzelne Objekt in der OMS- und auch der Modellebene kann für sich versioniert werden. Die Funktionalität dazu ist in der obersten Klasse, `ModelObject`, angesiedelt und umfaßt folgende Funktionen:

- `checkout, checkin`: Erzeugen von Arbeitskopien und Umwandlung dieser in feste Versionen
- `delete`: Es wird eine neue, spezielle Version erzeugt, die das Objekt als gelöscht markiert
- `dismiss`: Löschen einer Arbeitskopie
- `lock, unlock`: Sperren eines Objektes für andere Benutzer

Alle diese Methoden gehen direkt weiter in die OMS-Schicht, die über den Event-Mechanismus kurz darauf eine Notifikation sendet. Das Ein- und Auschecken bzw. das Löschen einer Arbeitskopie ist ein Sonderfall, da hier *alle* Daten potentiell ungültig sind und neu gelesen werden müssen. Dies ist jedoch Angelegenheit der Schicht über dem Modell, die eine entsprechende Benachrichtigung erhält:

versionListener werden benachrichtigt, wenn eine Arbeitskopie erzeugt, eingecheckt oder verworfen wurde. In all diesen Fällen gibt es eine Version in der Datenbank, auf die zurückgegriffen werden kann. Das Modell-Objekt setzt seine Referenz automatisch darauf um.

deprecatedListener werden benachrichtigt, wenn das Objekt auch aus der Datenbank verschindet und *keine* Version existiert, auf die zurückgegriffen werden kann. Die ist das Fall, wenn in einer Transaktion ein Objekt komplett neu erzeugt und durch `rollback` wieder gelöscht wurde. Das Modell-Objekt kann dann nicht weiter verwendet werden.

28.3. Implementierung

Autoren: *Sascha Lüdecke, Thomas Sparenberg*

In diesem Abschnitt gehen wir auf die konkrete Implementierung der Modellschicht-Objekte und eventuelle Besonderheiten ein.

28.3.1. Verbindung zur OMS-Schicht

Dieser Abschnitt führt die Beschreibung von Abschnitt 28.2 auf Seite 289 auf Implementierungsebene fort.

Das Gegenstück In jedem Objekt der Modell-Schicht wird eine private Referenz (`dbObject`) auf das Gegenstück in der OMS-Schicht gehalten. Mit Ausnahme einiger privater Methoden, wie `_register`, `_deregister` und dem Konstruktor, wird an keiner Stelle direkt darauf zugegriffen. Alle Methoden holen sich die Referenz über `getDBObject()`, um einen synchronisierten Zugriff zu gewährleisten (Schlüsselwort `synchronized`). Die Methode `setDBObject` setzt die Referenz in die OMS-Schicht unter Verwendung nachfolgender Methoden um. Das kommt vor allem bei `valid`-Events aus der OMS-Schicht zum Tragen.

Rekursives An- und Abmelden Das OMS-Gegenstück wird im Konstruktor und beim Ummelden nach einem `checkin`, `dismiss`, `checkout` o.ä. initialisiert. Das bedeutet, daß sich das Modell-Objekt zunächst bei dem veralteten OMS-Objekt als Listener ab- und dann bei dem neuen Objekt anmelden muß. Da `ModelObject` nicht weiß, welche Aspekte seine Nachfolger an dem OMS-Objekt interessieren, haben wir die Aufrufstruktur wie in Bild 28.4 implementiert. Dargestellt sind dort die beiden Wege, um ein Modell-Objekt abzumelden. Dabei führt `_deregister` die eigentliche Arbeit aus und `close` bzw. `deregister` realisieren den rekursiven Aufruf. `close` uberprüft vorher, ob es noch Listener auf dem Modell-Objekt gibt, was analog zu einem Referenzzähler ist. Die Abfolge der Aufrufe ist durch Ziffern markiert und geschieht beim Anmelden genau andersherum. Die Struktur ermöglicht es uns, zu jedem Objekt in der OMS-Schicht genau ein Objekt in der Modell-Schicht zu instantiieren. Dies ist noch nicht implementiert, läuft aber auf eine Variation des Entwurfsmusters Singleton hinaus.

Analog geht das Anmelden mit `register`, `_register` und dem Konstruktor vonstatten. In den Unterklassen gibt es zusätzlich die Hilfsmethode `listen`, die die An- und Abmeldung bei Relationen vornimmt.

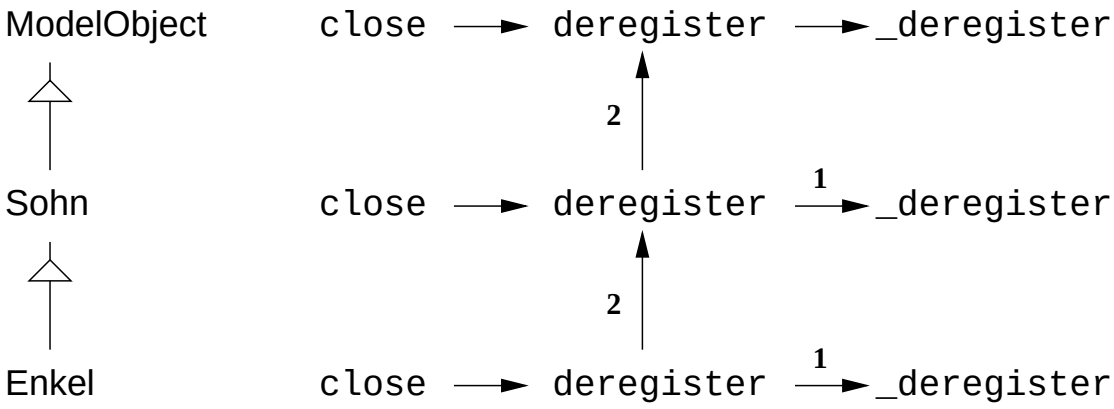


Abbildung 28.4.: Aufrufe beim Abmelden vom OMS-Objekt

Wieso ein `close`? Da die Garbage-Collection von Java Objekte nur dann entfernt, wenn es keine Referenzen auf sie mehr gibt, mußten wir eine zerstörende Methode wie

`close` einführen. Sie ist an allen Stellen aufzurufen, an denen das Modell-Objekt nicht mehr benötigt wird. Um ein verfrühtes Schließen aus der Schicht über dem Modell zu verhindern, testen wir vor dem Abmelden, ob es noch angemeldete Listener bei dem Modell-Objekt gibt.

28.3.2. ModelObject & Sohn

Dieser Teil beschreibt das Verhalten von `ModelObject` als abstrakte Oberklasse mit `ModelDoDLClass` und `ModelDoDLMethod` als Unterklassen.

28.3.2.1. Erschaffen und Löschen von `ModelObject`-Instanzen

Sicherlich ist jedem klar, daß man, um ein Objekt zu erzeugen, einen Konstruktor aufrufen muß, der einem dann die gewünschte Instanz liefert. Bei ModelObject & Sohn(`MDC`, `MDM`) gibt es nur einen Konstruktor, und der braucht genau zwei Parameter, nämlich das OMSObject (in der Ausprägung `OMSDoDLClass` oder `OMSDoDLMethod`), welches das Datenbank-Pendant bildet, und ein `OMSProject`. Dieses `OMSProject` ist zum Neuanlegen von `OMSObject`en nötig, was dem Erzeugen neuer Klassen, also auch Typen, und Methoden gleichkommt.

Auch, wenn die Garbage-Collection von Java viel Arbeit abnimmt, bleibt einiges nicht erspart. So das entgültige Abmelden bei den Listenern, wenn das Modell-Objekt zerstört werden soll. Da nämlich eben dieses Objekt bei seinen Listenern noch als Referenz gehalten wird, wird es, solange es die OMS-Objekte noch gibt, nicht von der Garbage-Collection gelöscht werden. Also sollte das Modell-Objekt sich an seinem "Lebensende" tunlichst bei allen beobachteten Ressourcen als Listener abmelden. Das geschieht mit der Methode `close`. Diese ruft das `lokale` deregister auf und zerstört noch private Instanzen. Der Aufruf wird an die Oberklasse weitergegeben, wie in Bild 28.4 auf der vorherigen Seite.

28.3.2.2. Verwaltung der Modell-Listener

Da ja Modell-Objekte nicht nur Listener, sondern auch Notifizierer (Instanzen, bei denen man sich als Listener anmeldet und von denen man Änderungen mitgeteilt bekommt) sind, wollen die bei uns angemeldeten Listener auch verwaltet werden. Die verschiedenen Modell-Objekte bieten folgende Ressourcen zum Überwachen an (das ist die Notation, in der die darauf aufbauenden Methoden benamt sind):

- `ModelObject`
  - `Name` Name des Objekts(Methode, Klasse)
  - `Version` Version des Objekts
  - `Documentation` Dokumentation
  - `Locker` Derjenige Benutzer, der dieses Objekt gesperrt hat
  - `Deprecated` zur Benachrichtigung, wenn ein Objekt veraltet ist, also gelöscht oder durch eine neuere Version abgelöst

- `ModelDoDLMethod`
  - `Parameters` Parameter einer Methode
  - `ReturnType` Rückgabetyp (einschließlich Kardinaltiät) einer Methode
  - `Body` Methodenrumpf
- `ModelDoDLClass`
  - `Aggregations` Aggregationen einer Klasse
  - `Methods` Methoden
  - `InnerClasses` innere Klassen
  - `SuperClass` Super-Klasse
  - `VisibleClasses` sichtbare Klassen, also alle bekannten Klassen, von der neue Instanzen angelegt werden können

Zu allen diesen überwachbaren Ressourcen gibt es An- und Abmeldemethoden, die `add<Ressource>Listener` bzw. `remove<Ressource>Listener` heißen. Mit deren Hilfe meldet man sich bei dem Modell-Objekt an/ab. Intern werden die ganzen angemeldeten Listener in `HashSet`s mit Namen `<Ressource>Listeners` gespeichert. Falls eine Änderung an einer Resource auftritt, was durch horchen an OMS-Objekten bekannt geworden sein dürfte, wird ganz nach Bean-Konzept [Eng97] die Methode `fire<Ressource>Changed` aufgerufen, die dann alle Listener über die Veränderung in Kenntnis setzt. Natürlich müssen vor dem Aufruf der `fire`-Methoden die Daten noch so angepaßt werden, daß die GUI sie auch versteht - das ist ja eine der Hauptaufgaben der Modellschicht.

28.3.2.3. Attribute verändern und PropertyChange

Wie bereits erwähnt, melden sich die spezielleren Modell-Objekte (`ModelDoDLClass`, `ModelDoDLMethod`) während ihrer Initialisierungsphase beim Datenbankpendant und bei diversen weiteren OMS-Objekten an (siehe dazu auch Kapitel 28.1 auf Seite 285. Dadurch werden die Modell-Objekte zu Listenern dieser OMS-Objekte und künftig von Veränderungen an diesen überwachten Ressourcen in Kenntnis gesetzt.

Lese-Methoden Um auf die Werte von Modell-Objekten zugreifen zu können oder um ihre Zustände abzufragen, wurden folgende Methoden implementiert:

- `ModelObject`
  - `getCreationDate` liest das Erzeugungsdatum des `ModelObject` aus
  - `getDBObject` gibt das Datenbank-Pendant zurück
  - `getDBProject` liefert das `OMSProject`, zu dem das Datenbank-Pendant gehört
  - `getDocumentation` holt die Dokumentation

  - `getLocker` gibt denjenigen aus, der das Datenbank-Pendant gesperrt hat
  - `getName` liefert den Namen des `ModelObject`
  - `getVersionNumber` zum Auslesen der Versionsnummer
  - `getNumberOfListeners` liest die Anzahl der Listener aus, die noch bei diesem `ModelObject` angemeldet sind
  - `getId` liefert die ID des `ModelObject`
  - `isDeprecated` beantwortet, ob dieses Objekt veraltet ist
  - `isEditable` überprüft, ob man dieses Objekt editieren kann.
  - `isDirty` dieses Flag speichert, ob das Objekt verändert wurde, damit es ggf. in die Datenbank gespeichert wird

- `ModelDoDLMethod`
  - `getBody` liest den Rumpf einer Methode aus
  - `getReturnType` gibt den Rückgabewert einer Methode zurück
  - `getParameters` holt eine aufsteigende Liste aller Methoden-Parameter
  - `getContainingClass` liefert die umgebende Klasse

- `ModelDoDLClass`
  - `getSystemType` liefert einen bestimmten `OMSDoDLSystemType`
  - `getSystemTypes` liefert alle `OMSDoDLSystemType`
  - `getVisibleClasses` liest alle sichtbaren Klassen aus, d.h. alle Datentypen, die diese `ModelDoDLClass` kennt
  - `getSuperClass` gibt die Superklasse zurück
  - `getOuterClass` liefert die umgebende Klasse
  - `getMethods` zum Auslesen aller Methoden der Klasse
  - `getMethod` zum Auslesen einer bestimmten Methode der Klasse
  - `getInnerClasses` liefert alle inneren Klassen
  - `getAggregation` sucht eine bestimmte Aggregation
  - `getAggregations` gibt alle Aggregationen der Klasse zurück
  - `isInnerClass` überprüft, ob diese Klasse innere Klasse einer anderen Klasse ist

Editier-Methoden Natürlich wollen Attribute nicht nur ausgelesen, sonder auch verändert werden. Dazu sind folgende Methoden vorgesehen:

- `ModelObject`
  - `lock` zum Sperren einer Ressource

  - `setDBObject` um das Datenbank-Pendant zu setzen
  - `setDocumentation` zum Setzen der Dokumentation
  - `setName` verändern des Namens
  - `unlock` Sperren wieder freigeben

- `ModelDoDLMethod`
  - `insertParameter` fügt einen Parameter in die Parameterliste ein. Diese Methode ist zweimal mit verschiedenen Parametern vorhanden
  - `removeParameter` löscht einen Parameter aus der Parameterliste einer *DoDL*-Methode
  - `replaceParameter` ersetzt einen Parameter
  - `setBody` setzt den Funktionsrumpf
  - `setReturnType` legt den Rückgabetyp mitsamt Kardinalität fest

- `ModelDoDLClass`
  - `insertAggregation` fügt eine neue Aggregationsbeziehung ein. Diese Methode ist zweimal vorhanden, jeweils mit einer anderen Parameterliste
  - `insertMethod` legt eine neue Methode an. Auch diese Methode gibt es zweimal mit verschiedenen Parametern
  - `removeSuperClass` löscht die Generalisierungsbeziehung
  - `replaceMethod` ersetzt eine Methode
  - `removeMethod` löscht eine Methode
  - `removeInnerClass` entfernt eine innere Klasse
  - `removeAggregation` löscht eine Aggregationsbeziehung
  - `replaceAggregation` ersetzt eine Aggregationsbeziehung
  - `setSuperClass` setzt die Generalisierungsbeziehung

`propertyChange` Wie schon öfter gesagt, werden die beobachtenden Modell-Objekte von den beobachteten OMS-Objekten notifiziert, falls eine Veränderung an ihnen auftritt. OMS-Objekte ändern sich durch die oben beschriebenen Methoden, die jedoch parallel von mehreren Benutzern ausgeführt werden können. Daher geschehen diese Veränderungen nach folgendem Schema: ein Modell-Objekt benutzt eine editierende Methode, wobei sich die Werte im OMS-Objekt ändern. Diese Editier-Methoden haben aber keinen Rückgabewert, und die Effekte solcher Veränderungen müssen deswegen auf anderem Wege das Modell-Objekt wieder erreichen, und gleichzeitig auch alle anderen Modell-Objekte, die sich für Veränderungen am entsprechenden OMS-Objekt interessieren. Dazu gibt es die Methode `propertyChange` [Eng97]). Diese Methode muß von jeder Klasse implementiert werden, die sich jemals irgendwo als Listener anmelden will, und wird vom

Notifizierer aufgerufen, um die neusten Veränderungen zu propagieren. Da die Modell-Objekte sowohl Listener bei der OMS-Schicht als auch Notifizierer für die GUI-Schicht sind, verfügen sie über relativ mächtige `propertyChange`-Methoden. In diesen Methoden laufen sämtliche von allen beobachteten OMS-Objekten gelieferten Veränderungen zusammen, müssen differenziert werden auf die überwachbaren Modell-Objekt-Ressoucen und geschickt verpackt an die GUI-Schicht weitergeleitet werden, genauer gesagt an die GUI-Adapter. Da eine Menge verschiedener `propertyChangeEvent`s in diesern Methoden betrachtet und richtig umgeformt und weitergeleitet werden müssen, wurden die `propertyChange`-Methoden von `ModelDoDLClass` und `ModelDoDLMethod` relativ umfassend.

28.3.2.4. Hilfs-Methoden und -Klassen

Es gibt eine Reihe solcher Methoden und Klassen, die einfach die an verschiedenen Stellen gemachte Arbeit bündeln. Die folgende Tabelle zeigt eine Übersicht inklusive kurzer Aufgabendarstellung.

- `ModelObject`
  - hier muss noch was kommen
- `ModelDoDLClass`
  - Klassen
    - `RelationEqualizer`
      - Methoden
      - `equals` Vergleichen zweier `OMSRelation`en nach unseren Gleichheitskriterien
      - `getValue` Auslesen der gespeicherten `OMSRelation`

      Es gab den Fall, daß alle eine Klasse betreffenden OMS-Relationen in einen `TreeMap` sortiert eingefügt werden sollten. Da aber die `equals`-Methode der `OMSRelation` nicht die für uns relavanten Kriterien verglich, haben wir kurzerhand eine eigene geschrieben. Diese wurde in der lokalen Klasse `RelationEqualizer` implementiert. Wenn jetzt eine `OMSRelation` in unseren sortierten `TreeMap` eingefügt werden soll, wird sie einem `RelationEqualizer` im Konstruktor übergeben. Dieser speichert zwar die `OMSRelation`, vergleicht sie aber nicht mit der `equals`-Methode von `OMSRelation`, sondern mit der eigenen, die genau unser Maß an die Gleichheit von Realtionen legt. Also werden eigentlich `RelationEqualizer`-Instanzen in den `TreeMap` sortiert, die alle wiederum eine Instanz von `OMSRealtion` enthalten. `RelationEqualizer` ist eine sogenannte `Wrapper-Klasse`, da sie eine andere Klasse (hier: `OMSRealtion`) umgibt. Damit man beim sortierten auslesen aus dem `TreeMap` auch wieder an die eigentlichen `OMSRealtion`en kommt, gibt es in `RelationEqualizer` noch die Methode `getValue`.

