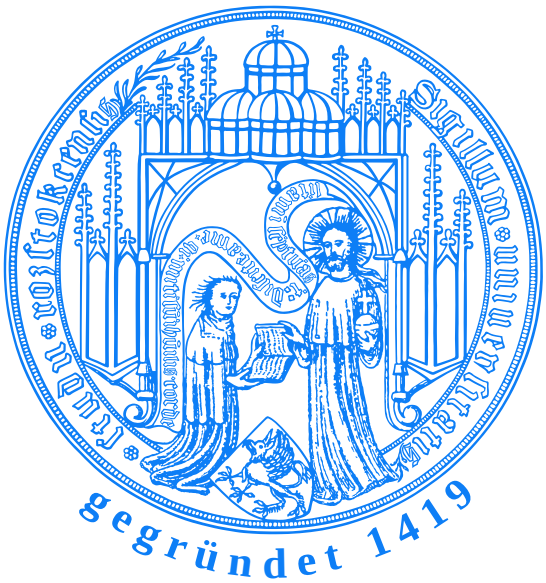
Studienarbeit

Integration von Entwurfsmuster in ArgoUML

vorgelegt von
René Lindhorst



Betreuer:

Prof. Dr.–Ing. Peter Forbrig

Institut für Informatik
Lehrstuhl Softwaretechnik

Universität Rostock

Rostock, den 19. Dezember 2006

Inhaltsverzeichnis

Abbildungsverzeichnis

Kapitel 1

Einleitung

Durch den Einsatz der Unified Modeling Language (UML) und Computer Aided Software Engineering (CASE) Tools in der Softwareentwicklung wird die Analyse und der Entwurf von objektorientierten Anwendungen vereinfacht. Zusätzlich ermöglicht die Nutzung von Entwurfsmustern (Design-Pattern) die Wiederverwendung von erfolgreichen Entwürfen und verbessert die Flexibilität des Entwurfs. Der Einsatz von Entwurfsmustern wird jedoch nicht von allen CASE-Tools unterstützt. So ist das Ziel dieser Studienarbeit die Entwicklung eines Pattern-Wizards für das Open Source Programm ArgoUML. Der Wizard wird als eine Erweiterung, ein so genanntes Modul, in ArgoUML eingebunden und soll die Integration von Entwurfsmustern in bestehende Softwaresysteme erleichtern sowie den Entwurf neuer Anwendungen unter Nutzung von Entwurfsmustern vereinfachen. Die Arbeit basiert auf der in [Man00] entwickelten und in [Sch02] überarbeiteten Pattern-Erweiterung für Rational Rose. Der entwickelte Wizard stellt eine Portierung und Weiterentwicklung dieser Erweiterung dar, wobei besonderer Wert auf eine objektorientierte Programmierung und eine leichtere Nutzbarkeit durch den Anwender gelegt wurde.

Die folgenden beiden Unterabschnitte geben eine kurze Einführung in die Unified Modeling Language und Entwurfsmuster. Im nächsten Kapitel wird ArgoUML ausführlich erklärt und in Kapitel 3 werden die für die Entwicklung des Pattern-Wizards wichtigsten internen Bestandteile von ArgoUML genauer betrachtet. Kapitel 4 beschreibt schließlich die Entwicklung des Pattern-Wizards und dessen Möglichkeiten. Abgeschlossen wird das Kapitel mit einem kurzen Beispiel-Szenario, das die Funktionsweise des Wizards noch einmal verdeutlichen soll. In den Schlussbemerkungen werden Möglichkeiten zur Erweiterung des Pattern-Wizards besprochen.

1.1 Unified Modeling Language

Die UML ist eine international anerkannte grafische Notation zur Erstellung objektorientierter Modelle für die Analyse und den Entwurf von objektorientierter Software. Wie in [Bal05] beschrieben, wurde sie von Grady Booch, Jim Rumbaugh und Ivar Jacobson entwickelt und 1996 als Version 0.91 veröffentlicht. Ende 1997 wurde die weiterentwickelte Version 1.1 von der Object Management Group (OMG) als Standard verabschiedet. Es folgten weitere Versionen mit einigen Verbesserungen. So etwa im Mai 2002 die UML 1.4. Wesentliche Änderungen und Erweiterungen erfuhr die Notation jedoch erst mit der 2005 veröffentlichten UML 2.0.

Die UML erfreut sich aus verschiedenen Gründen großer Beliebtheit. Zum einen können mit einer einheitlichen und verständlichen Notation die verschiedensten Elemente eines Softwaresystems visuell beschrieben werden. Zum anderen bildet sie eine einfache Grundlage für die Kommunikation zwischen den Entwicklern. In der UML gibt es eine Vielzahl von Diagrammen für die unterschiedlichsten Anwendungsgebiete. In dieser Arbeit werden jedoch nur die Klassendiagramme benötigt. Sie dienen der Darstellung von Klassen, mit ihren Attributen und Methoden, und der Beziehungen (Assoziationen, Aggregationen, Kompositionen, etc.) zwischen ihnen. Für weiterführende Informationen über UML, die verfügbaren Diagramme und Notationselemente möchte ich auf [ZGK04] und [Bal05] verweisen.

1.2 Entwurfsmuster

Ein Entwurfsmuster beschreibt mit einem gewissen Abstraktionsniveau eine Struktur zur Lösung eines allgemeinen objektorientierten Entwurfsproblems, wann es einsetzbar ist und welche Folgen sein Einsatz hat. Neben einer Klassenstruktur werden auch die innere Struktur der Klassen und die Schnittstellen beschrieben. Des Weiteren wird mit Hilfe von Codefragmenten die Funktionsweise des Entwurfsmusters verdeutlicht. Der Vorteil von Entwurfsmustern liegt u. a. in der Wiederverwendung von erfolgreich eingesetzten Entwürfen und einer daraus resultierenden Verbesserung der Qualität von Entwurfsentscheidungen.

Das Standardwerk über Entwurfsmuster ist [GHJV04] von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides und wurde Ende 1994 in der ersten Auflage veröffentlicht. Zu dieser Zeit war die Idee der Entwurfsmuster nicht mehr neu, aber es enthielt den ersten umfangreichen Musterkatalog und hatte damit einen großen Anteil an der starken Verbreitung von Entwurfsmustern in der Softwaretechnik. In diesem Buch der „Gang of Four“ (GoF), wie die vier Autoren auch genannt werden, sind die

Entwurfsmuster in drei Kategorien eingeteilt. Es wird zwischen erzeugenden, strukturorientierten und verhaltensorientierten Mustern unterschieden. Erzeugungsmuster beschäftigen sich mit der Objekterzeugung, Strukturmuster mit der Zusammensetzung von Klassen und Objekten und Verhaltensmuster mit der Zusammenarbeit und Zuständigkeitsverteilung zwischen verschiedenen Klassen und Objekten. Diese drei Kategorien werden noch bezüglich ihres Gültigkeitsbereiches in klassenbasierte und objektbasierte Muster unterteilt, je nachdem ob sich ein Muster hauptsächlich auf Klassen oder Objekte bezieht. Es sei erwähnt, dass es noch eine Vielzahl anderer Mustersammlungen in Büchern wie [BMR+98] oder auf Internetseiten wie [Hil] gibt, bei denen auch eine andere Klassifikation der Muster erfolgt.

Die Entwurfsmuster die im Verlauf dieser Studienarbeit Erwähnung finden, sollen anschließend kurz beschrieben werden. Da es sich dabei ausnahmslos um Entwurfsmuster der GoF handelt, stammen die Informationen dementsprechend aus [GHJV04].

1.2.1 Singleton

Beim Singleton handelt es sich um ein objektbasiertes Erzeugungsmuster das dafür sorgt, dass eine Klasse maximal ein Exemplar besitzt. Ferner stellt dieses Muster einen globalen Zugriffspunkt auf dieses Exemplar bereit. Die `Singleton`-Klasse selbst ist für die Verwaltung ihres einzigen Exemplars verantwortlich und kann so auf Befehle zum erzeugen neuer Objekte mit ihrem Exemplar antworten.

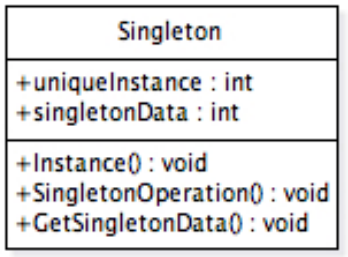


Bild 1.1: Struktur des Singleton-Entwurfsmusters

1.2.2 Facade

Die Fassade (engl. Facade) ist ein objektbasiertes Strukturmuster und wird eingesetzt um eine einheitliche Schnittstelle zur Funktionalität eines Subsystems bereitzustellen. Dies ist besonders hilfreich bei einem komplexen Subsystem, um dessen Benutzung zu vereinfachen. Die `Facade` enthält Informationen darüber welche Subsystemklasse für welche Anfragen zuständig ist und delegiert diese entsprechend an die zuständigen

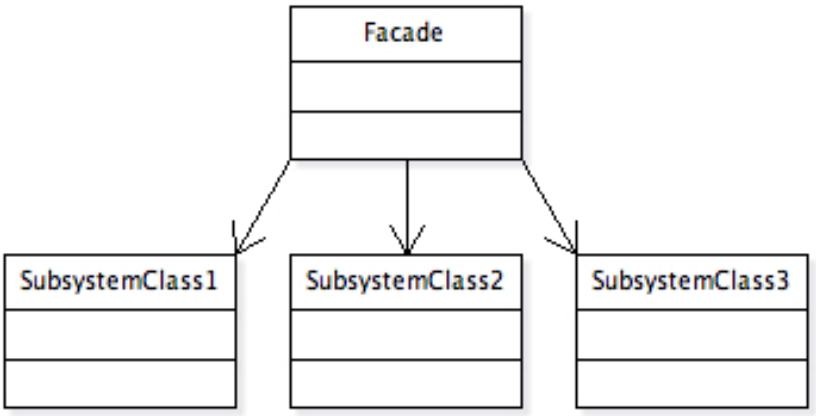


Bild 1.2: Struktur des Fassaden-Entwurfsmusters

Subsystemobjekte. Die Subsystemklassen enthalten die eigentliche Funktionalität und bearbeiten die ihnen zugewiesenen Anfragen.

1.2.3 Observer

Das objektbasierte Verhaltensmuster Observer wird im deutschen auch als Beobachter bezeichnet und dient dem Definieren einer 1-zu-n-Abhängigkeit zwischen Objekten. Es bewirkt, dass sobald sich der Zustand eines Objektes ändert, alle von ihm abhängigen Objekte benachrichtig werden und sich automatisch aktualisieren. Mit dem Beobachtermuster ist es somit möglich, die Konsistenz zwischen interagierenden Klassen sicherzustellen, ohne sie zu stark miteinander zu verzahnen. Bei Zu-

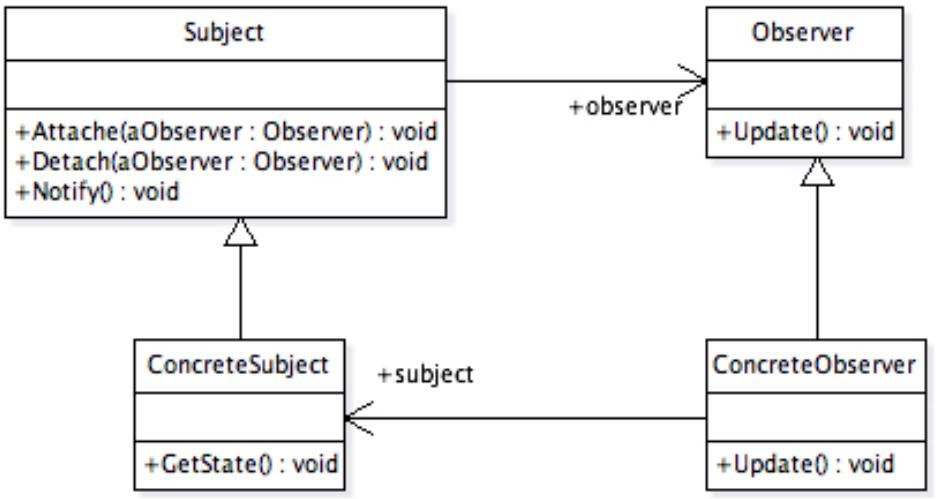


Bild 1.3: Struktur des Beobachter-Entwurfsmusters

standsänderungen des `ConcreteSubject` benachrichtigt es seine `Observer` woraufhin die einzelnen `ConcreteObserver`-Objekte das Subjekt nach dessen Zustandsänderung be-

fragen. Der `ConcreteObserver` nutzt diese Daten dann um seinen eigenen Zustand an den des `ConcreteSubject` anzupassen.

Kapitel 2

ArgoUML

ArgoUML ist ein Werkzeug, das den Softwareentwickler in der Analyse und beim Entwurf von objektorientierten Anwendungen unterstützt. Es unterscheidet sich in einigen Bereichen von anderen kommerziellen CASE-Tools wie Together oder Rose. ArgoUML ist Open Source und basiert auf vielen offenen Standards. Des Weiteren ist es eine reine Java Anwendungen und setzt Ideen der Kognitionspsychologie (engl. cognitive psychology) um. Auf diese Besonderheiten wird in folgenden Abschnitten noch genauer eingegangen, zuerst wird jedoch ein Blick auf die Geschichte des Projektes geworfen.

2.1 Entwicklung

ArgoUML wurde von einer kleinen Gruppe von Leuten unter der Führung von Ph.D. Jason Elliot Robbins als Forschungsprojekt der Universität von Kalifonien in Irvine entwickelt. Eine der Hauptziele Robins war es, wie er in einer E-Mail an die Entwickler-E-Mail-Liste äußerte: „ [...] to help promote good software engineering practices broadly throughout the industry, and to provide a basis on which more OOAD [Object Oriented Analysis and Design] tools could be built“[1].

Eine erste Version erschien 1998 und im Februar 1999 wurde ArgoUML als Open Source Projekt veröffentlicht. Zu diesem Zeitpunkt hieß es noch Argo/UML und wurde erst im November 2001 in ArgoUML umbenannt. Seit Januar 2000 wird das ArgoUML-Projekt über Tigris.org verwaltet. Dabei handelt es sich um eine mittelgroße Hosting-Plattform, vergleichbar mit SourceForge.net oder BerliOS, die auf Open Source Entwicklungswerkzeuge spezialisiert ist. Sie stellt die nötige Infrastruktur wie Bug-Tracker,

[1] `http://argouml.tigris.org/servlets/ReadMsg?list=dev&msgNo=20171` 22.08.2006

E-Mail-Listen, Versionsverwaltung, Webspace etc. zur Verfügung. Bis Mitte 2001 wurde ArgoUML über 100.000 mal von der Projekt-Homepage heruntergeladen und inzwischen sind es durchschnittlich 30.000 Downloads pro Monat[2].

