MTFLOW - Ein System für workflow-basierte Ausführung von Modelltransformationen

Universität Kaiserslautern
Fachbereich Informatik
AG Datenbanken und Informationssysteme
Prof. Dr. Dr. h.c. Theo Härder

MTFLOW - Ein System für workflow-basierte Ausführung von Modelltransformationen

Diplomarbeit

von

Mohamed Amine Chatti

Betreuer:

univ. dipl. inz. Jernej Kovse

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbständig und unter ausschließlicher Verwendung der angegebenen Literatur angefertigt habe.

Kaiserslautern, den 30.04.2004

Mohamed Amine Chatti

Inhaltsverzeichnis

Kapitel 1 Einleitung

Es sind heutzutage neue und größere Herausforderungen der Softwaretechnik anzusehen. Die Software werden immer komplexer und aufwendiger. Sie sollen nämlich mehr Merkmale und Anforderungen erfüllen, neue Bereiche abdecken, schwierige Aufgaben lösen und eine sehr hohe Qualität erreichen. Die sogenannte generative Programmierung hat Ziel, die Entwicklung von Software-Systemen zu vereinfachen, verbessern und beschleunigen, die immer komplexer werdenden Aufgaben und Systeme mit geringmöglichen Aufwand beherrschbar zu machen und automatisch ganze Software-Systemfamilien produzieren zu können. Software-Systemfamilien sind Softwaregruppen, deren Einzelsysteme aus einer gemeinsamen Menge von Komponenten bestehen. Kennzeichnend für eine Software-Systemfamilie ist die gemeinsame Software-Architektur, die allen Systemen zu Grunde liegt. Mit der Generativen Programmierung ergibt sich die Möglichkeit, rein durch eine genaue Spezifikation Software-Systeme automatisch ohne großen Aufwand generieren zu lassen. Der Hersteller eines Software-Systems kann seine Produkte als eine Software-Systemfamilie realisieren, anstelle jeden Kundenauftrag von Grund auf neu zu implementieren. Der Kunde hat dabei die Möglichkeit, seine Anforderungen in Form einer Konfiguration dem Hersteller mitzuteilen, der dann ein speziell auf den Bedarf zugeschnittenes System erstellt. Die Frage ist nun, in welcher Form die Bestellung erfolgen soll. Die Konfiguration könnte durch domänenspezifische Konfigurationssprachen (DSLs), in denen die typischen Merkmale der Anwendungsdomäne enthalten sind, unterstützt werden. Die Syntax und Semantik einer DSL sind allerdings meist schwer zu erlernen. Zudem ist ein DSL-Programm sehr fehleranfällig. Es ist nämlich nicht immer sicher, dass das DSL-Programm eine semantisch korrekte Spezifikation des Systems darstellt. Aus diesem Grund soll dem Kunden eine Möglichkeit zur Verfügung gestellt werden, damit er seine Anforderung leicht formulieren und sein erwünschtes System korrekt spezifizieren kann.

In dem mit der generativen Programmierung verwandten Ansatz der Modellgetriebenen Architekturen (MDA) spielen Modelle und Modelltransformationen eine große Rolle. Der Grundgedanke der MDA besteht darin, dass die Definition der Funktionalität des Systems von der Definition der Implementierung dieser Funktionalität für eine bestimmte Plattform getrennt wird. MDA sieht vor, dass zunächst ein System modelliert werden soll und erst dann wird das Modell durch einen Generator auf einer ausgewählten Plattform implementiert. Die schrittweise Spezifikation eines Systems kann über Modelltransformationen erfolgen, d. h. ein Modell soll auf ein anderes Modell abgebildet werden. Die XML Metadata Interchange (XMI) enthält Mechanismen, die es ermöglichen, die Differenzen zwischen zwei Modelle zu beschreiben. Diese Technik kann genutzt werden, um Transformationen eines Modells durch eine Sequenz von primitiven Operationen auf diesem Modell zu beschreiben.

Im Rahmen dieser Diplomarbeit wird der Ansatz der workflow-basierte Modelltransformationen analysiert und empirisch bewertet. dafür wird ein Werkzeug implementiert, um die resultierenden Ergebnisse zu überprüfen. *MTFLOW* (Model Transformation workFLOWs) bietet dem Entwickler eine Umgebung, in der alle Schritte, die der Spezifikation eines Software-Systems dienen, vorgenommen werden können. *MTFLOW* steht in enger Beziehung zu der generativen Programmierung und dem Ansatz der MDA und soll die einfache und schnelle Spezifikation großer Software-Systeme unterstützen. Diese Spezifikation erfolgt über aufeinanderfolgende Modelltransformationen, bei denen XMI als Technik zum Einsatz kommt. Diese Transformationen werden durch ein Workflow-Modell kontrolliert. Dabei spielt *MTFLOW* die Rolle einer Workflow-Engine, die ein Workflow-Modell instantiiert und die Prozess-Instanzen kontrolliert. *MTFLOW* spielt auch die Rolle eines Konfigurators, der dem Entwickler dabei helfen soll, ohne großen Aufwand sein erwünschtes System schrittweise und korrekt zu spezifizieren.

Im folgenden Kapitel 2 werden zunächst einige Grundlagen, Verfahren und Technologien vorgestellt, auf denen diese Arbeit aufbaut. Dabei handelt es sich einerseits um den Produktlinienansatz bzw. die Theorie, die dahinter steckt, nämlich die generative Programmierung und den verwandten Ansatz der Modellgetriebenen Architekturen (MDA). Andererseits werden die Modellierungssprachen, die in dieser Arbeit zur Spezifikation angewendet werden, vorgestellt. Das sind die Unified Modeling Language (UML) und die Object Action Language (OAL).

Als Beispiel für eine Software-Produktlinie werden in Kapitel 3 objektorientierte Versionierungssysteme vorgestellt. Versionierungssysteme sind gut geeignet, um sie im Rahmen einer Produktlinie zu entwickeln, da die Anwendungsdomäne viele Variationsmöglichkeiten bietet. In diesem Kapitel wird die Domäne der Versionierungssysteme präsentiert und die dabei definierten Operationen mit der Aktionssprache OAL dargestellt.

In Kapitel 4 werden zunächst die verschiedenen Ansätze der Modelltransformationen vorgestellt. Dann wird die XML Metadata Interchange (XMI) als Beschreibungssprache für Transformationen näher betrachtet. Abschließend wird die Java-basierte Template Engine von Velocity, die bei der Erstellung der Vorlagen eingesetzt wurde, präsentiert.

Im zentralen Kapitel 5 wird das im Rahmen dieser Arbeit entwickelte Werkzeug näher betrachtet. Dabei werden zunächst ein Workflow-Metamodell für *MTFLOW*-Workflow-Modelle und ein Workflow-Modell zum Spezifizieren von Versionierungssystemen definiert. Anschließend wird auf den Aufbau, das Funktionsprinzip und die besondere Eigenschaften von *MTFLOW* eingegangen.

In den nachfolgenden Kapitel 6 werden zunächst einige Metriken vorgestellt, mit denen anschließend die entwickelten Vorlagen untersucht und die Ergebnisse der automatisierten Modelltransformationen analysiert werden. Außerdem wird untersucht, welcher Zusammenhang einerseits zwischen der Anzahl der generierten XMI-Elemente und der Komplexität der Vorlagen und andererseits zwischen dieser Anzahl und dem Aufwand der Spezifikation eines Software-Systems besteht.

Abschließend werden in Kapitel 7 die wichtigsten Ergebnisse dieser Arbeit zusammengefasst und ein Ausblick auf weitere Forschungsthemen gegeben.

Kapitel 2 Modellgetriebene Architekturen

Die Techniken, Methoden, Werkzeuge und die vorherrschenden Verfahren, die in der konventionellen Software Engineering eingesetzt werden, sind hauptsächlich auf die Entwicklung einzelner Systeme ausgerichtet. Das Ziel in den letzten Jahren war eine Lösung zu finden, die nicht mehr nur der Entwicklung einzelner Systeme, sondern ganzer Familien von Softwaresystemen dient. Im Mittelpunkt der Lösung steht der Ansatz der sogenannten Generativen Programmierung, deren Grundgedanke die Bildung von Systemfamilien und Produktlinien ist, als notwendige Ergänzung der objektorientierten Methoden. Mit Wiederverwendung als Ausgangspunkt zielt dieses Konzept auf einen höheren Grad an Automatisierung und Qualität bei der Software-Produktion. Die Generative Programmierung bietet also eine methodisch fundierte Grundlage zur Entwicklung von Software-Systemfamilien und -Produktlinien. In ihrem Mittelpunkt steht das generative Domänenmodell mit Problemraum, Konfigurationswissen und Lösungsraum. Die Begriffe des Problemraums und ihre Merkmale werden als eine oder mehrere domänenspezifische Sprachen (Domain Specific Languages - DSLs) implementiert, das Konfigurationswissen in Form eines oder mehrerer kooperierender Generatoren und der Lösungsraum als parametrisierte Bausteine, die gemäß einer flexiblen Systemfamilienarchitektur zu einem fertigen System aufgebaut werden.

In diesem Kapitel werden zunächst einige Grundlagen, Verfahren und Technologien vorgestellt, auf denen diese Arbeit aufbaut. Das sind einerseits verschiedene Ansätze zur Wiederverwendung von Softwareprodukten, inbesondere der Produktlinienansatz bzw. die Theorie, die dahinter steckt, nämlich die generative Programmierung und der verwandte Ansatz der modellgetriebenen Architekturen (Model Driven Architecture, MDA). Andererseits werden die domänenspezifischen Sprachen, die in unserem System angewendet werden, vorgestellt. Es handelt sich dabei um die Unified Modeling Language (UML) und die Object Action Language (OAL).

2.1 Software-Produktlinien

Die physikalischen Produkte kommen in vielen Varianten vor, z. B. Motoren mit unterschiedlichen Zylinderanzahl, unterschiedlichen Einspritzsystemen, unterschiedlichen Kühlungssystemen etc. Die eingebettete Software kommt ebenfalls in unterschiedlichen Varianten vor. Auch sollen die unterschiedlichen Softwarevarianten auf einem Prinzip beruhen und eine Produktvariante soll eine für sie speziell konfigurierte Software beinhalten. Aber obwohl häufig verschiedene Varianten benötigt werden, werden Softwaresysteme überwiegend als Einzelsysteme entwickelt. Unter

der Bezeichnung Software-Produktlinien Engineering werden Ansätze zusammengefasst, die von vornherein die Variantenbildung von Softwaresystemen berücksichtigen.

2.1.1 Analogie Softwareentwicklung-Industrie

Die Erwartungen an die moderne Software sind hoch. Sie soll die immer größer werdende Komplexität meistern und gleichzeitig sowohl eine höhere Produktivität als auch eine bessere Qualität gewährleisten. Diese Erwartungen wurden vor über 100 Jahren bereits an die produzierende Industrie gestellt und im Laufe der Industrialisierung schrittweise realisiert. Abbildung 2.1 zeigt diese industrielle Revolution.

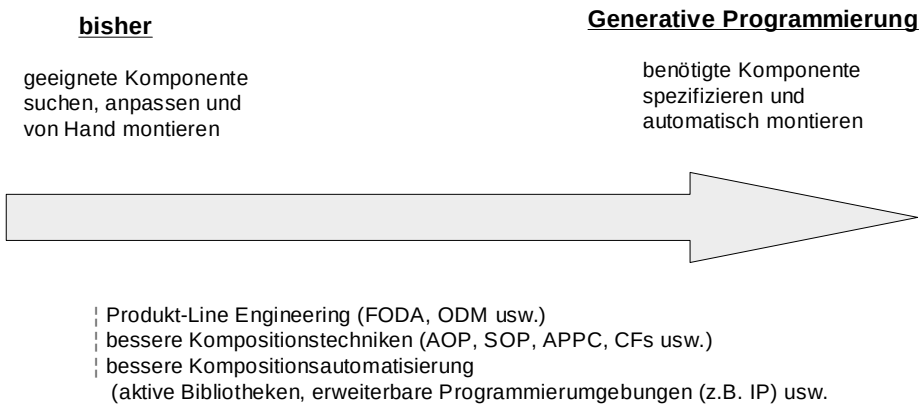
Abbildung 2.1 Industrielle Revolution

So gelang es 1826 John Hall nach jahrelangen erfolglosen Versuchen als Erstem das bereits 1800 aufgekommene Konzept austauschbarer Teile umzusetzen. Er verwirklichte dieses Konzept in der amerikanischen Rüstungsindustrie bei der Produktion von Gewehren. Es dauerte jedoch zwei weitere Jahrzente bis sich dieses Konzept weiter in der amerikanischen Rüstungsindustrie verbreitet hatte. Um 1885 bauten Karl Benz und fast zeitgleich Gottlieb Daimler die ersten beiden Automobile. Jedoch war der Herstellungsprozess von Automobilen langsam und somit das Endprodukt unerschwinglich für die Allgemeinheit. 1901 änderte sich diese Situation als Ransom Olds das erste Fließband einführte, das 1913 von Henry Ford verbessert wurde und dann seine Verbreitung fand. Erst in der zweiten Hälfte des letzten Jahrhunderts war die Geburtsstunde der für die Industrie wohl wichtigsten Technologie, der automatisierten Fließbänder. Zwar wurde der weltweit erste Industrieroboter zur automatisierten Produktion bereits 1961 bei General Motors installiert, jedoch ließ ein effizienter Einsatz noch 20 Jahren auf sich warten. So führten erst 1981 die Erforschung und Weiterentwicklung des Mikrochips zum erfolgreichen Einsatz von Robotern.

Abbildung 2.2 zeigt den aktuellen Stand der Softwareproduktion und die neue angebotene Lösung, die den Rückstand, den die Softwareentwicklung gegenüber der produzierten Industrie hat, aufholen soll. Es lassen sich ähnlich wichtige Meilensteine in der Softwareentwicklung ausmachen. Die erste “richtige“ Programmiersprache entwickelte Konrad Zuse zwischen 1942 und 1945. Er nannte diese Sprache “Plankalkül“. Anfang der 60er sprossen dann Programmiersprachen wie Pilze aus dem Boden. Viele der seit langem verfügbaren Sprachen wie COBOL oder FORTRAN halten sich bis heute. Auch Assembler ist im Bereich der Mainframes nicht zu ersetzen. All diesen imperativen Programmiersprachen ist jedoch gemein, dass die in ihnen geschrie-

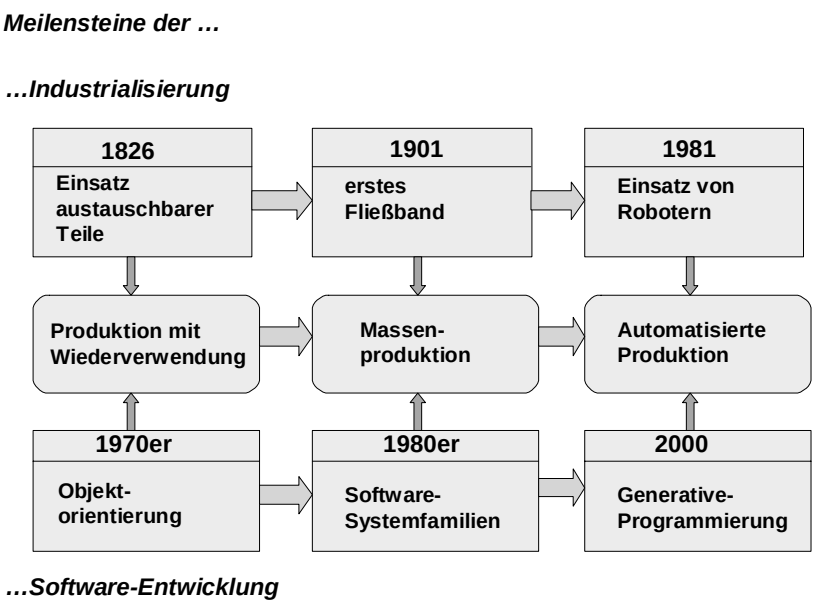
benen Programmteile nur für den Einsatz in einem bestimmten System vorgesehen sind.
Erst die Einführung der Objektorientierung in den 70ern brachte entscheidende Vorteile, bereits verwendete Code-Bausteine in anderen Systemen erneut zu verwenden. Diese Produktion von wiederverwendbaren Teilen ist vergleichbar mit dem Konzept der austauschbaren Teile, das in der Industrie bereits 1826 umgesetzt wurde.

Abbildung 2.2 Softwareproduktion



Ein weiterer entscheidender Schritt war die Anpassung und Ausrichtung der Softwareentwicklung an bzw. auf die Produktion von Software-Systemfamilien, die in den 80ern Fuß fasste. Software-Systemfamilien sind Softwaregruppen, deren Einzelsysteme alle aus einer gemeinsamen Menge von Komponenten bestehen. Kennzeichnend für eine Software-Systemfamilie ist die gemeinsame Software-Architektur, die allen Systemen zu Grunde liegt. Diese Erneuerungen in der Softwareentwicklung war vergleichbar mit dem Einsatz des Fließbandes der Beginn der Massenproduktion. Wobei hier unter Massenproduktion keineswegs die Produktion eines Produktes in großer Menge zu verstehen ist, sondern viel mehr die Produktion von verschiedenen Varianten einer Systemfamilie.
Der letzte Schritt um auf den Stand der Technik in der Industrie aufzuschließen, ist die Generative Programmierung, deren Entstehungsgeschichte um die Jahrtausendsende beginnt. Ihr Ziel ist es den Fortschritt der Software-Systemfamilien weiter zu treiben und die Automatisierung von Arbeitsschritten auch in die moderne Softwareentwicklung einzuführen. Generative Programmierung ermöglicht die automatisierte Produktion von konfigurierbaren Zwischen- oder Endprodukten, wie sie vergleichbar in der Automobilindustrie stattfindet, wo der Kunde sein Auto bis ins Detail selbst zusammenstellen kann. Diese Analogie zwischen Softwareproduktion und Industrie wird in Abbildung 2.3 zusammengefasst.

Abbildung 2.3 Analogie Softwareentwicklung-Industrie



2.1.2 Ziele und Prinzipien der Generativen Programmierung

Es sind neue und größere Herausforderungen der Softwaretechnik anzusehen. Die Software-Produkte werden heutzutage komplexer und aufwendiger. Sie sollen nämlich neue Bereiche abdecken, schwierige Aufgaben lösen, mehr Merkmale und Anforderungen erfüllen, leichte Benutzbarkeit als Akzeptanzkriterium haben und - einmal in lebenskritische Bereiche eingesetzt - eine sehr hohe Qualität erreichen. Außerdem sind sehr hohe wirtschaftliche Verluste bei Ausfall zu erwarten. Neben der Qualität ist auch die Produktivität anzustreben. Und dies wegen der wachsenden Nachfrage neuer Systeme und Varianten, unzureichender Produktivitäts- und Personalzuwachs und die Tatsache, dass die Standardsoftware nicht beliebig anpaßbar sind. Zudem haben die Software-Produkte lange Lebensdauer und es kommen immer neue Anforderungen dazu. Diese Software-Produkte sollen daher gewartet werden, denn die heutige Systeme sind Altlasten von morgen. Die Lösung für all diese Anforderungen besteht darin, Prinzipien wie systematische Wiederverwendung und Modularisierung bei der Softwareproduktion einzuführen. Die Widerverwendung soll die Produktivität und die Qualität steigern, während die Modularisierung die Wartung vereinfachen soll, da einzelne Bausteine leichter gewartet, getauscht und wiederverwendet werden können. Dies sind unter anderem die Ziele der Generativen programmierung. Wie bereits gesagt, versucht die Generative Programmierung die Automatisierung auch in der Softwareentwicklung umzusetzen. Es wird von ihr erwartet, ähnlich wie auch schon in der Industrie, eine kürzere Produktionszeit der zu entwickelnden Softwarekomponenten, die gleichzeitig eine höhere Produktivität zur Folge hat. Dies ist nicht nur ein wirtschaftlicher Aspekt, der zum einen zur Kostenreduktion und zum anderen zur Gewinnsteigerung beiträgt, sondern liegt auch im Interesse des Kunden, der durch einen beschleunigten Entwicklungsprozess nicht nur früher die fertig gestellte Software einsetzen kann, sondern auch von den geringeren Kosten, die für die Entwicklung anfallen, profitiert.

Die Generative Programmierung versucht eine bessere Wiederverwendbarkeit von Software-Komponenten zu erreichen, die einfacher und somit schneller an die Problemstellung anzupassen sind. Ein anderes Ziel ist es, die immer komplexer werdenden Aufgaben und Systeme mit geringmöglichem Aufwand beherrschbar zu machen. Hierzu soll die Realisierung des Problems im Lösungsraum, dem Entwickler aus der Hand genommen werden, der sich dadurch nicht mehr bis ins Detail über das fachspezifische Wissen des Anwendungsbereiches informieren muss. Außerdem soll mit möglichst einfachen Mitteln eine ganze Reihe kundenspezifischer system-Varianten produziert werden können. Weiter soll auch die Qualität der Produkte gesteigert werden. Es sind also optimierte Systeme zu entwickeln, die in Bezug auf Geschwindigkeit und Ressourcenverbrauch optimal an die gegebene Problemstellung angepasst sind. Zusätzlich resultiert eine bessere Qualität aus der Fehlervermeidung durch automatisierte Code-Generierung, die bei der Handarbeit nie ganz ausgeschlossen werden kann.
Die Generative Programmierung soll also allgemein gesagt, die Entwicklung von Software-Systemen vereinfachen, verbessern und beschleunigen, d.h. die Generative Programmierung, konsequent und richtig eingesetzt, soll ein hoch effizientes Mittel darstellen, um die Entwicklungskosten und -zeit zu senken und gleichzeitig die Softwarequalität zu steigern.

2.1.3 Definitionen

In den nächsten Abschnitten werden die wichtigsten Definitionen aus dem Gebiet der Generativen Programmierung vorgestellt.

Generative Programmierung

„Generative Programmierung modelliert Software-Systemfamilien so, dass ausgehend von einer Anforderungsspezifikation mittels Konfigurationswissen aus elementaren, wiederverwendbaren Implementierungskomponenten ein hochangepasstes und optimiertes Zwischen- oder Endprodukt nach Bedarf automatisch erzeugt werden kann.“ [CE00]
Mit anderen Worten wird also bei der Generativen Programmierung aus einer in einer speziellen Sprache aufgeschriebenen Anforderungsspezifikation automatisch ein Software-Produkt generiert, das diesen Anforderungen entspricht, ohne dass der Programmierer selbst in den Design- und Implementierungsprozess eingreift.

Software-Produktlinien

Die Begriffe *Produktlinie* und *Produktfamilie* bzw. *Systemfamilie* werden meinst gleichbedeutend gebraucht. Eine mögliche und sinnvolle Unterscheidung besteht jedoch darin, eine Produktfamilie als Menge von Produkten zu betrachten, die auf einer gemeinsamen Grundlage entwickelt werden; die Produkte können aus einer zugrunde liegenden Menge von Bausteinen erstellt werden und basieren auf der gleichen Architektur. Im Gegensatz dazu besteht eine Produktlinie aus einer Menge von Produkten, deren Merkmale auf die Bedürfnisse eines bestimmten Benutzerkreises oder Marktes abgestimmt sind [Wit96].
Eine Produktlinie könnte beispielsweise verschiedene Modelle bzw. Ausstattungsvarianten eines Automobiles umfassen, während eine Systemfamilie unterschiedliche Varianten eines Fahrzeugtyps vorsieht. Eine Produktlinie kann selbstverständlich auf der Grundlage von Produktfamilien realisiert werden.

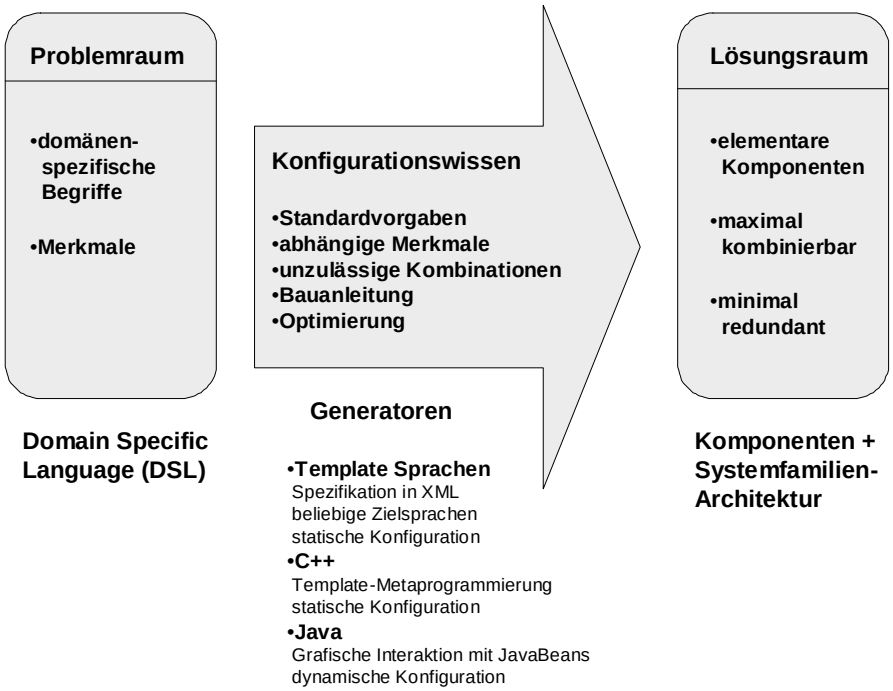
Domäne (domain)

In der Softwareentwicklung gibt es zwei Ansätze, was man unter der Domäne versteht. Der bislang verwendete Ansatz der Domäne als „reale Welt“ ist wie folgt zu definieren:
„Die Gesamtheit von Wissen, Vorgängen und Verfahren in einem bestimmten Bereich, die durch eine Menge von Begriffen und Terminologien gekennzeichnet und von Fachleuten in diesem Bereich verstanden werden“ [CE00].
Zum Beispiel die Bankbuchhaltungs-Domäne beinhaltet das Wissen über Konten, Kunden, Gutschriften, Abbuchungen, etc., jedoch nicht das Wissen über die Software, die die bankspezifischen Prozesse automatisiert und unterstützt.
Bei der Produktion von Software-Systemfamilien wird dieser Ansatz der Domäne noch durch das Wissen über das Software-Design und die Software-Implementierung ergänzt. Dieses Verständnis der Domäne wird als „Menge von Systemen“ bezeichnet.
Dementsprechend könnte die Definition von Domäne so lauten:
„Domäne ist der Bereich von Fachwissen, der für spezifische Anforderungen ausgewählt wird, Konzepte und Begriffe beinhaltet, die durch Anwender des Fachbereiches verstanden wird und das Wissen einschließt, wie Softwaresysteme (oder Teile davon) für diesen Fachbereich zu bauen sind“ [CE00].

Generatives Domänenmodell (generative domain model)

Das Kernstück der Generativen Programmierung ist das *generative Domänenmodell,* auf dessen Basis Software-Systemfamilien automatisch generiert werden können. Wie aus Abbildung 2.4 ersichtlich ist, besteht das generative Domänenmodell aus drei Teilen: dem Problemraum, dem Lösungsraum und dem Konfigurationswissen, mit dessen Hilfe eine Spezifikation aus dem Problemraum in ein Produkt transformiert wird, das aus den Komponenten des Lösungsraums zusammengesetzt ist.
Der Problemraum auf der einen Seite enthält domänenspezifische Begriffe und Merkmale aus dem Sprachgebrauch des Anwenders. Sie ermöglichen dem Anwender eine genaue Definition seiner Anforderungen. Um die Spezifikation für den Entwickler möglichst einfach zu gestalten und für deren automatisierte Weiterverwendung bekommt er hierzu eine eigene Sprache an die Hand, eine Domänenspezifische Sprache (DSL). Dieser Begriff wird später näher erläutert.
Der Lösungsraum auf der anderen Seite entspricht der Potenzmenge aller Komponenten und deren Kombinationen. In den Lösungsraum ist die gemeinsame Systemfamilien-Architektur eingebettet. Beim Entwurf der Komponenten des Lösungsraums soll auf maximale Kombinierbarkeit, so dass aus ihnen eine möglichst große Anzahl von System-Varianten generiert werden kann, und gleichzeitig auf minimale Redundanz (also keinen überflüssigen Code oder Code-Duplikationen) geachtet werden.
Das Konfigurationswissen definiert, welche Komponenten wie miteinander kombiniert werden dürfen und vor allem auch welche nicht. Es ergänzt die Spezifikation mit Standardvorgaben, beachtet Abhängigkeiten zwischen verschiedenen Merkmalen und beinhaltet letztendlich auch einen Produktionsplan (also das Wissen über die Implementierung) und Optimierungsregeln, die eng miteinander verknüpft sind.

Abbildung 2.4 Generatives Domänenmodell



In diesem Modell steckt das ganze Konzept der Generativen Programmierung, bei der ausgehend von der Spezifikation in der DSL durch den Generator, der das Konfigurationswissen enthält, ein Produkt der Software-Systemfamilie generiert wird.

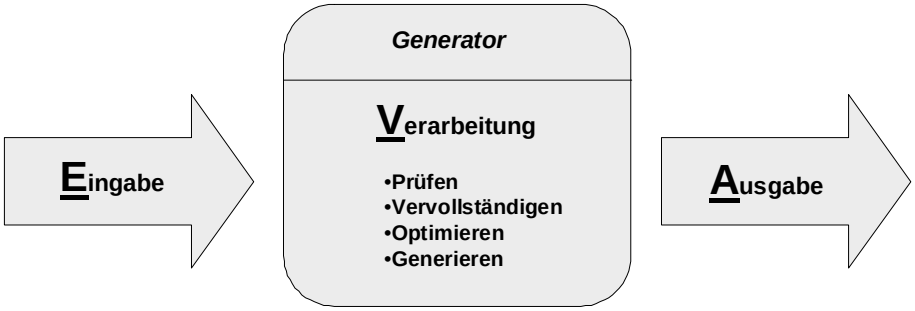
Domänenspezifische Sprachen (DSLs)

Eine domänenspezifische Sprache (Domain Specific Language, DSL) *"nimmt in ihren Ausdrücken auf das in einer Domäne enthaltene Wissen Bezug, indem Fachausdrücke und Konzepte eingebunden werden, die außerhalb der Domäne mit anderer Bedeutung oder überhaupt nicht definiert sind"*.
Bei der Anwendung der Generativen Programmierung, ist eine DSL der wichtigste Bestandteil, da sie das einzige Werkzeug des Entwicklers ist. Die Spezifikation erfolgt direkt in der DSL und nur durch Änderungen dieser kann das generierte Software-Produkt verändert werden. Eine DSL ist speziell auf die jeweilige Domäne ausgerichtet und somit eine problemorientierte Sprache.
In ihr lässt sich durch die angepasste Syntax und Semantik an die Domäne möglichst einfach und exakt eine Spezifikation festlegen, die ein bestimmtes Produkt aus der dazugehörigen Software-Systemfamilie beschreibt. Diese in der DSL geschriebene Spezifikation wird bei der Generativen Programmierung vom Generator als Eingabe verstanden, der daraus direkt das fertige Zwischen- oder Endprodukt generiert. Hierbei ist es unerheblich und vor allem domänenabhängig, ob die DSL graphisch oder textuell ist.
Bei der Entwicklung der DSL sollte möglichst darauf geachtet werden, dass sie direkt bei der Spezifikation keine unerlaubten Kombinationsmöglichkeiten oder Abhängigkeiten zulässt, da es ansonsten Probleme bei der Generierung geben kann, die dann eventuell zu einer mehrmals abzuändernden Spezifikation führen.

Software-Generatoren

"Software-Generatoren sind komplexe Programme, die durch die Verarbeitung von DSLs genau spezifizierte Systeme aus einer Software-Systemfamilie automatisch implementieren".
Wie aus Abbildung 2.5 ersichtlich ist, ist die Arbeitsweise von Generatoren in drei Teile, das sogenannte EVA-Prinzip, gegliedert: die Eingabe, die Verarbeitung und die Ausgabe.

Abbildung 2.5 Arbeitsweise von Generatoren (EVA-Prinzip)



Die Eingabe muss in einer für den Generator verständlichen Sprache, der DSL, erfolgen. Diese Eingabe wird dann zuerst auf Akzeptanz und Richtigkeit geprüft, was die Verwendung eines Lexers und eines Parsers einschließt. Wird die Eingabe so akzeptiert, kommt als nächster Schritt die Vervollständigung, die vom Benutzer offen gelassene Aspekte der Spezifikation mit vorgegebenen Standardwerten ergänzt. Der dritte Schritt ist die Optimierung, die die besonders effiziente System-Architektur der Systemfamilie und eine optimale Kombination der dem Generator zur Verfügung stehenden Komponenten verbindet. Zum Schluss erfolgt die Generierung der Ausgabe.

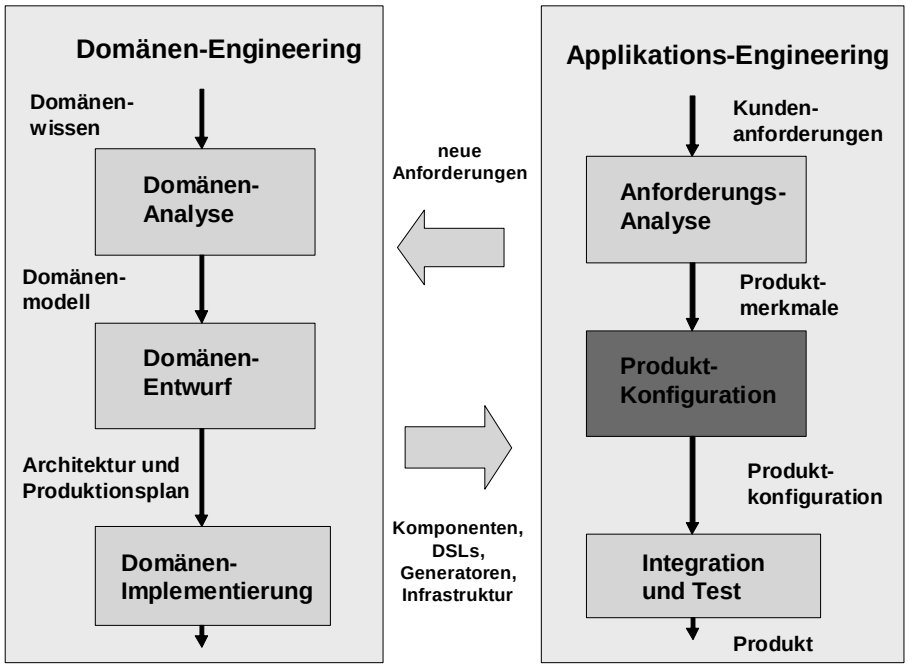
2.1.4 Die Phasen der Produktlinienentwicklung

Nachdem wir alle benötigten Begriffe der Generativen Programmierung definiert haben, kommen wir nun zum eigentlichen Prozess der Generativen Programmierung, infolge dessen Produktlinien entwickelt werden können. Es sind bei der Entwicklung von Software-Produktlinien zwei logische Phasen zu unterscheiden: Domänen-Engineering (domain engineering) und Applikations-Engineering (application engineering) [CE00]. Abbildung 2.6 enthält einen Überblick über das Zusammenspiel der in diesen beiden Phasen enthaltenden Vorgänge.

Domänen-Engineering (domain engineering)

"Domänen-Engineering ist das Sammeln, Organisieren und Festhalten bisheriger Erfahrungen im Erstellen von Systemen oder Teilen von Systemen in einer bestimmten Domäne in Form von wiederverwendbaren Werten, sowie das zur Verfügungstellen von ausreichenden Mitteln für das Wiederverwenden dieser Werte zum Erstellen neuer Systeme" [CE00].

Abbildung 2.6 Produktlinienentwicklung



Das Domänen-Engineering ist somit die Vorarbeit der Generativen Programmierung, die aus der Domänen-Analyse, dem Domänen-Entwurf und der Domänen-Implementierung besteht und die als Ergebnis zum einen das Domänenmodell mit der DSL, zum anderen den Generator liefert.
Die Aufgaben der Domänen-Analyse umfassen die Abgrenzung der Domäne, die Wissensgewinnung über die Domäne, die Definition von wiederverwendbaren, konfigurierbaren Anforderungen für die Systeme der Domäne und die Analyse der Gemeinsamkeiten, Variabilitäten und Abhängigkeiten von Merkmalen innerhalb der Domäne. Aus der intensiven Beschäftigung mit der Domäne entsteht ein Domänenmodell mit einer festgelegten DSL.
Dieses Domänenmodell geht dann über in die zweite Phase, das Domänen-Entwurf, deren Kernaufgabe die Entwicklung einer gemeinsamen und optimal angepassten Systemarchitektur für die ganze Systemfamilie ist. Die letzte Phase des Domänen-Engineerings ist die Domänen-Implementierung, die eine Grundlage für das spätere Applikations-Engineering ist, um konkrete Zwischen- oder Endprodukte zu erzeugen. Hier werden die DSL, sowie die wiederverwendbaren Komponenten und der Generator für deren automatische Kombination aus den Informationen der beiden vorangegangenen Phasen implementiert.

Applikations-Engineering (application engineering)

Ziel des Applikations-Engineering ist es mit Hilfe der Ergebnisse des Domänen-Engineerings, also dem Domänenmodell und dem Generator, aus einer konkreten Spezifikation eines Produktes die dazugehörige Software-Variante zu generieren. Während der Anforderungs-Analyse werden die Anforderungen, die der Kunde wünscht, mit dem Wissen aus dem Domänen-Engineering analysiert und in das Domänenmodell eingebettet. Hier gibt es allerdings ein Problem, und zwar, dass die vom Kunden gewünschten Anforderungen, die beim Domänen-Engineering noch nicht

berücksichtigt wurden, nicht mit dem Domänenmodell kombiniert werden können. In dem Fall besteht keine Möglichkeit, diese Anforderungen manuell ins System einzubauen. Die neuen Anforderungen müssen zuerst das Domänen-Engineering durchlaufen und in den Generator eingebunden werden, bevor das Produkt entwickelt werden kann.
Aus diesen Kundenanforderungen wird das erwünschte Produkt konfiguriert. Das konfigurierte, in der DSL geschriebene Produkt wird dann an die System-Generierung übergeben. Hier übernimmt der Generator die Aufgaben des System-Designs und der System-Implementierung, die bei herkömmlicher Softwareentwicklung von den Entwicklern durchgeführt wurden. Er kombiniert und optimiert die ihm zur Verwendung stehenden Komponenten in der ihm zu Grunde liegenden Systemarchitektur. Dieser Prozess des Applikations-Engineering ist die Anwendung der Generativen Programmierung. Während das Domänen-Engineering nur einmal durchgeführt wird, ist das Applikations-Engineering dazu gedacht, nicht nur ein einzelnes System zu entwickeln, sondern verschiedene Teilnehmer einer Software-Systemfamilien.