- Methoden
  - ▷ `AggregationsToTreeMap` um alle Aggregationen einer Klasse sortiert in ein `TreeMap` einzufügen, wurde diese Methode implementiert. Man übergibt eine Liste von `OMSAggregation`en und erhält eine `TreeMap` zurück. Falls Aggregationen doppelt vorkommen, wird eine `Exception` geworfen.
  - ▷ `TreeMapToAggregations` Umkehrmethode zu `AggregationsToTreeMap`. Durchläuft eine `TreeMap` und erstellt eine Liste von `OMSAggregation`en.
  - ▷ `RelationsToVector` hiermit werden alle übergebenen `OMSRelation`en in einen Vektor kopiert.
  - ▷ `VectorToRelations` macht aus einem Relationsvektor wieder eine Relationsliste.
  - ▷ `RelV2DestV` heißt ausgeschreiben `RelationVektor2DestinationVektor`. Da in einer `OMSRelation` ein Start- und ein Zielpunkt vorkommen, oberhalb der Modellschicht aber nur noch die Zielpunkte von Interesse sind, wurde diese Methode implementiert, um aus einer Liste von `OMSRelation`en die Zielklassen zu extrahieren und daraus eine Liste zusammenzustellen und zurückzugeben.
  - ▷ `getRelationWithDest` bekommt als Parameter eine Liste von `OMSRelation`en und ein `OMSObject` als Zielobjekt. Es werden aus der Liste der Relationen diejenigen herausgesucht, die als Ziel das übergebene Ziel haben, und die Liste der Herausgesuchten wird zurückgegeben.
  - ▷ `getSuperClassOfClass` gibt die Superklasse einer Klasse zurück.
  - ▷ `getReturnTypeOfMethod` gibt den Rückgabetypen einer Methode zurück.
  - ▷ `listen` bekommt eine Liste von `OMSObject`en übergeben, bei denen sich als Listener angemeldet wird.
  - ▷ `listenToMethods` da `OMSDoDLMethod`en keine `OMSObject`e sind, mußte für das Anmelden als Listener bei einer Liste von Methoden eine eigene Methode her.

- `ModelDoDLMethod`
  - Methoden
    - ▷ `RelationsToTreeMap` ähnlich `RelationsToTreeMap` von `ModelDoDLClass`, nur daß hier `OMSDoDLParameterRelation`en in die `TreeMap` eingefügt werden. Die Überprüfung, ob Relatonen doppelt vorkommen, wird hier am Namen und an der Position des Parameters festgemacht(beides Attribute von `OMSDoDLParameterRelation`).
    - ▷ `TreeMapToRelations` Umkehrung zu `RelationsToTreeMap` - macht aus einer `TreeMap` eine Liste von `OMSDoDLParameterRelation`en.
    - ▷ `checkParameterRelations` überprüft, ob eine übergebene Liste von `OMSDoDLParameterRelation`en bei Positionsnummer 0 beginnt und fortlaufend steigend ist.

- ▷ `getReturnTypeOfMethod` ließt den Rückgabetypen einer `OMSDoDLMethod` aus.
- ▷ `getNoNameType` liefert einen `OMSDoDLSystemType` (vgl. Abschnitt 27.4.3 auf Seite 254), und zwar denjenigen, der `noname` heißt und mit einem leeren String dargestellt wird. Dieser noname-Typ wird verwendet, wenn man ein neues Attribut oder eine neue Methode anlegt. Der Standardtyp, der dem Neuen zugewiesen wird, ist eben dieser noname-Typ, und er wird mit einem Leerstring dargestellt.
- ▷ `listen` gleiche Funktionalität wie die Methode `listen` von `ModelDoDLClass`.

28.3.3. Modell-Datencontainer

Wie schon weiter oben erwähnt, gibt es in der Modell-Schicht Datenstrukturen, die gezielt in der GUI darzustellende Werte bündeln und als eins greifbar und transportierbar machen. Dadurch wird der Umgang mit den Relationen leichter, die es zwar auf OMS-Seite noch gibt, aber auf der GUI-Seite nicht mehr benötigt werden. Darstellen will man die Relationen natürlich schon auf der Oberfläche.

Es gibt von diesen Transport-Datentypen genau zwei, nämlich den `ModelDoDLType`(`MDT`) und die `ModelDoDLVariable`(`MDV`). Der `MDT` ist die Superklasse von `MDV` und speichert die Zielklasse einer Relation, die bereits in einer `ModelDoDLClass` verpackt ist (die Zielklasse einer `OMSRelation` ist ein `OMSObject`, und zwar eine `OMSDoDLClass`. Da die GUI mit OMS-Objekten nichts anzufangen weiß, muß man sie gescheit verpacken, und das passiert mit der `MDC`), und die zugehörige Kardinalität. Das reicht für einfache Typenbeschreibungen wie bei Rückgabewerten von Methoden, Relationen zu inneren Klassen oder Vererbungsrelationen (wobei hier schon die Kardinalität überflüssig ist) aus. Die `MDV` bietet auch diese Speicherungsmöglichkeiten, erweitert sie jedoch noch um einen Namen. Dieser ist wichtig für Parameter und Aggregationen.

Diese Datencontainer besitzen nur die folgenden `get`-Methoden.

- `ModelDoDLType` und `ModelDoDLVariable`
  - `getCardinality` liest die Kardinalität des Datentyps aus
  - `getType` gibt den Datentyp zurück
- `ModelDoDLVariable`
  - `getPosition` liefert die Position dieses Datentyps in einer Parameterliste
  - `getName` gibt den Namen der Variable zurück, die vom Typ dieses Datentyps ist

Daher müssen sie ihre Inhalte schon im Konstruktor bekommen. Sie sollen nur als kurzzeite Datenspeicher benutzt werden, deren Lebenszeit sehr begrenzt ist, unter anderem auch, weil sie Werte zwischenspeichern (cache-n), die sich bei längeren Laufzeiten durchaus verändern könnten, was diese Container nicht mitbekämen.

Wie schon beschrieben, müssen `OMSDoDLClass`-Instanzen in `ModelDoDLClass`-Instanzen verpackt werden. Da aber bei einer solchen Verpackung die `ModelDoDLClass` sofort beginnt, sich bei allen Relationen der `OMSDoDLClass` als Listener anzumelden, kann eine sehr große und speicherfressende Datenstruktur entstehen. Es ist sehr darauf zu achten, daß diese zur temporären Datenspeicherung angelegten Objekte auch wieder geschlossen werden, um nicht unnötig den Speicherverbrauch in die Höhe zu treiben. Es wurde auch überlegt, das oben beschriebene Anmelden an allen interessanten Ressourcen durch einen separaten Methodenaufruf im `Modelobject` zu bewerkstelligen, so daß es `Modelobjecte` geben könnte, die auch nur als reine Datenspeicher fungierten, aber das wurde wieder verworfen. Die Funktionalität dazu ist vorhanden, man müßte nur den Aufruf von `register` aus den `Modelobject`-Konstruktoren entfernen und eben bei Interesse gezielt von der GUI aus aufrufen.

28.3.4. Und da waren noch ...

Es gibt noch weitere Klassen der Modellschicht, die zwar entworfen, aber aus Zeitgründen nicht mehr implementiert wurden. Dazu zählen alle Klassen der dritten Modellschicht-Schicht, also der Klassendiagrammschicht. Namentlich sind das folgende Klassen:

1. `ModelClass`
2. `ModelClassDiagramm`
3. `ModelRelation`
4. `ModelGeneralization`
5. `ModelAggregation`

28.4. Kritik der Modellschicht

Autor: *Sascha Lüdecke*

In diesem Abschnitt wollen wir die Modellschicht kritisch betrachten. Dies geht nicht, ohne auch auf die umgebenden Schichten Bezug zu nehmen, da die Modellschicht letztendlich durch diese determiniert ist. Das der Weg unserer Entwicklung trotz modularer Aufteilung des Entwurfs (siehe Abschnitt 25.1 auf Seite 209) ungleiche Schwerpunkte hatte, wird nicht nur durch den letzten Stand der GUI, sondern auch durch die Geschichte der Modellschicht deutlich.

Im Laufe der Erarbeitung der Klassenhierarchie haben wir uns stark auf die Schicht OMS (siehe Abschnitt 27 auf Seite 239) konzentriert, bei der es um eine werkzeug- und auch sprachunabhängige, vor allem aber persistenzorientierte Darstellung der Daten ging. Erst die Modellschicht war als genaues Abbild unserer Zielsprache *DoDL* vorgesehen.

Trotz dieses Schwerpunktes gelang es uns nicht, frühzeitig die Schnittstelle zwischen Modell- und OMS-Schicht festzulegen, so daß die Modellschicht ständig an das sich ändernde Interface angepaßt werden mußte. Davon waren auch die Dummyimplementierungen letzterer betroffen, die Grundlage für Funktionstests der Modellschicht waren. Es ist

fraglich, ob wir uns zuerst ausschließlich auf das Interface hätten konzentrieren können, ein Anforderungskatalog und weniger häufige Änderungen jedoch hätten mehr Raum für die Modellschicht und die Umsetzung des Bruchs (Damit ist die unterschiedliche Modellierung aufgrund verschiedener Zielsetzungen der Daten in den Schichten OMS und Modell gemeint, die eine nicht 1:1-Abbildung erzwang. Dies wird auch in den Abschnitten 25.2.1 auf Seite 210, 25.2.2 auf Seite 210 und 25.2.3 auf Seite 211 deutlich) gelassen. Diese erwies sich als schwieriger als gedacht, da die dynamische Struktur der Schichten hier eine wesentliche Rolle spielt.

Gerade diese Struktur, verkörpert durch den Notifikationsmechanismus, wurde von uns auf allen Ebenen vernachlässigt. Wir haben zwar viele Ereignisse und Nachrichten definiert, sind dabei aber nicht strukturiert vorgegangen und haben sie an keiner Stelle öffentlich dokumentiert. Dementsprechend ist deren Verarbeitung bis zum Ende der Implementierung Stückwerk geblieben und läßt sich wohl als das Haupthindernis einer flüssigen Integration ausmachen. Grund für diese Annahme ist die recht gute Arbeitsweise der Einzelteile mit Dummies und leeren Implementierungen.

Ein weiterer Grund für den unausgereiften Zustand der statischen Struktur ist die kaum stattgefundene Zusammenarbeit der Teams "Klassendiagrammeditor" und "Klasseneditor", was dazu führte, daß die Klassenhierarchie der Modellschicht stark auf letzteres ausgerichtet war und viel Spielraum für ein erfolgreiches Re-Engeneering bleibt (siehe auch Abschnitt 25.4.2 auf Seite 213). Ebenso haben Schwierigkeiten mit der Umsetzung einiger Ideen auf der GUI-Ebene Arbeitskraft gebunden, was sich durch genauere Vorüberlegungen in der Gruppe eventuell hätte vermeiden lassen.

Abschließend lassen sich drei Problemfelder ausmachen:

- spät konvergierende Interfaces,
- vernachlässigte Dynamik,
- zu wenig konzeptionelle Arbeit in der Modellschicht,

deren genauere Beachtung ein besseres Vorankommen der Gesamtentwicklung und damit ein stabileres System zur Folge gehabt hätten.

29. GUI - Die allgemeine Benutzungsoberfläche

In diesem Kapitel beschreiben wir den allgemeinen Teil der GUI. Dies ist zum einen der generische Editor mit seinem DFA, zum anderen die Adapter, mit deren Hilfe das Modell in eine Swing-GUI eingebettet werden kann.

29.1. Der generische Editor

Autoren: *Martin Uebing, Christoph Begall*

Die Idee des generischen Editors stammte schon aus unserem Prototypen, der Ampelstadt. In $\mathcal{HEU}$ wurde der generische Editor allerdings noch einmal überarbeitet und erweitert. Zum einen mußte der Editor von der 2-Schichten-Architektur des Prototypen an die 3-Schichten-Architektur von $\mathcal{HEU}$ angepaßt werden und somit auch Klassen der Modellschicht für den Editor identifiziert und entworfen werden. Zum anderen wollten wir den abstrakten Editor auch wirklich wiederverwendbar machen und von seinen konkreten Ausprägungen tatsächlich trennen. Dazu legten wir ein separates Package `genericeditor` an, in dem alle generischen Klassen und der DFA zusammengefaßt wurden.

Der DFA wurde in der Ampelstadt für jeden Editor komplett neu implementiert. Der Entwurf der $\mathcal{HEU}$ sah vor, die gemeinsamen Zustände und Transitionen aller Editoren in einem generischen DFA zusammenzufassen.

Um dies zu ermöglichen, wurden `NodeFactory` und `EdgeFactory` neu eingeführt. Für jeden neuen Knoten- bzw. Kantentyp kann man eine spezielle `NodeFactory` bzw. `EdgeFactory` beim Editor anmelden. Diese sorgt für die Erzeugung des konkreten Knotens bzw. der konkreten Kante.

Um zur Modelschicht generisch zu bleiben, wurden `ModelNode` und `ModelEdge` als Interfaces eingeführt, um von den konkreten *DoDL*-Klassen der Modellschicht zu abstrahieren, aber gemeinsame Funktionalität von Knoten bzw. Kanten - z.B. das Setzen und Abfragen von Positionen - allgemein verfügbar zu machen.

Die genannten Klassen und ihre Beziehungen zu den schon aus der Ampelstadt bekannten Drawable-Klassen sind in Abb. 29.1 auf der nächsten Seite dargestellt. Die `Drawables` verwenden hier wiederum die durch `JComponent` zur Verfügung gestellte Funktionalität wie Ereignisbehandlung, automatisches Neuzeichnen und einfache Möglichkeiten zur unterschiedlichen Darstellung (z.B. verschiedene Farben).

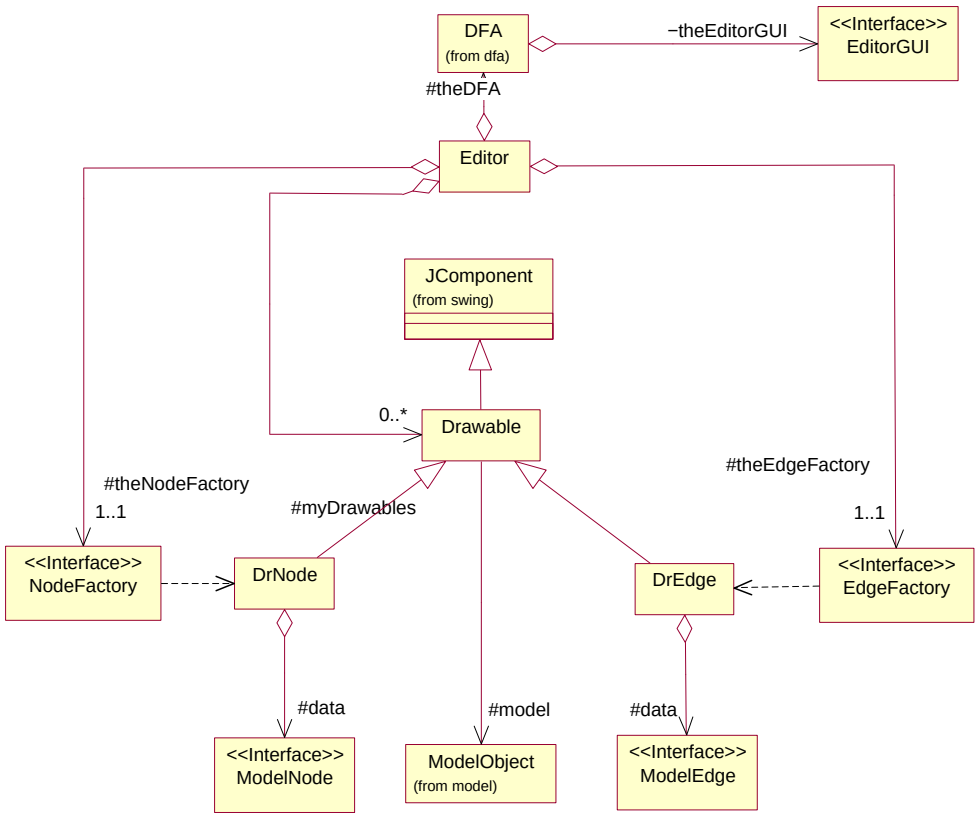


Abbildung 29.1.: Die Klassenstruktur des generischen Editors

29.2. Der endliche Automat

Der endliche Automat, der die Klassen `DFA`, `State` und `Transition` umfasst, wurde unverändert von der Ampelstadt übernommen. Es exisiert allerdings nur eine kurze, unvollständige, beispielhafte Implementierung in der Klasse `ClassDiagramEditor`. Mit "Implementierung" ist das Überschreiben der Methode `handleEvent` gemeint, die beim Ausführen einer Transition aufgerufen wird und in der die bei einer Zustandsänderung nötigen Aktionen ausgeführt werden.

Abb. 29.2 auf der nächsten Seite zeigt den Entwurf für den DFA des generischen Editors, aufgeteilt in die vier Hauptaktionen "Auswahl eines Drawables", "Erzeugen einer Kante", "Erzeugen eines Knotens" und "Verschieben eines Knotens". Um die Grafik noch überschaubar zu halten, sind nur die wichtigeren Transitionen eingezeichnet. Um dem Benutzer mehr Komfort zu geben, kann man weitere Übergänge zwischen den Zuständen hinzufügen. Transitionen, die von den Umrahmungen der Zustände ausgehen, sind von

jedem der enthaltenen Zustände erlaubt. Das abgebildete Zustandsübergangsdiagramm ist eine Verfeinerung des Diagramms aus der Ampelstadt, das im Abschnitt 17.2.2.3 auf Seite 150 beschrieben ist.