Jason E. Robbins hat die Leitung des Projektes bereits vor einigen Jahren abgegeben. Nachdem Toby Baier kurzzeitig verantwortlich war, hat Linus Tolke seit Juni 2002 die Rolle des Projektleiters inne. Das Kernentwicklerteam von ArgoUML bilden 32 Personen, die berechtigt sind Änderungen am Quellcode vorzunehmen. Eine Vielzahl weiterer Entwickler leiten ihre Modifikationen bzw. Erweiterungen an das Kernentwicklerteam weiter. Es gibt keine genauen Zahlen darüber wie viele es sind, jedoch können die mehr als 300 Abonnenten der Entwickler-E-Mail-Liste als Anhaltspunkt dienen.

2.2 Besonderheiten

ArgoUML steht unter der BSD-Lizenz und unterliegt somit so gut wie keinen Einschränkungen bei den Nutzungsbedingungen. Das Programm steht jedem kostenlos zur Verfügung und der Quellcode ist frei zugänglich um ihn zu überprüfen, an die eigenen Bedürfnisse anzupassen und zu verbreiten. So gibt es auch verschiedene UML-Werkzeuge die auf ArgoUML basieren, wie MyEclipse und Sygel. Auch Posseidon for UML ist ein solche Werkzeug und wird seit Juni 2001 von der deutschen Firma Gentleware kommerziell als Closed Source vertrieben. Sie bieten verschiedene Versionen des Tools mit unterschiedlicher Funktionalität an. Darunter auch eine Community Edition zur kostenlosen Nutzung. Gentleware bietet für seine Produkte auch kommerziellen Support, was für viele Nutzer sehr wichtig ist. Jason E. Robbins hat sich zur Nutzung des ArgoUML Quellcodes als Basis für andere CASE-Tools in einer E-Mail an die Entwickler-E-Mail-Liste wie folgt geäußert: „ I am happy to see vendors create and sell their own unique value-add[ed tools] and be rewarded for it. Encouraging more small tool vendors is great for the industry in the long run.“[3]

Eine weitere Besonderheit sind die offenen Standards. Die Entwickler von ArgoUML haben versucht, so viele wie möglich für das Programm und seine Schnittstellen zu nutzen. Der Hauptvorteil, der in der Verwendung von offenen Standards liegt, ist die Interoperabilität. So ist ArgoUML konform zum UML 1.4 Standard der Object Management Group (OMG) und nutzt deren XML Metadata Interchange[4] (XMI) Format

[2] `http://argouml.tigris.org/downloadcount.html`

[3] `http://argouml.tigris.org/servlets/ReadMsg?list=dev&msgNo=20171` 22.08.2006

[4] `http://www.omg.org/technology/documents/modeling_spec_catalog.htm`

zum Speichern der Model-Informationen. XMI dient auch als Austauschformat zwischen verschiedenen CASE-Tools. Somit ist es theoretisch möglich, das in ArgoUML erzeugte Model in ein anderes Tool zu importieren. Jedoch enthält die XMI Datei keine Informationen über die graphische Darstellung des Models (dies wurde erst in der UML 2.0 Spezifikation ergänzt), so dass die Layout-Informationen dabei verloren gehen. ArgoUML speichert diese Informationen momentan für jedes angelegte Diagramm in einer extra Datei und nutzt dazu die Precision Graphics Markup Language (PGML). Es ist geplant, die PGML durch die UML Diagram Interchange Specifikation abzulösen, sobald ArgoUML die UML 2.0 Spezifikation umsetzt. Es ist auch möglich, das Model aus einem anderen CASE-Tool in ArgoUML zu importieren. Dazu muss das Modell im XMI-Format der Version 1.0, 1.1 oder 1.2 vorliegen, jedoch würde auch in diesem Fall die graphische Darstellung verloren gehen. Programme wie Meta Integration Model Bridge[5] sind darauf spezialisiert Modelle zwischen unterschiedlichen Tools zu konvertieren. So ist es z. B. möglich Modelle aus Rational Rose, welches keinen XMI-Export beherrscht, in ein für ArgoUML lesbares XMI-Format zu konvertieren. ArgoUML verpackt die XMI-Datei zusammen mit den PGML-Dateien, einer Datei (*.argo), die die Informationen über das Projekt enthält und einer weiteren Datei (*.todo), die die vom Nutzer erstellten Prüfkriterien enthält, in einer so genannten komprimierten ArgoUML-Projektdatei mit der Dateiendung „zargo“. Dabei handelt es sich um ein normales zip-Archiv, das man dementsprechend nach einer Umbenennung der Endung mit gängigen Packprogrammen wieder entpacken kann. Die in ArgoUML erzeugten UML-Diagramme können auch als Grafiken exportiert werden. Zu diesem Zweck werden die Formate Encapsulated PostScript (EPS), Graphic Interchange Format (GIF), Portable Network Graphics (PNG), PostScript (PS) und Scalable Vector Graphics (SVG) unterstützt. Eine weitere Besonderheit von ArgoUML ist, dass es auch die Object Constraint Language (OCL) unterstützt. Dazu wird das an der Technischen Universität Dresden entwickelte Dresden OCL Toolkit[6] verwendet.

Die Entscheidung ArgoUML in Java zu entwickeln, hat aufgrund der Interpretation des Java-Bytecodes einen Geschwindigkeitsnachteil gegenüber kompilierten Programmen. Jedoch wird dieser Unterschied durch die Leistungsfähigkeit moderner Computer immer geringer. Der Einsatz von Java hat außerdem einen großen Vorteil. Das Programm steht ohne Anpassungen allen gängigen Plattformen und somit allen potentiellen Nutzern zur Verfügung. Des Weiteren ist es möglich, dass Programm als Java Web Start (JWS) auf der Projekt-Homepage jedem Interessenten zum Ausprobieren bereitzustellen. Somit kann ArgoUML getestet werden, ohne das Programm auf dem

[5] `http://www.metaintegration.net/Products/MIMB/`

[6] `http://dresden-ocl.sourceforge.net`

eigenen Computer zu installieren.

Eine andere Besonderheit von ArgoUML ist, dass es in 10 verschiedenen Sprachen, u. a. sowohl in Englisch als auch in Deutsch, Russisch und Chinesisch verfügbar ist. Die Standardsprache des Programms und auch der Community ist US-Englisch. Wenn man ArgoUML das erste Mal startet und dies nicht über die Kommandozeile mit dem Parameter „-local <language>“ erfolgt (vgl. „Command Line Options“ im Quickguide), so startet es mit der „default locale“ des Computers. Sollte die Sprache nicht verfügbar sein, so wird stattdessen die Standardsprache verwendet. Über das Menü „Bearbeiten“ → „Einstellungen“ → „Erscheinungsbild“ gelangt man zu einer Auswahlbox, durch die man die Sprache festlegen kann, die ArgoUML ab dem nächsten Start nutzen soll.

Ein Hauptfokus von ArgoUML liegt in der Umsetzung der Ideen der Kognitionspsychologie, um dem Entwickler bei der Konzeption objektorientierter Systeme zu unterstützen. Die Theorie wird in der Ph.D. Dissertation von Jason E. Robbins [Rob99] genauer beschrieben. Die so genannten „design critics“ laufen als asynchrone Prozesse, parallel zu ArgoUML im Hintergrund, um den UML Entwurf zu überwachen und dem Nutzer eine Vielzahl von nützlichen Vorschlägen und Hinweisen anzubieten. So wird der Nutzer z. B. darauf aufmerksam gemacht, dass er Klassennamen entgegen der Konvention klein geschrieben hat oder noch kein Konstruktor definiert wurde. Eine genaue Auflistung und Beschreibung der überprüften Kriterien befindet sich im Nutzerhandbuch. ArgoUML geht aber noch einen Schritt weiter um den Nutzer optimal zu unterstützen. Zusätzlich zur Information über erkannte Schwachstellen bzw. Fehler, bietet das Programm auch einen Wizard um diese zu beheben. Es ist selbstverständlich auch möglich einzelne oder alle Kriterien zu deaktivieren bzw. eigene Prioritäten für sie festzulegen.

2.3 Verwendung

Um ArgoUML nutzen zu können, sind nur einige wenige Systemanforderungen zu erfüllen. Es wird ein Betriebssystem benötigt, das Java unterstützt und es muss eine Java Runtime Evironment (JRE) oder ein Java Development Kit (JDK) ab Version 1.4 installiert sein. Das Programm selbst belegt etwa 10 MB Speicherplatz auf der Festplatte, nachdem man es von der Projekt-Homepage als ZIP-Archiv heruntergeladen und entpackt hat. Abbildung 2.1 zeigt das ArgoUML Programmfenster in dem ein leeres Projekt mit einem Klassendiagramm und einem Use-Case-Diagramm geöffnet ist. Ein solches ArgoUML Projekt beinhaltet genau ein Model mit den zugehörigen Diagramminformationen und mehreren Modellelementen, die die UML-Beschreibung des zu modellierenden Systems bilden. Die Modellelemente können in einem oder meh-

reren Diagrammen dargestellt werden. ArgoUML kann jedoch jeweils nur ein Projekt zum Bearbeiten geöffnet haben.

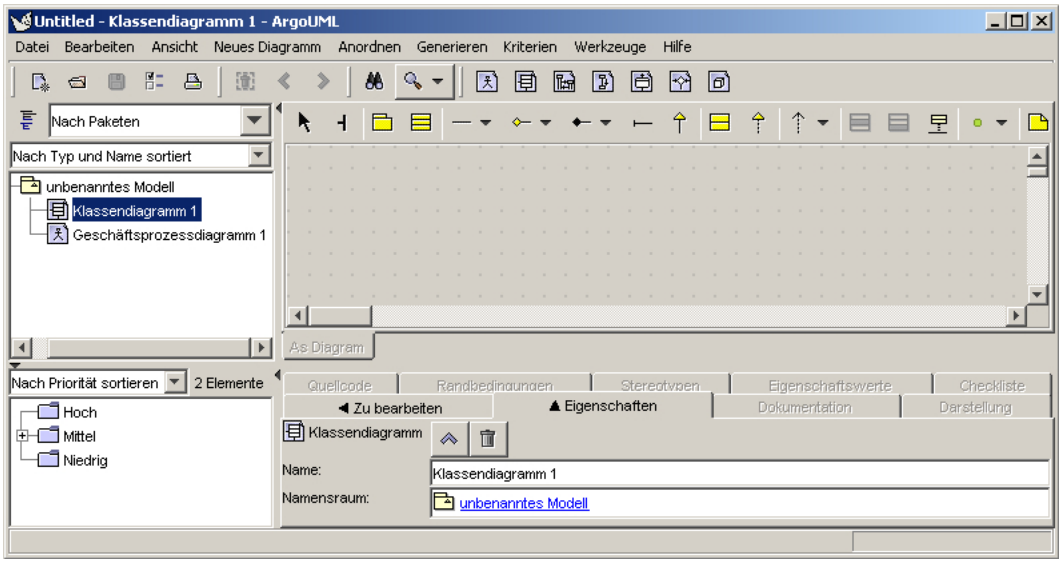


Abbildung 2.1: Benutzeroberfläche von ArgoUML

Die Benutzeroberfläche von ArgoUML hat im oberen Bereich, wie bei Programmen für Windows und Linux üblich, ein Menü und darunter eine Toolbar. Die für Mac OS X angepasste und in ein .app-Paket verpackte Version stellt das Menü in der Menüleiste am oberen Bildschirmrand dar, wie es unter OS X üblich ist. Der Bereich unterhalb der Toolbar unterteilt sich in vier Teilbereiche und nimmt den Großteil des Programmfensters in Anspruch. Oben links befindet sich der Explorer, der eine Baumstruktur des erzeugten UML Models mit allen Diagrammen und Modellelementen zeigt und somit eine zentrale Rolle bei allen Aktionen einnimmt. Die Elemente des Baumes lassen sich nach verschiedenen Kriterien gruppieren und sortieren, um die Ansicht den eigenen Bedürfnissen anzupassen. Rechts daneben liegt der Editierbereich, in dem die Klassen und ihre Beziehungen modelliert werden. Durch Drag&Drop können dem Diagramm Modellelemente vom Explorer hinzugefügt werden bzw. die Objekte eines Diagramms umgeordnet werden. Der Editierbereich hat eine eigene Toolbar, die sich dem zu bearbeitenden Diagrammtypen anpasst. Durch Auswahl der entsprechenden Schaltfläche in der Toolbar können im Diagramm neue Modellelemente erstellt werden. Eine sehr einfache Art neue Elemente zu erzeugen, die gleich mit bereits bestehenden Elementen verknüpft sind, ist über die so genannten „Quick-Links“. Diese erscheinen sobald man die Maus über ein selektiertes Element bewegt. Direkt unter dem Editierbereich findet der Detailbereich seinen Platz. In ihm können über eine Vielzahl von Reitern (Tabs) u. a. die Eigenschaften des gerade ausgewählten Model Elementes bearbeitet, die Darstellung des Elementes angepasst und OCL-Kriterien festgelegt

werden. Dort kann man auch den erzeugten Quellcode für eine gerade selektierte Klasse einsehen. Momentan wird die Quellcodegenerierung für Java, PHP, C++ und C# unterstützt. Es können aber weitere Sprachen hinzugefügt werden, da es sich bei der Quellcodegenerierung um ein „modulares framework“ handelt. Leider werden die Änderungen, die man in dem Quellcode vornimmt nicht gespeichert. Es ist also momentan für keine der unterstützten Sprachen ein Round-Trip Engineering möglich. Aber zumindest ein Reverse Engineering für Java Quellcode und JAR-Archive wird über „Datei“ → „Dateien importieren...“ unterstützt. Der Import für C++ Code ist ebenfalls in Arbeit. Jedoch ist er momentan noch in einer frühen Phase mit einigen Einschränkungen. Links neben dem Detailbereich befindet sich der ToDo-Bereich. In ihm werden die von den „design critics“ gefundenen Probleme und die vom Entwickler hinzugefügten Einträge in einer Baumstruktur dargestellt. Diese Einträge können nach Kriterien wie etwa Prioritäten, Problemen oder Zielen sortiert werden. Unterhalb dieser vier Bereiche bildet die Statusleiste den Abschluss des Programmfensters. Sie wird dazu genutzt dem Anwender kurze Hinweistext auszugeben. So wird man dort über das erfolgreiche Speichern des Projektes oder das Exportieren von Diagrammen informiert.