2.2 Modellgetriebene Architekturen (MDA)

Wie in der IT üblich, gibt es auf dem Markt mehrere Implementierungen der Komponententechnologie in der Form von Middleware. Zum Beispiel CORBA [OMG02c], Java 2 Enterprise Edition [JCP01] mit EJB, COM/DCOM/.NET von Microsoft sowie als neuere Technologie Web-Services. Keine dieser genannten Technologien hat sich völlig durchgesetzt und so sind alle Vertreter mehr oder weniger weit verbreitet. Weiter sind alle genannten Technologien umfangreich und daher komplex. Es bedarf einiges an Spezialwissen, um eine Anwendung in einer dieser Technologien zu realisieren. Noch komplexer stellte sich die Aufgabe dar, unterschiedliche Middleware Technologien miteinander zu kombinieren. Diese Aufgabe tritt allerdings immer wieder auf, da sich keine der genannten Technologien allein durchsetzen konnte.
Die Object Management Group (OMG) will die oben beschriebenen Probleme mit der Model Driven Architecture (MDA) [OMG01a] lösen. Der Grundgedanke der MDA besteht darin, dass die Definition der Funktionalität des Systems von der Definition der Implementierung dieser Funktionalität für eine bestimmte Plattform getrennt wird. Die Abstraktion der Funktionalitäten eines Systems bietet entscheidende Vorteile:

- Es ist leichter die Korrektheit eines Modells zu überprüfen, welches von Plattformspezifischen Details befreit ist.
- Es ist leichter Lösungen auf verschiedenen Plattformen zu implementieren, welche sich alle auf ein einheitliches definiertes Modell stützen.
- Es ist leichter die Integration von verschiedenen Systemen in Plattform-unabhängigen Begriffen zu beschreiben, als mit Plattformspezifischen Details.

So können Anwendungen zunächst völlig abstrakt von einer konkreten Middleware entwickelt werden. Dadurch ist der Entwickler unabhängig von einer konkreten Middleware und dies ist ein deutlicher Vorteil, denn die Idee, dass sich ein Unternehmen auf eine Middleware festlegt, hat sich als nicht praxistauglich herausgestellt. Während ohne MDA eine Wiederverwendung über verschiedene Middleware Technologien nicht möglich ist, können in einem MDA-Modell Teile eines anderen Modells relativ leicht wiederverwendet werden. Da die Geschäftslogik auf einer Ebene modelliert ist, die von der technischen Realisierung unabhängig ist, wird auch die Wieder-

verwendung in neuen Middleware-Plattformen erleichtert. So kann z.B. die Geschäftslogik von einem EJB System in ein COM+ oder .NET system einfacher übernommen werden. Es lässt sich außerdem ein System mit viel geringerem - in der Theorie sogar ohne - Aufwand auf eine andere Middlewareplattform portieren. Damit kann die Lebensdauer der Systeme deutlich verlängert werden, da ein komplettes Umschreiben aus rein technischen Gründen nicht mehr notwendig ist. Den Kern von MDA bilden die bisherigen Standards der OMG. Zu diesen Standards zählen:

- UML (Unified Modeling Language) [OMG03a]: Dient der (graphischen) Beschreibung von Modellen, die je nach Modellart z. B. die Struktur oder das Verhalten festlegen.
- XMI (XML Metadata Interchange) [OMG02a]: Hierbei handelt es sich um ein XML-basiertes Austauschformat für Modelle, was die Interoperabilität von Modellen zwischen verschiedenen Werkzeugen unterschiedlicher Hersteller ermöglicht. So lassen sich mittels der MDA die Modelle zwischen verschiedenen Werkzeugen und Werkzeugkategorien unterschiedlicher Hersteller austauschen.
- MOF (Meta Object Facility) [OMG02b]: Ist auf der Meta-Metamodellebene angesiedelt und definiert den Aufbau der Metamodellebene auf der z.B. UML und CWM liegen.
- CWM (Common Warehouse Model) [OMG01b]: Basiert auf dem MOF und verknüpft UML, XMI und MOF zu einer Anwendung im Bereich Data Warehouses.

2.2.1 Modelle in MDA

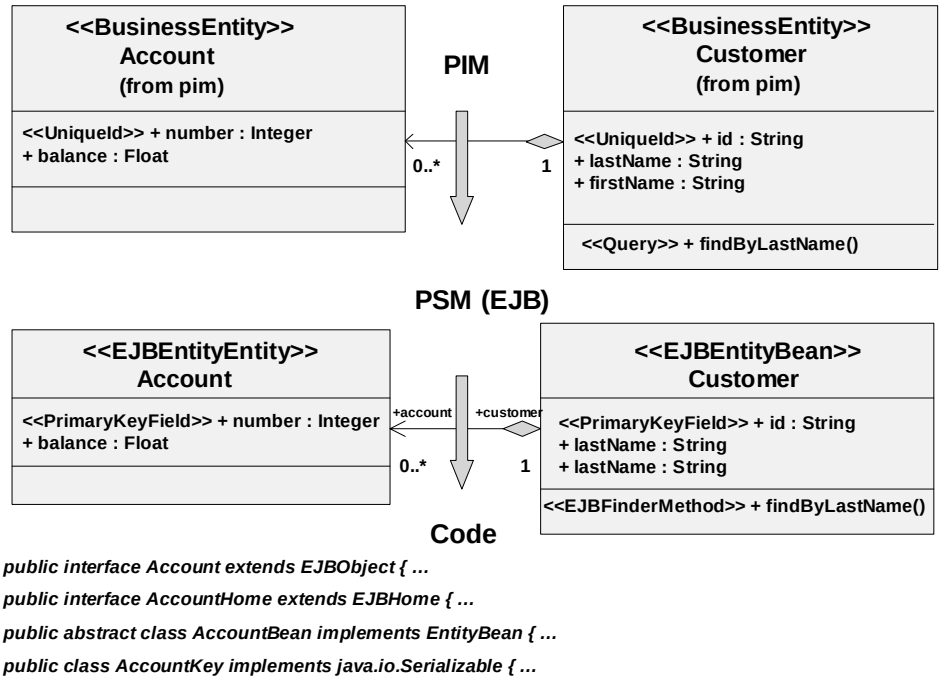
Ein Modell ist eine Abbildung der realen Welt und bildet eine formale Repräsentation eines Teils der Funktionalität und Struktur eines systems. Für die Erstellung eines Modells wird meist UML verwendet (von OMG als Sprache vorgeschlagen). Es können aber auch Schnittstellendefinitionen, Programmcode oder normaler Text eingesetzt werden. Es ist bei MDA wichtig, dass das Modell formal ist und somit Eindeutigkeit aufweist. Dadurch kann das Modell mit Werkzeugen zum Teil automatisch verarbeitet werden. In MDA wird vor allem zwischen zwei Arten von Modellen: dem plattformunabhängigen (PIM, Platform Independent Model) und dem plattformspezifischen (PSM, Platform Specific Model) Modell unterschieden. Abbildung 2.7 zeigt wie Entwurfsobjekte in den verschiedenen Modelltypen dargestellt werden können.
Der Entwickler beginnt mit der Modellierung der Geschäftslogik eines System. Da dieses Modell nicht von einer konkreten Implementierungstechnologie oder Programmiersprache abhängt, spricht man dabei von einem plattformunabhängigen Modell (PIM). Dieses PIM wird automatisch in ein Modell überführt, welches das PIM auf einer bestimmten technischen Plattform (z. B. EJB oder COM+) lauffähig macht - das plattformspezifische Modell (PSM).

Plattformunabhängige Modelle (PIM)

Dabei handelt es sich um eine formale Spezifikation, welche die Struktur bzw. die Funktionalität eines Systems beschreibt, ohne die Implementierungsdetails näher zu definieren. Für plattformunabhängige Modelle wird meistens UML als Modellierungssprache verwendet. Für die Verwendung von UML sprechen folgende Punkte:

- UML ist von OMG als Kern für die Modellierung definiert und ergänzt MDA optimal.
- UML kann sowohl für die visuelle als auch für die textuelle Darstellung von Modellen verwendet werden.
- UML ist semantisch vollständiger als andere deklarative Sprachen.

Abbildung 2.7 Darstellung von Entwurfsobjekte in den verschiedenen Modelltypen



Plattformspezifische Modelle (PSM)

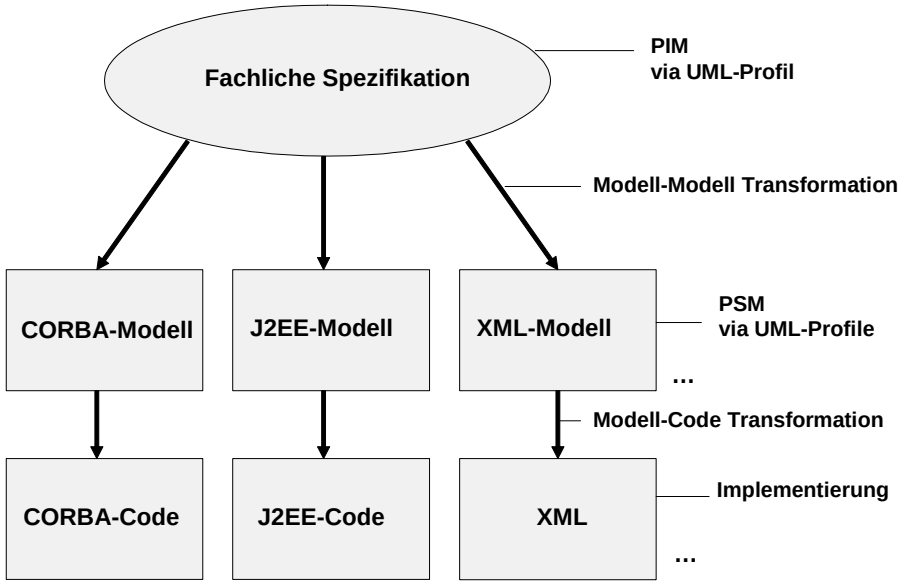
Durch Anreicherung mit zusätzlichen Plattform-spezifischen Informationen wird ein PSM erstellt. Prinzipiell ist es möglich plattformspezifische Modelle auf zwei Arten zu beschreiben: entweder mit UML-Modellen, die durch plattformspezifische UML-Profile erweitert werden, oder mit Schnittstellendefinitionen, die auf die entsprechenden Implementationstechnologien angepasst werden (z.B. IDL, XML, oder Java). Die plattformspezifischen Modelle basieren aber meistens auf OMG Standards und werden in UML ausgedrückt. Nun ist UML an sich aber ebenfalls Plattformunabhängig und nicht an eine bestimmte Middleware gebunden. Es stellt sich die Frage, wie mit Hilfe von UML ein PSM, welches spezifische Informationen bezüglich der ausgewählten Implementierungstechnologie enthält, beschrieben werden kann. Die Lösung für diese Problematik stellen UML Profile dar. Dabei handelt es sich um eine Anzahl von UML-Erweiterungen. Dazu wird der Erweiterungs-Mechanismus von UML verwendet (stereotypes, tagged values, constraints). Stereotypes werden dazu verwendet, um ein Modell-Element genauer zu definieren und seine Semantik auf einer Plattform zu kennzeichen. Zur Zeit existiert ein UML-Profil für CORBA sowie für EJB. Weitere Profile für .NET, Web Services befinden sich in Entwicklung. Also haben UML Profile bei den plattformspezifischen Modellen eine besondere Rolle. Sie ermöglichen dem Modellierer nämlich plattformspezifische Eigenschaften einzubinden. In dem Bespiel aus Abbildung 2.7 wird ein solcher Profil für die J2EE-Plattform verwendet.

2.2.2 Abbildung zwischen Modellen

Kernidee der MDA ist die schrittweise Verfeinerung von der Analyse zum Design ausgehend von einer Modellierung der fachlichen Applikationslogik, die völlig unabhängig von der technischen

Implementierung und der umgebenden IT-Infrastruktur ist. Das Grundprinzip zu MDA lässt sich wie in Abbildung 2.8 darstellen.

Abbildung 2.8 Überblick über den MDA-Prozess



Das Schema soll den Weg vom abstrakten Metamodell über die unabhängige Fachlogik zum konkreten Code auf der Plattform verdeutlichen, ähnlich einer CORBA-Spezifikation, der IDL (Interface Definition Language) und dem daraus folgenden Code. Es wird mit der Erstellung des Designs oder einer Referenz mit einem konventionellen UML-Tool begonnen. Das Tool muss aber fähig sein, mit UML-Profilen und constraints arbeiten zu können. Durch einen XMI-Export erhält man eine austauschbare Designdatei (PIM), die mit dem gewünschten UML-Profil einem MDA-Generator gegeben werden kann. Der Generator transformiert die Designdatei mit dem Profil nach spezifizierten Abbildungsregeln (Template) in einen konkreten Architekturrahmen. Nun wird das PSM erzeugt, das immer noch ein Modell ist. Der dritte Schritt ist die Abbildung (Mapping) des PSM auf eine konkrete Plattform. Dies kann, wie schon erwähnt, in einigen Fällen automatisch erfolgen, oder durch den gezielten Eingriff von “menschlichen“ Entwicklern. Das Ergebnis dieses letzten Mapping ist dann der source-Code der jeweiligen Implementierungsplattform.

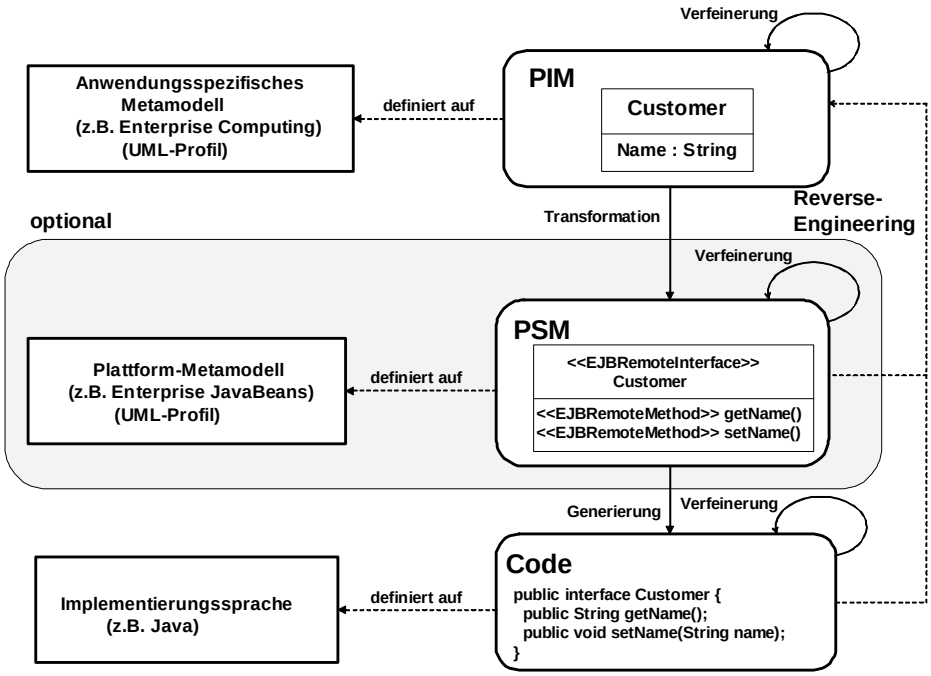
In MDA sind Abbildungen zwischen Modellen eine der Schlüsseleigenschaften. Eine Abbildung ist eine Menge von Regeln und Techniken, um ein Modell in ein anderes Modell zu transformieren. In Abbildung 2.9 sind mögliche Abbildungen dargestellt. Um ein plattformunabhängiges auf ein plattformspezifisches Modell abzubilden, gibt es drei Möglichkeiten [MM03]:

- Ein plattformspezifisches Modell kann aus dem Studium des plattformunabhängigen Modells manuell konstruiert werden.
- Aus einem plattformunabhängigen Modell kann ein Algorithmus das Gerüst für ein plattformspezifisches Modell erstellen. Das Gerüst kann dann manuell erweitert werden.

- Ein Algorithmus kann automatisiert aus einem plattformunabhängigen Modell ein komplettes plattformspezifisches Modell erstellen.

Die dritte Möglichkeit, eine vollautomatisierte Lösung, ist aber nur unter bestimmten Voraussetzungen möglich. Um diese Voraussetzungen zu schaffen, sollte die Semantik des Eingangsmodells gut beschrieben und der Algorithmus für die Transformation von hoher Qualität sein.

Abbildung 2.9 Abbildung zwischen Modellen in MDA



Grundsätzlich kann man in MDA folgende vier Transformationen anwenden:

PIM zu PIM

Diese Art von Transformation wird verwendet, wenn Modelle während des Entwicklungsprozesses erweitert, gefiltert oder verfeinert werden. Solche Operationen sind typischerweise Verfeinerung der Semantik, also Präzisierungen der Systemfunktionalität unabhägig von der technischen Realisierung. Diese Art von Transformation wird bei der Modelltransformation in dem in dieser Arbeit vorgestellten System verwendet. Dabei können die Transformationen durch den Benutzer definiert werden und dann durch das System, das eigentlich die Rolle eines Konfigurators übernimmt an das aktuelle Modell angepasst werden.

PIM zu PSM

Diese Transformation wird verwendet, wenn das plattformunabhängige Modell (PIM) auf die entsprechende Infrastruktur abgebildet werden soll. Dadurch kann ein PIM bis zur lauffähigen Anwendung verfeinern.

PSM zu PSM

Diese Transformation wird beispielsweise bei Abbildungen von Modellen auf verschiedene Komponenten im System verwendet. Komponenten werden dabei je nach gewählter Plattform unterschiedlich definiert. Es wird aber auch die Entwicklung auf unterschiedlichen Plattformen erleichtert. Zusätzlich kann durch die PSM-PSM-Transformation ein bestehendes Modell weiter verfeinert werden.

PSM zu PIM

Diese Transformation wird oft bei bestehenden Systemen bzw. Altsystemen eingesetzt, um daraus ein plattformunabhängiges Modell zu definieren. Es handelt sich dabei um eine Abstraktion des plattformspezifischen Modells. Das neue Modell (PIM) kann danach mit Hilfe der oben beschriebenen Transformationen weiterverwendet und eventuell auf neue Plattform abgebildet werden.

Fazit

Das Konzept der MDA ermöglicht also eine einheitliche Modellierung von Middleware-Lösungen für die verschiedenen Technologieplattformen. Entscheidend ist, dass dadurch ein hoher Grad der Wiederverwendbarkeit gegeben ist, was zu einer höheren Zukunftssicherheit der Anwendung führt, da dasselbe Modell der Geschäftsanwendung mit verschiedenen Implementierungstechnologien verwendet werden kann. Mit dieser Vorgehensweise kann mehr Zeit in die Modellierung der Geschäftsprozesse investiert werden, ohne dass sich der Entwickler Gedanken über die Details der Implementierung auf der Technologieplattform machen muss. Im nächsten Abschnitt wird untersucht, wie die Semantik modelliert werden kann.

2.3 Modellierung der Semantik

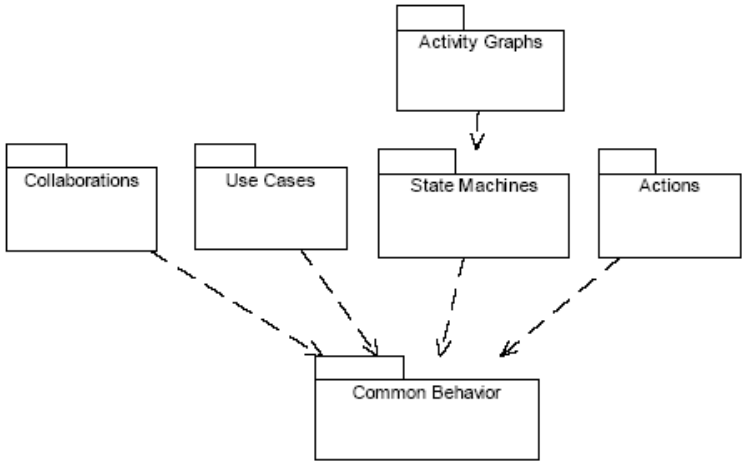
Im MDA-Ansatz wird UML für Architekturmodelle, Objektmodelle, Objektinteraktionen, Datenmodellierung und Designentwürfe von OMG als Standard vorgeschlagen. UML ist eine umfangreiche Sprache zum Modellieren und gut geeignet für Problemkonzeptualizierung, Erzeugung einer Software-Systemspezifikation sowie auch für Implementierung. Diese Sprache bietet verschiedene Diagrammtechniken, um den Zustand der Objekte beschreiben zu können. Das Problem war aber bei UML 1.4, dass die Operationen zwischen Objekten nicht so präzise beschrieben werden können. Zwar enthält das Package *Behavioral Elements* in der Spezifikation von UML 1.4 mehrere Subpackages für die Modellierung von dynamischen Verhalten, allerdings fehlt noch einen passenden Mechanismus, so dass eine präzise Spezifikation der Abläufe und Verhalten zwischen den Objekten erzeugt werden kann. Dieser Mechanismus war dann die Innovation in der Spezifikation von UML 1.5 mit dem im Package *Behavioral Elements* neu eingeführten Subpackage *Actions.*

2.3.1 Verhaltenssemantik in UML

Bei dem *Behavioral Elements* Package handelt es sich um eine gemeinsame Obermenge von Verhaltensweise, die bei der Modellierung mit den unterschiedlichsten Diagrammtypen verwendet werden können. Dieses Package ist innerhalb von UML für die Spezifikation von dynamischem Verhalten zuständig und es setzt sich wie in Abbildung 2.10 zu sehen ist aus folgenden Subpa-

ckages: *Common Behavior, Collaborations, Use Cases, State Machines* und *Activity Graphs.* Das Subpackage *Actions* kam als ein neues Subpackage hinzu.

Abbildung 2.10 Behavioral Elements Package



In *Common Behavior* sind die Grundlegenden Konstrukte vom dynamischen Verhalten zusammengefasst. *Collaborations* stellt die Sprachmittel für die Beschreibung des Verhaltens von Objekten beim Bearbeiten einer gegebenen Aufgabe. Das *Use Cases* Package spezifiziert Verhaltensbeschreibungen auf dem Weg über '*actors*' und '*use cases*'. Das *State Machines* Package erlaubt Verhaltensbeschreibungen mit endlichen Zustandsübergangsautomaten. In *Activity Graphs* wird ein Spezialfall davon für die Modellierung von Prozessabläufen bereitgestellt.

Common Behavior

Das *Common Behavior* Subpackage stellt die Grundlagen bereit für das gesamte *Behavioral Elements* Package. Dabei werden die Kernkonzepte der dynamischen Modellierung wie Signale, Ereignisse, Actions bereitgestellt, die für die verschiedenen Diagrammtechniken benötigt werden.

Collaborations

Im *Collaborations* Subpackage werden die Sprachmittel zur Definition von 'Collaborations' ('*ClassifierRole*', '*AssociationRole*', '*Interaction*', '*Message*') sowie '*CollaborationInstanceSets*' ('*Instance*', '*Stimulus*', '*Link*') bereitgestellt.

Use Cases

Das *Use Cases* Subpackage dient, das Verhalten eines Systems oder Subsystems im Großen zu beshreiben, ohne dabei in seine interne Struktur eindringen zu müssen. Die Basiskonstrukte sind '*Use Case*' und '*Actor*'. Jeder '*Use Case*' spezifiziert einen Teil des Systemverhaltens als Folge von Aktionen in Interaktion mit den (externen) Akteuren ('*Actors*'). Diese Aktionsfolgen können dann durch andere Techniken wie Collaborations, State Machines oder Activity Graphs detaillierter und im Hinblick auf die Realisierung beschrieben werden.

State Machines

Das *State Machines* Subpackage liefert eine Menge von Konzepten, mit denen diskretes Verhalten durch endliche Zustandübergangsautomaten (*Finite State Machines*) beschrieben werden kann. State Machines erlauben es, in vielfältiger Weise das Verhalten von Modellelementen zu spezifizieren z. B. durch Modellierung des Verhaltens individueller Entities wie Klassen-Instanzen.

Activity Graphs

Das *Activity Graphs* Subpackage definiert eine erweiterte Sicht auf das *State Machines* Subpackage. Beide sind im Kern Zustandsübergangssysteme, und benutzen viele gemeinsame Metamodell-Elemente. Ein *Activity Graph* ist ein Spezialfall einer *State Machine*, um damit Prozesse modellieren zu können, bei denen für einen oder mehrere '*Classifiers*' mehr darauf geblickt werden soll, welche Aktionsfolgen unter welchen Bedingungen ablaufen sollen und weniger welcher '*Classifier*' dies tut. Viele Zustände in solchen Graphen repräsentieren Aktionen, die andere aufrufen und dann auf deren Erledigung warten. Übergänge in solche Zustände werden typischerweise durch '*events*' ausgelöst. Das sind hier die Beendigung eines vorhergehenden '*action states*', die Verfügbarkeit eines Objekts in einem bestimmten Zustand, das Auftreten eines Signals, oder das Erfüllen einer gegebenen Bedingung.

Actions

Das Subpackage *Actions* stellt den Versuch dar, UML um eine präzise, sprachunabhängige Beschreibung des Verhaltens zu erweitern. Die Vorteile des präzisen Spezifikation der Verhältnisse sind einerseits, eine frühere Prüfung der Korrektheit auf eine abstraktere Softwareentwicklungsebene durchzuführen und andererseits die Wiederverwendung auf Spezifikationssebene zu ermöglichen. Wie bereits erwähnt, geben die oben vorgestellten Methoden kaum Möglichkeiten, eine präzise Spezifikation der komplexen Abläufe und Verhalten zu definieren. Das Actions Subpackage soll diese Lücke beim Modellieren des Verhaltens schließen, indem es *Action Semantics* definiert. Aktionen können auf verschiedene Weisen und auf mehrere Ebenen verschachtelt werden. Die *Procedure* ist die höchste dieser Ebenen. Eine Prozedur fasst eine Menge von Aktionen zusammen, die so im Gesamtmodell mit anderen Konstrukten assoziiert werden können. *Action Semantics* definieren den genauen Inhalt der Prozedur sowie die zu bearbeitenden Daten.

Die Actions-Beschreibung liefert einen Überblick über die Möglichkeit, wie man mit Hilfe der Verhältnisse der Objekte, ein Modell aufbauen kann. In dieser Beschreibung sind solche Bestandteile zu finden:

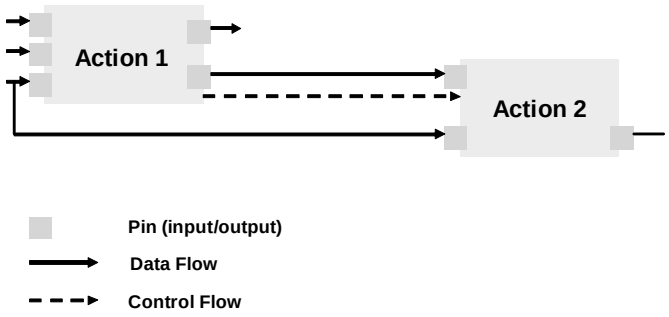
- *Datenfluss* (Data Flow) d. h. die Übertragung der Daten zwischen zwei Aktionen oder Folgen von Aktionen.
- *Kontrollfluss* (Control Flow), damit eine Regelung der Reihenfolge der Abläufe realisiert werden kann. Durch Daten- oder Kontrollflüsse können Abhängigkeiten zwischen Aktionen definiert werden.
- *Pins* zum Anschließen des Datenflusses an andere Bestandteile (*Actions*). Durch *Pins* wird jede Ein- und Ausgabe von Aktionen spezifiziert. Jede Datenübertragung hat eine Quelle, meistens ist das der Ausgang eines anderen Elements (*Output Pin*) sowie ein Ziel und meistens ist es der Eingang des nächsten Elements (*Input Pin*). Ein Ausgabe-Pin kann mit beliebig

vielen Eingabe-Pins verbunden sein; ein Eingabe-Pin kann jedoch mit höchstens einem Ausgabe-Pin verbunden sein.

- *Primitive Aktionen* (Primitive Actions): Das sind möglichst feingranulare Konstrukte, die nicht weiter unterteilt werden können z. B. arithmetische Funktionen.
- *Prozedur* (Procedure): Gruppe von Aktionen, die eine sinnvolle Funktion erfüllt.

In Abbildung 2.11 ist beispielhaft eine mögliche graphische Darstellung der Kooperation zwischen Aktionen, wo ein Paar der oben genannten Konstrukten zu erkennen sind.

Abbildung 2.11 Allgemeine Darstellung



Wie aus Abbildung 2.12 ersichtlich ist, gibt es mehrere Typen von Aktionen.

- *Composite Actions*: Mit solchen Aktionen können bedingte Ausdrücke ausgeführt oder verschiedene Iterationen durchgeführt werden. Sie bestehen meistens aus zwei Teilen, einem Test- und einem Körperteil. Der erste Teil prüft und gibt einen boolschen Wert zurück und erst dann kann der zweite Teil anfangen, den Prozess auszuführen.
- *Read Write Actions*: Sie führen Lese- und Schreibeoperationen auf Objekten und Assoziationen durch. Sie haben für verschiedene Elemente unterschieliche Bedeutungen. Sie dienen z.B. der Erzeugung von Objekten, dem Zuweisen von Werten an Variablen und Attributen oder dem Anlegen von Assoziationen.
- *Computation Actions*: Sie führen Berechnungen durch und greifen nicht auf Objekte zu. Das sind Aktionen, die mit mathematischen Funktionen verbunden sind und beschäftigen sich hauptsächlich mit der Berechnung der Variablenwerte.
- *Collection Actions*: Das sind Aktionen, die eine Iteration über eine bestimmte Menge von Elementen durchführen.
- *Messaging Actions*: Sie ermöglichen den synchronen oder asynchronen Austausch von Nachrichten zwischen Objekten.
- *Jump Actions*: Sie haben dieselbe Bedeutung wie Break, Continue oder Exceptions in anderen Programmiersprachen und ermöglichen das Verlassen des Kontrollflusses.

Abbildung 2.12 Action Package

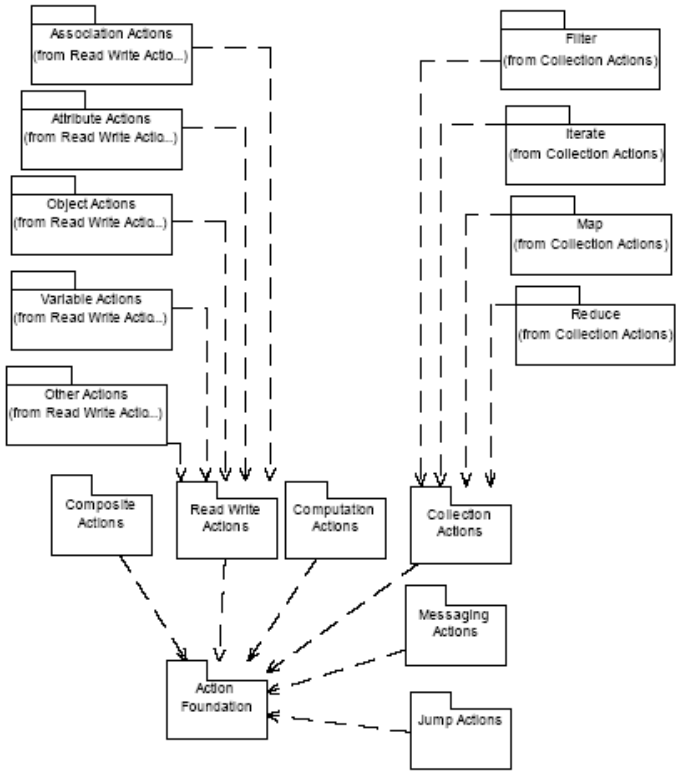
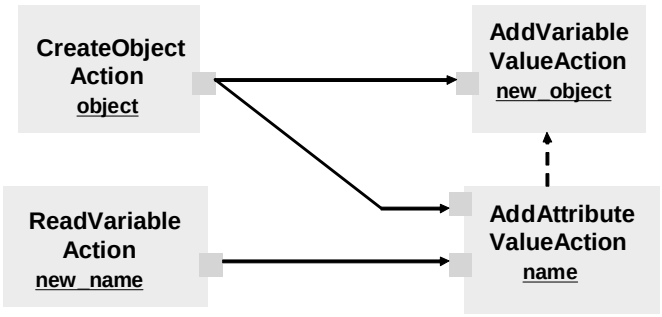


Abbildung 2.13 Erzeugung eines Objekts und Zuweisung eines Attributes

Der entsprechnde Code sieht so aus:

```
new_customer = (new Customer()).name = new_name;
```

Das UML Action Semantics Modell hat folgende Struktur:



Die Abbildung 2.13 gibt einen Überblick wie der UML Action Semantics Mechanismus funktioniert. Es soll dabei ein Objekt erzeugt werden und ein Name zugewiesen werden. Zunächst wird ein Objekt durch *CreateObjectAction* erzeugt. Durch *ReadVariableAction* wird in einer Variable

ein neuer Name angelegt und durch *AddAttributeValueAction* an dem neuen Objekt wie ein Attribut zugewiesen. Der Kontrollfluss zwischen *AddVariableValueAction* und *AddAttributeValueAction* regelt die Reihenfolge der Ausführung, so dass erst das Objekt erzeugt und dann der Name zugewiesen wird.

Object Constraint Language

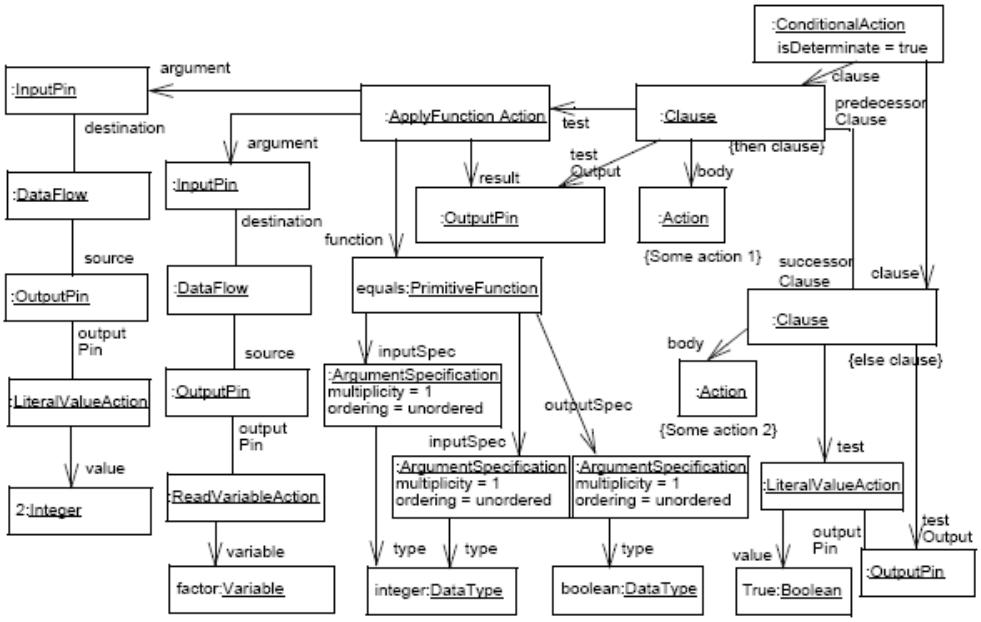
Constraints können folgendermaßen definiert werden:
"*Ein Constraint ist eine Restriktion von einem oder mehreren Werten eines objektorientierten Modells oder Systems bzw. einem Teil desgleichen.*"
So wurde die Object Constraint Language (OCL) entwickelt, um Bedingungen beim objektorientierten Softwareentwurf mit der UML ausdrücken zu können. Es handelt sich dabei um eine textbasierte Sprache, die es ermöglicht, Einschränkungen über einem gegebenen UML-Modell zu formulieren. OCL wird zur Spezifikation von Invarianten im Kontext von benutzerdefinierten Klassen, zur Formulierung von Vor- und Nachbedingungen für Operationen und in Bedingungen von Zuständsübergangen eingesetzt. OCL bietet nur ein bedingt geeignetes Mittel, um temporale zeitbehaftete Modelleigenschaften zu beschreiben. Insbesondere können mit OCL keine Einschränkungen über das dynamische Verhalten eines UML-Modells formuliert werden, die die Reihenfolge von Objektzuständen und Zuständsübergängen betreffen. Um ein korrektes Systemverhalten zu garantieren, ist es jedoch insbesondere bei zeitkritischen Anwendungen notwendig, solche zustandsbasierten zeitbehafteten Einschränkungen in einer formalen Art und Weise ausdrücken zu können. Zwar können über Vor- und Nachbedingungen Zustandsänderungen spezifiziert werden und es sind verschiedene Vorschläge veröffentlicht worden, in denen die OCL erweitert worden ist, um Modellierern die Möglichkeit zu geben, Verhalten zu beschreiben und temporale Einschränkungen zu formulieren; aber diese Versuche sind offenbar gegen die Natur von OCL als eine formale Sprache, die Einfachheit als wichtiges Merkmal hat und die ohne komplexe mathematische Formalismen auskommen soll.

Fazit

Mit *Action Semantics* wird erstmalig eine ernshafte Modellierung von Verhaltenssemantik in UML möglich. Die damit wachsende Modellierungsmächtigkeit von UML kommt auch der MDA zugute; es ist nämlich möglich geworden in plattformunabhängigen und plattformspezifischen Modellen das Verhalten eines Systems genauer und präziser zu spezifizieren. Zwar ist der Einsatz von *UML Action Semantics* beim Modellieren des Verhaltens sehr flexibel und umfassend, allerdings liegt der Nachteil in der beachtlichen Komplexität der Modelle. Abbildung 2.14 zeigt den Objektgraphen für eine einfache bedingte Aktion. Beim Modellieren komplexerer Semantiken enstehen Objektgraphen, die viel komplexer und nur schwer durchschaubar sind. Der Preis der Flexibilität der Action Semantics ist also in der Komplexität der enstehenden Modelle für die Spezifikation von Verhalten. Die Vorteile der *UML Action Semantics* sind daher unter dem Aspekt der guten Übersichtlichkeit und der Benutzerfreundlichkeit, die auch in allen Phasen der Software Entwicklung sehr wichtig sind, sehr fraglich. In der Praxis werden die *UML Action Semantics* in Entwicklerwerkzeugen oft mit einer weiteren Spezifikationssprache wie die Object Action Language (OAL) [Pro02] oder die Action Specification Language (ASL) [WKC+01] gekoppelt, um Verhalten zu modellieren. Im nächsten Abschnitt wird OAL als Aktionssprache vorgestellt, die in *MTFLOW* für die Darstellung der Operationen verwendet wird.

Abbildung 2.14 UML Action Semantics im Vergleich mit der textuellen Darstellung

```
if (factor == 2)
//Some action1
else
//Some action2
```



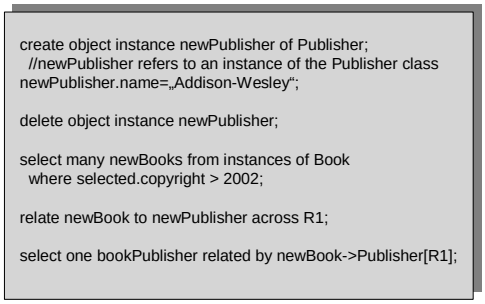
2.3.2 Object Action Language

Die Object Action Language (OAL) dient dazu, die Objekte zur Laufzeit zu verwalten. Diese Verwaltung betrifft Möglichkeiten zur Erstellung und Löschung von Objekten, zum Auslesen und Setzen von werten, zur Selektion von Objekten oder auch zur Erstellung und Auflösung von Verbindungen zwischen Objekten. OAL enstand als Spezifikationssprache für Action Semantics nach UML 1.5 im Entwicklerwerkzeug BridgePoint [Pro03], ist aber nicht an dieses gekoppelt [Pro02].

