Universität Hamburg
Fachbereich Informatik
Arbeitsbereich Datenbanken und Informationssysteme

Diplomarbeit

Objektorientierter Entwurf und Realisierung einer Benutzerschnittstelle für ein Internet-Produkt-Informationssystem

Betreut von:

Prof. Dr. Florian Matthes
Arbeitsbereich Softwaresysteme
Technische Universität Hamburg-Harburg

Prof. Dr. Christian Freksa
Arbeitsbereich Wissens- und
Sprachverarbeitung
Universität Hamburg

vorgelegt von:

Lan Zhang
Pestalozzistraße 3
69181 Leimen
Tel: 06224 – 55272

Leimen, den 01.November 1999

INHALTSVERZEICHNIS

1 MOTIVATION UND EINLEITUNG 7

1.1 MOTIVATION 7
1.2 ZIELSETZUNG 9
1.3 GLIEDERUNG 10

2 OBJEKTORIENTIERTE SOFTWAREENTWICKLUNG MIT UML 12

2.1 ALLGEMEINES VORGEHENSMODELL 13
2.1.1 Analyse 15
2.1.2 Entwurf 16
2.1.3 Implementierung 17
2.2 KOMPONENTENBILDUNG 17

3 TECHNISCHE GRUNDLAGEN 19

3.1 PIA SYSTEMARCHITEKTUR 19
3.2 DIE ZUSAMMENENTWICKLUNG DER DATENBANKSYSTEME 20
3.3 XML 21
3.3.1 XML-Dokumente und DTDs 22
3.3.2 Ein XML Beispiel in PIA 23
3.3.3 Parsen und Durchblättern von XML-Dokumenten 24
3.4 RMI SYSTEMARCHITEKTUR 25
3.5 DIE PROGRAMMIERSPRACHE JAVA 27
3.5.1 Schnittstellen 27
3.5.2 Polymorphismus 29
3.5.3 GUI-Programmierung mit Java 30
3.5.3.1 Java Foundation Classes (JFC) und Swing 30
3.5.3.2 Das Model-View-Controller (MVC) Konzept in Swing 33
3.5.3.3 Table mit MVC 35

4 ANALYSE 36

4.1 ANWENDUNGSFÄLLE FÜR KUNDEN 38
4.2 ANWENDUNGSFÄLLE FÜR REDAKTEURE 40
4.3 ANWENDUNGSFÄLLE FÜR ANBIETER 42

5 ENTWURF 44

5.1 ENTWURF EINES GENERISCHEN OBJEKT-MODELLS 45
5.2 ENTWURF DER BENUTZERSCHNITTSTELLE 49
5.2.1 „UML“-Klassendiagramme 50
5.2.2 Entwurfsmuster 50
5.2.3 Entwurf einer interaktiven Benutzerschnittstelle 53
5.3 BEWERTUNG DER ENTWURFSPHASE 69

6 IMPLEMENTATIONSPHASE 70

6.1 INTERAKTIONSDIAGRAMM 70
6.2 REALISIERUNG DER ANWENDUNGSFÄLLE 71
6.2.1 Anmelden eines Benutzers 71
6.2.2 Zusammenstellen einer Suchanfrage 74
6.2.3 Editieren eines Domänenobjektes 83
6.2.4 Editieren eines Attributs 89
6.2.5 Editieren eines Produkts 90
6.3 BEWERTUNG DER IMPLEMENTATIONSPHASE 93

7 ZUSAMMENFASSUNG UND AUSBLICK 94

LITERATURVERZEICHNIS 96

Abbildungsverzeichnis

Abbildung 2-1. Grobes Vorgehensmodell für die objektorientierte Softwareentwicklung [Nüttgens99] 14
Abbildung 2-2. Vorgehensmodell des Rational Unified Processes für die objektorientierte Softwareentwicklung [Kruchten99] 15
Abbildung 3-1: Java und seine Verwendung in Anwendungsarchtiekturen des Web 20
Abbildung 3-2: RMI Systemarchitektur 25
Abbildung 3-3. Mehrfachvererbung in Java mit Schnittstellen 28
Abbildung 3-4. Quelltextauszug aus der Schnittstelle DomainChangeListener 28
Abbildung 3-5. Quelltextauszug aus der Klasse DomainPanel 29
Abbildung 3-6. Schwergewichtige und leichtgewichtige Komponenten in Java 30
Abbildung 3-7. JFC in Überblick 31
Abbildung 3-8. Swing Klassenhierarchie 32
Abbildung 3-9. Kommunikation in der Model-View-Controller Architektur 33
Abbildung 3-10. Das Modell-Delegierter Konzept in Swing 34
Abbildung 4-1. Anwendungsfälle in Überblick 38
Abbildung 4-2. Anwendungsfälle für Kunden 39
Abbildung 4-3. Benutzerschnittstelle für Redakteure 41
Abbildung 4-4. Spezialisierung des Anwendungsfalls "Domänen verwalten" 41
Abbildung 4-5. Benutzerschnittstelle für Anbieter 43
Abbildung 5-1. Systemgrobaufbau 44
Abbildung 5-2. PIA Business-Klassen in Überblick 46
Abbildung 5-3: Struktur des Singletonmusters 51
Abbildung 5-4: Struktur des MVC Musters 52
Abbildung 5-5: Struktur des Musters Template Methode 53
Abbildung 5-6. Pakete in PIA Frondend System 54
Abbildung 5-7. Die Klassen IconToogleButtonList, ContentPane und Statusbar im Paket mainframe55
Abbildung 5-8. Klassen im Paket *mainframe* 56
Abbildung 5-9. DomainPanel und DomainEdiotorPanel 57
Abbildung 5-10. DomainEditor und seine Unterklassen 58
Abbildung 5-11. Ein *PredicateSelector* als Bestandteil der Registerkarten 59
Abbildung 5-12. Editor für Werte einer hierarchischen Domäne 60
Abbildung 5-13. Ein *UnitsEditor als Bestandteil der Registerkarte* in einem *NumericDomainEdiorPanel* 60
Abbildung 5-14. Ein *LanguageSelector* für einen *FulltextDomainEditor* 61
Abbildung 5-15. Das Paket *pia.maintenance.attribute* 62
Abbildung 5-16. *AttributeEditor* 62
Abbildung 5-17. *AlternativeEditor* und seine Bestandteile 63
Abbildung 5-18. Der Einsatz vom Entwurfsmuster Vermittler und Singleton im Paket pia.gui 65
Abbildung 5-19. Klassen im Paket pia.queryresult und ihre Beziehungen zueinander 66
Abbildung 5-20. Das Paket item 68
Abbildung 6-1. Singleton - der *ProfileManager* 72
Abbildung 6-2. Quelltextauszug der Klasse *CustomerProfile* 73
Abbildung 6-3. Interaktionsdiagramm für den Anwendungsfall: „Benutzeranmeldung“ 74
Abbildung 6-4. Quelltextauszug aus der Klasse alternativeEditor – Teil 1 75
Abbildung 6-5. Quelltextauszug aus der Klasse AlternativeEditor - Teil 2 76
Abbildung 6-6. Interaktionsdiagramm für den Anwendungsfall: "Zusammenstellung einer Suchanfrage" 77
Abbildung 6-7. Quelltextauszug aus der Klasse AlternativeEditor – Teil 3 78
Abbildung 6-8. Quelltextauszug aus der Interface CriterionChangeListener 78
Abbildung 6-9. Quelltextauszug aus der Klasse *AlternativeDisplayer* - Teil 1 79
Abbildung 6-10. Quelltextauszug aus der Klasse *AlternativeDisplayer* – Teil 2 80
Abbildung 6-11. Quelltextauszug aus der Klasse *AlternativeDisplayer* – Teil 3 83

Abbildung 6-12. Quelltextauszug aus der Klasse *DomainPanel* - Teil 1 84
Abbildung 6-13. Quelltextauszug aus der Klasse *DomainPanel* - Teil2 85
Abbildung 6-14. Quelltextauszug aus der Klasse *DomainPanel* – Teil 3 86
Abbildung 6-15. Quelltextauszug aus der Klasse *DomainEditorPanel* – Teil 1 87
Abbildung 6-16. Quelltextauszug aus der Klasse *DomainEditorPanel* - Teil 2 88
Abbildung 6-17. Interaktionsdiagramm für den Anwendungsfall „Domänen editieren" 89
Abbildung 6-18. Interaktionsdiagramm für den Anwendungsfall „Attribut Editieren" 90
Abbildung 6-19. Quelltextauszug aus der Klasse *ValueEditorManager* 92
Abbildung 6-20. Interaktionsdiagramm für den Anwendungsfall „Produkt editieren" 93

Zusammenfassung

Immer mehr moderne Unternehmen haben die wichtige Bedeutung von interaktiven Informationssystemen zur Erreichung ihrer strategischen Ziele erkannt und versuchen, über solche Informationssysteme zusätzliche Kommunikationskanäle zu potentiellen Kunden aufzubauen. Es besteht jedoch die Gefahr, daß Informationen von verschiedenen Anbietern unterschiedlich strukturiert sind und diese Informationen den Kunden nicht effektiv vermittelt werden können. Dadurch können die Informationen nicht zufriedenstellend verwendet werden und die Vorteile von Informationssystemen nicht ausgenutzt werden. Ziel des Projektes PIA (Personal Information Assistant) ist, ein interaktives und generisches Informationssystem zu entwerfen und zu realisieren, das als aktiver Assistent zwischen Informationsnutzern und Informationsanbietern agiert und vor allen Dingen den Informationsnutzern beim Suchen nach Informationen hilft, so daß die Konversation zwischen Informationsnutzern und Informationsanbietern effektiv gestaltet werden kann.
Ziel der vorliegenden Arbeit war die Konzeption und die prototypische Entwicklung der Benutzerschnittstelle für das PIA System. Die graphische Benutzeroberfläche spielt in PIA eine zentrale Rolle für den Erfolg und die Akzeptanz des Systems, weil ein Benutzer nur über die Benutzerschnittstelle Informationen anfragen, publizieren oder strukturieren kann und somit die Benutzeroberfläche die wichtigste Schnittstelle für die Benutzer zum System bildet. Im Rahmen dieser Arbeit wird das erste Inkrement der Benutzerschnittstelle für das PIA System entworfen und implementiert. Dem objektorientierten Ansatz folgend wird der Entwicklungsprozeß in drei Phasen – Analyse, Entwurf und Implementation geteilt, wobei sich diese Arbeit schwerpunktmäßig mit der Entwurf- und der Implementationsphase befaßt. In der Entwurfsphase wird die objektorientierte Modellierungstechnik eingesetzt, ein Objektmodell wird ausgearbeitet und in UML-Notation erfaßt. Basierend auf diesem Objektmodell wird die Benutzerschnittstelle mit der Programmiersprache Java realisiert. Bei der Realisierung werden verschiedene objektorientierte Techniken wie Entwurfsmuster und Frameworks eingesetzt. Das Ergebnis der Arbeit ist ein funktionsfähiges Frontendsystem, das über die in dieser Arbeit beschriebenen Funktionalitäten verfügt und somit einem Benutzer erlaubt, mit dem PIA System zu interagieren, um die für ihn nützlichen Informationen am effektivsten zu finden.

1 Motivation und Einleitung

Immer mehr moderne Unternehmen haben die Bedeutung von interaktiven Informationssystemen zur Erreichung ihrer strategischen Ziele erkannt und versuchen, über solche Informationssysteme zusätzliche Kommunikationskanäle zu potentiellen Kunden aufzubauen. Dabei können sich die Informationssysteme auf unterschiedliche Formen von Medien stützen, so daß sie z.B. sowohl online über das World Wide Web erreichbar sind, als auch offline auf CD-ROMs verteilt werden können. Ein erfolgreich eingeführtes Informationssystem verspricht eine direkte und effiziente Kommunikation zwischen Anbietern und Kunden und hilft dabei, eine langfristige Beziehung zwischen den beiden Parteien aufzubauen. Interaktive Informationssysteme bilden außerdem einen guten Kompromiß zwischen der traditionellen 1:1 Kunden-Verkäufer-Konversation und den Broadcast-Medien, wie Werbemitteln, Produktkatalogen usw. Einerseits bietet ein Informationssystem im Vergleich zur 1:1 Konversation eine höhere Verfügbarkeit und erfordert weniger Peronalaufwand, und andererseits erlaubt der Einsatz von Informationssystemen eine wesentlich höhere Interaktivität und mehr Personalisierungs- und Gruppierungsmöglichkeiten als die Broadcast-Medien.

1.1 Motivation

Was versteht man unter einem interaktiven Informationssystem? Ein interaktives Informationssystem agiert als aktiver Mittler zwischen Informationslieferanten und Informationssuchenden und dient dem effizienten Austausch der Informationen zwischen den beiden Parteien. Dabei versucht ein Informationssystem, so weit wie möglich zu verhindern, daß die Informationsanbieter und die Informationsnutzer „aneinander vorbeilaufen“ – das sog. „near miss“. Der Grund für ein solches „near miss“ liegt darin, daß Informationsanbieter ihre eigenen Vorstellungen von ihren Angeboten haben und versuchen, nach diesen Vorstellungen die Informationen am Käufermarkt zu vermitteln. Aber die Vorstellungen der Informationsanbieter entsprechen u.U. nicht genau denen der Informationsnutzer, die das Ganze aus einem anderen Blickwinkel betrachten. Diese Informationsnutzer – die Kunden auf dem Käufermarkt – haben ihrerseits wiederum eigene Vorstellungen von den Produkten, die sie suchen. Ein weiterer Grund für ein „near miss“ ist, daß die Informationen, die von unterschiedlichen Anbietern kommen, inhomogen und zugleich nicht einheitlich strukturiert sind, was u.a. daran liegt, daß kein anerkannter

Standard für Produktinformationen vorhanden ist. Um das Problem zu vermeiden oder zumindest zu entschärfen, versucht eine dritte Klasse von Benutzern eines Informationssystems – Redakteure – die inhomogenen und unstrukturierten Informationen verschiedener Anbieter zu formatieren und zu strukturieren. Dadurch entsteht eine einheitliche Struktur für die Informationen, die auch den Anforderungen der Informationsnutzer besser entsprechen. Die Rolle eines Redakteurs kann sowohl von einer Person, als auch von einem System eingenommen werden.
In dieser Arbeit wird zunächst ein interaktives, generisches Informationssystem – PIA (*Personal Information Assistant*) vorgestellt, das folgende Eigenschaften besitzt:

- Unterstützung der autonomen Informationslieferanten
- Unterstützung heterogener Sammlungen von Dokumenten
- Zulassen unscharfer Kategorien von Dokumenten
- Interaktive Benutzerschnittstelle
- Fließender Übergang zwischen Anbieter- und Käufermarkt:
    - auch Kunden publizieren Suchaufträge
    - auch Anbieter suchen Dokumente.

PIA agiert als Assistent für interaktive online oder offline Produktkataloge und hilft den Kunden beim Suchen nach Artikeln (Produkte, Dienstleistungen oder Informationsressourcen), die genau ihren persönlichen Interessen und Präferenzen entsprechen. Dabei betont der Name PIA die aktive Rolle des Systems. PIA unterstützt außerdem autonome Informationslieferanten, so daß ein Informationsanbieter seine Informationen selbständig ins PIA System einfügen kann, die ohne Überarbeitung durch Redakteure sofort für Informationsnutzer sichtbar werden. Ein Katalog in PIA ist eine Sammlung von Dokumenten. Das PIA System erlaubt eine heterogene Sammlung von Katalogen. D.h. die Dokumente in PIA, die Informationen beschreiben, dürfen in unterschiedlichen Formaten vorliegen – z.B. im HTML- oder XML-Format – und eine unterschiedliche Struktur besitzen. Die Dokumente müssen keiner bestimmten Kategorie zugeordnet werden. Vielmehr ist „Kategorie" ein normales Attribut, mit dem ein Dokument beschrieben werden kann. Mit dem PIA System wird ein fließender Übergang zwischen Kunden und Anbietern geschaffen, weil ein PIA-Kunde neben dem Suchen nach Informationen auch seine eigenen Suchaufträge im System publizieren kann, während ein Anbieter, genau wie ein Kunde, nach Dokumenten suchen kann. Der Umgang mit PIA führt zu einem wertvollen gemeinsamen Lernprozeß, der von den interagierenden Benutzern geführt wird und hilft, das „near miss" zwischen ihnen zu vermeiden.
Als Suchmechanismus setzt PIA die sog. Fuzzy-Suchmethoden ein, weil die herkömmlichen Technologien in dieser Hinsicht eine unzufriedenstellende Lösung liefern. Eine Volltext-Suchmaschine arbeitet auf der syntaktischen Ebene und prüft meistens auf die textuelle Wertgleichheit, ohne die Bedeutung und insbesondere die Struktur des Textes zu interpretieren. Bei herkömmlichen Suchdienstanbietern, die auf der Basis einer relationalen Datenbank arbeiten, werden nur exakte boolesche Anfragen erlaubt. Hierarchische Kataloge

schließlich erzwingen eine statische Navigationsstruktur für alle Kunden, was den Entscheidungsweg des einzelnen Kunden sehr einschränkt. Fuzzy-Logik hingegen liefert einen Lösungsansatz für die Formulierung unscharfer Anfragen und ihre Auswertung [Biewer97]. Durch den Einsatz von Fuzzy-Logik können auch unscharfe Suchanfragen formuliert werden, so daß die Suche sich nicht nur auf starre syntaktische Gleichheit beschränkt, sondern auch die Semantik, die durch die Struktur der Dokumente ausgedrückt wird, in Betracht zieht. Dadurch daß der Schwerpunkt bei der Suche auf die Semantik, nicht auf die Syntax gelegt wird, kann ein für die Kunden zufriedenstellendes Ergebnis geliefert werden.

In einem Informationssystem wie PIA spielt die graphische Benutzeroberfläche eine zentrale Rolle für den Erfolg und die Akzeptanz des Systems, weil ein Benutzer über die Benutzerschnittstelle Informationen anfragen, publizieren oder strukturieren kann und somit die Benutzeroberfläche die wichtigste Schnittstelle für die Benutzer zum System bildet. Mit Hilfe dieser Benutzeroberfläche kann ein Kunde seine Suchanfragen zusammenstellen und diese Suchanfragen nach und nach spezifizieren. Dafür bietet die Benutzerschnittstelle von PIA System nicht nur möglichst viele Tooltips als Suchhilfe, sondern auch möglich viele Informationen zum Suchergebnis für die zu editierende Zwischensuchanfrage, wie z.B. wie viele Produkte im System der aktuellen Suchanfrage entsprechen, oder welche weiteren Eigenschaften bei den meisten dieser Produkte vorhanden sind. Das Suchergebnis wird einem Benutzer ebenfalls auf der Benutzeroberfläche präsentiert. Um das Suchergebnis möglichst übersichtlich darzustellen, bietet die Benutzerschnittstelle von PIA verschiedene Formatierungs- und Sortierungsmöglichkeiten. Einem Anbieter stehen zwei Möglichkeiten zur Auswahl, um seine Produktdaten zu pflegen. Die eine arbeitet über eine Importschnittstelle, mit deren Hilfe die vorhandenen Produktdaten eines Anbieters in einem vom Anbieter spezifizierten Format ins PIA System aufgenommen werden können. Dabei werden diese Dokumente in ein vom PIA System vorgesehenen Format umgewandelt. Die andere Möglichkeit ist über die Benutzeroberfläche, mit deren Hilfe ein Anbieter manuell und interaktiv die einzelnen Produktdaten pflegen kann. Die Strukturierung der Informationen durch Redakteure geschieht ebenfalls über die Benutzerschnittstelle von PIA. Insgesamt wird bei dem Entwurf und der Implementation des PIA Systems viel Wert darauf gelegt, die Benutzerschnittstelle möglichst benutzerfreundlich und leicht bedienbar zu gestalten.

1.2 Zielsetzung

PIA System beruht auf einer modernen Client-Server Architektur - der sog. drei-Schicht-Architektur (*three-tier-architecture*). Demnach untergliedert sich das System in folgende Schichten:

- Datenbankschicht
- Applikationsschicht
- Präsentationsschicht.

Diese Arbeit wird auf die Präsentationsschicht, über die die Interaktionen zwischen den Benutzern und dem System stattfinden, fokussiert.
Es wird in PIA zwischen den folgenden 3 Gruppen von Benutzern unterschieden:

- Kunden
- Anbieter
- Redakteure.

Jeder Benutzer hat seine eigenen Ziele hinsichtlich der Arbeit mit dem PIA System und wird einer dieser Gruppen zugeordnet. Ziel der Arbeit ist die Konzeption und die Realisierung einer modernen, komfortablen und leicht handhabbaren Bedienoberfläche für das PIA System, damit Benutzer jeder Gruppe ihr Vorhaben mit PIA so einfach wie möglich umsetzen können. Diese Benutzeroberfläche soll von außen leicht konfigurierbar sein, damit für jede Benutzergruppe die geeignete Oberfläche zur Verfügung gestellt werden kann. Außerdem soll die Benutzeroberfläche plattformunabhängig sein, damit das PIA System sowohl online über Internet als auch offline auf ein CD-ROM zur Verfügung gestellt werden kann. In dieser Arbeit soll der objektorientierte Ansatz für Softwareentwicklung verfolgt werden und die Problematik in verschiedenen Phasen des Softwareentwicklungsprozesses durchgängig objektorientiert betrachtet werden. Die erstellte Benutzerschnittstelle soll die in dem vorigen Abschnitt genannten besonderen Eigenschaften wie z.B. Fuzzy-Anfragen und unscharfe Kategorien der Dokumente von PIA vollkommen unterstützen, so daß einerseits ein Kunde unscharfe Suchanfragen nach Produkten formulieren kann, andererseits ein Anbieter seine Produktdaten ins System einfügen kann, ohne eine Kategorie für das jeweilige Produkt festlegen zu müssen.

1.3 Gliederung

Das in dieser Arbeit verwirklichte erste Inkrement eines umfangreichen Frontends für das PIA System wurde in der objektorientierten Sprache Java implementiert und setzt auf dem in [Matthes99] entwickelten Datenmodell auf.
Nach der Einleitung wird im zweiten Kapitel die Vorgehensweise für die Entwicklung eines objektorientierten Softwaresystems mit UML (*Unified Modelling Language*) beschrieben. Nach dem objektorientierten Ansatz wird der Softwareentwicklungsprozeß in drei Phasen unterteilt: Analyse, Design und Implementation. UML ist eine Sprache und Notation zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen von Softwaresystemen und ist als de-facto Standard für objektorientierte Analyse und Design anerkannt [Nüttgens99].

Im dritten Kapitel werden die technischen Grundlagen für die Realisierung des PIA Systems beschrieben. Es beginnt mit der Vorstellung von XML (*eXtensible Markup Language*) - dem neuen Standardformat für das Internet [DuCharme99]. Die Vorteile und die Einsatzmöglichkeiten von XML werden beispielhaft geschildert. Danach wird RMI (*Remote Method Invocation*), das in PIA für die Client/Server-Kommunikation eingesetzt wird, kurz erläutert. Das Java-Toolkit für graphische Oberflächen – Swing bildet die Grundlage für die Realisierung der Benutzeroberfläche in PIA und wird daher schwerpunktmäßig behandelt. Die Basisarchitektur des Swing-Frameworks sowie der Aufbau der Swing-Komponenten werden ausführlich beschrieben.

Um die Systemanforderungen zu ermitteln, wird im vierten Kapitel eine Analyse für die drei Gruppen von Benutzern durchgeführt. Das Ergebnis der Analyse wird in Form von Anwendungsfällen ausgedrückt und dient als Grundlage für den Entwurf und die Implementation.

Aufbauend auf den Anwendungsfällen aus dem vierten Kapitel wird im fünften Kapitel ein Objektmodell entworfen. Die Klassen, die in der Analysephase ausgearbeitet wurden, werden in UML Klassendiagrammen dargestellt. Außerdem werden die Entwurfsmuster, die beim Entwurf und bei der Implementation verwendet werden, in diesem Abschnitt vorgestellt, und deren Einsatz in PIA wird erläutert.

Im sechsten Kapitel wird die Implementation der Benutzeroberfläche detailliert beschrieben, insbesondere werden die Interaktionen zwischen den Objekten zur Laufzeit dargelegt und die Realisierung der Anwendungsfälle vorgestellt. Die Anwendung von Entwurfsmustern in PIA wird ebenfalls anhand von Beispielen erläutert.

Den Schluß der Arbeit bildet eine Bewertung der gesamten Arbeit sowie ein Ausblick darüber, in welcher Hinsicht das System in Zukunft noch verbessert und erweitert werden kann.

2 Objektorientierte Softwareentwicklung mit UML

Traditionelles Software-Engineering stellt zwar einen notwendigen Ansatz dar, um verschiedene Probleme der Softwareproduktion in den Griff zu bekommen. Jedoch haben sich in der praktischen Ausführung einige Schwächen gezeigt [Burkhardt97]. Nach [Hesse94] ist das gesamte Gebiet des Softwareentwicklungsprozesses einem "tiefgreifenden Wandel" unterworfen. Gemeint ist dabei die Objekttechnologie, die sowohl auf der Implementierungs- als auch auf der Entwurfsebene inzwischen bewährt ist und derzeit die neueren Entwicklungen der Bereiche Analyse und Modellierung bestimmt. Die objektorientierte Modellierung ermöglicht zum einen den nahtlosen Übergang von einer Phase in die nächste, zum anderen wird aber auch - eines der Hauptmerkmale der Objekttechnologie - die Wiederverwendung vorhandener Konzepte und Implementierungen von Anfang an voll unterstützt. Somit können Zyklen - sie werden im Top-Down-Entwurf beim traditionellen Software-Engineering als Nachteil empfunden - zum Nutzen des Entwicklers eingesetzt und dadurch in einen Vorteil verwandelt werden.
Objektorientierte Softwareentwicklung ist ein iterativer und inkrementeller Prozeß, in dem die Software nicht mit einem großen Schlag am Ende des Projekts herausgegeben, sondern in Teilen entwickelt und freigegeben wird. Kennzeichnend für den iterativen Ansatz ist die Wiederholung einzelner Tätigkeiten eines ansonsten konkret vorgegebenen Ablaufs. Während dieser Ansatz den Nachteil aufweisen könnte, daß die Wirkung einer Iteration nicht unbedingt erkennbar und damit auch schwer nachprüfbar ist, ist mit dem inkrementellen Ansatz genau dieser Nachteil des iterativen Ansatzes aufgehoben. Dabei ist die Idee, pro Durchlauf ein gewisses, gegebenenfalls näher zu spezifizierendes Inkrement zu fordern.
Die Entwicklung hochwertiger Software wird immer komplexer und aufwendiger. Der Umstieg von alphanumerischen Benutzungsmodellen zu den ereignisorientierten Modellen, die grafischen Benutzerschnittstellen (*GUIs*) zugrunde liegen, die Einführung mehrschichtiger Client/Server-Architekturen, Internet usw. – die Berücksichtigung dieser Themen löste einen beachtlichen Komplexitätssprung aus. Es ist daher sinnvoll, sich nach geeigneten Methoden umzusehen, die diese Komplexität beherrschbar machen.
In dieser Arbeit wird dem Vorgehensmodell der UML (*Unified Modelling Language*) gefolgt. Die UML ist eine Sprache und Notation zur Spezifikation, Konstruktion, Visualisierung und Dokumentation von Modellen von Softwaresystemen und ist von Grady Booch, Ivar Jacobson und James Rumbaugh entwickelt worden [Fowler97]. Die UML berücksichtigt die gestiegenen Anforderungen bezüglich der Komplexität heutiger Softwaresysteme, deckt ein breites Spektrum von Anwendungsgebieten ab und eignet sich für konkurrierende, verteilte, zeitkritische, sozial eingebettete Systeme uvm [Oestereich97].

Im folgenden soll ein auf die UML zugeschnittener Softwareentwicklungsprozeß vorgestellt werden.

2.1 Allgemeines Vorgehensmodell

Ein Prozeßmodell für die Softwareentwicklung zu entwickeln, das auf alle realen Prozesse paßt, ist unmöglich. Deshalb wurde der Prozeß für die Softwareentwicklung mit Hilfe von UML als solcher zunächst ausgeklammert. Jedoch ist eine Softwareentwicklung ohne Prozeß unprofessionell. In der UML wird deshalb versucht, ein Rahmenwerk für Prozesse zur Verfügung zu stellen. Rahmenwerk bedeutet, daß alle Vorgehensweisen und Notationen von konkreten Prozessen mit der Notation darstellbar sind. Wie in [Fowler97a] beschrieben wurde, folgt die objektorientierte Systementwicklung einem iterativen und inkrementellen Vorgehensmodell. Kennzeichnend für dieses Vorgehensmodell ist die Gliederung der Entwicklungsaktivitäten in kleine Einheiten sowie die koordinierte Verkettung von Aktivitäten unterschiedlicher Detaillierungsebenen. In Abbildung 2.1 ist ein solches grobes Vorgehensmodell dokumentiert, welches ein mögliches Vorgehen im Rahmen der objektorientierten Systementwicklung beschreibt [Nüttgens99]. Wie aus der Abbildung zu entnehmen ist, untergliedert sich der Entwicklungsprozeß in mehrere Phasen. Begonnen wird mit der Analyse der Geschäftsprozesse, die in das zu entwickelnde System einzubetten sind. In dieser Phase werden die Anwendungsfälle (*use cases*) untersucht und ein Anwendungsfall-Modell wird erstellt. Während der Entwurfsphase werden die aus dem Anwendungsfall-Modell identifizierten Objekte in einem Klassendiagramm fixiert und die Beziehungen der Klassen untereinander beschrieben. Außerdem werden in dieser Phase die Zustands-, Sequenz- und Interaktionsdiagramme angefertigt. Es zeigt sich, daß die UML einen wesentlichen Beitrag zur iterativen und inkrementellen Entwicklung von Softwaresystemen liefert. Die Teilprozesse der objektorientierten Systementwicklung beschreiben einen Zyklus mit Rückkopplung zwischen den einzelnen Elementen des Vorgehensmodells. Treten beim objektorientierten Design Widersprüche oder Probleme auf, die nicht in dieser Phase gelöst werden können, erfolgt eine Rückdelegation zur objektorientierten Analyse. Sind die Designergebnisse freigegeben, bilden diese den Input für die objektorientierte Implementation. Die Design-Modelle werden mit Hilfe der gewählten Programmiersprache implementiert. Treten bei der Codegenerierung Probleme auf, die ihren Ursprung im Design haben, erfolgt wiederum eine Rückdelegation. Wird die entwickelte Software-Komponente für den Einsatz freigegeben, beginnt der operative Einsatz. Hier wird der Übergang von der Übersetzungszeit zur Laufzeit vollzogen. Ist eine Überarbeitung der Software-Komponente erforderlich bzw. sind weitere Software-Komponenten zu entwickeln, wird mit dem Eintritt in die objektorientierte Analyse der Zyklus erneut durchlaufen.

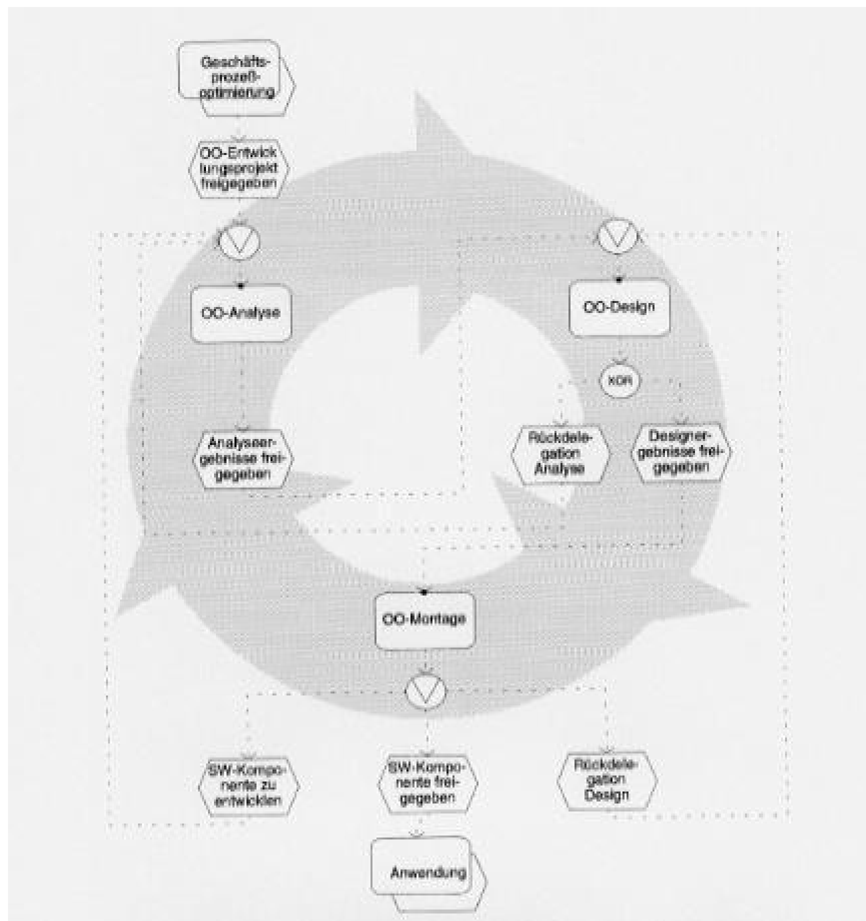


Abbildung 2-1. Grobes Vorgehensmodell für die objektorientierte Softwareentwicklung [Nüttgens99]

Im Vergleich dazu bildet der sog. *Rational Unified Process* von der Firma Rational Software den objektorientierten Softwareentwicklungsprozeß in ein zweidimensionales Modell ab. Der *Rational Unified Process* bietet eine disziplinierte Lösung für die Verteilung der Aufgaben und der Verantwortungen während der Softwareentwicklung innerhalb einer Entwicklungsorganisation. Sein Ziel ist die Sicherstellung der Produktion von Software mit hoher Qualität, die die Anforderungen von Endbenutzer erfüllt und gleichzeitig mit einem akzeptablen Zeit- und Kostenaufwand zu realisieren ist. Die folgende Abbildung (vgl. Abbildung 2.2) zeigt die Gesamtarchitektur des *Rational Unified Processes*. Der Prozeß hat zwei Dimensionen:

- Die horizontale Achse repräsentiert Zeit und zeigt den Lebenszyklen-Aspekt von des Prozesses.
- Die vertikale Achse repräsentiert die Kern-Prozeß-Workflows, die die einzelnen Aktivitäten während der Softwareentwicklung nach ihrer logischen Zusammengehörigkeit gruppiert.