2.4 Dokumentation

Das ArgoUML Projekt bietet verschiedene englischsprachige Dokumente für Nutzer bzw. Entwickler, um diese bei ihrer Arbeit zu unterstützen. So gibt es für die Nutzer neben einem „ArgoUML Quick Guide“ [WTOO06] auch ein „ArgoUML User Manual“ [WTB+06] und eine Liste mit „Frequently asked questions for ArgoUML“ [Mor]. Das „Cookbook for Developers of ArgoUML“ [TK06] richtet sich, wie der Name schon sagt, an die Entwickler.

Die Kurzanleitung („ArgoUML Quick Guide“) ist dafür gedacht, dem Nutzer einen groben Überblick über ArgoUML zu verschaffen. In ihr werden die Installation des Programms erklärt und verfügbare Kommandozeilenparamter vorgestellt. Außerdem wird erläutert, was man unter Windows einstellen muss, damit beim Doppelklick auf eine .zargo-Datei diese mit ArgoUML geöffnet. Zusätzlich wird noch ganz kurz der Aufbau und die Funktionsweise des Programms erklärt. Für weiterführende Informationen wird der Anwender auf das Nutzerhandbuch verwiesen. Dabei handelt es sich um einen eigenständigen Teil des ArgoUML Projektes. Die Ziele dieses Teilprojektes sind zum einen eine Anleitung für den Einsatz von ArgoUML in der Objektorientierten Analyse (OOA) und im Objektorientierten Design (OOD) bereitzustellen, und zum anderen

eine Darstellung aller Komponenten von ArgoUML zu bieten.

Das Nutzerhandbuch („ArgoUML User Manual“) hat momentan einen Umfang von fast 400 Seiten, wobei für die Kapitel über „Analyse“ und „Design“ momentan nur die Struktur festgelegt wurde und sie erst nach und nach mit Inhalten gefüllt werden. In der FAQ findet der Nutzer Antworten auf eine Vielzahl von Fragen. So wird dort etwa erklärt, wie man Diagramme erstellt und später ausdrucken kann. Es wird aber auch auf Java Web Start, XMI und Betriebssystem spezifische Fragen eingegangen. Die FAQ wird regelmäßig um Problemstellungen bzw. wichtige Informationen, die in der Anwender-E-Mail-Liste diskutiert werden, erweitert.

Für die Entwickler ist das Entwickler-Kochbuch („Cookbook for Developers of ArgoUML“) gedacht, denn darin wird die interne Funktionsweise von ArgoUML beschrieben. Viele Informationen des nächsten Kapitels stammen daraus. Mit seinen knapp 150 Seiten bietet es jedoch an vielen Stellen nur einen recht groben Überblick, so dass ein Blick in die Javadoc[7] bzw. ein Untersuchen des Quellcodes unerlässlich ist. Das Entwickler-Kochbuch wird jedoch regelmäßig überarbeitet und erweitert und enthält somit immer mehr wichtige Informationen, vor allem für neue Entwickler. Es ist zusätzlich ratsam, die Entwickler-E-Mail-Liste zu abonnieren und die Diskussionen der Entwickler zu verfolgen, bzw. sich auch selbst daran zu beteiligen, um so die Design-Entscheidungen besser nachvollziehen zu können.

[7] `http://argouml-stats.tigris.org/nonav/javadocs/javadocs-0.22/`

Kapitel 3

ArgoUML Erweitern

ArgoUML kann durch so genannte Module um neue Funktionen erweitert werden, ohne das Änderungen an ArgoUML selbst vorgenommen werden müssen. In anderen Tools nennt man solche Module auch Add-Ins oder Plug-Ins, wie etwa bei Gentlewares Poseidon. In den folgenden Abschnitten wird erklärt wie die Modul-Architektur von ArgoUML funktioniert und wie andere, für die Pattern-Erweiterung wichtige Bestandteile von ArgoUML aufgebaut sind und wie sie genutzt werden können.

3.1 Modul-Subsystem

Der Programmcode von ArgoUML ist in Subsysteme unterteilt, die jeweils für bestimmte Aufgaben verantwortlich sind, um so die Entwicklung zu vereinfachen. Das Modul-System ist ein solches Subsystem und es ist verantwortlich für das Einbinden von Erweiterungen. Momentan gibt es zwei verschiedene Mechanismen um Module zu laden. Einen alten „Module Loader“ und einen neuen, der das Schreiben von Modulen vereinfacht und für alle neuen Module genutzt werden soll. Beim Lesen des Entwickler-Kochbuchs ([TK06]) muss man daher auch genau aufpassen, welcher der beiden erklärt wird, so dass man nicht fälschlicherweise den alten für seine Erweiterung nutzt. Der ursprüngliche „Module Loader“ ist jedoch seit Version 0.21.1 als veraltet markiert. Aus diesem Grund wird sich die Situation in naher Zukunft wieder vereinfachen, da nach den in [TK06] beschriebenen Programmierstandards solche Klassen nach einer stabilen Versionen entfernt werden können.

Die Module werden beim Start von ArgoUML durch den `ModuleLoader2`, so der Name des neuen „Module Loaders“, der sich unter `org.argouml.moduleloader` befindet, geladen. Es ist auch vorgesehen, dass die Module aus ArgoUML heraus eingebunden,

bzw. wieder deaktiviert werden können. Damit der `ModulerLoader2` ein Modul erkennt, muss eine Klasse des Moduls das Interface `ModuleInterface` implementieren. Es enthält Methoden zum Aktivieren und Deaktivieren sowie zum Identifizieren des Moduls. Die Manifest-Datei die sich zusammen mit allen Klassen und den benötigten Ressourcen in dem Modul-jar-Archiv befindet, verweist auf diese Klasse. Bei den Modulen wird zwischen externen und internen unterschieden, wobei es nur darum geht, ob das Modul ein eigenständiges jar-Archiv oder Teil des `argouml.jar` ist. Die externen Module müssen sich in einem Unterodner mit dem Namen „ext“ im Programmordner befinden, damit sie vom „Module Loader“ gefunden werden. Sobald ein Modul aktiviert ist, kann es nicht mehr vom Rest von ArgoUML unterschieden werden. Es kann somit auf alle Programmierschnittstelle (APIs) der Subsysteme und anderer Module zugreifen.

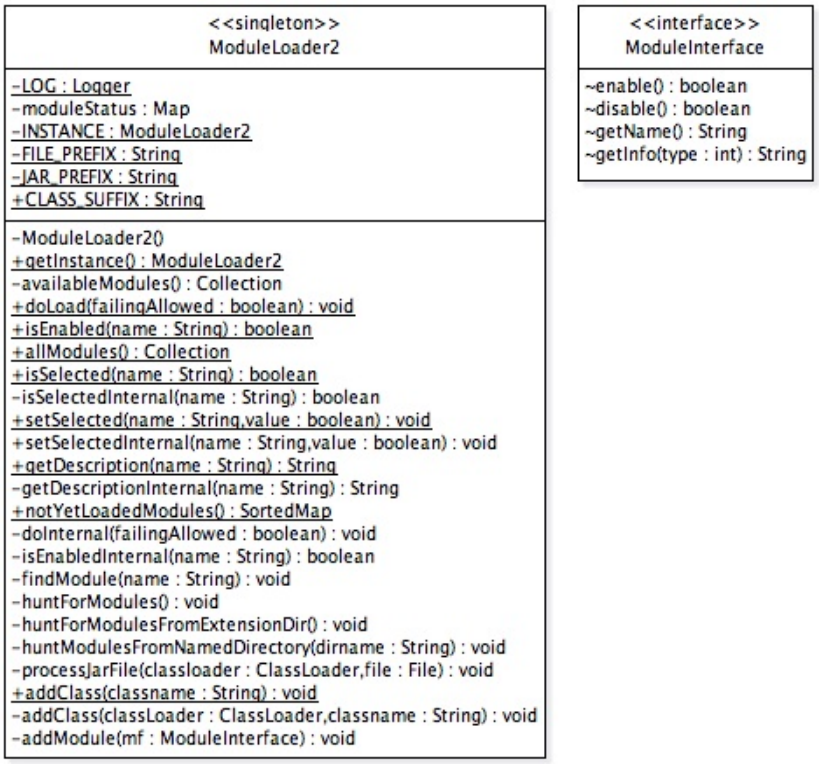


Abbildung 3.1: Wichtige Bestandteile des Modul-Subsystems

3.2 Modell-Subsystem

Das Modell-Subsystem wurde geschaffen, um allen anderen Komponenten von ArgoUML konsistente Schnittstellen bereitzustellen, mit denen Manipulationen des „model repository“ möglich sind. So ist es für die Komponenten uninteressant, ob NetBeans Model Data Repository (MDR), Novosoft UML Library (NSUML) oder ei-

ne andere Implementerieung verwendet wird. Für das UML 1.3 Metamodel wurde die NSUML Bibliothek verwendet und für das 1.4 Metamodel setzt man jetzt auf eine Implementierung, die NetBeans MDR nutzt. Die Manipulationen am Modell erfolgen über so genannte Factory- und Helper-Klassen. Die Factories dienen im Allgemeinen zum Erschaffen neuer Modellelemente, und die Helper bieten die Möglichkeit die Modellelemente zu verändern. Der Zugriff auf diese Klassen erfolgt über statische Methoden der Klasse `org.argouml.model.Model`. Sie ist es auch, die entscheidet welche der oben genannten Implementationen genutzt wird. Dies ist möglich, da jede Implementierung die selben Factory- und Helper-Klassen bietet und ihre Methoden durch Interfaces vorgegeben werden. Zu den wichtigsten dieser Klassen gehören `CoreHelper` und `CoreFactory`. So kann man z. B. mit `Model.getCoreHelper().setName(handle, name)` den Namen eines Modellelementes ändern oder mit `Model.getCoreFactory().buildAttribute(model, type)` ein neues Attribut erzeugen. Andere wichtige Klassen sind die in Abbildung 3.2 dargestellten `ModelManagementHelper` und `ModelManagementFactory`. Mit ihnen kann man z. B. alle Modellelemente eines bestimmten Typs ermitteln oder ein neues UML Package erzeugen. Eine weitere wichtige Klasse des Modell-Subsystems ist `Facade`. Wie

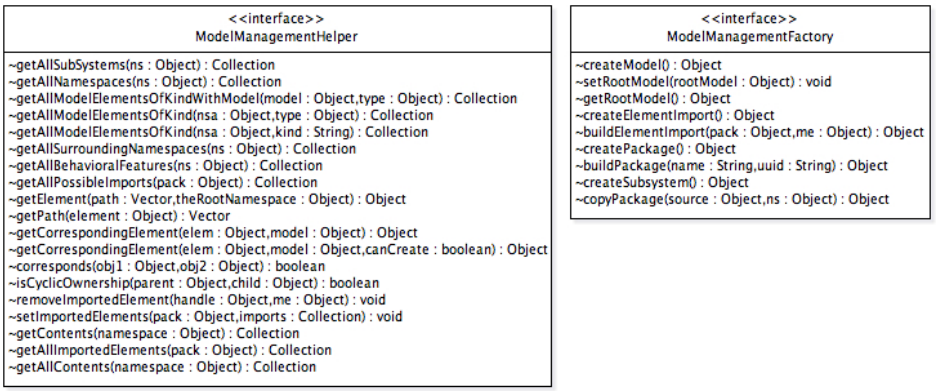


Abbildung 3.2: Beispiel für Factories und Helper

der Name schon andeutet, ist sie eine Umsetzung des Fassaden-Entwurfsmusters (engl. Facade). Der Zugriff auf sie erfolgt über `Model.getFacade()` und sie erlaubt nur einen lesenden Zugriff auf das Modell.

3.3 Internationalisierung

Die Internationalisierung, auch i18n genannt, ist ein Infrastruktur Subsystem von ArgoUML und es steht anderen Subsystemen zur Verfügung um Zeichenketten zu übersetzen. Alle Zeichenketten die in der GUI auftreten, werden in „property files“ unter einer Kennung abgelegt und diese Dateien werden während des Pro-

grammstarts in die `PropertyResourceBundles` geladen. Die Syntax der Kennungen ist `wort1.wort2.wort3`, wobei `wort1` der erste Teil des Namens der Datei ist, in dem sich die Kennung befindet. Es gibt Dateien für Labels, Menüs, Tabs etc. und alle haben die Endung „properties“. Dateien die zu einer anderen Lokalisierung als der der Standardsprache US-Englisch gehören, bekommen einen Namenszusatz um die entsprechende Sprache identifizieren zu können. Die deutschen Lokalisierungsdateien bekommen z. B. „_de“ angehängt. Listing 3.1 zeigt exemplarisch einen Teil einer solchen Datei.

```
tab.appearance = Appearance
tab.checklist = Checklist
tab.checklist.warning = Warning! What items you check is not saved
tab.checklist.description = Description
tab.constraints = Constraints
tab.documentation = Documentation
```

Listing 3.1: Auszug aus der `tab.properties`

Die Kennungen werden in der GUI als Parameter für die Lokalisierungsmethode `localize` verwendet (vgl. Listing 3.2), die während der Laufzeit des Programms die zu ihnen passenden Zeichenketten der gewählten Sprache findet und ausgibt. Die Lokalisierungsmethode wird von der Klasse `org.argouml.i18n.Translator` als statische Methode bereitgestellt.

```
import org.argouml.i18n.Translator;
...
String localized = Translator.localize("tab.appearance")
...
```

Listing 3.2: Verwendung des Internationalisierungs Subsystems

Die Dateien für die Standardsprache US-Englisch befinden sich in ArgoUML unter `org.argouml.i18n`. Alle anderen Sprachen sind als eigenständige Projekte konzipiert, die als Module in ArgoUML integriert werden. So ist es ohne größere Probleme möglich, ArgoUML um neue Sprachen zu erweitern.

3.4 Build-Prozess

ArgoUML verwendet für den Build-Prozess Apache Ant, um u. a. die einzelnen Bestandteile von ArgoUML zu kompilieren und aus den Class-Dateien verschiedene Jar-Archive zu erstellen. Ant ist ein in Java geschriebenes Build-Tool des Jakarta-Projekts[1], einem Teil der Apache Software Foundation, und steht unter der Apache

[1] `http://jakarta.apache.org`

Software License. Es erfreut sich großer Beliebtheit im Java Umfeld und in Publikationen wie [Edl02] wird es auch als Nachfolger von `make` in diesem Bereich bezeichnet. Ein Grund dafür ist nicht nur die Flexibilität von Ant, sondern sicherlich auch die gute Integration in Entwicklungsumgebungen (Integrated Development Environment) wie Eclipse und NetBeans. Ant ließt aus einer XML-Datei (`build.xml`) die Regeln darüber, welche Kommandos worauf angewendet werden sollen. Zu diesen Kommandos, die in Ant Tasks genannt werden, gehören u. a. der Aufruf verschiedener Java Compiler, Zugriff auf Versionsverwaltungen, Kopierkommandos und das Erstellen verschiedener Archive. Tasks werden in Anweisungsblöcken, so genannten Targets gruppiert und bilden einzelne Arbeitsschritte, die auch von anderen Arbeitsschritten abhängen können. In ArgoUML gibt es neben der Haupt-Build-Datei mit einer Vielzahl von Targets noch zwei kleinere für das Model Subsystem und die MDR Implementierung des Model Subsystems, die von der Hauptdatei verwendet werden. Ferner hat jedes Modul seine eigene `build.xml`-Datei. Weitere Informationen über Ant erhält man in der offiziellen Dokumentation[2] oder in Büchern wie [Mat05] oder [Edl02], welche auch als Basis für meine Ausführungen und Arbeiten mit Ant dienten.