Mit OAL als Aktionssprache kam XTUML [MB02], ausführbares und übersetzbares UML (engl. eXecutable and Translatable UML). XTUML ist ein der MDA sehr nahestehender Ansatz. XTUML nutzt eine Menge der Diagramme von UML und OAL als Aktionssprache und ermöglicht eine präzise Spezifikation der dynamischen und statischen Aspekte von Programmen. Dadurch können Modelle ausgeführt und getestet werden, bevor eine Implementierung erstellt wird. Modellcompiler übersetzen die Modelle auf eine bestimmte Softwareplattform.

Die Aktionssprache OAL ist, wie es auch bei ASL der Fall ist, im Gegensatz zu den UML Action Semantics textbasiert und ist somit für den Benutzer gut lesbar und übersichtlicher. Im Vergleich zu ASL sind die Möglichkeiten für die Definition und Verarbeitung von Ausdrücken detaillierter. Abbildung 2.15 zeigt einfache Beispiele von OAL-Konstrukten. Aus all diesen Vorteilen wird in dieser Arbeit OAL als Sprache zum Modellieren der Semantik eingesetzt.

Abbildung 2.15 Die Aktionssprache OAL



Kapitel 3 Versionierungssysteme

In diesem Kapitel werden objektorientierte Versionierungssysteme (Repositories) vorgestellt, die als Produktlinie betrachtet werden können. Da für jede Produktlinie eine Domäne definiert werden soll, wird hier zunächst versucht, die Domäne der Versionierungssysteme ausführlich zu präsentieren. Eine Domäne ist die Gesamtheit von Wissen, Vorgängen und Verfahren in einem bestimmten Bereich, die durch eine Menge von Konzepten und Begriffen gekennzeichnet werden [CE00]. In objektorientierten Versionierungssystemen können Daten versioniert gespeichert werden, so dass mehrere Versionen eines bestimmten Objekts, d. h. sowohl der aktuelle Zustand des Objekts als auch die Zwischenzustände, die seine Entwicklung dokumentieren, parallel existieren können. Versionierungssysteme ermöglichen den Zugriff auf jede der gespeicherten Versionen und die Anwendugsübergreifende Nutzung der Objekte. Auf die in einem Versionierungssystem gespeicherten Objekte und deren Versionen können verschiedene Operationen ausgeführt werden. Die Aktionssprache OAL wird zur Darstellung dieser Operationen verwendet.

3.1 Objektorientierte Versionierungssysteme

Im folgenden werden die wichtigsten Eigenschaften objektorientierter Versionierungssysteme vorgestellt, wie sie von Bernstein [Ber98] beschrieben wurden. Versionierungssysteme erlauben, wie es in Abbildung 3.1 zu sehen ist, eine anwendungsübergreifende Nutzung der Daten. Sie stellen nämlich eine parallel nutzbare Schnittstelle für Anwendungsprogramme und Entwicklungswerkzeuge bereit. Sie basieren auf Datenbanksystemen bzw. Datenverwaltungssystemen anderer Art, wo die Speicherung der Daten erfolgt und die große Datenmengen effizient verwaltet werden können. Der Zugriff auf die im Versionierungssystem gespeicherten Daten erfolgt über den Versionierungssystem-Manager. Seine Aufgabe besteht darin, Operationen für die Handhabung und Verwaltung der gespeicherten Daten dem Benutzer zur Verfügung zu stellen. Objektorientierte Versionierungssysteme zeichnen sich im Gegensatz zu objektorientierten Datenbanksystemen (OODBMS) z. B. durch ein Informationsmodell aus, dass speziell auf die zu speichernden Daten zugeschnitten ist. Ein solches Modell besteht wie es in Abbildung 3.2 zu erkennen ist aus einer beliebigen Anzahl von versionierten bzw. unversionierten Objekttypen mit ihren Attributen und Beziehungstypen zu anderen Objekttypen. Ein Datenobjekt ist eine Instanz des entsprechenden Objekttyps und ist die kleinste einzeln versionierbare Informationseinheit in einem Versionierungssystem. Bei der Speicherung der Datenobjekte wird zwischen versionierbaren und nicht versionierbaren Datenobjekten unterschieden, so dass der Einsatz versionierter Speicherung auf die Datenobjekte beschränkt werden kann, bei denen das für die Anwendung Sinn macht.

Abbildung 3.1 Anwendungsbeispiel für Versionierungssysteme

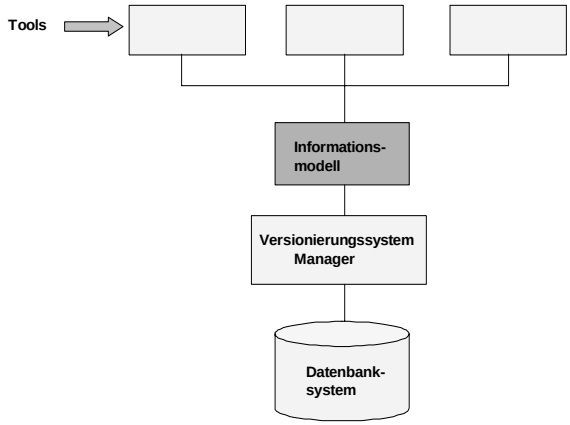
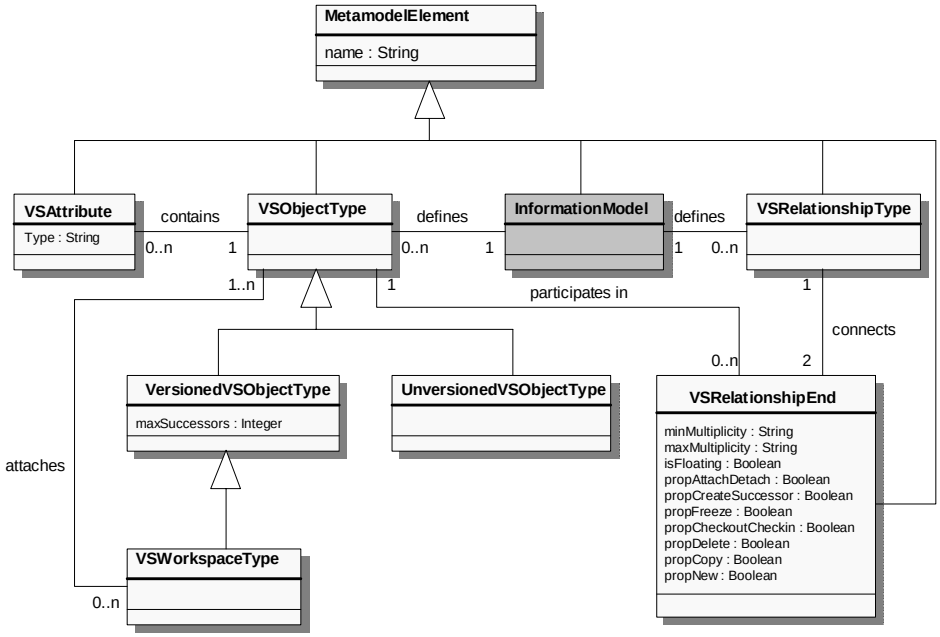


Abbildung 3.2 Metamodell für Versionierungssysteme

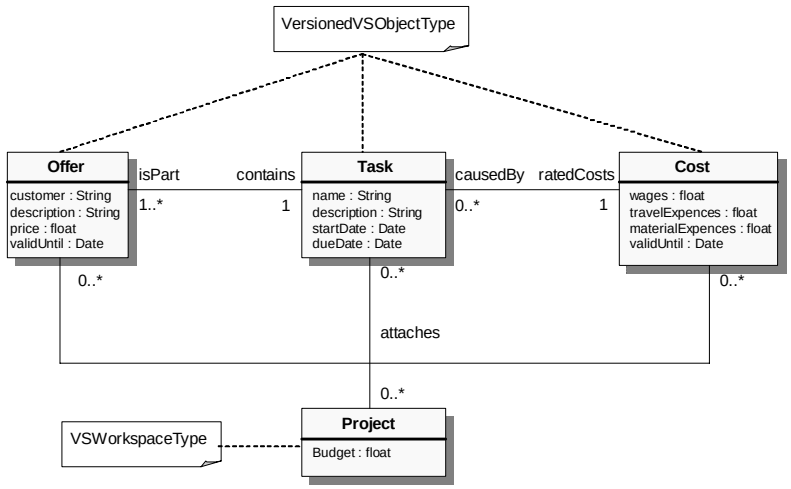


Für versionierbare Objekttypen ist es möglich, die maximal erlaubte Zahl der direkten Nachfolgeversionen (*maxSuccessors*) einer Instanz festzulegen. Nicht versionierbare Objekttypen sind ein trivialer Sonderfall der versionierbaren Objektypen, bei denen maximal eine Objektversion existieren darf. Versionierungssysteme können abweichend vom Normalfall auch unversionierte Objektypen enthalten, weil es vorkommen kann, dass zusätzliche Datenobjekte im Versionierungssystem gespeichert werden sollen, die in einer Beziehung mit den versionierten objekten sind, die aber einmal erzeugt nicht mehr verändert werden. Diese Objekttypen enthalten dann keine versionierungsspezifischen Attribute und es können keine Operationen der Versionsver-

waltung auf sie angewendet werden. Zwischen Objekttypen können Beziehungen bestehen, die ihrerseits Instanzen von Beziehungstypen sind. Die Beziehungen zwischen Datenobjekten sind im allgemeinen binär, d. h. sie verbinden genau zwei Objekttypen und bi-direktional, so dass die Navigation in beide Richtungen erfolgen kann. Die Kardinalität der Beziehungen kann 1:1, 1:n oder n:m betragen. Sowohl Datenobjekte als auch Beziehungen können Attribute enthalten, in denen sich Daten ablegen lassen.

Wie erwähnt wird die Struktur eines objektorientierten Versionierungssystem, also die enthaltenen Objekttypen und Beziehungstypen, in einem Informationsmodell beschrieben. Ein solches Informationsmodell lässt sich in diesem Fall als UML-Klassendiagramm darsellen. Das Diagramm zeigt dann Objektypen als Klassen und Beziehungstypen als Assoziationen zwischen den Klassen. Ein solches Modell ist in Abbildung 3.3 dargestellt und soll im folgenden als laufendes Beispiel dienen. Das Modell zeigt ein Beispiel eines Informationsmodells für die Speicherung von Projektmanagement-Daten. Die Objekttypen für die Versionierten Objekte *Task*, *Offer* und *Cost* werden dabei als Klassen dargestellt. Die Beziehungstypen für die Beziehungen *isPart_contains* und *causedBy_ratedCosts* werden als Assoziationen zwischen den Klassen dargestellt.

Abbildung 3.3 Beispiel eines Informationsmodells für ein Versionierungssystem



Bei der Verwendung objektorientierter Versionierungssysteme werden neben den typischen Operationen für das Schreiben und Lesen von Daten auch Operationen für die Objekt- und Versionsverwaltung sowie für die Navigation zwischen den Objekten benötigt. Es besteht auch die Möglichkeit, bestimmte Operationen über Beziehungen auf assoziierte Objekte propagieren zu lassen. Diese Operationen werden vom Versionierungssystem als eine Schnittstelle angeboten. Diese Operationen dienen dazu, nach gespeicherten Objekten zu suchen, die Daten eines gefundenen Objekts auszulesen bzw. zu verändern und die versionierten bzw. unversionierten Objekte zu verwalten. In den nächsten Abschnitten werden die verschiedenen Operationen und die Eigenschaften der Verwaltung vorgestellt.

3.2 Objekt- und Versionsverwaltung

Die im Versionierungssystem abgelegten Objekte enthalten neben den im Informationsmodell vorgegebenen Attributen, die praktisch die Nutzdaten der Objekte darstellen, noch weitere Systembedingte Attribute, die zur Verwaltung der Objekte benötigt werden. Das wichtigste Attribut ist der Primärschlüssel, der jedes gespeicherten Objekts eindeutig identifiziert. Dieser Primärschlüssel wird aus mehreren Attributen gebildet:

- *objectId* - Diese Nummer dient der Identifikation der Instanz. Sie identifiziert zwar ein Objekt eindeutig, allerdings kann im Fall eines versionierten Objekts über diese Nummer alleine nicht zwischen den verschiedenen Versionen dieses Objekts unterschieden werden, weil alle Versionen eines Objekts dieselbe *objectId* besitzen.
- *verId* - Diese Nummer wird für die Verwaltung der Versionsabfolge benötigt. Sie identifiziert die konkrete Version eines versionierten Objekts. Unversionierte Objekttypen besitzen dieses Attribut nicht.
- *globalId* - Bei versionierten Objekten ist die Angabe beider oben genannten IDs nötig, um eine Version einer Instanz eindeutig zu bestimmen. So werden *objectId* und *verId* zu *globalId* zusammengefasst, über die jedes Objekt und jede Version im Versionierungssystem eindeutig angesprochen werden können. *GlobalId* kann daher als Primärschlüssel im Datenbankschema dienen, das zur Speicherung der im Versionierungssystem definierten Daten benutzt wird. Durch diese *globalId* haben die unterschiedlichen Versionen eine eigenständige Identität und können daher als eigenständige Objekte behandelt werden. Dies stellt auch einen weiteren Unterschied eines objektorientierten Versionierungssystem zu anderen objektorientierten Systemen dar.

Versionierte Objekte besitzen neben ihrer *verId* noch weitere Attribute, die den Zustand der Version beschreiben.

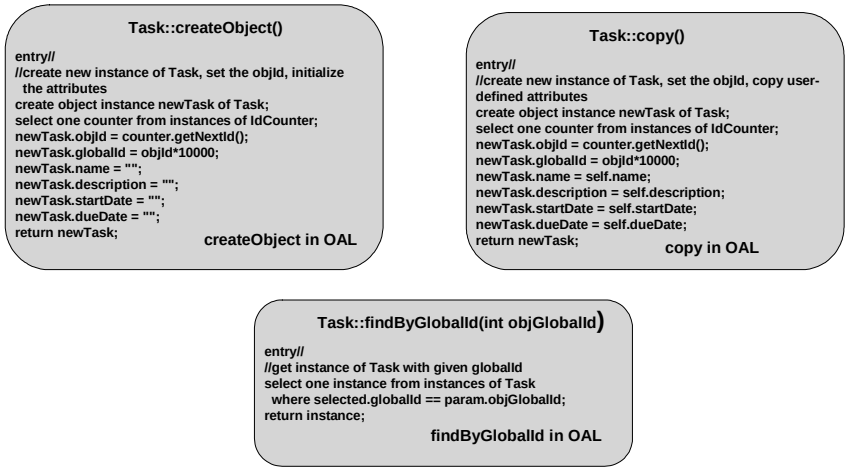
- *frozen* - Die Geschichte eines versionierten Objekts soll den Entwicklungsprozess dieses Objekts dokumentieren. Aus diesem Grund ist es sinnvoll, dass in der Versionshierarchie ältere Versionen ab einem bestimmten Zeitpunkt nicht mehr verändert werden können. Zu diesem zweck existiert das Attribut *frozen*, in dem wird vermerkt, dass die betreffende Version des Objekts gegen weitere Änderungen gesperrt wird.
- Weiterhin ist es möglich, einen Namen für die neu gelegte Version anzugeben (*versionName*) oder auch einen Zeitpunkt zu einer Version aufzuzeichnen, etwa das Datum der Versionserstellung oder das Datum der letzten Änderung (*versionDate*). Das Attribut *successorCount* bestimmt sie Anzahl der Nachfolgeversionen in der Versionshierarchie.

Die nachfolgenden Operationen können auf jedem Objekttyp ausgeführt werden d. h. sowohl auf unversionierbaren als auch versionierbaren Objektypen.

- *createObject* - Legt eine neue Instanz eines Objekttyps an. Falls für die Attribute keine Werte übergeben werden, werden die Attribute mit voreingestellten Standardwerten initialisiert. Die Instanz erhält eine neue noch nicht verwendete *objectId* (Abbildung 3.4). Handelt es sich um ein versionierbares Objekt, wird die *verId* auf 1 gesetzt, die *globalId*, die sich aus *objectId* und *verId* zusammensetzt, berechnet, das Attribut *frozen* auf false und *successorCount* auf 0 gesetzt.

- *copy* - Erzeugt ebenfalls wie *createObjekt* eine neue Instanz eines Objekts (Abbildung 3.4). Dabei werden aber die Attribute mit den Werten der bestehenden Instanz initialisiert. Wie bei *createObjekt* wird eine *objectId* vergeben, *globalId* berechnet und im Falle eines versionierten Objekts die *verId* auf 1 und *successorCount* auf 0 gesetzt. Das Attribut *frozen* wird immer auf false gesetzt, selbst wenn es in der Quellinstanz für die copy-Anweisung bereits auf true gesetzt sein sollte. So können die gesperrten Objekte, wo *frozen* auf true gesetzt wurde, weiter bearbeitet werden.
- *findByGlobalId* - liefert das objekt mit der gegebenen *globalId* zurück. Abbildung 3.4 enthält die OAL-Beschreibung dieser Operation.

Abbildung 3.4 Die Operationen *createObject*, *copy* und *findByGlobalId* in OAL

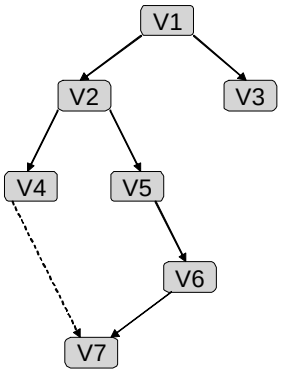


Neben den Operationen *createObject*, *copy* und *findByGlobalId* gibt es andere Operationen, die aber nur auf versionierbaren Objekten angewendet werden können. Sie dienen nämlich der Verwaltung der Versionshierarchie und der Unterstützung des Versionenzugriffs.

- *freeze* - Sperrt eine Objektversion gegen weitere Bearbeitung und Änderung der Attributwerte. Sollen dennoch Änderungen an den Attributwerten vorgenommen werden, so muss mit Hilfe der Operation *copy* ein neues Objekt oder mit der Operation *createSuccessor* eine Nachfolgeversion angelegt werden.
- *createSuccessor* - Erzeugt eine Nachfolgeversion eines versionierten Objekts, falls die maximale Versionsanzahl (*successorCount*) des Objekts noch nicht erreicht ist. Die *objektId* und die Attributwerte werden von der aktuellen Version übernommen und die *verId* wird erhöht. Das Attribut *frozen* wird ähnlich wie bei *copy* auf false gesetzt. Durch die Erzeugung einer Nachfolgeversion ist es daher möglich, Änderungen an Objekten durchzuführen, deren aktuelle Version durch die Operation *freeze* gesperrt wurde.
- *merge* - Mit der Operation *merge* können zwei Entwicklungszweige wieder vereinigt werden. Die *merge* Operation auf einer nicht gesperrten (Attribut *frozen* ist auf false gesetzt) Version V eines versionierten Objekts O nimmt zwei Eingaben als Parameter: Eine gesperrte version FV vom Objekt O und ein Flag, das entweder V oder FV als eine primäre Version identifi-

ziert. Die Operation *merge* legt zuerst FV als Vorgänger von V fest und führt dann V und FV wie folgt zusammen: Es wird zunächst mit Hilfe des Versionsgraphen die gemeinsame Basisversion (BV) bestimmt. Die gemeinsame Basisversion ist die am spätesten erzeugte Version, die sowohl im Pfad der Version V als auch im Pfad der Version FV zur Wurzel vorhanden ist. Im gezeigten Beispiel in Abbildung 3.5 ist V2 die gemeinsame Basisversion. Die Konflikte, die beim Zusammenführen der Versionen entstehen können, werden für jedes Attribut mit Hilfe von Regeln aufgelöst. Wenn ein Attribut in der primären Version im Vergleich zur Basisversion geändert wurde, enthält die zusammengeführte Version den Wert des Attributs der primären Version. Ist ein Attribut nur in der sekundären Version geändert, so wird dieser in die zusammengeführte Version übernommen [Ber99]. Die Tabelle in Abbildung 3.5 zeigt die Ergebnisse für die Operation *V7.merge(V4, true)*. Das Parameter *true* bedeutet, dass V7 die primäre Version sein soll.

Abbildung 3.5 Beispiel für die Operation *merge*



Offer / Version	customer	price	validUntil
V2	A	10	15.04.04
V4	A	15	30.04.04
V6	B	20	15.04.04
V7 (merged)	B	15	30.04.04

Die Operationen *freeze*, *createSuccessor* und *merge* werden in Abbildung 3.6 dargestellt. Für die Navigation im Versionsbaum eines versionierten Objekts existieren mehrere Operationen. Die OAL-Beschreibungen dieser Operationen sind in Abbildung 3.7 zu sehen.

- *getVersions* - Liefert alle Versionen eines Objekts zurück.
- *getSuccessors* - Liefert die direkten, durch die createSuccessor-Methode angelegten Nachfolger einer Objektversion zurück.
- *getPredecessor* - Liefert den direkten Vorgänger einer Objektversion zurück.
- *getAllPredecessors* - Liefert alle Vorgänger einer Objektversion zurück.

Abbildung 3.6 Die Operationen *freeze*, *createSuccessor* und *merge* in OAL

Task::freeze()

```
entry//
self.frozen = true;
```

freeze in OAL

Task::createSuccessor()

```
entry//
//check whether version is frozen
if((self.frozen == true) and (self.successorCount < 6))
//create new instance of Task, set verId, copy user-defined
attributes
create object instance newTask of Task;
//Should have the same objId
newTask.objId = self.objId;
//but a different verId
select one counter from instances of VerIdCounter
  where counter.objectId == self.objId;
newTask.verId = counter.getNextVerId();
newTask.globalId = newTask.objId*10000 + newTask.verId;
newTask.name = self.name;
newTask.description = self.description;
newTask.startDate = self.startDate;
newTask.dueDate = self.dueDate;
newTask.frozen = false;
//Does not have any successors yet
newTask.successorCount = 0;
newTask.versionName = "";
newTask.versionDate = TIM::current_date();
//relate the successor version to its predecessor
relate newTask to self across RelPred_Succ.'successors';
//increate the number of own successors
self.successorCount = self.successorCount + 1;
return newTask;
end if;
```

createSuccessor in OAL

Task::merge((Task FV, boolean primary))

```
entry//
//find  a least common ancestor(BV basis version) of self and FV(frozen version)
if((self.frozen == false) and (FV.frozen == true))
create object instance BV of Task;
BV.globalId = 0;
allPredecessorsCollection = self.getAllPredecessors();
allPredecessorsCollectionOfFV = FV.getAllPredecessors();
select many predecessors related by allPredecessorsCollection->Task[RelTask-
Collection.'object'];
for each predecessor in predecessors
select one item related by allPredecessorsCollectionOfFV->Task[RelTask-
Collection.'object']
  where item.globalId == predecessor.globalId;
 if ((not empty item) and (predecessor.globalId > BV.globalId))
BV = predecessor;
//BV basis version
end if;
end for;
//For each property P of Task, if only one of self or FV has a
//different value of P than BV, then the updated value is assigned
//to P in self. If both self and FV updated P, then the value of the
//primary is assigned to P in self.
if((self.name != BV.name) and (FV.name != BV.name))
if(primary == true)
self.name = FV.name;
end if;
elif(FV.name != BV.name)
self.name = FV.name;
end if;
if((self.description != BV.description) and (FV.description != BV.description))
if(primary == true)
self.description = FV.description;
end if;
elif(FV.description != BV.description)
self.description = FV.description;
 end if;
 if((self.startDate != BV.startDate) and (FV.startDate != BV.startDate))
if(primary == true)
self.startDate = FV.startDate;
end if;
elif(FV.startDate != BV.startDate)
self.startDate = FV.startDate;
end if;
if((self.dueDate != BV.dueDate) and (FV.dueDate != BV.dueDate))
if(primary == true)
self.dueDate = FV.dueDate;
end if;
elif(FV.dueDate != BV.dueDate)
self.dueDate = FV.dueDate;
end if;
end if;
```

merge in OAL

Abbildung 3.7 *getPredecessor*, getVersions, *getAllpredecessors und getSuccessors* in OAL

Task::getPredecessor()

```
entry//
select one predecessor related by self->Task[RelPred_Succ.'predecessor'];
return predecessor;
```

getPredecessor in OAL

Task::getAllPredecessors()

```
entry//
//create a collection to be filled with all prdecessors
create object instance collection of ReturnCollection;
actual = self;
while (true)
select one predecessor related by actual->Task[RelPred_Succ.'predecessor'];
if empty predecessor
break;
else
relate collection to  predecessor across RelTask-Collection.'returned objects';
actual = predecessor;
end if;
end while;
return collection;
```

getAllPredecessors in OAL

Task::getVersions()

```
entry//
//create a collection to be filled with all versions of Task
create object instance collection of ReturnCollection;
select many versions from instances of Task
   where selected.objId == self.objId
for each version in versions
relate collection to version across RelTask-Collection.'returned objects';
end for;
return collection;
```

getVersions in OAL

Task::getSuccessors()

```
entry//
//create a collection to be filled with successors
create object instance collection of ReturnCollection;
select many successors related by self->Task[RelPred_Succ.'successors'];
for each successor in successors
relate collection to  successor across RelTask-Collection.'returned objects';
end for;
return collection;
```

getSuccessors in OAL

3.3 Beziehungen zwischen Objekten

Jeder Objekttyp im Informationsmodell des Versionierungssystems definiert Methoden für die Navigation entlang der Beziehungen zwischen Objekten. Für ein konkretes Objekt im Versionierungssystem aufgerufen, liefern diese Methoden die assoziierten Objekte zurück. Es handelt sich dabei um die *direkte Navigation*, wobei der Benutzer über den Aufruf von entsprechenden Zugriffsmethoden direkt vom Ausgangsobjekt zum Zielobjekt gelangen kann. Der Methodenname aufgerufen ausgehend von dem Objekt, von dem aus navigiert wird, nimmt Bezug auf den Rollenname, den das andere Ende der zu navigierenden Beziehung trägt und hat die Form *get<Rollenname>*. Bei einer *indirekten Navigation* erhält der Benutzer zunächst Zugriff auf ein Beziehungsobjekt, von dem er dann auf die Beziehungsenden und damit auf das Zielobjekt zugreifen kann. D. h. bei der indirekten Navigation wird in zwei Schritten vorgegangen. Im ersten Schritt wird die Menge aller Beziehungsinstanzen des angegebenen Typs zurückgeliefert, an denen das im Parameter übergebene Objekt teilnimmt. Im zweiten Schritt kann dann, auf den Resultaten des ersten Schrittes aufbauend, auf die Objekte zugegriffen werden, die durch die Beziehungsinstanzen assoziiert werden.
Es sind auch andere Möglichkeiten bei der Beziehungsspezifikation möglich. Sie beziehen sich auf die *Kardinalität* der Beziehungsenden und in manchen Versionierungssystemen auch auf die erlaubten *Navigationsrichtungen*. Versionierungssysteme können 1:1, 1:n und n:m Beziehungen verwalten. Die Datenverwaltung wird vereinfacht im Fall, dass für eine Beziehung im Informationsmodell nur eine Navigationsrichtung spezifiziert wird. In diesem Fall bietet das Versionierungssystem Zugriffsmethoden nur beim Ausgangsobjekt der Beziehung an.

3.4 Beziehungen zwischen versionierten Objekten

Versionierungssysteme bieten die Möglichkeit, sowohl nicht versionierte als auch versionierte Objekte zu verwalten. Die Handhabung von versionierten Objekten ist komplexer als bei nicht versionierten Objekten, da bei einem Objektzugriff neben der *objectId* auch die entsprechende *verId* angegeben werden muss. Sofern der Zielobjekttyp nicht versioniert ist, ist das Zielobjekt einer Beziehung trivial zu bestimmten. Wenn er aber versioniert ist, können mehrere Objektversionen referenziert werden. Dafür gibt es zwei Lösungsansätze, entweder die Version genau fixieren, dann führt die Navigation immer zu dieser Version, selbst wenn in der Zwischenzeit neue Versionen angelegt wurden, oder aber die Version gar nicht fixieren, sondern dynamisch bestimmen, dann muss der Benutzer selbst entscheiden, welche Version er aus dem Versionsgraph des Zielobjekts verwenden möchte. In dem zuletzt genannten Fall bieten manche Versionierungssysteme Beziehungstypen, für deren Beziehungsenden der Benutzer spezifizieren kann, welche Objektversionen referenziert werden dürfen. Diese Beziehungsenden werden auch als gleitende Beziehungsenden (floating relationship ends) bezeichnet.

3.4.1 Gleitende Beziehungsenden

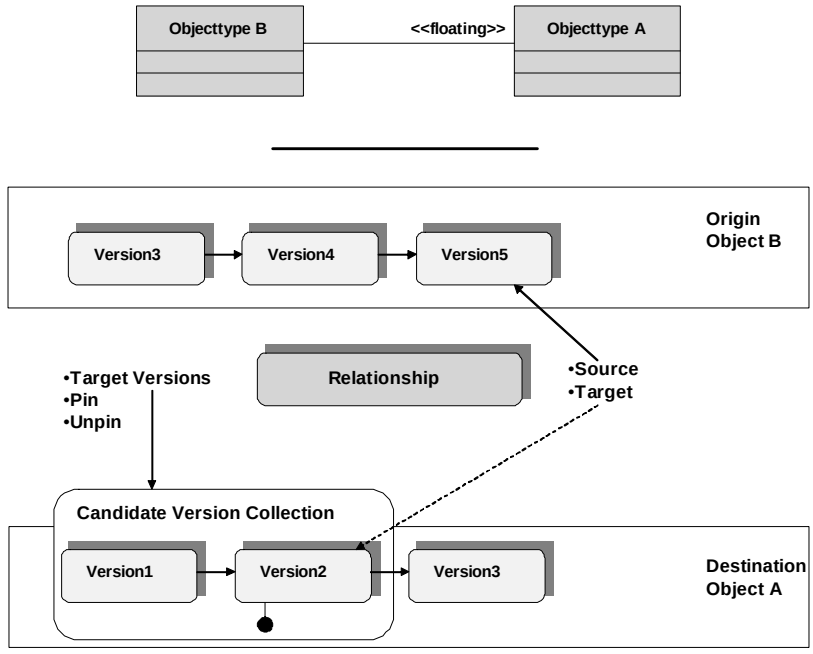
Ein gleitendes Beziehungsende macht nur Sinn, wenn es mit einem versionierten Objekt verbunden ist. Ist beispielsweise wie es in Abbildung 3.8 zu sehen ist ein versioniertes Objekt A mit einem (nicht unbedingt versionierten) Objekt B assoziiert, so ist diese Assoziation von B ausgehend nicht notwendigerweise auf eine bestimmte Version von A beschränkt, sondern sie kann

auch mehrere, potentiell auch alle Versionen von A umfassen. Die Menge der potentiell assoziierten Versionen seitens A wird als *Kandidatenmenge* der Versionen (*Candidate Version Collection, CVC*) bezeichnet. Für den Zugriff auf versionierte Objekte über eine Beziehung mit gleitendem Beziehungsende stehen folgende Varianten zur Verfügung:

- *Pinning* - Der Benutzer kann in der Kandidatenmenge eine *bevorzugte Version* auswählen, die beim Zugriff über die Beziehung zurückgeliefert wird. In Abbildung 3.8 ist eine *gepinnte* Version durch einen ausgefüllten Kreis zu erkennen.
- *Regelbasierte Auswahl* - Ist vom Benutzer keine bevorzugte Version festgelegt worden, kann das System mit Hilfe einer Regel eine Version aus der Kandidatenmenge auswählen und diese zurückliefern. Typischerweise handelt es sich hierbei um die neueste Version des angeforderten Objekts.
- *gesamte Kandidatenmenge* - Der Benutzer kann sich jedoch auch die gesamte Kandidatenmenge zurückliefern lassen und selbst entscheiden, mit welcher Version des Objekts er weiter arbeiten möchte.

Ein gleitendes Beziehungsende verändert nicht die Kardinalität einer Beziehung, da vom Benutzer zur Laufzeit nur eine Version aus der CVC ausgewählt wird. Das ist auch der Fall bei einer 1:1 Beziehung. Es wird ohne weitere Vorgaben des Benutzers nämlich die neueste Version des angeforderten Objekts zurückgeliefert, so dass die 1:1 Kardinalität der Beziehung gewahrt wird. Beim Festlegen einer bevorzugten Version, wird diese nach Abfrage zurückgeliefert.

Abbildung 3.8 Gleitendes Beziehungsende



3.4.2 Manipulation der Kandidatenmenge

Es wird zwischen drei Arten von Beziehungen unterschieden. Sie unterscheiden sich nämlich in der Anzahl der gleitenden Beziehungsenden. Es gibt Beziehungen mit keinem, einem oder zwei gleitenden Beziehungsenden. Die oben erwähnten Methoden, um Beziehungen zwischen Objekten zu manipulieren, sind auch bei gleitenden Beziehungsenden nützlich, allerdings wird nicht direkt über die Beziehung auf ein Beziehungsende zugegriffen, sondern wird zuerst auf die Kandidatenmenge zugegriffen und erst dann auf die verbundene Version des gleitenden Beziehungsendes. Bei Beziehungen, die mindestens ein gleitendes Beziehungsende besitzen, kann durch bestimmte Befehle die entsprechende Kandidatenmenge verändert. Zusätzlich existieren noch weitere Methoden, die in diesem Fall die Manipulation von Beziehungen erlauben. Diese verschiedene Befehle werden nachfolgend erklärt.

- *get<Rollenname>* - Liefert wie es auch bei Beziehungen zwischen unversionierten Objekten der Fall ist alle Objekte, die mit einem bestimmten Objekt in Beziehung stehen, zurück. Falls die Objektypen, die mit diesem Objekttyp in Verbindung stehen, ein gleitendes Beziehungsende auf ihrer Seite der Beziehung besitzen, werden die zuvor vorausgewählten Objektversionen zurückgegeben. Falls innerhalb der entsprechenden Kandidatenmenge keine Objektversion als bevorzugt markiert ist, wird die neueste Objektversion zurückgegeben (Abbildung 3.9).

- *get<Rollenname>Unfiltered* - Dadurch kann der Benutzer sich die gesamte Kandidatenmenge d. h. alle in der Beziehung beteiligten Versionen zurückliefern lassen (Abbildung 3.9).

Abbildung 3.9 *get<Rollenname> und get<Rollenname>Unfiltered in OAL*

```
Task::getRatedCosts()
entry//
//get Cost related to Task
//get candidate version collection related to Task
select one cvc related by self->CostCVC[RelCostCVC-Task.'ratedCosts'];
if not empty cvc
//try to get a pinned version, if there is one
select one pinnedCost related by cvc->Cost[Rel2Cost-CostCVC.'pinned version'];
if not empty pinnedCost
instanceOfCost = pinnedCost;
else
//there is no pinned version, get the latest version of the CVC
select one latestVer related by cvc->Cost[Rel3Cost-CostCVC.'latest version'];
instanceOfCost = latestVer;
end if;
end if;
return instanceOfCost;
```

getRatedCosts in OAL

```
Task::getRatedCostsUnfiltered()
entry//
//create a collection to be filled with connected ratedCosts
create object instance connectedCosts of ReturnCollection;
//get candidate version collection related to Task
select one cvc related by self->CostCVC[RelCostCVC-Task.'ratedCosts'];
//try to get all candidate versions
select many candidateversionCosts related by cvc->Cost[Rel1Cost-CostCVC.'candidate versions'];
for each candidateversion in  candidateversionCosts
relate connectedCosts to  candidateversion across RelCost-Collection.'returned objects';
end for;
return connectedCosts;
```

getRatedCostsUnfiltered in OAL

- *add<Rollenname>* - Erzeugt eine neue Beziehung zwischen zwei Objekten. Bei einem gleitenden Beziehungsende wird die Beziehung mit Einbezug der Kandidatenmenge erstellt. Dabei wird an beiden Beziehungsenden, unabhängig davon, ob es sich um ein gleitendes Beziehungsende handelt oder nicht, die im Informationsmodell spezifizierte maximale Kardinalität überprüft (Abbildung 3.10).
- *remove<Rollenname>* - Löscht eine bestehende Beziehung zwischen zwei Objekten. Dabei wird an beiden Beziehungsenden, die im Informationsmodell spezifizierte minimale Kardinalität überprüft (Abbildung 3.10).

Abbildung 3.10 *add<Rollenname> und remove<Rollenname> in OAL*

Task::addRatedCosts(int globalId)

```
entry//
//Cost has multiplicity 1 in the relation Task-Cost
object = self.getRatedCosts();
if empty object
select one instanceOfCost from instances of Cost
   where selected.globalId == param.globalId;
if not empty instanceOfCost
select one instanceOfCostCVC related by self->CostCVC[RelCostCVC-Task.'ratedCosts']
   where selected.objId == instanceOfCost.objId;
if not empty instanceOfCostCVC
relate instanceOfCost to instanceOfCostCVC across Rel1Cost-CostCVC.'candidate versions';
//set latest version
select one latestVer related by instanceOfCostCVC->Cost[Rel3Cost-CostCVC.'latest version'];
if(instanceOfCost.verId > latestVer.verId)
unrelate latestVer from instanceOfCostCVC across Rel3Cost-CostCVC.'latest version';
relate instanceOfCost to instanceOfCostCVC across Rel3Cost-CostCVC.'latest version';
end if;
else
create object instance newCostCVC of CostCVC;
newCostCVC.objId = instanceOfCost.objId;
relate newCostCVC to self across RelCostCVC-Task.'ratedCosts';
relate instanceOfCost to newCostCVC across Rel1Cost-CostCVC.'candidate versions';
relate instanceOfCost to newCostCVC across Rel3Cost-CostCVC.'latest version';
end if;
//add relation in the other way too
select one instanceOfTaskCVC related by instanceOfCost->TaskCVC[RelTaskCVC-Cost.'causedBy']
   where selected.objId == self.objId;
if not empty instanceOfTaskCVC
relate self to instanceOfTaskCVC across Rel1Task-TaskCVC.'candidate versions';
//set latest version
select one latestVer related by instanceOfTaskCVC->Task[Rel3Task-TaskCVC.'latest version'];
if(self.verId > latestVer.verId)
unrelate latestVer from instanceOfTaskCVC across Rel3Task-TaskCVC.'latest version';
relate self to instanceOfTaskCVC across Rel3Task-TaskCVC.'latest version';
end if;
else
create object instance newTaskCVC of TaskCVC;
newTaskCVC.objId = self.objId;
relate newTaskCVC to instanceOfCost across RelTaskCVC-Cost.'causedBy';
relate self to newTaskCVC across Rel1Task-TaskCVC.'candidate versions';
relate self to newTaskCVC across Rel3Task-TaskCVC.'latest version';
end if;
end if;
end if;
```

addRatedCosts in OAL

Task::removeRatedCosts(int globalId)

```
entry//
select one instanceOfCost from instances of Cost
   where selected.globalId == param.globalId;
if not empty instanceOfCost
select one instanceOfCostCVC related by self->CostCVC[RelCostCVC-Task.'ratedCosts']
   where selected.objId == instanceOfCost.objId;
if not empty instanceOfCostCVC
unrelate instanceOfCost from instanceOfCostCVC across Rel1Cost-CostCVC.'candidate versions';
select one latestVer related by instanceOfCostCVC->Cost[Rel3Cost-CostCVC.'latest version'];
if(instanceOfCost.globalId == latestVer.globalId)
unrelate  instanceOfCost from instanceOfCostCVC across Rel3Cost-CostCVC.'latest version';
//find the new latest version in the actual CVC
select any candidateVersion related by instanceOfCostCVC->Cost[Rel1Cost-CostCVC.'candidate versions'];
if not empty candidateVersion
//instanceOfCostCVC contains other versions
latestVer = candidateVersion;
select many versions related by instanceOfCostCVC->Cost[Rel1Cost-CostCVC.'candidate versions'];
for each version in versions
if(version.verId > latestVer.verId)
latestVer = version;
end if; end for;
relate latestVer to instanceOfCostCVC across Rel3Cost-CostCVC.'latest version';
else
//instanceOfCost was the last version in instanceOfCostCVC
unrelate instanceOfCostCVC from self across RelCostCVC-Task.'ratedCosts';
delete object instance instanceOfCostCVC;
end if; end if; end if;
//remove relation in the other way too
select one instanceOfTaskCVC related by instanceOfCost->TaskCVC[RelTaskCVC-Cost.'causedBy']
   where selected.objId == self.objId;
if not empty instanceOfTaskCVC
unrelate self from instanceOfTaskCVC across Rel1Task-TaskCVC.'candidate versions';
select one latestVer related by instanceOfTaskCVC->Task[Rel3Task-TaskCVC.'latest version'];
if(self.globalId == latestVer.globalId)
unrelate  self from instanceOfTaskCVC across Rel3Task-TaskCVC.'latest version';
//find the new latest version in the actual CVC
select any candidateVersion related by instanceOfTaskCVC->Task[Rel1Task-TaskCVC.'candidate versions'];
if not empty candidateVersion
//instanceOfTaskCVC contains other versions
latestVer = candidateVersion;
select many versions related by instanceOfTaskCVC->Task[Rel1Task-TaskCVC.'candidate versions'];
for each version in versions
if(version.verId > latestVer.verId)
latestVer = version;
end if; end for;
relate latestVer to instanceOfTaskCVC across Rel3Task-TaskCVC.'latest version';
else
//instanceOfTask was the last version in instanceOfTaskCVC
unrelate instanceOfTaskCVC from instanceOfCost across RelTaskCVC-Cost.'causedBy';
delete object instance instanceOfTaskCVC;
   end if; end if; end if; end if;
```

removeRatedCosts in OAL

- *pin<Rollenname>* - Markiert eine Version innerhalb der Kandidatenmenge als bevorzugte Version, die dann bei der Navigation über diese Beziehung genutzt wird (Abbildung 3.11).
- *unpin<Rollennamename>* - Hebt die Markierung einer bevorzugten Version innerhalb der entsprechenden Kandidatenmenge auf (Abbildung 3.11).