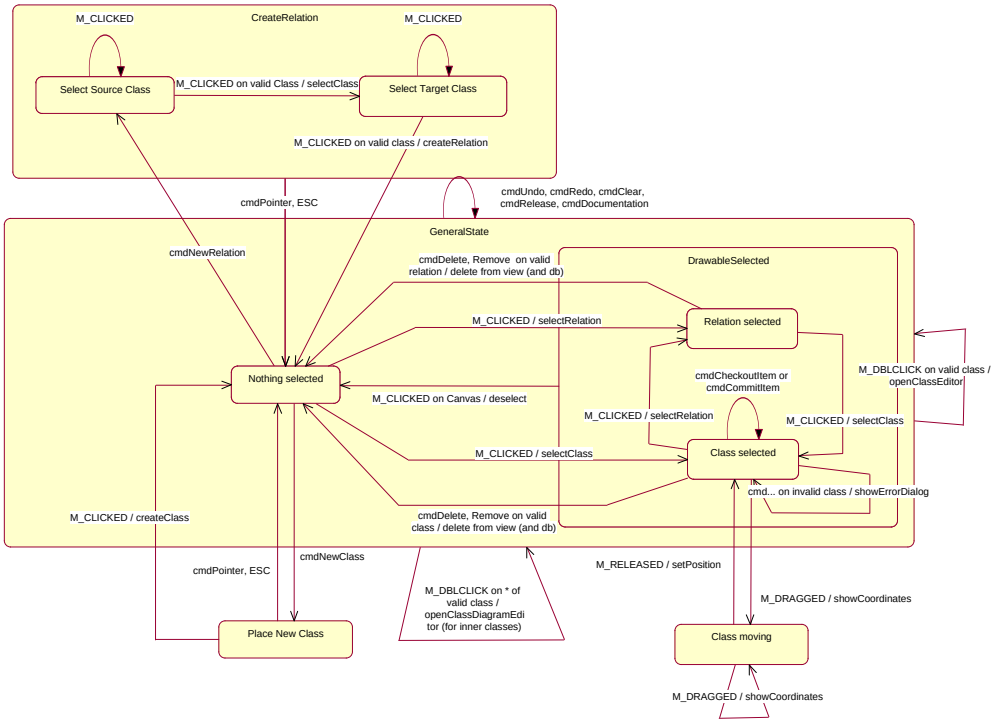


Abbildung 29.2.: Der endliche Automat des generischen Editors

29.3. Verwendung und Erweiterung des Editors

Die Verwendung des generischen Editors für den Klassendiagramm-Editor und die Beziehungen zwischen GUI- und Model-Klassen veranschaulicht Abb. 29.3 auf Seite 310. Als spezialisierte Factories treten hier `GeneralizationFactory`, `AggregationFactory` und `ClassFactory` auf. Durch sie werden die jeweiligen Drawable-Klassen erzeugt.

Mangels Zeit konnten weder der Klassendiagramm-Editor noch der generische Editor der $\mathcal{HEU}$ implementiert werden. Es existiert eine graphische Benutzungsoberfläche für den Klassendiagramm-Editor, die allerdings nur die Navigation durch Menüs und Dialoge erlaubt und keine darüberhinausgehende Funktionalität bietet. Entfernt man aus ihr die *DoDL*-spezifischen Elemente, so erhält man die Benutzungsoberfläche des generischen Editors, der sich für funktional ähnliche, spezielle Editoren wiederverwenden und nach Bedarf erweitern läßt.

Toolbar- und Menü-Elemente eines speziellen Editors, die zum Zeichnen konkreter Knoten- und Kantentypen dienen, sollten als Instanzen des Java-Interfaces `Action` realisiert werden. Dies bietet den Vorteil sie gemeinsam verwalten zu können, z.B. zu (de)aktivieren.

29.4. Die Adapter

Autor: *Jens Schröder*

Wie in Abschnitt 25.3 auf Seite 211 erwähnt, sind die Adapter eine Zwischenschicht in der Modellschicht. Sie treten bei der $\mathcal{HEU}$ an zwei Stellen auf: bei dem Klasseneditor und bei dem Projektmanager. Im folgenden werden die verschiedenen Adapter genauer vorgestellt.

29.4.1. Die Adapter des Klasseneditors

In der GUI des Klasseneditors werden eine Vielzahl von Informationen repräsentiert, die sich durch das verteilte Arbeiten durch andere Benutzer ändern können. Von diesen Änderungen des Modells werden die Views, also die GUI-Elemente, durch Notifikationen benachrichtigt. Um diese Kommunikation übersichtlich zu gestalten und nicht eine monolithische GUI-Klasse zu implementieren, haben wir uns entschieden, die Adapter als Schicht zwischen dem Model und der GUI einzusetzen (siehe Abschnitt 25.3 auf Seite 211).

Es gibt zwei Arten von Adaptern bei dem Klasseneditor: die Tabellenadapter, die zu den angezeigten Tabellen gehören und die Textadapter, die zu den als Textkomponenten dargestellten Informationen gehören.

29.4.1.1. Die Tabellenadapter

Die Tabellenadapter haben zwei Aufgaben. Sie haben nicht nur die bereits erwähnte Funktion, die Kommunikation der Modelschicht mit den Tabellen des Klasseneditors abzuwickeln, sondern sie sind auch das Modell der angezeigten Tabellen. Dies ist nötig, da wir als Tabellenklasse die Swingkomponenete `JTable` benutzen und diese nach dem MVC-Konzept arbeitet und alle Datenänderungen dem Modell und nicht der Tabelle selbst mitgeteilt werden.

Diese beiden Aufgaben spiegeln sich auch in der Struktur der Tabellenadapter wieder, sie alle erweitern das `DefaultTableModel` und implementieren das `PropertyChangeListener`-Interface.

Wie in Abbildung 29.4 auf Seite 311 zu sehen ist, gliedern sich die Tabellenadapter in verchiedene Klassen, die im weiteren vorgestellt werden.

AbstractTableAdapter Der `AbstractTableAdapter` ist die Oberklasse aller Tabellenadapter. Er erweitert das `DefaultTableModel` und implementiert das `PropertyChangeListener`-Interface. Er stellt Grundfunktionalität zu Verfügung, die von allen Tabellenadapter benötigt wird und muß noch spezialisiert werden.

Der `AbstractTableAdapter` bietet die Möglichkeit an, auf Zeilen über die in den Zeilen repräsentierten Modellobjekte zuzugreifen. Die Tabellenmodelle von Swing erlauben einen Zugriff auf die Zeilen einer Tabelle nur über die Zeilennummer. Da bei Notifikationen aus der Modellschicht aber Modellobjekte übergeben werden, hält sich der `AbstractTableAdapter` eine Datenstruktur (`LinkedList`), die eine Zuordnung der Modellobjekte zu den Zeilennummern, in denen die Modellobjekte angezeigt werden, bereitstellt. Der Index eines Modellobjekts in der `LinkedList` entspricht der Zeilennummer. Mittels der Methoden `addElementToModel` und `deleteElementFromModel` werden die Modellobjekte der Tabelle hinzugefügt bzw. entfernt.

Die `PropertyChange`-Methode aus dem `PropertyChangeListener`-Interface, die bei einer Notifikation aus der Modellschicht aufgerufen wird, ist hier nur abstrakt deklariert, damit die Spezialisierungen diese dann geeignet überschreiben.

Weiterhin soll der `AbstractTableAdapter` die Möglichkeit bieten, eine Spalte auf eine angemessene Breite zu skalieren. Dies soll die Methode `fitColumnSize` ausführen. Sie ist implementiert, funktioniert aber aus unbekannten Gründen nicht.

GenericVariableTableAdapter Der `GenericVariableTableAdapter` ist ein generischer Tabellenadapter, der Modellobjekte vom Typ `ModelDoDLVariable` repräsentiert. Es handelt sich also um ein Modell einer 3-spaltigen Tabelle, die den Namen des Objekts, den Typ und die Kardinalität des Typs anzeigt.

Der Typ und die Kardinalität sollen in `JComboBoxen` dargestellt werden, so daß man als möglichen Typ nur aus den nach den Sichtbarkeitsregeln von *DoDL* bekannten Klassen auswählen und bei der Kardinaliät nur zwischen *1* und *n* wählen kann. Die Methoden dazu sind bereits implementiert, aber die `JComboBoxen` werden aus unbekannten Gründen nicht angezeigt. Beim jetzigen Stand der Implementation werden der Typ und die Kardinalität als `String` repräsentiert.

Weiterhin wird ein Zugriff auf das Modellobjekt des Typs über die Zeilennummer ermöglicht. Hierzu wird wieder eine extra Datenstruktur (wie bei dem `AbstractTableAdapter` eine `LinkedList`) verwaltet, die mittels geeigneter Methoden manipuliert werden kann. Diese Funtionalität wird von der GUI benötigt, wenn ein Benutzer im Klasseneditor eine Änderung vornimmt und diese Änderung durch die Modellschicht in die Datenbank geschrieben werden soll.

Spezialisierungen Die drei Spezialisierungen des `GenericVariableTableAdapters` sind der `AggregationsTableAdapter` (das Modell für eine Tabelle, die die Aggregationen einer *DoDL*-Klasse anzeigt), der `MethodsTableAdapter` (das Modell für eine Tabelle, die die Methoden einer *DoDL*-Klasse anzeigt) und der `ParametersTableAdapter` (das Modell für eine Tabelle, die die Parameter einer *DoDL*-Methode anzeigt). Sie melden sich bei den geeigneten Modellobjekten als Listener an und implementieren entsprechend die `PropertyChange`-Methode. In der `PropertyChange`-Methode werden die verschiedenen Arten von `PropertyChangeEvents` unterschieden und durch eine Auswertung der Belegung der Attribute `oldValue` und `newValue` des `PropertyChangeEvent`-Objekts erkannt, ob ein Objekt gelöscht, verändert oder hinzugefügt werden muß.

Die Spezialisierungen sind implementiert und mit den `DUMMY`-Implementierungen der `OMS`-Interfaces erfolgreich getestet. Bei Tests mit der `HPCTE`-Implementierungen der `OMS`-Interfaces funktionierte zum Teil das Löschen von Methoden und Parametern nicht. Die Ursache muß nicht unbedingt in den Adaptern liegen, aber genaueres läßt sich leider nicht sagen.

ClassesTableAdapter Der `ClassesTableAdapter` spezialisiert direkt den `AbstractTableAdapter`. Er ist das Modell für eine Tabelle, die die inneren Klassen einer *DoDL*-Klasse anzeigt. Sie ist einspaltig und darf vom Benutzer nicht verändert werden, sondern ändert sich nur durch Notifikationen aus der Modelschicht, die durch Änderungen im Klassendiagrammeditor ausgelöst werden können.

Er ist vollständig implementiert und funktioniert.

29.4.1.2. Die Textadapter

Die Textadapter kapseln die Kommunikation der Modelschicht mit der GUI aus der GUI-Klasse. Sie übernehmen die Listeneraufgaben für Textelemente wie Namens-, Superklassen-, Versionnummernfeld, Dokumentation und Methodenbody. Wie man in Abbilung 29.5 auf Seite 312 sieht, ist die Hierarchie der Textadapter einfach. Es gibt die Oberklasse `AbstractTextAdapter`, die das `PropertyChangeListener`-Interface implementiert. Die `PropertyChange`-Methode wird hier schon vollstandig implementiert, da sie für alle Textadapter gleich ist, nämlich die entsprechende Textkomponente leeren und sie mit dem im `PropertyChangeEvent` übergebenen String neu füllen.

Die Spezialisierungen (`MethodBodyTextAdapter`, `NameTextAdapter`, `SuperClassTextAdapter`, `DocumentationTextAdapter`, `VersionTextAdapter`) sind nur nötig, um sich jeweils bei der richtigen Modellklasse als Listener für die richtige Eigenschaft anzumelden und die initialen Werte auszulesen.

Die Textadapter sind implementiert, getestet und funktionieren.

29.4.2. Die Adapter des Projektmanagers

Autor: *Martin Uebing*

Der Projektmanager verwendet einen `JTree`, in dem Projekte, Klassen und Methoden - später auch Klassendiagramme - hierarchisch angezeigt werden. Projekte stellen die Wurzelobjekte dar. Sie enthalten Klassen oder Klassendiagramme (noch nicht erzeugbar). Unterhalb von Klassen können wiederum Methoden liegen.

Wie die meisten Swing-Elemente arbeitet auch der `JTree` nach dem MVC-Konzept. Das Model wird dem `JTree` im Konstruktor übergeben oder kann über die Methode `setModel` gesetzt werden.

Die Elemente eines Treemodels sind TreeNodes - hier Instanzen der Klasse `DefaultMutableTreeNode`. Diese erhalten bei der Instanziierung ein `UserObject` vom Typ `Object` übergeben: das durch den Knoten repräsentierte Objekt. Der Texteintrag, den der Benutzer später im Baum sieht, wird einfach durch Aufruf der `toString`-Methode von diesem `UserObject` ermittelt.

Damit hier der Name der konkreten Ausprägung von `OMSObject` (`OMSProject`, `OMSDoDLClass`, `OMSDoDLMethod` oder `OMSDoDLClassDiagram`) angezeigt wird, benutzt der Projektmanager eine interne Wrapper- bzw. Adapterklasse `NodeItem` mit den Attributen `nodeName` und `userObject` und den Methoden `toString()` und `getUserObject()`. `toString()` liefert dem `JTree` den Objektnamen als anzuzeigenden Text, über `getUserObject()` kommt man wieder an das `OMSObject` aus dem Knoten des Baums.

Diese Klasse `NodeItem` wird dann als Kapsel für alle im `JTree` darzustellenden `OMSObjects` verwendet und stellt das `UserObject` dar, das bei der Instanziierung eines `DefaultMutableTreeNodes` übergeben wird.

Aus Zeitgründen wurde für den Projectstree nur eine sehr einfache und ineffiziente Art der Notifizierung implementiert. Der Projektmanager selbst ist als Listener bei der `RootSession`, bei Projekten und bei Klassen angemeldet. Wann immer Änderungen auftreten, wird von der `RootSession` die gesamte Struktur neu ausgelesen und der Projectstree neu aufgebaut.

Eine bessere Implementierung würde das Modell im MVC-Konzept, also die Adapterklassen selbst, notifizieren, so daß nur einzelne Objekte im Baum aktualisiert würden, neue Childobjekte dynamisch hinzugefügt und gelöschte Childobjekte dynamisch entfernt würden.

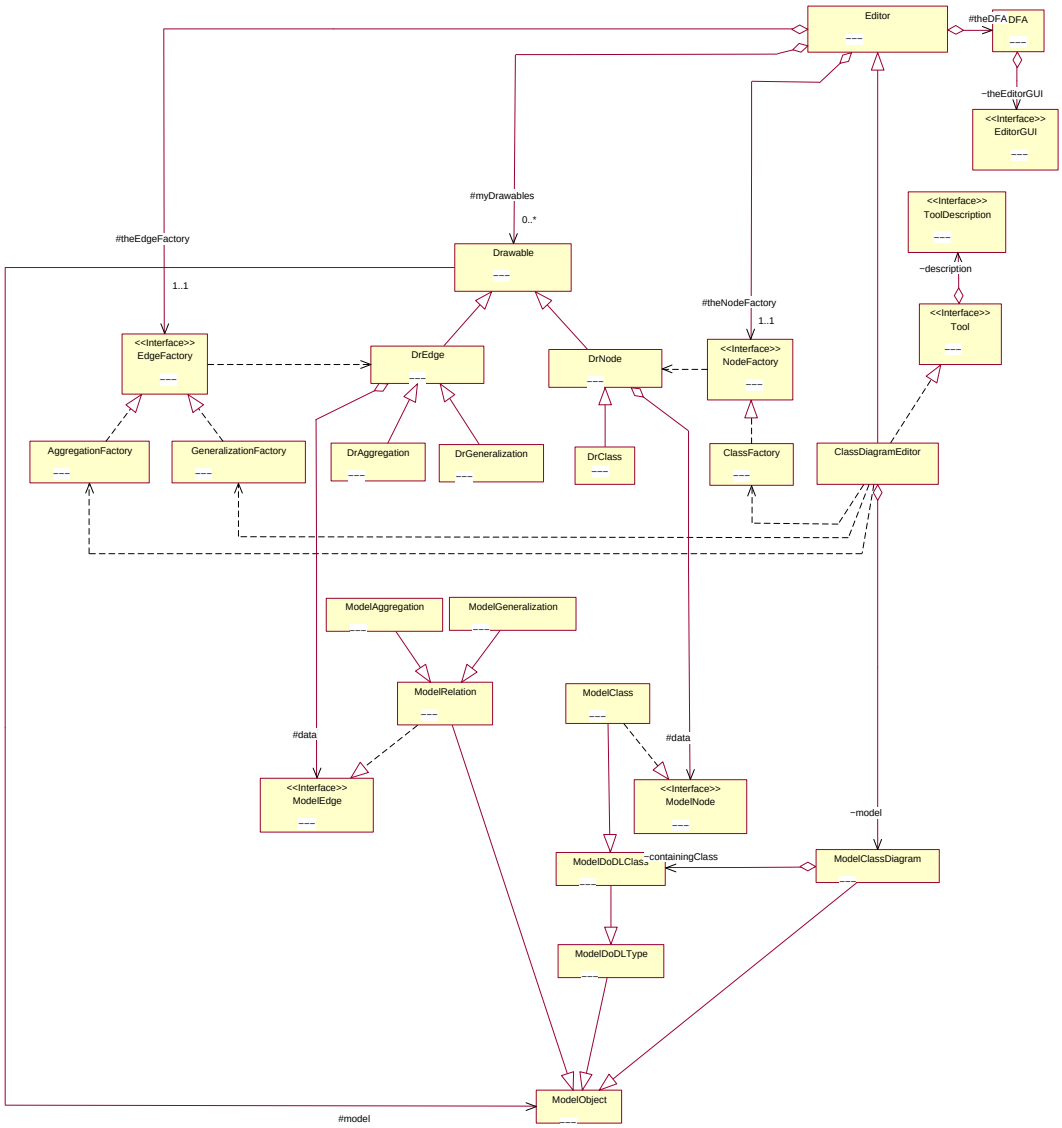


Abbildung 29.3.: Übersicht über die Klassenhierarchie des Klassendiagramm-Editors