3.5 Besonderheiten und Richtlinien

Beim Programmieren an ArgoUML bzw. an einem Modul für ArgoUML gibt es einige Besonderheiten bzw. Richtlinien die ein Entwickler beachten sollte. Im folgenden sollen einige der wichtigsten noch kurz besprochen werden.

Die Versionsnummer von ArgoUML setzt sich nach dem folgendem Schema zusammen: `Hauptversionsnummer.Nebenversionsnummer.Revisionsnummer`. Die Hauptvertsionsnummer ist noch immer eine 0, was jedoch nicht darauf hindeutet, dass die Entwicklung noch nicht weit fortgeschritten ist, sondern vielmehr, dass selbst nach fast acht Jahren Entwicklung noch nicht die gesteckten Ziele erreicht wurden. Eine stabile Version, die von jedem genutzt werden kann, trägt immer eine gerade Nebenversionsnummer, wie z. B. 0.22. Falls in einer solchen Version Fehler auftreten, die dringend beseitigt werden müssen, bekommt die verbesserte Version eine entsprechende Revisionsnummer angehängt, so z. B. 0.22.1. Entwicklerversionen tragen immer ungerade Nebenversionsnummer. So wird in den 0.23.* Versionen an Verbesserungen bzw. Erweiterungen gearbeitet, die schließlich zur Veröffentlichung der nächsten stabilen Version 0.24 führen werden.

Der Programmierstil in ArgoUML basiert auf den „Code Conventions for the Java

[2] `http://ant.apache.org/manual/index.html`

Programming Language“[3] von Sun Microsystems mit einigen Erweiterungen. Demnach müssen etwa alle Klassen, Methoden und Variablen mit Javadoc Kommentaren versehen werden und innerhalb von Methoden sollen keine Kommentare geschrieben werden. Wenn in einer Methode etwas komplexeres geschieht, was einer Erklärung bedarf, soll dieser Teil nach Möglichkeit in eine eigene Methode auslagert werden und diese einen entsprechenden Kommentar bekommen. Eine weitere wichtige Erweiterung der „Code Conventions“ ist der Umgang mit veralteten `public` oder `protected` Klassen, Methoden und Attributen. Bevor man solche löschen darf, müssen sie für eine stabile Version als `deprecated` markiert sein. Denn nur so ist es möglich, dass Modul-Entwickler ihre Anpassungen während den Entwicklerversionen von ArgoUML vornehmen können und aktualisierte Module zusammen mit einer neuen stabilen Version von ArgoUML erscheinen können. Des Weiteren ist es auch für die Entwicklung von ArgoUML selbst wichtig, um genügend Zeit für entsprechende Anpassungen und Test an abhängigen Programmteilen zu haben. Für zusätzliche Regeln zum Programmierstil, zum Umgang mit dem Subversion Repository und zu den Bestimmungen über die Nutzung externer Bibliotheken, soll an dieser Stelle auf das „Standards for coding“ Kapitel in [TK06] verwiesen werden.

Um Meldungen zum Suchen von Fehlern oder andere Informationen über den Programmablauf auf der Kommandozeile auszugeben, kann man das Logging Subsystem von ArgoUML verwenden. Dieses System nutzt log4j[4] für seine Aufgabe. Da man im ArgoUML Quellcode laut [TK06] kein `System.out.println` verwenden soll, ist es wichtig zu wissen, wie man das Logging System in eigenen Klassen nutzen kann. Der folgende Quellcode soll das kurz demonstrieren:

```
// Klassendefinition des Loggers importieren
import org.apache.log4j.Logger;

public class deineKlasse {
    // den Logger erzeugen
    private static final Logger LOG =
      Logger.getLogger(deineKlasse.class);

    public void eineMethode() {
        // eine Debugmeldung ausgeben
        LOG.debug("Debug-Test!");
    }
}
```

Listing 3.3: Verwendung des Logging Subsystems

[3] `http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html`
[4] `http://jakarta.apache.org/log4j/`

Eine Übersicht über die verschiedenen Level der Log-Einträge befindet sich ebenso wie eine Erklärung zum Aktivieren der verschiedenen Möglichkeiten des Logging in [TK06]. Eine sehr einfache Möglichkeit das Logging zu aktivieren, ist über einen der vorgefertigten Targets der `build.xml` Datei von ArgoUML. So startet `./build debug` ArgoUML von den kompilierten Quellen und aktiviert das Debug-Level Logging, worauf alle Debug-Meldungen in der Konsole ausgegeben werden.

Kapitel 4

ArgoUML Pattern-Wizard

Nachdem in den vorangegangenen Kapiteln ArgoUML und dessen interner Aufbau erklärt wurden, soll nun genauer auf den entwickelten Pattern-Wizard eingegangen werden. Dazu werden die wichtigsten Entwicklungsschritte erläutert und die Möglichkeiten des Wizards ausführlich beschrieben.

4.1 Implementierung

4.1.1 Vorbereitungen

Bevor mit der eigentlichen Entwicklung des Moduls begonnen werden kann, sind ein paar Vorbereitungen nötig. Als erstes benötigt man den Quellcode von ArgoUML. Man könnte den zu jeder Version zum Download bereitgestellten Quellcode nutzen. Jedoch ist es besser sich den aktuellsten Code aus dem Subversion Repository auszuchecken. Da für die Entwicklung Eclipse verwendet wurde, konnten die vom ArgoUML Projekt bereitgestellten „Team Project Set“ Dateien genutzt werden. Es werden etwa 30 MB Quellcode und Bibliotheken heruntergeladen und in verschiedene Eclipse-Projekte aufgeteilt. Auch die nötigen Einstellungen für Eclipse werden vorgenommen. Eine genaue Beschreibung dazu befindet sich im Kapitel „Setting up Eclipse 3“ des Entwickler-Kochbuchs.

4.1.2 Erzeugen der Ant Konfigurationsdatei

Nachdem der Quellcode heruntergeladen wurde und ArgoUML aus Eclipse heraus kompiliert und gestartet werden kann, muss für das zu erstellende Modul ein neues Java-

Projekt angelegt werden. In diesem Projekt wird als erstes eine Konfigurationsdatei für Ant benötigt. Die für den Pattern-Wizzard erstellte `build.xml` befindet sich im Anhang A.1. Die Build-Datei beginnt mit einer obligatorischen, XML-typischen Anweisung, die die Ant Version und die verwendete Kodierung festlegt. Anschließend erfolgt in Zeile 17 die Definition des Projektes mit dem Namen „argouml-pattern-wizard“ und der Festlegung von `usage` als Standardtarget, falls Ant ohne explizite Angabe eines Targets aufgerufen wird. Die einzelnen Aufgaben die Ant erfüllen soll, sind in separate Targets aufgeteilt, wobei jedes von ihnen seinen entsprechenden Vorgänger mit Hilfe des `depends`-Attributs aufruft. Dadurch entsteht eine lineare Verkettung der einzelnen Targets. Das Projekt enthält zehn Targets und importiert einen weiteren aus einer anderen Build-Datei. Dieser Import erfolgt in Zeile 23 mit dem `import`-Task. Die Aufteilung der Build-Datei orientiert sich an der im Abschnitt „Developing in a subproject“ des Entwickler-Kochbuchs für alle Unterprojekte geforderten Struktur und erweitert diese um einige zusätzliche Targets. Das für die Entwicklung wohl wichtigste Target ist `debug` in Zeile 166. Seine Aufgabe ist es ArgoUML im debug-Modus zu starten, um die Funktion des Moduls zu testen. Um ArgoUML ohne das Logging zu starten nutzt man den `run`-Target in Zeile 156. Bevor einer der beiden Targets ArgoUML ausführen kann, muss in Zeile 144 im `install`-Target das ext-Verzeichnisse für Erweiterungen im ArgoUML-Ordner erzeugt werden. Das zuvor im `jar`-Target (Zeile 120) erzeugte jar-Archiv des Pattern-Wizards wird anschließend dorthin kopiert. Beim Erstellen des jar-Archivs wird auch die benötigte `manifest.mf` Datei erzeugt. In ihr sind der „Class-Path“ und der Name der Klasse, die das `ModuleInterface` implementiert, die wichtigsten Einträge. Die Class-Dateien die im `jar`-Target zum Erstellen des jar-Archivs benötigt werden, entstehen im `compile`-Target (Zeile 104) beim Kompilieren des Quellcodes durch den `javac`-Task. Doch bevor der Java-Compiler seinen Dienst verrichten kann, wird der `prepare`-Target in Zeile 76 ausgeführt. Seine Aufgabe ist es im Ordner „build“ einen „bin“-Ordner anzulegen in dem die Class-Dateien abgelegt werden können. Weiterhin wird der Quellcode des Moduls ebenfalls in den „build“-Ordner kopiert (Zeile 81) und dabei werden die im Quellcode befindlichen Token `@VERSION@` und `@BUILD@` durch die entsprechenden Zeichenketten ersetzt. So ist sichergestellt, dass die später im Pattern-Wizard angezeigten Versionsinformationen auch korrekt sind. Wie man in Zeile 76 sehen kann ist der `prepare`-Target von drei weiteren abhängig. Zum einen vom `init`-Target (Zeile 28), der verschiedene so genannte Properties definiert, bzw. diese aus Property-Dateien einließt. Properties sind vergleichbar mit Konstanten. Sie dienen als Platzhalter für einen Wert und sobald sie gesetzt sind können sie nicht mehr geändert werden. Der zweite Target von dem `prepare` abhängt, ist `clean` (Zeile 180). Er ist dafür verantwortlich die Resultate eines früheren Durchlaufs zu löschen. Die dritte Abhängigkeit besteht zum importierten `generate-buildnumber`-Target, der

in der buildnumber.xml Datei definiert ist. Seine Aufgabe ist es, wie der Name schon sagt, die Build-Nummer zu erzeugen. Sie besteht aus vier Ziffern, wobei die erste die Anzahl der Jahre darstellt, die das Modul bereits existiert und die letzten drei stehen für die Nummer des Tages an dem das Modul kompiliert wurde. Die Build-Nummer 1242 steht somit für den Tag 242 des ersten Jahres, in dem das Modul verfügbar ist und die Nummer 242 steht für den 30. August. Da Ant keine Tasks für Kontrollstrukturen bietet, diese jedoch zum Berechnen der Nummer des Tages nötig sind, wird dafür Ant-Contrib[1] eingesetzt. Dabei handelt es sich um eine Sammlung von zusätzlichen Tasks, die man, nachdem sie per `taskdef`-Task eingebunden wurden, nutzen kann. Neben den bereits vorgestellten Targets gibt es noch einen `javadoc`-Target (Zeile 192) zum Erstellen einer Dokumentation und den bereits erwähnten Standardtarget `usage` (Zeile 54), der eine Information über verfügbare Targets ausgibt. Beide benötigen einige Properties und hängen deshalb ebenfalls vom `init`-Target ab.

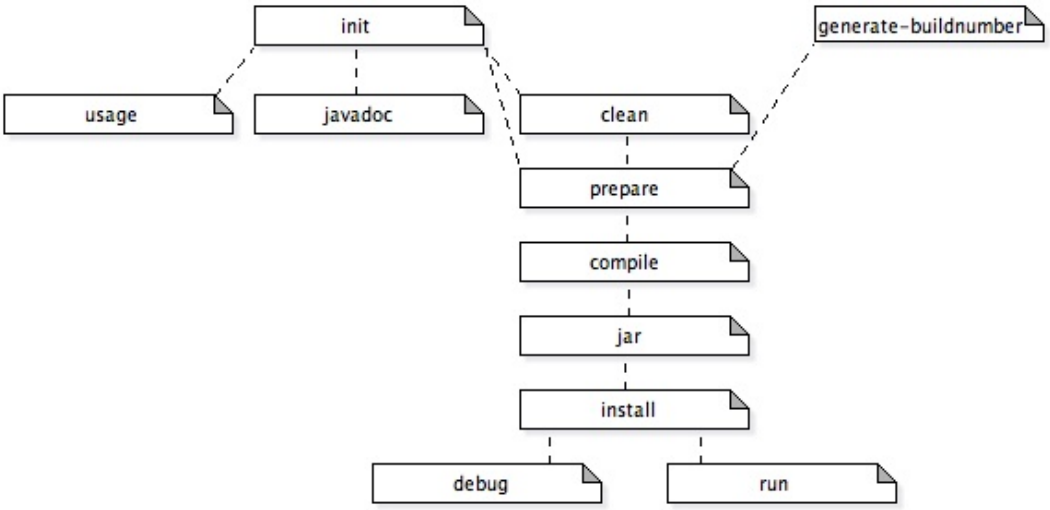


Abbildung 4.1: Abhängigkeiten der Ant Targets

4.1.3 Implementation der Hauptklasse des Moduls

Der nächste Schritt besteht darin eine Klasse zu schreiben, die das `ModuleInterface` implementiert. Die Klasse `ActionOpenPatternWizard` übernimmt diese Aufgabe und stellt somit die Hauptklasse dar, die ArgoUMLs `ModuleLoader2` erkennt und während des Programmstarts bzw. nach dem Aktivieren als Modul einbindet. Dafür muss sie vier Methoden implementieren (vgl. Abbildung 4.2). Zum einen die Methode `enable()` um das Modul aktivieren zu können, und zum anderen `disable()` um es bei Bedarf

[1] `http://ant-contrib.sourceforge.net/tasks/`

auch wieder zu deaktivieren. Die anderen beiden Methoden dienen dazu mehr über das Modul zu erfahren. So liefert `getName()` den Namen des Moduls und über `getInfo()` bekommt man je nach Parameter den Autor, eine kurze Beschreibung oder die Version des Moduls.