3.5 Propagierung von Zuständsänderungen

Bei der Anwendung von Operationen wie z. B. *freeze* oder *createSuccessor* auf Objekten im Versionierungssystem kann es sinnvoll sein, diese Operationen über Beziehungen propagieren zu lassen, d. h. auf den assoziierten Objekten automatisiert dieselben Operationen auch durchzuführen. Im Falle des Informationsmodells aus Abbildung 3.3 könnte beispielsweise beim Anlegen einer neuen Version eines *Offers* automatisch eine neue Version des zugehörigen *Tasks* angelegt

werden, da dieser voraussichtlich ebenfalls geändert werden muss. Die Propagierung von Operationen wird bei der Definition von Beziehungstypen im Informationsmodell des Versionierungssystems festgelegt und ist wie es im Metamodell in der Abbildung 3.2 zu sehen ist eine Eigenschaft der Beziehungsenden.

Abbildung 3.11 *pin<Rollenname> und unpin<Rollenname> in OAL*

```
Task::pinRatedCosts(int globalId)

entry//
select one instanceOfCost from instances of Cost
   where selected.globalId == param.globalId;
if not empty instanceOfCost
select one instanceOfCostCVC related by self->CostCVC[RelCostCVC-Task.'ratedCosts']
   where selected.objId == instanceOfCost.objId;
if not empty instanceOfCostCVC
//pin only one version
select one pinnedVersion related by instanceOfCostCVC->Cost[Rel2Cost-CostCVC.'pinned version']
unrelate pinnedVersion from instanceOfCostCVC across Rel2Cost-CostCVC.'pinned version';
relate instanceOfCost to instanceOfCostCVC across Rel2Cost-CostCVC.'pinned version';
end if;
end if;
```

pinRatedCosts in OAL

```
Task::unpinRatedCosts(int globalId)

entry//
select one instanceOfCost from instances of Cost
   where selected.globalId == param.globalId;
if not empty instanceOfCost
select one instanceOfCostCVC related by self->CostCVC[RelCostCVC-Task.'ratedCosts']
where selected.objId == instanceOfCost.objId;
if not empty instanceOfCostCVC
unrelate instanceOfCost from instanceOfCostCVC across Rel2Cost-CostCVC.'pinned version';
end if;
end if;
```

unpinRatedCosts in OAL

3.6 Arbeitskontexte

Ein *Arbeitskontext* (Workspace) ist *"ein versioniertes Objekt, das andere Objekte aggregiert, die beispielsweise zum Zuständigkeitsbereich eines bestimmten Benutzers gehören"* [Kat90]. So können Objekte mit Hilfe von Workspaceobjekten zu Arbeitskontexten zusammengefasst werden. Das Beispiel aus Abbildung 3.3 enthält auch einen Arbeitskontext, nämlich *Project*. Die Instanzen dieses Typs enthalten jeweils die *ProjectOffers*, die dort enthaltenen *Tasks* und die damit verbundenen *Costs*. Das Versionierungssystem stellt die Methoden *attachTo<Arbeitskontextname>* und *detachFrom<Arbeitskontextname>* zur Verfügung, mit denen Versionen in den Arbeitskontext eingefügt bzw. entfernt werden können. Abbildung 3.12 zeigt die OAL-Darstellung dieser Operationen. Zu einem Arbeitskontext kann immer nur eine Version eines Objekts hinzugefügt werden. Diese Tatsache kann genutzt werden, um einen versionsfreien Zugriff auf die Objekte zu ermöglichen. So kann bei einem Zugriff auf ein Objekt innerhalb eines Arbeitskontextes auf die Angabe der *verId* verzichtet werden. Dies vereinfacht die Handhabung der Objekte innerhalb von Arbeitskontexten. In einem Arbeitskontext sieht der Benutzer bei einer Suche nur die eingefügten Objektversionen und gelangt bei der Verfolgung einer Beziehung automatisch nur zu einer Version des Zielobjektes, die ebenfalls im gleichen Arbeitskontext liegt. Dabei werden das Pinning bzw. die sonstigen Auswahlregeln einer Version nicht beachtet. Der Benutzer braucht bei der Suche keine Angabe einer *verId* mehr, da es aus seiner Sicht innerhalb des Arbeitskontextes, in dem er sich befindet, keinen Unterschied zwischen versionierten und unversionierten Objekten gibt.

Abbildung 3.12 *attachTo<Arbeitskontextname> und detachFrom<Arbeitskontextname> in OAL*

```
Task::attachToProject(int workspaceGlobalId)
//Only if any workspace is defined
select one workspace from instances of Project
   where selected.globalId == param.workspaceGlobalId;
if not empty workspace
//try to attach only one version of an instance to the workspace
select one instanceOfTask related by workspace->Task[RelProject-Task.'Task']
   where selected.objId == self.objId;
if not empty instanceOfTask
instanceOfTask.detachFromProject(param.workspaceGlobalId);
end if;
relate self to workspace across RelProject-Task.'Task';
end if;
```

attachToProject in OAL

```
Task::detachFromProject(int workspaceGlobalId)
//Only if any workspace is defined
select one workspace from instances of Project
   where selected.globalId == param.workspaceGlobalId;
if not empty workspace
unrelate self from workspace across RelProject-Task.'Task';
end if;
```

detachFromProject in OAL

3.7 Schutzmechanismus in Versionierungssystemen

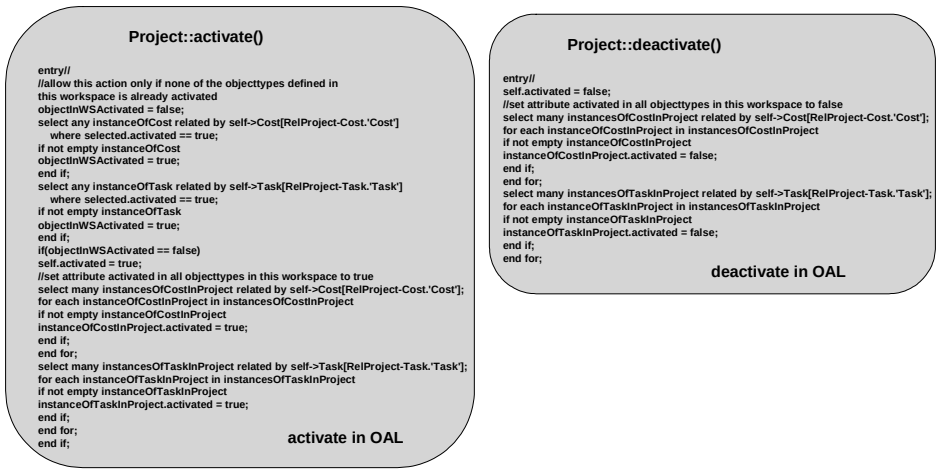
Versionierungssysteme bieten einen Schutzmechanismus für langanhaltende Arbeitsvorgänge. Um diesen Schutzmechanismus nutzen zu können, müssen sich die zu schützenden Objekte in einem Arbeitskontext befinden. Dabei kommen die *activate*- und *deactivate*-Operationen zum Einsatz, die die Objekte im Arbeitskontext als gesperrt bzw. ungesperrt markieren. Um die Sperre auf eine Objektversion zu aktivieren, wird die Methode *activate* auf dem Arbeitskontext, in dem diese Objektversion sich befindet, aufgerufen. Die so gesperrte Objektversion kann zwar noch von allen Benutzern gelesen werden, aber nur noch innerhalb des Arbeitskontextes manipuliert werden, von dem aus die Sperre aktiviert wurde. Falls andere Benutzer das Objekt gleichzeitig verändern wollen, müssen sie mittels *createSuccessor* eine alternative Version erzeugen und bearbeiten. Die parallelen Änderungen können dann zu einem späteren Zeitpunkt mittels *merge* kontrolliert zusammengeführt werden. Durch die Methode *deactivate* wird dieser Schutz wieder aufgehoben. Abbildung 3.13 zeigt Die OAL-Beschreibung der Operationen *activate* und *deactivate*.

3.8 Versionierungssysteme als Produktlinie

Nachdem in den letzten Abschnitten die Domäne der Versionierungssysteme präsentieret wurden, werden in diesem Abschnitt einige mögliche Variationspunkte erläutert, um zu motivieren, warum Versionierungssysteme als Produktlinie oder auch als Systemfamilie realisiert werden können. Die Eigenschaften eines Versionierungssystems können in seinem Informationsmodell spezifiziert werden. Aus diesem Grund unterscheiden sich Versionierungssysteme in vielen möglichen Punkten. Unterschiede zeigen sich beispielsweise bei der genauen Gestalt der zu speichernden Daten. Der Benutzer kann nämlich mehrere Objekttypen definieren und dann

auswählen, welche von denen versionierbar sind und welche nicht. Unterschiede gibt es auch noch bei den Anforderungen sowohl an die Versionsverwaltung als auch an die Beziehungssemantiken. So kann der Benutzer beispielsweise auswählen, ob ein Beziehungsende gleitend definiert werden soll oder auch, ob bestimmte Operationen propagieren sollen. Die strukturellen Ähnlichkeiten zwischen Versionierungssystemen sind jedoch so groß, dass es möglich ist, Versionierungssysteme nach dem Ansatz der generativen Programmierung aus einer Menge von Basiskomponenten zu erstellen und als Systemfamilie zu realisieren. Technische Gemeinsamkeiten bestehen nämlich bei der Unterstützung versionierter Speicherung, dem Konzept der gleitenden Beziehungsenden und der Arbeitskontexte sowie den Methoden für die Datenverwaltung und den navigierenden Zugriff. Diese strukturellen Gemeinsamkeiten erlauben es, Versionierungssyteme als Mitglieder einer Systemfamilie zu realisieren und die möglichen Variabilitäten dienen dazu, verschiedene Varianten in der Systemfamilie zu spezifizieren.

Abbildung 3.13 *activate und deactivate in OAL*



3.9 Fazit

In diesem Kapitel wurde die Domäne der Versionierungssysteme vorgestellt. Bei den Versionierungssystemen entstehen viele Variationsmöglichkeiten. Diese Variationsmöglichkeiten umfassen Angaben über versioniert und unversioniert zu speichernde Objekttypen, über die Beziehungen, die zwischen Objekten bestehen können, über die Typen von Arbeitskontexten, die angelegt werden können und über die Propagierung von Operationen über Beziehungen. *MTFLOW* berücksichtigt diese Variationsmöglichkeiten und gibt dem Benutzer die Möglichkeit das Modell seines erwünschten Versionierungssystems einfach zu gestalten. Dabei werden die Versionierungssemantiken mittels variabler, konfigurierbarer Transformationen in UML-Modelle integriert. Diese Transformationen basieren sich auf die von MDA vorgeschriebenen PIM zu PIM Transformationen und werden durch XMI realisiert. Die Möglichkeiten der Modelltransformationen mittels XMI werden im nächsten Kapitel erläutert.

Kapitel 4 Modelltransformationen

Das Vorgehensmodell der MDA sieht vor, dass zunächst ein plattform-unabhängiges Model (PIM) entwickelt wird. Das geschieht durch mehrfache Verfeinerung mit Hilfe von Abbildungen. Anschließend kann das PIM-Modell auf mehrere plattform-abhängige Modell (PSM) abgebildet werden und diese weiter verfeinert. Zwischen den verschiedenen Modellen werden Modelltransformationen angeboten, die dazu dienen, möglichst große Teile eines Zielmodells aus dem Quellmodell automatisch oder interaktiv zu erzeugen. Diese Modelltransformationen spielen bei MDA eine fundamentale Rolle. Es werden nicht nur PIMs in PSMs transformiert, sondern auch PSMs in PSMs und PIMs in PIMs und dies Hauptsächlich zur Erweiterung oder Verfeinerung von Modellen. In diesem Kapitel werden zunächst die verschiedenen Ansätze der Modelltransformationen vorgestellt. Anschließend wird die XML Metadata Interchange (XMI) als Beschreibungssprache für Transformationen näher betrachtet.

4.1 Klassifizierung der Modelltransformationsansätze

Czarnecki und Helson unterscheiden zwischen Modell-zu-Code und Modell-zu-Modell Ansätzen [CH03]. Im Folgenden betrachten wir nur die letzte Variante der Transformationen. Modelltransformationen sollen ein gegebenes Modell in ein anderes überführen. Sie bestehen aus einer Menge von Regeln und Techniken zur Abbildung eines gegebenen Modells in ein anderes Modell. Es wird bei der Modell-zu-Modell Kategorie zwischen mehreren Ansätzen unterschieden, auf die hier nur kurz eingegangen wird.

4.1.1 Ansätze zur direkten Manipulation

Diese Ansätze zur direkten Manipulation stellen dem Benutzer eine interne Darstellung eines Modells und eine API zum Manipulieren dieses Modells zur Verfügung. Der Benutzer soll aber die Transformationsregeln mit Hilfe einer Programmiersprache wie Java selbst implementieren. Diese Ansätze, die üblicherweise als ein objektorientiertes Framework implementiert werden, sollen nämlich nur eine minimale Infrastruktur zur Organisation der Transformationen bieten (z. B. eine abstrakte Klasse für Transformationen). Diese Kategorie beinhaltet beispielsweise Jamda [JAM03], das als ein open-source Framework beschrieben wurde, das dabei helfen soll, einen Generator zu erstellen, der fähig ist, Java-Code aus einem gegebenen Modell zu erzeugen.

4.1.2 Relationale Ansätze

Die Kategorie der relationalen Ansätze umfasst deklarative Ansätze mit mathematischen Beziehungen zwischen Modellen als Hauptkonzept. Die Idee dieser Ansätze besteht darin, den Quell- und den Ziel-Elementtyp einer Beziehung festzulegen und diese Beziehung an Hand von Bedingungen zu spezifizieren. Diese Bedingungen sollen dabei, eine frühzeitige Definition von Bedingungen im Modell zu gestatten, die durch die Anwendung zu realisieren sind. Ein Beispiel für solche Ansätze ist die Transformationssprache QVT (Queries Views Transformation) [QVT02]. Die Sprache QVT ist die von der OMG vorgesehene Beschreibungssprache für Modelltransformationen. Abbildung 4.1 zeigt einen QVT-Abschnitt. Die Einzelnen Bestandteile der QVT erklären sich folgendermaßen [QVT03]:

- *Queries* - Eine Auswahl spezifischer Elemente aus einem Eingangsmodell.
- *Views* - Modelle, die von anderen Modellen abgeleitet sind.
- *Tranformations* - Spezifikationen oder Implementierungen, die als Eingang ein Modell haben und dieses auf ein anderes Modell beziehen oder im Bezug darauf ein neues Modell erstellen.

Abbildung 4.1 QVT Queries/Views/Transformation

```
Transformation uml2java (source, Target, Tags) {
        ...
        TRule umlClassifierToJavaClass (X, Y) {
                X : UMLClassifier [name ->N]
                ->
                Y : JavaClass[name -> N] and
                track (Y, java_class_from_uml_classifier, X)
        }
        ...
}
```

4.1.3 Graphtransformation-basierte Ansätze

Diese Kategorie basiert auf Transformationen zwischen Graphen, die speziell ausgelegt wurden, um UML-ähnliche Modelle darzustellen. Zu dieser Kategorie gehört z. B. BOTL (Bidirectional Object-oriented Transformation Language) [BM03]. Diese Methode soll graphisch beschreiben, wie Modelle ineinander zu überführen sind. Mit Hilfe von BOTL ist es möglich, Transformationen von Objektmodellen mit Hilfe einer graphischen, an die UML angelehnten Sprache zu definieren. So lassen sich graphisch eine Reihe von Regeln definieren in denen festgelegt wird, wie einzelne Ausschnitte von Objektmodellen zu transformieren sind.

4.1.4 Strukturgetriebene Ansätze

Ansätze, die zu dieser Kategorie gehören, besitzen zwei ausgeprägte Phasen: die erste Phase befasst sich hauptsächlich mit dem Anlegen der hierarchischen Struktur des Ziel-Modells. Die zweite Phase legt die Attribute und die Referenzen im Ziel-Modell fest. Der Benutzer gibt dabei nur die Transformationsregeln an. Ein Beispiel solcher Anätze ist OptimalJ [OPT03]. OptimalJ bietet ein Framework an, das dem Benutzer die einfache Möglichkeit geben soll, seine gewünschten Tranformationsregeln zu implementieren.

4.1.5 Hybride Ansätze (Hybrid Approaches)

Diese Ansätze kombinieren verschiedene Techniken der oben genannten Kategorien. XDE (E**x**tended **D**eveloper **E**xperience) z. B. fällt unter diese Kategorie [XDE03]. XDE bietet eine vollständige Pattern Engine, die den Softwareentwurf und die Modelltransformation unterstützen soll.

4.2 Modelltransformation mit XSLT

Zur Verarbeitung von XML-Dokumenten hat das W3C die eXtensible Stylesheet Language (XSL) definiert [XSL99]. Eine wichtige Komponente von XSL ist XSLT (XSL-Transformation), deren Hauptfunktion ist, jedes beliebige XML-Format in jedes beliebige textbasierende Ausgabeformat umzuwandeln. Dieses Ausgabeformat kann XML, HTML und sonstiger strukturierter oder unstrukturierter Text sein. XSLT stellt eine Sprache dar, die die Struktur eines XML Dokumentes transformieren kann. Sie unterstützt im Zuge einer Transformation konditionelle und iterative Konstruktionen und kann auch einfache und Zeichenketten-basierende Funktionen durchführen. Eine Transformation in XSLT besteht aus Transformationsregeln. Jede Regel passt auf bestimmte XML-Elemente der Eingabedatei und gibt an, wie das XML-Element für die Ausgabedatei transformiert werden soll. Zur Transformation mit Hilfe von XSLT werden drei wichtige Komponenten benötigt, die in Abbildung 4.2 zu erkennen sind:

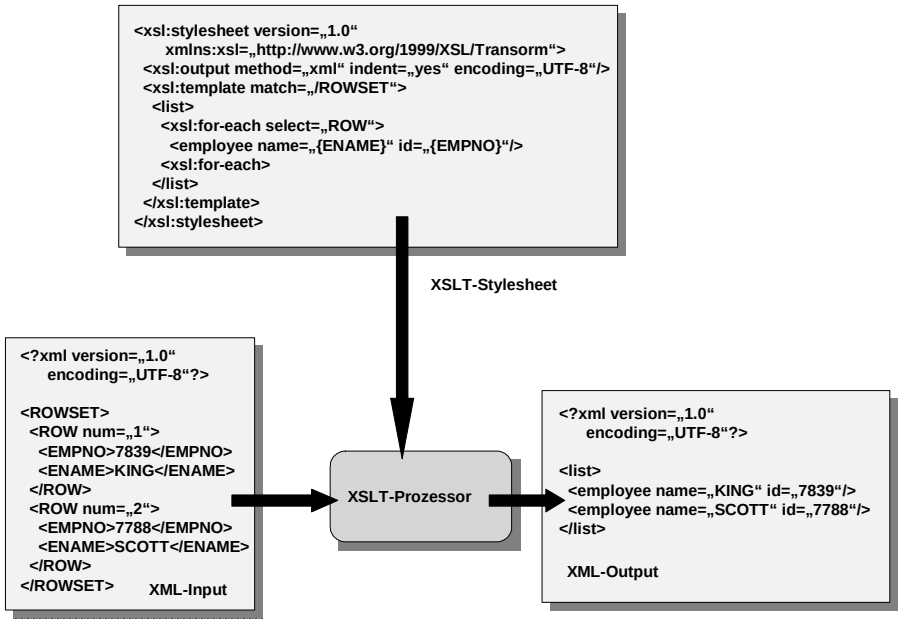
- Die XML-Datei bzw. XML-Daten bilden die Datenquelle für die Transformation.
- Das XSLT-Stylesheet beinhaltet die Transformationsregeln und schreibt vor, wie die Bestandteile der XML-Datei in das Zielformat zu überführen sind.
- Der Stylesheet-Prozessor führt den eigentlichen Transformationsprozess durch. Er wendet die Regeln des Stylesheets auf die XML-Daten an und generiert die Zieldatei. Ein bekanntes Produkt für diesen Zweck ist der Prozessor XALAN der Apache Group.

In [CH03] wird XSLT auch als einen weiteren Ansatz der Modelltransformation vorgeschlagen. Die Modelle werden nämlich sehr oft als XMI (XML Metadata Interchange)-Dokumente gespeichert bzw. ausgetauscht und XSLT hat den Vorteil, dass nach Transformation die Stylesheets wieder in XML-Format vorliegen.

4.3 Modelltransformation mit XMI

Modelle haben in der MDA eine große Bedeutung. Es gibt eine Menge von Modelltypen, die für die Modelltransformation in MDA unterschiedliche Relevanz haben. Transformation erfolgt aber nicht nur zwischen Modellen sondern auch zwischen Metamodellen, die in MDA alle auf der MOF (Meta Object Facility) als Standardsprache zur Beschreibung von Metamodellen basieren. Ohne die MOF können Modelltransformationen nicht hinreichend genau definiert werden und wären damit viel schwieriger bzw. überhaupt nicht zu realisieren. MOF ist sehr eng mit dem Metamodellierungsansatz der OMG verknüpft. Im Folgenden werden zuerst die vier Modellschichten der OMG vorgestellt, dann wird die Beziehung zwischen dieser Architektur und XMI näher betrachtet und letztendlich wird untersucht, warum und wie XMI bei den Modelltransformationen eingesetzt werden können.

Abbildung 4.2 XSLT-Beispiel



4.3.1 Modellschichten der OMG

Mit den Modellschichten der OMG wird eine Modellhierarchie beschrieben, die die verschiedenen Modellebenen definiert. Abbildung 4.3 verdeutlicht die Abhängigkeit der verschiedenen Schichten graphisch. Die Schichtenarchitektur besteht aus folgenden Elementen:

M3: Meta-Metamodell

Dies ist die Infrastruktur der Metamodellarchitektur und sie definiert die Sprache zur Spezifikation von Metamodellen (z. B. Metaklasse, MetaAttribut, MetaOperation). Auf dieser Ebene ist beispielsweise MOF angesiedelt. Metamodelle der UML oder des CWM sind Instanzen der MOF.

M2: Metamodelle

Ein Metamodell ist eine Instanz des Meta-Metamodells und definiert die Sprache zur Beschreibung der Modelle (z. B. Klasse, Attribut, Operation). Diese Schicht ist das zentrale Element der UML und die Konzepte werden bei der UML-Modellierung verwendet. Zudem werden auf dieser Ebene die Ergänzungen für die unterschiedlichen Plattformprofile spezifiziert.

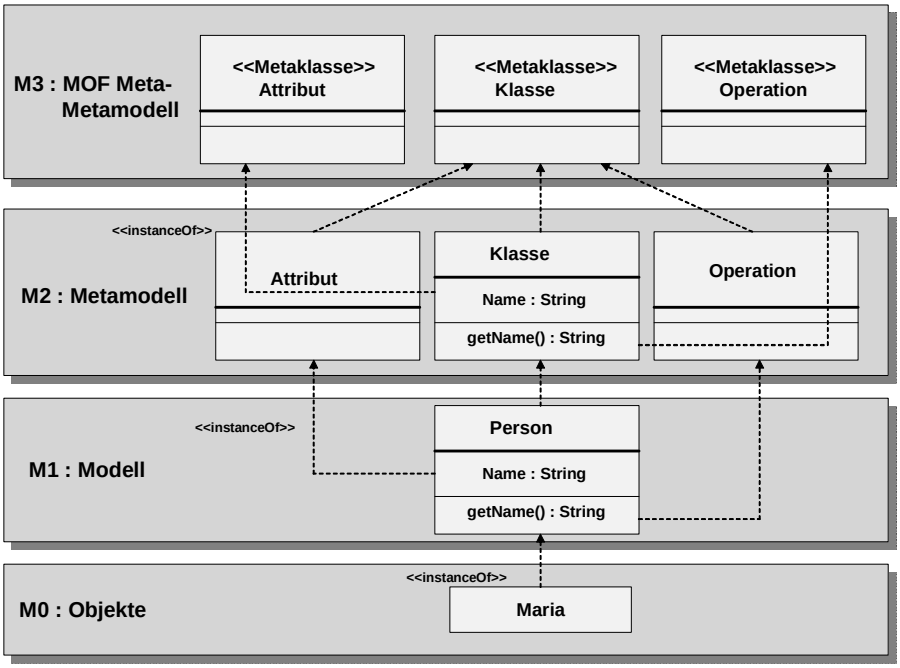
M1: Modelle

Ein Modell ist eine Instanz des Metamodells und definiert die sprache zur Beschreibung der Domäne (z. B. Klasse:Person, Operation der Klasse Buch: getName). Zu den Modellen gehören die bekannten UML-Modelle.

M0: Objekte

Ein Objekt ist ist eine Instanz des Modells und beschreiben die Ausprägungen einer bestimmten Domäne, wie z. B. den Namen einer Instanz der Klasse *Person*: „Maria“.

Abbildung 4.3 Vier Schichten Metamodell Architektur



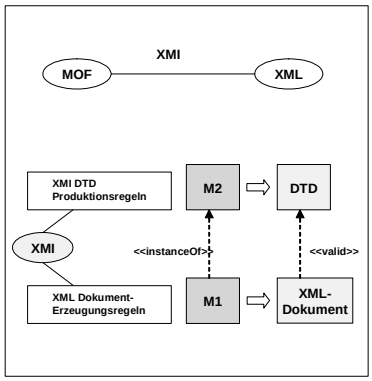
Abbidung 4.3 zeigt ein Beispiel, wie alle vier Schichten zusammenwirken. Die Klasse *Person* (M1) ist eine Instanz der Metaklasse *UML Class*. Das Objekt *Maria* (M0) ist wiederum eine Instanz der klasse *Person* aus der darüberliegenden Schicht. Abbildung 4.4 zeigt den Zusammenhang zwischen dieser Architektur und XMI. Die XMI-Spezifikation beschreibt im wesentlichen, wie aus einem MOF-basierten Meta-Modell automatisch eine XML Document Type Definition (DTD) und wie mit Hilfe dieser DTD Elemente des Modells in ein XML-Dokument abgebildet werden können. Speziell für das MOF-Modell und das Metamodell von UML sind in der XMI-Spezifikation vorgefertigte DTDs enthalten. Die XMI-Spezifikation enthält zudem mehrere Konstrukte, die zu der Modelltransformation beitragen können. Diese Konstrukte werden im nächsten Abschnitt näher betrachtet.

4.3.2 XML Metadata Interchange (XMI)

XMI ist ein textbasiertes Austauschformat für Metadaten (Modelle) und Metamodelle, die auf MOF basieren. Das Hauptziel von XMI ist es, den einheitlichen und einfachen Austausch von Metadaten zwischen Modellierungs-Werkzeugen (basierend auf OMG-UML) und Metadaten-Repositories (basierend auf OMG-MOF) in verteilten heterogenen Umgebungen zu ermöglichen. Die Abbildung 4.5 zeigt ein Beispiel möglicher Partner, zwischen denen XMI als Austauschformat eingesetzt werden kann.

Abbildung 4.4 Beziehung zwischen der Metamodell-Architektur und XMI

Meta-Ebene		Beispiele	DTD/XMI
M3	Meta-Metamodell	Das MOF Modell (z. B. eine abstrakte Syntax zum Definieren von Metamodellen)	MOF DTD
M2	Metamodell, Meta-Metadaten	UML Metamodell, CWM Metamodell	UML DTD
M1	Modell, Metadaten	UML Modelle, Warehouse Schemas	UML Modelle als XMI Dokumente
M0	Objekte, Daten	UML Instanzen, Warehouse Datenbanken	



Aufbau eines XMI-Dokumentes

Jede XMI-konforme DTD definiert zunächst ein XML-Element <XMI.header>, der für die Übertragung allgemeiner Informationen über das zugrundeliegende MOF-Modell oder dem Generator des Dokumentes dient. Der eigentliche Inhalt befindet sich dann in den XML-Elementen <XMI.content>, <XMI.difference> und <XMI.extensions>. Diese bilden die Top-Level Abschnitte eines XMI-Dokumentes. Abbidung 4.6 enthält einen Auszug aus der XMI-DTD, der die oben genannten XML-Elemente enthält und das darauf bezogene XMI-Dokument.

Abbildung 4.5 Austausch mit XMI

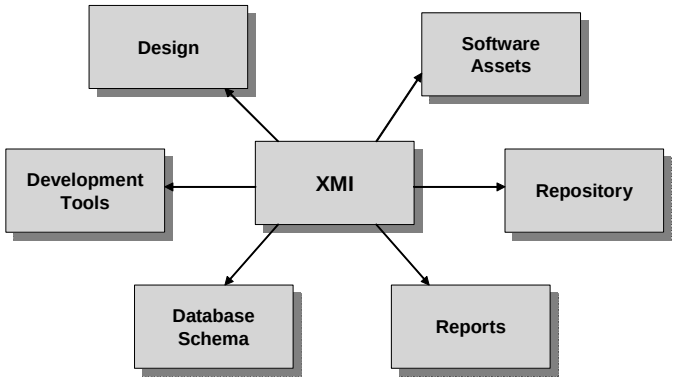
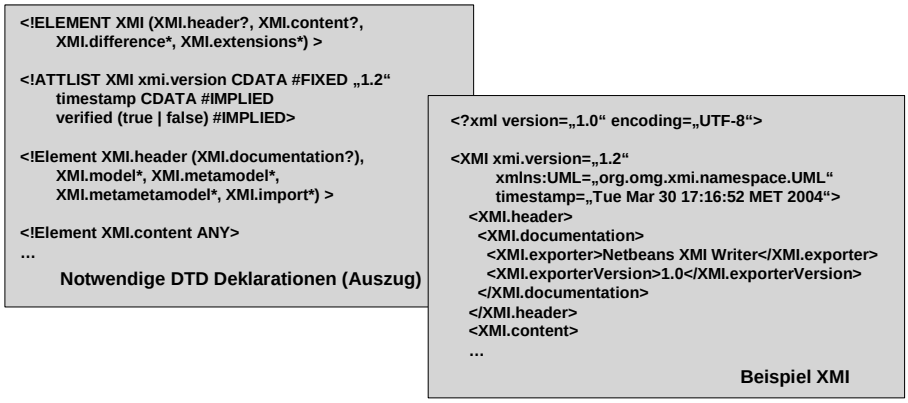


Abbildung 4.6 Aufbau eines XMI-Dokumentes



Im <XMI.content> Abschnitt befinden sich die eigentlichen zu übertragenden Objekte, wie es von allen beteiligten Nutzern verstanden wird. Der Inhalt (also die enthaltenen XML-Elemente) wird durch das zugrundeliegende MOF-Modell bestimmt. Informationen, die nicht von allen Teilnehmern verstanden werden können, werden in einem <XMI.extensions> Abschnitt gepackt. Jeder Teilnehmer kann sich seinen eigenen Abschnitt definieren und mit Hilfe des XML-Attributes xmi.extender individuell kennzeichnen. Diese Abschnitte werden von anderen Teilnehmern in der Regel ignoriert. Die <XMI.difference> Abschnitte dienen dazu, Differenzinformationen zu übertragen, wie etwa das Löschen oder Hinzufügen von Objekten zu einem bekannten Datenbestand. Ein <XMI.difference> Abschnitt kann sich auf den vorherigen <XMI.content> Abschnitt, auf andere <XMI.difference> Abschnitte bzw. auf Abschnitte in anderen XMI-Dokumenten beziehen. Alle <XMI.difference> Abschnitte müssen in der Reihenfolge ihres Auftretens interpretiert werden, um die Differenz auf die ursprünglichen Daten korrekt anzuwenden. XMI wird ausschließlich für die Übertragung von Kompletten Datenbeständen benutzt, <XMI.difference> Abschnitte tauchen nur selten bzw. gar nicht auf. Diese Abschnitte spielen dagegen in MTFLOW eine große Rolle. Sie erlauben nämlich nur Teile eines Modells zu definieren und tragen zu den Modelltransformationen bei. Die <XMI.difference> Abschnitte werden später an Hand von Beispielen näher betrachtet.

XMI enthält auch Standardmechanismen, um Objekte zu identifizieren oder auch Objekte innerhalb eines Dokumentes oder zwischen verschiedenen Dokumenten zu referenzieren. *xmi.id* und *xmi.uuid* dienen dazu, Modellelemente in XMI-Dateien zu identifizieren. Das *xmi.id* Element Identification Attribute ist für jedes XMI Element zu besetzen. Es dient der Referenzierung durch andere XML Elemente über die XML idrefs Beziehung. *xmi.id* Werte müssen innerhalb eines XMI Dokumentes eindeutig sein. Das *xmi.uuid* Element wird als Universally Unique Identifier bezeichnet. Es ist eine globale eindeutige 128 bit umfassende Identifizierungsmarke. *xmi.idref* und *href* dienen der Referenzierung von Modellelementen in XMI-Dateien. Bei *xmi.idref* erfolgt die Referenzierung innerhalb einer Datei. Dagegen handelt es sich bei *href* um eine dateiübergreifende Referenzierung. Mit Hilfe von *href* besteht die Möglichkeit ausgehend von einem XMI-Dokument auf Elemente eines anderen XMI-Dokumentes zuzugreifen.

XMI benutzt die Namen von Konzepten des zugrundeliegenden MOF-Modells (das sind im wesentlichen die strukturelle Konzepte *Package*, *Class*, *Attribute*, *Association*), um daraus

XML-Elementnamen zu generieren, die eine Zuordnung von Textbausteinen des XMI-Dokumentes zu den Konzepten des zugrundeliegenden Modells ermöglichen. UML-Modelle, wie es auch bei allen MOF-basierten Modellen der Fall ist, bestehen im wesentlichen aus Klassen die, in Paketen strukturiert sein können, deren Objekte Attribute haben können mit Datentypen, die entweder einfache Basistypen oder Klassen des Modells sind und deren Objekte durch zweistellige Assoziationen miteinander verbunden sein können. XMI stellt bereit eine DTD für UML-Metamodelle bzw. eine Beschreibungssprache für UML-Modelle zur Verfügung. Diese Beschreibungssprache ist auf Basis der oben genannten Konzepte aufgebaut. Da MTFLOW hauptsächlich die Transformation zwischen UML-Modellen unterstützt, wird In dem nächsten Abschnitt die XMI-Darstellung von UML-Modellen näher betrachtet.

XMI-Darstellung von UML-Modellen

Speziell für das UML-Metamodell sind in der XMI-Spezifikation vorgefertigte DTDs enthalten. Die nächsten Abbildungen zeigen, wie die verschiedenen Konstrukte eines UML-Modells textuell mit XMI dargestellt werden können. In einem XMI-Dokument ist das XMI-Element *UML:Model* das oberste Element bei der Definition eines UML-Modells. Für Jedes Modell „M“ wird ein XMI-Element <*UML-Model*> erzeugt, dem als XMI-Unterstruktur alle XMI-Elemente zugeordnet werden, die entweder aus Paketen oder Klassen oder Assoziationen enstanden sind (Abbildung 4.7). Für Jede Klasse „C“ wird ein XMI-Element <UML-Class> erstellt. Dieses XMI-Element enthält als Unterstruktur die XMI-Elemente aller in dieser Klasse enthaltenden Attribute und Operationen. In Abbildung 4.8 ist beispielhaft die Klasse *Task* mit dem Attribut *name* dargestellt. Wie es in Abbildung 4.9 zu erkennen ist, werden für jede binäre Assoziation zwei XMI-Elemente erzeugt, ein Element für jedes Assoziationsende. Die Namen dieser XMI-Elemente bilden sich aus den jeweiligen Rollennamen. Jedem dieser XMI-Elemente werden als Unterstruktur die XMI-Elemente der jeweils gegenüberliegenden Klasse und der Kardinalität des entsprechenden Assoziationsendes zugeordnet. In MTFLOW werden die Operationen mit Hilfe von OAL definiert. Für die verschiedenen OAL-Anweisungen haben wir XMI-Elemente definiert. Aus diesem Grund wurden die Operationen in den Beispielen nicht berücksichtigt. Die XMI-Darstellungen der OAL-Anweisungen werden im nächsten Abschnitt vorgestellt.