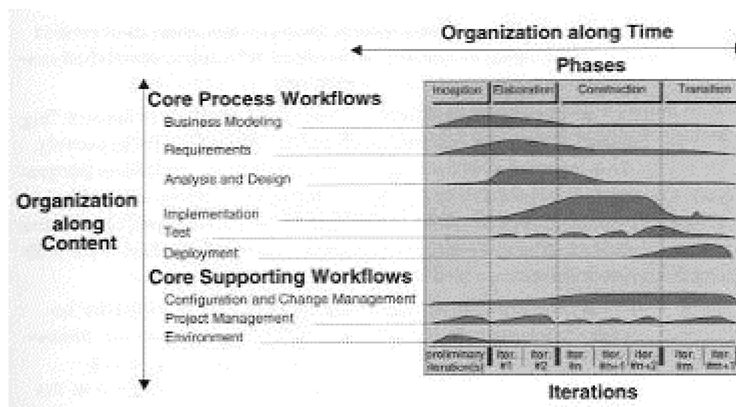


Abbildung 2-2. Vorgehensmodell des Rational Unified Processes für die objektorientierte Softwareentwicklung [Kruchten99]

Die erste Dimension repräsentiert den dynamischen Aspekt des Prozesses, während die zweite Dimension den statischen Aspekt des Prozesses repräsentiert. Näheres zu dem Thema *Rational Unified Process* findet sich in [Kruchten99].
Im folgenden werden die Teilprozesse des in dieser Arbeit verfolgten Vorgehensmodells, das dem in diesem Abschnitt beschriebenen ersten Modell eher entspricht, auf einer detaillierteren Abstraktionsstufe beschrieben.

2.1.1 Analyse

Innerhalb dieser Entwicklungsphase werden vordergründig die einzelnen softwarerelevanten Arbeitsflüsse untersucht und in typischen Anwendungsfällen beschrieben. Die Anwendungsfälle wurden von I. Jacobson eingeführt [Jacobson92]. In [Fowler97a] wird ein Anwendungsfall als eine typische Interaktion zwischen dem Anwendungssystem und dem Benutzer definiert, die notwendig ist, um einen Arbeitsgang durchzuführen. Die Benutzer werden als Akteure bezeichnet. Die Beschränkung der Akteure auf Menschen kann verallgemeinernd auch aufgehoben werden, und die Akteure können auch andere Systeme sein. Handelt es sich bei den Akteuren um Personen, so müssen diese nach ihrem Rollenverhalten im Bezug auf das System und nicht nach ihrer Identität unterschieden werden.
Anwendungsfälle sind textuelle Beschreibungen von Ausschnitten aus dem Geschäftsprozeßmodell. Diese skizzieren den grundsätzlichen Ablauf eines Geschäftsvorfalls. Sie können außerdem später in der Entwurfsphase durch Sequenz- und Aktivitätsdiagramme weiter detailliert werden. Alle Anwendungsfälle zusammen bilden ein Modell, das das Verhalten des Gesamtsystems beschreibt. Die Zusammenhänge der verschiedenen Anwendungsfälle werden in einem Anwendungsfalldiagramm dargestellt.

Innerhalb eines Anwendungfalldiagramms können die Anwendungsfälle durch Beziehungen verbunden sein. Typische Relationen zwischen den Anwendungsfällen sind die „*uses*“ („Wiederverwendung“) und „*extends*“ („Spezialisierung“) Beziehung. Die „*uses*“ Beziehung wird verwendet, wenn das gleiche Stück Anwendungsfallsbeschreibung in verschiedenen Anwendungsfällen vorkommt. Um dies zu vermeiden, wird der entsprechende Teil separiert und mit einer „*uses*“ Beziehung in die andere Anwendungsfallbeschreibung wieder eingebunden. Die „*extends*“ Beziehung wird bei Anwendungsfällen benutzt, die einem anderen Anwendungsfall ähnlich sind, aber noch ein wenig mehr bewerkstelligen.

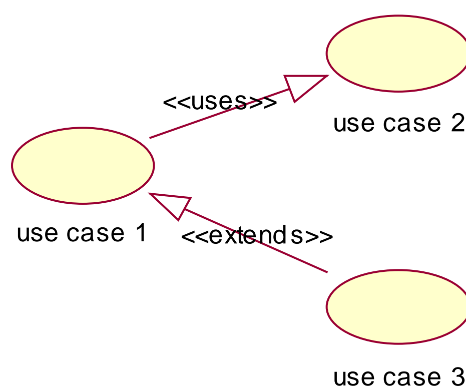


Abbildung 2-3. Anwendungsfalldiagramm: Beziehungen zwischen Anwendungsfällen

Das Ergebnis der Analysephase ist das erstellte Anwendungsfallmodell, in dem die gesamten Anwendungsfälle des Systems sowie ihre Zusammenhänge und die daran beteiligten Akteure beschrieben werden (vgl. z.B. Abbildung 5.1 im Kapitel 5).

2.1.2 Entwurf

Beim Schritt von der Analyse zum Entwurf werden die Begriffe und Sachverhalte der Anwendungswelt in ein abstraktes Modell zur Realisierung der Software überführt. Die während der Analyse erkannten Anforderungen müssen durch den Entwurf erfüllt werden. Das in dieser Phase zu entwickelnde Modell enthält

- Basiselemente (u.a. Klassen, Objekte, Attribute, Operationen, Zusicherungen),
- Statische Elemente (u.a. Assoziationen, Aggregationen, Generalisierungen bzw. Spezialisierungen, Rollen) und
- Dynamische Modellelemente (u.a. Aktivitäts- und Zustandsdiagramme, Sequenz- und Kollaborationsdiagramme).

Beim Aufbau eines solchen Modells werden zuerst die Klassen, die in der Anwendungsanalyse auftreten, identifiziert und in einem ersten Klassendiagramm fixiert. Im

Anschluß daran werden die Beziehungen zwischen den Klassen festgelegt, also Generalisierungen/Spezialisierungen, Kompositionen/Dekompositionen und allgemeine Assoziationen. Diese werden in dem Klassendiagramm durch bestimmte Notationen repräsentiert. Klassendiagramme sind der zentrale Bestandteil der UML und auch zahlreicher anderer objektorientierter Methoden (vgl. Abschnitt 5.2.1).
Wie die Klassen ermittelt werden, darüber gibt UML keine Auskunft. Hierfür gibt es andere Techniken, z.B. CRC-Karten (*Class, Responsibility and Collaboration*) oder die Substantiv-Technik [Rumbaugh91]. Die UML beschreibt lediglich die Notation und die Semantik der Klassen selbst.
Während ein Klassendiagramm die statische Struktur des Systems beschreibt, wird der Schwerpunkt in Interaktionsdiagrammen auf die zeitlichen Abläufe – d.h. die Aufrufsequenz zwischen den Objekten der Klassen gelegt. Auf eine detaillierte Ausführung der Interaktionsdiagramme muß an dieser Stelle verzichtet werden, dies kann aber in [Fowler97a] nachgelesen werden.
Spätestens seit dem Erscheinen des Buches – "*A System of Design Patterns"* der „Gang of four" [GHJV95] findet die Verwendung von Entwurfsmustern immer mehr Anerkennung. Während die UML Werkzeuge zur Unterstützung objektorientierter Entwurfsprozesse zur Verfügung stellt, betrachten Muster dagegen die Ergebnisse der Prozesse, nämlich Beispielmodelle. Muster helfen dabei, das Wissen erfahrener Software-Ingenieure zu nutzen und dokumentieren vorhandenes Entwurfswissen. Näheres bezüglich Entwurfsmuster ist im Abschnitt 5.3.1 zu finden.

2.1.3 Implementierung

Mit Implementierung bezeichnet man die Umsetzung des erarbeiteten Entwurfs in ein lauffähiges System, die sich ebenfalls auf mehreren Ebenen vollzieht. Die einzelnen Klassen werden entsprechend ihrer Designergebnisse codiert und ihre Übereinstimmung mit den Spezifikationen in Einzeltests geprüft. Auf der nächsten Ebene werden die Klassen in ein Teilsystem oder größere Pakete eingebettet und ihr Zusammenspiel getestet.

2.2 Komponentenbildung

Eine der ältesten Fragestellungen aus dem Bereich der Softwareentwicklungsmethoden ist, wie man ein großes System in kleinere Systeme zerlegen kann. Eine Idee ist, Klassen in Einheiten auf höherer Ebene zu gruppieren. In UML wird dieser Gruppierungsmechanismus als Paket (*Package*) bezeichnet. Ein Paketdiagramm zeigt die Abhängigkeiten zwischen diesen Paketen und den enthaltenen Klassen. Grundsätzlich besteht eine Abhängigkeit zwischen zwei Elementen, wenn Änderungen an der Definition eines Elements Änderungen

an der Definition des anderen Elements bedingen können. Die Abhängigkeiten zwischen Klassen existieren aus mehreren Gründen: Eine Klasse sendet eine Nachricht an eine andere; eine Klasse hat eine andere als Teil ihrer Daten; eine Klasse erwähnt eine andere in einem Operationsparameter. Wenn eine Klasse ihre Schnittstelle ändert, dann ist möglicherweise eine Nachricht, die sie empfängt, nicht mehr gültig.
Idealerweise sollten nur Änderungen an den Schnittstellen einer Klasse andere Klassen berühren. Die Kunst des Entwurfs großer Systeme beinhaltet die Minimierung von Abhängigkeiten. Auf diese Weise werden die Auswirkungen von Änderungen reduziert, so daß sie einen geringeren Gesamtaufwand nach sich ziehen.

3 Technische Grundlagen

In diesem Kapitel werden die Technologien, die in der Implementationsphase des PIA Systems eingesetzt werden, beschrieben. Zuerst wird ein Überblick über die Systemarchitektur des PIA Systems gegeben. PIA ist eine auf einer Client-Server-Architektur basierende Anwendung. Wie eingangs schon gesagt, können auf der Datenbasis die Dokumente sowohl in HTML als auch in XML vorliegen. Auch der Einsatz einer relationalen Datenbank ist nicht ausgeschlossen. In diesem Kapitel wird demonstriert, wie man Dokumente in PIA mit XML beschreiben kann und wie XML als ein wichtiges Werkzeug für den Austausch von Datenbankinformationen funktioniert.
Um die Kommunikation zwischen dem Client und dem PIA Server zu realisieren, wird RMI (*Remote Method Invocation*) eingesetzt. Es ist aber nicht der Schwerpunkt der Arbeit, dieses zu realisieren. Daher wird in dieser Arbeit nur ein kurzer Überblick über RMI gegeben.
Das Hauptziel der Arbeit ist der Entwurf und die Realisierung der Benutzerschnittstellen für das PIA System. Zur Implementation dieser Schnittstellen werden Java als Programmiersprache und insbesondere das neue Toolkit für die Interface-Programmierung - das Java Swing - verwendet. In den letzten Teilen dieses Kapitels wird das Swing-Toolkit detailliert beschrieben.

3.1 PIA Systemarchitektur

PIA ist eine Drei-Schicht-Architektur basierte Anwendung, die ihren Benutzern beim Verwalten und Suchen der Informationen in allen möglichen Formaten (HTML, XML o.ä.) helfen soll. Eine Drei-Schicht-Architektur ist eine Variante der Client-Server Architektur und besteht aus den folgenden Schichten:

- Repräsentationsschicht
- Anwendungsschicht
- Datenmodellierungsschicht.

Auf der Präsentationsschicht kann ein Benutzer einen PIA Client als eigenständige Anwendung (*standalone Application*) starten oder ein PIA-Applet in seinem Browser herunterladen. Mit Hilfe des Clients kann der Benutzer eine Anfrage formulieren und zum PIA Server, der auf der Anwendungsschicht arbeitet, schicken. Die Kommunikation zwischen einem Client und dem PIA Server wird durch das RMI von SUN Microsystems realisiert, das vollständig in der Programmiersprache Java integriert und gleichzeitig IIOP-kompatibel ist. Auf der Anwendungsschicht wird ein Fuzzy-Such-Algorithmus

implementiert, mit dessen Hilfe die möglichen Produkte, die einer (unscharfen) Anfrage entsprechen, aus der Datenbank des Systems herausgefiltert werden. RMI kann sowohl zusammen mit einem OO-Datenbanksystem als auch mit relationalen Datenbanksystemen eingesetzt werden. In der dritten Schicht, der sog. persistenten Schicht, werden alle Produktdaten als XML-Dokumente gespeichert.

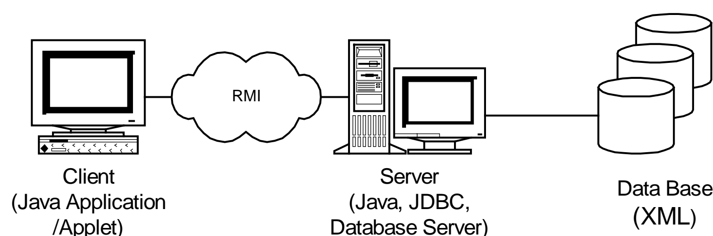


Abbildung 3-1: Java und seine Verwendung in Anwendungsarchtiekturen des Web

3.2 Die Zusammenentwicklung der Datenbanksysteme

Relationale Datenbanksysteme sind für sog. Online Transaktionsprozesse und Ad-hoc Queries entworfen und erledigen diese Prozesse problemlos. Wenn diese Datenbanksysteme für die Unterstützung der neuen typisierten Anwendungen eingesetzt werden sollen, zeigen sie jedoch einige Schwächen. Diese Schwächen sind unter anderem: fehlende Unterstützung der vordefinierten Datentypen, Mängel bei der Unterstützung von benutzerdefinierten Datentypen. „Impedance mismatch" ist auch ein bekanntes Problem, wenn auf die Daten von einer objektorientierten Programmiersprache aus zugegriffen werden sollen. Bei einem navigierenden Zugriff ist vor allem die Geschwindigkeit sehr niedrig [Schmidt87].
Im Vergleich dazu sind objektorientierte Datenbanksysteme für die Speicherung persistenter Objekte in einer objektorientierten Programmiersprache entworfen. Jedoch verlieren sie diese Vorteile, wenn der Datenbankzugriff durch eine andere Sprache unterstützt wird, selbst wenn diese Sprache auch objektorientiert ist. Deswegen haben sie weniger Unterstützung für Ad-hoc Queries, sowohl in der Einfachheit der Benutzung als auch in der Performanz.
Ein allgemeiner neuer Trend ist die Entwicklung der sog. objekt-relationalen Datenbanksysteme. Diese Systeme basieren auf relationalen Datenbanksystemen, sind jedoch um die Unterstützung für einige der objektorientierten Eigenschaften erweitert. Das Konzept macht sich die Vorteile relationaler Datenbanken zunutze und überwindet gleichzeitig deren Nachteile durch die Hinzunahme einiger vorteilhafter Eigenschaften der objektorientierten Datenbanksysteme.
Ganz gleich, ob letztendlich ein relationales Datenbanksystem, ein objektorientiertes Datenbanksystem oder ein objekt-relationales Datenbanksystem eingesetzt wird, das Design ist fokussiert auf die Verwaltung strukturierter Daten oder Objekte. Viele Daten sind jedoch

nicht strukturiert oder nur semistrukturiert (liegen z.B in HTML vor). Mit dem schnellen Wachstum des Web sind solche semistrukturierten Daten immer verbreiteter und wichtiger geworden. HTML Dokumente sind nicht einfach zu verwalten und abzufragen, weil sie nicht wohlstrukturiert sind und ursprünglich für die Repräsentation der Daten gedacht waren. Im Gegensatz dazu haben XML (*eXtensible Markup Language*)-Dokumente nicht diese Probleme. Sie sind immer wohlstrukturiert und außerdem unabhängig von der Präsentation. XML hat das Potential, das Web zu revolutionieren (für Datenaustausch, Datenpräsentation, für Suchen usw.) [Chang98]. Datenbanken tauschen Informationen meist mit Hilfe von einfachen Dateiformaten wie etwa „ein Datensatz pro Zeile mit Semikolons oder Kommata (*Comma Separated Values*) zwischen den Feldern" aus. Dies ist für objektorientierte Informationen nicht ausreichend. Objekte müssen interne Strukturen mit dazwischenliegenden Links besitzen. XML kann dies mittels Elementen und Attributen abbilden, um so ein allgemeines Format für die Datenübertragung von Datensätzen von und zu Datenbanksystemen zur Verfügung zu stellen.
Es muß erwähnt werden, daß mit XML zwar eine Möglichkeit geschaffen wird, die Dokumente des PIA Systems als strukturierte Daten zu speichern, die leicht abzufragen und zu übertragen sind. Jedoch ist der Einsatz einer relationalen Datenbank in PIA System nicht völlig ausgeschlossen. Zwar ist die Abbildung auf ein relationales Datenbanksystem schwierig, diese kann aber später Performanzvorteile bringen. Die Idee dabei ist, möglichst viele Indizes für einzelne Attribute anzulegen, um so eine Grundlage für die schnelle Auswertung von Fuzzy-Anfragen zu schaffen .

3.3 XML

Die Suche im Internet ergibt oft Hunderte, manchmal Tausende von Treffern zu einem Suchbegriff, von denen aber die meisten oft nicht der Erwartung des Benutzers entsprechen. Einen Ausweg verspricht eine neue Technik, mit der die HTML, das Standardformat von Dokumenten in WWW, auf intelligente Weise erweitert wird: XML bringt Struktur in die Dokumente ein [Goldfarb99]. HTML und XML haben SGML (*Standard Generalized Markup Language*) als ihren Ursprung. HTML erbt einige wichtige Stärken von SGML. Bis auf ein paar Ausnahmen sind seine Elementtypen verallgemeinert und deskriptiv, keine Formatkonstrukte wie in Sprachen wie TeX und Microsoft Word. Andererseits benutzt HTML nur einen festen Satz von Elementtypen. Ein Dokumenttyp kann nicht für alle Zwecke eingesetzt werden. So greift HTML nur die erste von SGMLs Grundweisheiten auf, nämlich daß die Darstellung von Dokumenten standardisiert sein müßte. Es ist nicht erweiterbar und kann deshalb nicht auf einzelne Dokumenttypen zugeschnitten werden. Mit wachsender Popularität des Web erschien der feste Dokumenttyp von HTML als nicht ausreichend. Daher beschloß das WWW-Konsortium, eine Untermenge von SGML zu entwickeln, die die Hauptvorzüge von SGML bewahrt, aber auch die Webethik der

minimalisierten Einfachheit berücksichtigt. So entstand XML, für das zugehörige Standards für erweitertes Hyperlinking und Stylesheets geschaffen werden sollten. Während HTML in erster Linie angibt, wie ein Dokument auf dem Bildschirm optisch dargestellt werden soll, gibt XML Auskunft über die Inhalte des Dokuments und ermöglicht so eine strukturierte Ausgabe sowie die Interpretation von Informationen. Genau diese Möglichkeit der Selbstbeschreibung von Daten ist das, was Entwicklungen wie E-Commerce nach vorne treibt [Chang98].
XML ist eine vereinfachte und explizit für die Webanwendung vorgesehene Submenge von SGML. Mit XML kann jedes Dokument im Internet ähnlich wie im Katalog einer Bibliothek mit Informationen über Autor, Stichworte oder Dateiformat versehen werden. XML wird meistens als eine Daten-Beschreibungssprache benutzt und ermöglicht es, die Daten in Datenstrukturen zu organisieren, auch wenn diese kompliziert sind. Wie HTML arbeitet auch XML mit Tags, doch bietet XML den Benutzern die Möglichkeit, eigene Tags zu definieren. Diese bestimmen Inhalt und Struktur von Datenfeldern und werden in einer sog. *Document Type Definition* (DTD) gespeichert.

3.3.1 XML-Dokumente und DTDs

Im Vergleich zu HTML ist ein XML Dokument beliebig erweiterbar. Während HTML einen festen Satz von Elementtypen vorsieht, der die Darstellung von Dokumenten standardisiert und deshalb nicht auf einzelne Dokumenttypen zugeschnitten sein kann, kann für jedes XML Dokument eine DTD definiert werden, die die Syntax für dieses XML Dokuments festlegt. In einer DTD-Datei werden die Tags, die vorhanden sein müssen oder optional sind, sowie die Reihenfolge und die erlaubte Schachtelung der einzelnen Tags spezifiziert. So können Daten in XML-Dokumenten in der gewünschten Struktur dargestellt werden. Zusätzlich können Dokumente auf ihre Validität überprüft werden, d.h. darauf, ob sie den Vorgaben ihrer DTD entsprechen.
Neben den angesprochenen Vorteilen kommen folgende Eigenschaften von XML bei der Übertragung von strukturierten Dokumenten über das Internet zum Tragen:

- Für die Übertragung von XML Dokumenten stellen Firewalls keine Hindernisse dar.
- XML ist in der Lage, auch komplexe Datenstrukturen abzubilden. Mittlerweile werden komplette Datenbanken im XML-Format realisiert.
- Ein XML-Dokument kann sowohl durch eine Anwendung verarbeitet werden als auch in Web-Browsern mit Hilfe einer Stilvorlage (*Style Sheet)* angezeigt werden.
- XML ist vom WWW-Konsortium als Norm definiert und wird von allen namhaften Softwareherstellern unterstützt.

So werden sich in Zukunft neben individuellen Lösungen auch branchenspezifische XML-Standards entwickeln. Ein Beispiel dafür ist CML (*Chemical Markup Language*) – eine

XML-basierte Sprache zur Beschreibung der Verwaltung von Molekularinformationen in Computer-Netzwerken [Goldfarb99].

3.3.2 Ein XML Beispiel in PIA

In PIA werden die einzelnen Produktdaten durch XML Dokumente beschrieben. Exemplarisch wird im folgenden ein kleines Beispiel – ein PC mit der Bezeichnung „Fujitsu Euroline A366“ – wiedergegeben:

```
<?xml version="1.0"?>
<!DOCTYPE catalog SYSTEM " item.dtd">
<catalog>
  <name>Some Catalog</name>
  <url>http://www.sts.tu-harburg.de</url>
  <item>
    <id>1</id>
    <attribute>Bezeichnung</attribute>
    <value>Fujitsu Euroline A 366</value>
    <attribute>Kategorie</attribute>
    <value>PC</value>
    <attribute>Beschreibung</attribute>
    <value>3AMD K6-2 Prozessor mit 366 MHz, 64 Mbyte Arbeitsspeicher, 6,4 Gbyte
        Festplatte,    32 fach CDROM Laufwerk, Maus und Tastatur. Softwareausstattung:
        MS   Windows 98, MS Windows 98 Start!, MS Word 97, MS Works 4.5, MS
        Publisher 98, MS Internet Explorer 4.0, MS Encarta Weltatlas, Thunderbyte
        Antivirenprogramm und MS Cart Precision Racing inkl. 43,18 cm (17")
        Farbmonitor</value>
    <attribute>CPU-Typ</attribute>
    <value>3AMD K6-2</value>
    <numattribute>Taktfrequenz</numattribute>
    <value>366</value>
  </item>
</catalog>
```

Die entsprechende DTD Datei sieht folgendermaßen aus:

```
<?xml version="1.0" encoding="ISO-10646-UCS-2"?>
<!-- Revision: 24 1.2 docs/data/personal.dtd, xml4jdocs, xml4j-jtcsv, xml4j_1_0_4  -->
<!ELEMENT catalog (name, url?, item*)>
```

```
<!ELEMENT item (id, url?, ((attribute | numattribute), value)*)>
<!ELEMENT name (#PCDATA)>
<!ELEMENT id (#PCDATA)>
<!ELEMENT attribute (#PCDATA)>
<!ELEMENT numattribute (#PCDATA)>
<!ELEMENT value (#PCDATA)>
<!ELEMENT url (#PCDATA)>
```

3.3.3 Parsen und Durchblättern von XML-Dokumenten

Ein XML Dokument ist nutzlos, wenn man es nicht weiterverarbeiten kann, um es z.B. in einem Webbrowser darzustellen, in ein Tool zu importieren oder um in dem Dokument nach bestimmten Elementen bzw. Attributen zu suchen. Andererseits kommt es auch häufig vor, daß ein XML-Dokument generiert wird, z.B. als das Exportformat eines Informationssystems oder als das Suchergebnis einer Datenbankabfrage. Es gibt verschiedene Werkzeuge, die für die Bearbeitung eines XML Dokuments verwendet werden können, wobei das wichtigste Werkzeug für die Interpretation eines XML Dokuments ein sogenannter XML-Parser ist. Auf dem Markt gibt es im Moment verschiedene XML-Parser, wie z.B. den XML-Parser von IBM, der in dieser Arbeit eingesetzt wird.
Die meisten XML-Parser werden in Java geschrieben und bieten eine Menge von Standardschnittstellen zum Zerlegen eines XML-Dokuments in Komponentenelemente. Zwei grundlegende Typen von XML APIs (*Application Programming Interface*) sind vorhanden:

- baumbasierte API
- ereignisbasierte API.

Eine baumbasierte API bildet ein XML-Dokument in einer internen Baum-Struktur ab, die der physikalischen Struktur des Dokuments entspricht. Das DOM (*Document Object Model*), standardisiert von W3C (*World wide Web Consortium*), ist in Beispiel einer baum-basierten API. Eine ereignisbasierte API benachrichtigt einen vorher registrierten Handler über bestimmte Ereignisse, wie z.B. den Anfang oder das Ende eines Elements. Die Aufgabe einer Anwendung liegt nun bei der Implementation des entsprechenden Event-Handlers. Eine ereignisbasierte API unter dem SAX (S*imple API for XML*) ist ebenfalls vorhanden. Es bleibt anzumerken, daß es möglich ist, sowohl bestimmte Ereignisse mit Hilfe einer baumbasierten API zu melden, als auch eine Baumstruktur im Speicher selbst zu konstruieren, wenn man ereignisbasiert arbeitet. Näheres zum Thema XML APIs kann in [Chang98][Goldfarb99] nachgelesen werden.
Die optische Darstellung und der Inhalt von XML Dokumente werden getrennt. Für die Präsentation ist zunächst weiter HTML zuständig, das XML-Dokumente integrieren kann. Im WWW-Konsortium wird aber derzeit an einem ergänzenden Format gearbeitet, das die

Formatierung von XML-Seiten und ihre visuelle Ausgabe im Browser vollzieht. Diese XSL (*eXtensible Style-Sheet Language*) zum Erstellung von Formatvorlagen soll voraussichtlich im Sommer 1999 als Standard verabschiedet werden. Die künftigen Browser werden neben HTML auch XML darstellen.

3.4 RMI Systemarchitektur

Als Middelware im PIA System wird RMI eingesetzt. RMI ist eine Java-spezifische Implementation eines verteilten Objekt-Modells. Daneben werden auch andere Verfahren zur Entwicklung verteilter Anwendungen eingesetzt, die Implementationen von CORBA (*Common Object Request Broker Architecture*) verwenden. Diese Verfahren verlangen aber allgemein sprachen-neutrale Schnittstellen, die in einer unabhängigen Schnittstellen-Beschreibungssprache (IDL, *Interface Definition Language*) geschrieben werden und getrennt von der Anwendung kompiliert werden müssen. Als Konsequenz erhöht sich die Komplexität der Entwicklung verteilter Anwendungen mittels dieser Verfahren. Im Vergleich dazu bietet RMI eine einfache Schnittstelle für die verteilten Objekte, die in der Programmiersprache Java implementiert sind. RMI ist entworfen worden, um

- multiple Invokationsmechanismen zu unterstützen – z.B. Invokation eines einzelnen Objekts oder eines Objekts, das an multiple Lokationen kopiert wird.
- verschiedene Referenzsemantiken für entfernte Objekte zu unterstützen – z.B. transiente Referenzen, persistente Referenzen und „spätere Aktivierung“.
- verteilte Garbage Collection von aktiven Objekten anzubieten.
- multiple Transport-Mechanismen zu unterstützen – z.B. TCP/IP und UDP/IP.

Das RMI System besteht aus drei Schichten: (vgl. Abb. 3-2):

- Die Stub/Skeleton Schicht – auf der Client-Seite Stubs und auf der Server-Seite Skeletons
- Entfernte Referenzschicht – entfernte Invokation und Referenzsemantik
- Transportschicht – Verbindungsmanagement und Überwachung entfernter Objekte.

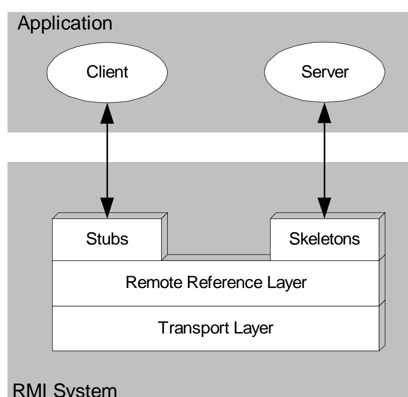


Abbildung 3-2: RMI Systemarchitektur

Die Stub/Skeleton Schicht

Die Stub/Skeleton Schicht ist die Schnittstelle zwischen den Applikationen und dem RMI System. Sie überträgt Daten zu der entfernten Referenzschicht, indem eine Objekt-Serialisierung durchgeführt wird, um die (lokalen) Objekte als Werte übertragen zu können (*pass by value*).

Ein Stub ist das Proxy der Clientseite für ein entferntes Objekt. Es implementiert alle entfernten Schnittstellen, die vom entfernten Objekt unterstützt werden. Es ist zuständig für:

- die Initialisierung eines entfernten Aufrufs eines bestimmten entfernten Objektes durch die entfernte Referenzschicht
- Serialisierung der Argumente zu einem Serialisierungsstrom
- Aufforderung der entfernten Referenzschicht zu einem Methodenaufruf
- Deserialisierung des Ergebnisses oder einer Ausnahme aus einem Serialisierungsstrom
- Benachrichtigung der entfernten Referenzschicht, daß der Funktionsaufruf vollständig durchgeführt wurde.

Ein Skeleton ist das Proxy der Serverseite für ein entferntes Objekt. Es analysiert die Parameterwerte einer ankommenden Nachricht, um herauszufinden, für wen sie gedacht ist – also für welches Objekt und für welche Methode. Es hat folgende Aufgaben:

- Deserialisierung der Argumente aus einem Serialisierungsstrom
- Weiterleiten der Nachrichten an das tatsächliche entfernte Objekt
- Serialisierung des Ergebnisses oder einer Ausnahme zu einem Serialisierungsstrom.

Die passenden Stub- und Skeleton-Klassen werden zur Laufzeit festgelegt und können bei Bedarf (d.h. bei der ersten Referenzierung des entfernten Objekts) dynamisch geladen werden.

Die entfernte Referenzschicht

Die entfernte Referenzschicht arbeitet mit der Transportschnittstelle. Sie ist zuständig für die Ausführung eines speziellen entfernten Referenzprotokolls. Verschiedene Invokationsprotokolle werden hier unterstützt [Downing98]. Diese Schicht hat zwei zusammenarbeitende Komponenten: die Komponente auf der Clientseite und die auf der Serverseite. Die clientseitige Komponente enthält die für den entfernten Server spezifische Information und kommuniziert mit Hilfe der Transportschicht mit der serverseitigen Komponente. Die serverseitige Komponente implementiert die spezifische entfernte Referenzsemantik, bevor sie die Methodenaufrufe an das Skeleton weiterleitet. Diese Schicht überträgt Daten an die darunterliegende Transportschicht durch eine sog. stromorientierte Verbindung.