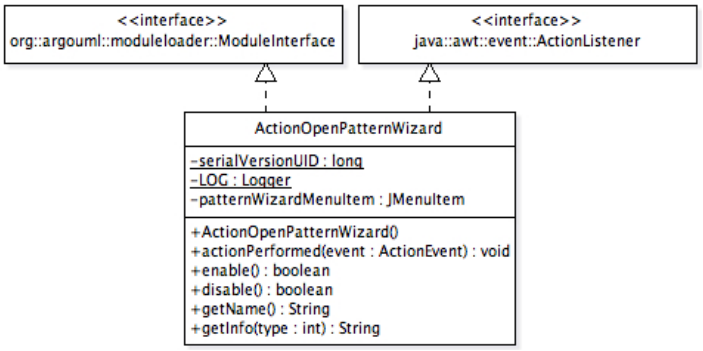


Abbildung 4.2: Hauptklasse zum Aktivieren des Pattern-Wizards

Beim Aktivieren des Moduls soll das Menü von ArgoUML um einen Eintrag für den Pattern-Wizard im Werkzeuge-Menü erweitert werden. Dazu ist es nötig, wie in Listing 4.1, in der `enable()`-Methode die `GenericArgoMenuBar` zu ermitteln. Sie definiert ArgoUMLs Menüleiste. Über die `getTools()`-Methode bekommt man das `JMenu`-Objekt des Werkzeug-Menüs, welchem man das `JMenuItem` des Pattern-Wizards anhängen kann.

```
public boolean enable() {
    GenericArgoMenuBar menubar =
     (GenericArgoMenuBar)ProjectBrowser.getInstance().getJMenuBar();
    menubar.getTools().add(patternExtMenuItem);
}
```

Listing 4.1: Menüeintrag hinzufügen

Analog muss der Menüeintrag auch wieder entfernt werden, wenn das Modul deaktiviert wird. Damit `ActionOpenPatternWizard` auf die Aktion des Nutzers reagieren kann, muss die Klasse auch einen `ActionListener` implementieren. Nur so ist es möglich auf das ausgelöste `ActionEvent` mit dem öffnen des Pattern-Wizards zu reagieren.

4.1.4 Aufbau der Benutzeroberfläche

Wenn der Pattern-Wizard über „Werkzeuge“ → „Pattern-Wizard...“ aktiviert wird, öffnet sich ein nicht-modales Dialogfenster. Dieser Dialog wird von der Klasse `PatternWizard` (vgl. Abbildung 4.3) erzeugt. Sie ist für die Interaktion mit dem Benutzer

zuständig und alle von ihm vorgenommenen Änderungen werden an das Datenmodel weitergegeben. Die Klasse erweitert `ArgoDialog` welche in ArgoUML für Dialoge mit lokalisierten Buttons genutzt wird.

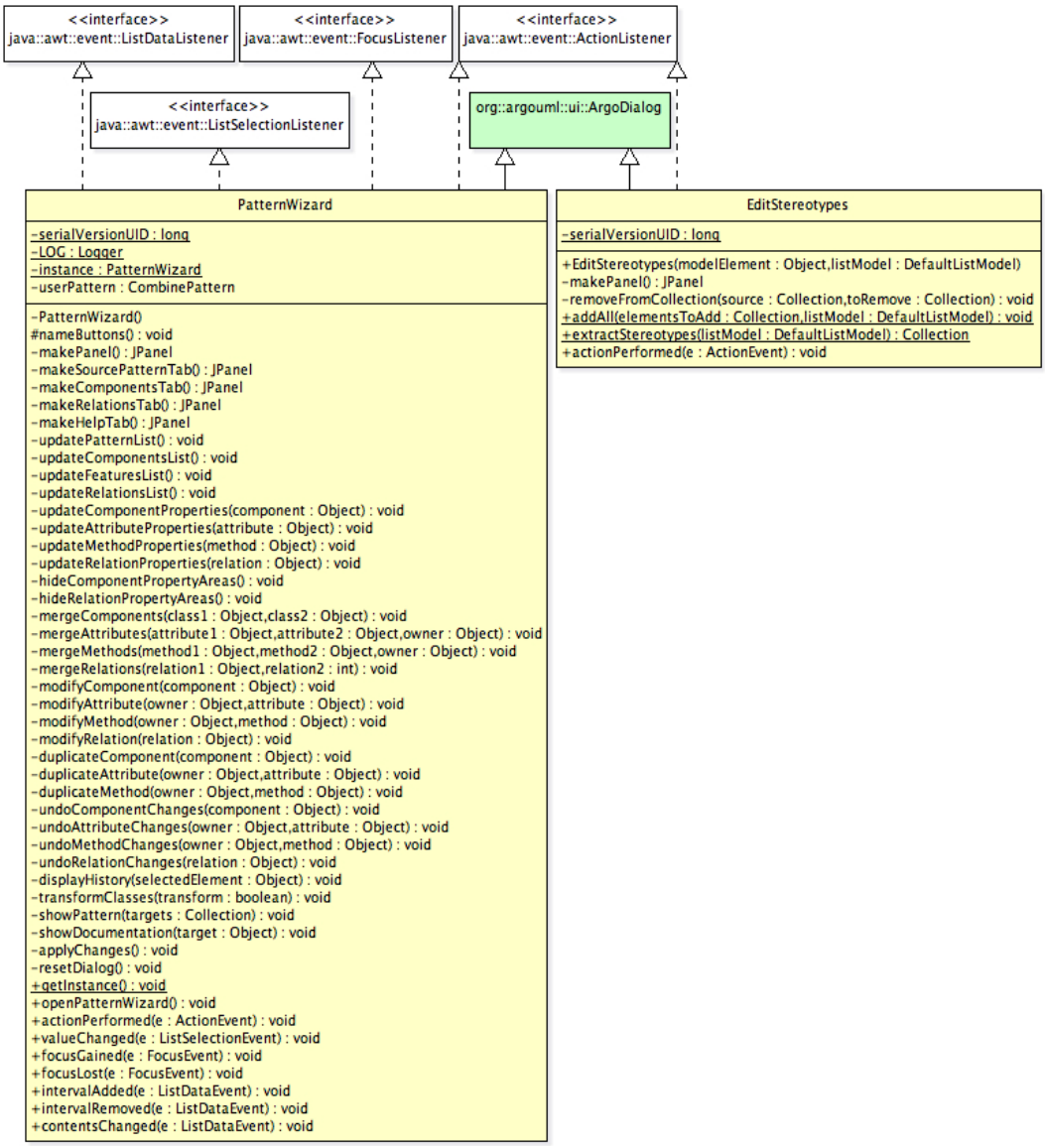


Abbildung 4.3: GUI-Klassen des Pattern-Wizards

Wie Abbildung 4.4 zeigt, ist das Fenster des Pattern-Wizards in drei Bereiche unterteilt. Ganz oben befindet sich der Transformationsbereich, in dem über drei Reiter (Tabs) die Manipulationen an den Klassen vorgenommen werden können. Eine ausführliche Beschreibung der Möglichkeiten in diesen Tabs erfolgt im Abschnitt 4.2. Unter diesem Bereich befindet sich der Hilfebereich in dem unterschiedlich viele Tabs geöffnet sein können. Einer der immer vorhanden ist, ist der Anleitungs-Tab, der mit einer Schritt-für-Schritt-Anleitung die Nutzung des Wizards erklärt. Weitere Tabs können die Informationen über ausgewählte Entwurfsmuster enthalten. Sie

werden über den Button „Zeige Dokumentation“ unterhalb der Liste der verfügbaren Pattern geöffnet. Die dort dargestellten Informationen stammen aus einer Übersicht über Entwurfsmuster[2] des Softwaretechnik Lehrstuhls der Universität Rostock und werden in der Dokumentation der Pattern-Packages gespeichert. Im untersten Bereich des Pattern-Wizards befinden sich drei Buttons. Einen „Anwenden“-Button, um die vorgenommenen Manipulationen in ArgoUML zu übernehmen, einen „Schliessen“-Button, um den Dialog vorübergehend zu schließen (wobei die vorgenommenen Änderungen nicht verloren gehen) und einen „Zurücksetzen“-Button, um alle vorgenommen Änderungen zu verwerfen.

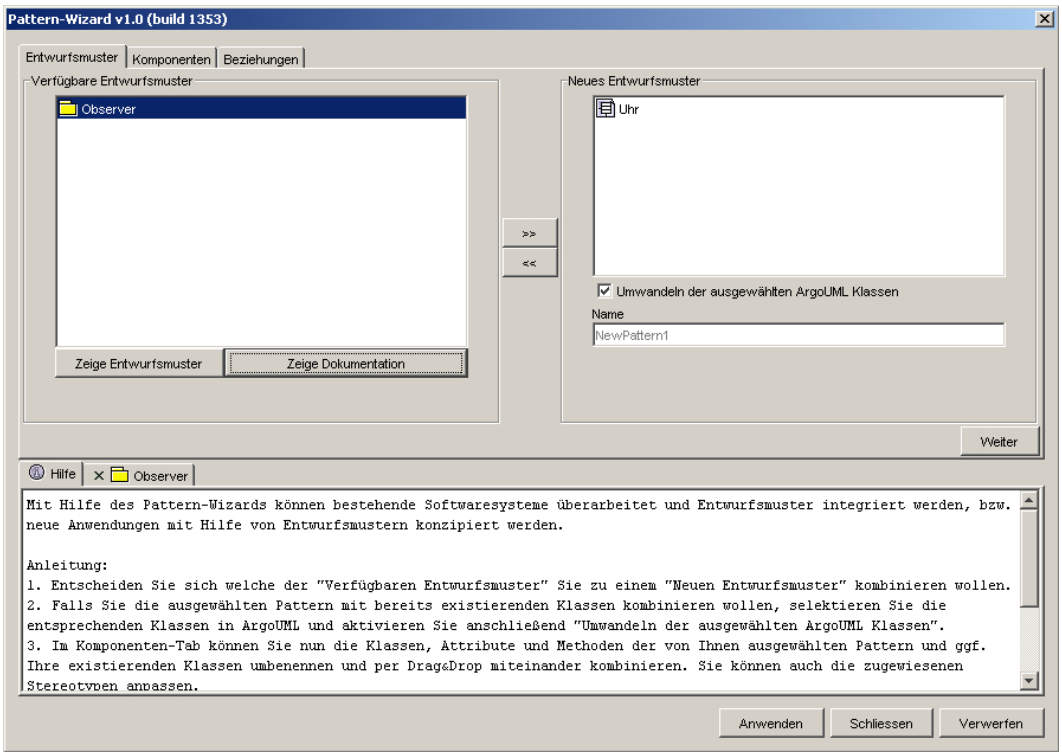


Abbildung 4.4: Benutzeroberfläche des Pattern-Wizards

4.1.5 Konzeption des Datenmodels

Alle Daten, die in der Benutzeroberfläche dargestellt werden, stammen von einem Objekt der Klasse `CombinePattern`. Es speichert alle Daten der an der Kombination beteiligten Modellelemente und sämtliche vom Nutzer vorgenommenen Änderungen an ihnen. Des Weiteren beinhaltet die Klasse die gesamte Logik zum Manipulieren und Kombinieren der Entwurfsmuster und der damit verbundenen Interaktion mit ArgoUML.

[2] `http://wwwswt.informatik.uni-rostock.de/deutsch/Infothek/Entwurfsmuster/patterns`

Um dem Nutzer die Möglichkeit zu geben die von ihm vorgenommen Änderungen wieder rückgängig zu machen, werden die Änderungen nicht sofort in ArgoUML umgesetzt, sondern bis zum Betätigen des „Anwenden“-Buttons gespeichert. Dazu werden, wie in [Sch02] Kapselobjekte für Klassen, Attribute, Methoden und Relationen von ArgoUML und Befehlsobjekte für die Änderungen des Nutzers verwendet. Instanzen der Klassen `...Wrapper` bilden die Kapselobjekte und enthalten jeweils ein unverändertes ArgoUML Modellelement. Diesen Kapselobjekten werden Befehlsobjekte vorgeschaltet falls der Nutzer Modifikationen (`...OperatorModify`) bzw. Kombinationen (`...OperatorMerge`) vornimmt. Der Vorteil bei diesem Vorgehen ist, dass das Rückgängig Machen von Manipulationen des Nutzers ohne Probleme möglich ist. Dazu wird einfach das letzte Befehlsobjekt gelöscht. Sollten keine Befehlsobjekte mehr vorhanden sein, so befindet sich das Modellelemente in seinem Ursprungszustand.

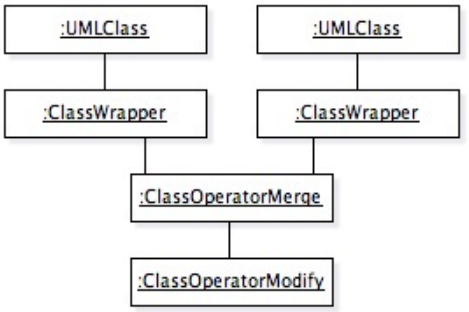


Abbildung 4.5: Kombination von Kapsel- und Befehlsobjekten

Mit Hilfe des Beispiels in Abbildung 4.5 soll die Verwendung der Kapsel- und Befehlsobjekte genauer beschrieben werden. Ausgangspunkt dieses Beispiels sind zwei Klassen (`UMLClass`) eines ArgoUML Klassendiagramms. Diese werden jeweils in einen `ClassWrapper` verpackt um einen einheitlichen Zugriff auf die Eigenschaften des Modellelementes sicherzustellen. Wenn der Nutzer diese beiden Klassen kombiniert, wird den beiden Kapselobjekten ein `ClassOperatorMerge` vorangestellt. Dieses Befehlsobjekt sorgt für die Kombination der Eigenschaften beider Vorgängerobjekte. Wenn der Nutzer anschließend etwa den Namen der entstandenen Klasse ändert, wird ein weiteres Befehlsobjekt, in diesem Fall ein `ClassOperatorModify`, davor geschaltet. Anfragen zum ermitteln des Namens würden dann von diesem Objekt beantwortet und alle anderen Anfragen würden an das dahinter liegende `ClassOperatorMerge`-Objekt weitergeleitet.

4.2 Möglichkeiten des Pattern-Wizards

Die Manipulationsmöglichkeiten des Pattern-Wizard sind im Transformationsbereich auf drei Reiter aufgeteilte. Im Entwurfsmuster-Tab (Abbildung 4.4) werden alle verfügbaren Entwurfsmuster aufgelistet. Zu diesen kann man sich das Klassendiagramm in ArgoUML anzeigen lassen und die Informationen die zu ihnen hinterlegt sind, können im unteren Bereich des Wizards, neben dem Hilfe-Reiter eingeblendet werden. Aus dieser Liste wählt man die Entwurfsmuster aus, die kombiniert werden sollen. Die Auswahl kann per Drag&Drop oder über die Buttons in der Mitte erfolgen. Für das aus der Kombination hervorgehende Entwurfsmuster muss ein neuer Name festgelegt werden. Es ist ebenfalls möglich bereits bestehende Klassen mit Entwurfsmustern zu kombinieren. Dazu muss man das Kontrollfeld „Umwandeln der ausgewählten ArgoUML Klassen“ aktivieren. Dabei werden alle zu dem Zeitpunkt in ArgoUML selektierten Klassen mit ihren Attributen, Methoden und Relationen in den Pattern-Wizard übernommen und können im Weiteren kombiniert und manipuliert werden. In diesem Fall ist es nicht nötig, einen neuen Namen anzugeben, da das Ergebnis in das bestehende Klassenmodel übernommen wird.