Abbildung 4.7 Darstellung des Modells

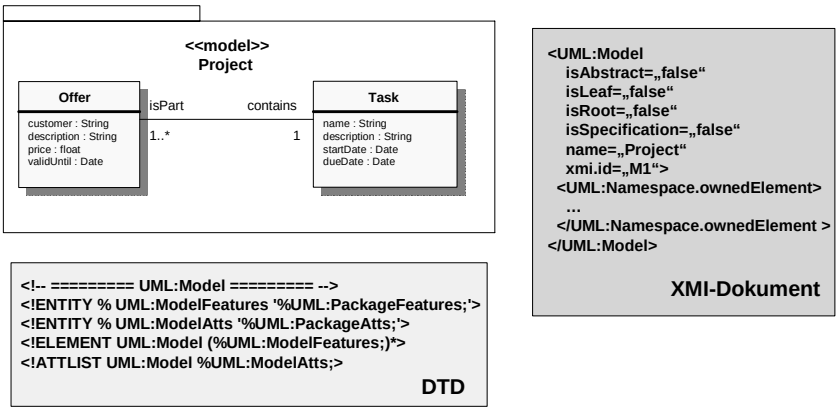


Abbildung 4.8 Darstellung der Klassen

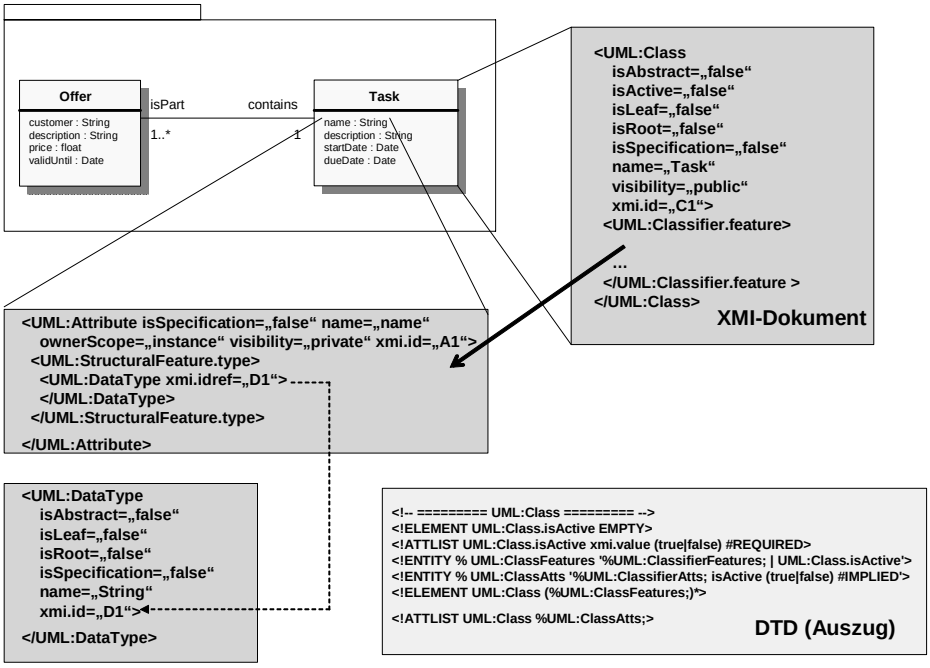
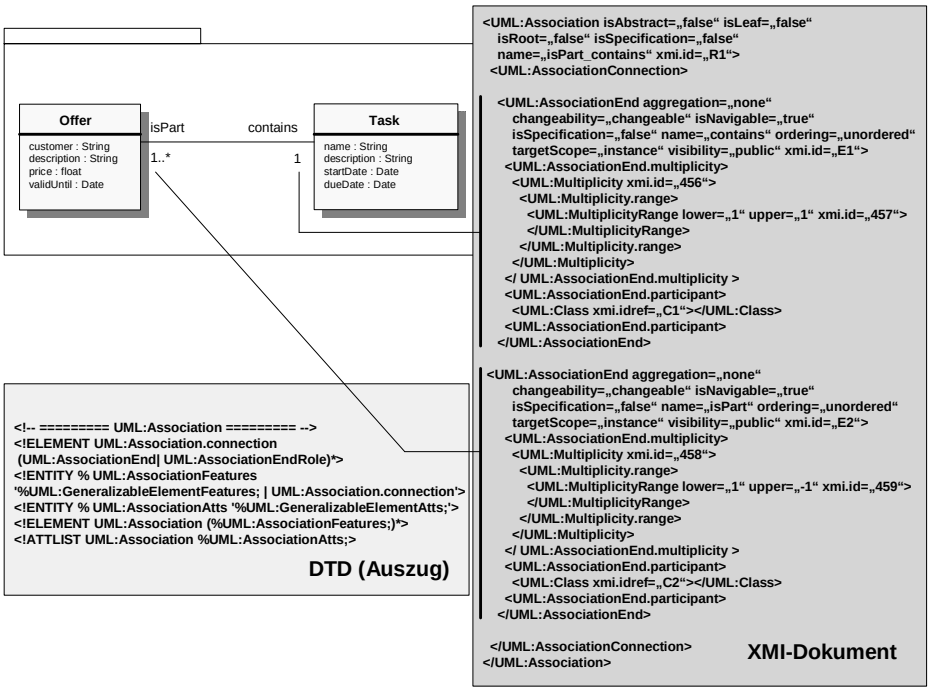


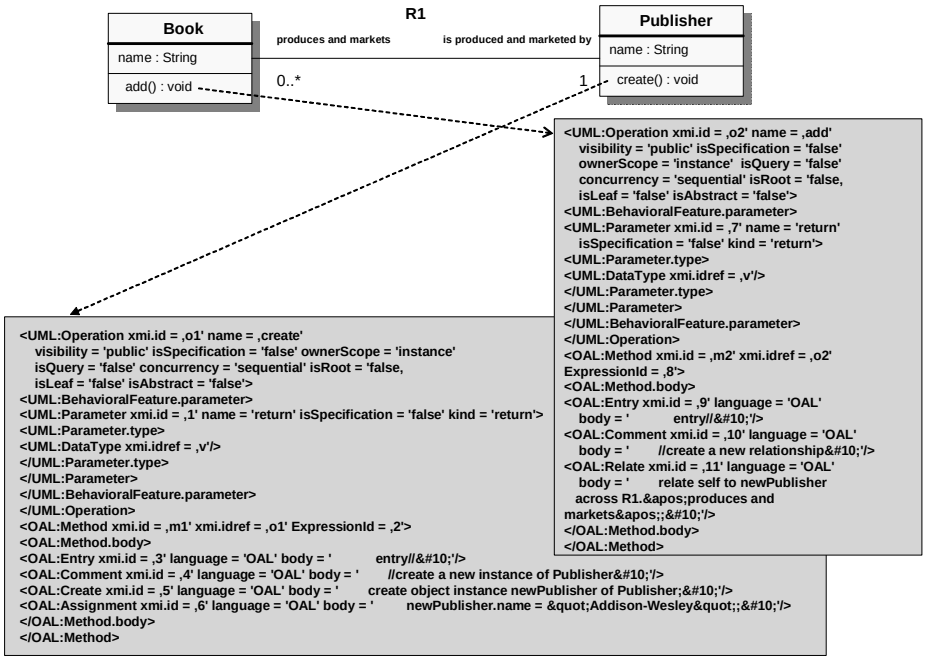
Abbildung 4.9 Darstellung der Assoziationen



Erweiterung der XMI-Spezifikation

Wie erwähnt werden die Operationen in MTFLOW mit OAL beschrieben. Die XMI-Spezifikation soll daher erweitert werden mit neuen XMI-Elementen, die eine XMI-basierte Beschreibung der Aktionen erlauben. Wir haben dafür 20 zusätzliche XMI-Elemente definiert, die die Konstrukte der OAL beschreiben können: *OAL:Method, OAL:Method.body, OAL:Entry, OAL:Comment, OAL:Select, OAL:Create, OAL:Assignment, OAL:If, OAL:ElIf, OAL:Else, OAL:EndIf, OAL:For, OAL:EndFor, OAL:While, OAL:EndWhile, OAL:Relate, OAL:Unrelate, OAL:Delete, OAL:Break und OAL:Return.* Abbildung 4.10 enthält ein Beispiel, wie OAL-Operationen mit XMI-Elemente dargestellt werden können. Dabei sind zwei Operationen definiert: Die Operation *create()* der Klasse *Publisher* dient dazu, eine neue Instanz (*newPublisher*) zu erzeugen. Mit der Operation *add()* der Klasse *Book* soll eine Instanz dieser Klasse (*self*) mit einer Instanz der Klasse *Publisher* (*newPublisher*) über die Beziehung *R1* verbunden werden.
Im nächsten Abschnitt wird diskutiert, welche Konstrukte XMI bereitsellt, die zu den Modelltransformationen beitragen können. XMI enthält Mechanismen, die uns u. a. erlauben, nur Teile eines XMI-Dokumentes zu beschreiben. So können bestimmte Abschnitte eines XMI-Dokumentes oder sogar einzelne XMI-Elemente die Einheit eine Austausches sein. Die XMI-Spezifikation stellt dafür das Element *<XMI.difference>* zur Verfügung, dessen Eigenschaften im nächsten Abschnitt näher betrachtet werden.

Abbildung 4.10 XMI-Darstellung von OAL Konstrukte

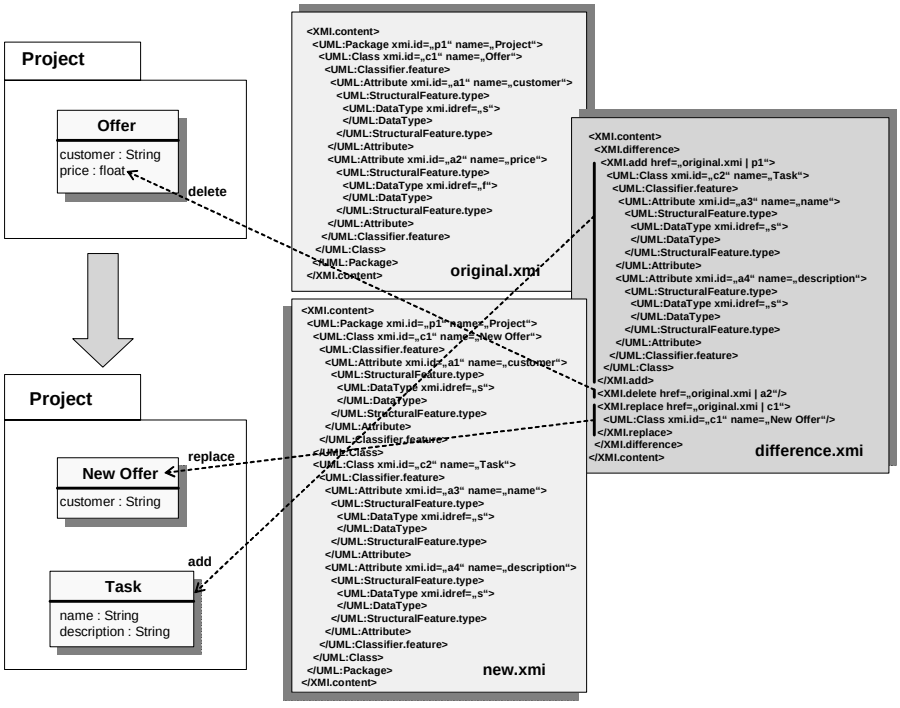


Das <XMI.difference> Element in XMI

XMI unterstützt nicht nur den Austausch von kompletten Modellen zwischen den verschiedenen Modellierungs-Werkzeugen, sondern sieht Mechanismen vor, die die optimierte Übertragung partieller Modelle erlauben. Dies geschieht auf der Ebene der XMI-Elemente von XMI-Doku-

menten mit Hilfe des <*XMI.difference*> Elements. Die XMI-Spezifikation definiert hierbei mit den Unterelementen <*XMI.add*>, <*XMI.delete*> und <*XMI.replace*> drei verschiedene Arten von Operationen zum Hinzufügen, Löschen oder Ersetzen von Modellelementen. Alle diese Operationen müssen in der Reihenfolge ihres Auftretens interpretiert werden, um die Differenz korrekt anzuwenden. Bei der *XMI.add* Operation wird das Unterelement des XMI-Elements <*XMI.add*> als Unterelement des refrenzierten XMI-Elements eingefügt. Die *XMI.delete* Operation bewirkt das Löschen des referenzierten XMI-Elements mit allen seiner Unterelemente. Die *XMI.replace* Operation führt zum Löschen des verlinkten Elements und dessen Ersetzung durch das in <*XMI.replace*> enthaltene Unterelement. Dabei bleiben die Inhalte des ursprunglichen Elements erhalten und werden in das neue Element eingefügt. Abbildung 4.11 enthält die originale und die neue Version eines vereinfachten Modells, jeweils als UML- und XMI-basierte Beschreibung. Zusätzlich ist auch eine XMI-Basierte Differenzbeschreibung enthalten. Diese beschreibt den Weg vom dem alten zum neuen Modell in Form einer Menge von primitiven Operationen auf dem originalen Modell. Dieses Beispiel soll die Wirkung der drei vorgestellten Operationen <*XMI.add*>, <*XMI.delete*> und <*XMI.replace*> demonstrieren. Im Beispiel wird die *XMI.add* Operation verwendet, um die neue Klasse *Task* mit ihren Attributen *name* und *description* dem Package *Project* im Modell hinzufügen. Mit der Operation *XMI.delete* wird anschließend das Attribut *price* gelöscht. Schließlich wird mit der Operation *XMI.replace* das *name* Attribut der Klasse *Offer* verändert Die alten XMI-Unterelemente der Klasse *Offer* bleiben dabei erhalten.

Abbildung 4.11 Modelltransformation mittels <XMI.difference>

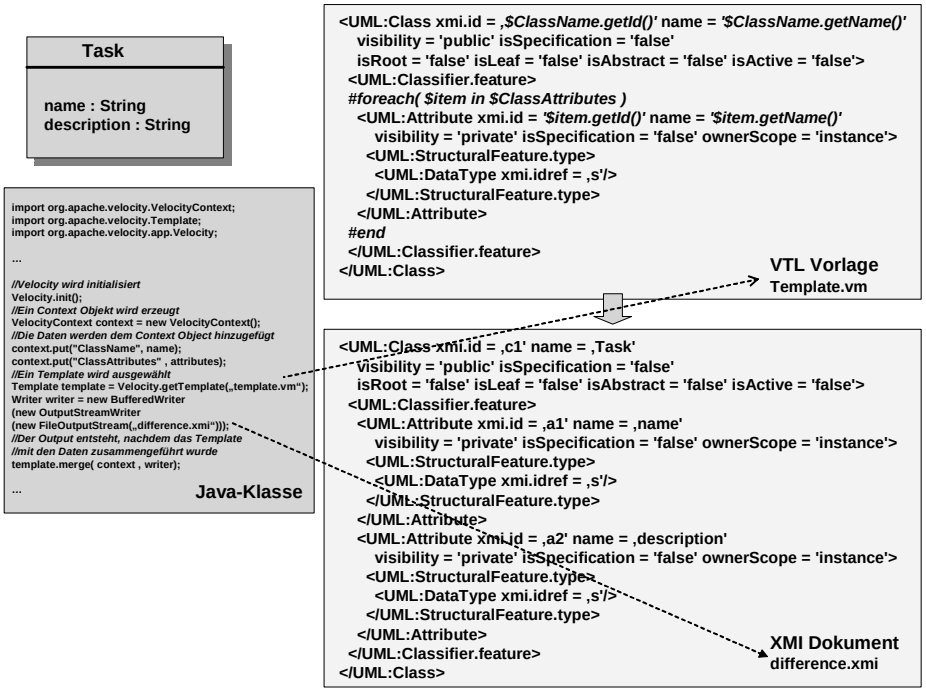


XMI-Beschreibungen für das alte und neue Modell lassen sich relativ einfach erzeugen. Das neue Modell wird nämlich durch Zusammenführen (Merging) des originalen Modell mit dem Differenz-Modell gewonnen. Bei der Differenzbeschreibung ist dies allerdings schwieriger. Die Manuelle XMI-Beschreibung des Differenz-Modells ist aufgrund der Größe der XMI-Dokumente und die Komplexität der XMI-Sprache nicht praktikabel und fehlerträchtig. Die Erstellung der XMI-Differenzbeschreibung soll daher nicht mehr hauptsächlich von Hand, sondern mit Hilfe einer Technik, die die automatisierte Generierung von Differenz-Modellen erlaubt. MTFLOW verwendet als Lösung die *Velocity Template Language* (VTL). Der Anwender müsste also die XMI-Dokumente nicht mehr manuell definieren, sondern gibt einfach mit Hilfe der GUI-Oberfläche von MTFLOW seine neuen Anforderungen an. Eine entsprechende VTL-Vorlage erzeugt dann daraus das XMI-Differenz-Dokument, das anschließend mit dem alten Modell zusammengeführt werden soll. Die Eigenschaften dieser Sprache werden im nächsten Abschnitt näher betrachtet.

Die Velocity Template Language (VTL)

Velocity [Apa03a]ist eine Java basierte Template Engine. Velocity zeichnet sich durch zwei wesentliche Merkmale aus: Zum einen ist Velocity entwickelt worden, um eine strikte Trennung zwischen Geschäftslogik und Darstellung zu erzielen. Zum anderen beinhaltet die Template-Sprache von Velocity, mit der Geschäftsdaten in die Darstellung eingefügt werden, mächtige, aber dennoch leicht verwendbare Script-Sprachen-Elemente. Beispielsweise ist es ohne großen Aufwand möglich, für jedes Element einer Liste von Objekten XML-Code mit Informationen aus dem jeweiligen Objekt generieren zu lassen. Die Trennung zwischen Darstellung (vm-Datei) und Geschäftslogik (Java-Klasse) bewirkt, dass Designer und Entwickler nicht an den selben Dateien arbeiten und sich lediglich über die Schnittstelle zwischen Ihnen, nämlich die Benennung der zu übergebenden Objekte, klar sein müssen. Viele Vorteile sprechen für den Einsatz von Velocity. Velocity stellt eine klare Syntax für den Template Designer zur Verfügung. Es offeriert außerdem ein einfaches Programmier-Modell für den Entwickler. Da Velocity Java-basiert ist, lässt es sich in eine Java-Anwendung integrieren. Darüber hinaus kann unabhängig entwickelt und gewartet werden, weil Templates und Code separat gehalten werden. Beim Einsatz von Velocity werden immer die gleichen Schritte ausgeführt. Zuerst wird Velocity initialisiert. Dann wird ein Context Objekt erzeugt, zu dem Daten hinzugefügt werden können. Im nächsten Schritt wird ein Template ausgewählt. Letztendlich entsteht der Output, nachdem das Template mit den Daten zusammengeführt wurde. Das Context-Konzept, dessen Benutzung gleich wie bei einer Hashtabelle ist, basiert auf einem Container der zwischen dem Programmierer und dem Template Designer Daten austauschen kann. Der Programmierer kann verschiedene Objekte in dem Context anlegen. Diese Objekte und deren Methoden, sowie Properties werden über Template-Elemente dem Designer zugänglich gemacht. In MTFLOW spielt Velocity die Rolle eines Generators für das XMI-Differenz-Dokument. Aus den Benutzer-Spezifikationen werden mit Velocity-Vorlagen XMI-Differenz-Dokumente erzeugt, die dann für die Modelltransformationen verwendet werden. Abbildung 4.12 zeigt ein Beispiel, wie Velocity bei der Erstellung eines Differenz-Dokumentes eingesetzt werden kann. Der Benutzer kann z. B. eine klasse *Task* mit zwei Attributen *name* und *description* definieren. Die VTL-Vorlage verfügt über Platzhalter, wo der Klassen-Name bzw. die Klassen-Attribute platziert werden können. So erhalten wir ein XMI-Dokument, das die Klasse *Task* mit ihren Attributen beschreibt.

Abbildung 4.12 Die Velocity Template Language



FAZIT

Das Hauptziel von XMI war hauptsächlich, den einfachen und einheitlichen Austausch von Modellen zwischen den verschiedenen UML CASE-Tools zu ermöglichen. In Bezug auf den in diesem Kapitel erwähnten XMI-Konstrukten erweisen sich weitere wichtige Eigenschaften von XMI besonders vorteilhaft:

- Die in XMI enthaltenen Mechanismen, um Objekte innerhalb eines Dokumentes oder zwischen Dokumenten zu verknüpfen nämlich *xmi.idref* und *xmi.href* erlauben es, auf die Verarbeitung kompletter häufig großer Modelle zu verzichten. Mit Hilfe dieser Referenzen können beispielsweise gemeinsam genutzte Modellfragmente in externe Dokumente ausgelagert werden.

- Die *XMI.difference* Elemente ermöglichen es, die Differenzen zwischen zwei Modellen zu beschreiben. Diese Differenzbeschreibung kann zum Beispiel dazu genutzt werden, sich bei Modelländerungen eine Übertragung des vollständigen Modells zu ersparen und stattdessen nur die Differenzbeschreibung zu übertragen. Die XMI-Technik kann deshalb dazu beitragen, die Performanz zu verbessern, wenn z. B. große Modelle verteilt in einem Netwerk bearbeitet werden sollen.

- Diese Technik kann auch genutzt werden, um die Transformation eines Modells durch eine Sequenz von primitiven Operationen auf diesem Modell zu beschreiben. Nimmt man diese XMI-basierte Differenzbeschreibung und „mergt“ sie mit dem alten Modell, dann erhalten wir so ein neues Modell, das dann weiter verfeinert werden kann.

Aus allen diesen Vorteilen wird diese Technik nämlich XMI in MTFLOW bei der Modelltransformationen verwendet. In MTFLOW werden große Software-Systeme mit Hilfe von UML und OAL spezifiziert. Diese Spezifikation erfolgt über aufeinanderfolgende Modelltransformationen, wo XMI als Technik zum Einsatz kommt Diese Transformationen erfolgen aber nicht willkürlich, sie werden in MTFLOW nämlich durch ein Workflow-Modell kontrolliert, das den Ablauf der Transformationen vorschreibt. Im nächsten Kapitel werden der Aufbau und das Funktionsprinzip von MTFLOW näher betrachtet.

Kapitel 5 MTFLOW

Nachdem in den letzten Kapiteln die Grundlagen, Verfahren und Technologien, auf denen diese Arbeit aufbaut, vorgestellt wurden, wird in diesem Kapitel das im Rahmen dieser Arbeit entwickelte Werkzeug näher betrachtet. Dabei wird auf den Aufbau und die besondere Eigenschaften von *MTFLOW* eingegangen. In diesem Kapitel werden zunächst Workflow-Modelle vorgestellt. Dabei werden ein Workflow-Metamodell für *MTFLOW*-Workflow-Modelle und ein Workflow-Modell zum Spezifizieren von Versionierungssystemen definiert. Dann werden der Aufbau und das Funktionsprinzip von *MTFLOW* näher erläutert.

5.1 Einleitung

MTFLOW bietet dem Softwareentwickler eine Umgebung, in der alle Schritte, die der Spezifikation eines Software-Systems dienen, vorgenommen werden können. *MTFLOW* soll dabei die einfache und schnelle Spezifikation großer Software-Systeme mit Hilfe von UML und OAL unterstützen. Diese Spezifikation erfolgt über aufeinanderfolgende Modelltransformationen, bei welchen XMI als Technik zum Einsatz kommt. Diese Transformationen erfolgen aber nicht willkürlich, sondern werden durch ein Workflow-Modell kontrolliert, das den Ablauf der Transformationen vorschreibt. *MTFLOW* steht in enger Beziehung zu dem Produktlinienansatz bzw. der generativen Programmierung (GP) und dem verwandten Ansatz der MDA. In Bezug auf der GP spielt *MTFLOW* die Rolle eines Konfigurators, der den Kern der zweiten Phase der GP, nämlich die Applikations-Engineering-Phase, bildet. In dieser Phase wird versucht, aus einer konkreten Spezifikation eines Software-Systems die dazugehörige Software-Variante zu generieren. Diese Spezifikation erfolgt in *MTFLOW* nicht mehr direkt mit einer DSL, deren Syntax und Semantik schwer zu erlernen sind, sondern mit Hilfe eines Konfigurators, der das erwünschte Produkt konfigurieren soll, vergleichbar mit der Automobilindustrie, wo der Kunde sein Auto bis ins Detail selbst zusammenstellen kann. Mit einer Benutzeroberfläche stellt *MTFLOW* dem Entwickler ein einfaches Mittel zur Verfügung, damit er seine Anforderung leicht formulieren und sein erwünschtes System schrittweise und korrekt spezifizieren kann. Der Entwickler kann dabei die gewünschten System-Merkmale wählen und wird dabei durch vordefinierte Arbeitsschritte geleitet, bis er dann am Ende das ganze System spezifiziert. Das konfigurierte, in der UML beschriebenes System wird später an den Generator übergeben, der das System implementieren soll. Aus verschiedenen Konfigurationen entstehen verschiedene Varianten des Systems, die nach der Code-Generierung die Mitglieder der Produktlinie darstellen sollen. Als Beispiel einer Produktlinie haben wir objektorientierte Versionierungssysteme genommen, allerdings beschränkt sich *MTFLOW* nicht auf eine einzelne Produktlinie. Es handelt sich dabei nämlich um einen generi-

schen Ansatz, d. h. es können mehrere Software-Produktlinien unterstützt werden. Dafür sollen einfach andere Workflow-Modelle definiert werden. Ein Hersteller von Versionierungssystemen z. B. kann seine Produkte als eine produktlinie realisieren, anstelle jeden Kundenauftrag von Grund auf neu zu implementieren. *MTFLOW* bietet dabei dem Kunden die Möglichkeit, seine Anforderungen in Form einer Konfiguration dem Hersteller mitzuteilen, der dann ein speziell auf den Bedarf zugeschnittenes Versionierungssystem erstellt. Die Konfiguration enthält Angaben, die die Anforderungen des Kunden an das Versionierungssystem widerspiegeln wie z. B. über versioniert und unversioniert zu speichernde Objekttypen, über die Beziehungen, die zwischen Objekten bestehen können, über die typen von Arbeitskontexten, die angelegt werden können und über die Propagierung von Operationen über Beziehungen.
MTFLOW steht auch in Beziehung zu MDA. Die Modelltransformationen spielen in MDA eine große Rolle. Es werden dabei u. a. PIMs in PIMs transformiert und dies im Wesentlichen zur Erweiterung von Modellen. Die Spezifikation eines Software-Systems in *MTFLOW* erfolgt über Modelltransformationen. Dabei werden XMI und VTL als Techniken eingesetzt, um ein UML-Modell auf ein anderes UML-Modell abzubilden. Diese Methode der Modelltransformation kann auch bei der PIM-zu-PIM Transformation in MDA verwendet werden. MDA sieht vor, dass zunächst ein System modelliert werden soll und dann wird das Modell durch einen Generator auf einer ausgewählten Plattform implementiert. *MTFLOW* unterstützt dieses Vorgehensmodell. Das Software-System wird dabei in UML modelliert, so wird die Mühe erspart, einen Generator zu finden, der aus einem gegebenen Modell Code in einer bestimmten Implementierungssprache erzeugen soll. In vielen CASE-Tools wie Poseidon [Gen03] und Rational Rose [IBM03] ist ein Generator integriert, der aus einem UML-Modell Java-Code erzeugen kann. Compiler wie MC-2020 und MC-3020 [Pro03] können aus Modellen in Executable-UML [MB02] C++- und C-Code erzeugen.

5.2 Workflow-Modelle

In diesem Abschnitt werden zunächst die für diese Arbeit relevanten Aspekte aus dem Bereich Workflow Management vorgestellt. In [WFMC04] ist ein Überblick über die gesamte Thematik zu finden. Die Begriffe *Prozeß*, *Workflow* und *Workflow-Management-System (WFMS)* stehen in einem unmittelbaren Zusammenhang. So definiert, startet und verwaltet ein Workflow-Management-System die Ausführung von Workflows, die wiederum als Automatisierung von Prozessen verstanden werden können. Diese aufeinander aufbauenden Begiffe und andere wesentliche Begiffe, die für diese Arbeit relevant sind werden nach der *Workflow-Management-Coalition (WFMC)* in den folgenden Abschnitten genauer beschrieben. Die WFMC ist ein nicht kommerzieller Zusammenschluss von diversen Herstellern, Forschern, Benutzern und Verkäufern von WFMS. Das Primäre Ziel von WFMC ist die Standardisierung von WFMS.

5.2.1 Überblick und Begiffsbestimmung

Ein Prozess besteht aus entweder einer Aktivität (eine Menge von Arbeitsschritten) oder einer Menge von verknüpften Aktivitäten oder Prozessen (diese werden als Subprozesse bezeichnet). Ein Prozess dient zur Erreichung eines bestimmten Ziels. Die Beschreibung eines Prozesses erfolgt durch ein zugehöriges Prozessmodell. Prozesse können mehrstufig hierarchisch aufgebaut sein, indem sie Subprozesse enthalten. Ein Workflow stellt nach [WFMC04] einen automatisier-

ten Prozess dar, der ausgehend von einem auslösenden Ereignis entlang einer definierten Kette von Aktivitäten bis zu einem definierten Arbeitsergebnis führt, wobei der Grad der Vervollständigung der Arbeitsergebnisse mit jeder einzelnen Aktivität zunimmt. Ein Workflow ist ein Prozess, der von einem WFMS zum Ablauf gebracht werden kann. Daher bestehen die Unterschiede bei der Begiffsbildung zwischen Prozess und Workflow lediglich darin, dass ein Workflow stets durch ein WFMS zur Ausführung gebracht werden kann, während diese Eigenschaft nicht notwendigerweise für Prozesse gilt. Ein WFMS ist ein Softwaresystem, das Workflows gemäß der entsprechenden Prozessmodelle zum Ablauf bringt und dem Management von Arbeitsvorgängen (Arbeitsabläufen) dient. WFMS machen oft vieles gleichzeitig: Sie steuern die Aufgabenbearbeitung, sie stellen die benötigten Daten und Tools zur Verfügung, verwalten Daten, legen Kontrollflüsse fest, rufen Applikationsprogramme auf, benachrichtigen den Benutzer über anstehenden Aufgaben und verwalten Historien. Der Workflow ergibt sich durch Instanziierung mittels eines WFMS aus einem Workflow-Modell. Das Workflow-Modell ist eine Computer bearbeitbare Form des Prozess-Modells. Es wird zwischen der Erstellungszeit (Build-Time) und Laufzeit (Run-Time) unterschieden. Zur Erstellungszeit wird ein Prozess in einer sogenannten Workflow-Spezifikation modelliert (Workflow-Modell). Zur Laufzeit werden mehrere Instanzen des Workflows nach diesem Modell erstellt (Abbildung 5.1). Im Kern eines WFMSs befindet sich die *Workflow-Engine.* Sie ist die wichtigste Komponente eines WFMSs, die die Laufzeitumgebung für die Prozessinstanz zur Verfügung stellt. Sie kann die Workflow-Modelle interpretieren und instanziieren. Die Workflow-Engine erzeugt somit die Prozess-Instanz, steuert und verwaltet sie. MTFLOW spielt die Rolle einer Workflow-Engine. Dabei wird nämlich ein Workflow-Modell instanziiert und die Prozess-Instanzen kontrolliert. So können mehrere Software-Produktlinien unterstützt werden. Dafür sollen einfach andere Workflow-Modelle definiert werden. Der Ansatz der WFMC unterscheidet zwei Typen von Daten in WFMS, die zwischen Aktivitäten ausgetauscht werden und für die Durchführung von Workflows relevant sind: Kontrolldaten und Nutzdaten. Diese Daten sind typisiert, wobei sowohl einfache als auch zusammengesetzte Datentypen zugelassen werden. Kontrolldaten dienen dem Ablauf des Workflows. Mit ihrer Hilfe bestimmt das WFMS mit welchen Aktivitäten fortgefahren wird. Ihre Verwaltung unterliegt dem WFMS. Nutzdaten sind Daten, die auch ohne das WFMS angefalllen wären. Der Kontrollfluss beschreibt die chronologische und logische Ordnung der Aktivitäten . Im Kontroll-Fluss ist es möglich, Bedingungen und Schleifen zu formulieren. Der Datenfluss beschreibt, wie die für Aktivitäten benötigten Daten zwischen den Aktivitäten weitergeleitet werden.

5.2.2 *MTFLOW*-Workflow-Metamodell

Ein *Workflow-Modell* ist nach [WFMC04] eine formale, sehr detailreiche Modellierung des Prozesses. Es besteht aus Aktivitäten und deren Beziehungen, Bedingungen zum Starten und Beenden des Prozesses sowie Informationen über die einzelnen Aktivitäten, z. B. die betroffene Daten. Das Workflow-Modell liegt in einer Computer-bearbeitbare Form vor. *MTFLOW* benutzt ein Workflow-Modell, um die Schritte der Spezifizifikation zu kontrollieren. Wie erwähnt, könnten auch andere Workflow-Modelle definiert werden, die zum Spezifizieren anderer Software-Systeme beitragen würden. Ein Metamodell für *MTFLOW*-Workflow-Modelle wird durch Abbildung 5.2 dargestellt.

Abbildung 5.1 Unterschied zwischen Prozess und Workflow

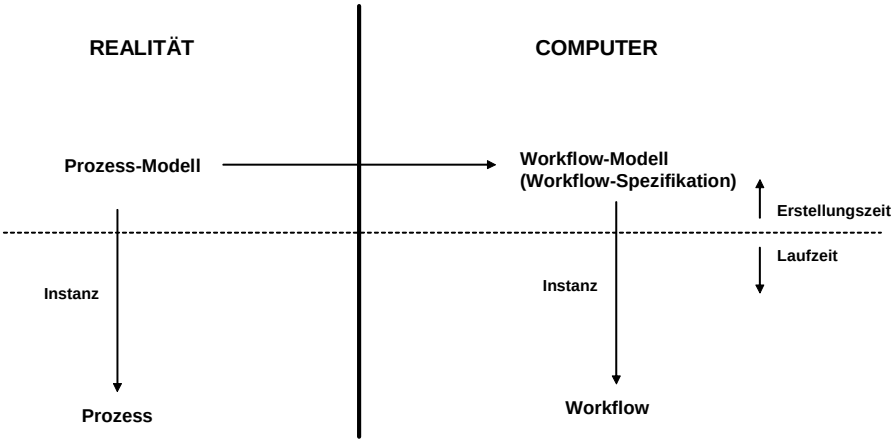
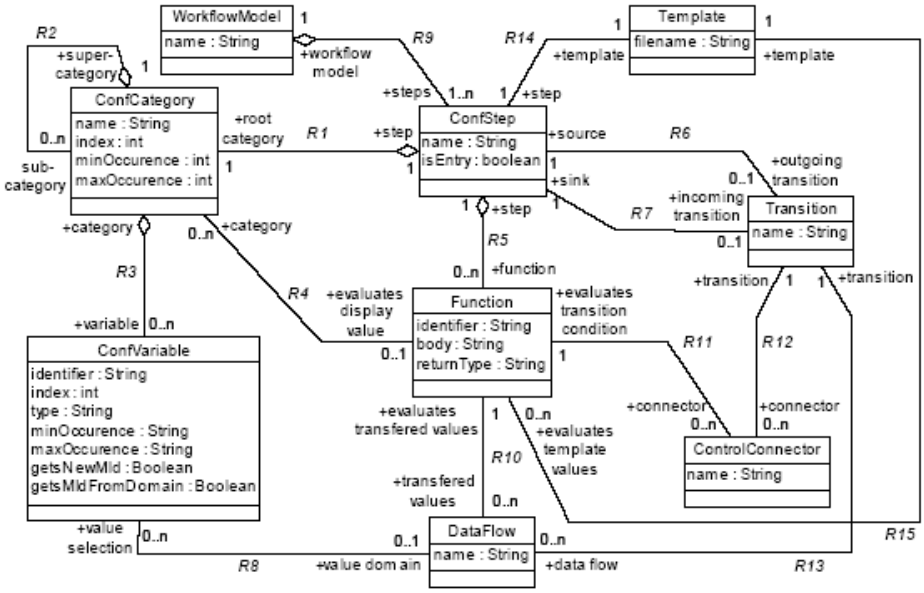


Abbildung 5.2 Metamodell für *MTFLOW*-Workflow-Modelle



Ein Workflow-Modell besteht aus mehreren Konfigurations-Schritten (*ConfStep*), wobei ein Konfigurations-Schritt als *entry* bezeichnet wird. Dieser stellt dann die erste Aktivität eines Workflows (Prozesses) dar. Jeder Schritt definiert eine Wurzel-Konfigurations-Kategorie (*root ConfCategory*), die in mehreren Teil-Kategorien (*sub-categories*) unterteilt werden kann. Eine Kategorie besteht aus mehreren Konfigurations-Variablen (*ConfVariable*), die die benutzerdefinierten Parameter wiederspiegeln und die dann später bei der Generierung der Transformations-Dokumente verwendet werden. Die Werte dieser Variablen sind immer einfache Datentypen. Die Kategorien und die Variablen sind indexiert und kommen in der Benutzeroberfläche in einer

bestimmten Reihenfolge vor. Eine Konfigurationsvariable in einem Konfigurationsschritt kann mit einem Daten-Fluss aus einem vorherigen Schritt verbunden sein, der die erlaubten Werte für diese Variable beschränkt. Der Daten-Fluss ist dann eine Liste mit bestimmten Werten, die die Variable aufnehmen kann. Jeder Variable wird ein Benutzer-definierter Wert und ein eindeutiges Identifier (Id) zugewiesen. Die Id dient dazu, ein Modell-Element, das in einem Modell-Transformations-Schritt erzeugt wird, auch in den folgenden Transformationen manipuliert werden kann. Zwischen den Konfigurations-Schritten fließen Nutz- und Kontrolldaten. Der Datenfluss und der Kontrollfluss werden im Metamodell durch die Metamodell-Elemente *Transition* und *Function* dargestellt. Mit jedem Konfigurations-Schritt in einem Workflow-Modell können mehrere Funktionen verbunden werden. Diese Funktionen ändern die Variablen-Werte nicht, sondern dienen nur der Bewertung von Anzeige-Werten der Konfigurations-Kategorien (R4), von Werten der in einem Daten-Fluss übertragenen Daten (R10), oder von Werten, die durch die Vorlagen (Templates) manipuliert werden (R15), oder auch Bewertung der in einem Kontroll-Fluss übertragenen Konditionen (R11). Für jeden Konfigurationsschritt existiert genau eine Vorlage, die aus den in diesem Schritt vom Benutzer definierten Parametern ein Differenz-Dokument erzeugen soll.