Die Transportschicht

Die Transportschicht ist zuständig für:

- die Einrichtung der Verbindungen zu entfernten Adressräumen
- die Steuerung der Verbindung
- die Abstraktion von der darunterliegenden Verbindung
- das Warten auf ankommende Nachrichten
- die Verwaltung der Tabelle der entfernten Objekte
- das Lokalisieren des Verteilers für das Ziel des entfernten Aufrufs und die Übergabe der Verbindung an den Verteiler.

Näheres zu RMI ist in [Downing98] zu finden. Es bleibt zu erwähnen, daß in dem im Rahmen dieser Arbeit entwickelten prototypischen System zunächst kein RMI eingesetzt wurde.

3.5 Die Programmiersprache Java

Sowohl auf der Applikationsschicht als auch auf der Repräsentationsschicht im PIA System wurde Java als Implementationssprache gewählt. Syntaktisch ist Java eine C++ ähnliche Programmiersprache. Sie weicht in einigen Eigenschaften von C++ ab, um einfacher und robuster zu werden, und bietet außerdem zusätzliche Eigenschaften gegenüber C++. Ihre Entwicklung begann 1990 bei der Firma SUN in einem Team, das unter der Leitung von James Gosling stand. Die Designer versuchten, die Syntax der Sprachen C und C++ soweit wie möglich nachzuahmen, verzichteten aber auf einen Großteil der komplexen und fehlerträchtigen Merkmale beider Sprachen. Das Ergebnis ihrer Bemühungen haben sie wie folgt zusammengefaßt:
"Java soll eine einfache, objektorientierte, verteilte, interpretierte, robuste, sichere, architekturneutrale, portable, performante, nebenläufige, dynamische Programmiersprache sein" [Eckel98].

3.5.1 Schnittstellen

In Java gibt es die von C++ und anderen objektorientierten Sprachen bekannte Mehrfachvererbung nicht. Die möglichen Schwierigkeiten im Umgang mit mehrfacher Vererbung und die Einsicht, daß das Ererben von nicht-trivialen Methoden aus mehr als einer Klasse in der Praxis selten zu erreichen ist, haben die Designer dazu veranlaßt, diese Eigenschaft (s.o.) nicht zu realisieren. Andererseits ist es wünschenswert, Methodendeklarationen von mehr als einer Klasse zu erben. Um die Einschränkungen in den

Designmöglichkeiten, die durch Einfachvererbung entstehen können, zu vermeiden, wurde mit Hilfe der Schnittstellen (*Interfaces)* ein Konstrukt geschaffen, das genau diese Eigenschaft der Mehrfachvererbung bietet. Damit wurde eine neue, restriktive Art der Mehrfachvererbung eingeführt (vgl. Abb. 3.3).

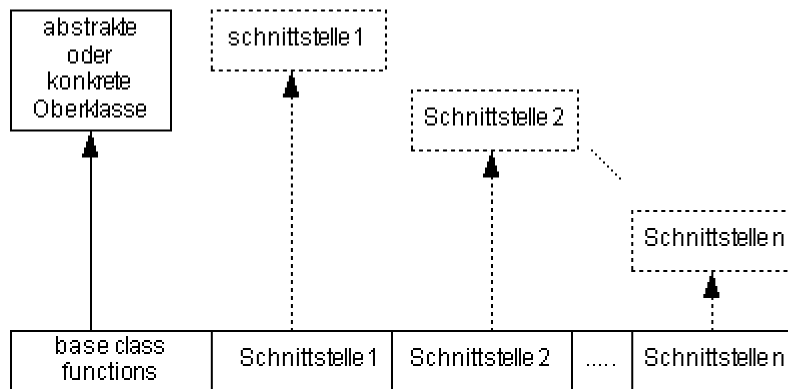


Abbildung 3-3. Mehrfachvererbung in Java mit Schnittstellen

Schnittstellen sind die Methodendeklarationen, die keinerlei Implementierung, sondern ausschließlich abstrakte Methoden und Konstanten enthalten. Eine Schnittstelle dient der Beschreibung des Verhaltens von Objekten, die diese Schnittstelle implementieren. Eine Klasse kann mit dem Schlüsselwort *implements* eine oder auch mehrere Schnittstellen ererben. Diese Klasse erbt damit alle Konstanten und die Methodendeklarationen. Dadurch ist die Klasse dann verpflichtet, alle Methoden, die in der Schnittstelle deklariert werden, geeignet zu implementieren. Mehrere Klassen können unabhängig voneinander die gleiche Schnittstelle implementieren, ohne auf Super- und Subklassen-Beziehungen zu achten. Eine in PIA verwendete Schnittstelle sieht z.B. so aus:

```
package pia.maintenance.domain;
...

public interface DomainChangeListener
{
  public void domainCreated(AbstractDomain domain);
  public void domainChanged(AbstractDomain domain);
}
```

Abbildung 3-4. Quelltextauszug aus der Schnittstelle DomainChangeListener

und eine Klasse, die diese Schnittstelle implementiert sieht folgendermaßen aus:

```
public class DomainPanel implements DomainChangeListener{
  ...
  public void domainCreated(AbstractDomain domain)
   {// Hier die Implmentierung der Methode}
   public void domainChanged(AbstractDomain domain)
   {// Hier die Implmentierung der Methode }
   ...
}
```

Abbildung 3-5. Quelltextauszug aus der Klasse DomainPanel

Eine beliebige Anzahl an Klassen kann die gleiche Schnittstelle implementieren, und eine Klasse kann gleichzeitig eine beliebige Anzahl an Schnittstellen implementieren (vgl. Abb. 3.3). Schnittstellen besitzen zwei wichtige Eigenschaften, die auch Klassen haben:

- Sie lassen sich vererben.
- Variablen können vom Typ einer Schnittstelle sein. Das bedeutet, daß man ihnen Objekte zuweisen kann, die aus Klassen abstammen, die diese oder eine der daraus abgeleiteten Schnittstellen implementieren.

Es macht also Sinn, eine Schnittstelle als eine Typvereinbarung anzusehen. Eine Klasse, die diese Schnittstelle implementiert, ist dann auch vom Typ der Schnittstelle. Wegen der Mehrfachvererbung von Schnittstellen kann eine Instanzvariable damit insbesondere mehrere Typen haben und zu mehr als einem Typ zuweisungskompatibel sein.
Der intensive Gebrauch von Schnittstellen zeichnet sich in den Klassendiagrammen im Kapitel 5 ab.

3.5.2 Polymorphismus

Polymorphismus ist die dritte wesentliche Eigenschaft einer objektorientierten Programmiersprache neben Kapselung und Vererbung. Der Polymorphismus macht es möglich, daß Klassen Nachrichten an ihre Oberklasse verstehen, obwohl die technische Umsetzung der Reaktion auf diese Nachricht völlig unterschiedlich sein kann. Polymophismus wird dadurch realisiert, daß ein Objekt eine geerbte Methode abändern kann, um auf eine Nachricht in der gewünschten Weise zu reagieren, nämlich durch Überschreiben der geerbten Methode. Während Vererbung die Behandlung eines Objekts sowohl als Objekt seiner eigenen Klasse als auch als Objekt der Superklasse erlaubt, führt ein polymorpher Methodenaufruf dazu, daß verschiedene Objekte auf dieselbe Nachricht mit unterschiedlichem Verhalten reagieren, da sie auf unterschiedliche Implementationen zugreifen. In der Beschreibung der Entwurfsphase wird der intensive Gebrauch von

Polymorphismus im PIA Frontend System mittels Klassendiagrammen detailliert erläutert (vgl. Kapitel 5).

3.5.3 GUI-Programmierung mit Java

Das PIA Frontend wird vollständig in Java implementiert. Dabei wird das Toolkit JFC (*Java Foundation Classes*) verwendet, das das Standard Java-Framework um zusätzliche Funktionen erweitert.
Die Java Laufzeit-Bibliothek bietet umfassende graphische Fähigkeiten. Diese sind im wesentlichen plattformunabhängig und können dazu verwendet werden, portable Programme mit GUI-Fähigkeiten zu erstellen. In der früheren Version - JDK 1.1 wurden die graphischen Fähigkeiten der Sprache durch das *Abstract Windowing Toolkit* (AWT) zur Verfügung gestellt. Zwar konnten mit Hilfe des AWT einfache Knöpfe, Textfelder und vieles mehr erstellt werden, aber das AWT hatte Mängel an vielen Stellen. So war es nicht möglich zu drucken, Scrolling war nicht eingebaut, und es gab keine ausreichende Ereignisbehandlung. Vor allem aber beruhte das AWT auf betriebsystemabhängigen Implementierungen der Benutzerschnittstellen-Elemente. Aus diesen Gründen wurde ein neues Toolkit - das *Java Foundation Toolkit* (JFC) von SUN entwickelt, das anstelle der schwergewichtigen Komponenten des AWT (*heavyweight*, lokale Betriebssystem-Implementierung) leichtgewichtigte Komponenten (*lightweight*, nur in Java selbst implementiert) verwendet (vgl. Abbildung 3.6). Ab Version 1.2 des JDK ist das JFC als fester Bestandteil der Klassenbibliothek von Java integriert.

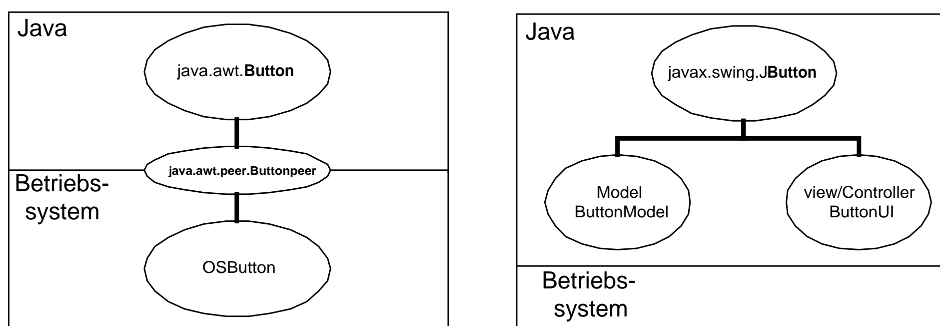


Abbildung 3-6. Schwergewichtige und leichtgewichtige Komponenten in Java

3.5.3.1 Java Foundation Classes (JFC) und Swing

Die wichtigsten Bestandteile der JFC werden nachfolgend kurz aufgeführt (vgl. Abbildung 3.7).

- *Abstract Windowing Toolkit* (AWT)
- *Swing*
- *Java 2D*
- *Accessbility*
- *Drag & Drop*

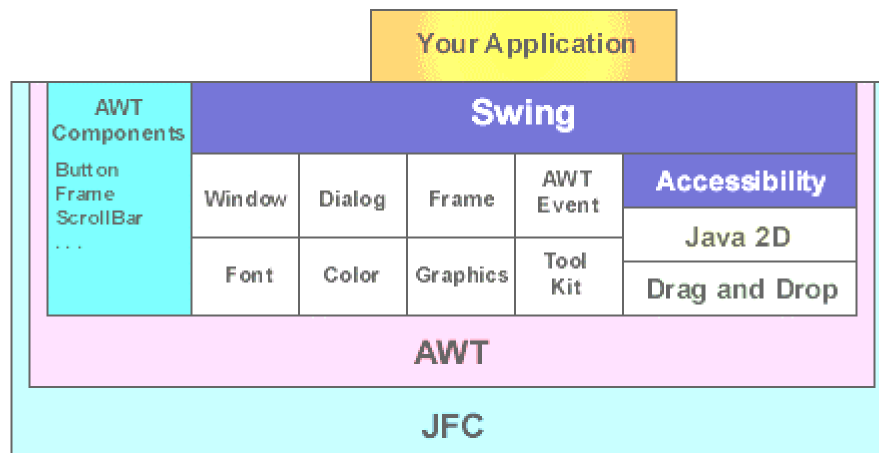


Abbildung 3-7. JFC in Überblick

Der grundlegende Unterschied zwischen dem AWT und dem Swing besteht darin, daß das AWT lediglich die größte gemeinsame Teilmenge von Benutzerschnittstellen-Elementen aller unterstützten Betriebssysteme verwenden konnte. Swing ist frei von solchen Einschränkungen und definiert eine beliebige Obermenge von Benutzerschnittstellen-Elementen, ohne lokale Implementierungen berücksichtigen oder verwenden zu müssen.
Das Swing bietet eine breite Ansammlung von Komponenten. Eine der interessanten und neuartigen Eigenschaften der Swing Architektur ist die Tatsache, daß mit Swing implementierte Applikationen ein plattformunabhängiges Look-and-Feel (*PLAF*) haben, weil Swing die Existenz von Benutzerschnittstellen-Elementen in einem Betriebsystem nicht voraussetzt, sondern alle von sich aus in Java zeichnet und verwaltet, was auch den Hauptunterschied zwischen dem Java AWT und JFC ausmacht. Als Konsequenz davon hat eine Anwendung, die in JFC entwickelt wird, genau dasselbe Aussehen z.B. auf einem Windows 95 Rechner und auf einem Solaris Rechner. Hinzu kommt, daß Swing den Benutzern das Umschalten zwischen verschiedenen look-and-feels zur Laufzeit ermöglicht, ohne die laufende Applikation schließen zu müssen. Swing Klassen können auch auf die gleiche Weise wie die des AWT in Applets benutzt werden, wobei die Sicherheitsaspekte, die Applets in JDK 1.1 mit sich bringen, erhalten bleiben.
In dieser Arbeit wird vorwiegend das Toolkit Swing eingesetzt. Swing bezeichnet eine Menge von Erweiterungen des AWT. Der hauptsächliche Teil der Swing-Klassen bildet einen Ast in der AWT-Hierarchie. Die Gemeinsamkeiten beschränken sich allerdings auf

zwei Klassen: die Swing-Oberklasse *JComponent* ist eine Unterklasse von *java.awt.Container* und diese wiederum von *java.awt.Component*. Im übrigen hängt an *JComponent* eine Hierarchie von Benutzerschnittstellen-Elementen, die die entsprechenden Elemente des AWT ersetzt und erweitert. In einigen weiteren Eigenschaften unterscheiden sich Swing-Komponenten von den alten AWT-Komponenten:

- Eine umfangreiche Variation von neuen Komponenten wie z.B. Tabellen, Bäume, Sliders, Progress-Bars, interne Frames und Textkomponenten (vgl. Abb. 3.8).
- Swing Komponenten enthalten Unterstützung für den Ersatz ihrer Seitenränder durch eine beliebige Anzahl von konzentrischen Borders.
- Swing Komponenten können Tooltips haben, die angezeigt werden, wenn sich der Mauszeiger über diese Komponenten bewegt.

Als fundamentales Designmuster hinter jeder Swing-Komponente wird die Model-View-Controller (MVC) Architektur eingesetzt. Mittels Unterstützung des MVC Konzeptes kann Swing seine Funktionalität und Gestalt beliebig anpassen.

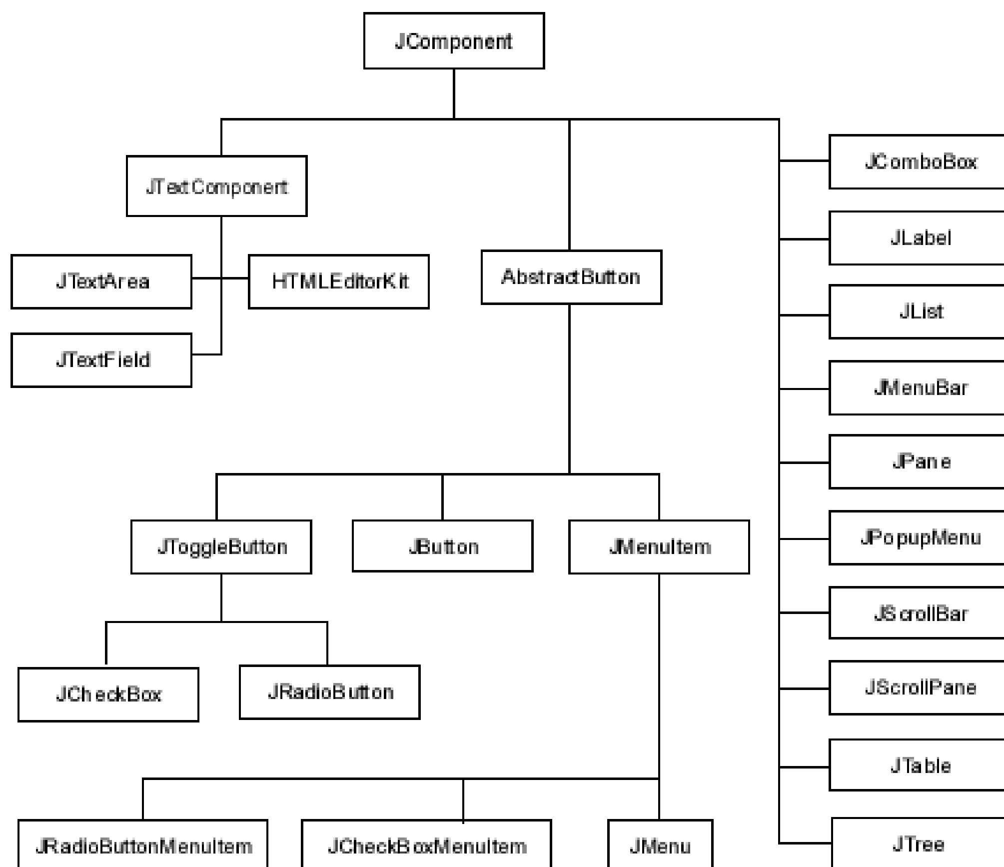


Abbildung 3-8. Swing Klassenhierarchie

3.5.3.2 Das Model-View-Controller (MVC) Konzept in Swing

MVC ist eines der bekanntesten Entwurfsmuster (vgl. Abschnitt 5.2.2 Entwurfsmuster). MVC teilt eine GUI Komponente in drei Elemente – Model, View und Controller [Gamma95]. Jedes dieser Elemente spielt eine entscheidende Rolle dabei, wie sich die graphische Komponente verhält. Das Modell-Objekt stellt das Anwendungsobjekt dar, das View-Objekt seine Bildschirmrepräsentation, und das Controller-Objekt bestimmt die Möglichkeiten, mit denen die Benutzerschnittstelle auf Benutzerinteraktivitäten reagieren kann. Abbildung 3.9 zeigt wie Model, View und Controller zusammenarbeiten.

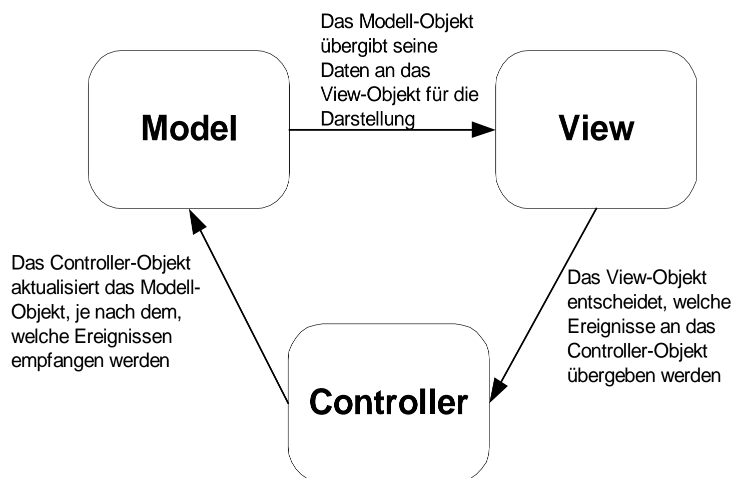


Abbildung 3-9. Kommunikation in der Model-View-Controller Architektur

Die Entkopplung dieser drei Funktionen bringt eine Reihe von Vorteilen mit sich: Das Datenmodell muß sich nicht um seine Repräsentation kümmern. Es können für ein einziges Datenmodell vielerlei verschiedene Views verwendet werden, die die Informationen im Modell repräsentieren. Die konkrete Darstellung einer View kann verändert werden, ohne das Datenmodell zu beeinflussen. In der Umsetzung des MVC-Musters in der Praxis hat es sich herausgestellt, daß View und Controller meistens stark voneinander abhängig sind. Es muß zu jeder View genau einen Controller geben. Deshalb wird sehr häufig, so auch in Swing, eine leicht abweichende Version des MVC-Musters – das *model-delegate* – verwendet. Dieses Design kombiniert das View- und das Controller-Objekt in einem einzelnen Element, das die Komponente auf den Bildschirm zeichnet und die GUI Ereignisse (*Events*) behandelt. Dieses Element wird in Swing als der UI-Delegierter (*UI-Delegate*) bezeichnet. Die Kommunikation zwischen dem Model und dem UI-Delegierten erfolgt somit in zwei Richtungen, wie in der Abbildung 3-10. illustriert wird.

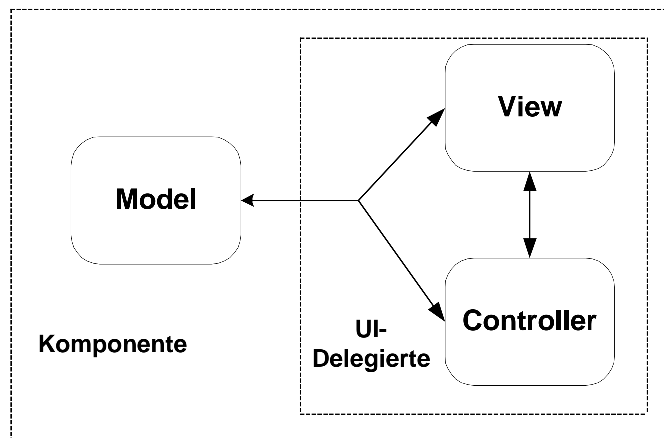


Abbildung 3-10. Das Modell-Delegierter Konzept in Swing

Jede Swing-Komponente besteht aus einem Modell und einem Delegierten. Das Modell-Objekt enthält Informationen über den aktuellen Zustand der Komponente, während der Delegierte für die Steuerung der Informationen über die Repräsentation einer Komponente auf dem Bildschirm zuständig ist. Darüber hinaus reagiert der UI-Delegierte in Verbindung mit dem AWT direkt auf verschiedene Ereignisse, die durch die Komponente propagiert werden.

Die Trennung von Modell und Delegiertem ist sehr vorteilhaft. Ein wichtiger Aspekt der MVC-Architektur ist, mehrere Darstellungen mit einem einzigen Modell-Objekt zu verbinden, so daß bei einer Änderung der Daten nur das Modell-Objekt aktualisiert werden muß, während sich alle Darstellungsobjekte entsprechend automatisch ändern. Auf gleiche Weise ermöglicht diese Trennung ein Umschalten zwischen den unterschiedlichen Look-and-Feels zur Laufzeit, ohne eine Änderung der Datenobjekte vorzunehmen.

Alle Swing-Komponenten unterstützen das MVC-Konzept. Zu diesem Zweck bestehen Modell-Klassen und Delegierte, die entweder einfach übernommen oder geerbt und angepaßt werden können. Bei der einfachen Variante werden die *Defaultmodel*-Klassen implizit verwendet. Ebenso wird der Delegierte implizit definiert, kann aber durch eigene Look-and-Feel-Implementierungen verändert werden. Jedem Swing-Komponenten-Objekt sind zu jedem Zeitpunkt genau ein Modell und ein Delegierter zugewiesen.

Eine *Defaultmodel*-Klasse bietet die folgende Schnittstelle:

- *addOberserver()*
- *deleteOberserver(), deleteOberservers(),*
- *countOberservers()*
- *setChanged()*
- *clearChanged()*
- *hasChanged()*
- *notifyOberservers()*

Im folgenden wird anhand der Komponente *JTable* in Swing erläutert, wie sich das MVC Konzept in Swing bei der Implementierung auswirkt.

3.5.3.3 Table mit MVC

Für die Darstellung von Daten in Tabellenform dient die Swing-Komponente *JTable*. *JTable* eignet sich für die Anzeige zweidimensionaler Datenstrukturen, zum Beispiel Tabellen aus relationalen Datenbanken. Die Komponente *JTable* bildet nur das Schaufenster (Delegierter, respektive View und Controller). Im Hintergrund besteht ein Zusammenspiel verschiedener Klassen und Schnittstellen, welches die flexible Verwendung der Tabellen-Komponente ermöglicht. Die Tabellendaten werden im Tabellenmodell (*TableModel*) gespeichert. Die Darstellung der einzelnen Tabellenzellen erfolgt mittels des Renderers *CellRenderer*. Für diese Klassen stehen vorgefertigte Implementierungen zur Verfügung (*DefaultTableModel* und *DefaultCellRenderer*), damit Tabellen mühelos erstellt werden können.
Die Ereignisse, die durch Tabellenänderungen generiert werden, sind *TableModelEvent* (Daten im Tabellenmodell), *TableColumnModelEvent* (Daten in Spaltenmodell) und *ChangeEvent* (Änderungen in einer Zelle durch *CellEditor*). Ein Objekt der Klasse *JTable* wird beim Instanziieren oder später beim Aufruf der entsprechenden *set*-Methode als ein Beobachter beim zugrundeliegenden Modell-Objekt registriert, so daß es über diese Ereignisse unterrichtet werden kann.
TableModel ist die Schnittstelle mit der Definition des Datenmodells für Tabellen. Gewöhnlich wird deren Implementierung in der Klasse *AbstractTableModel* für eigene Tabellen verwendet. In *AbstractTableModel* wird auch die Verwaltung der Beobachterliste für die verschiedenen Tabellenereignisse implementiert. Das Modell für eine Tabellenkomponente kann in *JTable* mittels *setModel()* gesetzt werden und mittels *getModel()* abgefragt werden.

4 Analyse

Das PIA System versteht sich als Mittler zwischen Informationsanbietern und Informationssuchenden. Auf der einen Seite liegen bei den Anbietern Informationen in Form von HTML oder XML Dokumenten bzw. von Datenbankinhalten vor, die sie ins System einfügen möchten. Auf der anderen Seite suchen Kunden mit Hilfe des PIA Systems die auf ihre jeweiligen Bedürfnisse abgestimmten Informationen. Die zur Realisierung benötige Funktionalität ist in mehreren Schichten aufgeteilt.

Wie schon in der Einleitung erwähnt, unterscheiden wir drei Klassen von PIA Benutzern:

- Kunden
- Anbieter
- Redakteure.

Die Ziele eines Kunden sind:

- Gewinn der Informationen über die im System vorhandenen Produkte durch Ansehen einzelner Produktbeschreibungen, durch Anfordern von repräsentativen Beispielen; Ansicht ganzer Produkte, die seinen Suchkriterien entsprechen, in einer Reihenfolge, in der die Produkte, die am ehesten den Bedürfnissen entsprechen, an vorderster Stelle stehen (*Ranking*).
- inkrementelle Verfeinerung seiner persönlichen Interessen, von intitialen qualitativen Fuzzy-Anforderungen hin zu detaillierteren quantitativen Suchkriterien, die möglicherweise auch in Entscheidungsalternativen umstrukturiert werden können.
- Automatischer Bezug spezieller Angebote, die seinen persönlichen Interessen entsprechen.
- Möglichkeit, eine Entscheidung über die darauf folgenden Aktionen zu treffen, z.B. ein Produkt zu kaufen, einen Service anzufordern, Informationen zu benutzen oder ein Abonnement einzurichten, so daß er in Zukunft über ähnliche Artikel benachrichtigt werden kann.
- Möglichkeit, zu einem späteren Zeitpunkt zum PIA System zurückzukommen und die frühere Interaktion mit dem System ohne Probleme fortsetzen zu können, sowie vom System als individueller Kunde mit einer entsprechenden Konversationshistorie identifiziert werden zu können.

Die Ziele eines Anbieters sind:

- Bereitstellung der Informationen über alle seine Produkte, ohne auf ein allgemeines Schema zur Beschreibung bestimmter Produkte eingeschränkt zu werden.

- Anbieten multipler Zugriffspfade auf dasselbe Produkt, um einem Kunden möglichst viel Spielraum in seinem Entscheidungsprozeß einzuräumen.
- Aus den persönlichen Interessen der Kunden zu lernen, um passende Produkte anzubieten, und auf der anderen Seite, die persönlichen Interessen der Kunden zu wecken.
- Identifikation und Quantifikation von Kundengruppen mit gemeinsamen Interessen und Präferenzen.
- Aus den Reaktionen der Kunden auf seine Angebote zu lernen.

Die Ziele eines Redakteurs sind:

- Integrieren der Informationen von verschiedenen Anbietern unterschiedlicher Produkte.
- Vereinheitlichung von Attributnamen, Maßeinheiten und Terminologie, die von den verschiedenen Anbietern benutzt werden.
- Evolution von Domänen, Attributen und Eigenschaften über die Zeit, mit minimaler Unterbrechung der existierenden Kundenbeziehungen.
- Entwicklung und Durchsetzung einer konsistenten Terminologie für eine reibungslose Kommunikation zwischen Kunden und Anbietern.

Die Benutzer des PIA interagieren direkt mit Java Frontends. Insbesondere können mit Hilfe der Frontends

- Redakteure Information strukturieren
- Anbieter Information aktualisieren
- Kunden Information suchen und finden.

Im folgenden Diagramm werden die gesamten Anwendungsfälle des PIA Systems dargestellt, die in den nachfolgenden Abschnitten detaillierter beschrieben werden. Es bleibt zu erwähnen, daß im PIA System eine weitere Benutzergruppe – die Systemadministratoren - vorhanden ist (vgl. Abbildung 3.1), die jedoch wegen des Umfangs der Arbeit außer Betracht gelassen wird. Die Aufgaben eines Administrators ist die Benutzerverwaltung - insbesondere gehört dazu die Vergabe der Benutzeraccounts [Zhang98].
Jeder Benutzer von PIA macht sich beim Anmelden am System durch Eingabe des Benutzernamen und Paßworts mit PIA bekannt (*Anmeldung*). Dadurch wird erreicht, daß die persönlichen Präferenzen eines Benutzers im System gespeichert und beim jeweiligen Wiedereinloggen vom System wieder vom Server zum Clienten übertragen und anschließend geeignet repräsentiert werden können.

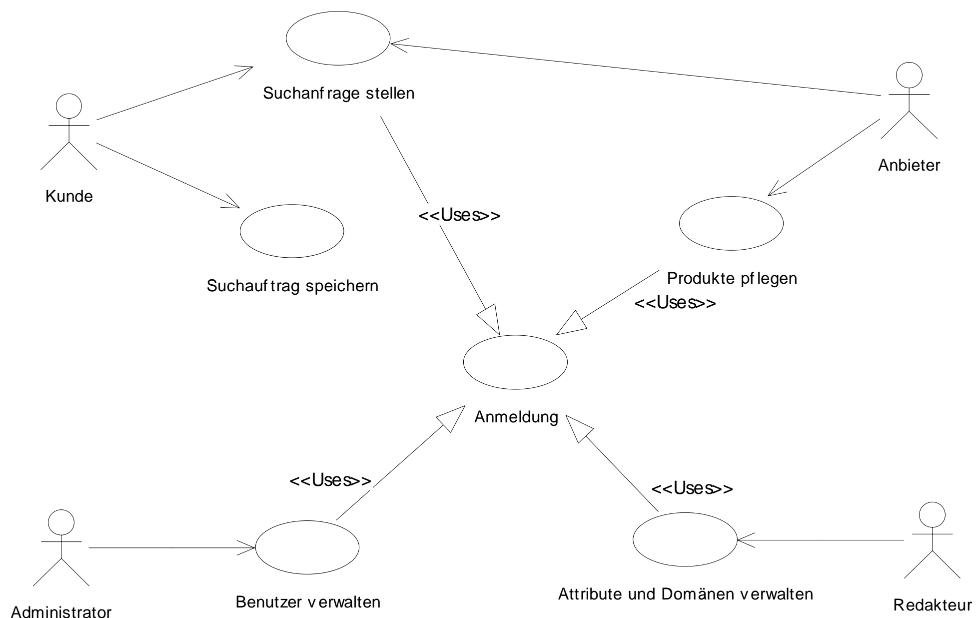


Abbildung 4-1. Anwendungsfälle in Überblick

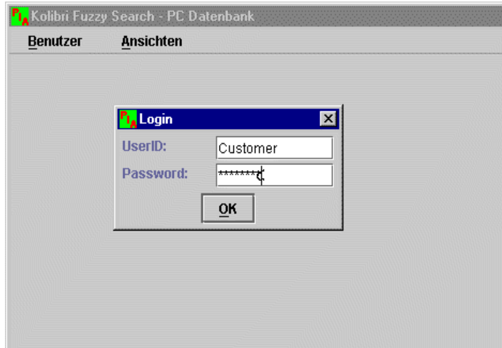
4.1 Anwendungsfälle für Kunden

Wie eingangs bereits beschrieben wurde, sind die Hauptziele eines Kunden Suchen und Auffinden der Informationen, die seinen individuellen Interessen entsprechen. Beim Suchen nach Informationen nennt ein Kunde die Eigenschaften, die die gesuchten Produkte in etwa haben sollten. Kunden müssen dabei ein gesuchtes Produkt nicht von vornherein konkret benennen können. Stattdessen bietet PIA seinen Kunden jederzeit folgende alternative Vorgehensweisen:

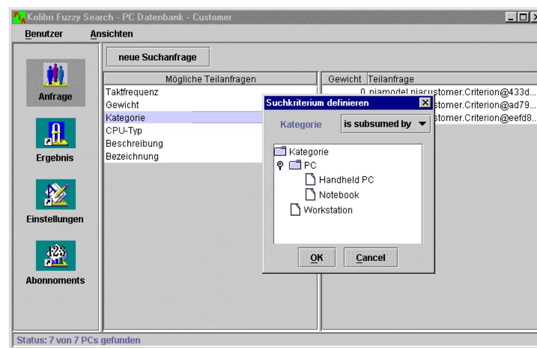
- Bei der Suche können gewünschte Produkteigenschaften genannt werden
- Der Kunde kann sich über Kategorie-Bäume orientieren.