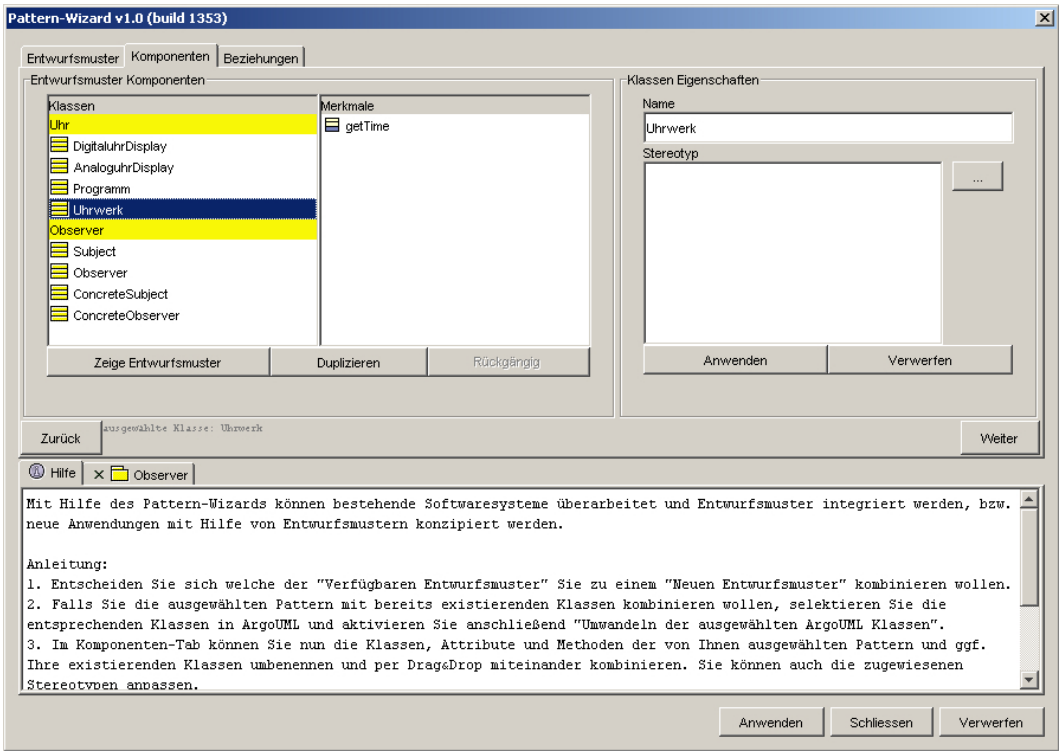


Abbildung 4.6: Komponenten-Tab des PatternWizards

Über den Komponenten-Tab (Abbildung 4.6) können sowohl die Klassen der ausgewählten Entwurfsmuster als auch ihre Attribute und Methoden transformiert werden.

Zu diesem Zweck befinden sich auf der linken Seite zwei List-Boxen. In der linken werden alle Klassen entsprechend ihrer Zugehörigkeit zu den Entwurfsmustern gruppiert angezeigt und in der List-Box daneben werden die Attribute und Methoden der gerade selektierten Klasse dargestellt. In der Klassen-Liste werden, falls die entsprechende Option im Entwurfsmuster-Tab aktiviert wurde, auch die zuvor in ArgoUML ausgewählten Klassen aufgeführt. Per Drag&Drop können die einzelnen Elemente kombiniert werden. Dabei werden die ursprünglichen Elemente aus der Liste entfernt und das kombinierte Element unter „geändert“ eingefügt. Dabei ist zu beachten, dass das Element welches auf ein anderes gezogen wird, den Namen des neuen Elementes bestimmt. Wie im vorhergehenden Reiter ist es auch hier möglich die Klassendiagramme der aufgeführten Klassen in ArgoUML anzuzeigen. Ferner können Komponenten dupliziert werden und bereits vorgenommene Änderungen können wieder rückgängig gemacht werden. Auf der rechten Seite des Komponenten-Tab können der Name und die zugewiesenen Stereotypen der gerade selektierten Klasse bzw. des selektierten Merkmals verändert werden.

Der dritte Reiter des Transformationsbereichs ist der Beziehungen-Tab (Abbildung 4.7). Sein Aufbau ist ähnlich dem des Komponenten-Tabs. Auf der linken Seite werden alle zwischen den Klassen bestehenden Assoziationen, Aggregationen, Kompositionen und Vererbungsbeziehungen in einer List-Box dargestellt. Die Beziehungen die zwischen den selben Klassen bestehen, können per Drag&Drop kombiniert werden. Auf der rechten Seite können wieder die Eigenschaften der Beziehungen verändert werden. Für alle Beziehungen können sowohl der Name als auch die zugewiesenen Stereotypen angepasst werden und bei Assoziationen können zusätzlich die Rollen und Multiplizitäten modifiziert werden. Selbstverständlich ist es auch hier möglich alle vorgenommen Transformationen rückgängig zu machen und die Entwurfsmuster in denen die Beziehungen vorkommen in ArgoUML aufzurufen. Nachdem alle Transformationen vorgenommen wurden, können sie durch den „Anwenden“-Button in ArgoUML übernommen werden. Es ist jedoch zu beachten, dass diese Aktion endgültig ist.

4.3 Beispielszenario

Anhand eines Beispiels soll im folgenden die Nutzung des Pattern-Wizards noch einmal verdeutlicht werden. Dazu wird das leicht verständliche Uhrenbeispiel aus [For01] in einer etwas abgewandelten Form verwendet.

Ausgangspunkt dieses Szenarios ist ein Programm, welches die Uhrzeit in zwei verschiedenen Arten auf dem Bildschirm darstellt. Zum einen als ein Display mit Analoguhr und zum anderen als eins mit Digitaluhr. Abbildung 4.8 zeigt die wichtigsten

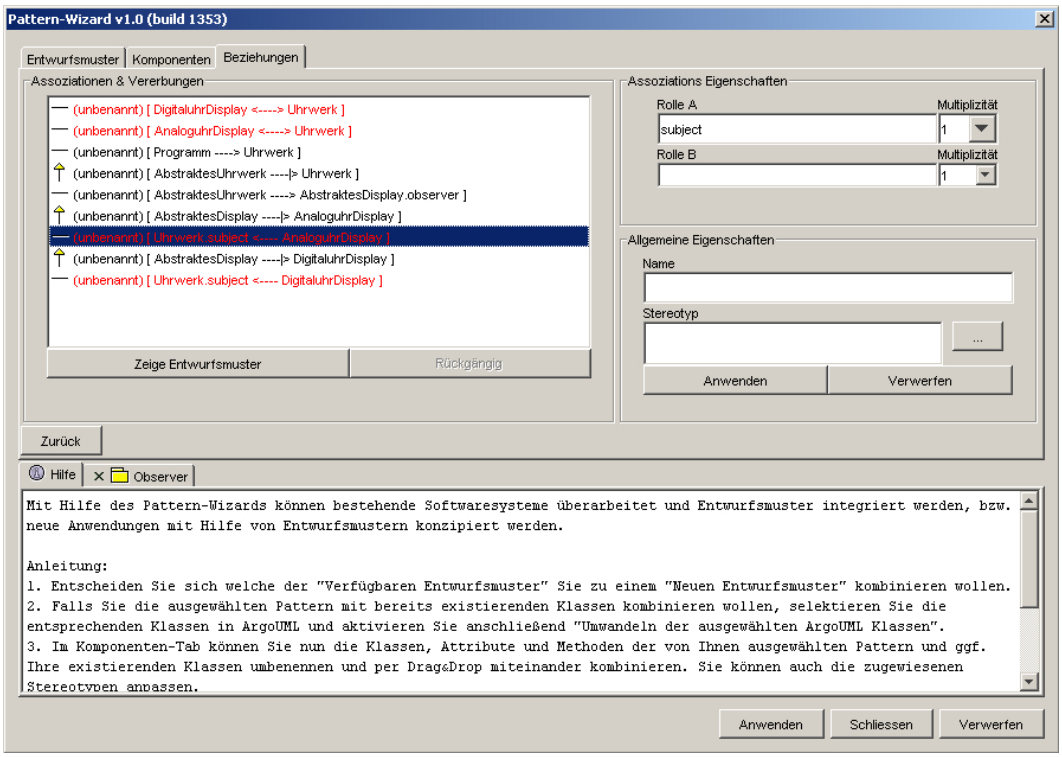


Abbildung 4.7: Beziehungen-Tab des PatternWizards

Klassen dieses Programms. Die Klasse `Uhrwerk` enthält die Logik der Uhr und die Klassen `AnalaguhrDisplay` und `DigitaluhrDisplay` sind für die graphische Darstellung der Uhrzeit verantwortlich. Sobald sich die Uhrzeit in `Uhrwerk` ändert, ruft es die `repaint`-Methode der beiden Displays auf, damit diese die Anzeigen aktualisieren. Zum Abfragen der aktuellen Uhrzeit steht ihnen die Methode `getTime` zur Verfügung.

Das Problem an dieser Umsetzung ist die fehlende Flexibilität. Sobald das Programm um eine weitere Anzeige erweitert werden soll, muss auch die Implementierung von `Uhrwerk` angepasst werden. Man erkennt jedoch leicht, dass der Einsatz des in [GHJV04] beschriebenen Beobachter-Entwurfsmusters dieses Problem beheben

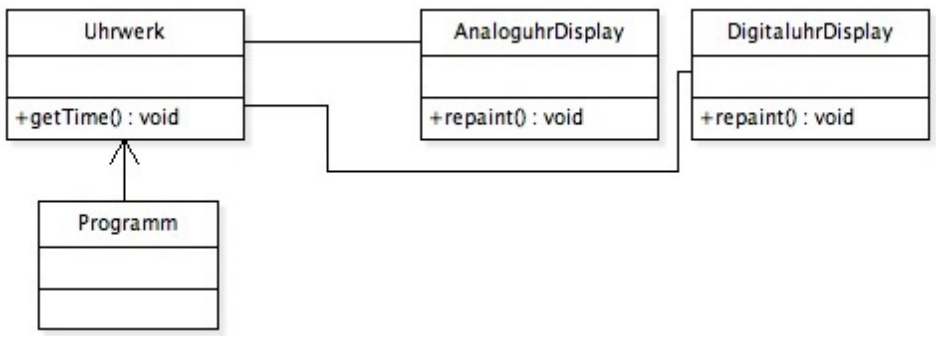


Abbildung 4.8: Wichtige Klassen des Uhrenbeispiels

könnte. Mit Hilfe des Pattern-Wizards ist es auf einfache und schnelle Weise möglich den bestehenden Entwurf mit dem im Abbildung 1.3 dargestellten Observer zu kombinieren. Dazu müssen die Klassen aus Abbildung 4.8 markiert werden. Anschließend öffnet man den Pattern-Wizard und setzt im Entwurfsmuster-Tab den Haken bei „Umwandeln der ausgewählten ArgoUML Klassen“. Nachdem man in der Liste der verfügbaren Entwurfsmuster den Observer ausgewählt hat und ihn per Drag&Drop der Liste auf der rechten Seite hinzugefügt hat, kann man im Komponenten-Tab mit der Kombination der Klassen beginnen. Da das `Uhrwerk` das beobachtete Objekt ist, wird es mit `ConcreteSubject` kombiniert. Dazu wird in der linken List-Box `Uhrwerk` auf `ConcreteSubject` gezogen. Die daraus entstehende Klasse trägt weiterhin den Namen Uhrwerk und besitzt jetzt zwei Methoden, die die selbe Funktion haben und daher ebenfalls per Drag&Drop kombiniert werden. Die beiden Displays des ursprünglichen Entwurfs können als Beobachter des Uhrwerks aufgefasst werden. Das bedeutet, dass sie mit der Klasse `ConcreteObserver` kombiniert werden müssen. Da davon jedoch nur eine im Entwurfsmuster zur Verfügung steht, muss zuerst mit Hilfe des „Duplizieren“-Buttons eine Kopie erzeugt werden. Danach müssen nur noch die im ursprünglichen Entwurf nicht enthaltenen Klassen `Subject` und `Observer` über den Eigenschaftenbereich auf der rechten Seite einen passenderen Namen erhalten. Die erste wird in `AbstraktesUhrwerk` und die andere in `AbstarktesDisplay` umbenannt. Danach wird im Relationen-Tab die Rolle dieser Assoziation von `observer` in `Anzeige` umbenannt. Abschließend werden die Assoziationen zwischen `Uhrwerk` und `AnaloguhrDisplay` respektive `Uhrwerk` und `DigitaluhrDisplay` kombiniert und die Rolle `subject` in `Uhrwerk` umbenannt. Nachdem alle Transformationen vorgenommen wurden und der Pattern-Wizard diese auf das ArgoUML Diagramm angewendet hat, muss ggf. noch ein kleine Korrektur durch den Nutzer vorgenommen werden. Die Methoden `repaint` vom `AnaloguhrDisplay` und `DigitaluhrDisplay` können gelöscht werden, falls die von `AbstraktesDisplay` geerbte bereits implementiert ist.

Das Ergebnis der Kombination des ursprünglichen Entwurfs mit dem Beobachter-Entwurfsmuster ist in Abbildung 4.9 dargestellt. Der neue Entwurf ist nun viel flexibler. Weitere Displays können jetzt ohne Probleme dem Programm hinzugefügt werden. Sie müssen sich nur mit der Methode `Attach` beim `Uhrwerk` anmelden und erhalten dann bei jedem Zeitwechsel von der Methode `Notify` die Botschaft `repaint`.