5.2.3 *MTFLOW*-Workflow-Modell für Versionierungssysteme

MTFLOW basiert auf einem Workflow-Modell zum Spezifizieren von Versionierungssystemen. Ein Workflow besteht in der Regel aus mehreren, zum Teil hierarchisch angeordneten Aktivitäten. Ein Modell besteht aus einem Netz von Aktivitäten und Beziehungen zwischen diesen Aktivitäten. Zur Modellierung und Beschreibung von Workflow Prozessen können unterschiedliche Darstellungsmodelle verwendet werden. Als Standart werden Statecharts und Activitycharts eingesetzt, die auch in UML inkludiert sind. Damit wurde für die breite Anerkennung und Standardisierung gesorgt Alternativ können die Workflow Prozesse auch mit Hilfe von Petri-Netz Varianten beschrieben werden oder auch durch Skript Sprachen. Die Spezifikationsmethode, die auf State- und Activitycharts basiert, kombiniert die Einfachheit und Genauigkeit von Automatenmodellen mit weitreichenden Möglichkeiten zur Visualisierung und hat gleichzeitig eine mit Petri-Netzen vergleichbare Ausdrucksmächtigkeit. Von grundlegender Bedeutung ist außerdem die durch Schachtelung von Zuständen gegebene Möglickeit, Spezifikationen zu modularisieren, zur schrittweisen Verfeinerung oder zur Komposition existierender Workflows. Eine Verhaltensbeschreibung eines Systems muss komplexe Folgen von Ereignissen, Aktionen und Bedingunen berücksichtigen. Eine brauchbare Modellierungsmethode muss modular, hierarchisch und gut strukturiert sein und muss Konstrukte enthalten, mit denen parallele Abläufe dargestellt werden können. Diese Anforderungen werden vom Formalismus der State- und Activitycharts erfüllt. Mit Hilfe von Activitycharts kann ein System aus funktionaler Sicht modelliert werden. Konkret bedeutet dies eine Untergliederung des zu modellierenden systems in Aktivitäten und Datenflüsse zwischen den Aktivitäten. Statecharts beschreiben ein System aus Verhaltenssicht. Sie definieren zeitabhängige Vorgänge, die durch Zustandsübergänge modelliert werden. Statecharts bieten auch Konstrukte für die hierarchische Anordnung von Zuständen, für die Darstellung paralleler Zustände und für die Kommunikation zwischen Zuständen mittels Ereignissen. In Abbildung 5.3 ist der *MTFLOW*-Workflow dargestellt, der mit einem Activitychart und einem Statechart modelliert ist. Die Verfeinerungen der Komplexen Zustände werden in den Abbildungen 5.4, 5.5 und 5.6 dargestellt. Der obere Teil der abbildung 5.3 enthält das Activitychart *Versioning System_AC,* das aus vier Activitäten *Define_ObjectType_A*, *Make_ObjectType_Versionable_A, Define_RelationshipType_A* und *Define_WorkspaceType_A*,

dem durch die Pfeile dargestellten Datenfluss und einem Verweis auf das Statechart *Versioning System_SC* besteht. Die vier Aktivitäten sind Instanzen von *ConfStep* aus dem oben eingeführten Metamodell. Die Daten, die zwischen den verschiedenen Aktivitäten fließen sind Instanzen von *Transition* und definieren zusammengesetzte Datentypen. So sind *ObjectTypes (OTs)* Datenstrukturen, die die vom Benutzer in dem ersten Konfigurations-Schritt definierten *Name* und *Attribute* eines ObjectTypes kapseln. *Name* und *Attribute* sind dabei Instanzen vom Metamodell-Element *DataFlow. Attribute* ist wiederum ein komplexer Datentyp und beinhaltet *name* und *type* als einfache Datentypen, die Instanzen von *ConfVariable* sind. Die Daten, die zwischen den verschiedenen Konfigurations-Schritten beim Spezifizieren eines Versionierungssystems fließen, sind in der Tabelle in Abbildung 5.7 zusammengefasst. Das Statechart *Versioning System_SC* ist im unteren Teil der Abbildung 5.3 wiedergegeben und beschreibt das Verhalten des Activitycharts, insbesondere den Kontrollfluss zwischen den Aktivitäten. Die Rechtecke mit den abgerundeten Ecken in *Versioning System_SC* stellen Zustände dar. Die Pfeile geben den Kontroll-Fluss an. Dabei sind die Zustände Instanzen vom *ConfCategory*-Element des Metamodells und die Kontroll-Flüsse Instanzen von *Function*. Diese Funktionen ermöglichen verschiedene Bewertungen und helfen dabei, die chronologische Ordnung der Aktivitäten zu bestimmen. Sind zwei Zustände durch einen Kontroll-Fluss mit der Beschriftung E[C]/A verbunden, werden beim Eintreten des Ereignisses E, sofern Bedingung C erfüllt ist, der Ausgangszustand des Kontroll-Flusses verlassen, sein Zielzustand betreten und die Aktion A ausgeführt.

Zu Beginn der Ausführung des Workflows wird die Aktivität *Define_ObjectType_A* durch die Anweisung *start* (abgekürzt *st!*) *(Define_ObjectType_A)* gestartet und der Zustand *Define_ObjectType_S* betreten, nachdem das Ereignis *new OT* (new ObjektType) generiert wurde. Wurde das Ereignis *new verOT* (new versionable ObjectType) generiert, wird die Aktivität *Make_ObjektType_Versionable_A* gestartet und der nächste Zustand *Make_ObjektType_Versionable_S* betreten. Wurde dagegen das Ereignis *new OT* wieder generiert, wird der Zustand *Define_ObjectType_S* nicht verlassen und so kann ein neuer Objekttyp definiert werden. Eine Äquivalente Erklärung gilt für die anderen Zustände im Statechart. In den Abbildungen 5.4, 5.5 und 5.6 wird gezeigt, wie die Zustände *Define_ObjectType_S, Make_ObjektType_Versionable_S, Define RelationshipType_S* und *Define_WorkspaceType_S* verfeinert werden können. Die Unterzustände sind wiederum Instanzen von *ConfCategory* und sind mit dem Vater-Zustand über die Beziehung *super-category_sub-category* in Verbindung. Beispielsweise sind die zustände *Define_Name_S* und *Definie_Attribute_S* in Abbildung 5.4 Teil-Kategorien der Wurzel-Konfigurations-Kategorie Define_ObjectType_S.

MTFLOW basiert auf diesem Workflow-Modell und erlaubt mit dessen Hilfe dem Benutzer eine korrekte Spezifikation des Versionierungssystems zu definieren. Unmögliche Kombinationen sind dabei nicht erlaubt. So können zum Beispiel *floating* oder *createSuccessor propagated* im Fall eines nicht versionierbaren ObjektTypes nicht auf *true* gesetzt werden (Abbildung 5.5).

Abbildung 5.3 Activity- und Statechart zur Modellierung des Workflows beim Spezifizieren eines Versionierungssystems

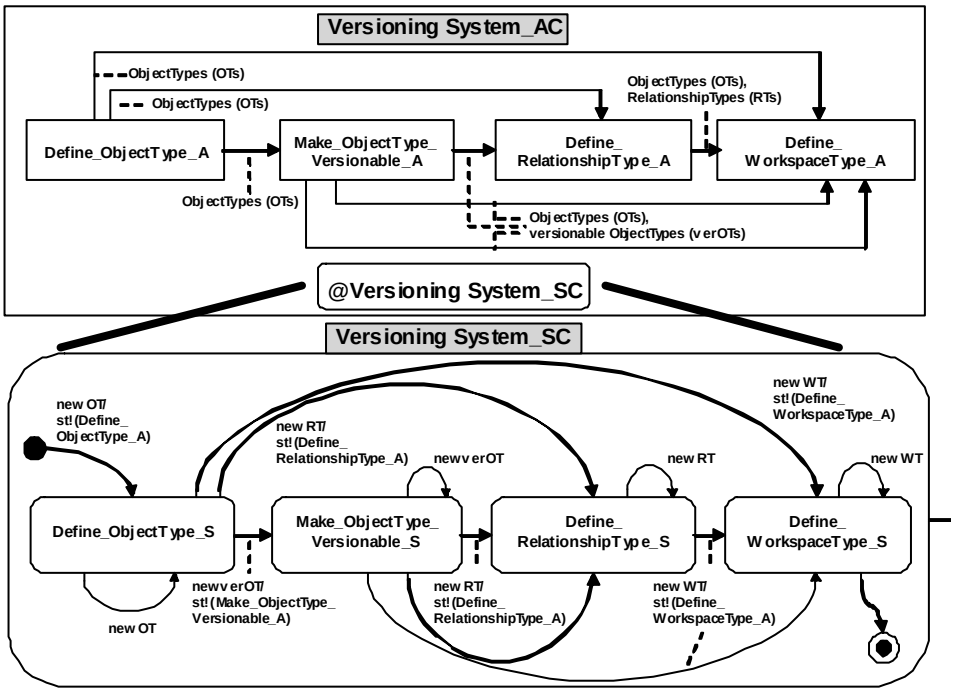


Abbildung 5.4 Verfeinerte Statechart, Fortsetzung von Abbildung 5.2

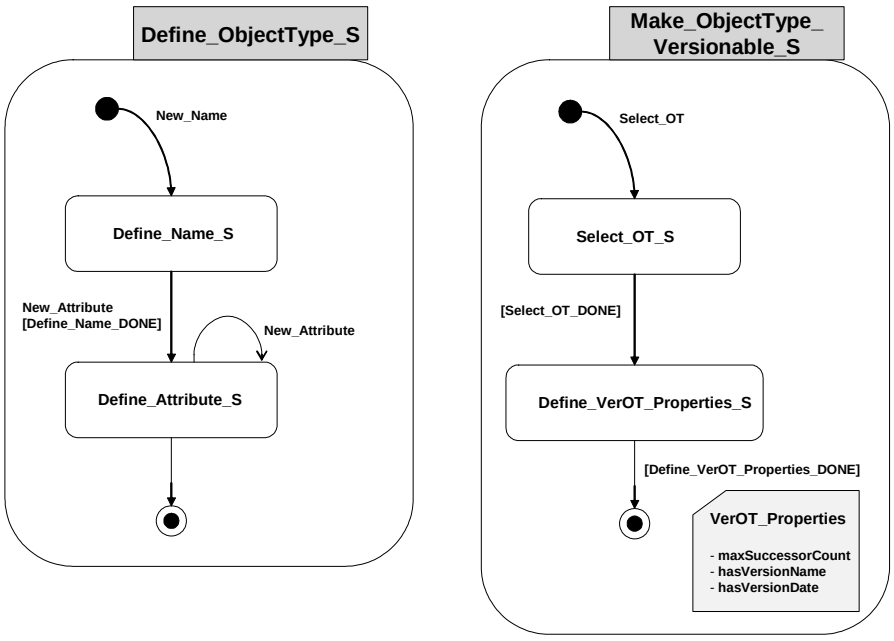


Abbildung 5.5 Verfeinerte Statechart, Fortsetzung von Abbildung 5.2

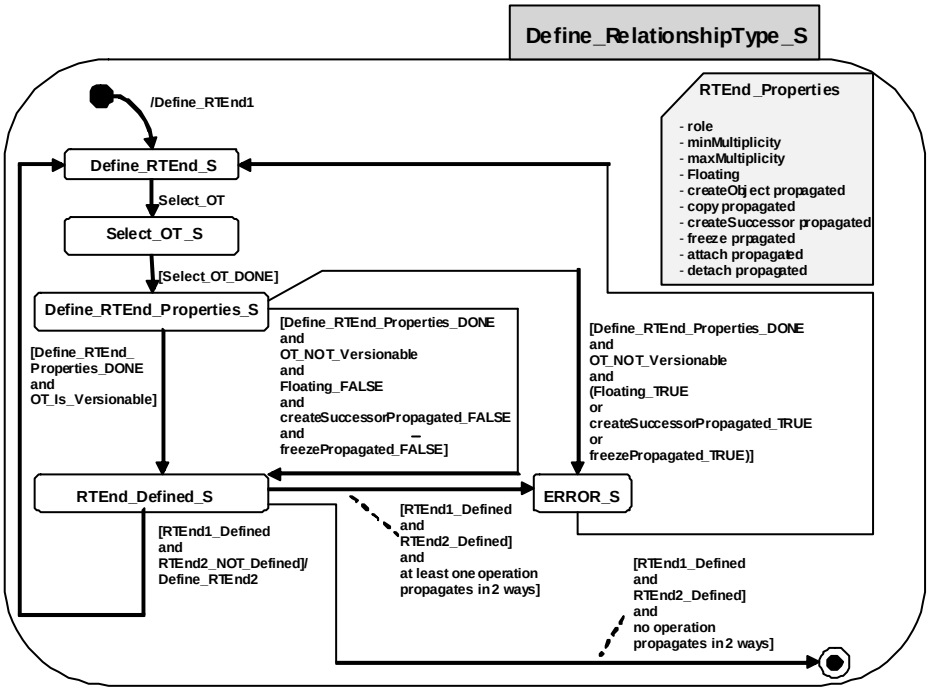


Abbildung 5.6 Verfeinerte Statechart, Fortsetzung von Abbildung 5.2

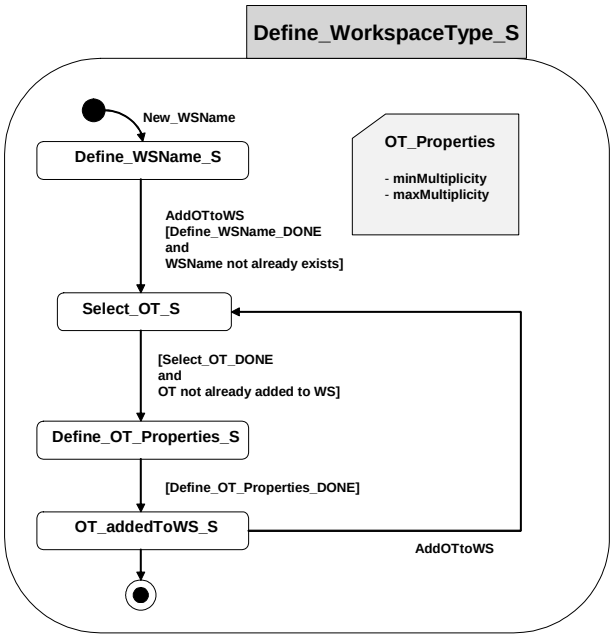


Abbildung 5.7 Daten-Fluss in *MTFLOW*

Instance of ConfStep	Instance of Transition	Instance of DataFlow	Instance of ConfVariable
name : Define_ObjectType_A isEntry : true	name : ObjectType	name : Name name : Attribute	identifier : name identifier : name identifier : type
name : Make_ObjectType_Versionable_A isEntry : false	name : VersionableObjectType	name : VersionableObjectType	identifier : objectType_Name identifier : maxSuccessorCount identifier : hasVersionName identifier : hasVersionDate
name : Define_RelationshipType_A isEntry : false	name : RelationshipType	name : End1 name : End2	identifier : role identifier : objectType_Name identifier : minMultiplicity identifier : maxMultiplicity identifier : floating identifier : createObject propagated identifier : copy propagated identifier : createSuccessor propagated identifier : freeze propagated identifier : attach propagated identifier : detach propagated
name : Define_WorkspaceType_A isEntry : false			

5.3 Aufbau und Funktionsprinzip von *MTFLOW*

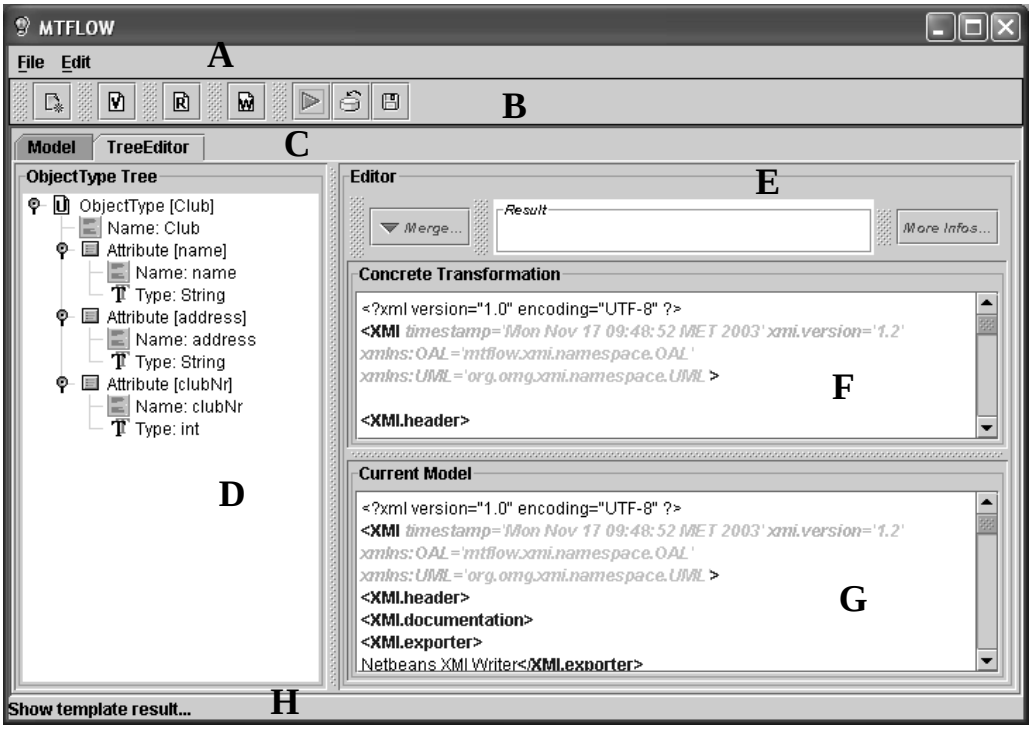
Im Allgemeinen bietet *MTFLOW* eine Umgebung, in der verschiedene Arbeitsschritte, die einer domänenspezifischen Spezifikation eines Softwaresystems dienen, vorgenommen werden können. *MTFLOW* stellt dem Softwareentwickler eine Benutzeroberfläche zur Verfügung, die ihn bei seiner Konfiguration und Spezifikation des Versionierungssystems unterstützen soll. Das System ist als Java-Programm mit einer graphischen Benutzeroberfläche (Graphical User Interface, GUI) konzipiert. Diese Benutzeroberfläche dient dazu, dem Benutzer die Möglichkeit zu geben, sein System einfach und korrekt zu modellieren. Der Benutzer braucht dabei keine Vorkenntnisse über die verwendete Modellierungssprache. Er tätigt Eingaben über Maus und Tastatur, Ausgaben erflogen auf dem Bildschirm. Der Dialog zwischen Benutzer und Applikation läuft in einem Hauptfenster und verschiedenen unterstützenden Dialogfeldern ab.

5.3.1 Die Benutzeroberfläche von *MTFLOW*

Abbildung 5.8 enthält ein Bildschirmfoto der Benutzeroberfläche von MTFLOW. Die verschiedenen Elemente der Benutzeroberfäche wurden durch Buchstaben markiert. Die Oberfäche lässt sich grob in drei Teile gliedern: Am oberen Rand befinden sich Menü- und symbolleiste (A und B), eine Statusleiste (H) bildet den unteren Rand. Die mittlere und größte Fläche enthält die Unterflächen für die Registertabs (C), die verschiedenen Bäume (D), die in den verschiedenen Konfigurationsschritten vom Benutzer definiert werden sollen, Concrete-Transformation-Editor (F), Current-Model-Editor (G) und eine Unterfläche zum Anzeigen der Transformationsinformationen (E). Im folgenden werden die einzelnen Komponenten vorgestellt. Dann werden die zen-

tralen Elemente näher betrachtet. Dabei wird erläutert, welche Rollen diese Elemente bei der Spezifikation und Modelltransformation spielen.

Abbildung 5.8 Hauptfenster von *MTFLOW*



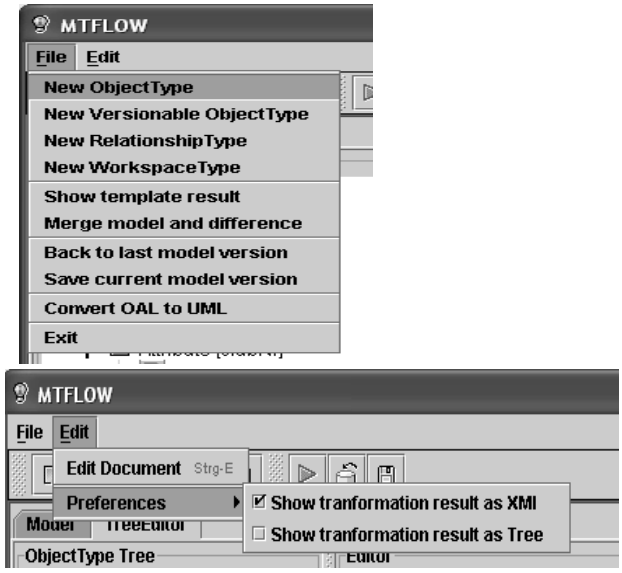
Menüleiste, Symbolleiste und Statusleiste

Die Menüleiste (A) enthält die Menüs *File* und *Edit* mit mehreren Menüpunkten, die den Zugriff auf globalen Funktionen erlauben, die in den verschiedenen Konfigurationsschritten sinnvoll ausgeführt werden können. Diese Menüpunkte sind in Abbildung 5.9 zu sehen und definieren die folgenden Funktionen:

- Die ersten vier Menüpunkte des Menüs *File* (*New ObjectType*, *New Versionable ObjectType*, *new RelationshipType* und *New WorkspaceType*) dienen dazu, neue Objektypen bzw. versionierbare ObjektTypen zu definieren und Beziehungen und Arbeitskontexte anzulegen.
- *Show template result* dient der Erstellung eines XMI-Differenz-Dokumentes, das die Parameter enthält, die in einem Konfigurationsschritt durch den Benutzer definiert worden sind.
- *Merge model and difference* erlaubt das Zusammenführen des neu erstellten XMI-Differenz-Dokumentes mit dem im vorherigen Schritt erhaltenen XMI-Modell-Dokument. Die Funktion führt damit die eigentliche Modelltransformation aus.
- *Back to last model version* hilft dabei, einen Transformationsschritt rückgängig zu machen. Die zuletzt am Modell vorgenommene Änderung wird nicht mehr betrachtet. Der Benutzer kann dann eine andere Spezifikation definieren.

- *Convert OAL to UML* dient dazu, eine in OAL geschriebene Methode in eine in XMI beschriebene UML-Methode umzuwandeln. Das erlaubt, das erhaltene Modell-Dokument in XMI-Format zu exportieren. Dieses Dokument kann dann durch ein CASE-Tool importiert werden.
- Das Menüpunkt *Edit Document* im Menü *Edit* gibt dem Benutzer die Möglichkeit, das XMI-Dokument manuell zu editieren. Dabei wird das Dokument als XMI-Baum dargestellt. Das System unterstützt das Editieren des XMI-Dokuments, indem neue Baum-Knoten, die den XMI-Elementen entsprechen hinzugefügt bzw. gelöscht werden können.
- Durch *Preferences* kann den Benutzer entscheiden, ob er das XMI-Differenz-Dokument als XMI-Code oder als XMI-Baum bearbeiten will.

Abbildung 5.9 Menüleiste in *MTFLOW*



Die Symbolleiste (B) ermöglicht einen alternativen Zugriff auf die am häufigsten benötigten Funktionen aus der Menüleiste. In der Symbolleiste werden die Funktionen durch Symbole identifiziert, womit die integrierten und wichtigsten Aktionen schneller gestartet werden können. Die Symbole enthalten auch Hinweise, was für Aktionen eine Funktion auslöst. In Abbildung 5.10 sind die verschiedenen Symbole zu sehen.

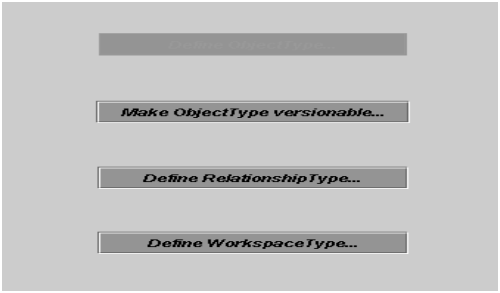
Die Statusleiste (H) informiert den Benutzer über den Zustand der Anwendung und den Ablauf der gestarteten Aktionen. Der Benutzer kann diesem Feld beispielsweise entnehmen, welche Aktion er zuletzt aufgerufen hat.

Abbildung 5.10 Symbolleiste in *MTFLOW*

Registertabs, Baum, Concrete-Transformation-Editor und Current-Model-Editor

MTFLOW bietet drei Registertabs (C), die dem Benutzer ermöglichen, zwischen verschiedenen Arbeitsflächen umzuschalten. In der *Model-Tab* werden die in jedem Zeitpunkt der System-Konfiguration die möglichen Arbeitsschritte angezeigt, mit denen der Benutzer seine Spezifikation fortsetzen kann. Die Entscheidung, ob eine bestimmte Aktivität in einem bestimmten Zeitpunkt möglich ist, wird dem Workflow-Modell entnommen, das die Reihenfolge der Konfigurationsschritte vorschreibt. *MTFLOW* sieht bei der Spezifikation eines Versionierungssystems vier Konfigurationsschritte vor, genau wie es im Activity- und Statechart des Workflow-Modells definiert wurde. Der Benutzer kann im ersten Konfigurationsschritt (*Define ObjectType*) neue Objekttypen definieren. Er kann dann im zweiten Konfigurationsschritt (*Make ObjectType versionable*) ein Objekttyp als versionierbar definieren. Im dritten Konfigurationsschritt (*Definie RelationshipType*) können neue Beziehungen zwischen den bereits definierten Objekttypen angelegt werden. Im letzten Konfigurationsschritt (*Define WorkspaceType*) können abschließend neue Arbeitskontexte definiert werden. Betritt der Benutzer einmal einen Konfigurationsschritt, besteht dann keine Möglichkeit mehr zu den vorherigen Konfigurationsschritten zurückzukehren. Wenn der Benutzer z.B. den ersten Konfigurationsschritt verläßt, um versionierbare Objekttypen zu markieren, können neue Objekttypen nicht mehr definiert werden. Diese Beschränkungen werden in *MTFLOW* unterstützt, indem in der *Model-Tab* der unmögliche Konfigurationsschritt nicht mehr auswählbar ist, der entsprechende Menüpunkt entfernt wird und das zugehörige Symbol nicht mehr aktiv ist. In Abbildung 5.11 ist die *Model-Tab* mit einem Beispiel einer Beschränkung zu sehen. Diese Sicht der *Model-Tab* erfolgt, wenn der Benutzer sich im zweiten Konfigurationsschritt befindet. In dieser Situation ist es dann nicht mehr möglich neue Objektypen zu definieren.

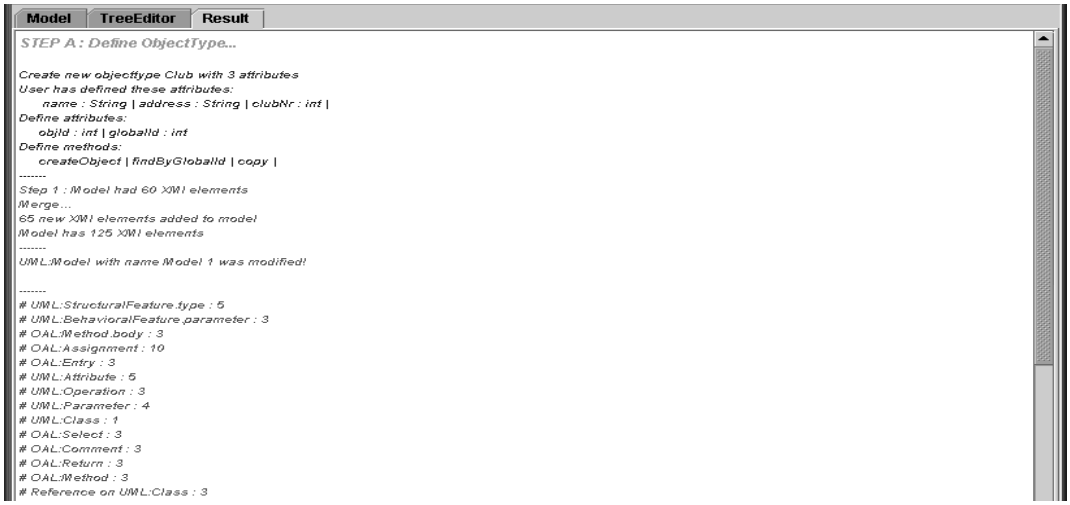
Abbildung 5.11 Model-Tab in *MTFLOW*



Die Felder *Baum (D)*, *Concrete-Transformation-Editor* (F) und *Current-Model-Editor*(G) nehmen den größten Teil der Arbeitsfläche der *TreeEditor-Tab* und bilden die zentrale Funktionalität von *MTFLOW.* Diese Felder werden hier kurz vorgestellt und deren Eigenschaften werden in den nächsten Abschnitten ausführlich erläutert. Im Baum kann der Benutzer in jedem Konfigurationsschritt seine Spezifikationen angeben. Der Benutzer kann beispielsweise im ersten Konfigurationsschritt einen neuen Objekttyp definieren. Der Name und die Attribute dieses Objekttyps werden in Baumform dargestellt und können geändert bzw. gelöscht werden. Jeder Konfigurationsschritt wird mit einem Baum assoziiert, der eine einfache Eingabe bzw. Änderung der Spezifikation ermöglicht. Aus den Elementen des Baums kann in jedem Konfigurationsschritt durch die Funktion *Show template result* automatisch mit Hilfe einer entsprechenden VTL-Vorlage eine XMI-Darstellung der Elemente erzeugt werden. Das damit erhaltene XMI-Differenz-Dokument wird dann im Concrete-Transformation-Editor angezeigt. Dieses Dokument kann anschließend mit dem alten XMI-Dokument, das eine XMI-Dargestellung der in den letzten Konfigurationsschritten bereits definierten Spezifikationen enthält, zusammengeführt. Diese Aktion erfolgt über die Funktion *Merge model and difference* oder durch Bestätigen der Schaltfläche *Merge* im Feld E der Arbeitsfläche. Das neue XMI-Modell-Dokument wird im Current-Model-Editor als XMI-Code oder XMI-Baum angezeigt.

Die dritte Registertab ist die *Result-Tab,* die allerdings nur durch Bestätigen der Schaltfläche *More Infos* im Feld E sichtbar wird. In der *Result-Tab* wird die Historie einer Anwendung angezeigt. Die Historie ist die Dokumentation eines Ablaufs einer Aktivität oder eines Workflows. In ihr werden alle wichtigen Punkte festgehalten, wie z. B. der Name und die Attribute eines im ersten Konfigurationsschritt definierten Objekttyps. Die Historie hat in *MTFLOW* zwei wichtige Funktionen: Im Fehlerfall oder im Zweifelfall kann der Ablauf der Aktivität nachvollzogen werden. Auf diesem Weg kann der Fehler behoben, indem der Benutzer die Änderungen durch die Funktion *Back to last model Version* rückgängig machen kann. Im Normalfall hilft die Historie als wichtige Informationsquelle. Sie kann beispielsweise für statistische Zwecke verwendet werden. Abbildung 5.12 zeigt die *Result-Tab* mit einem Beispiel einer Historie.

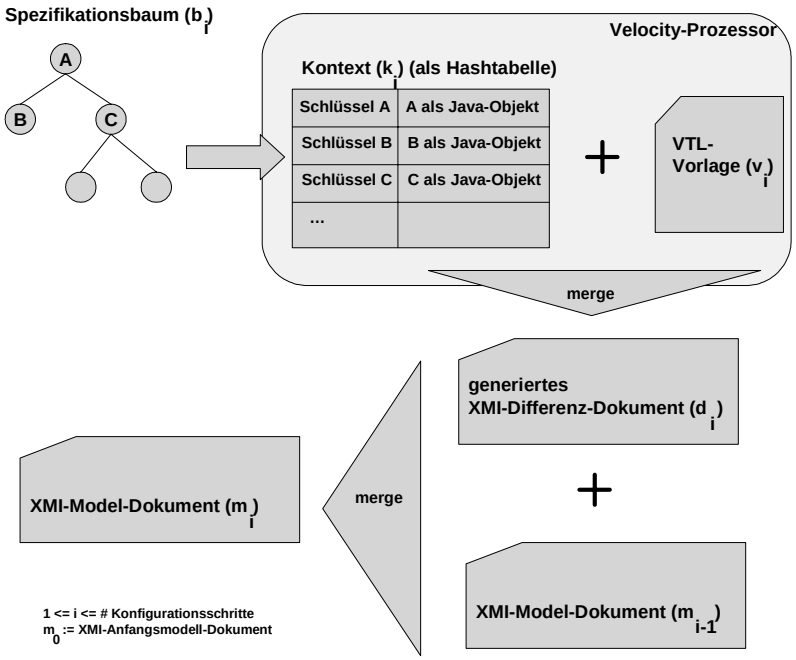
Abbildung 5.12 Result-Tab in *MTFLOW*



5.3.2 Ablauf der Modelltransformationen in MTFLOW

Die Spezifikation eines Software-Systems erfolgt in *MTFLOW* mit Hilfe aufeinanderfolgender Modelltransformationen. Am Ende jedes Konfigurationsschrittes wird ein neues XMI-Modell-Dokument erzeugt, das den aktuellen Zustand des System-Modells nach diesem Konfigurationsschritt beschreibt. Die Modelltransformationen haben einen gemeinsamen Ablauf, der in Abbildung 5.13 dargestellt ist.

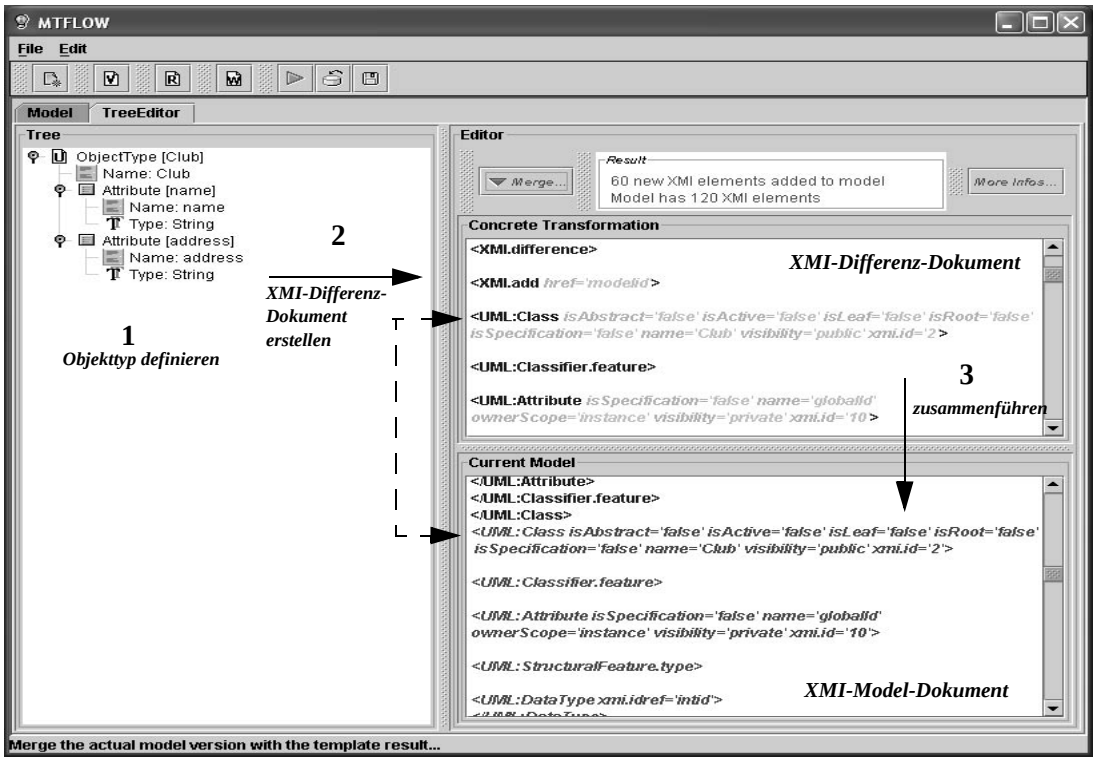
Abbildung 5.13 Genereller Ablauf der Modelltransformation



In jedem Konfigurationsschritt gibt der Benutzer seine Spezifikation in einer Baumform an. Die Knoten des Baums sind dann die Anwendungsdaten, die für die Verarbeitung der VTL-Vorlage benötigt werden. Die VTL-Vorlagen in *MTFLOW* beinhalten die statischen Anteile der zu generierenden XMI-Dokumente und Platzhalter, die bei der Verarbeitung der Vorlagen durch die tatsächlichen Anwendungsdaten ersetzt werden. Die Anwendungsdaten werden als Java-Objekte gekapselt. Diese Java-Objekte werden dann unter einem Schlüsselwort im Kontext abgelegt. Das Kontext-Prinzip basiert auf einem Daten-Container, der zwischen der Java-Schicht (dem Programmierer) und der Template-Schicht (dem Designer) bidirektional Daten transferieren kann. Der Programmierer kann ganz verschiedene Objekte in dem Kontext ablegen. Diese Objekte und deren Methoden, sowie Properties werden über Vorlagen-Elemente dem Designer zugänglich gemacht. Objekte mit primitiven Datentyp (int, double, etc...) können allerdings nicht abgelegt werden. Die Benutzung des Kontextes ist gleich wie bei einer Hashtabelle. Im Fall von *MTFLOW* werden in jedem Konfigurationsschritt die relevanten Anwendungsdaten als Java-Objekte unter einem Schlüssel im Kontext abgelegt. Für jeden Konfigurationsschritt existiert eine VTL-Vorlage. Die Vorlage enthält Referenzen, die den Zugriff auf die im Kontext bereitgestellten Anwendungsdaten ermöglichen. Die Java-Objekte, die die Anwendungsdaten darstellen

sowie deren Properties und Methoden können mit diesen Referenzen angesprochen werden. Neben diesen Referenzen enthält die Vorlage mehrere Fallunterscheidungen, Schleifen und Velocitymacros, die die Definition eines wiederholten Segmentes von VTL-Code erlauben. Bei der Merge-Aktion, die durch den Velocity-Prozessor ausgeführt wird, wird dann die Vorlage mit den im Kontext enthaltenen Anwendungsdaten zum generierten Dokument vereinigt. Damit entsteht das XMI-Differenz-Dokument, das die in diesem Konfigurationsschritt definierten Anwendungsdaten durch entsprechende XMI-Elemente beschreibt. Dieses XMI-Differenz-Dokument wird dann mit dem alten XMI-Model-Dokument zusammengeführt, das aus dem vorherigen Konfigurationsschritt enstanden ist. Das Zusammenführen beider XMI-Dokumente erfolgt mit Hilfe der XMI-Differenz-Elemente (hauptsächlich <*XMI.add*>), die in dem XMI-Differenz-Dokument definiert sind. Die XMI-Differenz-Elemente enthalten Referenzen auf XMI-Elemente aus dem XMI-Model-Dokument. Die Unter-Elemente der XMI-Differenz-Elemente im XMI-Differenz-Dokument werden beim Zusammenführen in den referenzierten Stellen des XMI-Model-Dokuments hinzugefügt. Abbildung 5.14 zeigt das Ergebnis einer möglichen Modelltransformation.

Abbildung 5.14 Beispiel einer Modelltransformation in *MTFLOW*



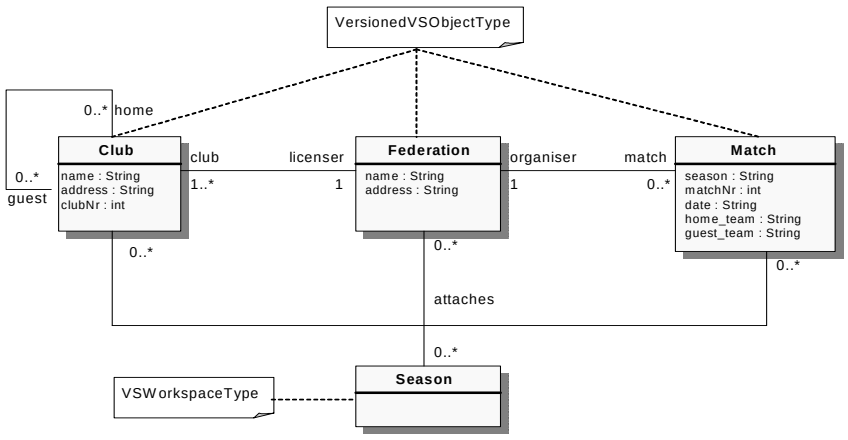
Es wird dabei im ersten Konfigurationsschritt ein neues Objekttyp *Club* mit zwei Attributen *name* und *address* definiert (Schritt 1). Aus diesen Daten wird mit Hilfe einer VTL-Vorlage ein XMI-Differenz-Dokument erzeugt und im Concrete-Transformation-Editor angezeigt (Schritt 2). Das bereits vorhandene XMI-Model-Dokument war im Current-Model-Editor zu sehen, bevor ein Zusammenführen ausgeführt wird. Nach dessen Zusammenführen mit dem XMI-Differenz-Dokument (Schritt 3) ensteht ein neues XMI-Model-Dokument, das das alte Dokument im Cur-

rent-Model-Editor ersetzt. Dabei werden die neu hinzugefügten XMI-Elemente in einer grünen Farbe dargestellt und deren Anzahl wird in einem bestimmten Feld angegeben.
Das nach einem Konfigurationsschritt erzeugte XMI-Model-Dokument enthält eine XMI-Darstellung des Modells, das bis zu diesem Konfigurationsschritt das Versionierungssystem spezifiziert. Dieses XMI-Model-Dokument wird im nächsten Konfigurationsschritt nach dem oben erklärten Modelltransformations-Prinzip erweitert. Nach dem letzten Konfigurationsschritt bekommt der Benutzer ein in XMI beschriebenes Modell seines erwünschten Systems. Nach diesem iterativen Prinzip erfolgt in *MTFLOW* die Spezifikation eines Software-Systems. Der Benutzer kann einfach in jedem Konfigurationsschritt seine Spezifikation angeben. Das System reagiert entsprechend mit einer Fehlermeldung, falls die Spezifikation nicht möglich ist. Am Ende der Konfigurationsschritte erhält der Benutzer eine Korrekte Spezifikation seines Systems. Die Spezifikations-Datei wird in XMI geschrieben und kann daher von jedem CASE-Tool importiert werden. Ein Generator kann dann aus der Spezifikation das System implementieren. Der Generator braucht dabei keine semantische Analyse der Spezifikation durchzuführen, um ihre Korrektheit zu prüfen. Er braucht auch nicht die DSL des Systems zu verstehen. Seine Aufgabe wird so stark vereinfacht, er hat nämlich nur die Aufgabe eines Übersetzers, der aus UML und OAL den Implementierungscode erzeugen soll.