In der Abbildung 4.2 wird die Vorgehensweise beim Suchen nach Produkten in einem PC-Katalog illustriert.

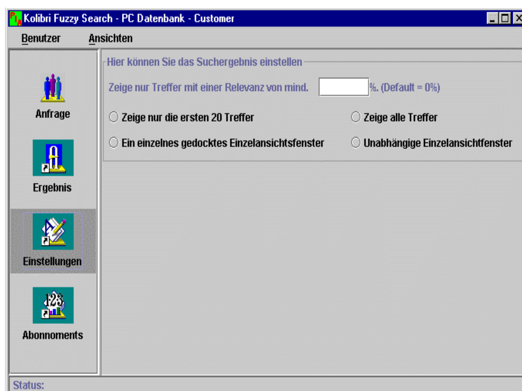
a) Kundenanmeldung



b) Suchanfrage stellen



c) Suchergebnis formatieren



d) Detailansicht eines gefundenen Produktes

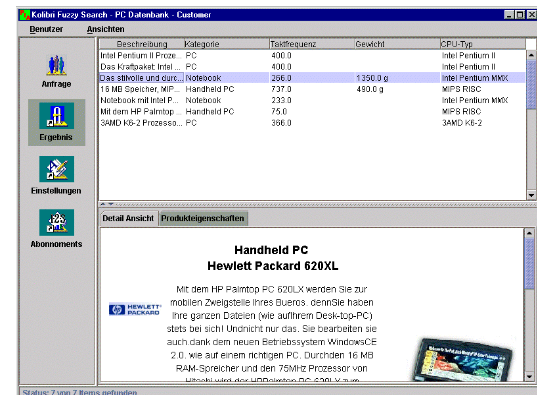


Abbildung 4-2. Anwendungsfälle für Kunden

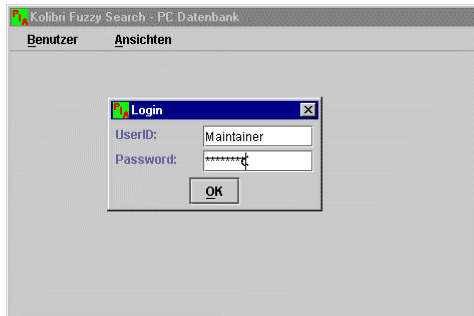
Aus der obigen Abbildung läßt sich erkennen, daß sich ein Kunde zuerst am System anmelden muß, bevor er die Informationsdienste des PIA in Anspruch nehmen kann (vgl. a) in der Abbildung 4.2). Nach der Anmeldung kann ein Kunde seine Suchanfragen stellen. Dabei werden ihm die gesamten Attribute aller Produkte eines Katalogs zur Verfügung gestellt. Diese sind nach Häufigkeit ihres Vorkommens im System sortiert. Der Kunde sucht nun Attribute aus, die seiner Meinung nach wichtig bzw. relevant für die von ihm gesuchten Produkte sind, und legt die gewünschten Werte für diese Attribute unter Verwendung von Fuzzy-Prädikaten fest. Mit Hilfe eines Navigationsfensters im linken Teil des Hauptfensters kann ein Kunde jederzeit zwischen der (Zwischen-) Suchanfrage und dem Suchergebnis für diese Anfrage navigieren (vgl. b) in der Abbildung 4.2). Das Suchergebnis wird ebenfalls in einer Tabelle dargestellt, und jedes zur Anfrage passende Dokument wird mit einer Ranking-Zahl zwischen 0 und 1 versehen, die eine Auskunft darüber gibt, inwiefern das Dokument der Fuzzy-Anfrage entspricht [Matthes99]. Die gefundenen Dokumente werden in der Tabelle standardmäßig nach dieser Ranking-Zahl sortiert. Zugleich ist die Tabelle aber auch nach jeder anderen Spalte sortierbar. Zu jedem Produkt gehört eine URL als Verweis auf detaillierte Informationen. Dieses Dokument kann nach Kundenwunsch in einem gedockten Fenster innerhalb des Ergebnisfensters repräsentiert werden (vgl. d) in der Abbildung 4.2).

PIA bietet dem Kunden weiterhin die Möglichkeit, Einstellungen für die Darstellungsform des Suchergebnisses zu ändern (vgl. c) in der Abbildung 4.2). Dazu gehört insbesondere das Festlegen der Anzahl der anzuzeigenden Treffer. Ein Kunde kann seine Suchanfragen im PIA System speichern lassen. Insbesondere bietet PIA einen Abonnementservice, mit dem sich ein Kunde stets über die aktuellen Informationen gemäß seiner Suchanfragen auf dem Laufenden halten kann. Falls ein Kunde schon einige Abonnements bei PIA eingerichtet hat, werden diese gespeicherten Suchanfragen dem Kunden nach der erfolgreichen Anmeldung beim PIA System aufgelistet, damit er diese Abonnements editieren oder auch löschen kann. Daß jeder PIA-Benutzer beim Anmelden vom System eindeutig identifiziert wird und seine individuellen Interessen und Präferenzen vom System gespeichert und verwaltet werden können, um die Konversation zwischen dem Benutzer und dem Informationssystem effektiver gestalten und unterstützen zu können, entspricht einem der wichtigsten Konzepte von PIA: Personalisierung. Das bedeutet insbesondere, daß die Benutzeroberfläche individuell konfigurierbar ist und auf den jeweiligen Benutzer zugeschnitten ist.

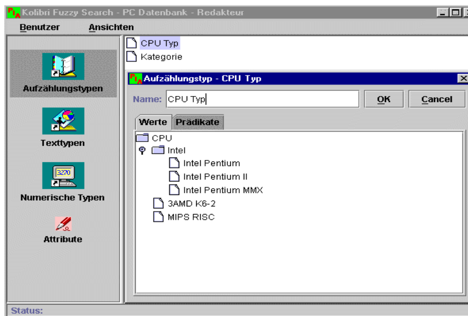
4.2 Anwendungsfälle für Redakteure

Es wurde eingangs gesagt, daß das PIA System eine heterogene Sammlung von Dokumenten in seinem Datenbestand enthält und keine scharfe Kategorisierung eines Dokuments erzwingt. Um jedoch die Informationen nutzbar und konsistent zu halten, wird in PIA System jedes Attribut, das zur Beschreibung einer Produkteigenschaft dient, einer Domäne zugeordnet, die die erlaubten Werte für dieses Attribut definiert. Informationsanbieter benutzen beim Einfügen von Dokumenten ins PIA System Attribute, die von Redakteuren vordefiniert werden. Zur Strukturierung der Information im PIA System steht den Redakteuren ein für sie bestimmtes Java-Frontend für die Bearbeitung der Domänen und der Attribute zur Verfügung. In Abbildung 4.3 werden die Anwendungsfälle und die damit verbundene Vorgehensweise der Redakteure illustriert.
Ein Redakteur muß sich anmelden, bevor er mit Hilfe des PIA Systems Informationen strukturieren kann (vgl. a) in der Abbildung 4.3).
Nach der erfolgreichen Anmeldung kann ein Redakteur die Domänen und die Attribute bearbeiten. Daß Volltextdomänen, hierarchische und numerische Domänen unterschiedliche Eigenschaften haben, führt dazu, daß verschiedenen Benutzeroberflächen für die Bearbeitung der Domänen-Objekten bereitgestellt werden (vgl. b), c) und d) in der Abbildung 4.3).

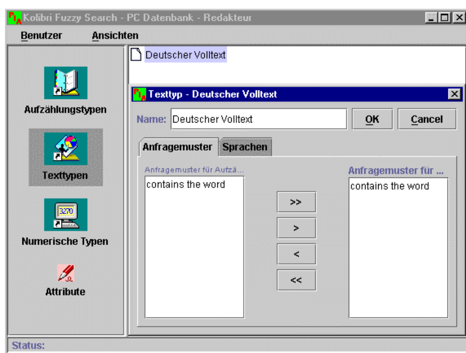
a) Redakteuranmeldung



b) Aufzählungs-Domänen editieren



c) Volltextdomänen editieren



d) numerische Domänen editieren

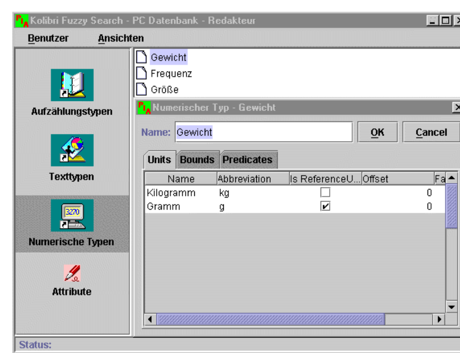


Abbildung 4-3. Benutzerschnittstelle für Redakteure

Dieses schlägt sich auch im folgenden Anwendungsdiagramm nieder (vgl. Abb. 4.4).

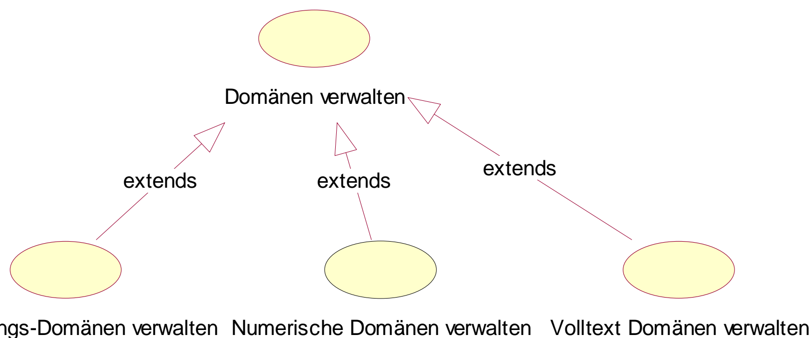


Abbildung 4-4. Spezialisierung des Anwendungsfalls "Domänen verwalten"

Im Katalog eines PC-Anbieters beispielsweise finden sich sehr häufig Attribute wie Kategorie, Preis, Beschreibung, Gewicht, Taktfrequenz, Bezeichnung usw. zur Beschreibung der PC-Produktdaten wieder. Offenbar sind einige dieser Attribute, wie z.B. Bezeichnung und Beschreibung, textuelle Attribute, während einige andere, z.B. Gewicht und Preis, numerisch sind. Wieder andere Attribute haben einen ganz anderen Datentyp, beispielsweise

das Attribut Kategorie zur Klassifikation eines PCs, dessen Werte in hierarchische Konzeptbäume eingeordnet werden können. Als Konsequenz daraus wird in PIA zwischen den beschriebenen drei Klassen von Domänen unterschieden. Da bei der Bearbeitung unterschiedlicher Klassen von Domänen verschiedene Aufgaben durchzuführen sind, muß für jede Klasse dieser Domänen eine eigene Benutzerschnittstelle entworfen und realisiert werden: Domänen für Attribute wie Bezeichnung und Beschreibung werden als *Volltextdomänen* bezeichnet, während wir bei Domänen für Attribute wie Gewicht und Preis von *numerischen Domänen* sprechen. Wieder andere Domänen für Attribute wie Kategorie bezeichnen wir als *hierarchische Domäne*n (*Aufzählungsdomänen, diskrete Domänen*[1]). Näheres zur Klassifikation der Domänen findet sich im Kapitel 5. Allgemeine Aufgaben eines Redakteurs bei der Verwaltung einer Domäne sind, den Wertbereich für diese Domäne festzulegen, indem er bei einer Aufzählungsdomäne die zugelassenen Werte in einem Baum einordnet, bei einer Volltext-Domäne die Sprache zur Beschreibung definiert (vgl. c) in der Abbildung 4.3) oder bei einer numerischen Domäne den Zahlenbereich sowie die erlaubten Maßeinheiten festlegt, in denen die Werte angegeben werden (vgl. d) in der Abbildung 4.3). Während Domänen und Werte unabhängig von Dokumenten existieren, setzt die Existenz eines Attributes das Vorhandensein einer passenden Domäne voraus. Bei der Bearbeitung eines Attributs wird dem Redakteur die Auswahl der vorhandenen Domänen zur Verfügung gestellt, und er kann sich dann für eine Domäne aller in einem Baum dargestellten Domänen entscheiden, der das Attribut zugeordnet werden soll.

4.3 Anwendungsfälle für Anbieter

Ein Anbieter verwaltet seine Produktdaten ebenfalls mit Hilfe des PIA Frontends. Nach dem Einloggen ins PIA System werden ihm ausschließlich die Produkte, die von ihm selbst eingefügt worden sind, aufgelistet. Dies ist ein Sicherheitsaspekt des Systems. Ein Anbieter kann ein neues Dokument mit Hilfe eines von PIA bereitgestellten Editors ins System einfügen. In diesem Editor werden die verfügbaren Attribute angezeigt. Ein Anbieter kann eine Menge von diesen Attributen wählen und die Werte dafür angeben, um so die Eigenschaften des einzufügenden Dokuments zu beschreiben. Falls ein gewünschtes Attribut nicht vorhanden ist, kann ein Anbieter eine Nachricht an den zuständigen Redakteur schicken, der dafür sorgt, daß das entsprechende Attribut zur Verfügung gestellt wird.
PIA bietet außerdem eine Importschnittstelle, über die ein Anbieter seine Dokumente direkt in den vorgesehenen Formaten – HTML oder XML - einfügen kann. Dafür muß ein Anbieter nur einfach die URL nennen, unter der das Dokument gespeichert ist. Aus diesem Dokument werden vom PIA System automatisch XML Dokumente generiert, in denen die vom PIA generierten Produkteigenschaften gespeichert werden. Auch für Produkte, die über HTML

[1] In dieser Arbeit werden diese drei Begriffe synonym verwendet.

Seiten oder XML Dokumente ins System aufgenommen werden, hat ein Anbieter jederzeit die Möglichkeit, vorhandene Eigenschaften zu ändern oder den Produkten weitere Eigenschaften hinzuzufügen, falls beispielsweise seiner Meinung nach beim Parsen durch das PIA System wichtige Produkteigenschaften nicht mitaufgenommen wurden.

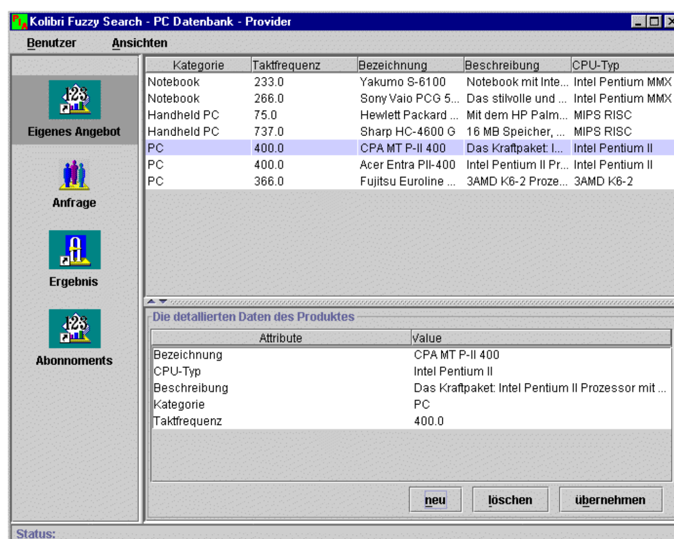


Abbildung 4-5. Benutzerschnittstelle für Anbieter

Anbieter dürfen ebenfalls nach Dokumenten im PIA System suchen. Damit wird durch PIA ein gemeinsamer Lernprozeß zwischen den Benutzern verschiedener Gruppen unterstützt, so daß einem „near miss“ entgegengewirkt werden kann. Darüber hinaus wird ein fließender Übergang zwischen Kunden und Anbietern geschaffen.

5 Entwurf

Obwohl sich die Implementation im Rahmen dieser Arbeit schwerpunktmäßig mit der Benutzerschnittstelle befaßt, ist es ohne eine Beschreibung des zugrundeliegenden Objektmodells nicht möglich, die Anwendungslogik zu verstehen und die Benutzerschnittstelle darauf aufzubauen.
Insgesamt wird zwischen zwei Objekttypen unterschieden. Auf diese Weise lassen sich Objektklassen identifizieren, und diesen Klassen können Funktionen zugewiesen werden, so daß ein stabiles System entsteht und Veränderungen soweit wie möglich auf lokaler Ebene stattfinden können [Yourdan 96].

- *Business-Objekte* entsprechen wirklichen Objekten oder Objekten, die man direkt im Geschäftsbereich erkennen kann.
- *Interface-Objekte* werden vom System zur Kommunikation mit externen Komponenten verwendet, um Funktionen zu kapseln, die von der Umgebung abhängig sind. Sie isolieren Funktionen, die zu Anwender- und Maschinenschnittstellen gehören, so daß nur die Interface- und nicht die Business-Objekte von Detailänderungen der Schnittstellen betroffen sind.

Entsprechend wird das PIA System zunächst grob in zwei Teile geteilt, wie im folgenden Diagramm (Abb. 5.1) zu sehen ist. Das Paket **piamodel** kapselt die Entityklassen, während die Interfaceklassen in dem Paket **pia** lokalisiert werden.

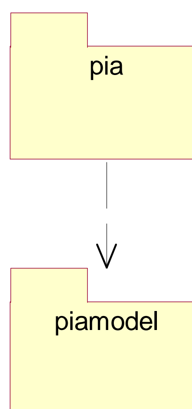


Abbildung 5-1. Systemgrobaufbau

Aufgrund dieser Unterteilung wird in diesem Kapitel in einem Datenmodell zuerst ein Überblick über die Entity-Objekte im PIA System, die die Anwendungslogik des Systems kapseln, und über ihre Beziehungen zueinander gegeben. Danach werden die Interfaceklassen, die zur Darstellung der Entityklassen und der Interaktion zwischen Nutzern und System dienen, im zweiten Teil dieses Kapitels beschrieben. Die wichtigsten Entwurfsmuster, die für das Frontend System verwendet werden, werden ebenfalls in diesem Kapitel betrachtet. Ihr Einsatz im PIA Frontend System wird dann in der Beschreibung der Implementationsphase anhand von Implementationsbeispielen detailliert erläutert.

5.1 Entwurf eines generischen Objekt-Modells

In diesem Abschnitt werden die zentralen Objekte in PIA System wie Katalog, Produkt, Kundenanfrage, Kriterium und Eigenschaft erläutert und ihre Beziehungen zueinander beschrieben. Diese Klassen werden in einem gemeinsamen Klassendiagramm (vgl. Abb. 5.2) dargestellt [Matthes99].

Katalogbenutzer

Wie in der Analysephase erkannt, können Benutzer des PIA Systems in drei Gruppen unterteilt werden. Die entsprechenden Klassen *Kunde*, *Redakteur* und *Anbieter* sind Unterklassen einer gemeinsamen Superklasse *Akteur*. Jeder Benutzer macht sich beim Anmelden durch Eingabe seines Benutzernamens und Paßworts in das Anmeldeformular mit dem System bekannt. Dieser Standard Login-Prozeß kann insbesondere dazu verwendet werden, um einen Anbieter oder einen Redakteur zu identifizieren und zu autorisieren. Damit kann das System vor unautorisiertem Zugriff geschützt werden. Einem Anbieter wird z.B. schreibender Zugriff nur auf seine eigene Produkte gewährt.

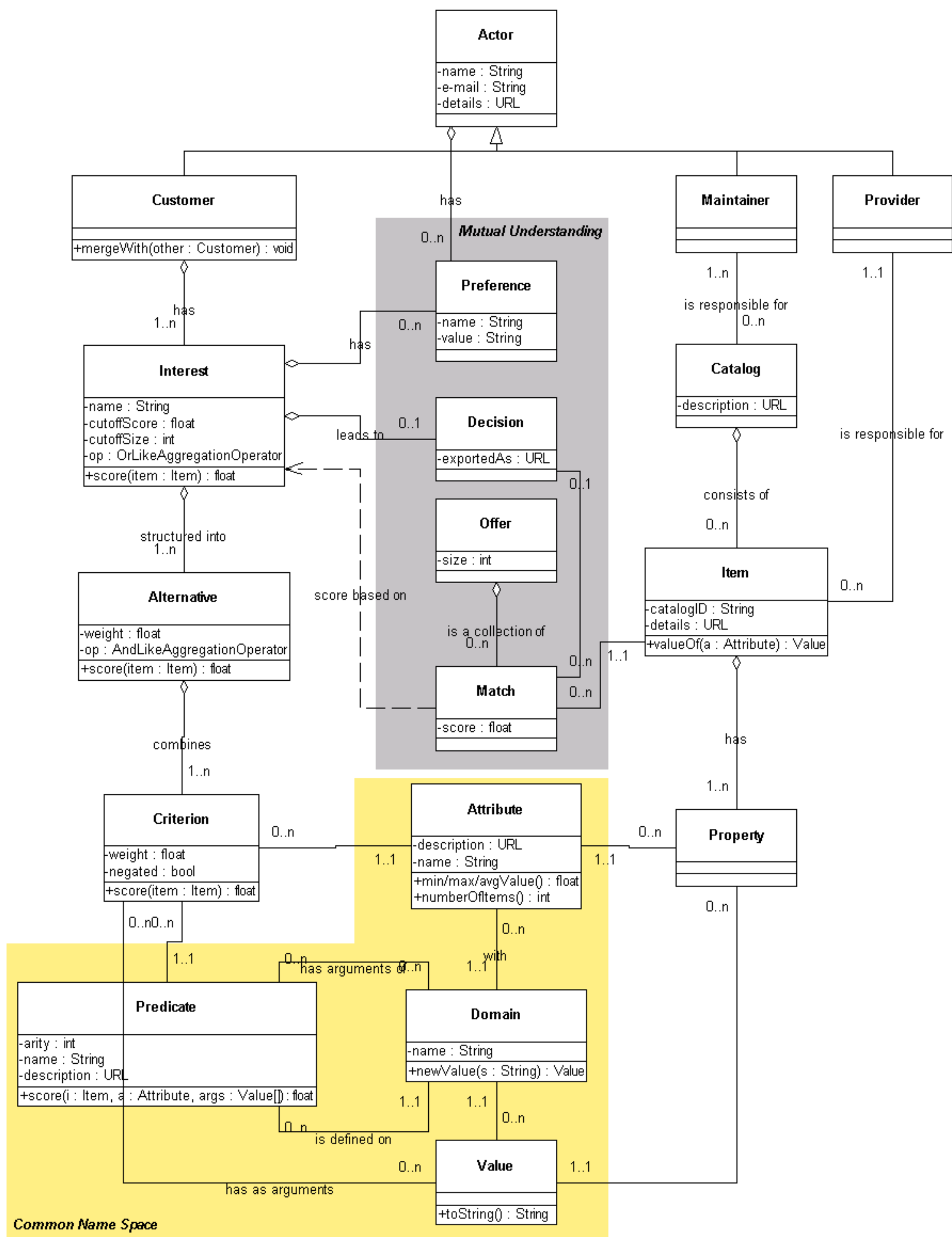


Abbildung 5-2. PIA Business-Klassen in Überblick

Kataloge und Produkte

Die Datenbasis in PIA ist eine Kollektion der verschiedenen Produktkataloge, die von verschiedenen Anbietern in das System eingefügt worden sind und auch von ihnen gepflegt werden. Ein Produktkatalog ist wiederum eine Kollektion von Produkten, die jeweils durch Angaben ihrer wichtigsten Eigenschaften beschrieben werden. Eine Produkteigenschaft ist ein Paar, das aus einem Attribut und der Angabe des Wertes für dieses Attribut besteht. Jedes Produkt im PIA System besitzt außerdem eine eindeutige Produkt-ID. Eine Produktbeschreibung in PIA muß nicht alle Eigenschaften dieses Produktes enthalten, stattdessen können weitere Detailinformationen zu dem Produkt in einem URL Objekt gespeichert werden. Ein Beispiel für ein Produkt in PIA wäre:

> (1345, {(Produkt Name, BMW 328i), (Produkt Kategorie, Sportwagen), (Preis, 30000 USD), (Durchschnittlicher Benzinverbrauch, (11 1/100km)), (Maximale Geschwindigkeit, 230 Km/h),...})

Jedes Attribut in PIA kann durch seinen Namen identifiziert werden und beruht zugleich auf einer Domäne, die alle möglichen Werte für dieses Attribut und eine Menge von Prädikaten (s.u. Fuzzy-Prädikate von Domänen), die für die Werte dieses Attributs definiert sind, bestimmt. Mehrere Attribute dürfen dieselbe Domäne haben.

In PIA wird die Produktkategorie eines Produkts wie eine normale Produkteigenschaft behandelt. Es ist nicht einmal notwendig, daß ein Produkt in PIA überhaupt einer Kategorie zugeordnet wird. Daher ist auch nicht von vornherein festgelegt, welche Eigenschaften ein Produkt hat. Als Konsequenz daraus hat ein Anbieter freien Spielraum beim Hinzufügen neuer Eigenschaften zu einem Produkt und beim Weglassen bestimmter Eigenschaften, deren Werte unbekannt oder nicht spezifiziert sind.

Domänen und Attribute

Obwohl kein Standardschema für Produktattribute existiert, wird in PIA über die Festlegung von Domänen und den dazugehörigen Werten dennoch ein Satz von Typregeln bereitgestellt, die für die Konsistenz und Nutzbarkeit des Katalogs von wesentlicher Bedeutung sind. Domänen und Werte existieren unabhängig von den Produkten und werden bei der Definition möglicher Produkteigenschaften verwendet. Sie bilden eine gemeinsame Sprache für die Kommunikation zwischen Anbietern und Kunden.

Wie schon in der Analysephase erkannt worden ist, haben die Attribute unterschiedliche Typen. Attribute können textuell sein, wie z.B. Beschreibung eines Produkts, oder numerisch sein, wie z.B. Preis. Wieder andere Attribute können einen anderen Datentyp haben, der aus einer endlichen Menge von möglichen Werten besteht, wie z.B. es bei Prozessortyp der Fall ist. Dementsprechend wird in PIA zwischen drei Kategorien von Domänen unterschieden:

- Volltextdomäne
- Numerische Domäne

- Diskrete Domäne.

Außer der Definition der Wertebereiche für Attribute sind in Domänenobjekten auch die Informationen zu Prädikaten gekapselt. Näheres zu Prädikaten findet sich im Abschnitt "Fuzzy-Prädikate der Domänen". Darüber hinaus ist in einer Volltext-Domäne eine Festlegung der Sprache aus einer Auswahl von verschiedenen Sprachen möglich, während bei einer diskreten Domäne die hierarchische Anordnung aller möglichen Werte im Vordergrund steht. Die numerischen Domänen lassen sich noch weiter untergliedern. Hier unterscheiden wir zwischen Domänen für Attributwerte, die in bestimmten Maßeinheiten gemessen sind, und denjenigen für Attributwerte, die innerhalb eines bestimmten Intervalls liegen sollen, sowie denjenigen für Attributwerte, die diese beiden Kriterien gleichzeitig erfüllen oder nicht erfüllen [Zhang98].

Suchanfrage von Kunden

Eine Suchanfrage besteht aus einer Menge von Suchkriterien. Jedes Suchkriterium ist wiederum aus einem Prädikat auf einem Attribut sowie aus einem möglichen Gewicht zusammengesetzt. Jedes Prädikat kann dabei auch negiert werden, um auszudrücken, daß der Kunde insbesondere an den Produkten, die die genannte Eigenschaft nicht besitzen, interessiert ist.

Das Gewicht eines Kriteriums drückt seine relative Wichtigkeit im Verhältnis zu den anderen Kriterien in der Suchanfrage aus. Aus technischer Sicht ist ein Gewicht eine Zahl zwischen 0 und 1. Um die Kommunikation mit Kunden zu vereinfachen, werden die Gewichte jedoch durch Strings wie „sehr wichtig“, „wichtig“, „normal“, „nicht so wichtig“, „wäre schön, wenn die Eigenschaft vorhanden ist“ repräsentiert. Die Kriterien werden implizit durch „und“ miteinander verknüpft.

Fuzzy Prädikate von Domänen

Wie oben schon beschrieben, legen Domänen außer dem Wertebereich eines Attributes auch die verfügbaren Prädikate fest. Für jede Klasse von Domänen werden Prädikate vordefiniert, die von PIA unterstützt werden. Aus dieser Menge können bei der Bearbeitung eines Domänen-Objektes beliebige ausgewählt werden. Diese werden dann später beim Zusammenstellen einer Suchanfrage für das jeweilig gewählte Attribut angezeigt. In der folgenden Tabelle kann man einige der im Prototyp bereits verwendeten Prädikate für die jeweilige Klasse von Domänen entnehmen. Für die gesamte Menge der Prädikate sei auf die Arbeit [Büchner99] verwiesen.

Name des Prädikats	Übergebene Argumente
"is exactly"	(DiscreteDomain)
"is subsumed by"	(DiscreteDomain)
"is subsumed by one of"	(DiscreteDomain, DiscreteDomain[])
"is similar to"	(DiscreteDomain)
"is exactly"	(NumericDomain)
"is approximately"	(NumericDomain)
"is between"	(NumericDomain, NumericDomain)
"is larger than"	(NumericDomain)
"is smaller than"	(NumericDomain)
"is average"	()
"is large"	()
"is small"	()
"is as large as possible"	()
"is as small as possible"	()
"contains"	(FulltextDomain)

Tabelle 1. Fuzzy-Prädikate in Überblick

Suchergebnis (Offer)

Das Suchergebnis für eine Suchanfrage in PIA besteht aus einer Menge von Treffern. Ein Treffer unterscheidet sich von einem Produkt dadurch, daß er mit einer reellen Zahl (*Ranking*) zwischen 0 und 1 versehen wird, die als Relevanz bezeichnet wird. Diese Zahl wird von PIA beim Ausführen einer Suchanfrage ermittelt und beim Anzeigen des Suchergebnisses als die erste Spalte in der Tabelle dargestellt.
Das erarbeitete Objektmodell enthält die wichtigsten Business-Klassen im PIA System sowie ihre Beziehungen zueinander und dient als die Grundlage für die Weiterentwicklung des gesamten Systems. Basierend auf diesem Modell wird im Folgenden die Benutzerschnittstelle für das System entworfen und realisiert.

5.2 Entwurf der Benutzerschnittstelle

Die graphische Benutzerschnittstelle stellt eine der wichtigsten Komponenten eines Informationssystems dar. Mit ihr werden Informationen erfragt, Datenpflege betrieben und Prozeßabläufe gesteuert. In diesem Abschnitt werden die Interfaceklassen, die sich in der Analysephase herauskristallisiert haben, sowie ihre Beziehungen zueinander erläutert.

5.2.1 „UML“-Klassendiagramme

Klassendiagramme in UML werden zur Veranschaulichung von Klassenmodellen verwendet. Sie stellen die statische Klassenstruktur aller identifizierten Objekte dar und beschreiben ihre Beziehungen zueinander mittels Assoziationen. Bei den Assoziationen handelt es sich um „Vererbungsbeziehung“, „Aggreagation“ oder „Komposition“ zwischen den Klassen. Sie können in uni- oder bidirektionaler Form auftreten, was später in der Implementierung Einfluß auf die Art der Verkettung von Objekten hat und wesentlich die Navigation zwischen Objekten bestimmt. Zusätzlich besitzt jede Assoziation eine Kardinalität, die angibt, ob es sich um eine 1:1-, 1:n- oder n:m-Beziehung handelt.
Neben Assoziationen werden auch Attribute und Methoden aller aufgeführten Klassen beschrieben. Attribute und Methoden werden durch ihren Namen und Angabe ihres Typs bzw. ihrer Signatur spezifiziert. Das Klassendiagramm hat demnach einen deklarativen Charakter und bezieht sich überwiegend auf die Typebene. Es bleibt zu erwähnen, daß eine Generierung von Programmrümpfen aus dem Klassendiagramm heraus möglich ist, da alle wesentlichen Informationen vorhanden sind.
Auf eine Einführung zur Notation von Klassendiagrammen in UML muß verzichtet werden. Diese kann aber in [Fowler97a, Oestereich97] nachgelesen werden.
Im folgenden wird ein Überblick über die wichtigsten Klassen im PIA-Frontend-System gegeben. Einleitend werden die Entwurfsmuster, die in der Arbeit verwendet werden, erläutert.