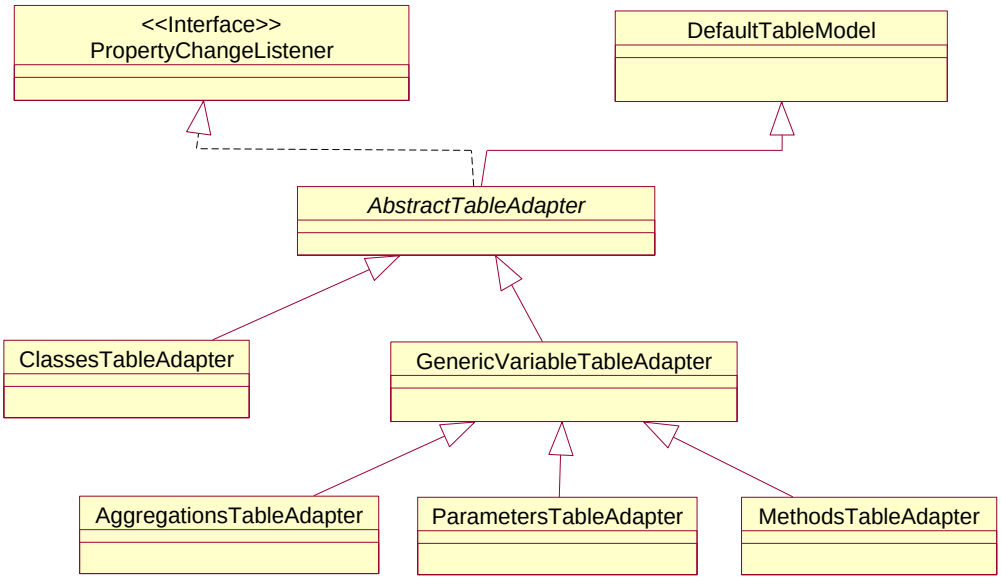


Abbildung 29.4.: Die Hierachie der Tabellenadapter

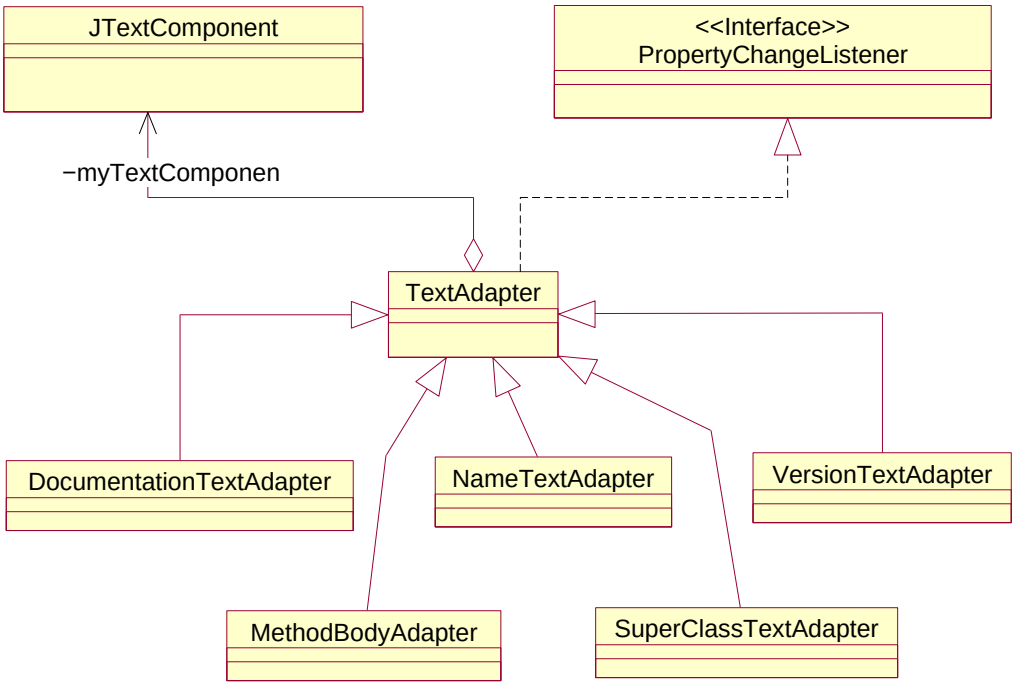


Abbildung 29.5.: Die Hierachie der Textadapter

Teil VIII.

Die Anwendungen der $\mathcal{HEU}$

30. Der Projektmanager

Autor: *Matthias Dorka*

Dem Tool "Project Manager" fällt im Rahmen dieses Systems eine besondere Rolle zu, da es als Eintrittswerkzeug in die Hypermedia-Entwicklungsumgebung dient. Es ist das Werkzeug, das vom Benutzer initial zu starten ist, um die $\mathcal{HEU}$ zu verwenden. Damit ist der Projektmanager für eine Reihe von Aktivitäten zuständig, die ihn von den anderen Werkzeugen unterscheidet. Dazu zählt insbesondere der Aufbau der Verbindung zur Datenbank einschließlich Anmelden des Benutzers sowie das Starten der eigentlichen Editierungswerkzeuge Klassendiagrammeditor, Klasseneditor und Methodeneditor.

Im Rahmen der Entwicklung des Projekmanagers wurde versucht, dieses Tool so allgemein wie möglich zu halten, um einerseits weitere, andersartige Editoren ohne großen Änderungsaufwand in die $\mathcal{HEU}$ einfügen zu können, andererseits aber auch Unterstützung für weitere Projektarten außerhalb des *DoDL*-Kontextes zu bieten.

30.1. Funktionsweise und Bedienungsanleitung

30.1.1. Starten des Projektmanagers

Der Start des Projektmanagers gestaltet sich zum Zeitpunkt des Entwicklungsendes der $\mathcal{HEU}$ als nicht ganz einfache Prozedur. Der Benutzer hat zunächst einmal dafür zu sorgen, daß ein Datenbankserver unter seinem Benutzerkonto läuft. Dies geschieht duch Eingabe des Kommandos `jhpcte -s`, sofern die Datenbank bereits installiert und das Datenbankschema `heu.sds` kompiliert ist. Das Starten des JHPcte-Servers sorgt dabei automatisch für das Starten des zugrundeliegenden H-PCTE-Servers.

Anschließend ist die Datenbank mit den *DoDL*-Systemtypen zu initialisieren, wozu das Skript `run_JHPcteDoDLSystemTypesInit` im Verzeichnis `/heu/system/oms/hpcte` existiert. Schließlich wird der Projektmanager mittels `run_ProjectManager` in `/heu-/system/projectmanager` gestartet. Es erscheint das Anmeldefenster der Datenbank wie in Abb. 30.1 auf der nächsten Seite gezeigt, in dem Benutzername und Paßwort eingegeben werden. Optional kann das Verzeichnis der laufenden Datenbank geändert werden, falls der aus der H-PCTE-Umgebungsvariable ausgelesene Wert nicht übernommen werden soll.

Damit wird der Projektmanager gestartet, eine initiale Datenbankverbindung aufgebaut und eventuell bereits vorhandene Projekte werden im Hauptfenster (s. Abb. 30.2) in Form eines Navigationsbaumes angezeigt.

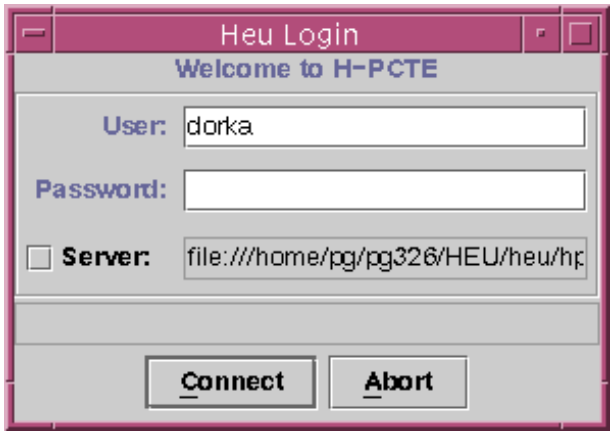


Abbildung 30.1.: Das Anmeldefenster der Datenbank

30.1.2. Anlegen neuer Projekte

Das Anlegen eines neuen Projektes geschieht über den Menüpunkt *File/New/Project*. Es erscheint ein Dialogfenster, in dem ein Name für das neue Projekt vergeben werden muß. Dieses wird daraufhin am linken Rand der Anzeigefläche dargestellt, um zu symbolisieren, daß es sich hierbei um ein Containerobjekt höchsten Ranges handelt, das gleichberechtigt neben anderen Projekten steht, aber Objekte wie Klassendiagramme, Klassen und Methoden enthalten kann. An dieser Stelle sei bereits darauf hingewiesen, daß leere Projekte und Klassen in dieser Baumansicht durch das Java-Standardicon eines leeren Blattes visualisiert werden, während Objekte, die mindestens ein weiteres Objekt enthalten, als Ordner erscheinen. Es gehört zur nicht implementierten Funktionalität, daß diese Symbole durch geeignetere ersetzt werden sollten. Insbesondere sollten Symbole gewählt werden, die das repräsentierte Objekt mnemonisch darstellen, also beispielsweise ikonifizierte Klassendiagramme oder Klassensymbole der *DoDL*-Notation.

30.1.3. Anlegen neuer Elemente

Das Anlegen anderer neuer Elemente geschieht auf analoge Weise. Das Menü *File/New* enthält neben *Project* auch die Punkte *Class Diagram*, *Class* und *Method*, welche Klassendiagramme, Klassen und Methoden innerhalb des zu diesem Zeitpunkt selektierten Objekts erzeugen. Auch hierbei erscheint zunächst jeweils ein Dialogfenster, in dem das neue Objekt benannt werden muß.

30.1.4. Starten der Editoren

Eine wesentliche Grundidee beim Entwurf der $\mathcal{HEU}$ war es, verschiedenartige Werkzeuge zu unterstützen und der Erweiterbarkeit um neue Werkzeuge dabei möglichst geringe Schranken in den Weg zu stellen. Deshalb fungiert der hier beschriebene Projektmanager mit seinen allgemein gehaltenen Projekten als Eintrittspunkt in die $\mathcal{HEU}$ und somit

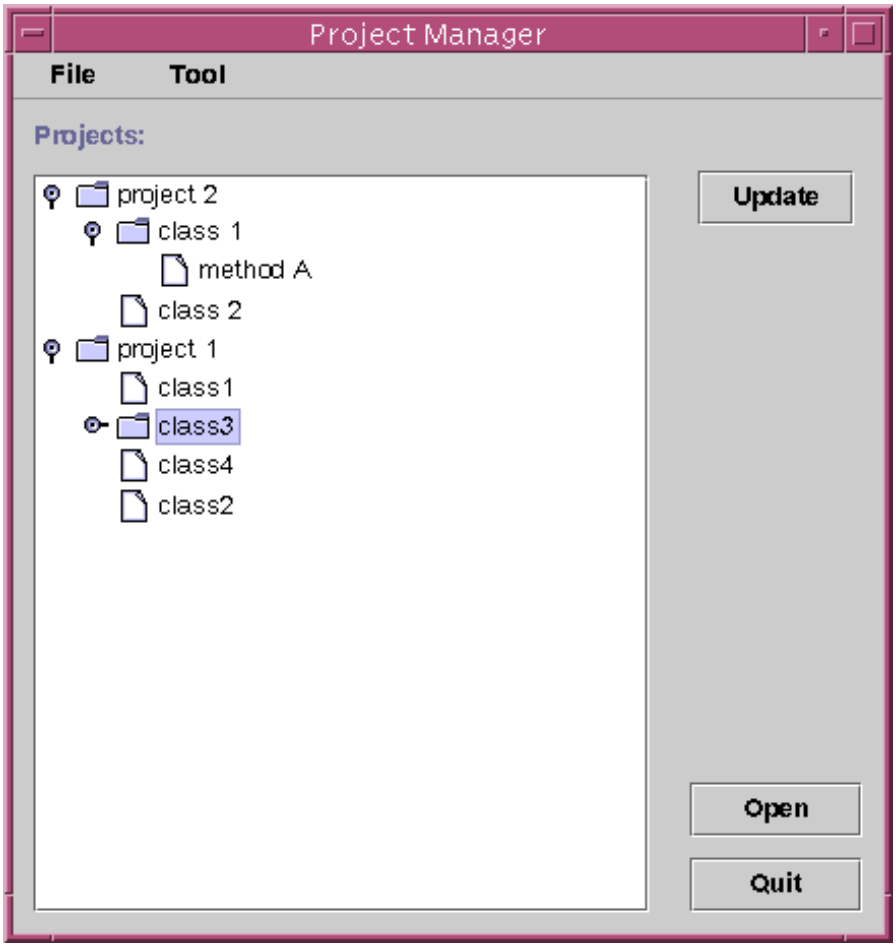


Abbildung 30.2.: Das Hauptfenster des Projektmanagers

auch als Startwerkzeug für alle weiteren Tools. Welche speziellen Werkzeuge überhaupt verfügbar sind, läßt sich mit dem Menüpunkt *Tool/Show available Tools* anzeigen. Das Starten dieser externen Programme geschieht hingegen mit *File/Open* oder mittels Button *Open* im Hauptfenster, sofern für das selektierte Objekt ein entsprechendes Werkzeug verfügbar ist. Somit startet bei einer ausgewählten Klasse der Klasseneditor, bei einer Methode der Methodeneditor usw.

Eine genaue Beschreibung des Kontrollflusses beim Starten eines Editors findet sich im Abschnitt 25.5 auf Seite 213.

30.2. Entwurf

Da der Projektmanager als vollwertiges Werkzeug im Laufe des Softwareprozesses hinter die *DoDL*-Editoren zurückgestellt wurde, entstammt die vorliegende Version einer Evolution des Hilfsprogramms *Toolstarter*, das als Einstiegspunkt für die Entwicklungsumgebung unabdingbar war.

Dieser unglückliche Zustand spiegelt sich im zugrundeliegenden Entwurf wider, der im Sinne eines softwaretechnologischen Entwurfs gar nicht stattgefunden hat. Aus diesem Grunde wurde auch keinerlei Modellierung in UML durchgeführt und demzufolge kein Code mittels Rational Rose erzeugt. Der Projektmanager wird lediglich durch die beiden Klassen `ProjectManager.java` und `ProjectManagerGUI.java` realisiert.

Dabei handelt es sich bei `ProjectManagerGUI.java` im Wesentlichen um den von Visaj erzeugten Code, mit dessen Hilfe die grafische Benutzungsoberfläche dieses Werkzeugs erstellt wurde. Um die Implementierung der Funktionalität der Oberfläche wenigstens grob von der Deklaration ihrer grafischen Elemente zu trennen, dient die Klasse `ProjectManager.java`, die sich unter dem Namen `gui` ein Objekt vom Typ `ProjectManagerGUI` instanziiert und darauf arbeitet. Darüber hinaus implementiert die Klasse `ProjectManager.java` die Interfaces `ToolStarter` und `PropertyChangeListener`.

Das Interface `ToolStarter` dient in unserem Entwurf als einheitliche Schnittstelle, die die Methode `startTool` enthält, mit deren Hilfe ein Werkzeug ein anderes aufrufen kann. Dabei muß dem neu zu startenden Werkzeug das Modellobjekt übergeben werden, auf dem bisher gearbeitet wurde. Dieser Mechanismus erlaubt uns das oben beschriebene Starten des Klasseneditors aus dem Projektmanager heraus.

Die Implementierung des Java-eigenen `PropertyChangeListener`-Interfaces dient hingegen dem Zweck, daß Instanzen des Projektmanagers auf Änderungen an den angezeigten Projekten reagieren können, die von anderen Benutzern zur gleichen Zeit durchgeführt werden, etwa das Hinzufügen neuer Klassen und Methoden oder das Ändern des Projektnamens. Dazu meldet sich jeder Projektmanager als `ContentDifferenceListener` bei dem zugrundeliegenden `OMSProject` an, um über `PropertyChangeEvents` von Modifikationen in Kenntnis gesetzt zu werden.

Dies geschieht beim Aufbau des JTree, der im linken Fenster alle verfügbaren Projekte anzeigt. Die hierfür vorhandene Methode `createProjectsTree` bedient sich dabei der dieser Sitzung zugrundeliegenden `OMSRootSession`, bei der mittels `getProjects` alle in der Datenbank befindlichen Projekte erfragt werden können. Für jedes in dem Rückga-

bearray gelieferte Projekt wird ein `DefaultMutableTreeNode` (s. u.) erzeugt, mit diesem assoziiert und in den JTree `top` eingehängt. Analog werden alle in einem Projekt befindlichen Objekte erfragt, durchlaufen und als Kindknoten unterhalb des Projektknotens in den Baum eingefügt. Wird nun ein `PropertyChangeEvent` abgefangen, das von einem Projekt ausgelöst wurde, so wird, aus Gründen der einfachen Implementierbarkeit, der JTree einfach neu aufgebaut. Eine genaue Beschreibung des Ablaufs eines solchen Ereignisses ist im Kapitel 25.5 auf Seite 213 zu finden.

Somit läßt sich die realisierte Architektur des Projektmanagers durch Abb. 30.3 auf der nächsten Seite beschreiben, die im Sinne einer Rekonstruktion nach der Implementierung anhand des Quelltextes erstellt wurde.

30.3. Fehlende Funktionalitäten

Abschließend soll auf die Differenzen zwischen Entwurf und tatsächlicher Realisierung eingegangen werden. Zunächst folgt eine Aufzählung technischer Einzelheiten, deren Umsetzung dem Zeitmangel des Projekts zum Opfer fiel:

- Die Visualisierung der verschiedenen Objekttypen im Navigationsbaum mit individuellen Icons wurde unterlassen. Statt dessen verwendet der Projektmanager die Standardsymbole gemäß der Java-Klasse `DefaultMutableTreeNode`.
- Das Anlegen neuer Klassendiagramme und das Starten des Klassendiagrammeditors wird nicht unterstützt.
- Die Eventverarbeitung nach Veränderungen an Projekten ist nur rudimentär implementiert. Z. Z. wird lediglich nach Hinzufügen neuer Klassen oder Methoden die Anzeige aktualisiert.

Weiterhin ist anzumerken, daß diskutiert wurde, das Administrationswerkzeug der $\mathcal{HEU}$, wie es in Kapitel 27.5 auf Seite 258 beschrieben wird, in den Projektmanager zu integrieren. Dies erschien naheliegend, da beim Starten des Werkzeuges abhängig von des Rechten des Benutzers einige zusätzliche Menüpunkte hätten eingebunden werden können, die die zur Administration der Umgebung notwendigen Funktionen bieten.

Dazu wäre neben einem zusätzlichen Menü zum Anlegen und Löschen von Benutzern beispielsweise die Integration einer zweiten Anzeigefläche für vorhandene Benutzer denkbar, mittels derer Berechtigungen an den ohnehin angezeigten Objekten vergeben werden könnten. Eine weiterreichende Spezifikation dieser Idee wurde jedoch recht früh aus den bereits häufig genannten Gründen unterlassen.