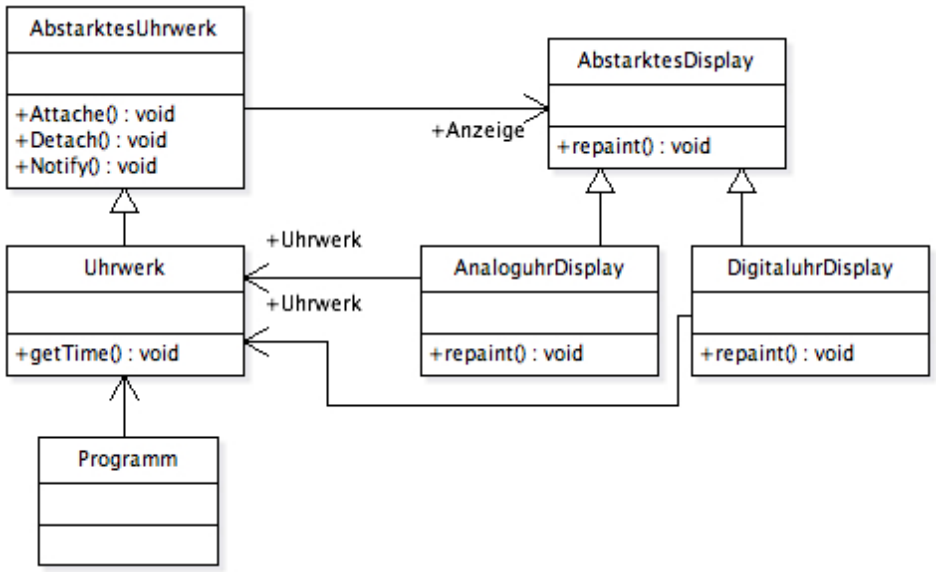


Abbildung 4.9: Uhrenbeispiel in Kombination mit Beobachter-Entwurfsmuster

Kapitel 5

Schlussbemerkungen

5.1 Erweiterungsmöglichkeiten

Wie eingangs erläutert, stellt der ArgoUML Pattern-Wizard eine Weiterentwicklung der Pattern-Erweiterung für Rational Rose aus [Man00] und [Sch02] dar. Ein Großteil der Funktionalität der ursprünglichen Implementierung wurde umgesetzt und mit besonderem Fokus auf objektorientierte Programmierung und eine leichtere Nutzbarkeit erweitert. Im folgenden soll kurz auf Erweiterungsmöglichkeiten eingegangen werden, die aus verschiedenen Gründen noch nicht implementiert wurden, jedoch einen Mehrwert für den Pattern-Wizard darstellen würden und daher in zukünftigen Versionen integriert werden sollten.

5.1.1 Design by Contract

In [Sch02] wurden Klasseninvarianten für die Komponenten eines Entwurfsmusters und Vor- und Nachbedingungen für die Methoden dieser Komponenten genutzt. Durch diese formale Spezifikation ist es leichter möglich Fehler im Model zu entdecken. Aus Zeitgründen war es mir nicht möglich diese Konzepte im Pattern-Wizard umzusetzen. Es wäre jedoch möglich, dazu das in ArgoUML integrierte Dresden OCL Toolkit zu nutzen und aus dem Pattern-Wizard heraus den OCL Editor zu öffnen, um entsprechende „Verträge“ zu definieren.

5.1.2 Pattern-Kardinalität

Die ebenfalls in [Sch02] eingeführte Pattern-Kardinalität zur Definition der erlaubten Anzahl an Vervielfältigungen einer Komponente, eines Attributes bzw. einer Methode ist dazu geeignet, dass Verständnis des Anwenders für ein Entwurfsmuster zu verbessern und einfache Abhängigkeiten zwischen den Elementen zu definieren. Dieses Konzept wurde im Pattern-Wizard noch nicht umgesetzt, da es nicht möglich ist die vorhandenen Model-Elemente um neue Eigenschaften zu erweitern und ich eine Lösung über Stereotypen für nicht praktikabel halte. Dieses Problem könnte jedoch mit der folgenden Erweiterungsmöglichkeit beseitigt werden.

5.1.3 Integration der Entwurfsmuster-Bibliothek

Eine weitere Verbesserungsmöglichkeit wäre die Bibliothek mit den Entwurfsmustern in den Pattern-Wizard zu integrieren, so dass nicht immer das ArgoUML Projekt mit der Entwurfsmuster-Bibliothek geladen sein muss. Der Pattern-Wizard könnte bei Bedarf, wenn z. B. „Zeige Entwurfsmuster“ gewählt wurde, dieses generieren. Die Entwurfsmuster könnten aber auch bei der Kombination genutzt werden, ohne das sie zuvor erzeugt werden müssten. Diese Umsetzung hätte den Vorteil, dass das zuvor erwähnte Problem des Speicherns der Pattern-Kardinalität gelöst wäre. Diese Informationen könnten dann in den Wizard integriert und entsprechend dargestellt werden. Durch die Integration würde sich jedoch ein neues Problem ergeben. Der Anwender könnte nicht mehr wie in der aktuellen Version des Wizards eigene Entwurfsmuster definieren, die dann genutzt werden können. Durch die Möglichkeit eine zusätzliche externe Bibliothek zu nutzen und zur Verwaltung dieser ein Werkzeug bereitzustellen, könnte auch dieses Problem umgangen werden.

5.2 Fazit

Trotz der beschriebenen Erweiterungsmöglichkeiten stellt der Pattern-Wizard in seiner jetzigen Version bereits ein praktisches und leicht zu benutzendes Werkzeug dar, um in ArgoUML mit Entwurfsmustern zu arbeiten. Damit können Softwareentwickler, die nicht die Möglichkeit haben auf kommerzielle CASE-Tools zurückzugreifen bzw. dies aus bestimmten Gründen nicht wollen, nun das frei verfügbare ArgoUML in Verbindung mit dem Pattern-Wizard nutzen. Sie können damit bestehende Softwaresysteme überarbeiten und Entwurfsmuster integrieren oder neue Anwendungen mit Hilfe

von Entwurfsmustern realisieren. Dazu wurde eine Entwurfsmuster-Bibliothek aus den in [GHJV04] vorgestellten Entwurfsmustern zusammengestellt, wobei evt. nicht alle von ihnen spezifisch genug sind, um effektiv mit dem Pattern-Wizard genutzt zu werden. Diese Entscheidung soll jedoch dem Anwender überlassen bleiben. Die Bibliothek kann jederzeit um weitere Muster erweitert werden, so dass es dem Nutzer möglich ist, sich die für seine Bedürfnisse passenden Entwurfsmuster zusammenzustellen. Neben seiner eigentlichen Funktion kann der Wizard auch als eine Art Pattern-Browser verwendet werden. Man kann sich zu den in der Bibliothek vorhandenen Entwurfsmustern die Struktur anzeigen lassen und wird zusätzlich mit Informationen über Funktion und Verwendung versorgt. Der Pattern-Wizard wurde von mir unter der BSD-Lizenz als Projekt auf Tigris.org veröffentlicht, um so eine freie Verfügbarkeit und die stetige Weiterentwicklung des Wizards zu ermöglichen. Das Projekt ist unter `http://argouml-pattern-wizard.tigris.org` erreichbar. Falls der Pattern-Wizard sowohl von den Nutzern als auch von den Entwickler gut angenommen wird, ist auch eine feste Integration in ArgoUML denkbar.

Anhang A

Anhang

A.1 Apache Ant Konfigurationsdatei (build.xml)

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<!-- ============================================================================ -->
<!-- Building the latest versions of this ArgoUML module is automated using      -->
<!-- Apache Ant which is a very handy tool that uses a build file written in     -->
<!-- XML (this one) to describe the build process and its dependencies.          -->
<!--                                                                              -->
<!-- You can use this build file by running Ant from the command line with       -->
<!--     ant [options] [target]                                                   -->
<!-- or from within an Integrated Development Environment like Eclipse.          -->
<!--                                                                              -->
<!-- For more information:                                                        -->
<!--     - about Ant refer to http://ant.apache.org/                              -->
<!--     - about what this build file can do see the target "usage".              -->
<!-- ============================================================================ -->

<project name="argouml-pattern-wizard" default="usage" basedir=".">

  <!-- ========================================================================= -->
  <!-- Import another build-file that provides a task for calculating the        -->
  <!-- buildnumber <years on market> (1 digit) + <day number> (3 digits)         -->
  <!-- ========================================================================= -->
  <import file="buildnumber.xml"/>

  <!-- ========================================================================= -->
  <!-- Initialise the needed properties                                          -->
  <!-- ========================================================================= -->
  <target name="init">
```

```
    <echo message="Initialising the needed properties ..."/>
    <!-- set the root dir for ArgoUML
         (it has to be a absolute path which is used in the *.properties) -->
    <property name="argo.root.dir" value="${basedir}/.."/>
    <!-- import Eclipse specific ArgoUML properties -->
    <property file="${argo.root.dir}/argouml/eclipse-ant-build.properties"/>
    <!-- set global properties for this build -->
    <property file="module.properties"/>
    <!-- set ArgoUML default properties -->
    <property file="${argo.root.dir}/argouml/default.properties"/>
    <!-- set the classpath -->
    <path id="argo.classpath">
      <!-- all jar files in the lib directory -->
      <fileset dir="${argo.build.dir}">
        <include name="*.jar"/>
      </fileset>
    </path>
    <!-- set users ant properties (if present) -->
    <property file="${user.home}/.argo.ant.properties"/>
    <property file="${user.home}/.ant.properties"/>
  </target>

  <!-- ================================================================== -->
  <!-- Help on usage                                                      -->
  <!-- ================================================================== -->
  <target name="usage" depends="init" description="Display usage informations.">
    <echo message="'${module.name}' Build file"/>
    <echo message="------------------------------------------------------------"/>
    <echo message="ant [options] [target]"/>
    <echo message=" "/>
    <echo message="Available targets:"/>
    <echo message=" "/>
    <echo message="    compile --> compiles the source code to the tree"/>
    <echo message="                below the ${module.build.dir} folder"/>
    <echo message="    package --> generates the ${module.jarfile.name}"/>
    <echo message="    run     --> runs ArgoUML with ${module.jarfile.name}"/>
    <echo message="    install --> moves the created ${module.jarfile.name}"/>
    <echo message="                into the ArgoUML ext folder."/>
    <echo message="    usage   --> show this message (default target)"/>
    <echo message=" "/>
    <echo message="------------------------------------------------------------"/>
    <echo message=" See the comments inside the build.xml file for more details"/>
  </target>

  <!-- ================================================================== -->
```

```
<!-- Prepares the build directories                                          -->
<!-- ====================================================================== -->
<target name="prepare" depends="init, clean, generate-buildnumber">
  <echo message="Preparing the build directories ..."/>
  <!-- create build directorie -->
  <mkdir dir="${module.build.dest.dir}"/>
  <!-- copy the sources into the compilation src folder -->
  <copy todir="${module.build.src.dir}">
    <fileset dir="${module.src.dir}"/>
    <filterset>
      <!-- replace tokens in the source files with the corresponding values -->
      <filter token="VERSION" value="${module.version}"/>
      <filter token="BUILD" value="${build.number}"/>
    </filterset>
  </copy>
  <!-- create i18n directorie and copy the localization files -->
  <mkdir dir="${module.build.i18n.dir}"/>
  <copy todir="${module.build.i18n.dir}">
    <fileset dir="${module.src.i18n.dir}"/>
  </copy>
  <!-- create image directorie and copy all image files-->
  <mkdir dir="${module.build.image.dir}"/>
  <copy todir="${module.build.image.dir}">
    <fileset dir="${module.src.image.dir}"/>
  </copy>
</target>

<!-- ====================================================================== -->
<!-- Compiles the source directory                                          -->
<!-- ====================================================================== -->
<target name="compile" depends="prepare" description="Compile the sources.">
  <echo message="Compiling the sources ..."/>
  <!-- Compile the source code -->
  <javac srcdir="${module.build.src.dir}" destdir="${module.build.dest.dir}"
         optimize="${optimize}" verbose="${verbose}" debug="${debug}"
         deprecation="${deprecation}" excludes="${excluded.files}"
         source="${source.level}" target="${target.level}">
    <classpath>
      <path refid="argo.classpath"/>
    </classpath>
  </javac>
</target>

<!-- ====================================================================== -->
<!-- Creates the jar file                                                   -->
```

```
<!-- ================================================================== -->
<target name="jar" depends="compile" description="Create the module jar file.">
  <echo message="Creating the module jar file ..."/>
  <!-- put everything in ${module.build.dest} into the ${module.jarfile} -->
  <jar jarfile="${module.jarfile}" basedir="${module.build.dest.dir}"
       includes="${included.files}" excludes="${excluded.files}">
    <!-- create the manifest.mf -->
    <manifest>
      <attribute name="Class-Path" value="argouml.jar"/>
      <section name="${main.class.name}">
        <attribute name="Extension-name" value="module.${module.name}"/>
        <attribute name="Specification-Title" value="ArgoUML Dynamic Load Module"/>
        <attribute name="Specification-Version" value="${module.version}"/>
        <attribute name="Specification-Vendor" value="University of California"/>
        <attribute name="Implementation-Title" value="${module.name} - ArgoUML Module"/>
        <attribute name="Implementation-Version" value="${module.version}"/>
        <attribute name="Implementation-Vendor" value="Rene Lindhorst"/>
      </section>
    </manifest>
  </jar>
</target>

<!-- ================================================================== -->
<!-- Move the created jar file into the ArgoUML extension directory     -->
<!-- ================================================================== -->
<target name="install" depends="jar"
        description="Move the created jar into the ArgoUML ext directory.">
  <echo message="Installing the module ..."/>
  <!-- create argo ext directorie -->
  <mkdir dir="${argo.build.ext.dir}"/>
  <!-- copy the module into the ext dir -->
  <copy file="${module.jarfile}" todir="${argo.build.ext.dir}"/>
</target>

<!-- ================================================================== -->
<!-- Run ArgoUML with the created module                                -->
<!-- ================================================================== -->
<target name="run" depends="install"
        description="Run ArgoUML with the created module.">
  <echo message="Executing ArgoUML with the module ..."/>
  <!-- run ArgoUML -->
  <java jar="${argo.jarfile}" dir="${argo.build.dir}" fork="yes"/>
</target>