5.2.2 Entwurfsmuster

Ein Muster zielt auf ein in speziellen Entwurfssituationen häufig auftretendes Entwurfsproblem ab und beschreibt eine generische Lösung für dieses Problem. Entwurfsmuster stellen einen vielversprechenden Ansatz für die Entwicklung von Software-Systemen mit definierten Eigenschaften dar. Man kann sie zusammen mit jedem Programmierparadigma nutzen und fast in jeder Programmiersprache implementieren. Ein Satz von wichtigen Entwurfsmustern wird in [Gamma95] dargestellt. Nachfolgend werden die wichtigsten Muster, die hauptsächlich im PIA Frontend System eingesetzt werden, beschrieben und ihr Einsatz erläutert.

Das Entwurfsmuster „Singleton“

Dieses Muster gewährleistet, daß eine Klasse genau ein Exemplar besitzt, und stellt einen globalen Zugriffspunkt darauf bereit. Um sicherzustellen, daß eine Klasse über genau ein Exemplar verfügt und daß einfach auf dieses Exemplar zugegriffen werden kann, wird die

Klasse selbst für die Verwaltung ihres einzigen Exemplars verantwortlich gemacht. Die folgende Abbildung zeigt die Struktur des Singletonmusters.

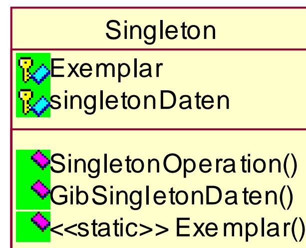


Abbildung 5-3: Struktur des Singletonmusters

Die Singletonklasse verfügt nur über private Konstruktoren, die von außen nicht aufrufbar sind. Das Erzeugen von Objekten dieser Klassen wird in der statischen Operation *Exemplar()* gekapselt, die dafür sorgt, daß nur eine einzige Singleton-Instanz angelegt und eine Referenz auf diese Instanz zurückgeliefert wird. Clienten können ausschließlich durch diese Exemplar-Operation eine Referenz auf das Singletonobjekt erhalten. Das Singletonmuster hat mehrere Vorteile:

- Zugriffskontrolle auf das Exemplar: Der Konstruktor in einer Singletonklasse ist geschützt, und Clienten können ausschließlich durch die statische *Exemplar()* Methode auf das Singleton zugreifen. Die Singletonklasse kapselt somit ihr einziges Exemplar und kann genau kontrollieren, wie und wann die Clienten auf das einzige Exemplar zugreifen.
- Einfacher Zugriff auf das einzige Exemplar: Das Singletonmuster bietet insbesondere einen einfachen globalen Zugriffspunkt auf das einzige Exemplar.
- Eingeschränkter Namensraum: Das Singletonmuster ist eine Verbesserung gegenüber globalen Variablen.
- Verfeinerung von Operationen und Repräsentationen: Die Singletonklasse kann abgeleitet und spezialisiert werden.
- Variable Anzahl von Exemplaren: Das Singletonmuster kann leicht abgeändert und dazu verwendet werden, die Anzahl der von den Anwendungen benutzten Exemplare zu steuern.

Das Entwurfsmuster „Observer“

Das Muster Observer (Beobachter) definiert eine 1-zu-n-Abhängigkeit zwischen Objekten, so daß die Änderung des Zustandes eines Objektes dazu führt, daß alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.

Teilt man ein System in eine Menge von interagierenden Klassen auf, so ergibt sich häufig der Nebeneffekt, daß die Konsistenz zwischen den miteinander in Beziehung stehenden Objekten aufrechterhalten werden muß. Man möchte diese Konsistenz üblicherweise nicht dadurch sicherstellen, daß man diese Klassen eng miteinander koppelt, weil dies ihre Wiederverwendbarkeit einschränkt.

Das Beobachter-Muster beschreibt, wie man diese Beziehungen etabliert. Die zentralen Objekte in diesem Muster heißen Subjekt (Model) und Beobachter (View). Ein Subjekt kann eine beliebige Anzahl von abhängigen Beobachtern besitzen. Alle Beobachter werden benachrichtigt, wenn das Subjekt seinen Zustand ändert (vgl. Abschnitt 3.5.3.2 Das Model-View-Controller (MVC) Konzept in Swing). Die folgende Abbildung zeigt die Struktur dieses Musters.

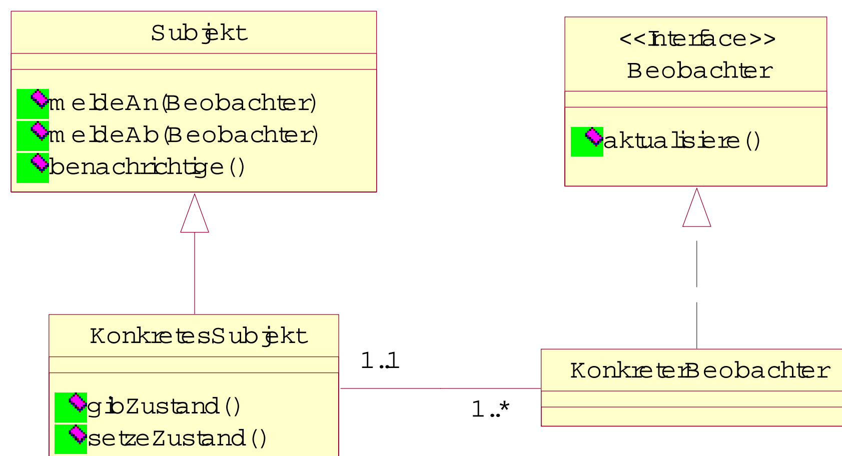


Abbildung 5-4: Struktur des MVC Musters

Das Entwurfsmuster „Template Methode"

Eine Template Methode, auch als „Schablonenmethode" bekannt, ist ein objektbasiertes Verhaltensmuster. Dieses Muster definiert das Skelett eines Algorithmus in einer Operation und delegiert einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.

Unter Verwendung dieses Musters kann man insbesondere folgendes erreichen:

- Die invarianten Teile eines Algorithmus werden genau einmal festgelegt, und es wird den Unterklassen überlassen, das variierende Verhalten zu implementieren.
- Die Verdoppelung von Code kann so vermieden werden, weil das gemeinsame Verhalten von Unterklassen herausfaktorisiert und in einer allgemeinen Klasse plaziert wird.

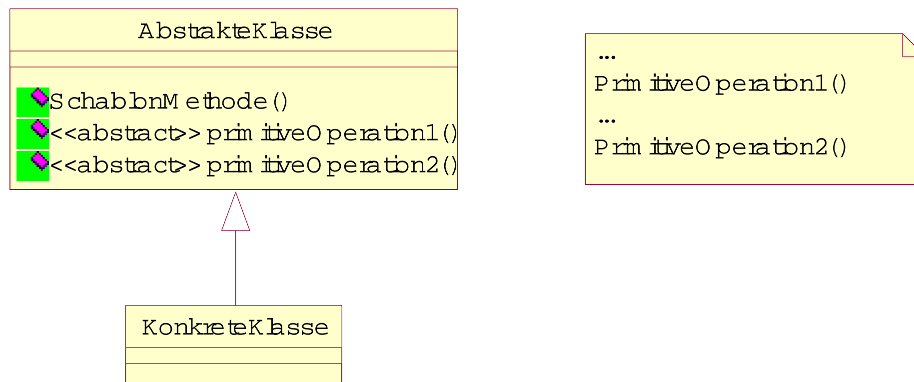


Abbildung 5-5: Struktur des Musters Template Methode

Das Entwurfsmuster „Mediator“

Das Muster Mediator, auch als Vermittler bekannt, definiert ein Objekt, welches das Zusammenspiel einer Menge von Objekten in sich kapselt. Vermittler fördern lose Kopplung, indem sie Objekte davon abhalten, aufeinander explizit Bezug zu nehmen.
Objektorientierter Entwurf fördert die Verteilung von Verhalten zwischen Objekten. Als Konsequenz dieser Verteilung kann aber eine Objektstruktur mit vielen Beziehungen zwischen den Objekten entstehen. Im schlimmsten Fall bedeutet dies, daß jedes Objekt jedes andere kennt. Viele Verbindungen zwischen den Objekten zu haben, reduziert die Wiederverwendbarkeit der einzelnen Objekte, weil sie ohne Unterstützung anderer Objekte nicht arbeiten können. Weiterhin kann es schwierig sein, das Verhalten des Systems auf bedeutsame Weise zu ändern, weil es über so viele Objekte verstreut ist.
Das Problem läßt sich vermeiden, indem das Verhalten in einem separaten Vermittlerobjekt gekapselt wird. Ein Vermittler ist für die Kontrolle und Koordination der Interaktion innerhalb einer Gruppe von Objekten zuständig. Der Vermittler hält die Objekte in einer Gruppe davon ab, direkt aufeinander Bezug zu nehmen. Die Objekte kennen nur den Vermittler, wodurch die Verbindungen reduziert werden.

5.2.3 Entwurf einer interaktiven Benutzerschnittstelle

Die Entwicklung sehr großer Softwaresysteme ist nur zu bewältigen, wenn das gesamte System in einzelne Komponenten zerlegbar ist und die eingesetzte Programmiersprache entsprechende unterstützende Mittel dazu liefert. Das PIA-Frontend-System basiert auf dem in [Matthes99] entwickelten Datenmodell und wird seinerseits wieder in mehrere Pakete zerlegt. Beim Bilden der Pakete werden die funktionalen und die logischen Zusammenhänge zwischen den Klassen berücksichtigt, so daß die Klassen, die eng miteinander

zusammenarbeiten, in einem Paket zusammengefaßt und dadurch die Abhängigkeiten zwischen den Paketen minimiert werden können, was eine bessere Struktur und Erweiterbarkeit des Systems ausmacht. Im folgenden Diagramm werden die wichtigsten Pakete des PIA Frontends dargestellt.

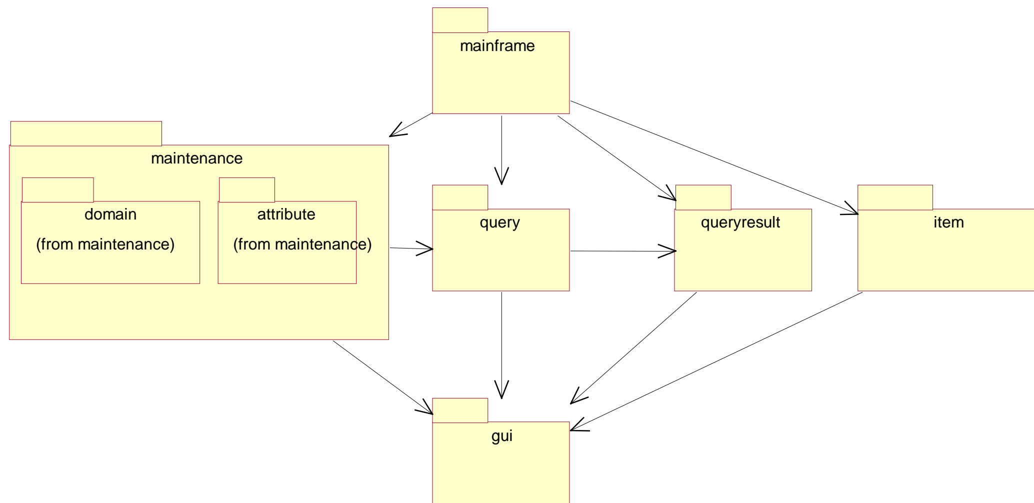


Abbildung 5-6. Pakete in PIA Frondend System

Das Paket "pia.mainframe"

Die Klassen im Paket *mainframe* stellen das Hauptfenster für jeden PIA Benutzer zusammen. Die Klasse *MainFrame* aggregiert ein *LoginDialog*, wo ein Benutzer seinen Benutzernamen und sein Paßwort eingeben kann, eine *IconToggleButtonList*, die die Icons zum Navigieren zwischen den einzelnen Komponenten des Hauptfensters enthält (vgl. Abb. 5.7), sowie ein *contentPane* der Klasse JPanel (als ein Attribut in der Klasse *MainFrame* deklariert), das zum Anzeigen der mit dem jeweiligen selektierten Icon assoziierten graphischen Benutzerschnittstellen-Komponente dient. Welche Komponenten das im einzelnen sein können, wird weiter unten erklärt. Ein weiterer Bestandteil des Hauptfensters ist ein *Statusbar*, das im unteren Teil lokalisiert ist und dem Benutzer den jeweiligen Prozeßstatus vermittelt. Dieses *Statusbar* kann zum Beispiel beim Zusammenstellen einer Suchanfrage verwendet werden, um den Benutzer zu informieren, wieviele Treffer sich für die momentane Anfrage ergeben würden.

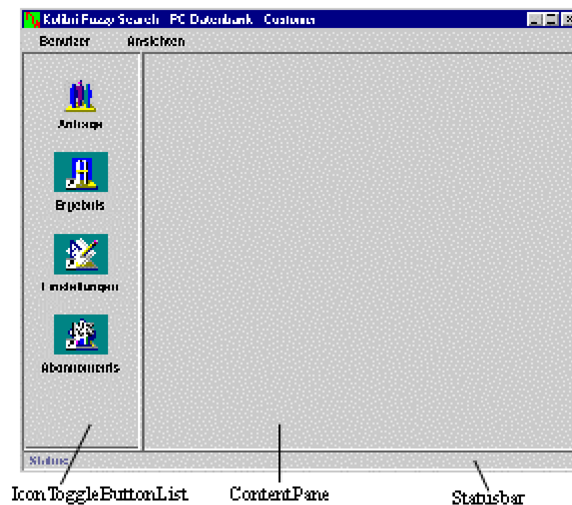


Abbildung 5-7. Die Klassen IconToogleButtonList, ContentPane und Statusbar im Paket mainframe

Wie schon verschiedentlich angedeutet wurde, bedienen sich PIA Benutzer unterschiedlicher Klassen (Kunde, Anbieter oder Redakteur) verschiedener Benutzerschnittstellen, um ihre individuellen Ziele und Aufgaben zu erfüllen. Jeder Benutzer wird beim Anmelden von PIA identifiziert, und anhand der festgestellten Klasse des Benutzers werden die geeigneten Benutzerschnittstellen-Komponenten bereitgestellt. In einem *UserProfile* werden die Informationen darüber gespeichert, welche Icons für welche Benutzerklasse definiert sind, und diese können durch Aufruf der Methode *getButtons()* der Klasse *UserProfile* abgefragt werden. Da wir in PIA drei Klassen von Benutzern unterscheiden, wird an dieser Stelle Polymorphie verwendet. Die Klassen *CustomerProfile, MaintainerProfile* und *ProviderProfile* sind Unterklassen der gemeinsamen abstrakten Klasse *UserProfile* und erben die abstrakte Methode *getButtons()* von der Oberklasse. Beim Überschreiben dieser Methode werden jedoch unterschiedliche Implementationen vorgenommen. Welches graphische Icon mit welcher Komponente assoziiert ist, kann bei der Singleton-Instanz *PiaWindowManager* durch Aufruf der Methode *getComponent(String key, Boolean flag)* abgefragt werden. Die boolesche Variable *flag* gibt an, ob eine Komponente neu erzeugt werden soll. Die wichtigsten der erwähnten Klassen und ihre Beziehungen zueinander werden in dem folgenden Klassendiagramm dargestellt (vgl. Abb. 5.8).
Die mit den Icons in dem *IconToggleButtonList* verbundenen Komponenten, die in dem rechten Teil des Hauptfensters erscheinen, werden jeweils in den Paketen **maintenance**, **query**, **queryresult**, **item** implementiert und in den folgenden Abschnitten detailliert beschrieben. Die Beziehungen zwischen den Paketen sind bereits durch die Pfeile in der Abb. 5.6 repräsentiert.

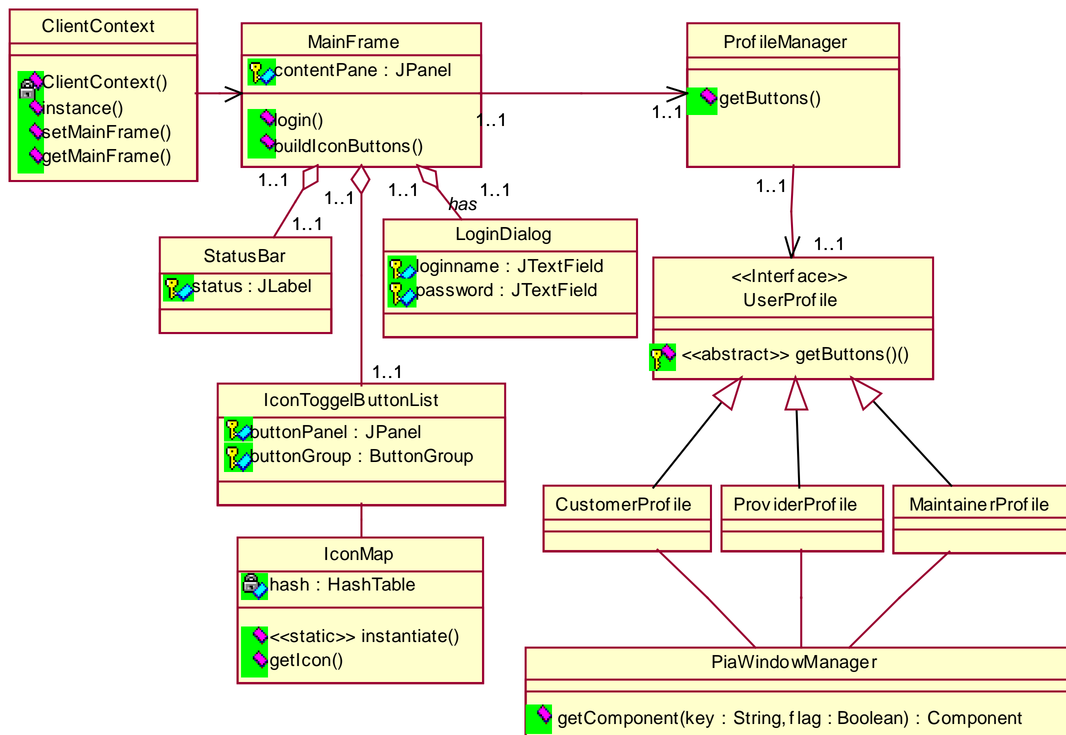


Abbildung 5-8. Klassen im Paket *mainframe*

Das Paket "pia.maintenance"

Wie man aus dem Namen des Pakets entnehmen kann, stellt das Paket **maintenance** die Benutzeroberfläche für Redakteure bereit. Zwar setzt das Einfügen eines Attributs in das PIA System das Vorhandensein der dafür vorgesehenen Domäne voraus. Die Klassen für die Benutzerschnittstellen zur Verwaltung von Domänen auf der einen und Attributen auf der anderen Seite sind jedoch relativ unabhängig voneinander. Aus diesem Grund wird das Paket noch einmal in zwei Unterpakete geteilt, die die jeweilige Implementation enthalten – das Paket **attribute** zur Verwaltung der Attribute und das Paket **domain** zum Pflegen der Domänenobjekte (vgl. Abb. 5.6).

Die wichtigsten Komponenten im Paket **maintenance** sind der *DomainEditorPanel* und der *AttributeEditor*, die jeweils zum Editieren einer Domäne/eines Attributs dienen. Da bei den verschiedenen Domänenkategorien (numerische Domänen, diskrete Domäne oder Volltext-Domäne) die Aufgaben bei der Bearbeitung der einzelnen Domänen unterschiedlich sind, werden auch hier mehrere Unterklassen von *DomainEditor* implementiert. Dadurch findet die Polymorphie auch hier wieder Gebrauch, was sich im Diagramm in der Abbildung 5.9 nachvollziehen läßt. Die Aufgaben der Domänenbearbeitung werden mit Hilfe des *DomainEditorPanel*s, dessen Hauptbestandteil eine Registerkarte ist, durchgeführt. Die

Bauteile der Registerkarte sind bei verschiedenen Domänenklassen zwar sehr unterschiedlich, jedoch werden beim Aufbau des Editors ähnliche Schritte durchlaufen. Zuerst wird ein Namensfeld im Editor bereitgestellt, das später mit Inhalt (Name der Domäne) aufgefüllt werden kann (*updateContent()*). Danach wird die Registerkarte mit ihren Teilkomponenten zusammengestellt und zum Editor hinzugefügt (*fillTabs()*). Dies spricht für den Einsatz des Entwurfsmusters Templatemethode**,** dessen Einsatz sich anhand des folgenden Diagramms (vgl. Abb. 5.10) ebenfalls erkennen läßt.

Das Paket "pia.maintenance.domain"

Ein DomainPanel dient als Platzhalter für die ihm übergebenen Domänenobjekte. Es stellt diese Domänen in einem Baum dar (vgl. Abb. 5.9), der einen *MouseListener* als Beobachter bei sich registriert hat. Beim Selektieren und Doppelklick auf einen Baumknoten, der für ein Domänenobjekt steht, wird ein Dialogfenster erzeugt, und durch Aufruf der Methode *showValueInDialog(AbstractDomain*[2] *domain)* wird das Domänenobjekt in diesem Dialogfenster dargestellt.

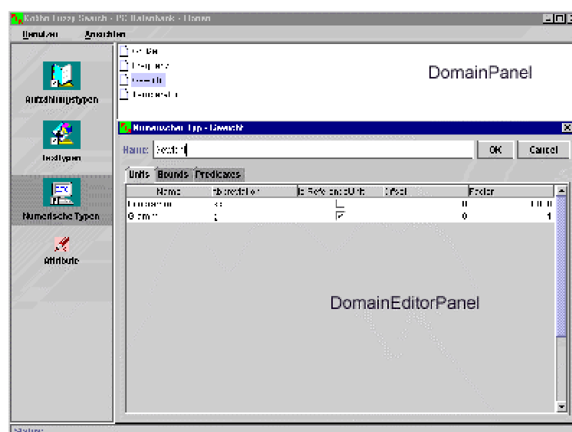


Abbildung 5-9. DomainPanel und DomainEdiotorPanel

Dadurch, daß sowohl die Klasse *DomainPanel* als auch die Klasse *DomainEditorPanel* spezialisiert wird, kann der Einsatz einer Singleton-Instanz an dieser Stelle vermieden werden. In der Klasse *DomainPanel* wird eine abstrakte Methode *getDomainEditorPanel(AbstractDomain domain)* deklariert, die innerhalb des Methodenrumpfs von *showValueInDialog(AbstractDomain domain)* verwendet wird, um festzustellen, durch welche Art von *DomainEditorPanel* die selektierte Domäne darzustellen ist. Diese Methode wird in der jeweiligen Unterklasse implementiert und liefert eine

[2] Die Klasse *AbstractDomain* ist eine abstrakte Klasse von dem Package *piamodel* und entspricht der Domänenklasse, die in dem Abschnitt 5.1 beschrieben wurde (vgl. [Matthes99]).

Komponente zurück, die mindestens so spezifisch wie das *DomainEditorPanel* ist. Durch die Spezialisierung der Klasse *DomainEditorPanel* kann z.B. die 1:1 Beziehung zwischen einem numerischen Domänenobjekt und einem *NumericDomainEditorPanel* ohne Einsatz einer Singleton-Instanz dadurch realisiert werden, daß in einem *NumericDomainEditorPanel* nur numerische Domänen dargestellt werden und die Methode *getDomainEditorPanel()* in dieser Klasse immer genau dieselbe Instanz der Klasse *NumericDomainEditorPanel* zurückliefert.

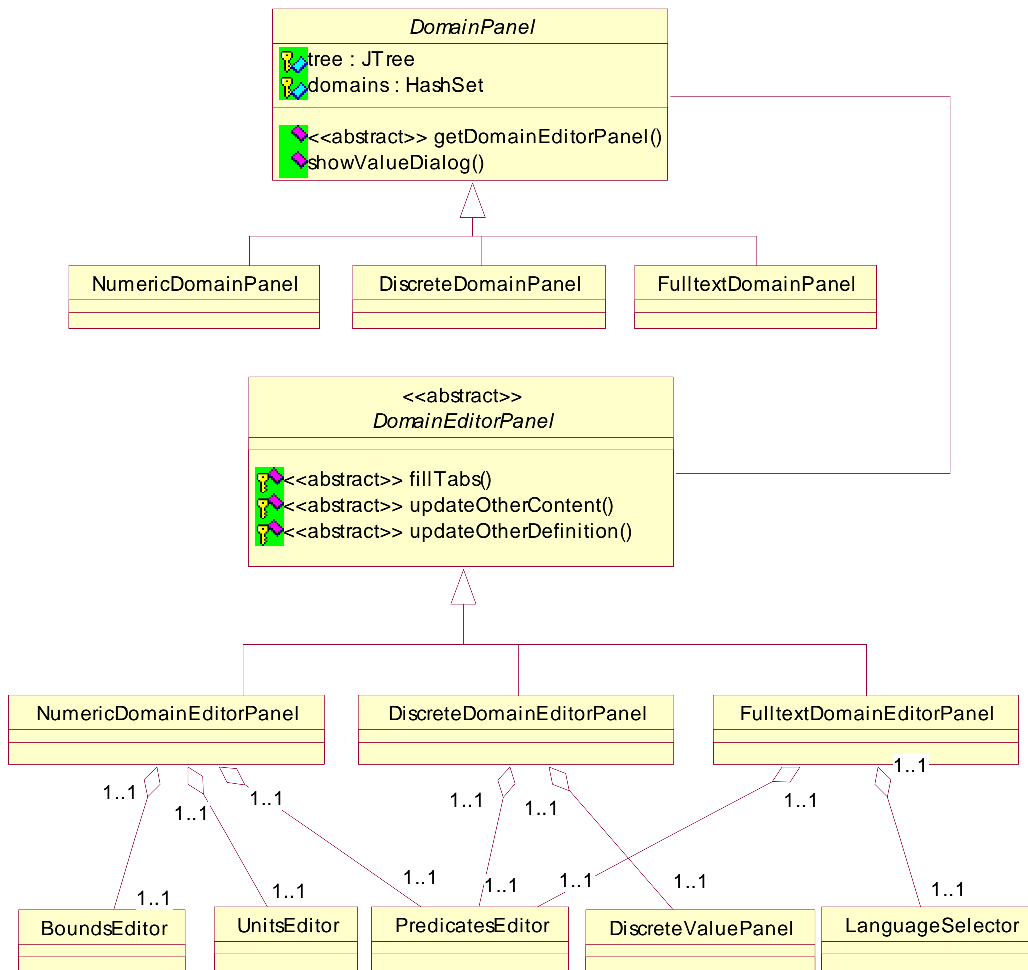


Abbildung 5-10. DomainEditor und seine Unterklassen

Die Klasse "*DomainEditorPanel*"

Wie oben schon beschrieben wurde, besteht ein *DomainEditorPanel* hauptsächlich aus einem Namensfeld und mehreren Registerkarten (vgl. Abb. 5.9). Die Anzahl der Registerkarten hängt von der jeweiligen Domänenklasse ab. Bei einer numerischen Domäne sind z.B. drei Registerkarten im *DomainEditorPanel* enthalten (vgl. Abb. 5.11). Um das Namensfeld und die Registerkarten eines *DomainEditorPanel*s zusammenzustellen und diese jeweils mit dem adäquaten Inhalt aufzufüllen, werden bei der Konstruktion eines

DomainEditorPanels die Methoden *fillTabs()* und *updateOtherContents()* nacheinander aufgerufen. Diese Methoden werden wegen des Mangels an Informationen zunächst als abstrakt deklariert und erst in den Unterklassen implementiert. Die Methode *filltabs()* stellt die Komponenten in den Registerkarten im Editor zusammen. Eine gemeinsame Registerkarte in allen Editoren ist die Komponente *PredicateSelector*, welche die selektierten Anfragemuster (Prädikate) für die gerade zu bearbeitende Domäne und die Gesamtmenge aller möglichen Anfragemuster für diese Domänenkategorie gegenüberstellt, wie in der Abbildung 5.11 zu sehen ist. Die Methoden *updateContent()* und *updateOtherContent()* sorgen schließlich dafür, daß das Namensfeld und die Komponenten in den Registerkarten mit dem entsprechenden Inhalt aufgefüllt werden.

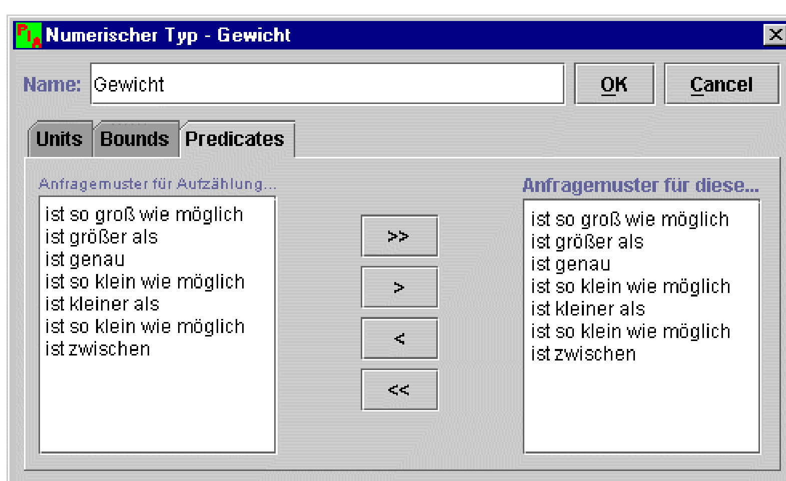


Abbildung 5-11. Ein *PredicateSelector* als Bestandteil der Registerkarten

Bei einem *DiscreteDomainEditorPanel* gehört zu den Registerkarten noch eine weitere Komponente namens *DiscreteValueEditor*, die speziell für die Definition hierarchisch geordneter Werte geeignet ist und daher erst in dieser Unterklasse generiert und zu den Registerkarten hinzugefügt wird. Dieser Editor bietet ein Popupmenü mit den Menüeinträgen „Wert hinzufügen", mit dem ein Kind zu dem markierten Knoten im Baum hinzugefügt werden kann, „löschen", mit dem man den markierten Knoten aus dem Baum löschen kann sowie „umbenennen", mit dem man den gegenwärtig markierten Knoten umbenennen kann. In der folgenden Abbildung (Abb. 5.12) wird als Beispiel der Editor für das Editieren einer diskreten Domäne CPU Typ illustriert.

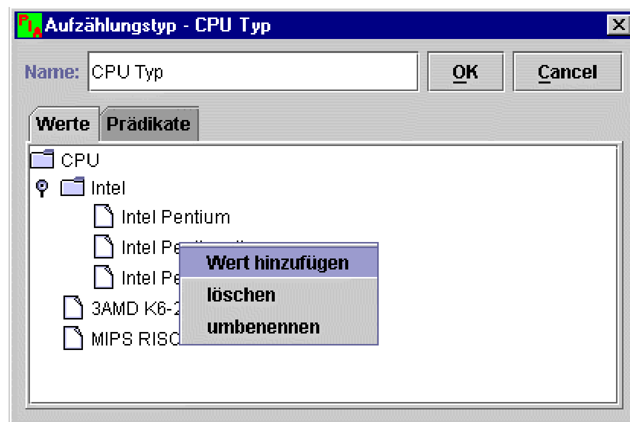


Abbildung 5-12. Editor für Werte einer hierarchischen Domäne

Beim Editieren einer numerischen Domäne sind weitere Aufgaben wie Festlegung von keinem, einem oder zwei Grenzwerten oder die Angabe einer Referenzeinheit und mehrerer abgeleiteter Maßeinheiten erforderlich. Bei der Überschreibung der Methode *filltabs()* werden die für diese Aufgaben vorgesehenen graphischen Komponenten – der *BoundsEditor* und der *UnitsEditor* – bereitgestellt und zu den Registerkarten hinzugefügt. Mit einem *UnitsEditor* können die Einheiten, die zu einer numerischen Domäne gehören, definiert werden. Die definierten Einheiten werden tabellarischer Form dargestellt, wobei eine Einheit als Referenzeinheit ausgezeichnet und mit einem aktivierten Checkbox versehen wird. Alle anderen Einheiten werden als abgeleitete Einheiten bezeichnet und besitzen Angaben wie einen Faktor (*factor*) und einen Summanden (*offset*) für die Umrechnung in die dazugehörige Referenzeinheit. Allgemein gilt dabei:

1 abgeleitete Einheit = 1 Referenzeinheit *Faktor + Summand

Beispielhaft werden in der folgenden Abbildung (Abb. 5.13) die Einheiten der Domäne „Gewicht“ dargestellt, wobei die momentan vorhandenen Einheiten durch die Bezeichnungen „Gramm“ und „Kilogramm“ repräsentiert werden.