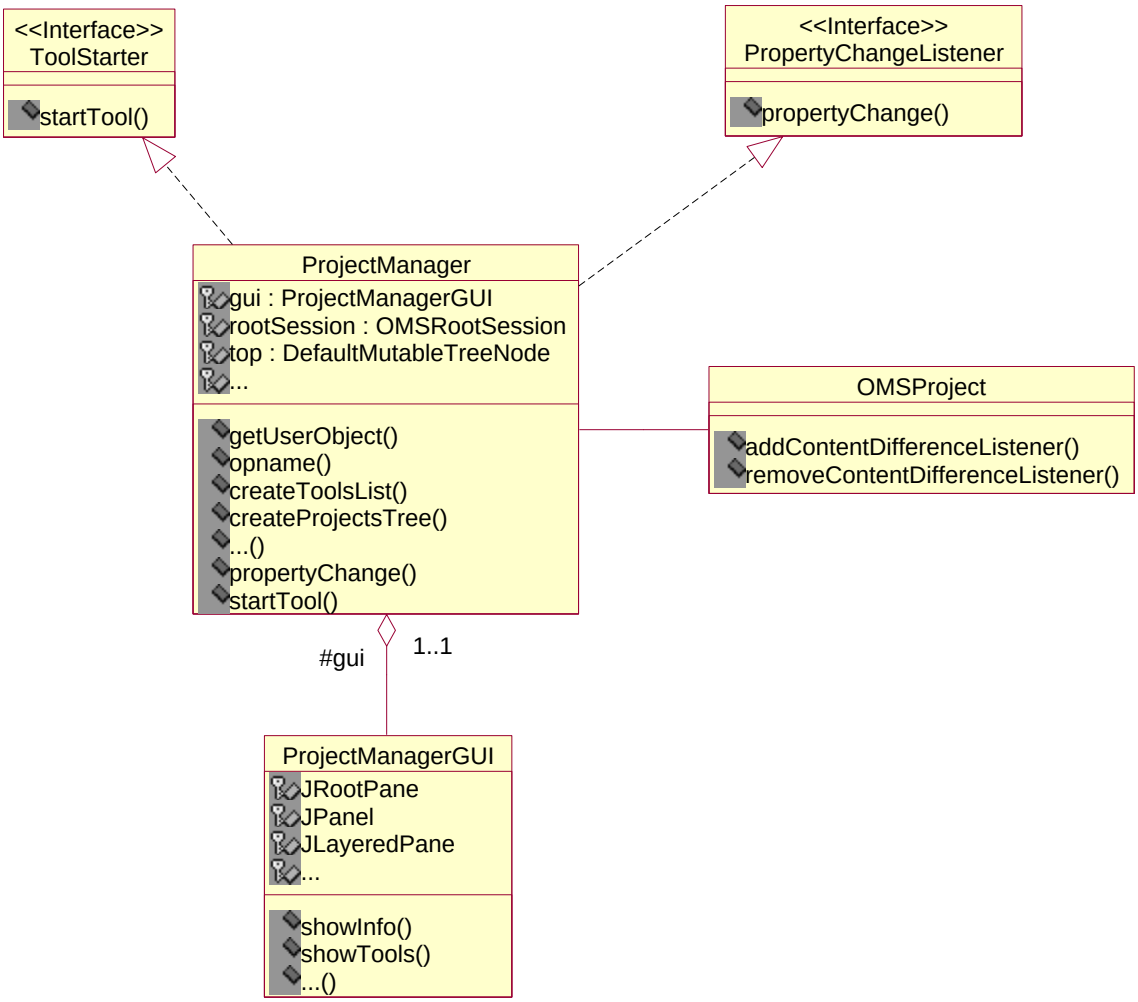


Abbildung 30.3.: Die Architektur des Projektmanagers

31. Klassen-, Methoden- und Klassendiagrammeditor

Autor: *Adil Kassabi*

In der $\mathcal{HEU}$ gibt es drei Editoren, der Klassen-, Methoden- und Klassendiagramm-Editor. Der Klasseneditor und Methodeneditor haben eine Fülle von Gemeinsamkeiten im Entwurf und in der Realisierung, die im folgenden näher beschrieben werden, bevor auf die Editoren im einzelnen eingegangen wird.

31.1. Entwurf

Die GUI des Klasseneditors bzw. Methodeneditors wurde so entworfen, daß sie von der Datenbankschicht völlig entkoppelt ist. Nach dem Konzept der komponentenbasierten Entwicklung haben wir versucht, die GUI als atomare Einheit zu entwickeln und die Schnittstelle datenbankunabhängig zu halten. Die GUI dieser beiden Editoren ist nach dem MVC-Konzept entworfen und realisiert worden. Sie ist eine Sicht auf die Daten in der Modellschicht. Die vom Benutzer vorgenommenen Änderungen werden erst angezeigt, wenn sie per Notifikation von der Modellschicht hochgereicht werden.

31.1.1. Beziehung zu den Adaptern

Die Adapter sind Klassen, die Funktionen zum Reagieren auf die Modellschichtnotifikationen zur GUI, kapseln. Sie sind aus der gewonnenen Erkenntnis entstanden, daß alleine die Reaktion auf die von der Modellschicht gereichten Notifikationen schon eine Vielzahl von Funktionen bilden, die zusamengehören. Der Übersichtlichkeit halber, und um keine riesige monolithische GUI-Klassen zu bilden, hat man diese Funktionalitäten in eigenen Klassen gekapselt. Eine vollständige Spezifikation der Funktionsweise dieser Adapter ist in Abschnitt 29.4 auf Seite 306 zu finden.

31.2. Der Klasseneditor

Der Klasseneditor ist ein Tool, mit dem die im Klassendiagramm generierten Klassen bearbeitet werden können. Die Bearbeitung umfasst das Einfügen, Löschen und Bearbeiten von Klassendokumentationen, Attributen, Methodennamen und Rückgabetypen von Methoden. Angezeigt werden auch die inneren Klassen einer gegebenen Klasse, sie lassen

sich aber nur in eigenen Klasseneditoren und nicht im Klasseneditor der umgebenden Klasse bearbeiten. Der Klasseneditor implementiert das `heu.System.Tool` Interface und kann über die dort definierte Methode `start()` gestartet werden.

31.2.1. Funktionsweise und Bedienungsanleitung

Die graphische Benutzungsschnittstelle des Klasseneditors ist ein Fenster `ClassEditorFrame` aus drei Tabs und einer Buttonleiste bestehend. Die Buttonleiste bietet drei Funktionen an:

- Undo-Button: Zum Zurücknehmen eines Schrittes der vorgenommenen Änderungen in der Arbeitskopie. Zu diesem Zweck wird die Methode `DBSession.undo()` in der aggregierten Instanz `dbsession` aufgerufen.

- Close-Button: Zum Speichern und Beenden des Klasseneditors. Dazu wird die Methode `ModelDoDLClass.close()` im aggregierten Objekt `doDLClassModel` aufgerufen. Das Objekt `doDLClassModel` ist eine Repräsentation einer *DoDL*-Klasse auf der GUI.

- Cancel-Button: Zum Beenden des Klasseneditors und um die Sitzungsänderungen nach dem letzten Auschecken zu verwerfen. Damit die Sitzungsänderungen verworfen werden, wird die Methode `DBSession.close(boolean)` mit dem Parameter `false` aufgerufen.

Im folgenden sind die Tab-Funktionalitäten im einzelnen aufgeführt:
Abb 31.1 auf der nächsten Seite zeigt den Class-Tab an. In diesem Tab werden die folgenden Informationen und Bearbeitungsmöglichkeiten einer Klasse dem Benutzer angeboten:

- Das oberste GUI-Element zeigt den Klassennamen an. Sobald die Klasse ausgecheckt ist, kann der Klassenname geändert werden. Diese Änderung wird der Datenbank durch das Drücken der Enter-Taste mitgeteilt. Das Drücken der Enter-Taste führt zum Aufruf der Methode `ModelDoDLClass.setName(String)` im aggregierten Objekt `doDLClassModel`.

- Die nächsten GUI-Elemente sind das Version- und SuperClass-Feld. Sie sind vom Klasseneditor aus nicht direkt änderbar. Die Version wird aber durch Einchecken und neu Auschecken automatisch inkrementiert.

- Die inneren Klassen werden in der innerClasses-Tabelle angezeigt und können in dieser nicht bearbeitet werden. Dies geschieht in einer eigenen Instanz des Klasseneditors.

- Die Dokumentation der Klasse kann – sobald die Klasse ausgecheckt ist – bearbeitet werden und mit Apply in der Datenbank persistent gemacht werden. Dazu wird die Methode `ModelDoDLClass.setDocumentation(String)` aufgerufen.

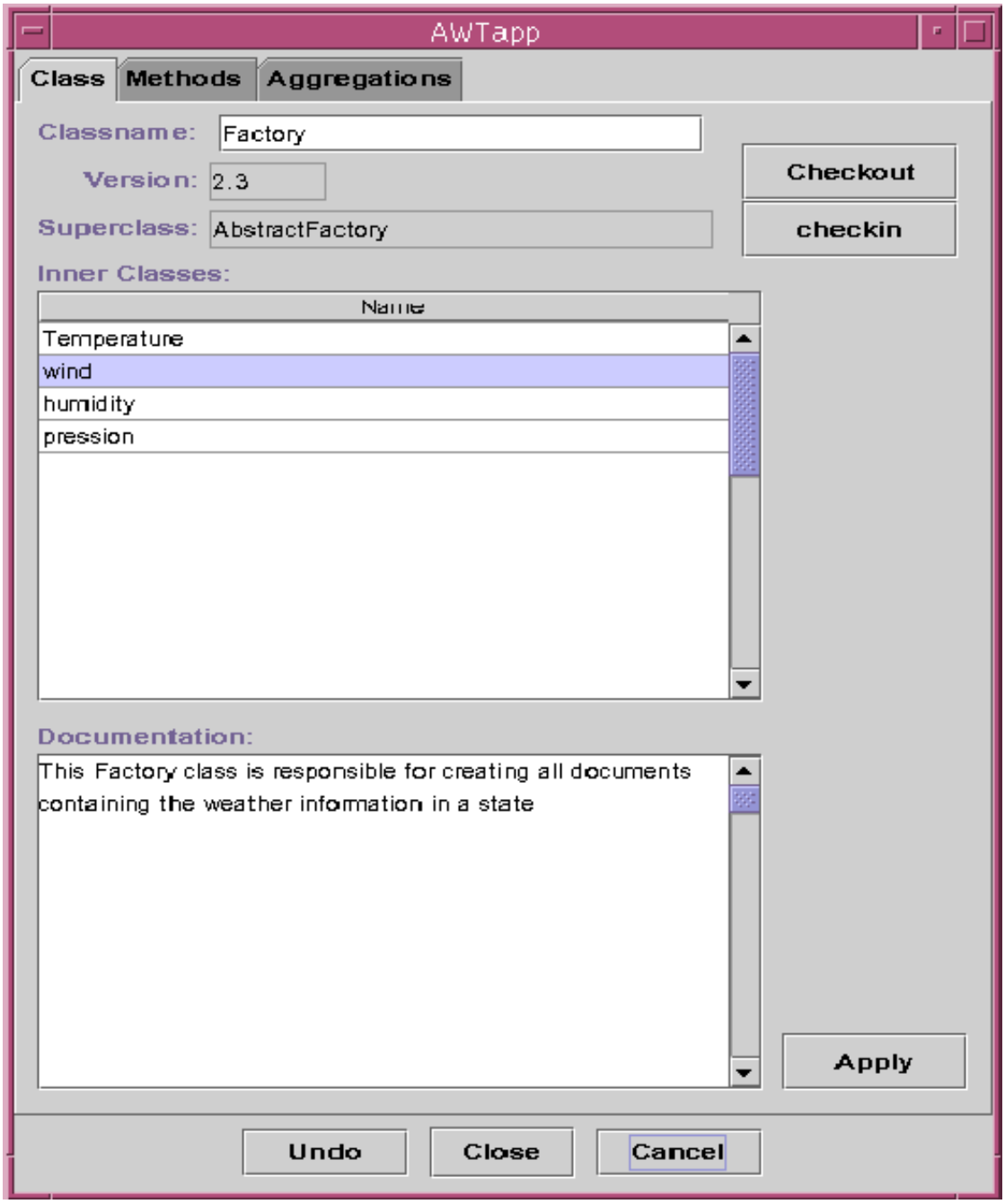


Abbildung 31.1.: Klasseneditor: Klassen-Tab

- Der Checkout-Button steht für das Auschecken der Klasse und wird, nur solange die Klasse ausgecheckt ist, ausgegraut (disabled). Das Auschecken erfolgt durch den Aufruf der Methode `ModelDoDLClass.checkout()` im Objekt `doDLClassModel`.

- Der Checkin-Button steht für das Einchecken der Klasse und wird mit dem Apply-Button ausgegraut, wenn die Klasse noch nicht ausgecheckt ist. Dazu dient die Methode `ModelDoDLClass.checkin()` im Objekt `doDLClassModel`.

Abb 31.2 auf der nächsten Seite zeigt den Methoden-Tab an. In diesem Tab werden die folgende Informationen und Bearbeitungsmöglichkeiten der Methoden einer Klasse dem Benutzer angeboten:

- In der Methodentabelle werden die Methoden aufgeführt, die zu dieser Klasse gehören. Angezeigt werden auch die Rückgabetypen. Beim Selektieren einer Methode in der Methodentabelle wird die entsprechende Dokumentation im darunter liegenden Dokumentationsbereich angezeigt.

- Um eine Methode der Klasse hinzuzufügen, steht dem Benutzer der Add-Button zur Verfügung. Das Klicken auf den Add-Button führt – sofern die Klasse ausgecheckt ist – zur automatischen Generierung einer Methode mit Standard Name und Standard Rückgabetyp. Die so generierte Methode kann mit dem Methodeneditor bearbeitet werden. Dazu wird die Methode `ModelDoDLClass.insertMethod(String)` mit einem automatisch generierten Namen aufgerufen.

- Eine selektierte Methode kann – sofern die Klasse ausgecheckt ist – über den Remove-Button von der Arbeitskopie entfernt werden. Dazu wird die Methode `ModelDoDLClass.removeMethod(ModelDoDLMethod)` verwendet.

- Der Open-Button steht für das Öffnen des Methodeneditors mit der einer eigenen Instanz des Klasseneditorsselektierten Methode. Dazu wird die Methode `DBSession.getToolStarter.startTool(ModelDoDLObject,dbSession)` aufgerufen.

- Die Checkout- und Checkin-Buttons dienen zum Einchecken und Auschecken selektierter Methoden. Dazu wird die Methode `ModelDoDLMethod.checkout()` bzw. `ModelDoDLMethod.checkin()` des entsprechenden Objektes der `ModelDoDLMethod`-Klasse der selektierten Methode.

Abb 31.3 auf Seite 326 auf Seite 326 zeigt den Methoden-Tab an. In diesem Tab werden die folgende Informationen und Bearbeitungsmöglichkeiten der Attributen einer Klasse dem Benutzer angeboten.

- In der Aggregationentabelle werden die Aggregationen mit ihren Typen angezeigt und können durch das Klicken auf die einzelnen Einträge bearbeitet werden. Das Beenden einer Änderungsaktion wird mit der Enter-Taste bekanntgegeben. Die Reaktion darauf ist der Aufruf der Methode `ModelDoDLClass.replaceAggregation()` mit dem alten und dem neuen Wert der Aggregation.