5.4 Das Spezifizieren eines Versionierungssystems

MTFLOW erlaubt durch höchstens vier aufeinanderfolgende Modell-Transformationen das Spezifizieren eines Versionierungssystems. In diesem Abschnitt wird sowohl die Wirkung der Modell-Transformationen beschrieben, als auch detailliert auf deren Implementierung eingegangen. Dazu verwenden wir das Informationsmodell aus Abbildung 5.15, das ein einfaches Versionierungssystem beschreibt. Dieses Modell dient der Speicherung der Daten einer Saison irgendeiner Sportart wie z. B. Fussball oder Handball. Es sollen dabei allerdings nicht nur die Spiele der aktuellen Saison gespeichert werden, sondern auch gleichzeitig die Spiele aus zurückliegenden und zukünftigen Jahren.

Abbildung 5.15 Beispiel eines Informationsmodell für ein Versionierungssystem



MTFLOW sieht bei der Spezifikation eines Versionierungssystem vier Konfigurationsschritte vor, wie es im Workflow-Modell aus Abbildung 5.3 definiert wurde: *Define ObjectType*, *Make ObjectType versionable*, *Define RelationshipType* und *Define WorkspaceType*. In den nächsten Abschnitten werden diese Konfigurationsschritte einzeln betrachtet und die Wirkungen des Transformationsvorgangs in jedem Schritt detailliert erklärt.

5.4.1 Konfigurationsschritt A: *Define ObjectType*

In diesem Konfigurationsschritt kann der Benutzer ein neues Objekttyp mit einem Namen und mehreren Attributen definieren. Diese definition erfolgt über kontextmenüs der Elemente im Baum. Wählt der Benutzer ein Element aus dem Baum aus, öffnet sich ein Popup-Menü, das die Aktionen anbietet, die auf dem ausgewählten Element durchgeführt werden können. Kann das Element Unter-Elemente besitzen, werden zusätzlich Aktionen zum Anlegen neuer Unter-Elemente angeboten. Abbildung 5.16 zeigt dies am Beispiel eines Objekttyps. Die Generierung des XMI-Differenz-Dokumentes und dessen Zusammenführen mit dem aktuellen XMI-Modell-Dokument werden vom Benutzer über Menüpunkt oder Symbolleiste ausgelöst. Das XMI-Anfangsmodell-Dokument enthält zunächst eine XMI-Beschreibung eines UML-Modells mit einem Package, in dem die wichtigsten Datentypen definiert sind (String, int, double, etc...) und die Klassen, die in den nächsten Konfigurationsschritten häufig angesprochen werden, nämlich *ReturnCollection*, *IdCounter* und *VerIdCounter*. Die ReturnCollection-Klasse kann in einer OAL-Methode verwendet werden, um die möglichen Rückgabewerte dieser Methode zu erfassen. Die IdCounter- bzw. VerIdCounter-Klasse dienen der Vergabe neuer Ids bzw. VerIds. Die konkrete Transformation, die mit Hilfe einer geeigneten VTL-Vorlage (Abbildung 5.17 enthält einen Auszug aus dieser Vorlage) realisiert wird, erweitert das UML-Modell mit einer neuen Klasse und deren Attributen. Neben den benutzerdefinierten Attributen werden sowohl die Attribute *globalId* und *objId* als auch die Methoden *createObject*, *copy* und *findByGlobalId* hinzugefügt. Die Wirkung der Transformation ist in Abbildung 5.18 zu sehen. Abbildung 5.19 enthält als Beispiel die OAL-Beschreibung der Methode *createObject* und der entsprechende Auszug aus der VTL-Volage. Die Generierung der Methoden *copy* und *findByGlobalId* erfolgt nach dem gleichen Prinzip. Der Konfigurationsschritt A kann beliebig oft wiederholt werden, um andere Objekttypen dem Modell hinzufügen. In unserem Beispiel wird dieser Schritt insgesamt drei mal durchgeführt, um die Objekttypen *Club*, *Federation* und *Match* zu definieren.

Abbildung 5.16 Popup-menü eines Objekttyps

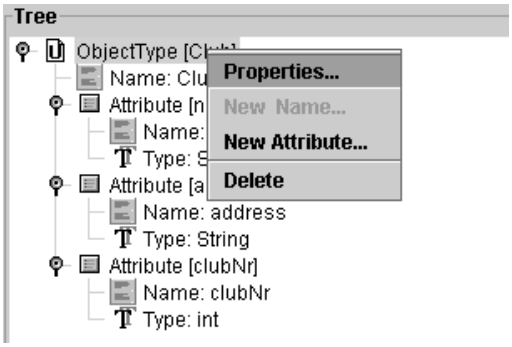


Abbildung 5.17 Auszug aus der VTL-Vorlage vom Konfigurationsschritt A

```
<XMI.difference>
<XMI.add href='modelid'>
<UML:Class xmi.id = '$ClassName.getId()' name = '$ClassName.getName()'
 visibility = 'public, isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
 <UML:Classifier.feature>
   <UML:Attribute xmi.id = '$Model.incModelId()' name = 'globalId' visibility = 'private' isSpecification = 'false' ownerScope = 'instance'>
    <UML:StructuralFeature.type>
     <UML:DataType xmi.idref = 'intid'/>
    </UML:StructuralFeature.type>
   </UML:Attribute>
   <UML:Attribute xmi.id = '$Model.incModelId()' name = 'objId' visibility = 'private' isSpecification = 'false' ownerScope = 'instance'>
    <UML:StructuralFeature.type>
     <UML:DataType xmi.idref = 'intid'/>
    </UML:StructuralFeature.type>
   </UML:Attribute>
   #foreach( $item in $ClassAttributes )
    #set($foo = $item.getType())
   <UML:Attribute xmi.id = '$item.getId()' name = '$item.getName()' visibility = 'private' isSpecification = 'false' ownerScope = 'instance'>
    <UML:StructuralFeature.type>
    #getID($foo)
     <UML:DataType xmi.idref = '$foo'/>
    </UML:StructuralFeature.type>
   </UML:Attribute>
   #end
   <UML:Operation xmi.id = '$ObjectType.getMethod("createObject").getId()' name = 'createObject'
    visibility = 'public' isSpecification = 'false' ownerScope = 'classifier' isQuery = 'false' concurrency = 'sequential' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
    <UML:BehavioralFeature.parameter>
     <UML:Parameter xmi.id = '$Model.incModelId()' name = 'return' isSpecification = 'false' kind = 'return'>
      <UML:Parameter.type>
       <UML:Class xmi.idref = '$ClassName.getId()'/>
      </UML:Parameter.type>
     </UML:Parameter>
    </UML:BehavioralFeature.parameter>
   </UML:Operation>
   <OAL:Method xmi.id = '$Model.incModelId()' xmi.idref = '$ObjectType.getMethod("createObject").getId()' ExpressionId = '$Model.incModelId()'>
    <OAL:Method.body>
    ...
     <OAL:Entry xmi.id = '$Model.incModelId()' language = 'OAL' body = '          entry//&#10;'/>
     ...
    </OAL:Method.body>
   </OAL:Method>
   ....
  </UML:Classifier.feature>
</UML:Class>
</XMI.add>
</XMI.difference>
```

Eine neue Klasse mit ihren Attributen und Methoden wird zum Modell hinzugefügt

Attribut globalId

Attribut objId

Benutzer-definierte Attribute

Methode createObject

Hier werden die Methoden copy und findByGlobalId definiert

Abbildung 5.18 Definition eines neuen Objekttyps

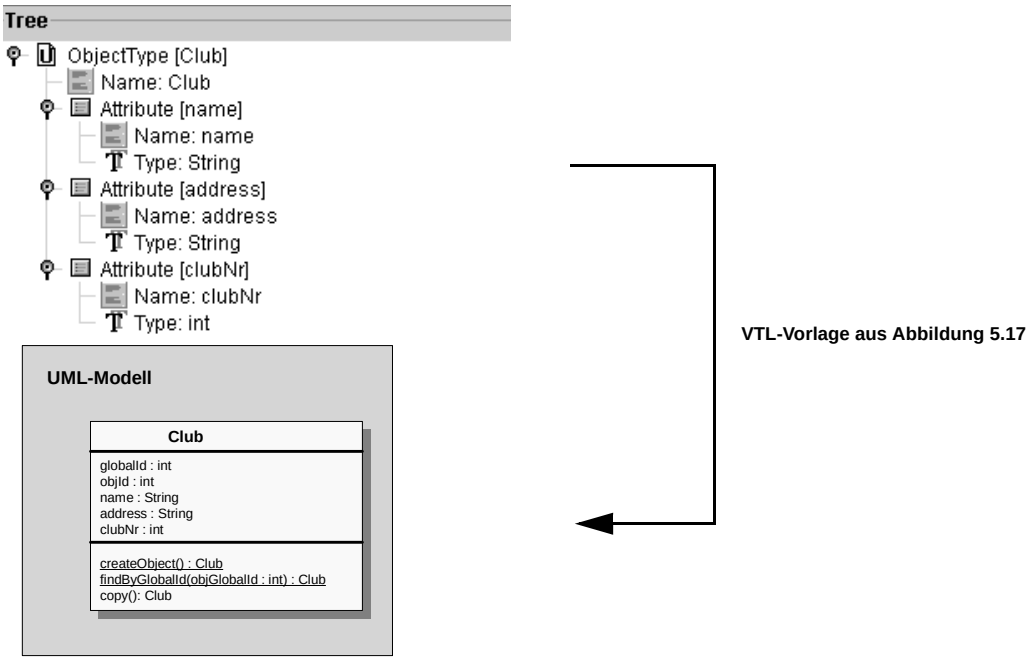
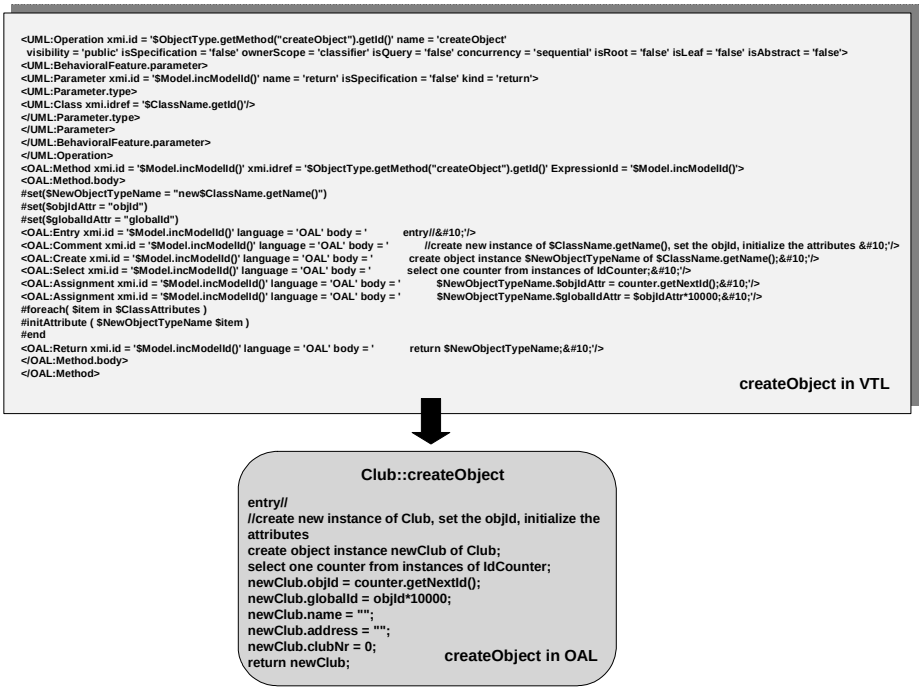


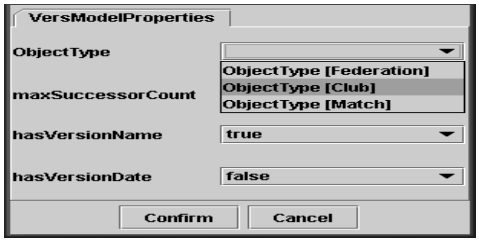
Abbildung 5.19 Generierung der Methode createObject



5.4.2 Konfigurationsschritt B: *Make ObjectType versionable*

In diesem Konfigurationsschritt spezifiziert der Benutzer die Objekttypen, die versionierbar sein können. Die Spezifikation erfolgt auch in einer Baumform. Um ein versionierbares Objekttyp zu definieren, werden in einem Einstelldialog die bereits im konfigurationsschritt A definierten Objekttypen dem Benutzer zur Verfügung gestellt (In unserem Beispiel die Objektypen Club, Federation und Match). Der Benutzer kann aus der Liste ein Objekttyp auswählen und Änderungen vornehmen. Er kann die maximale Anzahl der Nachfolgeversionen beschränken und entscheiden, ob eine neu angelegte Version des Objektyps die Attribute *versionName* und *versionDate* besitzen darf. In Abbildung 5.20 ist der Einstelldialog eines versionierbaren Objekttyps dargestellt.

Abbildung 5.20 Einstelldialog eines versionierbaren Objekttyps



Die Transformation nach diesem Konfigurationsschritt erfolgt mit Hilfe einer zweiten VTL-Vorlage, die etwas komplizierter ist als die Vorlage, die im ersten Konfigurationsschritt verwendet wurde. Die Wirkung der Transformation ist in Abbildung 5.21 dargestellt. Das <*XMI.difference*> wird in dieser Vorlage in mehreren Stellen gebraucht, da nicht nur das Modell sondern auch die Klassen und die Methoden erweitert werden sollen:

- *Erweiterung der Klassen mit neuen Attributen* - Die konkrete Transformation erweitert jede Klasse, die ein versionierbares Objekttyp darstellen sollte, mit einem Attribut, das eine konkrete Version identifiziert (*verId*) und Attributen, die den Zustand bzw. die Charakteristiken der Version beschreiben (*frozen, successorCount, versionName und versionDate*). Allerdings werden die Attribute *versionName* und *versionDate* zu einer Klasse nur dann hinzugefügt, wenn die entsprechenden Einträge *hasVersionName* und *hasVersionDate* im Einstelldialog auf *true* gesetzt wurden.
- *Erweiterung der Klassen mit neuen Methoden* - Die konkrete Transformation erweitert die Klassen mit neuen Methoden, die der Verwaltung der Versionshierarchie und der Unterstützung des Versionenzugriffs dienen (*freeze*, *createSuccessor* und *merge*), Methoden für die Navigation im Versionsbaum (*getPredecessor*, *getSuccessors*, *getAllPredecessors* und *getVersions*) und Setter-Methoden, mit denen den Benutzer-definierten Attributen neue Werte zugewiesen werden können (z. B. *setName*, *setAddress* und *setClubNr* in der Klasse *Club*).
- *Erweiterung des Modells mit neuen Beziehungen* - Die konkrete Transformation erweitert das Modell mit Beziehungen zwischen den Klassen, die versionierbare Objekttypen darstellen und der ReturnCollection-Klasse und reflexiven Beziehungen, die den Zugriff einer Version auf ihren Nachfolger- bzw. Vorgänger-Versionen erlauben.
- *Erweiterung der Methoden mit neuen Methoden-Konstrukten* - Die konkrete Transformation erweitert die Methoden *createObject* und *copy* mit neuen Anweisungen. Bei der Erweiterung werden die nach diesem Konfigurationsschritt neu eingefügten Attribute (*verId, frozen, successorCount, versionName* und *versionDate)* initialisiert.

Abbildung 5.21 Definition eines versionierbaren Objekttyps

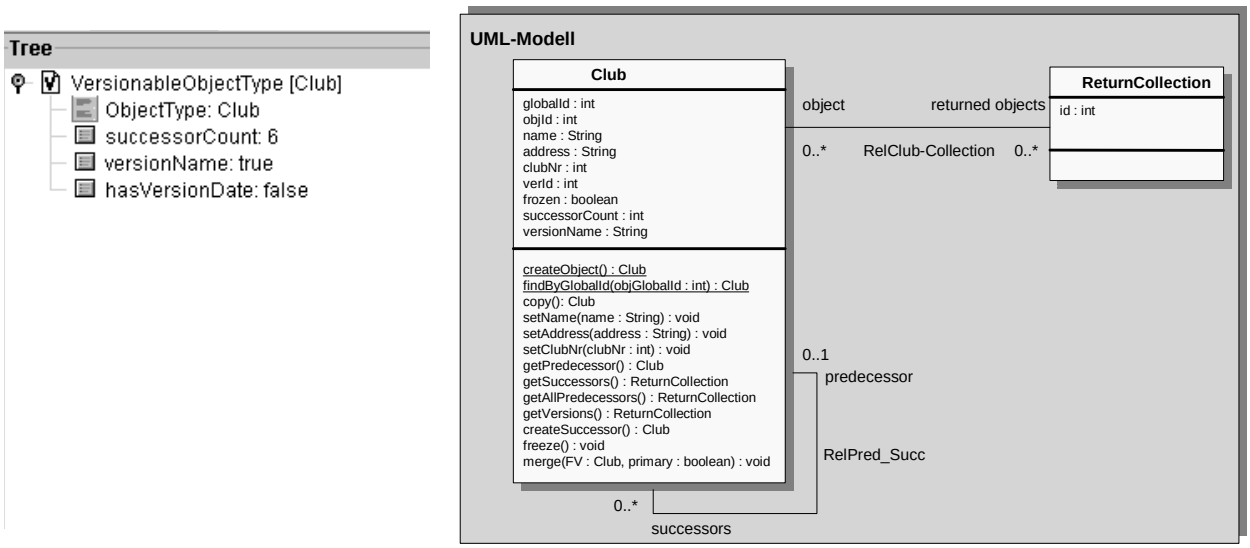


Abbildung 5.21 zeigt die Wirkung der Transformation, wenn der Benutzer beispielhaft das ObjectTyp *Club* als versionierbar definiert. Abbildung 5.22 enthält die OAL-Beschreibungen der Setter-Methoden und den entsprechenden Auszug aus der VTL-Volage, der der Generierung dieser Methoden dient.

Abbildung 5.22 Generierung der Setter-Methoden

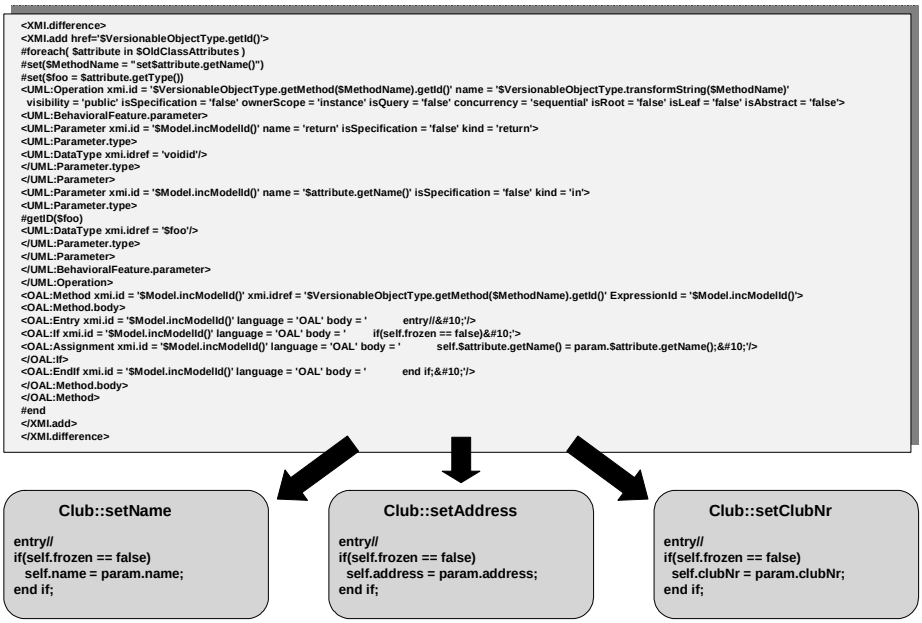


Abbildung 5.23 Einfügen einer Assoziation zum Modell

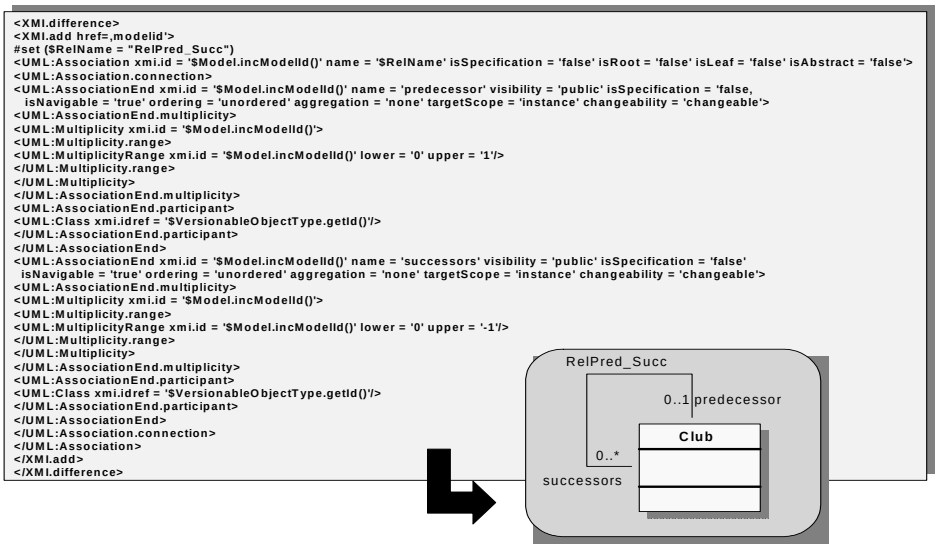
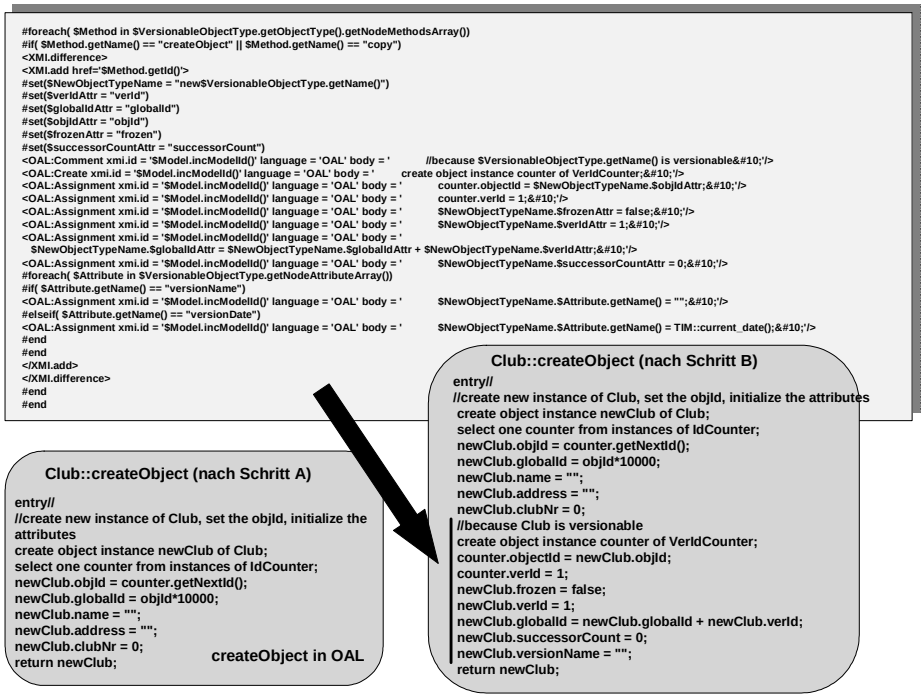


Abbildung 5.23 zeigt, wie eine Beziehung zum Modell hinzugefügt werden kann. Die Erweiterung der Methoden *createObject* und *copy* mit neuen Anweisungen ist in Abbildung 5.24 zu sehen.

Abbildung 5.24 Erweiterung der *createObject*- und *copy*-Methoden



5.4.3 Konfigurationsschritt C: *Define RelationshipType*

Zwischen Objekttypen im Informationsmodell des Versionierungssystems können Beziehungstypen definiert werden. In diesem Fall definieren die Objektypen Methoden für die Navigation entlang der Beziehungen. Der Benutzer kann eine neue binäre Beziehung definieren, in dem er über einen Einstelldialog die Eigenschaften jedes Beziehungsendes angibt. Diese Eigenschaften beinhalten Definitionen der Rollennamen, der Namen der Beziehungsenden (dafür kann der Benutzer aus einer Liste mit den bereits im ersten Konfigurationsschritt definierten Objekttypen einen auswählen), der Kardinalitäten, der gleitenden Beziehungsenden (floating relationship ends) und der zu propagierenden Operationen. Der Einstelldialog, der diese Einstellungen ermöglicht, ist in Abbildung 5.25 dargestellt. Unmögliche Einstellungen werden vom System verweigert. Versucht der Benutzer beispielsweise ein gleitendes Beziehungsende auf der Seite eines nicht versionierbaren Objekttyps zu definieren oder eine Methode in beiden Richtungen der Beziehung propagieren zu lassen, reagiert das System darauf mit einer entsprechenden Fehlermeldung. Die VTL-Vorlage, die für die Transformation in diesem Konfigurationsschritt zuständig ist, enthält eine Menge von Fallunterscheidungen, weil verschiedene Einstellungen verschiedene Verhalten der Methoden bewirken. Hier spielen besonders die Kardinalitäten und die Entscheidung, ob ein Beziehungsende gleitend ist, eine entscheidende Rolle bei der OAL-Darstellung der Methoden. Bei diesen Methoden wird z. B. im Fall eines gleitenden Beziehungsendes nicht direkt über die

Beziehung auf ein Beziehungsende zugegriffen, sondern wird zuerst auf die Kandidatenmenge zugegriffen und erst dann auf die verbundene Version des gleitenden Beziehungsendes. Die konkrete Transformation erweitert das Modell mit neuen Beziehungen und die Klassen der Objekttypen, zwischen denen eine Beziehung definiert wurde mit mehreren Methoden zur Manipulation dieser Beziehung. Die Wirkung der Transformation ist in Abbildung 5.26 dargestellt. Dabei wird eine neue Beziehung *Club-Federation* mit einem gleitenden Beziehungsende auf der *Club*-Seite definiert.

Abbildung 5.25 Einstelldialog eines Beziehungsendes

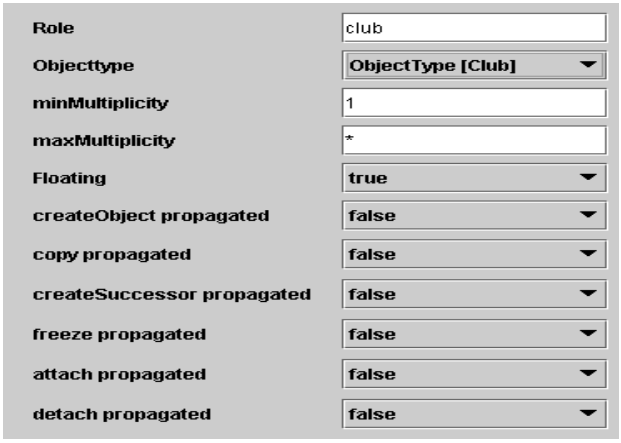
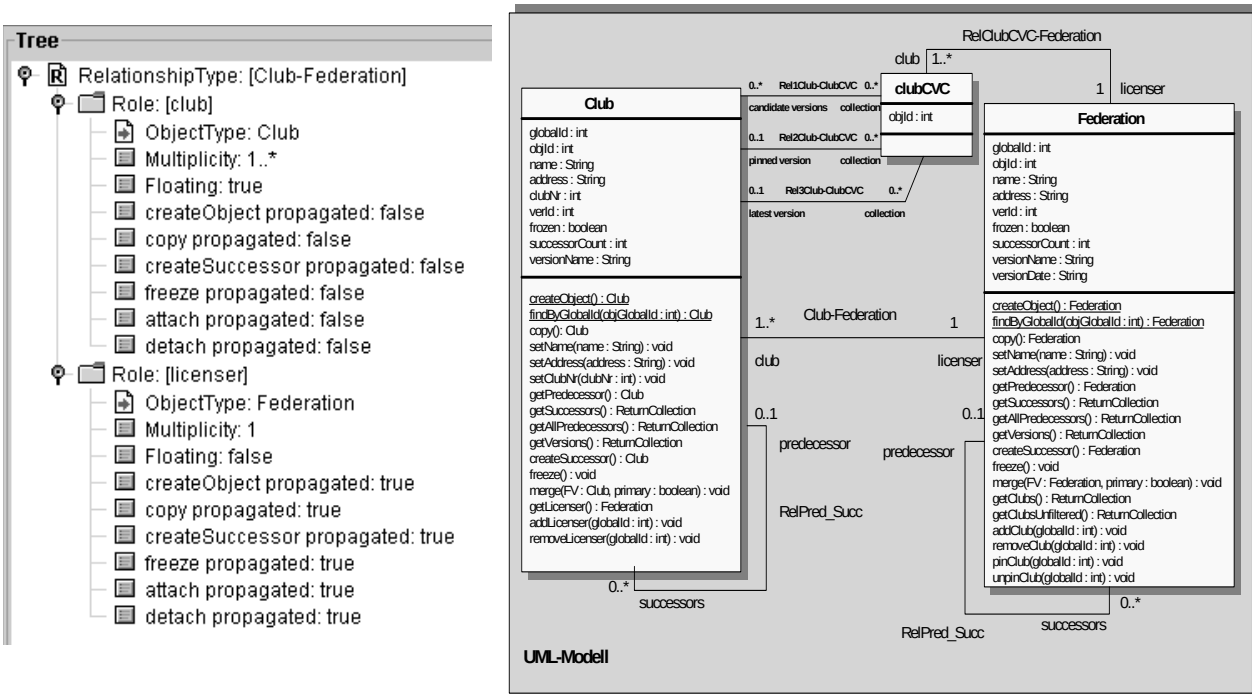


Abbildung 5.26 Definition eines Beziehungstyps



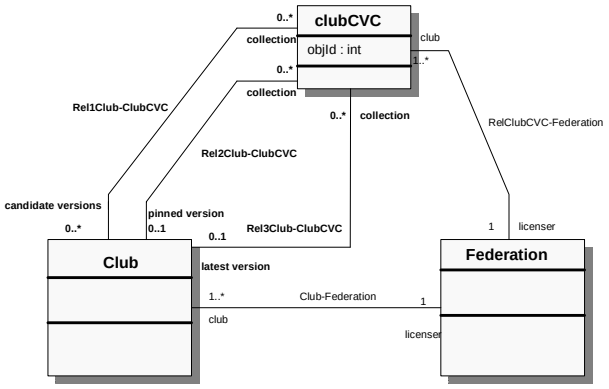
Die konkrete Transformation führt die folgenden Erweiterungen durch:

- *Erweiterung des Modells mit neuen Klassen und Beziehungen* - Die Entscheidung, welche Klassen und Beziehungen dem Modell hinzugefügt werden müssen hängt davon ab, welche Beziehungesenden gleitend definiert wurden. Die Definition eines Beziehungsendes als gleitend bewirkt die Erweiterung des Modells zunächst mit einer Klasse und vier Beziehungen, mit denen auf die Kandidatenmenge der Versionen (Candidate Version Collection, CVC) bzw. auf die versionierbaren Objekte zugegriffen werden kann. Abbildung 5.27 zeigt ein Beispiel dieser Erweiterung. Die Klasse *ClubCVC* und die Beziehungen *RelClubCVC-Federation* für den Zugriff auf die Kandidatenmenge der Versionen, *Rel1Club-ClubCVC* für den Zugriff auf die gesamte Kandidatenmenge, *Rel2Club-ClubCVC* für den Zugriff auf die bevorzugte Version in der Kandidatenmenge und *Rel3Club-ClubCVC* für den Zugriff auf die neueste Version in der Kandidatenmenge werden dem Modell hinzugefügt. Wird ein Beziehungsende als nicht gleitend definiert, wird dem Modell nur eine Beziehung hinzugefügt (z. B. die Beziehung *Club-Federation* in Abbildung 5.27), die den direkten Zugriff auf dieses Beziehungssende erlaubt. Die dargestellten Vorlagen-Auszüge beschreiben, wie die Klasse *clubCVC* und die Beziehung *RelClubCVC-Federation* zum Modell hinzugefügt werden können.