<!-- ================================================================== -->
```

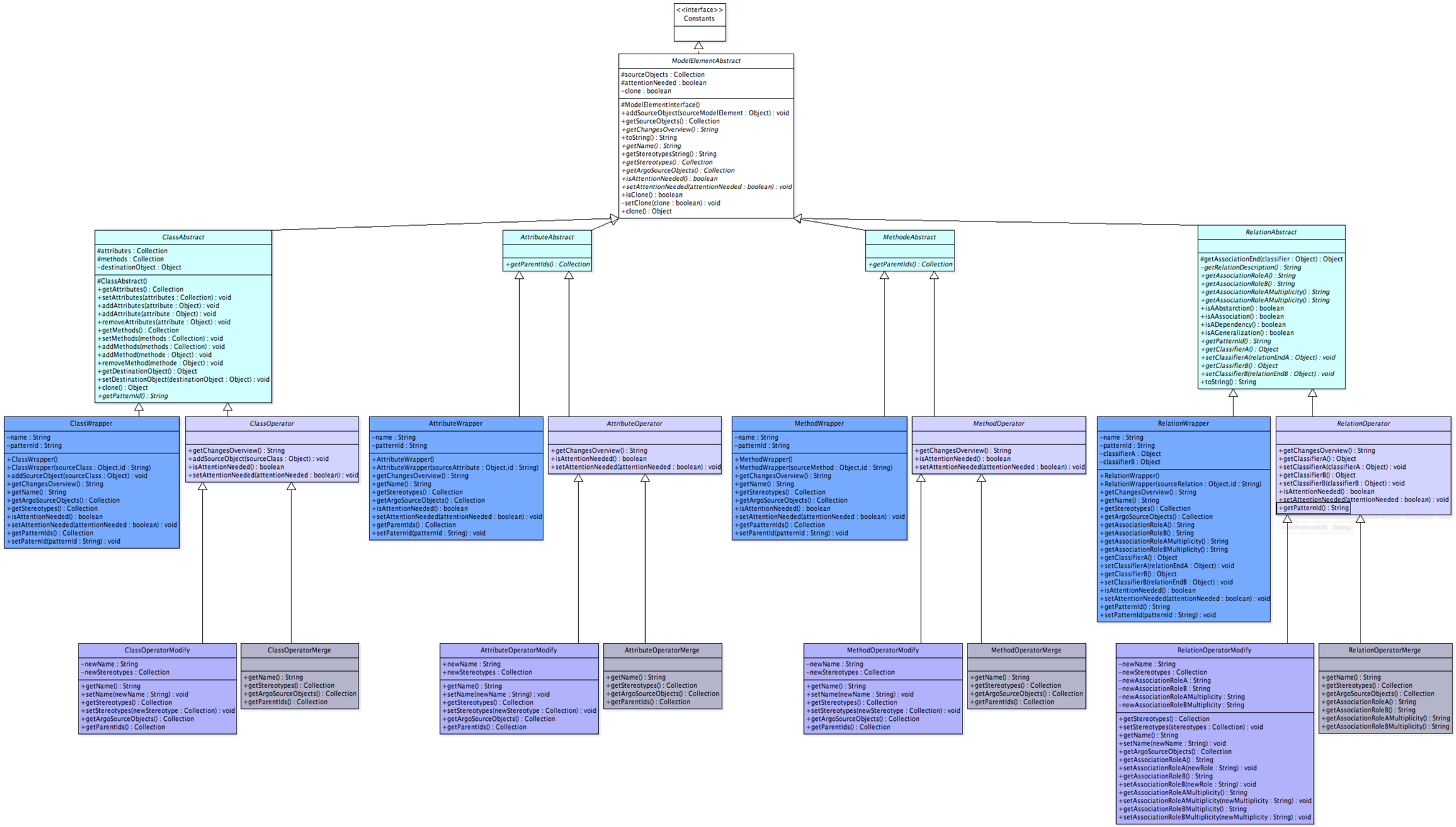
```
    <!-- Run ArgoUML with the created module in debug mode                    -->
    <!-- ================================================================== -->
    <target name="debug" depends="install"
            description="Run ArgoUML with the created module in debug mode.">
      <echo message="Executing ArgoUML with the module in debug mode ..."/>
      <echo message="+ ${module.name} v${module.version} (build ${build.number})"/>
      <!-- run ArgoUML wit logging activated-->
      <java jar="${argo.jarfile}" dir="${argo.build.dir}" fork="yes">
        <sysproperty key="log4j.configuration"
                             value="org/argouml/resource/full_console.lcf"/>
      </java>
    </target>

    <!-- ================================================================== -->
    <!-- Clean targets                                                      -->
    <!-- ================================================================== -->
    <target name="clean" depends="init"
            description="Remove files generated by any of the other targets.">
      <echo message="Clean up ..."/>
      <!-- Delete the .jar file -->
      <delete file="${argo.build.dir}/ext/${module.jarfile.name}"/>
      <!-- Delete the ${module.build.dir} directory tree -->
      <delete dir="${module.build.dir}"/>
    </target>

    <!-- ================================================================== -->
    <!-- Generate the code documentation (API)                              -->
    <!-- ================================================================== -->
    <target name="javadoc" depends="init"
            description="Generate the code documentation (API).">
      <echo message="Generating the Javadoc ..."/>
      <mkdir dir="${module.javadoc.dir}"/>
      <javadoc doctitle="${module.name} API Documentation" use="true"
               private="true" windowtitle="${module.name} ${module.version}"
               author="true" header="" bottom="© ${copyright}" splitindex="true"
                   group="${module.name} 'org.argouml.*'" packagenames="org.argouml.*"
                   sourcepath="${module.src.dir}" destdir="${module.javadoc.dir}"
                       classpathref="argo.classpath"/>
    </target>

  </project>
  <!-- End of file -->
```

A.2 Klassendiagramm des Datenmodels


<<interface>>
Constants
ModelElementAbstract
#sourceObjects : Collection
#attentionNeeded : boolean
-clone : boolean
#ModelElementInterface()
+addSourceObject(sourceModelElement : Object) : void
+getSourceObjects() : Collection
+getChangesOverview() : String
+toString() : String
+getName() : String
+getStereotypesString() : String
+getStereotypes() : Collection
+getArgoSourceObjects() : Collection
+isAttentionNeeded() : boolean
+setAttentionNeeded(attentionNeeded : boolean) : void
+isClone() : boolean
-setClone(clone : boolean) : void
+clone() : Object
ClassAbstract
#attributes : Collection
#methods : Collection
-destinationObject : Object
#ClassAbstract()
+getAttributes() : Collection
+setAttributes(attributes : Collection) : void
+addAttributes(attribute : Object) : void
+addAttribute(attribute : Object) : void
+removeAttributes(attribute : Object) : void
+getMethods() : Collection
+setMethods(methods : Collection) : void
+addMethods(methods : Collection) : void
+addMethod(methode : Object) : void
+removeMethod(methode : Object) : void
+getDestinationObject() : Object
+setDestinationObject(destinationObject : Object) : void
+clone() : Object
+getPatternId() : String
AttributeAbstract
+getParentIds() : Collection
MethodeAbstract
+getParentIds() : Collection
RelationAbstract
#getAssociationEnd(classifier : Object) : Object
-getRelationDescription() : String
+getAssociationRoleA() : String
+getAssociationRoleB() : String
+getAssociationRoleAMultiplicity() : String
+getAssociationRoleAMultiplicity() : String
+isAAbstarction() : boolean
+isAAssociation() : boolean
+isADependency() : boolean
+isAGeneralization() : boolean
+getPatternId() : String
+getClassifierA() : Object
+setClassifierA(relationEndA : Object) : void
+getClassifierB() : Object
+setClassifierB(relationEndB : Object) : void
+toString() : String
ClassWrapper
-name : String
-patternId : String
+ClassWrapper()
+ClassWrapper(sourceClass : Object,id : String)
+addSourceObject(sourceClass : Object) : void
+getChangesOverview() : String
+getName() : String
+getArgoSourceObjects() : Collection
+getStereotypes() : Collection
+isAttentionNeeded() : boolean
+setAttentionNeeded(attentionNeeded : boolean) : void
+getPatternIds() : Collection
+setPatternId(patternId : String) : void
ClassOperator
+getChangesOverview() : String
+addSourceObject(sourceClass : Object) : void
+isAttentionNeeded() : boolean
+setAttentionNeeded(attentionNeeded : boolean) : void
AttributeWrapper
-name : String
-patternId : String
+AttributeWrapper()
+AttributeWrapper(sourceAttribute : Object,id : String)
+getChangesOverview() : String
+getName() : String
+getStereotypes() : Collection
+getArgoSourceObjects() : Collection
+isAttentionNeeded() : boolean
+setAttentionNeeded(attentionNeeded : boolean) : void
+getParentIds() : Collection
+setPatternId(patternId : String) : void
AttributeOperator
+getChangesOverview() : String
+isAttentionNeeded() : boolean
+setAttentionNeeded(attentionNeeded : boolean) : void
MethodWrapper
-name : String
-patternId : String
+MethodWrapper()
+MethodWrapper(sourceMethod : Object,id : String)
+getChangesOverview() : String
+getName() : String
+getStereotypes() : Collection
+getArgoSourceObjects() : Collection
+isAttentionNeeded() : boolean
+setAttentionNeeded(attentionNeeded : boolean) : void
+getPaatternIds() : Collection
+setParentId(patternId : String) : void
MethodOperator
+getChangesOverview() : String
+isAttentionNeeded() : boolean
+setAttentionNeeded(attentionNeeded : boolean) : void
RelationWrapper
-name : String
-patternId : String
-classifierA : Object
-classifierB : Object
+RelationWrapper()
+RelationWrapper(sourceRelation : Object,id : String)
+getChangesOverview() : String
+getName() : String
+getStereotypes() : Collection
+getArgoSourceObjects() : Collection
+getAssociationRoleA() : String
+getAssociationRoleB() : String
+getAssociationRoleAMultiplicity() : String
+getAssociationRoleBMultiplicity() : String
+getClassifierA() : Object
+setClassifierA(relationEndA : Object) : void
+getClassifierB() : Object
+setClassifierB(relationEndB : Object) : void
+isAttentionNeeded() : boolean
+setAttentionNeeded(attentionNeeded : boolean) : void
+getPatternId() : String
+setPatternId(patternId : String) : void
RelationOperator
+getChangesOverview() : String
+getClassifierA() : Object
+setClassifierA(classifierA : Object) : void
+getClassifierB() : Object
+setClassifierB(classifierB : Object) : void
+isAttentionNeeded() : boolean
+setAttentionNeeded(attentionNeeded : boolean) : void
+getPatternId() : String
ClassOperatorModify
-newName : String
-newStereotypes : Collection
+getName() : String
+setName(newName : String) : void
+getStereotypes() : Collection
+setStereotypes(newStereotype : Collection) : void
+getArgoSourceObjects() : Collection
+getParentIds() : Collection
ClassOperatorMerge
+getName() : String
+getStereotypes() : Collection
+getArgoSourceObjects() : Collection
+getParentIds() : Collection
AttributeOperatorModify
+newName : String
+newStereotypes : Collection
+getName() : String
+setName(newName : String) : void
+getStereotypes() : Collection
+setStereotypes(newStereotype : Collection) : void
+getArgoSourceObjects() : Collection
+getParentIds() : Collection
AttributeOperatorMerge
+getName() : String
+getStereotypes() : Collection
+getArgoSourceObjects() : Collection
+getParentIds() : Collection
MethodOperatorModify
-newName : String
-newStereotypes : Collection
+getName() : String
+setName(newName : String) : void
+getStereotypes() : Collection
+setStereotypes(newStereotype : Collection) : void
+getArgoSourceObjects() : Collection
+getParentIds() : Collection
MethodOperatorMerge
+getName() : String
+getStereotypes() : Collection
+getArgoSourceObjects() : Collection
+getParentIds() : Collection
RelationOperatorModify
-newName : String
-newStereotypes : Collection
-newAssociationRoleA : String
-newAssociationRoleB : String
-newAssociationRoleAMultiplicity : String
-newAssociationRoleBMultiplicity : String
+getStereotypes() : Collection
+setStereotypes(stereotypes : Collection) : void
+getName() : String
+setName(newName : String) : void
+getArgoSourceObjects() : Collection
+getAssociationRoleA() : String
+setAssociationRoleA(newRole : String) : void
+getAssociationRoleB() : String
+setAssociationRoleB(newRole : String) : void
+getAssociationRoleAMultiplicity() : String
+setAssociationRoleAMultiplicity(newMultiplicity : String) : void
+getAssociationRoleBMultiplicity() : String
+setAssociationRoleBMultiplicity(newMultiplicity : String) : void
RelationOperatorMerge
+getName() : String
+getStereotypes() : Collection
+getArgoSourceObjects() : Collection
+getAssociationRoleA() : String
+getAssociationRoleB() : String
+getAssociationRoleAMultiplicity() : String
+getAssociationRoleBMultiplicity() : String

Literaturverzeichnis

[Bal05] Balzert, Heide: *UML 2 kompakt.* 2. Auflage. Spektrum Akademischer Verlag, 2005

[BMR+98] Buschmann, Frank ; Meunier, Regine ; Rohnert, Hans ; Sommerlad, Peter ; Stal, Michael: *Pattern-orientierte Software-Architektur. Ein Pattern-System.* 2. Auflage. Addison-Wesley, 1998

[Edl02] Edlich, Stefan: *Ant - kurz & gut.* Köln : O'Reilly Verlag, 2002

[For01] Forbrig, Peter: *Objektorientierte Softwareentwicklung mit UML.* Fachbuchverlag Leipzig, 2001

[GHJV04] Gamm, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwufsmuster - Elemente wiederverwendbarer objektorientierter Software.* Addison Wesley Verlag, 2004

[Hil] Hillside Group: *Home of Patterns Library.* `http://hillside.net/patterns/`, Abruf: 26.11.2006

[Man00] Mannhaupt, Danko: *Integration of Design Patterns into Object-Oriented Design using Rational Rose*, Universität Rostock, Diplomarbeit, 2000

[Mat05] Matzke, Bernd: *Ant - Eine praktische Einführung in das Java-Build-Tool.* 2., überarbeitete und aktualisierte Auflage. Heidelberg : Dpunkt Verlag, 2005

[Mor] Morris, Tom: *Frequently asked questions for ArgoUML.* `http://argouml.tigris.org/faqs/users.html`, Abruf: 22.11.2006

[Rob99] Robbins, Jason E.: *Cognitive Support Features for Software Development Tools*, University of California, Irvine, Diss., 1999. `http://argouml.tigris.org/docs/robbins_dissertation/index.html`

[Sch02] Schmacka, Björn Rik: *Integration von Design Patterns in bestehende Softwarespezifikationen*, Universität Rostock, Diplomarbeit, 2002

[TK06] Tolke, Linus ; Klink, Markus: *Cookbook for Developers of ArgoUML: An introduction to Developing ArgoUML.* 0.22, September 2006. http://argouml-stats.tigris.org/documentation/pdf/cookbook/cookbook.pdf

[WTB+06] Wulp, Michiel van d. ; Tolke, Linus ; Bennett, Jeremy ; Odutola, Kunle ; Rueckert, Andreas ; Vanpeperstraete, Philippe ; Ramirez, Alejandro: *ArgoUML User Manual: A tutorial and reference description.* 0.22, September 2006. http://argouml-stats.tigris.org/documentation/pdf/manual/argomanual.pdf

[WTOO06] Wulp, Michiel van d. ; Tolke, Linus ; Oguntimehin, Anthony ; Odutola, Kunle: *ArgoUML Quick Guide: Get started with ArgoUML 0.22.* 0.22, September 2006. http://argouml-stats.tigris.org/documentation/pdf/quick-guide/quickguide.pdf

[ZGK04] Zuser, Wolfgang ; Grechenig, Thomas ; Köhle, Monika: *Software Engineering. Mit UML und dem Unified Process.* 2., überarbeitete Auflage. Pearson Studium, 2004