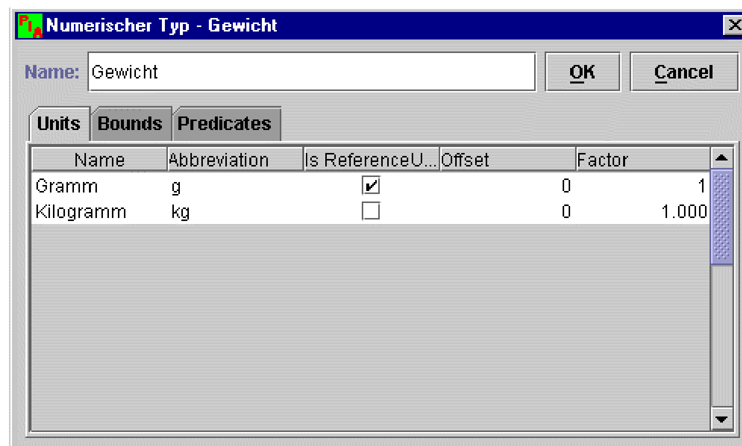


Abbildung 5-13. Ein *UnitsEditor als Bestandteil der Registerkarte* in einem *NumericDomainEdiorPanel*

Ein *BoundsEditor* enthält zwei *NumericFields* zur optionalen Eingabe von einem oder von zwei Grenzwerten für die Domäne. Diese Grenzwerte legen fest, in welchem Bereich die Ausprägungen der Attribute, die dieser Domäne zugeordnet sind, fallen sollen. Wenn z.B. ein Kunde beim Suchen versucht, eine Ausprägung für ein so beschränktes Attribut außerhalb des Bereichs anzugeben, wird vom System eine Fehlermeldung generiert.
Beim Editieren einer Volltextdomäne kann man mit Hilfe eines *LanguageSelectors* die Sprache festlegen. Diese Komponente stellt alle zur Verfügung stehenden Sprachen hinter Radiobuttons in einer einzigen Gruppe dar, so daß eine multiple Auswahl ausgeschlossen werden kann (vgl. Abbildung 5.14). Die Auswahl gibt Auskunft darüber, in welcher Sprache die textuellen Beschreibungen interpretiert werden sollen.

Abbildung 5-14. Ein *LanguageSelector* für einen *FulltextDomainEditor*

Das Paket "pia.maintenance.attribute"

Das Anlegen eines neuen Attributs im PIA System setzt die Existenz der passenden Domäne voraus. Aus diesem Grund werden alle vorhandenen Domänenobjekte in einem nicht editierbaren Baum dargestellt. Die Baumdarstellung ordnet die Domänenobjekte gleichzeitig einer der drei Kategorien (numerische Domänen, diskrete Domänen und Volltextdomänen) in verschiedenen Pfaden zu, so daß die Auswahl aus der Vielfalt der Domänen vereinfacht wird (vgl. Abb. 5-16). Der Aufbau des Pakets ist im folgenden Diagramm (Abb. 5.15) illustriert. Während ein *AttributePanel* alle ihm übergebenen Attributobjekte darstellt, kann mit einem *AttributeEditorPanel* ein einzelnes Attribut editiert werden. Hier wird ebenfalls das Beobachter-Muster als Designmuster verwendet. Die Klasse *AttributeEditorPanel* bietet die Schnittstelle *addAttributeChangeListener (AttributeChangeListener listener)* und *removeAttributeChangeListener (AttributeChangeListener listener)* zum Anmelden/Abmelden von Beobachtern, die sich für bestimmtes Ereignis im *AttributeEditorPanel* interessieren. Dafür implementiert die Klasse *AttributePanel* das

Interface *AttributeChangeListener,* in dem zwei abstrakte Methoden *attributeCreated()* und *attributeChanged()* deklariert werden, die durch die implementierenden Klassen überschrieben werden können.

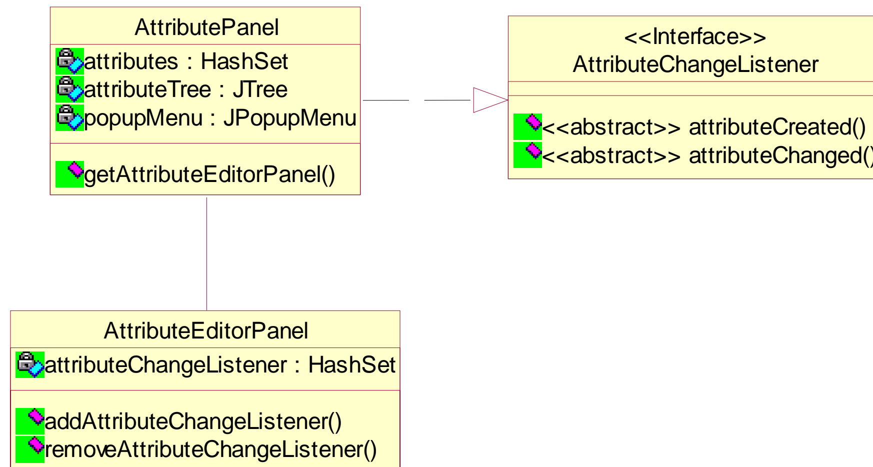


Abbildung 5-15. Das Paket *pia.maintenance.attribute*

In der folgenden Abbildung (Abb. 5.16) wird die Benutzerschnittstelle für das Editieren eines Attributs (*CPU-Typ*) dargestellt. In dem Namensfeld kann man den Namen des Attributs ändern, durch Selektieren einer Domäne (z.B. „CPU Typ" in der Abbildung) aus dem Domänenbaum kann man die Domäne für das Attribut festlegen bzw. ändern.

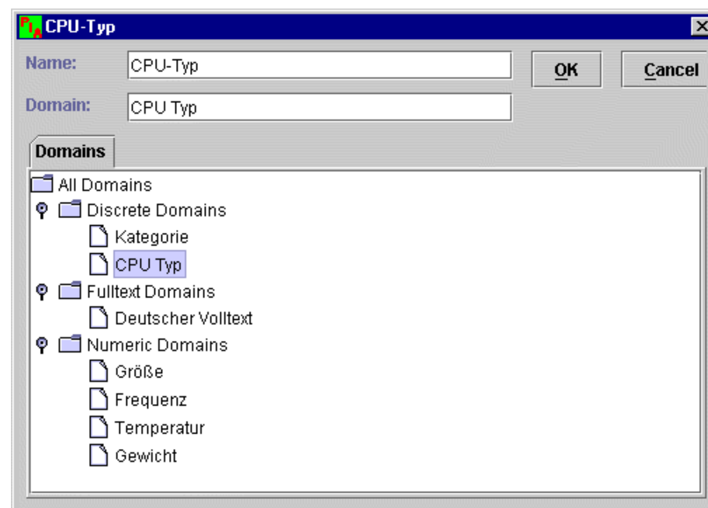


Abbildung 5-16. *AttributeEditor*

Das Paket "pia.query"

Zum Paket **query** gehören die Klassen, die die Benutzerschnittstelle für das Zusammenstellen einer Suchanfrage und für das Formatieren des Suchergebnisses zur Verfügung stellen. Der Editor für eine Suchanfrage wird in PIA als ein *AlternativeEditor* bezeichnet, passend dazu, daß eine Suchanfrage im Datenmodell des PIA Systems als eine *Alternative* definiert ist.

Wichtige Bestandteile eines *AlternativeEditors* sind der *AttributeLister*, der *AlternativeDisplayer* sowie ein *CriterionEditor,* wie im folgenden Diagramm (vgl. Abb. 5.17) illustriert. Ein *AttributeLister* stellt alle im System vorhandenen Attribute dar und bietet einem Kunden einen ersten Einstiegspunkt zum Editieren seiner Suchanfrage. Standardmäßig werden alle Attribute des Systems nach den Häufigkeiten, mit denen sie in den Produkten der Kataloge vorkommen, sortiert und angezeigt. Die aktuelle (Zwischen-) Suchanfrage wird im rechten Fensterteil - dem *AlternativeDisplayer* - in einer Tabelle dargestellt, wobei jede Zeile ein Suchkriterium repräsentiert, die die gesuchten Produkte haben sollen. Ein Kunde wählt beim Suchen ein Attribut aus dem *AttributeLister* aus, sucht ein Prädikat im *PredicateSelector* aus und legt anschließend den Wert für dieses Attribut mit Hilfe eines Dialogfensters – des *CriterionEditors* fest. Damit wird ein neues Kriterium (*Criterion*) definiert und zu der bestehenden *Alternative* hinzugefügt. Dieses neue Kriterium wird entsprechend im *AlternativeDisplayer* dargestellt. Eine neue Zeile wird zur Tabelle hinzugefügt, und so wird die Konsistenz zwischen dem *Alternative*-Objekt und dem *AlternativeDisplayer*-Objekt aufrechterhalten. Es bleibt zu erwähnen, daß jedes Kriterium mit einem sog. Fuzzy-Operator sowie einem Fuzzy-Gewicht versehen werden kann. Mit dem Fuzzy-Operator kann ein Kunde insbesondere zum Ausdruck bringen, wenn ein bestimmtes Kriterium bei den gesuchten Produkten nicht vorhanden sein soll (Negation), während mit dem Fuzzy-Gewicht ausgedruckt werden kann, wie wichtig ein genanntes Kriterium für die gesuchten Produkte ist.

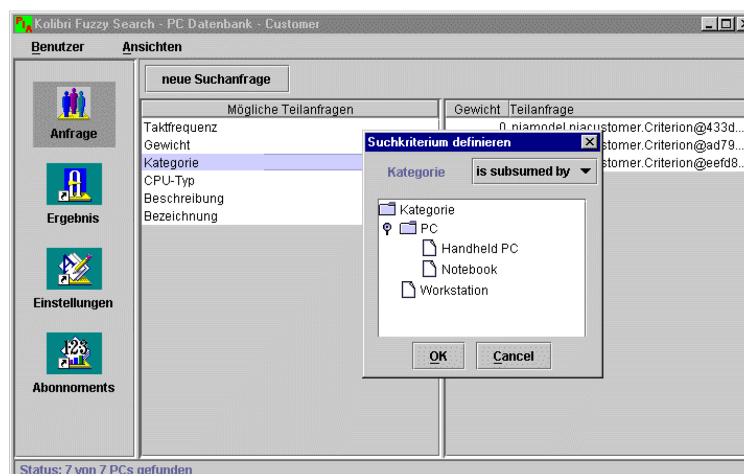


Abbildung 5-17. ***AlternativeEditor*** **und seine Bestandteile**

Allgemein besteht ein *CriterionEditor* aus einem Namensfeld für das gewählte Attribut und einem *PredicateSelector* in Form eines Pulldownmenüs, in dem alle für die Domäne des gewählten Attributs definierten Prädikate dargestellt werden, sowie aus einem *ValueSelector*, mit dem man den Vergleichswert für das Attribut festlegen kann. Wie dieser *ValueSelector* konkret aussieht, hängt nicht nur von der Kategorie der dazugehörigen Domäne, sondern auch vom selektierten Prädikat ab. Demzufolge werden auch hier Polymorphie als Designgrundlage und das Singleton-Muster sowie das Templatemethoden-Muster als Entwurfsmuster eingesetzt. Für jedes Fuzzy-Prädikat existiert eine passende *ValueEditor*-Klasse, und das Mapping zwischen dem Prädikat und dem *ValueEditor* ist in der Singleton-Instanz, dem ValueEditorManager, gekapselt (vgl. Abb. 5.18).
Die GUI-Komponenten *AlternativeEditor*, *AttributeLister,* *CriterionEditor* und *AlternativeDisplayer* arbeiten eng miteinander zusammen. Immer wenn ein Attribut aus dem *AttributeLister* gewählt wird, erzeugt ein *AlternativeEditor* einen *CriterionEditor* und übergibt das Attribute als Übergabeparameter. Wenn ein neues Kriterium mit Hilfe des *CriterionEditors* definiert wird, wird das Dialogfenster mit dem *CriterionEditor* geschlossen, und der *AlternativeDisplayer* fügt dieses neue Kriterium in seiner Tabelle als eine neue Zeile hinzu. Um zu vermeiden, daß diese Objekte sich untereinander kennen und dadurch die Wiederverwendbarkeit dieser Klassen reduziert wird, wird die Interaktion zwischen diesen Objekten durch die zentrale Instanz, den *AlternativeEditor* gesteuert. Da der *AlternativeEditor* seine Bestandteile ohnehin kennt, ist diese Anforderung programmiertechnisch einfach zu realisieren. Er registriert sich jedesmal beim Erzeugen seiner Komponenten zugleich als deren Beobachter. Die Komponentenklassen bieten jeweils die passenden Schnittstellenmethoden zum Anmelden bzw. Abmelden eines Beobachters für bestimmte Ereignisse (vgl. Abb. 5.18). Der *AlternativeEditor* dient somit als ein Vermittler zwischen seinen einzelnen Komponenten.
Die zwei Singleton-Klassen *CriterionEditorManager* und *ValueSelectorManager* kapseln die Informationen zum Mapping zwischen einer Domänenkategorie und einem *CriterionEditor* sowie zwischen einem Prädikat und einem *ValueSelector*.

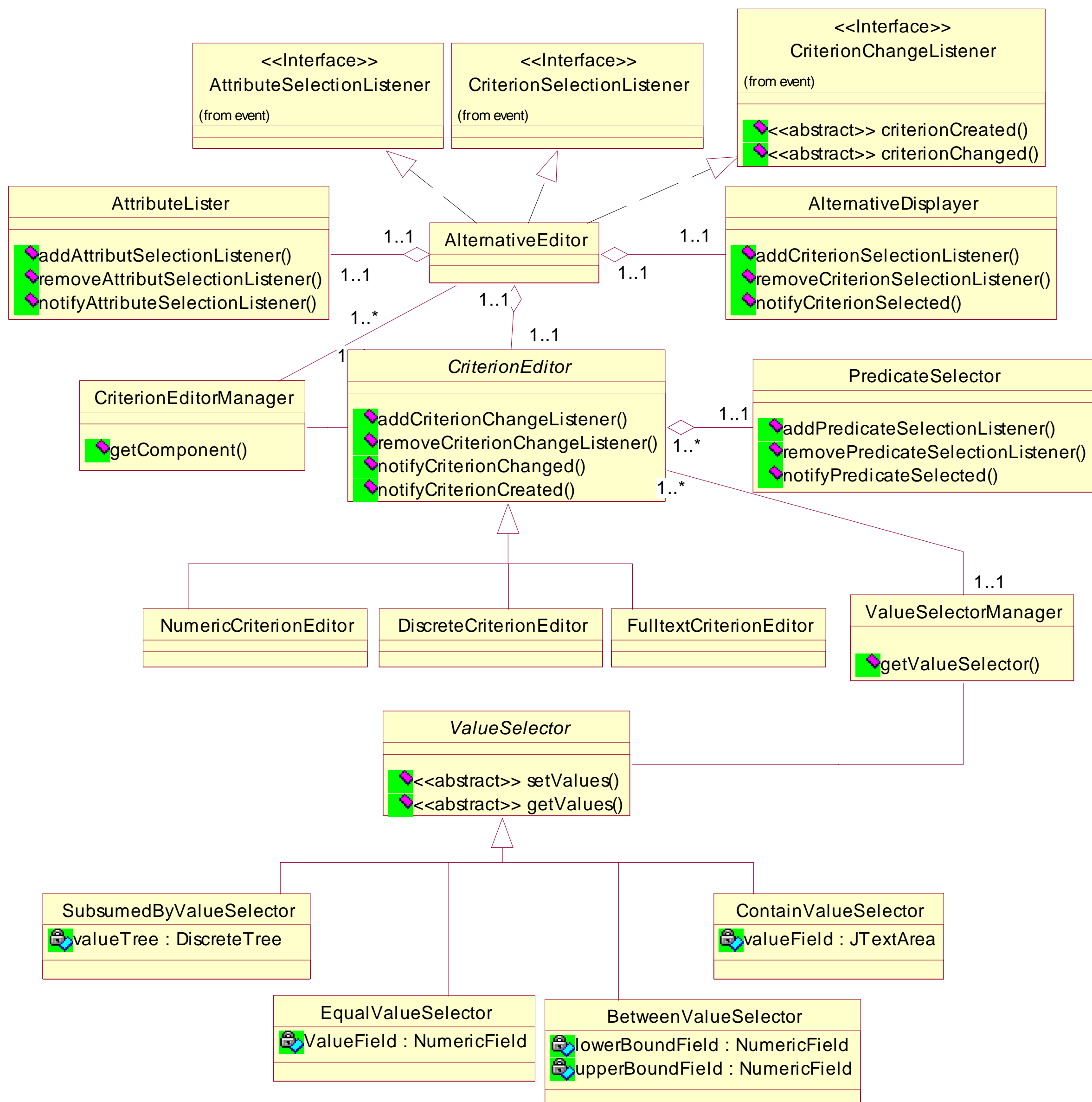


Abbildung 5-18. Der Einsatz vom Entwurfsmuster Vermittler und Singleton im Paket pia.gui

Das Paket "pia.queryresult"

Das Suchergebnis für eine Suchanfrage wird durch die in dem Paket **queryresult** implementierten Klassen repräsentiert. Ein *OfferTableView* stellt das Suchergebnis in tabellarischer Form dar. In jeder Tabellenzeile wird ein die Anfrage erfüllendes Produkt dargestellt und wird mit einer Ranking-Zahl versehen, die zwischen 0 und 1 liegt. Diese Zahl gibt Auskunft darüber, inwiefern das Produkt der Anfrage entspricht. Standardmäßig werden die Produkte nach dieser Ranking-Zahl absteigend sortiert. Da die Tabelle, die im PIA Frontend System verwendet wird, im Grunde genommen nach jeder beliebigen Spalte

sortierbar ist, hat ein Kunde auch die Möglichkeit, die Produkte nach Ausprägung eines beliebigen Attributs sortieren zu lassen.
Zur Detailansicht eines Produktes dienen die Klassen *ItemHTMLView* und *ItemPropertyView*. Ein Kunde kann durch Doppelklick (die Klasse *TableMouseListener* in der Abbilung 5.19) zur Detailansicht eines Produkts gelangen, und diese wird in einer Registerkarte angezeigt, deren Bestandteile der *ItemHTMLView* und der *ItemPropertyView* sind. Mit einem *ItemHTMLView* können Produktdaten im Format einer HTML-Datei dargestellt werden, falls ein entsprechendes Dokument vorliegt. Hingegen repräsentiert ein *ItemPropertyView* das Produkt in Tabellenformat, wobei jede Tabellenzeile eine Produkteigenschaft anzeigt. In der ersten Tabellenspalte stehen die Attributnamen und in der zweiten Spalte die Ausprägungen für das jeweilige Attribut. Mit Hilfe des *DisplayModeEditors* kann ein Kunde den gewünschten Anzeigemodus einstellen. Dazu gehören z.B. die Anzahl der Treffer, die angezeigt werden sollen, sowie die Eigenschaften der Treffer, die jeweils in Tabellenspalten repräsentiert werden sollen. Die beschriebenen Klassen und ihre Beziehungen zueinander werden im folgenden Diagramm (vgl. Abbildung 5.19) dargestellt.

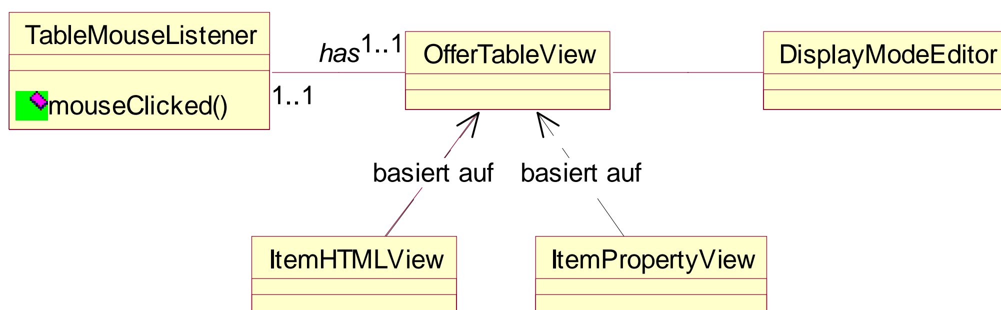


Abbildung 5-19. Klassen im Paket pia.queryresult und ihre Beziehungen zueinander

Das Paket "pia.item"

Das Paket **item** enthält Klassen, die zusammen Informationsanbietern einen Editor zum Editieren ihrer Produktdaten bereitstellen.
Beim Einloggen eines Anbieters ins System wird ihm sein Katalog durch einen *CatalogViewer* repräsentiert. Der Aufbau eines *CatalogViewers* ist ähnlich wie der eines *OfferTableViews*. Die Produkte im Katalog des Anbieters werden jeweils in Tabellenzeilen dargestellt, mit dem Unterschied, daß der Anbieter hier das selektierte Produkt auch editieren kann. Ein *CatalogViewer* bietet ein Popupmenü mit den Menüeinträgen „Produkt hinzufügen", „Produkt löschen" und „Produktdaten editieren". Der *ItemPropertyEditor* dient zum Editieren der Eigenschaften eines Produktes, sowohl beim Anlegen eines neuen

Produkts, als auch für Änderungen eines bereits in der Datenbank gespeicherten Produkts. In einer zukünftigen Version soll noch die Importschnittstelle realisiert werden, die bereits in einem früheren Abschnitt (s. Abschnitt 4.3) beschrieben wurde. Im *ItemPropertyEditor* kann zunächst ein verfügbares Attribut (= alle möglichen Attribute des Systems ohne diejenigen, die bereits zur Beschreibung des Produktes verwendet wurden) gewählt werden. Mit der Hilfe eines *ValueEditors* kann anschließend eine Ausprägung für das Attribut festgelegt werden, wobei Polymorphie wiederum eine wichtige Rolle spielt. So werden vier Unterklassen von *ValueEditor* definiert, *DiscreteValueEditor*, *NumericValueEditor, MeasuredNumericValueEditor* und *FulltextValueEditor*. Um eine enge Kopplung zwischen den einzelnen Klassen zu vermeiden und dadurch die Wiederverwendbarkeit der Klassen zu erhöhen, wird auch hier das Entwurfsmuster Beobachter eingesetzt. Die Klasse *PropertyEditorDialog* verwaltet eine Liste von Beobachtern, die sich für die Ereignisse wie Änderungen einer bestimmten Eigenschaft interessieren. Dafür bietet diese Klasse die Schnittstellenmethoden wie *addPropertyChangeListener(PropertyChangeListener listener)* und *removePropertyChangeListener(PropertyChangeListener listener)* zum Anmelden bzw. Abmelden eines Beobachters. Wenn eine neue Eigenschaft erzeugt wird oder eine alte Eigenschaft geändert wird, werden durch Aufruf der Methoden *notifyPropertyCreated()* bzw. *notifyPropertyChanged()* die entsprechenden Nachrichten an die Beobachter geschickt, die darauf geeignet reagieren. Der *ItemPropertyEditor* ist z.B. ein Beobachter der Klasse *PropertyEditorDialog*. Um sich beim *PropertyEditor* als Beobachter registrieren zu können, der sich für das Ereignis *PropertyChanged* interessiert, muß der *ItemPropertyEditor* das Interface *PropertyChangedListener* implementieren. Wenn der *ItemPropertyEditor* die Nachricht *propertyCreated* oder *propertyChanged* von dem *PropertyEditorDialog* erhält, sorgt er dafür, daß die Darstellung in seiner Tabelle angepaßt wird, so daß diese neue bzw. geänderte Eigenschaft für den Benutzer sichtbar wird. Die genauen Interaktionen zwischen den Objekten beim Anwendungsfall "Produktdaten Editieren" werden im nächsten Kapitel erläutert.

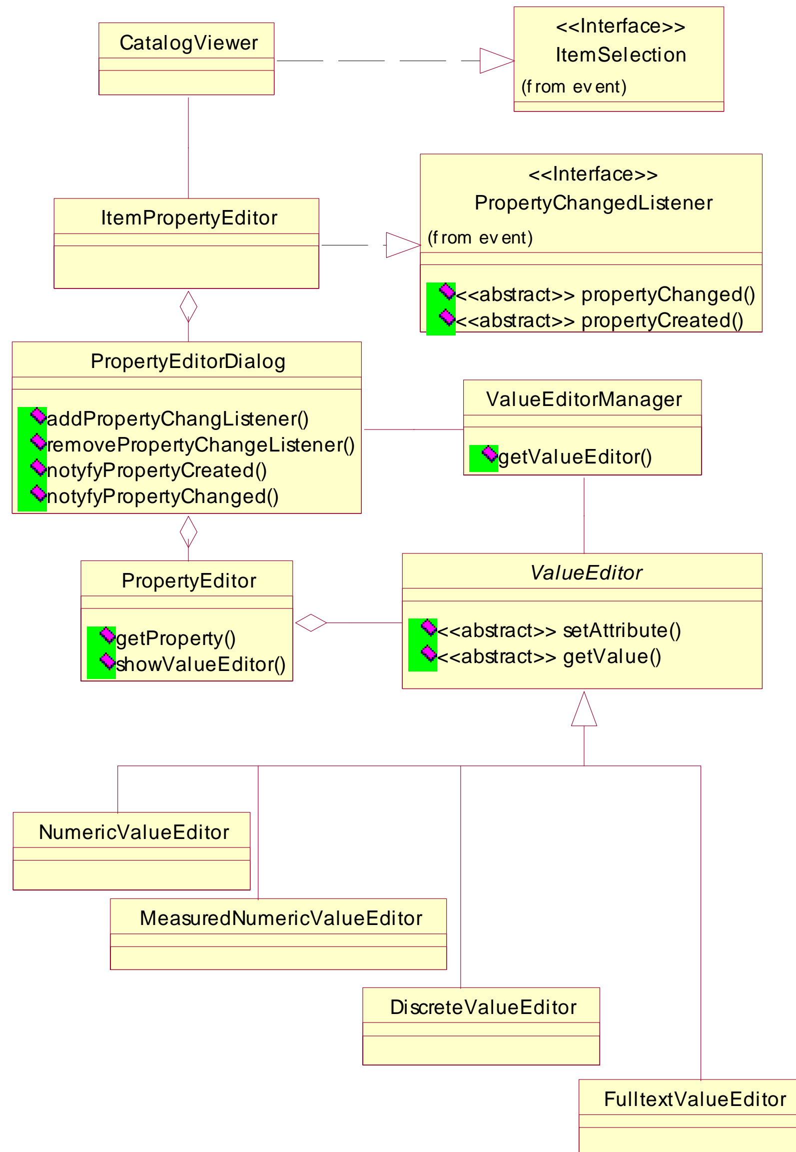


Abbildung 5-20. Das Paket item

Das Paket "gui"

Alle anderen Pakete in PIA-Frontend-System basieren auf dem Paket **gui**, in dem die grundlegenden graphischen Komponenten von PIA Frontend System definiert werden. Zwar kann man mit Hilfe des Java-Swing Toolkits anspruchsvolle Benutzerschnittstellen implementieren, jedoch stellt sich heraus, daß bei einigen Komponenten gewünschte Eigenschaften nicht vorhanden sind, so daß diese Swing Komponenten noch erweitert werden müssen, um die in der Analysephase festgelegten Anforderungen zu erfüllen. Zum einen sollen die Tabellen, die im PIA-Frontend verwendet werden, sortierbar sein, und zwar nach beliebigen Spalten. Zum anderen sollen Textfelder vorhanden sein, die als Eingaben nur Zahlen erlauben. Diese Textfelder werden z.B. im *NumericCriterionEditor*, dessen

ValueEditor nur numerische Angaben annimmt, eingesetzt. Solche PIA-spezifischen graphischen Komponenten wie *SortableTable* und *NumericField* sind im Package **gui** vorhanden.

5.3 Bewertung der Entwurfsphase

Die Entwurfsphase ist geprägt durch den Entwurf von Klassendiagrammen. Dabei bewährt sich der Einsatz von Entwurfsmustern. Sie verhelfen den Entwicklern zu einer gemeinsamen „Sprache“ und tragen zu eleganten Lösungen von Entwurfsproblemen bei. Zusätzlich verspricht der Einsatz von Entwurfsmustern eine bessere Wartbarkeit des Systems und die einfachere Wiederverwendung von Klassen. Die Wiederverwendbarkeit von Klassen durch den Einsatz von Patterns läßt sich schon beim Entwurf der Klassendiagramme zeigen und später in der Implementationsphase (siehe Kapitel 6.) verifizieren.

6 Implementationsphase

In diesem Kapitel wird die Implementationsphase für das PIA-Frontend-System beschrieben. In ihr werden die in der Analysephase gestellten Anforderungen durch die Umsetzung des in der Entwurfsphase gelieferten Entwurfs erfüllt. Schon in der Entwurfsphase fällt die Entscheidung für eine Programmiersprache auf Java (vgl. Kapitel 5). In diesem Kapitel wird gezeigt, daß sich die Entwurfsmodelle mittels der gewählten Sprache in einen Prototyp umsetzen lassen. Des weiteren wird näher auf die Realisierung einiger der wichtigsten Anwendungsfälle, die bereits in der Analysephase beschrieben wurden, eingegangen. Es handelt sich dabei um folgende Anwendungsfälle: Anmelden eines Benutzers, Zusammenstellen einer Suchanfrage, Editieren einer Domäne/eines Attributs, Pflegen eines Produktes.

6.1 Interaktionsdiagramm

Um das dynamische Verhalten des Systems zur Laufzeit zu beschreiben, werden in diesem Kapitel neben Quelltextauszügen aus verschiedenen Klassen auch Interaktionsdiagramme aus UML verwendet. Interaktionsdiagramme können eingesetzt werden, um zu beschreiben, wie Gruppen von Objekten zusammenarbeiten [Fowler97a]. Ein Interaktionsdiagramm erfaßt üblicherweise das Verhalten eines einzelnen Anwendungsfalls. Das Diagramm zeigt eine Anzahl von exemplarischen Objekten und die Nachrichten, die zwischen diesen Objekten innerhalb des Anwendungsfalls ausgetauscht werden.
Zu Interaktionsdiagrammen gehören Sequenzdiagramme und Kollaborationsdiagramme. Die beiden Diagrammarten zeigen im Grunde die gleichen Sachverhalte, jedoch aus verschiedenen Perspektiven. Beim Kollaborationsdiagramm geht es in erster Linie um die Zusammenarbeit der Objekte, während in einem Sequenzdiagramm der zeitliche Verlauf der Nachrichten im Vordergrund steht. In dieser Arbeit werden hauptsächlich Sequenzdiagramme zur Beschreibung der Interaktionen zwischen den Objekten eingesetzt. In einem Sequenzdiagramm werden die Objekte durch gestrichelte, senkrechte Linien dargestellt, und die ausgetauschten Nachrichten durch horizontale Linien zwischen den Objekten. Dadurch wird der zeitliche Ablauf der Nachrichten hervorgehoben. Die Zeit ist von oben nach unten repräsentiert. Auf die Details von Interaktionsdiagrammen bzw. Sequenzdiagrammen muß hier leider verzichtet werden, sie können aber in [Fowler97a, Oesterreich97] nachgelesen werden.

6.2 Realisierung der Anwendungsfälle

In diesem Abschnitt werden folgende Anwendungsfälle aus der Perspektive der Implementation betrachtet: Benutzeranmeldung, Stellen von Suchanfragen, Editieren einer Domäne/eines Attributs, Pflegen eines Produktes. Es wird anhand von Interaktionsdiagrammen und Quelltextauszügen erläutert, wie diese Anwendungsfälle in der Programmiersprache Java realisiert werden.