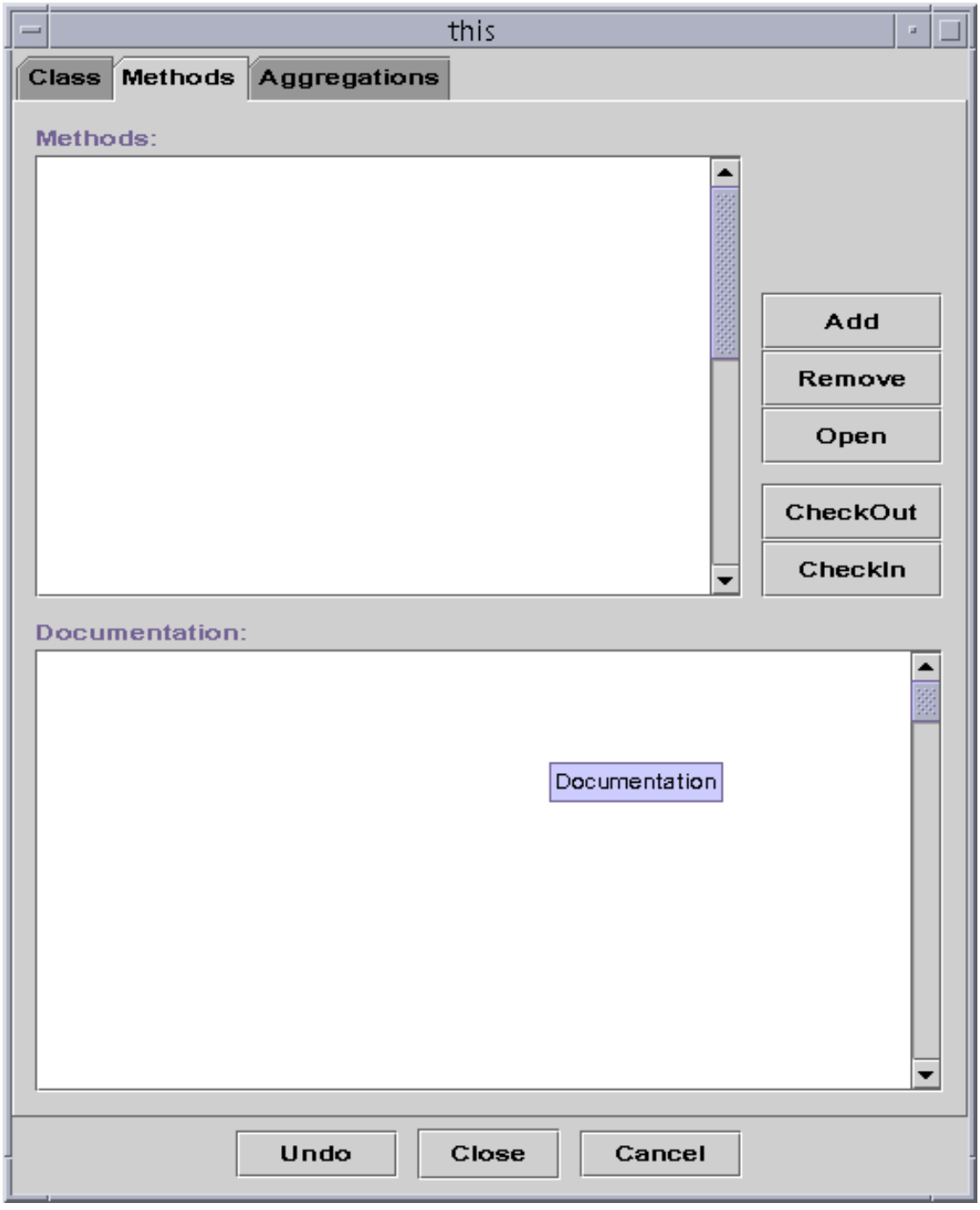


Abbildung 31.2.: Klasseneditor: Methoden-Tab

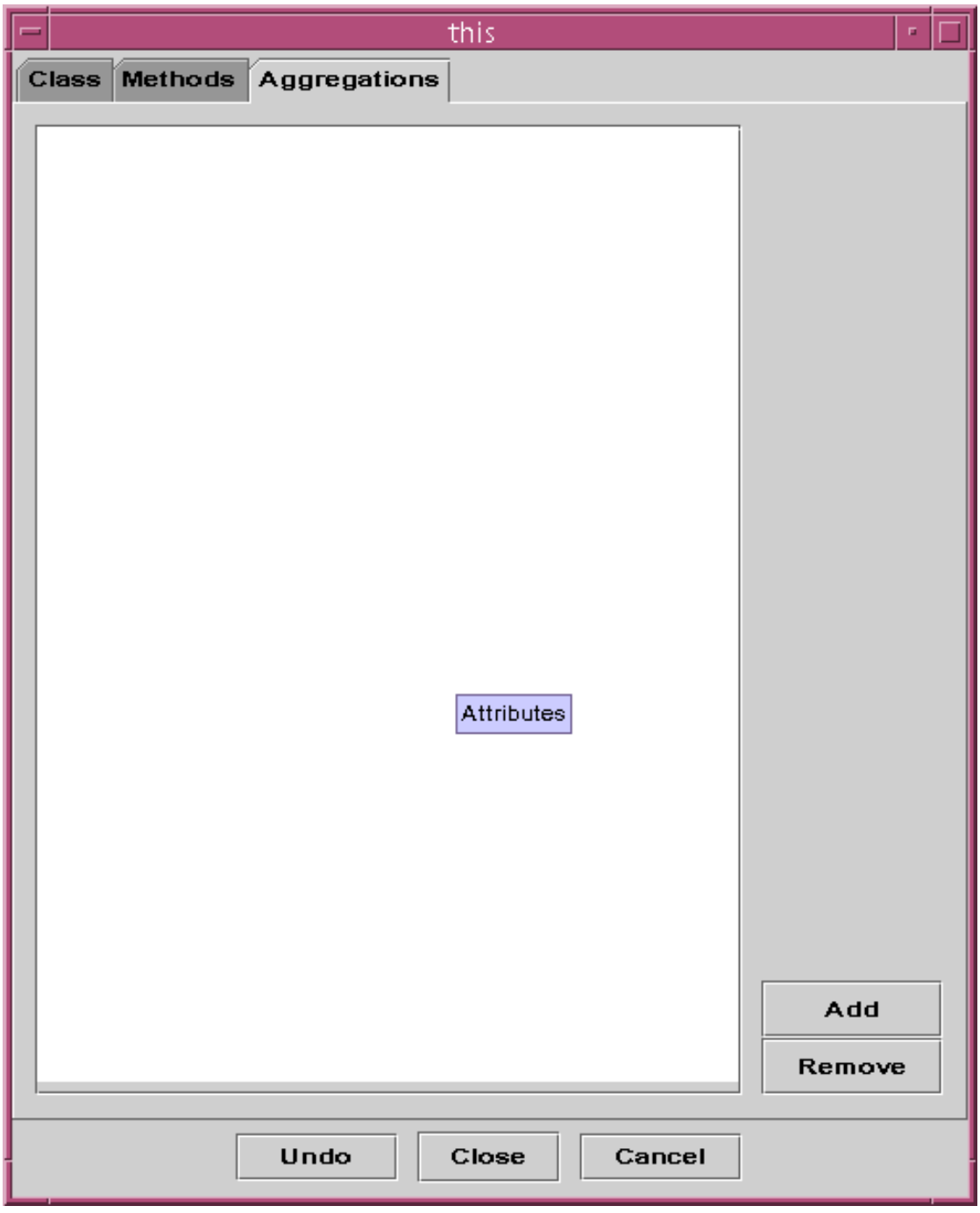


Abbildung 31.3.: Klasseneditor: Aggregationen-Tab

- Einfügen und Löschen der Attribute geht über die Add- und Remove-Buttons. Beim Einfügen werden automatisch Attributnamen und Attributtypen generiert. Diese können dann vom Benutzer bearbeitet und geändert werden. Beim Einfügen, genauso wie beim Löschen, wird die Methode `ModelDoDLClass.insertAggregation()` mit einem automatisch generierten Namen bzw. `ModelDoDLClass.removeAggregation()`mit der selektierten Variable als Parameter übergeben aufgerufen.

31.2.2. Beziehung der Editoren zur Modellschicht

Die Modellschicht ist die Schicht, die die GUI mit der Datenbank verbindet und für die Entkopplung der GUI von der Datenbank sort. Diese Schicht stellt die Klassen und Methoden zur Verfügung, die der Klasseneditor braucht, um die vom Benutzer vorgenommenen Änderungen der Datenbank mitzuteilen. Die Klassen der Modellschicht, die der Klasseneditor benutzt, sind:

- `ModelObject`: Das Basisobjekt aller *DoDL*-Objekte
- `ModelDoDLClass`: modelliert eine *DoDL*-Klasse.
- `ModelDoDLMethod`: modelliert eine *DoDL*-Methode und reicht alle Aufrufe an die Datenbankschicht weiter.
- `ModelDoDLVariable`: Datenstruktur zum Speichern einer *DoDL*-Variable wie Attribut und Parameter.

Eine nähere Beschreibung der einzelnen Klassen und die darin enthaltenen Methoden ist in der API-Dokumentation und im Abschnitt 28.3 auf Seite 291 zu finden. Die Modelschichtklassen befinden sich in den Paketen `heu.model` und `heu.dodl.model`.

31.2.3. Fehlende Funktionalitäten

Einige Enwurfs-Ideen, -Konzepte und -Ziele ließen sich aus Zeitmangel nicht vollständig realisieren, und viele Funktionen blieben unvollständig realisiert. Hier folgt eine Liste aller Funktionen, die zum Zeitpunkt der Dokumentation nocht gefehlt haben:

- Aggregationsnamen ändern
- Aggregationstypen ändern
- Methodennamen ändern
- Methodenrückgabetypen ändern
- Aus- und Einchecken von Klassen bzw. Methoden.

31.3. Der Klassendiagrammeditor

Autor: *Wilhelm Leibel*

Der Klassendiagrammeditor dient zum Modellieren von Klassendiagrammen. Man kann Klassen hinzufügen und löschen. Es können Vererbungs- und Aggregationsbeziehungen zwischen den Klassen erstellt werden. Außerdem können innere Klassen erstellt werden. Nicht-gesperrte Klassen können ausgecheckt und editierbare Klassen wieder eingecheckt werden. Anhand der Farben der Klassen erkennt man dessen Zustand:

- Rot: Die Klasse ist von einem anderen Benutzer gesperrt.
- Blau: Die Klasse ist für andere Benutzer gesperrt.
- Grün: Die Klasse ist gesperrt und ausgecheckt. Das heißt, die Klasse ist editierbar.
- Schwarz: Die Klasse ist nicht gesperrt.

31.3.1. Funktionsweise und Bedienungsanleitung

An linken Rand des Klassendiagrammeditors befindet sich eine Toolbar, dessen Befehle auch im "Insert"-Pulldown-Menü wiederzufinden sind. Über diese Buttons bzw. über die entsprechenden Menübefehle können neue Klassen, innere Klassen und Vererbungs- und Aggregationsbeziehungen zwischen den Klassen hinzugefügt werden. Die Funktionen der oberen Toolbar, die auch im Edit-Menü wiederzufinden sind, werden nachfolgend beschrieben:

- Undo: Mit dem Undo-Befehl kann die letzte Aktion rückgängig gemacht werden.
- Redo: Dieser Befehl macht die letzte Undo-Aktion rückgängig.
- Lock: Hiermit wird eine Klasse gesperrt und somit für andere Benutzer nicht mehr editierbar.
- Unlock: Damit wird eine Klasse wieder entsperrt.
- Checkin: Diese Funktion checkt eine Klasse ein.
- Checkout: Hiermit wird eine Klasse ausgecheckt.
- Remove from view: Damit wird ein Element im grafischen Editor entfernt, nicht aber in der Datenbank.
- Remove from database: Dieser Befehl löscht ein Element aus der Datenbank.
- Properties: Es wird der Klasseneditor mit der ausgewählten Klasse geöffnet.

31.3.2. Fehlende Funktionalitäten

In der zur Verfügung stehenden Zeit konnten leider nicht alle Funktionalitäten realisiert werden. Es wurde die grafische Benutzungsoberfläche erstellt und das Inteface `heu.System.Tool` implementiert. Die DFA-Klassen wurden aus der Ampelstadt übernommen. Der endliche Automat wurde noch nicht erstellt. Es fehlt außerdem die Spezifikation der OMS-Interfaces und deren Implementierung, sowie die Implementierung der Modellobjekte, der Drawable-Objekte und der Programmcode der meisten GUI-Events. Es funktionierte lediglich das Öffnen und Schliessen des Klassendiagramm-Fenster aus dem Projektmanager und der drei Dialogfenster.

31.4. Methodeneditor

Autor: *Adil Kassabi*

Der Methodeneditor ist ein Tool, mit dem die im Klasseneditor generierten Methoden bearbeitet werden können. Die Bearbeitung umfasst das Einfügen, Löschen und Bearbeiten von Methodendokumentation, Parametern, Methoden-Namen und -Rückgabetypen. Angezeigt wird auch die Methodenversion. Der Methodeneditor implementiert die `heu.System.Tool` Schnittstelle und kann über dort definierte Methode `start()` gestartet werden.

31.4.1. Funktionsweise und Bedienungsanleitung

Die graphische Benutzungsschnittstelle des Methodeneditors aggregiert die Klasse `MethodEditorFrame`. Die Klasse `MethodEditorFrame` ist ein Fenster bestehend aus zwei Tabs und einer Buttonleiste. Die Buttonleiste bietet drei Funktionen an:

- Undo-Button: Zum Zurücknehmen eines Schrittes der vorgenommenen Änderungen in der Arbeitskopie.
- Close-Button: Zum Speichern und Beenden des Methodeneditors.
- Cancel-Button: Zum Beenden des Methodeneditors, und um die Sitzungsänderungen nach dem letzten Auschecken zu verwerfen.

Im folgenden sind die Tab-Funktionalitäten im einzelnen aufgeführt:

Abb 31.4 auf der nächsten Seite zeigt den Signatur-Tab an. In diesem Tab werden die folgende Informationen und Bearbeitungsmöglichkeiten einer Klasse dem Benutzer angeboten.

- Das oberste GUI-Element zeigt den Methodennamen an. Sobald die Methode ausgecheckt ist, kann der Methodennamen geändert werden. Diese Änderung wird der Datenbank durch das Drücken der Enter-Taste mitgeteilt.

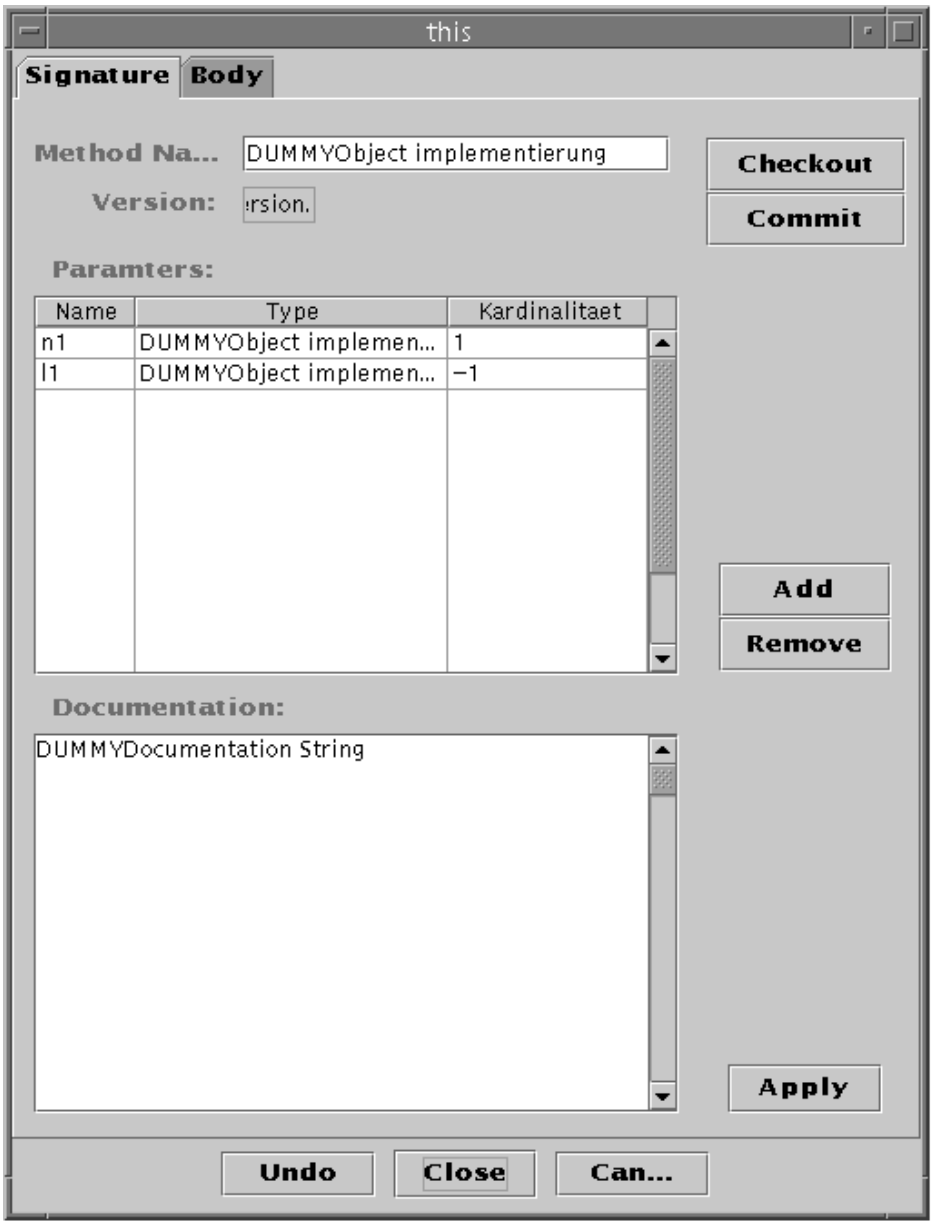


Abbildung 31.4.: Methodeneditor: Methoden Sigantur-Tab

- Die Parameterliste zeigt an den Namen, Typ und Kardinalität eines Parameters. Die Kardinalität ist entweder 1 oder n. Durch das Klicken auf eine Zelle der Parametertabelle kann der Name, Typ oder die Kardinalität editiert und anschließend mit der Enter-Taste die Änderung in die Datenbank geschrieben werden.
- Hinzufügen und Löschen von Parameter geht über den Add- und Remove-Button.
- Das nächste GUI-Element ist das Versionsfeld. Dieses Feld ist nicht direkt änderbar, die Version wird aber durch Einchecken und Auschecken automatisch inkrementiert.
- Die Dokumentation der Methode kann – sobald die Methode ausgecheckt ist – bearbeitet werden und mit dem Apply-Button in der Datenbank persistent gemacht werden.
- Der Checkout-Button steht für das Auschecken der Methode und wird, solange die Methode ausgecheckt ist, ausgegraut (disabled).
- Der Checkin-Button steht für das Einchecken der Methode und wird, wie auch der Apply-Button, ausgegraut, wenn die Methode noch nicht ausgecheckt ist.

Abb 31.5 auf der nächsten Seite zeigt den Body-Tab an. In diesem Tab werden die folgenden Informationen und Bearbeitungsmöglichkeiten des Methodenrumpfs dem Benutzer angeboten:

- Im Body-Bereich wird der Rumpf einer Methode angezeigt und kann, sobald die Methode ausgecheckt ist, als reiner Text bearbeitet werden.
- Um die Änderungen im Rumpf in der Arbeitskopie zu speichern, wird der Apply-Button verwendet.

31.4.2. Beziehung zur Modellschicht

Die Modellschicht ist die Schicht, die die GUI mit der Datenbank verbindet und dafür sorgt, die Entkoplung der GUI von der Datenbank zu gewährleisten. Diese Schicht stellt die Klassen und Methoden zur Verfügung, die der Methodeneditor braucht, um die vom Benutzer vorgenommenen Änderungen der Datenbank mitzuteilen. Die Klassen der Modellschicht, die der Methodeneditor benutzt sind:

- `ModelObject`: Das Rootobject aller *DoDL*-Objekte
- `ModelDoDLMethod`: Modelliert eine *DoDL*-Methode und reicht alle Aufrufe weiter an die Datenbankschicht.
- `ModelDoDLVariable`: Datenstruktur zum Speichern einer *DoDL*-Variable wie Attribute und Parameter.

Eine nähere Beschreibung der einzelnen Klassen und die darin enthaltenen Methoden ist von der API-Dokumentation zu beziehen. Die Modelschichtklassen befinden sich in den Packeten `heu.model` und `heu.dodl.model`.