6.2.1 Anmelden eines Benutzers

Beim Starten eines PIA Clients wird das Hauptfenster *MainFrame* angelegt, das zuerst nur einen modalen Dialog zum Login enthält. Nachdem ein Benutzer seinen Benutzernamen und das Paßwort in das Formular eingegeben hat, werden diese Daten von dem *LoginForm* an das Hauptfenster weitergeleitet, das den Benutzer beim *UserManager* auf der Serverseite anmeldet. Im Falle einer erfolgreichen Anmeldung wird die Rolle des Benutzers, die in der jetzigen Version als String dargestellt wird, vom *UserManager* zurückgeliefert. Danach wird die Benutzeroberfläche der Rolle entsprechend aufgebaut. Wie bereits im letzten Kapitel beschrieben wurde, besteht die Benutzeroberfläche aus einer Navigationsleiste (*IconToggleButtonList*) auf der linken Seite und einem Inhaltspanel (*ContentPane*) auf der rechten Seite, der über die Buttons in der Navigationsleiste gesteuert werden kann. Da Benutzer unterschiedlicher Rollen auch unterschiedliche Aktivitäten mit dem System durchführen, wird jeder Rolle ein eigener Satz von Navigationsbuttons zugeordnet. Um aber nicht für jede Rolle ein eigenes GUI-Element zur Realisierung der Navigationsleiste programmieren zu müssen, wird an dieser Stelle eine wiederverwendbare Komponente implementiert, die einfach als ein Container von Navigationsbuttons betrachtet werden kann. Für diesen Container kann zur Laufzeit von außen festgelegt werden, welche Buttons er enthalten soll. Das Wissen über die Abbildung einer Rolle auf die Buttons wird in eine Managerinstanz verlagert, den *ProfileManager*, der eine Singleton-Instanz ist und mit einer Hashtabelle diese Abbildung der Rollen auf *UserProfiles* verwaltet. Durch den Aufruf der Methode *getButtons(String role)* des *ProfileManagers* wird ermittelt, aus welchen Icons und welchen Komponenten die Navigationsleiste für die Benutzerrolle besteht. Dieses Konzept kann nicht nur für Gruppierung, sondern auch für Individualisierung der Benutzeroberfläche verwendet werden, so daß die Navigationsleiste für jeden Benutzer individuell definierbar und konfigurierbar sein kann. Die Implementation der Methode *getButtons(String role)* ermittelt zuerst das zugehörige *UserProfile* und delegiert die Anfrage an das *UserProfile*.
Im folgenden Quelltextausschnitt (vgl. Abbildung 6.1) aus der Klasse *ProfileManager* wird gezeigt, wie ein Singleton-Muster, das gewährleistet, daß nur ein Exemplar von dieser Klasse existiert, in Java implementiert wird.

```
package pia.mainframe;
...
public class ProfileManager
{
  protected Hashtable hash;
  /**
  * Singletoninstanz.
  */
  protected static ProfileManager manager = null;
  private ProfileManager(){ ...  }
  /**
 * Singletonmethode.
 * @return die Singletoninstanz
 */
 public static ProfileManager instance(){
   if(manager==null)
     manager = new ProfileManager();
   return manager;
 }
 public void reset() { ... }
 public Vector getButtons(String key)
 {
   Vector data = (Vector)hash.get(key);
   if(data==null){
     try{
       String className = (String)defaults.get(key);
       Class profileClass = Class.forName(className);
       Profile profile = (Profile)profileClass.newInstance();
       data = profile.getButtons();
       hash.put(key, data);
     }catch(Exception e)
     {
       e.printStackTrace();
     }
   }
   return data;
 }
}
```

Abbildung 6-1. Singleton - der *ProfileManager*

In der Abbildung 6.1 kann man erkennen, daß der Konstruktor der Klasse als privat deklariert wird. Dadurch kann sichergestellt werden, daß eine Instanz dieser Klasse nur innerhalb dieser Klasse erzeugt werden kann. Um aber von außen eine Referenz auf ein Objekt dieser Klasse zu bekommen, muß die statische Klassenmethode *instance()* verwendet werden, die dafür sorgt, daß nicht mehr als eine Instanz erzeugt wird. Damit hat die Klasse eine strenge Kontrolle über die Anzahl ihrer Instanzen. Die Variable ***manager***, die die Singleton-Instanz enthalten soll, wird mit **null** initialisiert, da hier die Instanz nur bei Bedarf angelegt wird. D.h., die Singleton-Instanz wird erst beim ersten Aufruf der statischen Methode *instance(),* wo die Singleton-Instanz gebraucht wird, angelegt (*Lazy initialization*).
Die Klasse *CustomerProfile* kennt die Buttons, die für Kunden in der Navigationsleiste sichtbar sind. Diese Klasse sorgt außerdem dafür, daß diese Buttons mit den entsprechenden Komponenten in Verbindung gebracht werden, indem sie die Assoziation zwischen einem Button und der jeweiligen Komponente beim *PiaWindowManager* registriert (vgl. Abb. 6.2).

```
package pia.mainframe;
public class CustomerProfile implements Profile{
  public CustomerProfile(){ }
  public Vector getButtons()
  {
    PiaWindowManager windowManager = PiaWindowManager.instance();
            ...
    windowManager.setDefaultComponent("Anfrage", "pia.query.AlternativeEditor");
    windowManager.setDefaultComponent("Ergebnis", "pia.piacatalog.OfferDisplayer");
            ...
    windowManager.setDefaultComponent("Einstellungen",
    "pia.queryresult.DisplayModesEditor");
            ...
    Vector icons = new Vector();
    IconMap iconMap = IconMap.instantiate();
    JToggleButton button = new JToggleButton("Anfrage",
   iconMap.getIcon("Anfrage"));
            ...
    button.setActionCommand("Anfrage");
            ...
    icons.addElement(button);
            ...
    return icons;
  }
}
```

Abbildung 6-2. Quelltextauszug der Klasse *CustomerProfile*

Die Assoziation zwischen einem Navigationsbutton und dessen GUI-Komponente für das Inhaltspanel wird dadurch hergestellt, daß die GUI-Komponente unter dem *ActionCommand* des Navigationsbuttons beim *PIAWindowManager* registriert wird (vgl. Abb. 6.2). Nach dem Aufbau der Oberfläche kann ein Benutzer anfangen, die eigentlichen Aktivitäten durchzuführen. Der zeitliche Ablauf für den Anwendungsfall "Benutzeranmeldung" wird noch einmal im folgenden Sequenzdiagramm (Abb. 6.3) illustriert.

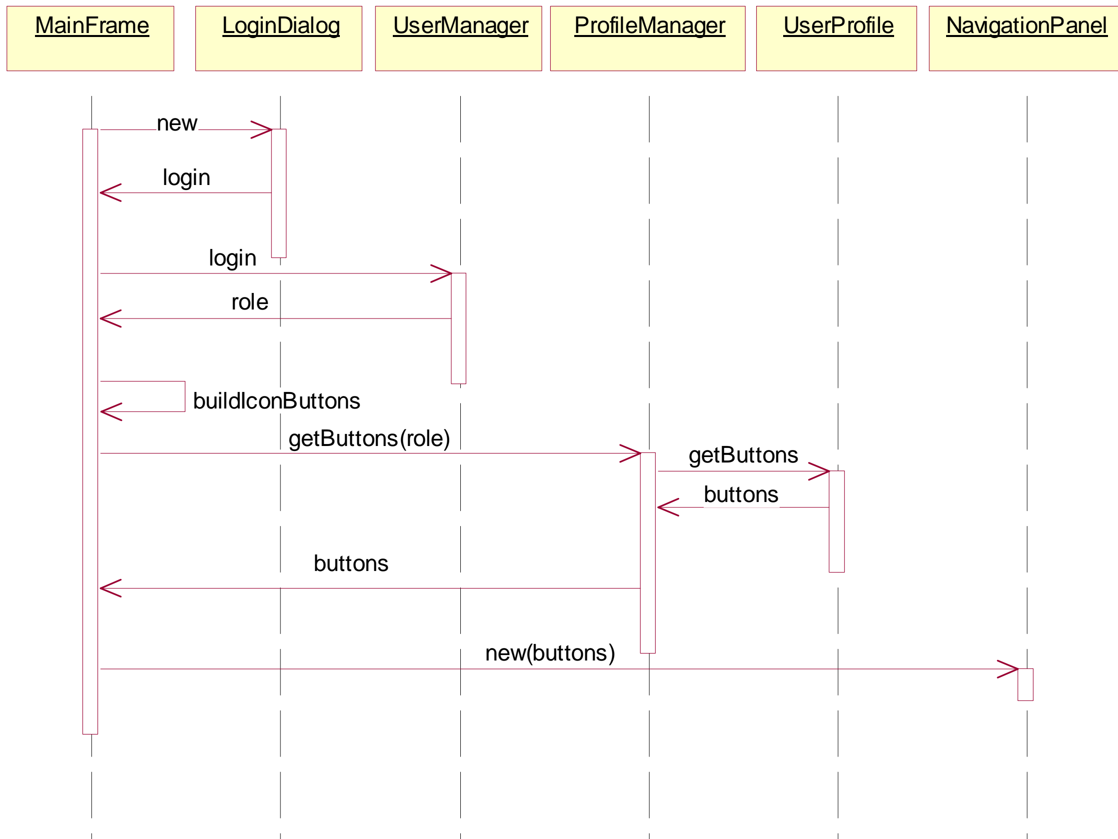


Abbildung 6-3. Interaktionsdiagramm für den Anwendungsfall: „Benutzeranmeldung“

6.2.2 Zusammenstellen einer Suchanfrage

Wie bereits im letzten Kapitel erwähnt wurde, bildet ein *AlternativeEditor* einen wichtigen Bestandteil der Benutzerschnittstelle zum Zusammenstellen einer Suchanfrage. Dabei werden einem Benutzer zuerst alle vorhandenen Attribute in einem *AttributeLister* aufgelistet, aus denen er dann ein beliebiges Attribut auswählen kann. Der *AttributeLister* wird vom *AlternativeEditor* erzeugt. Nach der Erzeugung registriert sich der *AlternativeEditor* als Beobachter beim *AttributeLister*, um auf das Selektionsereignis zu reagieren. Die vom Benutzer zusammengestellte Suchanfrage wird im *AlternativeDisplayer* im rechten Teil des Inhaltspanels dargestellt, der ebenfalls vom *AlternativeEditor* erzeugt und beobachtet wird. Um sich als Beobachter bei seinen beiden Bestandteilen registrieren zu

lassen, implementiert die Klasse *AlternativeEditor* die Interfaces *AttributeSelectionListener* und *CriterionChangeListener* (vgl. Abb. 6.4).

```
package pia.query;
        ...
public class AlternativeEditor extends JPanel implements AttributeSelectionListener,
CriterionChangeListener{
  protected AttributeLister attributeLister;
  protected AlternativeDisplayer AlternativeDisplayer;
  protected CriterionEditor criterionEditor;
  protected Alternative alternative;
  protected QueryManager queryManager = QueryManager.newInstance();

  public AlternativeEditor(){
        ...
    alternative = new Alternative();
    queryManager.setAlternative(alternative);
    updateContent();
  }
  public void updateContent(){
    Iterator iterator =Attributes.getIterator();
    attributeLister = new AttributeLister(iterator);
        ...
    attributeLister.addAttributeSelectionListener(this);
    AlternativeDisplayer = new AlternativeDisplayer(alternative);
        ...
  }
}
```

Abbildung 6-4. Quelltextauszug aus der Klasse alternativeEditor – Teil 1

Sobald ein Attribut im *AttributeLister* gewählt wird, wird der *AlternativeEditor* benachrichtigt. Der *AlternativeEditor* ermittelt zuerst die zugehörige Domäne des ausgewählten Attributs und fragt danach beim *CriterionEditorManager* ab, welcher *CriterionEditor* für die Domäne vorgesehen ist. Es handelt sich bei dem *CriterionEditorManager* wiederum um eine Singleton-Instanz, die die Abbildung von Domänen auf *CriterionEditor*-Objekte verwaltet und einen einfachen globalen Zugriff erlaubt. Eine Instanz von der passenden *CriterionEditor*-Klasse wird von dem *CriterionEditorManager* angelegt und an den *AlternativeEditor* zurückgegeben (vgl. Abb. 6.5). Zwar bestehen alle *CriterionEditors* im PIA System aus einem *JLabel* zur Darstellung des gewählten Attributs, einem *PredicateSelector* und einem *ValueSelector,* jedoch

unterscheiden sich diese Editoren in ihren Layouts, so daß die Spezialisierung des *CriterionEditors* notwendig ist. Die Komponenten in einem *NumericCriterionEditor* sind horizontal ausgerichtet, während die Komponenten in einem *FulltextCriterionEditor* sowie in einem *DiscreteCriterionEditor* vertikal angeordnet sind. Für die Implementierung werden verschiedene Layout Klassen in Swing wie *FlowLayout*, *BoxLayout* und *BorderLayout* usw. eingesetzt [Eckstein98].

```
public void attributeSelected(AttributeSelectionEvent event){
  Attribute attribute = event.getSelectedAttribute();
  AbstractDomain domain = attribute.getDomain();
  CriterionEditor =
((CriterionEditor)manager.getComponent(domain.getClass().getName()));
  criterionEditor.setAttribute(attribute);
  criterionEditor.addCriterionChangeListener(this);
        ...
}
```

Abbildung 6-5. Quelltextauszug aus der Klasse AlternativeEditor - Teil 2

Welcher *ValueSelector* letztendlich in den *CriterionEditor* eingebracht wird, ist wiederum vom jeweiligen selektierten Prädikat abhängig, so daß an dieser Stelle eine weitere Singleton-Instanz – der *ValueSelectorManager* eingesetzt werden muß. Der *CriterionEditor* registriert sich als ein Beobachter beim *PredicateSelector*, wo die gesamte Menge der Prädikate für die Domäne des gewählten Attributs in einem Pulldownmenü aufgelistet wird. Der *CriterionEditor* wird benachrichtigt, sobald ein Prädikat gewählt wird. Beim Empfang der Nachricht *PredicateSelected* fragt der *CriterionEditor* beim *ValueSelectorManager* nach der für das selektierte Prädikat vorgesehenen Komponente, dem passenden *ValueSelector,* und fügt den *ValueSelector* zu sich selbst hinzu.
Nachdem ein Wert in dem *ValueSelector* festgelegt ist (durch Selektion oder Eingabe), wird ein vollständiges *Criterion*-Objekt erzeugt. Der *CriterionEditor* verpackt dieses Objekt in einer Nachricht *CriterionCreated* und schickt sie an den *AlternativeEditor*. Daraufhin schließt der *AlternativeEditor* den *CriterionEditor,* aktualisiert das Model-Objekt, die *Alternative,* und gibt das *Criterion*-Objekt an den *AlternativeDisplayer* weiter, der dieses neue *Criterion* entsprechend darstellt, also als eine neue Zeile seiner Tabelle hinzufügt. Im folgenden Interaktionsdiagramm werden diese Interaktionen zwischen den Objekten noch einmal verdeutlicht (vgl. Abb. 6.6[3]).

[3] Aus Platzgründen müssen leider in diesem Diagramm einige Objekte wie AttributeLister, PredicatesEditor und ValueEditor außer Betracht gelassen werden.

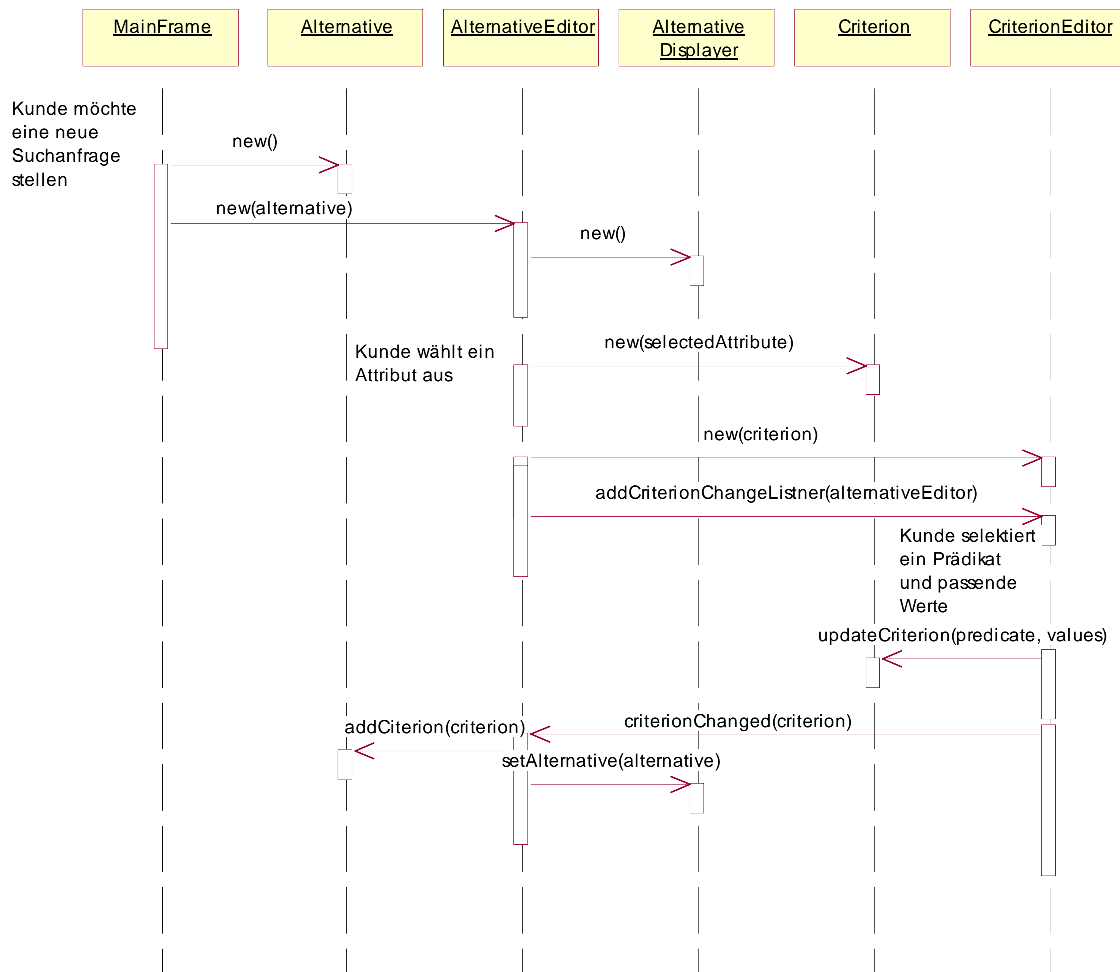


Abbildung 6-6. Interaktionsdiagramm für den Anwendungsfall: "Zusammenstellung einer Suchanfrage"

Ein genanntes Kriterium für die gesuchten Produkte kann auch geändert werden. Hierfür selektiert der Benutzer die entsprechende Zeile im *AlternativeDisplayer*. Da sich der *AlternativeEditor* beim Erzeugen des *AlternativeDisplayer* gleichzeitig als ein Beobachter registriert hat, erhält er bei der Selektion eines *Criterions* die Nachricht *CriterionSelected* vom *AlternativeDisplayer*. Der *AlternativeEditor* erzeugt beim Erhalten der Nachricht einen passenden *CriterionEditor*. Der Ablauf hier ist ähnlich wie beim Spezifizieren einer neuen Eigenschaft für die Produktsuche, bis auf den Unterschied, daß nach der Generierung des *CriterionEditor* der Editor noch mit *updateContent(Criterion crierion)* mit dem selektierten Kriterium initialisiert wird (vgl. Abb. 6.7).

```
public void criterionCreated(Criterion criterion){
  alternative.addCriterion(criterion);
  AlternativeDisplayer.addCriterion(criterion);
```

```
        ...
 }

public void criterionChanged(Criterion criterion){
  alternative.updateCriterion(criterion);
  AlternativeDisplayer.update(criterion);
        ...
}

public void criterionSelected(Criterion criterion){
  AbstractDomain domain = criterion.getAttribute().getDomain();
  CriterionEditor =
((CriterionEditor)manager.getComponent(domain.getClass().getName()));
        ...
  criterionEditor.updateContent(criterion);
}
        ...
}
```

Abbildung 6-7. Quelltextauszug aus der Klasse AlternativeEditor – Teil 3

Das Interface - *CriterionChangeListener* ist vorgesehen für alle Objekte, die sich für das Erzeugen bzw. Modifizieren eines Criterion-Objekts interessieren (vgl. Abb. 6.8).

```
package pia.query;

public interface CriterionChangeListener {
  public abstract void criterionChanged(Criterion criterion);
  public abstract void criterionCreated(Criterion criterion);
}
```

Abbildung 6-8. Quelltextauszug aus der Interface CriterionChangeListener

Dieses Interface wird u.a. von *AlternativeEditor* implementiert. Der *AlternativeEditor* speichert eine Referenz auf das *Alternative*-Objekt, das gerade editiert wird. Immer wenn eine neue Eigenschaft genannt wird oder eine bereits vorhandene Eigenschaft geändert wird, wird der *AlternativeEditor* benachrichtigt, so daß er in den Methoden *criterionChanged(Criterion criterion)* und *criterionCreated(Criterion criterion)* den Zustand des entsprechenden *Alternative*-Objektes aktualisieren kann. Als Vermittler sorgt der Editor auch dafür, daß diese Nachrichten an den *AlternativeDisplayer* weitergeleitet werden (vgl. Abb. 6.7).

Der *AlternativeDisplayer* stellt den aktuellen Zustand des Alternative-Objekts dar und hält eine Referenz auf das dargestellte Objekt. Beim Erzeugen des *AlternativeDisplayers* wird das Alternative-Objekt als Parameter übergeben (vgl. Abb. 6.9).

```
package pia.query;
...
public class AlternativeDisplayer extends JPanel implements CriterionChangeListener
{
 ...
 protected AlternativeTableModel dataModel;
 protected SortableTable tableView;
 protected JPopupMenu popupMenu;
 protected Alternative alternative;

 public AlternativeDisplayer(Alternative alternative){
  this.alternative = alternative;
  dataModel = new AlternativeTableModel(alternative);
  tableView = new SortableTable(dataModel);
  ...
 }
```

Abbildung 6-9. Quelltextauszug aus der Klasse *AlternativeDisplayer* - Teil 1

Wenn die zu editierende *Alternative* geändert wird (z.B. eine neue Eigenschaft mit Hilfe des *CriterionEditors* genannt und hinzugefügt wird o.ä.), sorgt der *AlternativeEditor* dafür, daß die Konsistenz des Alternative-Objekts erhalten bleibt, indem er eine der folgenden Methoden *setAlternative(Alternative alternative), criterionChanged(Criterion criterion) oder criterionCreated(Criterion criterion)* von *AlternativeDisplayer* aufruft, die die Alternative entsprechend aktualisieren (vgl. Abb. 6.10).

```
 public void setAlternative(Alternative alternative){
  this.alternative = alternative;
  tableView.setModel(new AlternativeTableModel(alternative));
  tableView.validate();
  tableView.repaint();
 }
public void criterionChanged(Criterion criterion){
 dataModel.replaceCriterion(tableView.getSelectedRow(),criterion); ...
}
public void criterionCreated(Criterion criterion){
```

```
    dataModel.addCriterion(criterion); ...
  }
```

Abbildung 6-10. Quelltextauszug aus der Klasse *AlternativeDisplayer* – Teil 2

Wie man aus dem folgenden Quelltextauszug ersehen kann, basiert eine sortierbare Tabelle auf einem Model-Objekt einer Klasse, die Unterklasse der abstrakten Klasse *AbstractTableModel* ist. Der Grund hierfür liegt darin, daß jede Swing Komponente auf einer leicht veränderten Version des MVC-Konzeptes basiert, die eine klare Trennung von Datenmodell (*Model*) und Darstellung (*View*) erlaubt. (vgl. Kapitel 3). Wenn z.B. der *AlternativeDisplayer* die Nachricht vom *AlternativeEditor* bekommt, daß Änderungen an dem *Alternative*-Objekt aufgetreten sind, so aktualisiert er das Model-Objekt. Das Model-Objekt meldet daraufhin die Änderung an alle Beobachter. Da das UI-Delegate für das Model-Objekt als Beobachter bei dem Model-Objekt registriert ist, wird auch das UI-Delegate von der Änderung informiert und kann seine optische Darstellung ebenfalls aktualisieren.

Die Modell-Klasse wird in der Klasse *AlternativeDisplayer* als eine innere Klasse *AlternativeTableModel* implementiert (vgl. Abb. 6.11), die die abstrakte Klasse *AbtractTableModel* spezialisiert. Zwar kann man mit Hilfe der von Swing angebotenen Klasse *DefaultTableModel* schnell eine Tabelle programmieren. Wie bereits im Kapitel 3 "Technische Grundlagen" beschrieben wurde, können aber durch die Implementierung einer eigenen TableModel-Klasse speziellere Anforderungen erfüllt werden. Das *AlternativeTableModel* für einen AlternativeDisplayer versteht z.B. Nachrichten wie *removeCriterionAtIndex(int i)*, *getAllCriterions(), addCriterion(Criterion criterion)* usw., die speziell auf die PIA Anwendung zugeschnitten sind.

```
class AlternativeTableModel extends AbstractTableModel
{
  ...
  private Vector criterions = new Vector();

  public AlternativeTableModel()
  {
   super();
  }

  public AlternativeTableModel(Alternative alternative){
   super();
   Iterator iterator = alternative.getCriterions().iterator();
   while(iterator.hasNext()){
```

```
    criterions.addElement(iterator.next());
  }
}

public int getRowCount()
{
  return criterions.size();
}

public String getColumnName(int column)        {
  return names[column];
}

public Class getColumnClass(int c){
  return getValueAt(0, c).getClass();
}

public int getColumnCount() {
  return names.length ;
}

public Object getValueAt(int row, int col){
  Criterion criterion = (Criterion)criterions.elementAt(row);
  switch(col) {
    case 0:
      return new Double(criterion.getWeight());
    case 1:
      return criterion;
  }
  return null;
}

public boolean isCellEditable(int row, int column ){
  return (column==0) ;
}

public void setValueAt(Object obj, int row, int column){
  return;
}
```

```
/**
*   Adds a new Criterion to the model and fires the change event.
*/
public void addCriterion(Criterion criterion)
{
  criterions.addElement(criterion);
  fireTableChanged(new TableModelEvent(this));
}

/**
*   Removes a criterion from the model and fires the change event.
*/
public void removeCriterion(Criterion criterion)
{
  if(criterions.removeElement(criterion))
  fireTableChanged(new TableModelEvent(this));
}

/**
*   Removes a criterion from the model and fires the change event.
*/
public void removeCriterionAtIndex(int index)
{
  criterions.removeElementAt(index);
  fireTableChanged(new TableModelEvent(this));
}

/**
*   Replaces a criterion at index i.
*/
public void replaceCriterion(int index, Criterion criterion)
{
  criterions.removeElementAt(index);
  criterions.insertElementAt(criterion, index);
  fireTableChanged(new TableModelEvent(this));
}

/**
*   Returns the criterion at the index.
*/
```

```
    public Criterion getCriterionAt(int index)
    {
      return (Criterion)criterions.elementAt(index);
    }

    /**
    *   Returns the criterions.
    */
    public Vector getCriterions()
    {
      return criterions;
    }
  ...
  }
}
```

Abbildung 6-11. Quelltextauszug aus der Klasse *AlternativeDisplayer* – Teil 3

6.2.3 Editieren eines Domänenobjektes

Die Redakteure im PIA System sind für die Verwaltung der Metainformationen wie Domänen und Attribute zuständig. In der Navigationsleiste des Hauptfensters für einen Redakteur befinden sich die graphischen Icons, die entweder mit einem *DomainPanel* zur Verwaltung von Domänen einer bestimmten Domänenkategorie oder mit einem *AttributePanel* zur Verwaltung von Attributen assoziiert sind. Konkret handelt es sich um ein *DiscreteDomainPanel*, ein *FulltextDomainPanel*, ein *NumericDomainPanel* und ein *AttributePanel*.
Ein *DomainPanel* stellt alle ihm übergebenen Domänenobjekte in einer baumartigen Struktur dar. Über ein Popupmenü kann ein Redakteur die Domänenobjekte im Baum editieren, wobei das Editieren das Hinzufügen einer neuen Domäne, das Löschen einer vorhandenen Domäne oder das Modifizieren einer vorhandenen Domäne bedeuten kann. Daß die Klasse *DomainPanel* weiter spezialisiert wird, ergibt sich aus der Vielfalt der Domänenkategorien. Im folgenden Quelltextauszug aus der Klasse *DomainPanel* kann ein Überblick darüber gewonnen werden, wie diesen Anforderungen genügt wird (vgl. Abb. 6.12).

```
package pia.maintenance.domain;
...
```

```
public abstract class DomainPanel extends JPanel implements DomainChangeListener
{
  protected HashSet domains;
  protected JTree tree;
  protected DefaultTreeModel model;
  protected JPopupMenu popupMenu;
  ...
  public DomainPanel(Iterator iterator)
  {
    domains = new HashSet();
    DefaultMutableTreeNode root = new DefaultMutableTreeNode("");
    while (iterator.hasNext()){
     ...
      DefaultMutableTreeNode node = new DefaultMutableTreeNode(obj);
      root.add(node);
      domains.add(obj);
    }
    model = new DefaultTreeModel(root);
    tree = new JTree(model);

    tree.addMouseListener(new MouseAdapter()
    {
      public void mousePressed(MouseEvent event)
      {
...
           showValueDialog(selectedDomain);
       }
      }
    });
...
}
public void showPopup(int x, int y) { ...  }
protected JPopupMenu constructPopupMenu(){
...
  return popupMenu;
}
```

Abbildung 6-12. Quelltextauszug aus der Klasse *DomainPanel* - Teil 1

Beim Selektieren eines Domänenobjekts im Domänenbaum wird die Methode showValueInDialog(AbstractDomain domain) aufgerufen, die ein Dialogfenster zum Anzeigen und Editieren der Detailinformationen des Domänenobjekts generiert (vgl. Abb. 6.13).

```
public void showValueDialog(AbstractDomain domain)
{
  ...
   DomainEditorPanel editor = getDomainEditorPanel(domain);
   editor.addDomainChangeListener(this);
 ...
}
```

Abbildung 6-13. Quelltextauszug aus der Klasse *DomainPanel* - Teil2

Die verschiedenen Domänenobjekte gehören zu unterschiedlichen Kategorien und basieren somit auf unterschiedlichen Objektmodellen. Daher unterscheiden sich die Editoren für die verschiedene Domänenkategorien im Layout und im Aufbau der Komponenten, um den verschiedenen Anforderungen beim Editieren der zugrundeliegenden Domänenkategorie gerecht zu werden. Um den gleichen Code nicht für jede Kategorie wiederholen zu müssen, wird an dieser Stelle das Template-Muster implementiert. Eine abstrakte Methode *getDomainEditorPanel(AbstractDomain domain)* wird als Platzhalter definiert, deren Implementation den Unterklassen überlassen wird. So kann der Ablauf zum Anzeigen des Dialogfensters einmal in der Oberklasse *DomainPanel* abstrakt implementiert werden und in den 3 Unterklassen wiederverwendet werden. Ein *DomainPanel* ist zugleich ein Beobachter des *DomainEditorPanels*, so daß es über jede Änderung der Domäne benachrichtigt werden kann. Dazu implementiert die Klasse das Interface *DomainChangeListener*, das zwei Methoden *domainCreated(Domain domain)* und *domainChanged(Domain domain)* für das Erzeugen und Ändern von Domänen definiert. Beim Erzeugen einer neuen Domäne fügt das *DomainPanel* einen neuen Knoten in den Baum ein, der die neue Domäne repräsentiert, während es beim Ändern einer vorhandenen Domäne den entsprechenden Knoten aktualisiert (vgl. Abb. 6.14).

```
public abstract String getTitle(AbstractDomain domain);

public void domainCreated(AbstractDomain domain)
{
  DefaultMutableTreeNode node = new DefaultMutableTreeNode(domain);
          ...
  model.insertNodeInto(node, root, root.getChildCount());
```

```
    domains.add(domain);
  }

  public void domainChanged(AbstractDomain domain)
  {
    model.valueForPathChanged(tree.getSelectionPath(), domain);
  }

  protected abstract DomainEditorPanel getDomainEditorPanel(AbstractDomain
  domain);
}
```

Abbildung 6-14. Quelltextauszug aus der Klasse *DomainPanel* – Teil 3

Wie bereits im letzten Kapitel beschrieben wurde, besteht ein *DomainEditorPanel* aus einem Namensfeld und mehreren Registerkarten. Da der Name eine gemeinsame Eigenschaft von allen Domänen ist, wird das Namensfeld schon bei der Konstruktion des *DomainEditorPanels* bereitgestellt. Je nach der Domänenkategorie werden entsprechende Komponenten durch Aufruf der Methode *fillTabs()* in der Registerkarte bereitgestellt. Hier wird noch einmal das angesprochene Template-Muster verwendet, um die Unterschiede zwischen den Kategorien an die konkreten Unterklassen von *DomainEditorPanel* zu delegieren (vgl. Abb. 6.15).

```
package pia.maintenance.domain;
...
public abstract class DomainEditorPanel extends JPanel
{
  protected AbstractDomain domain;
  protected JTextField txtName;
  protected JTabbedPane tabPane;
  protected HashSet listeners;

  public DomainEditorPanel()
  {
    this(null);
  }

  /** Constructor */
  public DomainEditorPanel(AbstractDomain domain)
  {
```

```
        ...
  tabPane = new JTabbedPane();
  fillTabs();
        ...
  if(domain!=null)
    updateContent();
        ...
}
```

Abbildung 6-15. Quelltextauszug aus der Klasse *DomainEditorPanel* – Teil 1

Wenn das Domänenobjekt geändert ist, wird die Methode *saveDomain(domain)* aufgerufen. Innerhalb dieser Methode wird die Methode *updateDefinition()* aufgerufen, die das Model-Objekt, die Domäne, aktualisiert. In *updateDefinition()* wird aber nur die gemeinsame Eigenschaft, nämlich der Name, aktualisiert und die Aktualisierung der domänenkategoriespezifischen Eigenschaften über das Template-Muster an die Methode *updateOtherDefinition()* delegiert. Anschließend sorgt der *DomainEditorPanel* dafür, daß alle Beobachter, wie das *DomainPanel,* benachrichtigt werden, damit sie die Darstellung für das Domänenobjekt aktualisieren können. Je nach dem, ob der *DomainEditorPanel* eine neue Domäne erzeugt hat oder eine vorhandene geändert hat, ruft der *DomainEditorPanel* die Methode *domainCreated(domain)* oder *domainChanged(domain)* auf, so daß die Beobachter entsprechend reagieren können. Zur Unterscheidung der beiden Aktionen wird auf das bei der Konstruktion des *DomainEditorPanels* als Parameter übergebene Domänenobjekt zurückgegriffen. Falls es sich um ein Null-Objekt handelt, bedeutet dieses, daß eine neue Domäne erzeugt wird, und daher wird eine Nachricht *domainCreated(domain)* an den *DomainPanel* geschickt. Dieser sorgt dafür, daß in der Baumdarstellung der Domänenobjekte ein entsprechender Knoten hinzugefügt wird. Ansonsten wird ein schon vorhandenes Domänenobjekt geändert. Daher wird in diesem Fall die Nachricht *domainChanged(domain)* an den *DomainPanel* geschickt, der dann nur den selektierten Knoten aktualisiert. (vgl. Abb. 6.16).

```
public void addDomainChangeListener(DomainChangeListener listener)
{
  listeners.add(listener);
}
public void removeDomainChangeListener(DomainChangeListener listener)
{
  listeners.remove(listener);
}
/** Fills the tab pane with contents */
protected abstract void fillTabs();
```

```
  /** Update the view with definition contents */
  protected void updateContent()
  {
    txtName.setText(domain.getName());
    updateOtherContent();
  }
  /** Updates the contents of other tabs. */
  protected abstract void updateOtherContent();
  /** Save the domain object.  */
  public void saveDomain()
  {
    boolean creating = (domain==null);
    if(!updateDefinition()){
      if(creating)
        domain = null;
      return;
    }
    Iterator itera = listeners.iterator();
    while(itera.hasNext()){
      DomainChangeListener listener = (DomainChangeListener)itera.next();
      if(creating)
        listener.domainCreated(domain);
      else
        listener.domainChanged(domain);
    }
    cancel();
  }
  /** Update the domain definition. */
  protected boolean updateDefinition(){... }
  /** Update other attributes of the domain definition. */
  protected abstract boolean updateOtherDefinition();
}
```

Abbildung 6-16. Quelltextauszug aus der Klasse *DomainEditorPanel* - Teil 2

In der folgenden Abbildung (vgl. Abb. 6.17) wird der Ablauf eines solchen Prozesses für den Aufzählungstyp anhand eines Sequenzdiagramms dargestellt. Die Registerkarte für den Aufzählungstyp umfaßt einen *DomainTree* zum Editieren von zugelassenen Werten und einen *PredicatesEditor* zum Editieren von den Prädikaten, die auf diese Domäne angewendet

werden können. Innerhalb des Methodenaufrufs *fillTabs()* werden ein *DomainTree*, der die Werte eines Aufzählungstyps in einem Baum darstellt, und ein *PredicatesEditor* generiert.