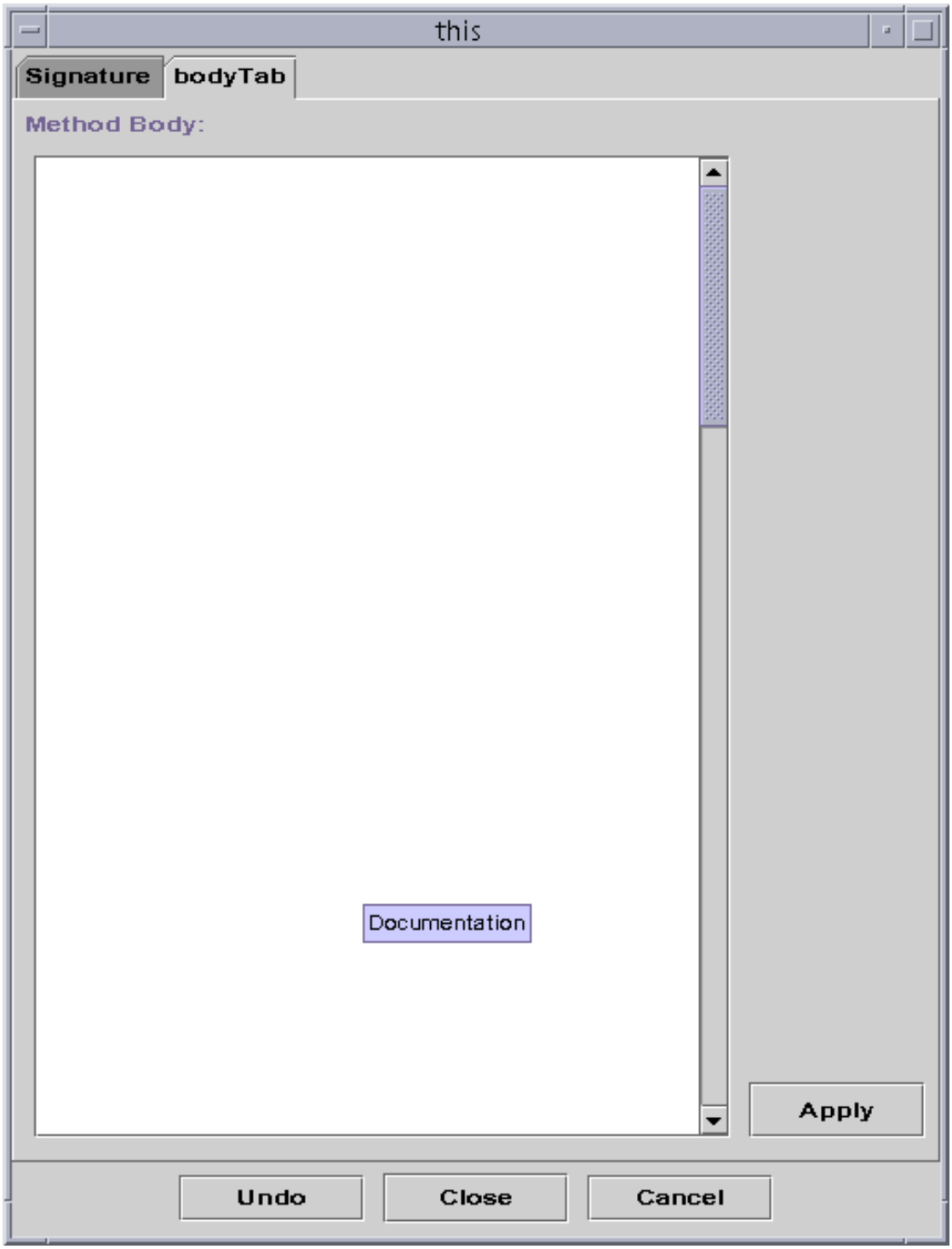


Abbildung 31.5.: Methodeneditor: Body-Tab

31.5. Fehlende Funktionalitäten

Einige Enwurfs-Ideen, -Konzepte und -Ziele ließen sich aus Zeitmangel nicht realisieren und folgende Funktionen blieben unvollständig realisiert. Hier folgt eine Liste aller Funktionen, die zum Zeitpunkt der Dokumentation nocht gefehlt haben:

- Parameternamen ändern
- Parametertypen ändern
- Rückgabetyp ändern
- Aus- und Einchecken von Klassen bzw. Methoden.

Teil IX.

Technische Dokumente

32. Richtlinien für die Implementierung

Dieses Kapitel enthält die Richtlinien für die Implementierung von Java-Code. Es besteht aus vier Teilen:

- Generelle Richtlinen für Quelltexte
- Richtlinien zum Dokumentieren des Quellcodes
- Verwendung der Klasse `Debug`
- Richtlinien zum Testen der Implementierung

Ziel dieser Richtlinien ist, ein einheitliches Erscheinungsbild und eine möglichst hohe Qualität des erzeugten Quellcodes zu erhalten.

32.1. Generelle Leitlinien für Java-Quelltext

Autoren: *Sascha Lüdecke*
Jens Schröder

Dieser Abschnitt enthält allgemeinere Hinweise, die nicht in einen der folgenden Abschnitte einzuordnen waren. Dies sind Namenskonvention, der Umgang mit veralteten Methoden und das Einchecken von Code.

Namenskonvention für get/set-Mehoden Diese Methoden müßen immer einen Namen der Form `get<AttributName>` oder `set<AttributName>` haben. Selbstverständlich sind diese Methoden nur für Attribute sinnvoll, die außerhalb einer Klasse ansprechbar sein sollen.

Veraltete Methoden sind nach vorhergehender Absprache mit der Gruppe mit dem Schlüsselwort `deprecated` zu versehen und damit für den weiteren Gebrauch zu vermeiden. Der Java-Compiler wird dann bei der Verwendung dieser Methoden eine Warnung ausgeben.

Einchecken von Code Es darf *ausschließlich* Quelltext mit korrekter Syntax eingecheckt werden. Das Repository wird nicht syntaxgerechten Code ablehnen, indem es vor dem Einchecken einen Javacompiler zum Testen aufruft. Dieser erfasst jedoch nicht logische Fehler oder Fehler in den Kommentaren.

32.2. Gebrauch von javadoc

Autor: *Christoph Begall*

Um eine einheitliche Dokumentation mit *javadoc* zu gewährleisten, sind die folgenden Regeln für alle Programmierer verbindlich.

1. Javadoc Kommentare werden immer in der Form

   ```
   /**
    * Text
    */
   ```

   geschrieben. (So wird es auch in den JDK-Klassen gemacht.) Alternative Möglichkeiten wie

   ```
   /**
     Text
    */
   ```

   oder

   ```
   /** Text
   */
   ```

   sollten nicht benutzt werden. Das erleichtert die Lesbarkeit, da sich andere Programmierer nicht jedesmal umstellen müssen.

2. Es wird stark empfohlen zu jeder Klasse, jedem Attribut und jeder Methode einen solchen Kommentar zu schreiben. Schliesslich schreibt Quelltext nicht für den Compiler, sondern für die Augen anderer, die ihn später verstehen und warten müßen.

3. Die Verwendung von HTML-Tags in Javadoc-Kommentaren wird durch *javadoc* erlaubt. Man sollte davon möglichst wenig Gebrauch machen, weil es den Text im Kommentar unleserlich macht. Wenn dennoch diese Möglichkeit genutzt wird, so sind nur die folgenden sinnvoll: `<br>` für Zeilenende, `<p>` für Absatz (meist in der Darstellung einer Leerzeile) in längeren *javadoc*-Kommentaren, `<tt>text...dumdidum</tt>` für Beispiele (nicht übertreiben), `<b>Wichtig</b>` für wesentliche Dinge, und eventuell `<i>Wert</i>` um etwas zu betonen oder den Wert einer Variable von ihrem Namen zu unterscheiden (nochmal – nicht übertreiben). Ein ähnliches Problem ergibt sich bei HTML-Ersatz-Darstellung wie `&auml;` oder `&quot;`, von deren Gebrauch abgeraten wird. Umlaute sollten einfach hingeschrieben werden, andere Zeichen nicht nötig sein. In diesem Text wurde aus drucktechnischen Gründen ganz auf sie verzichtet.

4. Der Klassenkommentar enthält ausführliche Informationen darüber, wozu die Klasse dient, was sie darstellt, wie sie benutzt wird, und wer sie wann geschrieben hat. Beispiel:

```
/**
 * Die Klasse Dictionary stellt ein systemweites Woerterbuch
 * zur Verfuegung. Im konkreten Fall wird es als Englisch-
 * Deutsch-Woerterbuch eingesetzt, kann aber auch fuer
 * andere Zwecke benutzt werden.<br>
 * Die Instanzierung erfolgt normalerweise nur einmal,
 * danach sollten alle anderen Klassen auf das Woerterbuch
 * zugreifen koennen. Private Woerterbuecher sind nur
 * bedingt sinnvoll.
 *
 * @author Sascha Luedecke, Christoph Begall
 * @date 11.12.1998
 */
public class Dictionary {

⋮

}
```

   Die Schlüsselworte `@author` und `@date` geben Autor und Datum an, wobei Autor immer der letzte ist, der größere/mittlere Änderungen durchgeführt und sich hinten an die Autorenliste angehängt hat und Datum das Datum ist, an dem diese Änderungen abgeschlossen wurden.

5. Attribute beschreiben den Zustand eines Objekts, und genau das muß in den Attribut-Kommentaren deutlich gemacht werden: Wozu dienen die Variablen? Eventuell auch, wer sie verändern darf. Wichtig: Der erste Satz sagt alles wichtige kurz und prägnant. Er endet beim **ersten** Punkt und wird wegen seiner Sonderrolle immer mit einer Leerzeile abgesetzt. Beispiel:

```
/**
 * Flag zur Synchronisaton, um gleichzeitiges
 * Schreiben mehrerer Klienten zu verhindern.
 *
 * Wird nur von der Methode
 * addTranslation() ver"andert.
 */
private boolean sync;
/**
 * Hashtable, die alle Worte als Schluessel und deren
```

```
     * Uebersetzung als Wert enthaelt.
     *
     * Sie verrichtet eigentlich die ganze Arbeit.
     */
    private Hashtable storage;
```

⋮

6. Methoden-Kommentare müssen sehr unterschiedliche Aufgaben erfüllen. „private'-Methoden können nur innerhalb der Klasse aufgerufen werden. Es sollte auf jeden Fall klar erläutert werden, was die Methode tut und möglichst auch mit `@param` ihre Parameter und mit `@return` ihr Rückgabewert erläutert werden. Bei „protected'-Methoden kann es sein, daß diese überladen werden. Es muß auf jeden Fall jeder Parameter und der Rückgabewert erläutert werden, außerdem muß es eventuell eine Extra-Erläuterung für eventuelle Überlader geben, falls die Methode bestimmte Dinge sicherstellen muß. Für „public'Methoden gilt das Gleiche, nur daß sie als Schnittstelle nach außen höchstens noch besser erläutert werden muß. Jede Exception sollte erläutert werden, weil anhand des Kommentars der Aufrufer entscheiden kann, was in diesem Fall zu tun ist. Beispiel:

```
    /**
     * Speichert ein neues Wort inklusive Uebersetzung
     * im Woerterbuch.
     *
     * Kann z.B. so benutzt werden:<br><tt>
     * woerterbuch.addTranslation("one",
     * "eins").addTranslation("two", "zwei");</tt><br>
     * F"ur ueberlader: Es mu"s sichergestellt werden,
     * dass bei Anfang der Aktion sync true gesetzt
     * wird, und am Ende wieder false.
     *
     * @param word Das Wort, das gespeichert wird. Z.B.
     *  das englische Wort <i>seal</i>.
     * @param trans Die Uebersetzung des Wortes. In
     *  diesem Beispiel das deutsche Wort <i>Seehund</i>.
     * @return Referenz auf dieses Woerterbuch.
     * @exception DoubleDefineException Wenn der Wert
     *  von word schon eingetragen wurde.
     * @exception NullPointerException Wenn word oder
     *  trans den Wert null hatten.
     */
    public addTranslation(String word, String trans) throws
            NullPointerException, DoubleDefineException {
```

⋮

32.3. Debugausgaben

Autor: *Jens Schröder*

Es wurde beschlossen, die Debugausgaben über ein Objekt und nicht auf die Standardausgabe zu schreiben. Dadurch sind wir flexibler (wir können verschiedene Debuglevel verwenden) und haben die Möglichkeit, später leicht ein Logfile zu erzeugen.

Ich stelle zunächst die Debugausgabeklasse `DebugPrint` und anschließend die Richtlinien zur Verwendung von Debugausgaben vor.

32.3.1. Die Verwendung von DebugPrint

Die Klasse `DebugPrint` muß in der `main`-Methode einmal instanziiert werden. Der weitere Zugriff erfolgt dann über das statische Attribut `Debug`. Die Klasse `DebugPrint` bietet fünf verschiedene DebugLevel an:

1. `errorlevel`
2. `warninglevel`
3. `loglevel`
4. `infolevel`
5. `debuglevel`

Der Debuglevel ist bei der Instanziierung defaultmäßig auf `loglevel` gesetzt, es kann im Konstruktor aber auch explizit ein Debuglevel gesetzt werden. Zwei Methoden erlauben das Manipulieren des Debuglevels:

- `setLevel(DEBUGLEVEL)`
  Die Methode setzt den als Parameter übergeben Debuglevel. Es werden die oben angegeben Debuglevel als symbolische Konstanten verwendet. Der Debuglevel wird auf einen Stack gelegt, damit die Level später wieder zurückgesetzt werden können.
- `resetLevel()`
  Setzt den Debuglevel auf den letzten Wert. Man kann den Level beliebig oft zurücksetzten, bis man beim Startlevel angekommen ist.

Mit folgenden Methoden können die Debugausgaben gesetzt werden:

- `error(MESSAGETEXT)`
- `warning(MESSAGETEXT)`
- `log(MESSAGETEXT)`
- `info(MESSAGETEXT)`
- `debug(MESSAGETEXT)`

- `enter(MESSAGETEXT)`
- `exit(MESSAGETEXT)`

Es werden alle Debugausgaben mit ausgegeben, die kleiner bzw. gleich dem gesetzten Debuglevel sind. Ist z.B. der Debuglevel auf `loglevel` gesetzt, werden die Debugmeldungen von `log(MESSAGETEXT)`, `warning(MESSAGETEXT)` und `error(MESSAGETEXT)` ausgegeben.

Die Methoden `enter(MESSAGETEXT)` und `exit(MESSAGETEXT)` bieten die Möglichkeit, das Eintreten und Verlassen wesentlicher Methoden zu kennzeichen. Diese Debugausgaben werden entsprechend der Aufruftiefe eingerückt ausgegeben. Daher muß man sie paarweise verwenden! Die Ausgabe erfolgt auf der gleichen Ebene wie `info`-Ausgaben.

Zum Abschluß führe ich noch ein kleines Beispiel an:

```
// in der main-Methode ist die Klasse DebugPrint
// bereits instanziiert worden
DebugPrint.setLevel(DebugPrint.infolevel);
DebugPrint.log("Testklasse: Welteditor geöffnet");
DebugPrint.resetlevel();
```

32.3.2. Einsatzrichtlinien

Um das Setzen von Debugausgaben zu vereinheitlichen, haben wir beschloßen, Richtlinien aufzustellen.

Allgemein muß in der Debugmeldung der Klassenname immer mit angegeben werden (Beispiel s.o.). Bei Exceptions muß der Stacktrace mit ausgegeben werden. So hat man die Möglichkeit, die genaue Fehlerstelle zu lokalisieren. Aus der Debugmeldung sollte klar werden, in welcher Methode man sich gerade befindet, sofern es für das Verständnis wichtig ist.

Im Folgenden erläutere ich, wofür die einzelnen Ausgabemethoden verwendet werden sollen:

- **error**
  Ausgabe, wenn ein Fehler auftritt, der das System in keinem lauffähigen Zustand hinterläßt. Ein passender Ort wäre die Stelle im Code, an dem die Exception 'Datenbankverbindung abgebrochen' geworfen wird.
- **warning**
  Wenn ein Fehler auftritt, der nicht zum Programmabsturz führt, soll die `warning`-Methode verwendet werden
- **log**
  Die `log`-Ausgabe soll an markanten Stellen (Meilensteinen) im Programm benutzt werden, z.B. `DebugPrint.log('Editor: Stadt eingecheckt.')`. Es sollen Ausgaben sein, die später in ein Log-File geschrieben werden.
- **info**
  Die `info`-Ausgabe muß in jedem Konstruktor stehen. Es soll die Parameterliste mit ausgegeben werden. Sie ist für detailiertere Debugausgaben gedacht.

- **debug**
  Diese Ausgaben sind als Trace gedacht. Sie dienen zum genauen Debuggen.

- **enter/exit**
  Diese Methoden sollen nur beim Ein- bzw. Austritt von *wichtigen* Methoden benutzt werden (z.B. nicht get/set-Methoden). Es soll die Parameterliste bzw. der Rückgabewert mit angegeben werden.

32.4. Testen von Quellcode

Autor: *Sascha Lüdecke*

In diesem Abschnitt finden sich die Richtlinien zum Testen der Implementierungen des Entwurfs. Der Entwurf der Ampelstadt wurde in kleinere Teile gespalten, die von je zwei bis vier Leuten bearbeitet werden. Von diesen ist einer später auch für die Pflege des Teils verantwortlich, d.h. Integration von Änderungen am Entwurf bzw. die Delegation derselben.

32.4.1. Methodik des Testens

Generell testet niemand seinen eigenen Quellcode, man testet immer den anderer Programmierer. Dabei wird anhand der Spezifikation, und nicht nach "was implementiert wurde" getestet. Für wichtige Methoden[1] wird je eine Testmethode geschrieben, die später alle in einer zentralen Testklasse übernommen werden. So kann man später idealerweise mit einem Aufruf von `make test` oder ähnlich das gesamte Programm auf seine Korrektheit hin überprüfen.

32.4.2. Wie sehen die Testmethoden aus ?

Eine Testmethode testet idealerweise alle möglichen und vor allem kritische Fälle für die zugehörige Methode. Ihr Ziel ist es, die korrekte Arbeitsweise bei normalen und unnormalen[2] Parameterwerten zu prüfen. Treten später Fehler auf, so werden diese dort als Testfall mit aufgenommen.

Die Testmethode liefert `true` oder `false` zurück, je nachdem, ob der Test erfolgreich war, oder nicht. Eventuell auftretende Exceptions werden von ihr abgefangen und über das Werkzeug *Debug* inklusive ihrem "StackTrace" ausgegeben, sofern es sich um einen wirklichen Fehler handelt. Provozierte Exceptions, um die Reaktion auf falsche Parameterwerte zu testen, werden i.d.R. nicht protokolliert. Dabei sollen entsprechend der Vorgaben für das Wekzeug *Debug* folgende Ebenen verwendet werden:

1. **error** Eine Methode steigt mit einer ungewollten Exception aus, die bei einem normalen Lauf das Programm zum Abbruch zwingen oder Stillstand bringen würde.

[1] Dies sind z.B. nicht triviale oder größere Methoden, die bezüglich der Applikationslogik wesentlich sind. get/set-Operationen zählen sicher nicht dazu.
[2] zum Beispiel nicht initialisierte Objekte oder `null`-Werte

2. **warning** Eine Methode hat eine Fehlfunktion, die sich in einem falschen Rückgabewert äußert.

3. **log** Hier wird protokolliert, ob ein Testfall erfolgreich war. Dazu kommen Name der Methode und Kurzbezeichnung des Testfalls.