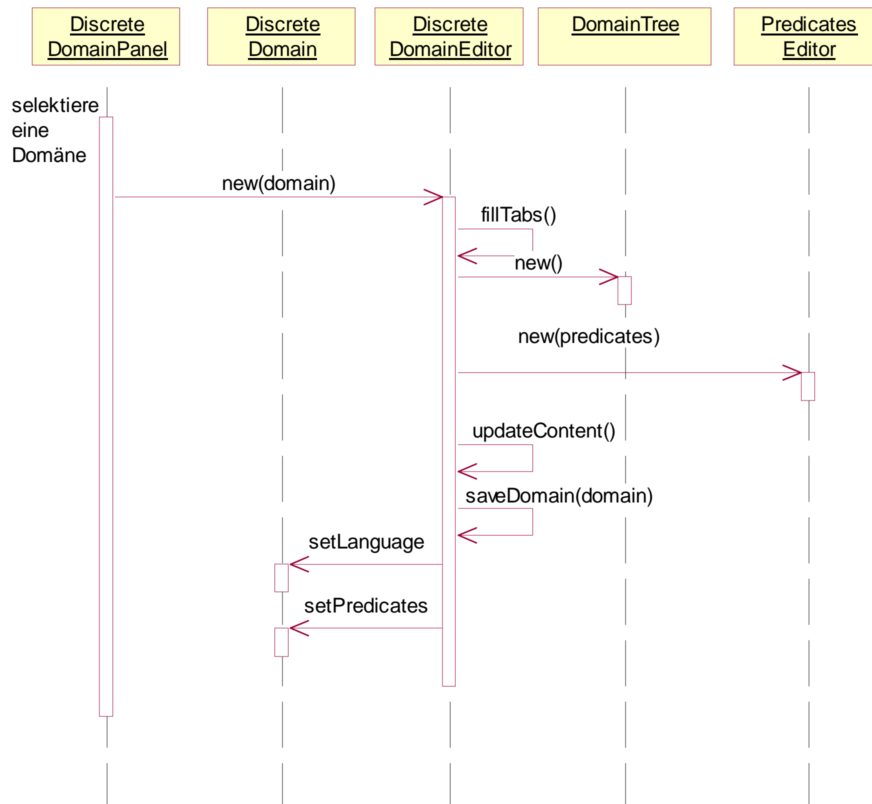


Abbildung 6-17. Interaktionsdiagramm für den Anwendungsfall „Domänen editieren“

6.2.4 Editieren eines Attributs

Ein Redakteur kann außerdem die Attribute im System bearbeiten. Im Paket **pia.maintenance.attribute** befinden sich die Klassen zur Bereitstellung der Benutzerschnittstellen für Redakteure zur Unterstützung bei der Attributverwaltung im PIA System. Der Einstiegspunkt ist ein *AttributePanel*, das mit dem in dem *IconToggleButtonList* stehenden Icon, das mit der Unterschrift "Attribute" versehen wird, assoziiert ist. Das *AttributePanel* enthält die gesamte Menge der Attribute im PIA System. Jedes Attribut wird als ein Baumknoten dargestellt und beim Doppelklick auf einen Knoten oder beim Selektieren des Menüeintrages "neues Attribut Hinzufügen" im Popupmenü wird ein Dialogfenster mit einem *AttributeEditor* generiert, der mit dem selektierten Attribut oder einem leeren Attribut gefüllt wird. Mit Hilfe des *AttributeEditors* kann ein Redakteur den Namen sowie die dazugehörige Domäne eines Attributes editieren. Im folgenden Sequenzdiagramm wird die Interaktion zwischen den beteiligten Objekten beim Editieren eines selektierten Attributs illustriert. Nachdem der *AttributeEditor* mit der Referenz auf das

Attribut-Objekt, das er bei der Initialisierung von dem *AttributePanel* als Übergabeparameter erhält, instanziiert wird, registriert sich das *AttribuePanel* als Beobachter beim *AttributeEditor*, so daß es über die Änderungen an dem zu editierenden Attribut benachrichtigt werden kann. Dafür bietet der *AttributeEditor* eine Schnittstelle zum Anmelden bzw. Abmelden eines Beobachters. Die Klasse, die sich als Beobachter registrieren möchte, muß die Interface *AttributeChangeListener* implementieren (vgl. Abbildung 6.18).

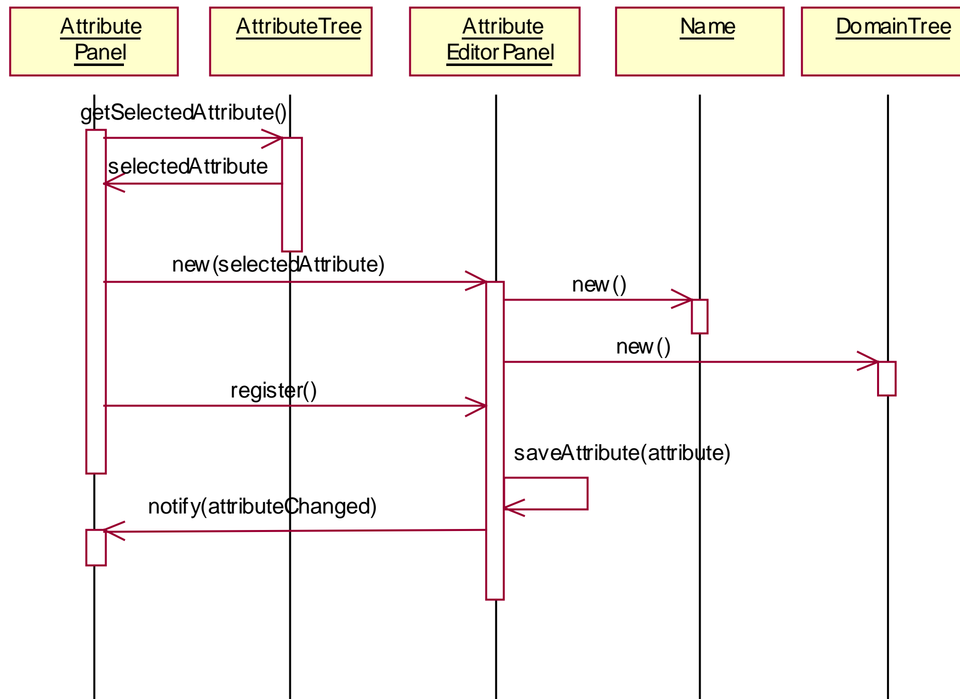


Abbildung 6-18. Interaktionsdiagramm für den Anwendungsfall „Attribut Editieren“

Ein ähnliches Interaktionsdiagramm ergibt sich für den Anwendungsfall "Hinzufügen eines neuen Attributs". Statt die Nachricht "attributeChanged" an die Beobachter zu schicken, benachrichtigt der *AttributeEditor* seine Beobachter über die Erzeugung eines neuen Attributs. Nach dem Erhalt der Nachricht sorgt der *AttributePanel* dafür, daß das Model-Objekt, das hinter der Baumdarstellung aller Attribute steht, ebenfalls aktualisiert wird, so daß die Änderungen der Attribute im Anschluß persistent gespeichert werden können.

6.2.5 Editieren eines Produkts

Die Hauptziel eines Anbieters mit PIA System ist, seine Produktinformation seinen Kunden zu vermitteln. PIA bietet zwar eine Importschnittstelle, mit deren Hilfe ein Anbieter seine Produktdaten in das PIA System importieren lassen kann. Gleichzeitig ist aber auch das

Einfügen oder das Ändern von Produktdaten per Hand mit Hilfe der Benutzerschnittstelle möglich. Die Interface-Klassen zur Unterstützung der Produktdatenverwaltung werden im Paket **pia.item** implementiert. Die wichtigsten Klassen dieses Pakets wurden bereits im vorigen Kapitel erläutert. Ein *CatalogViewer* dient als der Einstiegspunkt für einen Anbieter, der beim Anmelden vom System identifiziert wird. Nach der Prüfung der Eingabedaten vom PIA Server werden die Produktdaten des Anbieters zum Client geschickt, und diese werden im *CatalogViwer* in tabellarischer Form dargestellt.
Wie man aus dem folgenden Sequenzdiagramm (vgl. Abb. 6.20) entnehmen kann, kann der Anbieter eine Zeile aus der Tabelle im *CatalogViewer* selektieren und daraufhin mit Hilfe eines *ItemPropertyEditors* das selektierte Produkt editieren. Der *ItemPropertyEditor* stellt die bereits vorhandenen Eigenschaften des Produkts in einer Tabelle dar und bietet eine Menüauswahl zum "Hinzufügen", "Löschen" und "Übernehmen" von Eigenschaften. Das folgende Diagramm (vgl. Abbildung 6.19) illustriert die Interaktionen zwischen den Objekten für den Fall, daß der Anbieter den Menüpunkt "Hinzufügen" aktiviert. Dabei wird zuerst ein neues *Property*-Objekt erzeugt. Das Property-Objekt wird als Übergabeparameter beim Erzeugen eines *PropertyEditors* weitergegeben. Der *ItemPropertyEditor* registriert sich außerdem als ein Beobachter beim *PropertyEditor,* um sich über die Änderungen der Eigenschaft zu informieren. Beim Empfang einer Nachricht des *PropertyEditors* speichert er die neu editierte bzw. geänderte Eigenschaft im entsprechenden Item-Objekt, indem er ihm die Nachricht *setProperty(Property property)* schickt. Es bleibt zu erwähnen, daß der PropertyEditor aus zwei Hauptteilen besteht: einem Pulldownmenü mit allen noch nicht belegten Attributen und einem *ValueEditor*. Beim Selektieren eines Attributes aus dem Pulldownmenü wird ein entsprechender *ValueEditor* zur Verfügung gestellt, der von der Domäne des selektierten Attributs abhängt. Für das Mapping zwischen einem Domänenamen und dem Klassennamen der entsprechenden Klasse, die Unterklasse von *ValueEditor* ist, ist die Singleton-Instanz *ValueEditorManager* vorgesehen.

```
package pia. item;
...
public class ValueEditorManager
{
 /**
  * Mapping von Key auf einen Klassennamen.
  * Beim ersten Zugriff über den Key wird ein Objekt der Klasse erzeugt.
  */
 protected Hashtable defaults;
 /**
  * Singletoninstanz.
  */
 protected static ValueEditorManager manager = null;
```

```
  private ValueEditorManager()
  {
    defaults = new Hashtable();
    defaults.put("piamodel.numericDomain.NumericDomain",
"pia.maintenance.item.NumericValueEditor");
          ...
    defaults.put("piamodel.fulltextDomain.FullTextDomain",
"pia.maintenance.item.FullTextValueEditor");
  }
          ...
  public static ValueEditorManager instance()
  {
    if(manager==null)
      manager = new ValueEditorManager();
    return manager;
  }
          ...
  public ValueEditor getValueEditor(String key)
  {
     ValueEditor com = null;
     try
     {
       String className = (String)defaults.get(key);
       Class componentClass = Class.forName(className);
       com = (ValueEditor)componentClass.newInstance();
     }catch(Exception e)
    {
      e.printStackTrace();
    }
    return com;
  }
}
```

Abbildung 6-19. Quelltextauszug aus der Klasse *ValueEditorManager*

Im folgenden Interaktionsdiagramm soll noch einmal verdeutlicht werden, wie die Interaktionen zwischen den Objekten für den Anwendungsfall „Produkteditieren" realisiert werden (vgl. Abb. 6.20).

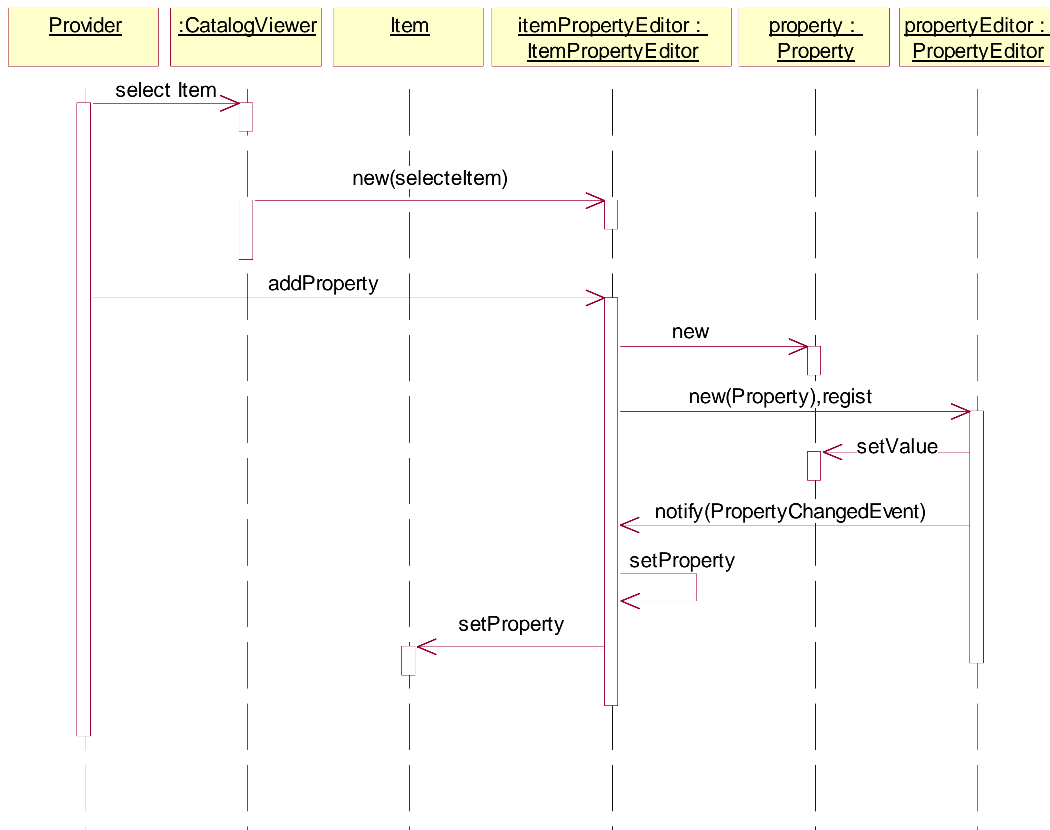


Abbildung 6-20. Interaktionsdiagramm für den Anwendungsfall „Produkt editieren“

6.3 Bewertung der Implementationsphase

In der Implementationsphase dieser Arbeit wird das erste Inkrement einer vollständigen Benutzerschnittstelle entwickelt. Das PIA-Frontend System besteht aus ca. 120 Java Source-Dateien und 160 Java Klassen, die in 7 Pakete unterteilt sind. In der Implementation werden die Entwurfsmuster wie z.B. „Singleton“, „Vermittler“, „Template“ und „Observer“ an vielen Stellen im PIA-Frontend System eingesetzt, was den Wiederverwendungsgrad der Komponenten erhöht und die Fehlerquellen reduziert. Die Anwendungsfälle, die in der Analysephase aufgestellt worden sind, sind im System umgesetzt.
Da Java eine relativ junge Sprache ist, ist die Erstellung von professionellen GUI-Oberflächen mit Hilfe von Java noch keine triviale Aufgabe. Das neue Toolkit Swing hat zwar die Situation erheblich verbessert, leidet aber noch unter den zahlreich vorhandenen Fehlern in der Implementation. Wegen des Mangels an guten GUI-Buildern ist es nicht möglich, Oberflächen per Drag & Drop zusammenzustellen. Diese müssen alle noch manuell kodiert werden.

7 Zusammenfassung und Ausblick

Die vorliegende Arbeit beschreibt den Entwurf und die Realisierung der Benutzerschnittstelle des interaktiven Informationssystems PIA. Dabei wird durchgängig der objektorientierte Ansatz für Softwareentwicklung verfolgt, der aus den drei Phasen Analyse, Entwurf und Implementierung besteht.
Zuerst werden die wichtigsten Anwendungsfälle des Systems beschrieben, die auf [Zhang98] basieren und darüber hinaus einige Änderungen und Erweiterungen aufweisen. Diese Anwendungsfälle werden zusammen in einem gemeinsamen Anwendungsfalldiagramm dargestellt, das der wichtigste Bestandteil des Analysemodells ist und als Grundlage für die weiteren Phasen dient. Auf der Basis dieses Analysemodells wird in der Entwurfsphase ein Objektmodell erstellt, das die gesamte Struktur des Systems u.a. anhand von Klassendiagrammen darstellt, die die Eigenschaften und Schnittstellen der Klassen und ihre Beziehungen zueinander repräsentieren. Im Entwurfsmodell werden verschiedene Entwurfsmuster zur Unterstützung der Entwicklung des PIA Systems eingesetzt, die entsprechend in der Beschreibung der Entwurfsphase betrachtet wurden. Daraufhin wird das erste Inkrement des Java-Frontends für das PIA System in der objektorientierten Programmiersprache Java implementiert, in dem die wichtigsten Anwendungsfälle realisiert werden. Die Implementation basiert auf dem neuen GUI-Toolkit Swing und kann somit von dessen Eigenschaften profitieren. Neben hoher Wiederverwendbarkeit und großer Flexibilität wird bei der Implementierung des PIA Frontend-Systems stets auf die Benutzerfreundlichkeit geachtet, so daß die Oberfläche von allen Benutzern leicht bedient werden kann.
Insgesamt betrachtet können der Entwurf und die Implementation als gelungen bezeichnet werden. Die wichtigsten Funktionalitäten konnten im Rahmen der Arbeit realisiert werden, und die wichtigsten Anwendungsfälle im System konnten abgedeckt werden.
Da diese Arbeit in erster Linie auf den Entwurf und die Implementierung der Benutzerschnittstellen für das PIA System fokussiert ist, bleiben noch einige technische Details für die gesamte Anwendung ungeklärt:

- Die Auswahl von Datenbanksystemen und konkreten Datenmodellen:
  In Frage kommen ein RDBMS oder OODBMS, die jeweils ihre Vor- und Nachteile besitzen. Bei der Auswahl des Datenbanksystems ist eine direkte Unterstützung von XML wünschenswert, so daß man kein eigenes Datenmodell für XML-Dokumente entwerfen muß.
- Datenbankanbindung:

Die Datenbankanbindung hängt vom dem jeweils eingesetzten Datenbanksystem ab. Es ist zwar wünschenswert, eine Persistenzschicht einzuführen, die den Persistenzmechanismus nach oben versteckt. Die Anforderung, daß die Persistenzschicht sowohl mit RDB als auch mit OODB umgehen kann, ist dabei jedoch nur mit großen Aufwand zu realisieren, so daß eine Entscheidung zwischen diesen beiden Alternativen unausweichlich erscheint. Für die Anbindung an RDBs steht JDBC zur Verfügung, während im Falle von OODBs die ODMG API verwendet werden kann.

- Client/Server Kommunikation:
  In einem System mit verteilter Architektur wie PIA müssen die Clients und der Server ständig miteinander kommunizieren. Es existieren verschiedene Techniken und Protokolle, um die Kommunikation zu realisieren. Wichtig ist dabei, daß das eingesetzte Protokoll kein herstellerspezifisches, sondern ein offenes Protokoll sein soll, um die Portabilität und die Interoperabilität des Systems zu erhöhen. Besonders interessant sind RMI von Javasoft, das eine sehr gute Integration zu Java bietet und CORBA von OMG, das der de-facto Standard für verteilte Objektsysteme ist.
- Weiterentwicklung des Frontends:
  Die Softwareentwicklung ist ein iterativer und inkrementeller Prozeß, der aus mehreren Zyklen besteht. In jedem Zyklus wird angestrebt, die Qualität der Software zu verbessern und die Funktionalität zu erweitern. Dementsprechend soll das Frontend von PIA erweitert und vervollständigt werden, so daß die nicht implementierten Anwendungsfälle realisiert werden. Besonders wichtig dabei sind Funktionalitäten wie z.B. Abo-Dienste für Suchaufträge und Personalisierung der Benutzeroberfläche. Durch Anbieten mehrerer Personalisierungsmöglichkeiten soll die Benutzeroberfläche in Zukunft für jeden Benutzer individuell konfiguierbar sein, so daß nach dem Anmelden eines Benutzers die Benutzeroberfläche genau so aussieht, wie der Benutzer sie individuell eingestellt hat. Bei dem Abo-Dienst handelt es sich um eine Funktionalität, die erlaubt, daß ein Benutzer seine Suchanfragen in PIA als Abonnement speichern kann. Die registrierten Abonnements werden regelmäßig vom System ausgewertet, und der Benutzer wird über das aktuelle Suchergebnis informiert. Ein weiterer Verbesserungspunkt ist die Optimierung der Systemperformance und das Vermeiden von Speicherlecks, um das Laufzeitverhalten des Frontends zu verbessern, da eine gute Performance einen sehr wichtigen Beitrag zur Akzeptanz des gesamten Systems leisten soll.
- Webfähige Frontends:
  Der Trend geht dahin, daß immer mehr Anwendungen in einer Browserumgebung laufen sollen. Da ein Browser mittlerweile fast auf jedem Rechner verfügbar ist, ist damit die Voraussetzung für eine einheitliche Laufzeitumgebung bestens erfüllt. Der wichtigste Vorteil von Webanwendungen ist das Entfallen des Wartungs- und Verteilungsaufwands von Software. Dadurch daß die Software nicht mehr an jeder

Arbeitsstation fest installiert wird, sondern erst zur Laufzeit dynamisch heruntergeladen wird, oder gar auf der Servermaschine ausgeführt wird, kann die Software zentral gepflegt werden, was den Administrationsaufwand deutlich reduziert. Es gibt grundsätzlich zwei Möglichkeiten, das PIA Frontend webfähig zu machen. Die erste Möglichkeit basiert auf Applets, die vom Server zur Laufzeit heruntergeladen werden und im Browser ausgeführt werden. Das Frontend von PIA kann ohne großen Aufwand in ein Applet umgewandelt werden, dazu ist das Erben des Hauptfensters von *JApplet* statt *JFrame* erforderlich. Darüber hinaus müssen einige Sicherheitsaspekte bzgl. der virtuellen Maschine (*virtual machine*) des Browsers mit eingeschränkten Zugriffen berücksichtigt werden. Es besteht jedoch ein Problem bei dieser Lösung, daß das PIA Frontend sehr viele Eigenschaften des neusten JDK wie Swing-Toolkit verwendet, die noch nicht von den gängigen Browsern unterstützt werden. Eine Lösung dazu bietet das Java Plug-In von *Sun Microsystem*, das erlaubt, daß das Applet in der virtuellen Maschine von Sun statt in der virtuellen Maschine des Browsers abläuft, so daß die neusten APIs verwendet werden können. Die zweite Alternative für ein webfähiges Frontend ist die Kombination von HTML-Ausgabe und Serverprogrammen in Form von CGI oder Servlets, wobei eine auf Servlets basierende Lösung gegenüber CGI über mehrere Vorteile wie z.B. höhere Effizienz und schnellere Antwortzeit verfügt. Bei dieser Lösungsvariante wird die Anwendungslogik auf dem Server ausgeführt, und das Ergebnis wird in Form von HTML zurück zum Browser geschickt. Vorteile dieser Lösung sind die höhere Kompatibilität und die geringere Anforderung an den Browser, da der Browser nur HTML verstehen und anzeigen muß. Sie weist aber zugleich einige Nachteile auf, die die Funktionalität des ganzen Systems einschränkt. HTML ist statisch und deshalb nicht geeignet für interaktive Benutzerschnittstellen. Ein Benutzer kann erst dann eine Antwort bekommen, wenn ein Formular abgeschickt wird, oder ein Link verfolgt wird. Bei jeder Aktion findet immer mindestens eine Netzwerkkommunikation statt. Bei dieser Lösung besteht noch ein anderes Problem, daß die Kommunikation zwischen Browser und Servlets auf HTTP basiert und somit zustandslos ist. Um den Zustand der Konversation zu aufrechtzuerhalten, muß ein zusätzlicher Mechanismus zur Verwaltung der Sitzungsinformation implementiert werden. Dieses Problem kann in der Praxis durch Verschicken von Cookies oder Umschreiben von URLs durch Anhängen von Sitzungsinformation gelöst werden.

Literaturverzeichnis

[Biewer97] Benno Biewer, *Fuzzy Methoden - praxisrelevante Rechenmodelle und Fuzzy-Programmiersprache,* Springer Verlag, 1997.

[Bradley98] Neil Bradley, *The XML Companion,* Addison-Wesley Longman, 1998.

[Booch91] Grady Booch, *Object Oriented Design with Applications,* The Benjamin/Cummings Publishing Company, Inc., 1991.

[Büchner99] Thomas Büchner, *Konzeption und Implementation von Fuzzy-Anfrageprädikaten in digitalen Bibliotheken,* Studienarbeit, Arbeitsbereich Softwaresysteme, Technische Universität Hamburg-Harburg, November 1999.

[Burkhardt97] Rainer Burkhardt, *UML – Unified Modeling Language, Objektorientierte Modellierung für die Praxis*, Addison-Wesley, 1997

[Buschmann98] Frank Buschmann, Regine Meunier, Hans Rohnert, Peter Sommerlad, Michael Stahl, *Pattern-orientierte Software Architektur, Ein Pattern-System,* Addison-Wesley Longman Verlag GmbH, 1998.

[Chang98] Dan Chang, Dan Harkey, *Client/Server Data Access with Java™ and XML,* Wiley Computer Publishing, 1998.

[Downing98] Troy Bryan Downing, *Java™ RMI, Remote Methode Invocation,* IDG Books Worldwide, Inc., 1998.

[D'Souza96] Desmond D'Souza, *Java: Design and Modelling opportunities,* Journal of Object-oriented Programming, September 1996, Volume 9, No. 5, Seite14-18.

[DuCharme99] Bob DuCharme, *XML : The Annotated Speciafication*, Prentice-Hall, Inc., 1999.

[Eckel98] Bruce Eckel, *Thinking in Java*, Prentice Hall PTR, Prentice-Hall, Inc., 1998.

[Eckstein98] Robert Eckstein, Marc Loy, Dave Wood, *Java™ Swing,* O'Reilly & Associates, Inc., 1998.

[Eriksson98] Hans-Erik Eriksson, Magnus Penker, *UML Toolkit,* John Weley & Sons, Inc., 1998.

[Fowler97a] Martin Fowler, *UML Distilled*. Addison-Wesley Longman, Inc, Massachusetts, 1997.

[Fowler97b] Martin Fowler, *Analysis Patterns, Resusable Object Models,* Addison-Wesley Longman, Inc, Massachusetts, 1997.

[Gamma95] Erich Gamma, *Design Patterns: Elements of resusable object-oriented Software.* Addison-Wesley Publishing Company, Inc., Massachusetts, 1995.

[Grand98] Mark Grand, *Patterns in Java, A Catalog of Reusable Design Patterns Illustrated with UML.* Willy Computer Publishing, 1998.

[Goldfarb99] Charles F. Goldfarb, Paul Prescod, *XML Handbuch*, Prentice Hall, 1999.

[Hesse94] W. Hesse, G. Barkow, H. von Braun, H.-B. Kittlaus, G. Scheschonk, *Terminologie der Softwaretechnik, Ein Begriffsystem für die Analyse und Modellierung von Anwendungssystemen, Teil 2: Tätigkeits- und ergebnisbezogene Elemente*, Informatik-Spektrum (1994) 17 96-105

[Jacobson92] Ivar Jacobson, *Objekt-oriented Software-Engineering, A use-case driven approach*, Addison-Wesley, Harlow, England, 1992.

[Kruchten99] Philippe Kruchten, *The Retinal Unified Process, An Introduction,* Addison-Wesley Longman, Inc., 1999.

[Krüger97] Guido Krüger, *Java™ 1.1 lernen, Anfangen, Anwenden, Verstehen,* Addison-Wesley Longman GmbH, 1997.

[Matthes99] Florian Matthes, Ulrike Steffens, *PIA - A Generic Model and System for Interactive Product and Service Catalogs*, Proceedings of ECDL'99 (European Conference on Digital Libraries), Paris, Springer-Verlag, 1999.

[McGrath98] Sean McGrath, *XML by Example, Building E-Commerce Applications,* Prentice Hall PTR, Prentice-Hall, Inc., 1998.

[Megginson98] David Megginson, *Structuring XML Documents*, Prentice Hall PTR, Prentice-Hall, Inc., 1998.

[Meyer98] Andre Meyer, *JFC 1.1 mit Java Swing 1.0, Ein Tutorial für die Gestaltung graphischer Benutzerschnittstellen,* Addison-Wesley Longman Verlag GmbH, 1998.

[Middendorf96] Stefan Middendorf, Reiner Singer, Stefan Strobel, *Java Programmier Handbuch und Referenz,* dpunkt, Heidelberg, 1996.

[Nüttgens99] Markus Nüttgens, Michael Hoffmann, Thomas Feld, *Objektorientierte Systementwicklung mit der Unified Modeling Language (UML),* Institut für Wirtschaftsinformatik (IWi), Universität des Saarlandes, Saarbrücken.

[Oestereich97] Bernd Oestereich, *Objektorientierte Softwareentwicklung mit der Unified Modeling Language, 3. Auflage*, Oldenbourg, 1997.

[Orfali99] Robert Orfali, Dan Harkey, Jeri Edwards, *Client/Server Survival Guide, Third Edition,* John Wiley & Sons, Inc., 1999.

[Pooley99] Rob Pooley, Perdita Stevens, *Using UML, Software Engineering with Objects and Components,* Addison Wesley Longman Limited, 1999.

[Quatrani98] Terry Quatrani, *Visual Modeling with Rational Rose and UML,* Addison-Wesley Longman, Inc., 1998.

[Rumbaugh91] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, William Lorensen, *Object-Oriented Modeling and Design,* Prentice-Hall Inc., 1991.

[Schmidt87] Joachim W. Schmidt, Peter C. Lockmann, *Datenbankhandbuch*, Springer Verlag, 1987.

[Topley98] Kim Topley, *Core, Java^{TM} Foundation Classes,* Prentice-Hall Inc., 1998.

[Wahl98] Günter Wahl, *UML Kompakt*, Objekt Spektrum 2/1998.

[Weiner98] Scott R. Weiner, Stephen Asbury, *Programming with JFC^{TM},* Wiley Computer Publishing, 1998.

[Zamir98] Saba Zamir, *Handbook of Object Technology,* CRC Press LLC, 1998.

[Zhang98] Lan Zhang, *Objektorientierte Analyse und Entwurf eines interaktiven, generischen Internet-Informationssystems,* Studienarbeit, AB 4-022 Softwaresysteme, Technische Universität Hamburg-Harburg, 1998.