Alle Testmethoden werden für je eine Klasse des Entwurfs in einer Testklasse zusammengefasst, die diese sukzessive aufruft und deren Ergebnis protokolliert. Diese Klassen werden im Repository in einem eigenen Verzeichnis abgelegt und haben einen Namen nach folgendem Schema: `Test<zuTestendeKlasse>`

33. Das Model-View-Controller Konzept

Autor: *Sebastian Schütte*

Das Model-View Konzept stellt ein Entwurfsmuster dar, daß in einer Anwendung zum Einsatz kommen kann, bei der unterschiedliche Sichten auf eine gemeinsame Datenbasis angeboten werden. Wird in einer dieser Ansichten eine Änderung der Datenbasis vorgenommen, werden alle übrigen Ansichten aktualisiert.

Es wird versucht, eine kurze Einführung zum Model-View Konzept zu geben. Im vorliegenden Text steht das Prinzip der Aufgabenteilung, sowie die Art der Kommunikation zwischen Modell und Beobachter im Vordergrund, Implementationsalternativen werden nur angerissen. Im Folgenden wird davon ausgegangen, daß der Leser mit den Prinzipien der Objektorientierung vertraut ist, sowie der Begriff der Kopplung von Modulen aus der Softwaretechnologie bekannt ist.

33.1. Problemstellung

Eine Tabellenkalkulation bietet zahlreiche Möglichkeiten Datenreihen darzustellen:

- Eine Datenblattansicht, in der die Werte in Spalten und Zeilen angeordnet als Zahlen auf dem Bildschirm dargestellt werden.
- Ein Balkendiagramm, daß die zugrundeliegenden Daten als Rechtecke entsprechender Länge darstellt.
- Ein Tortendiagramm, in dem das Verhältnis der Werte zueinander dargestellt wird.

Ändert der Benutzer in einer dieser Ansichten einen Wert, werden typischerweise alle Ansichten aktualisiert. Das Verhalten des Programmes erweckt den Anschein, als ob sich die einzelnen Ansichten kennen und gegenseitig benachrichtigen können. Als zusätzliche Anforderung könnten die einzelnen Ansichten auf unterschiedlichen Rechnern dargestellt werden.

Im Allgemeinen läßt sich das Model-View Konzept verwenden, wenn an mehreren Stellen in einem Programm Werte einer Datenmenge geändert werden können und mehrere, möglicherweise auch unterschiedliche, Sichten auf diese Daten existieren. Die Änderung der Daten durch eine der Sichten wird dazu führen, daß auch die übrigen Sichten ihre Darstellung aktualisieren.

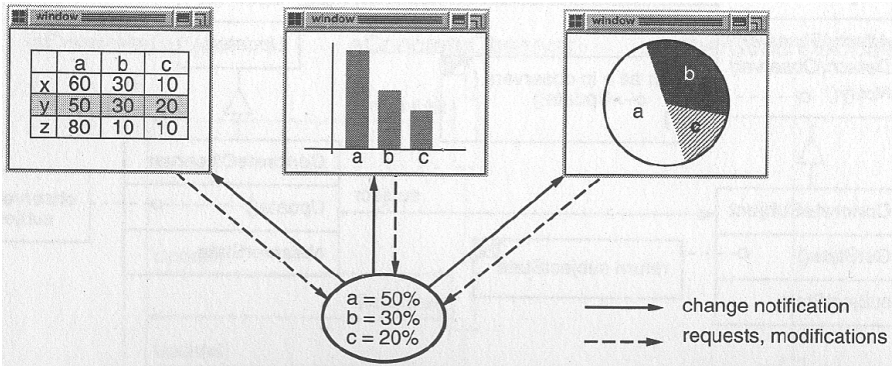


Abbildung 33.1.: Anwendung des MVC-Konzeptes

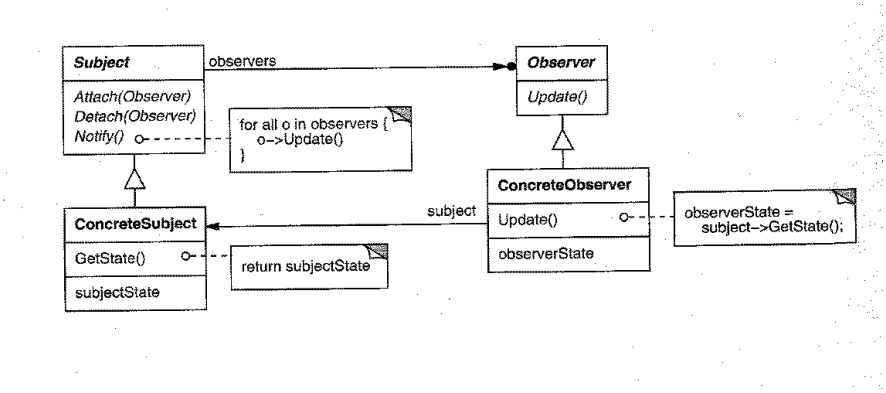


Abbildung 33.2.: Struktur des MVC-Konzeptes

33.2. Struktur

Das im vorherigen Abschnitt skizzierte Verhalten einzelner Programmteile wird erreicht, in dem ein zusätzliches Objekt eingeführt wird, daß keine direkte Repräsentation auf der Benutzeroberfläche besitzt. Dieses Objekt dient der Speicherung der Daten, die von dem Benutzer manipuliert werden können. Im folgenden wird ein derartiges Objekt "Modell" genannt. Das Modell bietet Methoden zur Manipulation und zur Inspektion der zu speichernden Daten an. Das Modell ist zur Verwendung durch andere, darstellende Komponenten gedacht.

Die verschiedenen Sichten besitzen selbst keine Repräsentation der zugrundeliegenden Daten, sie stützen sich auf die Methoden des verwendeten Modellobjektes ab. Derartige Ansichten werden im folgenden Beobachter genannt. Ein Modell kann unterschiedliche Beobachter besitzen, die jeweils eine andere Sicht auf die Daten darstellen. Insbesondere bieten Beobachter auch die Möglichkeit die Daten im verwendeten Modellobjekt zu verändern.

Ein Modell verfügt über die Fähigkeit, eine Menge von Beobachter zu verwalten. Dazu stehen zwei Methoden zur Verfügung, die ein Beobachterobjekt an- bzw. abmelden. Beobachter und Modell kennen und verwenden sich gegenseitig, Beobachter untereinander kommunizieren jedoch nicht miteinander.

33.3. Interaktion

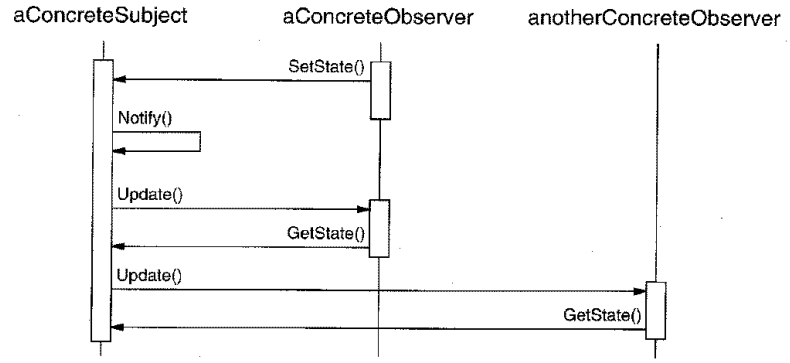


Abbildung 33.3.: Interaktion im MVC-Konzept

Das geforderte Verhalten wird durch ein geschicktes Kommunikationsprotokoll erzeugt. Erhält ein Beobachter Benutzereingaben, die zur Veränderung der Daten im Modell führen sollen, so werden die entsprechenden Methoden des Modellobjektes verwendet, um diese Veränderungen durchzuführen. Das Beobachterobjekt führt zu diesem Zeitpunkt noch keine Veränderung der Darstellung durch, dies geschieht erst später. Werden die Daten im Modellobjekt verändert, so werden alle angemeldeten Beobachter benachrichtigt (insbesondere auch der Beobachter, der die Datenänderung verursacht hat). Auf diese Benachrichtigung hin passen die Beobachter ihre Darstellung der Daten entsprechend an, wobei evtl. Inspektionsmethoden des Modells aufgerufen werden. Eine solche Benachrichtigung wird oft Notifikation genannt.

33.4. Eigenschaften

- Die Kommunikation der beteiligten Objekte verläuft immer nur zwischen Modell und Beobachter. Die verschiedenen Beobachter stehen in keiner direkten Verbindung zueinander.

- Ein Modell verwendet kein konkretes Beobachterobjekt sondern eine Menge von Beobachtern, die ohne Schwierigkeiten zur Laufzeit vergrößert und verkleinert werden kann. Als Schnittstelle zum Beobachter genügt dem Modell der Aufruf einer

einzigen Methode: Aktualisiere deine Ansicht. Ein Modell ist also nur lose an die Beobachter gekoppelt.

- Ein konkreter Beobachter verwendet inspizierende Methoden des Modelles, um seine Darstellung der Daten auf den aktuellen Stand zu bringen. Ein Beobachter ist also fester an ein Modell gekoppelt. Insbesondere kann ein neuer Beobachter die Einführung neuer inspizierender Methoden des Modelles notwendig machen.

- Beobachter müssen die Art der Zustandsänderung ohne Hilfe des Modelles herausfinden. Dies kann unter Umständen ineffizient sein.

- Da sich die Beobachter nicht kennen, sind die Kosten einer Datenänderung den Beobachtern unbekannt. Unter Umständen löst eine Datenänderung eine Kaskade von Benachrichtigungen und inspizierenden Methodenaufrufen aus.

- Das Model-View Konzept läßt sich leicht auf entfernte Beobachter anwenden. Beobachter müssen nicht auf dem gleichen Rechner laufen, wie das verwendete Modell.

- Das Model-View Konzept läßt sich verwenden, um mehreren Benutzern gleichzeitig die Arbeit an einer gemeinsamen Datensammlung zu ermöglichen.

33.5. Implementationsalternativen

33.5.1. Auslösen der Aktualisierung

Ein robuster Ansatz wäre es, auf jeden Aufruf einer zustandsändernden Methode des Modelles eine Benachrichtigung aller Beobachter auszulösen. So ist garantiert, daß die Darstellung der Beobachter immer auf dem aktuellen Stand ist. Alternativ sind die Beobachter selbst für das Auslösen der Benachrichtigung zuständig (z.B. durch Aufruf einer geeigneten Methode des Modelles). So läßt sich eine Folge von Datenänderungen mit einer Aktualisierung der Beobachter am Ende vornehmen, was in der Regel effizienter ist.

33.5.2. Push/Pull Methode

Bei der Pull-Methode werden die Beobachter darüber informiert, daß eine Zustandsänderung des Modelles stattgefunden hat. Der neue Zustand des Modelles muß über inspizierende Methodenaufrufe erfragt werden. Bei der Push-Methode schickt das Modell detailliertere Information über die Art der Zustandsänderung bei der Notifikation. (Im Extremfall sogar eine vollständige Zustandsbeschreibung). Dies erhöht die transportierte Informationsmenge, macht jedoch die Aktualisierung der Beobachter effizienter. Unter Umständen wird die Kopplung des Modelles an die Beobachter verstärkt, da von Seiten des Modelles Annahmen über die Beobachter und ihre Darstellung gemacht werden.

33.5.3. Aspekt-Konzept

Eine Alternative zur Verwendung der Pull- bzw. Push-Methode ist die Verwendung eines Aspekte-Konzeptes. Hierbei können sich Beobachter für bestimmte Aspekte der Daten anmelden und werden nur gemäß ihrer Anmeldung notifiziert. Dies gestattet eine detailliertere Information der Beobachter über die Art der Zustandsänderung des Modelles ohne jedoch bei jeder Notifikation Daten an Beobachter zu übermitteln, die diese gar nicht benötigen.

33.5.4. Verwendung eines zusätzlichen Controller-Objektes

Um verschiedene Aktualisierungsstrategien zu verwirklichen, kann ein zusätzliches Objekt verwendet werden, daß die Kommunikation zwischen Modell und Beobachter realisiert. Dann werden Beobachter bei einem sogenannten Controller angemeldet, der seinerseits bei dem zugrundeliegenden Modell angemeldet wird. Die Verwendung eines Controllers kann sinnvoll sein, wenn mehrere Beobachter mehrere Modelle beobachten, oder wenn sich durch die Verwendung von Transaktionen o.ä. die Kommunikation effizienter gestalten läßt. So könnte ein Beobachter die Notifikation zeitweise unterbrechen, um eine Folge von Änderungen am Modell vorzunehmen. Der Controller puffert die vorzunehmenden Aktualisierungen solange, bis die Änderungsfolge abgeschlossen ist.

Abbildungsverzeichnis

Literaturverzeichnis

[AHM98] Alfert, Klaus, Willi Hasselbring und Arnulf Mester: *Skriptum zur Vorlesung: Programmierkurs UML und Java (im SoSe 98)*. Technischer Bericht Universitat Dortmund, 1998. `http://www4.cs.uni-dortmund.de/PK-UML-Java`.

[Ban95] Bank, David: *The Java Saga*. Technischer Bericht Wired Digital, 1995. `http://www.wired.com/wired/archive/3.12/java.saga.html`.

[DK94] Därr, Dirk und Udo Kelter: *Rapid prototyping of graphical editors in an open SDE*, 1994.

[Dob96] Doberkat, Ernst-Erich: *A Language for Specifying Hyperdocuments*. Software – Concepts and Tool, (17):163–172, 1996.

[Dör98] Dörr, Andreas: *Geist aus dem Netz*. iX – Magazin für professionelle Informationstechnik, (11):166, November 1998.

[Ebe94] Eberts, Ray E.: *User interface design*. Prentice Hall, Englewood Cliff, New Jersey, 1994.

[ECM] European Computer Manufacturers Association: *Portable Common Tool Environment - Abstract Specification (Standard ECMA-149)*.

[Eng97] Englander, Robert: *Developing Java beans*. The Java series. O'Reilly, 1st ed. Auflage, 1997.

[ESP85] ESPRIT: *Esprit '84:Status report of ongoing work*, 1985.

[Fer98] Ferber, Rafael: *Philosophische Grundbegriffe – Eine Einführung*. Beck, 1998.

[Fla98] Flanagan, David: *Java in a nutshell*. O'Reilly, 2. deutsche Ausgabe Auflage, 1998.

[Gam92] Gamma, Erich: *Objektorientiere Software-Entwicklung am Beispiel von ET++*. Springer, Berlin, 1992.

[GHJV96] Gamma, Erich, Richard Helm, Ralph Johnson und John Vlissides: *Design patterns*. Professional computing. Addison-Wesley, 1996.

[GJM91] GHEZZI, CARLO, MHEDI JAZAYERI und DINO MANDRIOLI: *Software Engineering*. Prentice Hall, 1991.

[Hen88] HENDLER, JAMES A.: *Expert systems – the user interface*. Ablex, Norwood, New Jersey, 1988.

[Jav] JAVASOFT INC.: *JavaTM Platform 1.2 API Specification*, 1.2.1 Auflage. `http://java.sun.com/products/jdk/1.2/docs/api/index.html`.

[Kel89] KELTER, U.: *Gruppen-Transaktionen vs. gruppenorientierte Zugriffsrechte*. In: PAUL, M. (Herausgeber): *Proceedings 19. Jahrestagung der GI*, Informatik-Fachberichte, Seiten 287–300. Springer-Verlag, Oktober 1989.

[Kel98] KELTER, UDO: *Einführung in H-PCTE*. Universität Siegen, Fachbereich Informatik, Lehrstuhl für Praktische Informatik, 1998.

[KLLMM93] KNUDSEN, J.L., M LÖFGREN, O. LEHERMANN-MADSEN und B. MAGNUSSON: *Object oriented environments: the Mjølner approach*. object oriented series. Prentice Hall, 1993.

[Mey98] MEYER, ANDRÉ: *JFC 1.1 mit Java Swing 1.0*. Addison-Wesley-Verlag, 1998.

[OHE95] ORFALI, ROBERT, DAN HARKEY und JERI EDWARDS: *The Essential Distributed Objects Survival Guide*. Wiley, 1995.

[PCT96] UNIVERSITÄT SIEGEN, FACHBEREICH INFORMATIK, LEHRSTUHL FÜR PRAKTISCHE INFORMATIK: *H-PCTE vs. PCTE*, 1996.

[Pla96] PLATZ, DIRK: *Die Benachrichtigungsmechanismen von PCTE und H-PCTE*, 1996.

[Rog98] ROGAT, AXEL: *Objektorientiertes Programmieren mit C++ und JAVA*. Technischer Bericht 1998. `http://wmwap1.math.uni-wuppertal.de/~axel/skripte/oop/oop1.html`.

[RT89] REPS, THOMAS W. und TIM TEITELBAUM: *The synthezizer generator – a system for constructing language based editors*. Springer, New York, 1989.

[SF98] SCOTT, KANDAL und MARTIN FOWLER: *UML Konzentriert*. Addison-Wesley, 1998.

[Shn87] SHNEIDERMAN, BEN: *Designing the user interface – strategies for effective human computer interaction*. Addison-Wesley, Reading, Massachusetts, 1987.

[SIG] SIGCHI. Informationen der Special Interest Group der ACM zum Thema Computer Human Interface, `http://www.acm.org/sigchi/`.

[Smo92] SMOLKA, G.: *Feature-constraint logic for unification grammars.* Journal of Logic Programming, (12):51–87, 1992.

[Sun] SUN MICROSYSTEMS: *Products and APIs.* `http://www.java.sun.com/products/index.html`.

[Sun98a] SUN MICROSYSTEMS: *The Swing Connection*, 1998. `http://www.java.sun.com/products/jfc/tsc/index.html`.

[Sun98b] SUN MICROSYSTEMS: *Where did Java Technologie come from?*, 1998. `http://www.sun.com/java/comefrom.jhtml`.

[Tan92] TANENBAUM, ANDREW S.: *Modern Operating Systems.* Prentice Hall, 1992.

[Tho93] THOMPSON, KAREN: *PCTE - An Overview.* HMSO, London, 1993.

[TSMB95] TEUFEL, S., C. SAUTER, T. MÜHLHERR und K. BAUKNECHT: *Computerunterstützung für die Gruppenarbeit.* 1995.

[Weg90] WEGNER, PETER: *Concepts and Paradigms of Object-Oriented Programming.* Technischer Bericht Juni 1990.

[Wei92] WEINAND, ANDRE: *Objektorientierte Architektur für graphische Benutzeroberflächen.* Springer, Berlin, 1992.

[Wil95] WILLIAMS, TOM: *Beyond the toaster model.* Computer Design, nov 1995. `http://www.computer-design.com/Editorial/1995/11/Directions/beyond.html%`.

[WJ93] WAKEMAN, LOIS und JONATHAN JOWETT: *PCTE - The standard for open repositories.* Prentice Hall International (UK) Ltd., 1993.