Abbildung 5.27 Erweiterung des Modells mit neuen Klassen und Beziehungen

```
#if ($RelType.getRelationshipTypeEnd1().getFloating() == true)
<XMI.difference>
<XMI.add href='modelid'>
<UML:Class xmi.id = '$RelType.getRelationshipTypeEnd1().getObjectType().getName()CVCid'
 name = '$RelType.getRelationshipTypeEnd1().getObjectType().getName()CVC' visibility = 'public'
 isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false' isActive = 'false'>
<UML:Classifier.feature>
<UML:Attribute xmi.id = '$Model.incModelId()' name = 'objId'
 visibility = 'private' isSpecification = 'false' ownerScope = 'instance'>
<UML:StructuralFeature.type>
<UML:DataType xmi.idref = 'intid'/>
</UML:StructuralFeature.type>
</UML:Attribute>
</UML:Classifier.feature>
</UML:Class>
</XMI.add>
</XMI.difference>
#end
```

```
#if ($RelType.getRelationshipTypeEnd1().getFloating() == true)
 <XMI.difference>
 <XMI.add href='modelid'>
 #set($name = "Rel$RelType.getRelationshipTypeEnd1().getObjectType().getName()CVC-
 $RelType.getRelationshipTypeEnd2().getObjectType().getName()")
 #set($end1 = "+$RelType.getRelationshipTypeEnd1().getName()")
 #set($minMult1 = $RelType.getRelationshipTypeEnd1().getMinMultiplicity())
 #set($tempMaxMul = $RelType.getRelationshipTypeEnd1().getMaxMultiplicity())
 #getMaxMult($tempMaxMul)
 #set($maxMult1 = $tempMaxMul)
 #set($participant1Id = "$RelType.getRelationshipTypeEnd1().getObjectType().getName()CVCid")
 #set($end2 = "+$RelType.getRelationshipTypeEnd2().getName()")
 #set($minMult2 = $RelType.getRelationshipTypeEnd2().getMinMultiplicity())
 #set($tempMaxMul = $RelType.getRelationshipTypeEnd2().getMaxMultiplicity())
 #getMaxMult($tempMaxMul)
 #set($maxMult2 = $tempMaxMul)
 #set($participant2Id = $RelType.getRelationshipTypeEnd2().getObjectType().getId())
 #makeAssociation($name $end1 $minMult1 $maxMult1 $participant1Id $end2 $minMult2 $maxMult2 $participant2Id)
 </XMI.add>
 </XMI.difference>
 #end
```



- *Erweiterung der Klassen mit neuen Methoden* - Die konkrete Transformation erweitert die Klassen mit neuen Methoden, die entweder der Navigation entlang der Beziehungen oder der Manipulation dieser Beziehungen dienen. Die Methoden zur Auffindung der relevanten Objektversionen im Rahmen der Navigation liefern alle Objekte, die mit einem bestimmten Objekt in Beziehung stehen, zurück. Im Fall eines gleitenden Beziehungsendes ist der Vorgang etwas aufwendiger. Es werden nämlich die zusätzlichen Beziehungen ausgewertet, die festlegen, welche die neuesten Objektversionen der Kandidatenmenge sind und welche

Objektversionen durch den Benutzer vorausgewählt (pinned) wurden. Dabei wird, falls vorhanden, die vorausgewählte Objektversion zurückgeliefert, ansonsten die neueste Objektversion. Abbildung 5.28 zeigt Ausschnitte aus der Vorlage zur Generierung der Methoden *getLicenser* und *getClubs*. In Fall eines gleitenden Beziehungsendes kommt noch eine Methode dazu, dadurch der Benutzer sich die gesamte Kandidatenmenge zurückliefern lassen kann (z. B. *getClubsUnfiltered*). Die Klassen werden auch mit Methoden erweitert, die dem Einfügen bzw. Löschen von Beziehungen dienen. Bevor eine neue Beziehung angelegt werden kann, muss zunächst für beide Enden der Beziehung überprüft werden, ob die neu einzufügende Beziehung die Kardinalitätsbedingungen der Beziehungsenden verletzen würde. Beim Anlegen einer neuen Beziehung ändert die Beteiligung eines gleitenden Beziehungsendes in dieser Beziehung die Methode, indem zuerst die Objektversion zur Kandidatenmenge des Objekts hinzugefügt werden muss und die bestehenden Beziehungen zur Kennzeichnung der neuesten Objektversionen in der Kandidatenmenge überprüft und gegenfalls geändert werden müssen. Falls keine Kandidatenmenge vorhanden war, muss eine neue erstellt werden. Beim Löschen einer Beziehung muss überprüft werden, ob die zu löschende Objektversion die neueste Version in der Kandidatenmenge war und gegenfalls die zukünftige neueste Version der Kandidatenmenge ermitteln. Die Kandiadenmenge soll auch gelöscht werden, falls sie keine andere Objektversionen mehr enthält. Das Einfügen bzw. Löschen einer Beziehung soll in beiden Richtungen erfolgen. Abbildungen 5.29 und 5.30 zeigen dies an den Beispielen von *addClub* und *removeLicenser.*

Abbildung 5.28 Generierung der Methoden *getLicenser* und *getClubs*

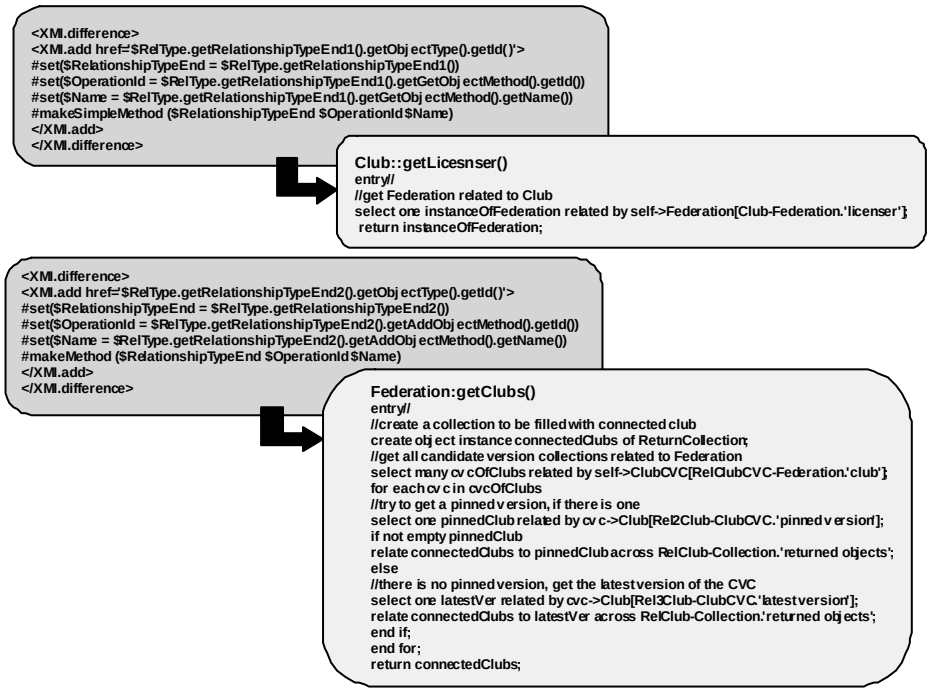


Abbildung 5.29 Einfügen einer Beziehung am Beispiel von *addClub*

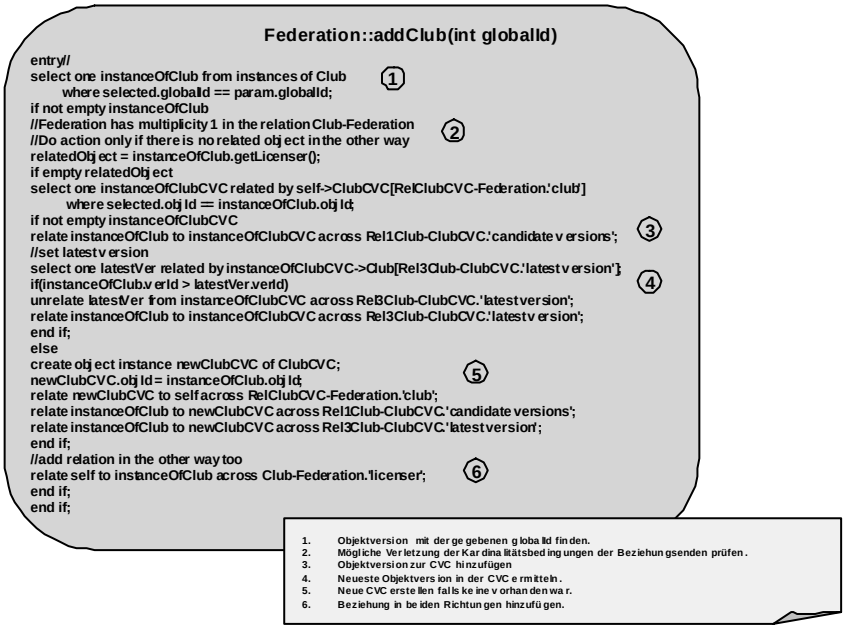
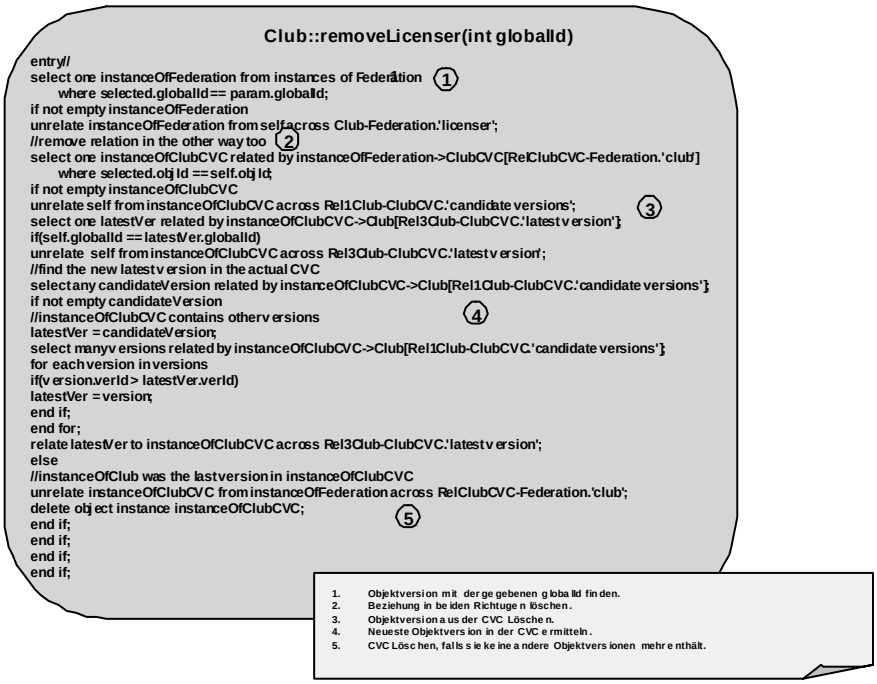


Abbildung 5.30 Löschen einer Beziehung am Beispiel von *removeLicenser*



Wird ein Beziehungsende gleitend definiert, kommen noch die Methoden pin<*Rollenname*> und unpin<*Rollenname*> dazu. Die erste Methode markiert eine Version innerhalb der Kandidatenmenge als bevorzugte Version, die dann bei der Navigation über die Beziehung genutzt wird. Dabei soll geachtet werden, dass nur eine bevorzugte Version in der Kandidatenmenge sich befindet. Die zweite Methode hebt die Markierung einer bevorzugten Version innerhalb der entsprechenden Kandidatenmenge auf. In Abbildung 5.31 sind *pinClub* und *unpinClub* Beispiele dieser Methoden. Bei der Spezifikation kann für jedes Beziehungsende bestimmt werden, welche Operationen propagieren sollen. Eine Operation darf allerdings nicht in beiden Richtungen propagieren. Die Propagierung wird rekursiv durchgeführt. Mit Hilfe der oben beschriebenen Navigationsmethoden werden die für die Propagierung relevanten verbundenen Objektversionen bestimmt. Auf diesen Objektversionen wird dann die Operation rekursiv aufgerufen. Abbildung 5.32 zeigt Ausschnitte aus der Vorlage zur Propagierung der Operation *createSuccessor* und die OAL-Beschreibung dieser Operation nach der Propagierung.

Nach diesem Konfigurationsschritt erweitert die konkrete Transformation das Modell mit mehreren Beziehungen und Methoden. Aus diesem Grund wurden Velocitymacros definiert, die die Definition eines wiederholten Segmentes von VTL-Code erlauben. Die Velocitymacros können ein Mal definiert und dann mit verschiedenen Parametern aufgerufen werden. Abbildung 5.33 enthält ein Beispiel eines Velocitymacros, das der Definition einer Beziehung dient.

Abbildung 5.31 Erweiterung der Klasse *Federation* mit den Methoden *pinClub* und *unpinClub*

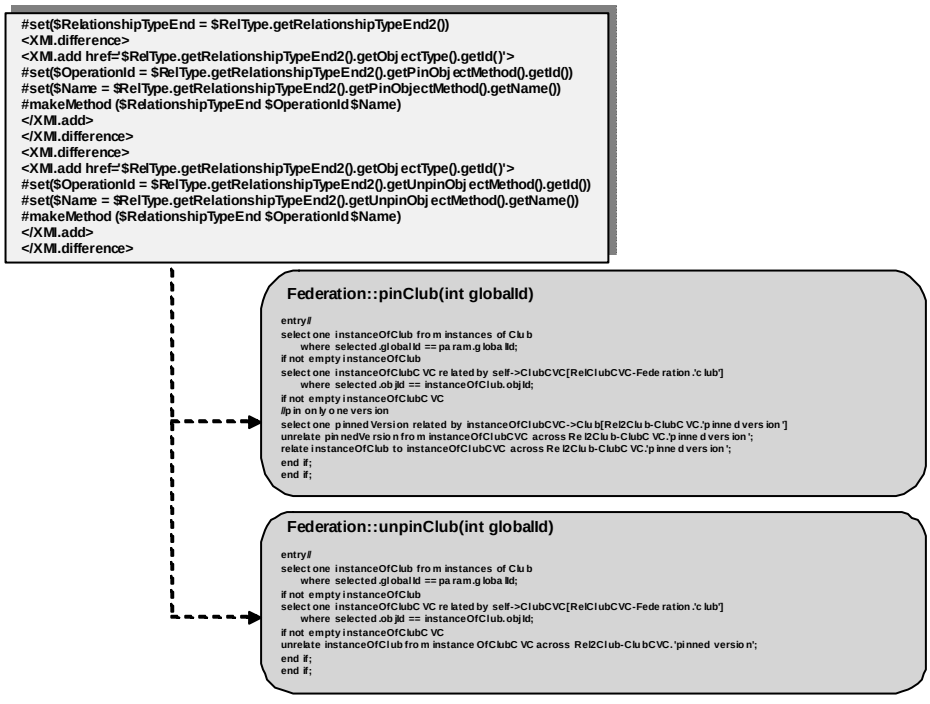


Abbildung 5.32 Propagierung einer Operation am Beispiel von *createSuccessor*

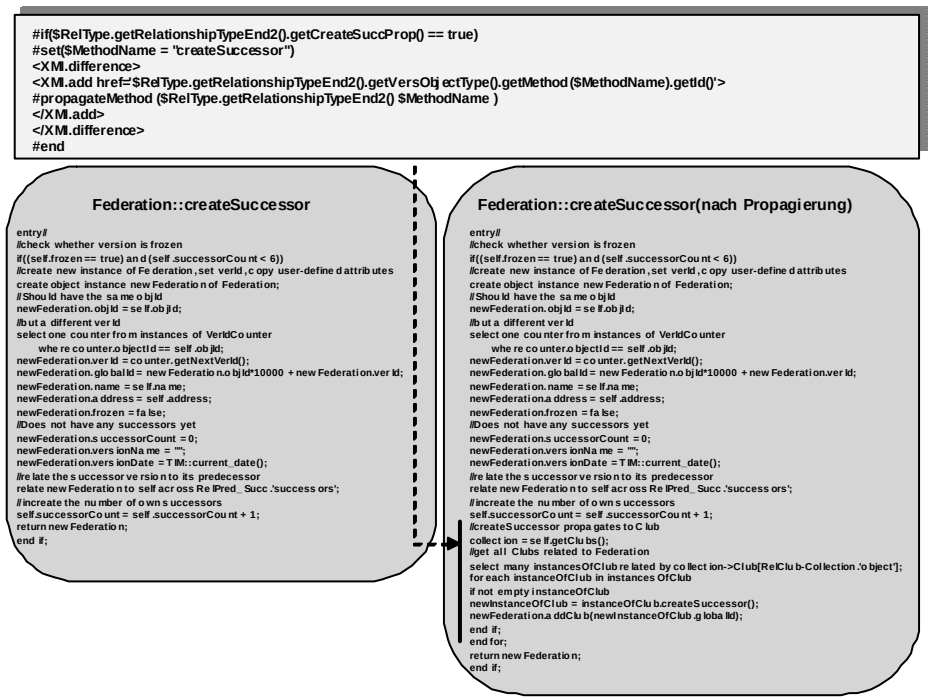
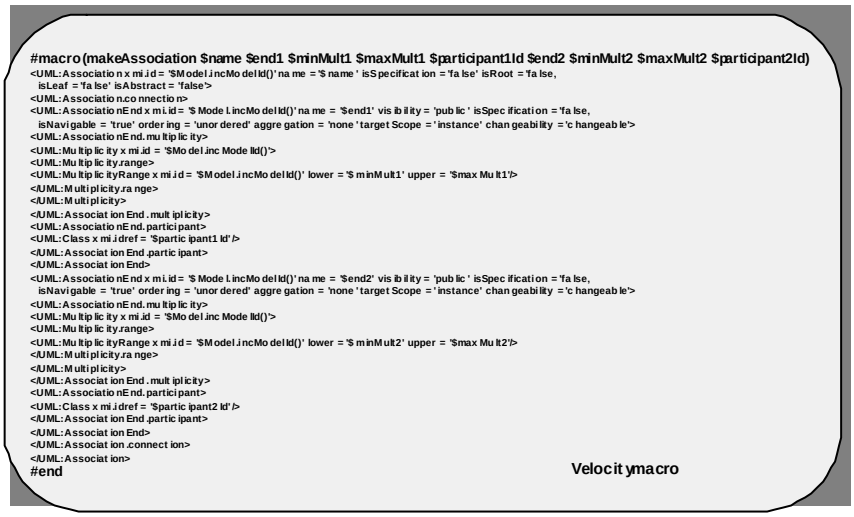


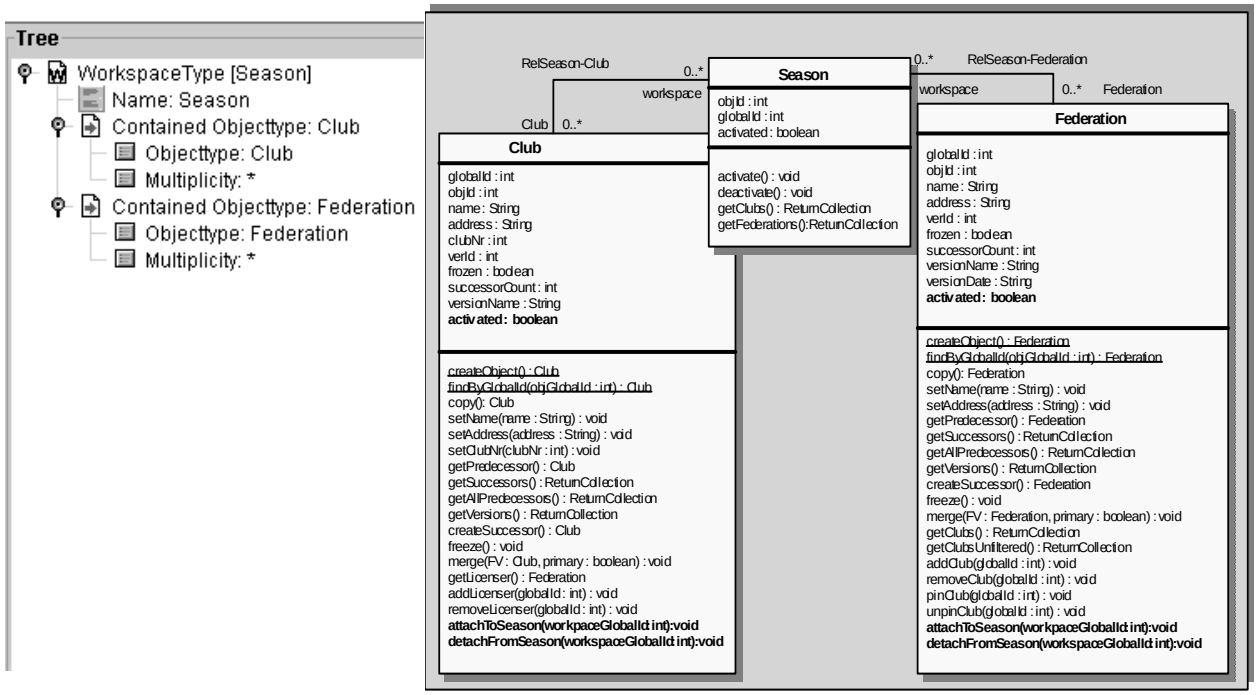
Abbildung 5.33 Velocitymacro zur Definition einer Beziehung



5.4.4 Konfigurationsschritt D: *Define WorkspaceType*

In diesem Konfigurationsschritt kann der Benutzer über einen Einstelldialog einen Arbeitskontext definieren und wählt aus, welche Objekte, mit welcher Anzahl in diesem Arbeitskontext gespeichert werden können. Zu einem Arbeitskontext kann immer nur eine Version eines Objekts hinzugefügt werden. In diesem Arbeitskontext sieht der Benutzer dann bei einer Suche nur die eingefügten Objektversionen und gelangt bei der Verfolgung einer Beziehung automatisch nur zu einer Version des Zielobjekts, die ebenfalls im gleichen Arbeitskontext liegt. Dabei werden das Pinning bzw. die sonstigen Auswahlregeln einer Version nicht beachtet. Die konkrete Transformation erweitert das Modell anhand einer vierten VTL-Vorlage mit einer neuen Klasse für den Arbeitskontexttyp und Beziehungen zwischen dem Arbeitskontexttyp und den in ihm gespeicherten Objekttypen. Die Klassen der Objekttypen werden mit neuen Attributen und Methoden erweitert. Die Wirkung der Transformation ist in Abbildung 5.34 dargestellt.

Abbildung 5.34 Definition eines Arbeitskontexttyps



Das Modell wird mit einer Klasse für den definierten Arbeitsskontexttyp erweitert. Diese Klasse enthält dann die üblichen Attribute *objId* und *globalId* sowie ein weiteres Attribut *activated* zum Markieren des Arbeitsskontextes als ausgewählt. Die Methoden *activate* und *deactivate*, die dem Benutzer den Zugriff auf einen Arbeitskontext und die in ihm enthaltenen Objekte ermöglichen und die Getter-Methoden, um diese Objekte bei Anfrage zurückzuliefern sind auch Bestandteile dieser Klasse. Abbildung 5.35 zeigt das Beispiel der Methoden *Season::getClubs* und *Season::getFederations* und den Auszug aus der Vorlage zur Generiereung dieser Methoden. Zwischen dieser Klasse und der Klassen der in diesem Arbeitskontexttyp sich befindenden Objektypen entstehen neue Beziehungen (z. B. *RelSeasonClub* und *RelSeasonFederation* in Abbildung 5.34). Der Vorlagen-Auszug aus Abbildung 5.36 ist für die Generierung dieser Bezie-

hungen zuständig. Wird ein Objekttyp zu einem Arbeitskontext hinzugefügt, wird seine Klasse mit dem Attribut *activated* und den Methoden *attachTo<Arbeitskontextname>* und *detachFrom<Arbeitskontextname>*, mit denen eine Objektversion zum Arbeitskontext hinzugefügt bzw. aus dem Arbeitskontext entfernt werden kann, erweitert. Die im vorherigen Konfigurationsschritt eingeführten Navigationsmethoden (z. B. *getClubs* und *getLicenser*) werden nach der Definition eines Arbeitskontexttyps so erweitert, dass zunächst überprüft wird, ob ein Arbeitskontext von diesem Arbeitskontextyp aktiviert ist. Ist dies der Fall, werden aus der Kandidatenmenge alle diejenigen Objektversionen ausgewählt, die sich im aktuellen Arbeitskontext befinden. Dabei wird automatisch nur eine Objektversion pro Objekt ausgewählt, da in einem Arbeitskontext per Definition nur eine Objektversion pro Objekt enthalten sein kann. Abbildung 5.37 zeigt dies am Beispiel von *getClubs*. Erst in diesem Konfigurationsschritt werden Attach/Detach-Operationen in den Klassen der Objekttypen definiert. Diese Operationen können propagieren. Die Propagierung wird hier auch rekursiv durchgeführt. In Abbildung 5.38 ist die Propagierung der Operation *Federation::attachToSeason* dargestellt. In der Vorlage, die die Transformation nach diesem Konfigurationsschritt bestimmt, werden auch velocitymacros definiert wie z. B. expandMethod zur Erweiterung der Navigationsmethoden, propagateMethod zur Erweiterung der Attach/Detach-Operationen nach Propagierung und getObjectInWS zum Zugriff auf die Objekte in einem Arbeitskontext.

Abbildung 5.35 Generierung der Getter-Methoden in einem Arbeitskontext

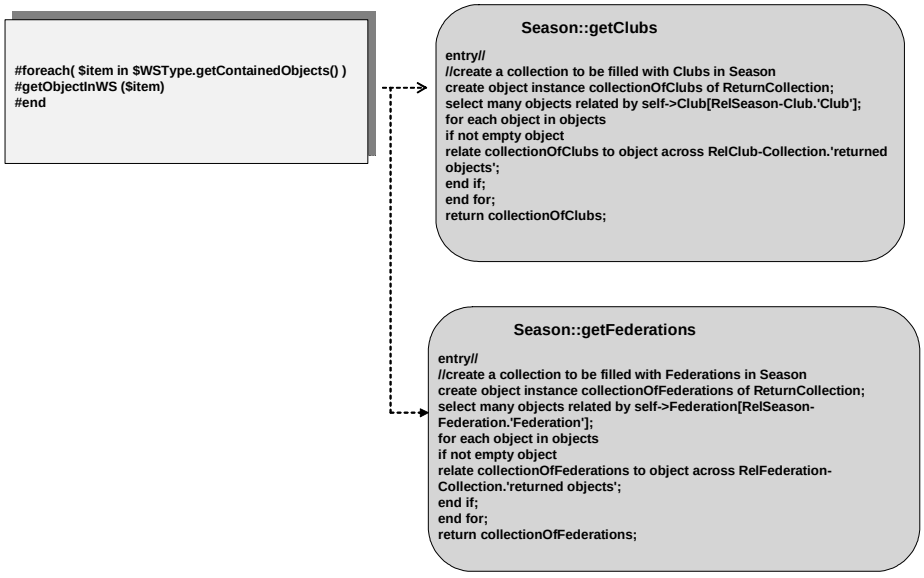


Abbildung 5.36 Generierung der Beziehung zwischen Arbeitskontexttyp und Objekttyp

```
<XMI.difference>
<XMI.add href='modelid'>
#foreach( $item in $WSType.getContainedObjects() )
<UML:Association xmi.id = '$Model.incModelId()' name = 'Rel$WSType.getName()-$item.getName()' isSpecification = 'false' isRoot = 'false' isLeaf = 'false' isAbstract = 'false'>
<UML:Association.connection>
<UML:AssociationEnd xmi.id = '$Model.incModelId()' name = '+workspace' visibility = 'public'
   isSpecification = 'false' isNavigable = 'true' ordering = 'unordered' aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
<UML:AssociationEnd.multiplicity>
<UML:Multiplicity xmi.id = '$Model.incModelId()'>
<UML:Multiplicity.range>
<UML:MultiplicityRange xmi.id = '$Model.incModelId()' lower = '0' upper = '-1'/>
</UML:Multiplicity.range>
</UML:Multiplicity>
</UML:AssociationEnd.multiplicity>
<UML:AssociationEnd.participant>
<UML:Class xmi.idref = '$WSType.getId()'/>
</UML:AssociationEnd.participant>
</UML:AssociationEnd>
<UML:AssociationEnd xmi.id = '$Model.incModelId()' name = '+$item.getName()' visibility = 'public'
   isSpecification = 'false' isNavigable = 'true' ordering = 'unordered' aggregation = 'none' targetScope = 'instance' changeability = 'changeable'>
<UML:AssociationEnd.multiplicity>
<UML:Multiplicity xmi.id = '$Model.incModelId()'>
<UML:Multiplicity.range>
#set($tempMaxMul = $item.getMaxMultiplicity())
#getMaxMult($tempMaxMul)
<UML:MultiplicityRange xmi.id = '$Model.incModelId()' lower = '$item.getMinMultiplicity()' upper = '$tempMaxMul'/>
</UML:Multiplicity.range>
</UML:Multiplicity>
</UML:AssociationEnd.multiplicity>
<UML:AssociationEnd.participant>
<UML:Class xmi.idref = '$item.getObjectType().getId()'/></UML:AssociationEnd.participant>
</UML:AssociationEnd>
</UML:Association.connection>
</UML:Association>
#end
</XMI.add>
</XMI.difference>
```

Abbildung 5.37 Erweiterung der Navigationsmethoden

```
#if($Relation.getRelationshipTypeEnd1().getFloating() == true)
#set($RelationshipTypeEnd = $Relation.getRelationshipTypeEnd2())
#set($OperationId = $Relation.getRelationshipTypeEnd2().
getGetObjectMethod().getId())
#expandMethod ($RelationshipTypeEnd $OperationId)
#end
```

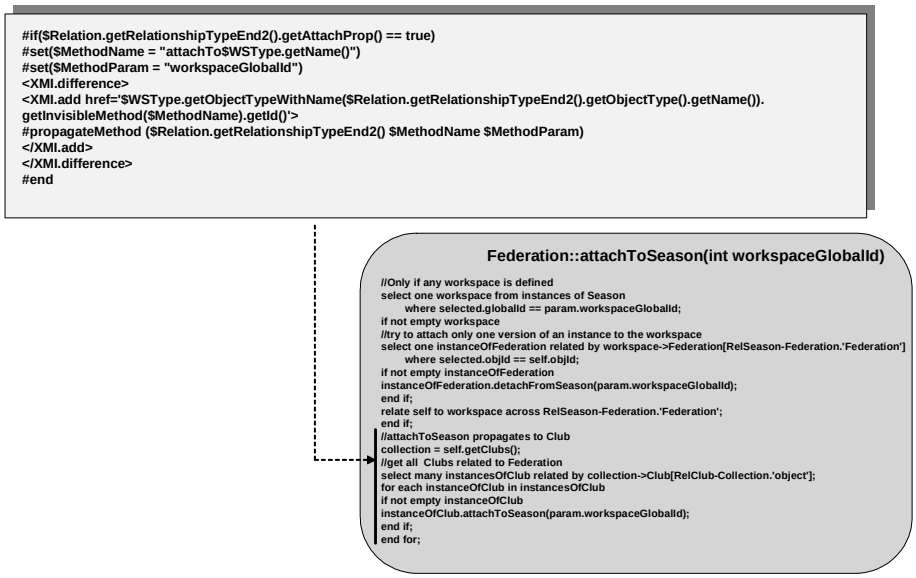
Federation::getClubs() (nach Schritt C)

```
entry//
//create a collection to be filled with connected club
create object instance connectedClubs of ReturnCollection;
//get all candidate version collections related to Federation
select many cvcOfClubs related by self->ClubCVC[RelClubCVC-Federation.'club'];
for each cvc in cvcOfClubs
//try to get a pinned version, if there is one
select one pinnedClub related by cvc->Club[Rel2Club-ClubCVC.'pinned version'];
if not empty pinnedClub
relate connectedClubs to pinnedClub across RelClub-Collection.'returned objects';
else
//there is no pinned version, get the latest version of the CVC
select one latestVer related by cvc->Club[Rel3Club-ClubCVC.'latest version'];
relate connectedClubs to latestVer across RelClub-Collection.'returned objects';
end if;
end for;
return connectedClubs;
```

Federation::getClubs() (nach Schritt D)

```
entry//
//create a collection to be filled with connected club
create object instance connectedClubs of ReturnCollection;
if(self.activated == false)
//get all candidate version collections related to Federation
select many cvcOfClubs related by self->ClubCVC[RelClubCVC-Federation.'club'];
for each cvc in cvcOfClubs
//try to get a pinned version, if there is one
select one pinnedClub related by cvc->Club[Rel2Club-ClubCVC.'pinned version'];
if not empty pinnedClub
relate connectedClubs to pinnedClub across RelClub-Collection.'returned objects';
else
//there is no pinned version, get the latest version of the CVC
select one latestVer related by cvc->Club[Rel3Club-ClubCVC.'latest version'];
relate connectedClubs to latestVer across RelClub-Collection.'returned objects';
end if;
end for;
else
//Only if any workspace is defined
while(true)
// get Workspace Season
select one workspace related by self->Season[RelSeason-Federation.'workspace']
      where selected.activated == true;
if not empty workspace
//get all candidate version collections related to Federation
select many cvcOfClubs related by self->ClubCVC[RelClubCVC-Federation.'club'];
for each cvc in cvcOfClubs
//try to get the related version in this workspace
select many candidateVersionsOfClub related by cvc->Club[Rel1Club-
ClubCVC.'candidate versions'];
for each candidateVersionOfClub in candidateVersionsOfClub
select one instanceOfClubInWS related by workspace->Club[RelSeason-Club.'Club']
      where selected.globalId == candidateVersionOfClub.globalId;
if not empty instanceOfClubInWS
relate connectedClubs to instanceOfClubInWS  across RelClub-Collection.'returned
objects';
break;
end if;
end for;
end for;
break;
end if;
end while;
end if;
return connectedClubs;
```

Abbildung 5.38 Propagierung der Attach-Operation



5.5 Der OAL-Parser in *MTFLOW*

Wie in den letzten Abschnitten zu erkennen war, können nicht nur Modelle mit neuen Klassen und Beziehungen oder Klassen mit neuen Attributen und Methoden sondern auch Methoden mit neuen Konstrukten erweitert werden. Die OAL-Methoden können in jedem Konfigurationsschritt durch die Modelltransformation geändert werden und die Operationen können über Beziehungen propagieren. Aus diesem Grund sollte eine Lösung gefunden werden, um die einfache Manipulation der OAL-Methoden zu ermöglichen. Es wäre möglich, den Inhalt einer OAL-Methode als eine Zeichenkette zu betrachten und dementsprechend zu manipulieren. Sinnvoller ist aber die XMI-basierte Beschreibung der OAL-Methoden zu beachten und die <XMI.difference>-Elemente, auf denen die Modell-Transformation in MTFLOW basiert, zu verwenden, um neue Konstrukte den Methoden hinzufügen. Daher wurden für die verschiedenen Konstrukte der OAL-Methoden neue XMI-Elemente definiert. Diese wurden bereits in Kapitel 4 vorgestellt. Damit das am Ende der Konfiguration enstandene XMI-Modell-Dokument, welches eine XMI-Spezifikation des Software-Systems enthält, von einem CASE-Tool importiert werden kann, wird ein Parser implementiert, der aus der XMI-Darstellung einer OAL-Methode mit den neu-eingeführten XMI-Elementen eine XMI-Darstellung erzeugen soll, die von jedem CASE-Tool akzeptiert wird. Abbildung 5.39 zeigt das Ergebnis dieser Transformation bei der Methode *createSuccessor.*

Abbildung 5.39 Transformation der Methode *createSuccessor* mit dem OAL-Parser

```
<UML:Operation concurrency='sequential' isAbstract='false' isLeaf='false' isQuery='false' isRoot='false' isSpecification='false' name='createSuccessor'
  ownerScope='instance' visibility='public' xmi.id='52'>
<UML:BehavioralFeature.parameter>
<UML:Parameter isSpecification='false' kind='return' name='return' xmi.id='132'>
<UML:Parameter.type>
<UML:Class xmi.idref='2'>
</UML:Class>
</UML:Parameter.type>
</UML:Parameter>
</UML:BehavioralFeature.parameter>
</UML:Operation>
<OAL:Method ExpressionId='134' xmi.id='133' xmi.idref='52'>
<OAL:Method.body>
<OAL:Entry body='           entry//' language='OAL' xmi.id='135'></OAL:Entry>
<OAL:Comment body='        //check whether version is frozen' language='OAL' xmi.id='136'></OAL:Comment>
<OAL:If body='          if((self.frozen == true) and (self.successorCount < 6))' language='OAL' xmi.id='137'>
<OAL:Comment body='        //create new instance of Club, set verId, copy user-defined attributes' language='OAL' xmi.id='138'></OAL:Comment>
<OAL:Create body='        create object instance newClub of Club;' language='OAL' xmi.id='139'></OAL:Create>
<OAL:Comment body='        //Should have the same objId' language='OAL' xmi.id='140'></OAL:Comment>
<OAL:Assignment body='        newClub.objId = self.objId;' language='OAL' xmi.id='141'></OAL:Assignment>
<OAL:Comment body='        //but a different verId' language='OAL' xmi.id='142'></OAL:Comment>
<OAL:Select body='          select one counter from instances of VerIdCounter
          where counter.objectId == self.objId;' language='OAL' xmi.id='143'></OAL:Select>
<OAL:Assignment body='        newClub.verId = counter.getNextVerId();' language='OAL' xmi.id='144'></OAL:Assignment>
<OAL:Assignment body='        newClub.globalId = newClub.objId*10000 + newClub.verId;' language='OAL' xmi.id='145'></OAL:Assignment>
<OAL:Assignment body='        newClub.name = self.name;' language='OAL' xmi.id='146'></OAL:Assignment>
<OAL:Assignment body='        newClub.address = self.address;' language='OAL' xmi.id='147'></OAL:Assignment>
<OAL:Assignment body='        newClub.clubNr = self.clubNr;' language='OAL' xmi.id='148'></OAL:Assignment>
<OAL:Assignment body='        newClub.frozen = false;' language='OAL' xmi.id='149'></OAL:Assignment>
<OAL:Comment body='        //Does not have any successors yet' language='OAL' xmi.id='150'></OAL:Comment>
<OAL:Assignment body='        newClub.successorCount = 0;' language='OAL' xmi.id='151'></OAL:Assignment>
<OAL:Assignment body='        newClub.versionName = "";' language='OAL' xmi.id='152'></OAL:Assignment>
<OAL:Comment body='        //relate the successor version to its predecessor' language='OAL' xmi.id='153'></OAL:Comment>
<OAL:Relate body='          relate newClub to self across RelPred_Succ.'successors';' language='OAL' xmi.id='154'></OAL:Relate>
<OAL:Comment body='        //increate the number of own successors' language='OAL' xmi.id='155'></OAL:Comment>
<OAL:Assignment body='        self.successorCount = self.successorCount + 1;' language='OAL' xmi.id='156'></OAL:Assignment>
</OAL:If>
<OAL:Return body='          return newClub;' language='OAL' xmi.id='157'></OAL:Return>
<OAL:EndIf body='        end if;' language='OAL' xmi.id='158'></OAL:EndIf>
</OAL:Method.body>
</OAL:Method>
```

```
<UML:Operation concurrency='sequential' isAbstract='false' isLeaf='false' isQuery='false' isRoot='false' isSpecification='false' name='createSuccessor'
  ownerScope='instance' visibility='public' xmi.id='52'>
<UML:BehavioralFeature.parameter>
<UML:Parameter isSpecification='false' kind='return' name='return' xmi.id='132'>
<UML:Parameter.type>
<UML:Class xmi.idref='2'></UML:Class>
</UML:Parameter.type>
</UML:Parameter>
</UML:BehavioralFeature.parameter>
</UML:Operation>
<UML:Method isQuery='false' isSpecification='false' xmi.id='133'>
<UML:Method.body><UML:ProcedureExpression body='entry//&amp;#10;//check whether version is frozen&amp;#10;if((self.frozen == true) and (self.successorCount &amp;lt; 6))
&amp;#10;//create new instance of Club, set verId, copy user-defined attributes&amp;#10;create object instance newClub of Club;
&amp;#10;//Should have the same objId&amp;#10;newClub.objId = self.objId;&amp;#10;//but a different verId
&amp;#10;select one counter from instances of VerIdCounter&amp;#10;          where counter.objectId == self.objId;&amp;#10;newClub.verId = counter.getNextVerId();
&amp;#10;newClub.globalId = newClub.objId*10000 + newClub.verId;&amp;#10;newClub.name = self.name;&amp;#10;newClub.address = self.address;
&amp;#10;newClub.clubNr = self.clubNr;&amp;#10;newClub.frozen = false;&amp;#10;//Does not have any successors yet
&amp;#10;newClub.successorCount = 0;&amp;#10;newClub.versionName = "";
&amp;#10;//relate the successor version to its predecessor&amp;#10;relate newClub to self across RelPred_Succ.&amp;apos;successors&amp;apos;;
&amp;#10;//increate the number of own successors&amp;#10;self.successorCount = self.successorCount + 1;&amp;#10;return newClub;&amp;#10;end if;&amp;#10;' language='java' xmi.id='134'>
</UML:ProcedureExpression>
</UML:Method.body>
<UML:Method.specification>
<UML:Operation xmi.idref='52'>
</UML:Operation>
</UML:Method.specification>
</UML:Method>
```

Kapitel 6 Empirische Untersuchungen

Die XMI-basierten und die durch VTL-Vorlagen automatisierten Modelltransformationen werden in diesem Kapitel nach verschiedenen Kriterien bewertet. Die Bewertung soll hauptsächlich die Frage beantworten können, welcher Zusammenhang bei der Spezifikation eines Software-Systems zwischen dem Umfang des generierten XMI-Model-Dokumentes, das die XMI-Beschreibung des Systems enthält und dem Aufwand der Spezifikation dieses Systems besteht.

6.1 Bewertung durch Metriken

In diesem Abschnitt werden zunächst einige Metriken vorgestellt, mit denen anschließend die Ergebnisse der automatisierten Modelltransformationen bewertet werden.

6.1.1 Bewertung durch Softwaremetriken

Zur Abschätzung des Entwicklungsaufwands stehen viele Metriken zur Verfügung. Die einfachste und häufig verwendete Metrik, die bei der Messung der Größe von Quellcode eingesetzt werden kann, ist die Zählung der Quellcodezeilen (Lines of Code, LOC). Dabei wird ganz einfach die Anzahl der Quellcodezeilen eines Programms als Maß für dessen Größe herangezogen. Leerzeilen und Kommentarzeilen werden im allegemeinen mitgerechnet. Dieses Maß hängt deshalb sehr stark von der Formatierung des Quellcodes ab und hat nur eine beschränkte Aussagekraft über komplexe Kontrollflüsse im Programm. Zur Abschätzung der Komplexität der Vorlagen wurden daher drei aussagekräftigere Metriken ausgewählt: Die Anzahl der Anweisungen, die zyklomatische Komplexität und der Halstead-Aufwand.

Anzahl der Anweisungen

Der Quellcode kann neben dem eigentlichen Programm auch Leerzeilen und Kommentare enthalten. Diese können die Messung der Aufwandsabschätzung eines Programms stark beeinflussen. Eine Zählung der Quellcodeanweisungen anstelle von Quellcodezeilen ist daher sinnvoller. Es werden nach einer notwendigen Syntaxanalyse nur die im Quellcode enthaltenen Anweisungen gezählt, um den Einfluss der Formatierung des Quellcodes umzugehen.

Zyklomatische Komplexität

Das obige Maß berücksichtigt nicht die Komplexität der Kontrollflüsse im Programm. Die zyklomatische Komplexität nach McCabe [McC76] tut das, indem sie die Anzahl der Bedingungen in die Berechnung des Aufwands eines Programms mit einbezieht. Die Formel für die zyklomatische Komplexität ist V(G) = e - n + 2; e ist die Anzahl der Kanten des Kontrollflussgraphen G, n ist die Anzahl der Knoten. Je größer diese Zahl ist, desto komplexer und schwierigerer ist ein Programm zu warten und testen, da mehr Testfälle konstruiert werden müssen, um alle Kontrollpfade abzudecken.

Halstead-Aufwand

Die Komplexität eines Programms ist nicht nur von der Struktur des Kontrollgraphen abhängig, sondern auch von der Anzahl der zu verarbeitenden Eingabedaten und der Anzahl der dazu verwendeten Operationen. Halstead [Hal77] sah Programmieren als einen nicht festgelegten Prozess des Selektierens von Operatoren und Operanden aus einer vorher festgelegten Liste und definierte Metriken, die auf den folgenden Variablen basieren:

n_1	Anzahl unterschiedlicher Operatoren
n_2	Anzahl unterschiedlicher Operanden
N_1	Gesamtanzahl Operatoren
N_2	Gesamtanzahl Operanden

Daraus leitet Halstead folgende Metriken ab:

Länge (N)	$N = N_1 + N_2$
Vokabular (n)	$n = n_1 + n_2$
Volumen(V)	$V = N * log2(n)$
Schwierigkeit (D)	$D = \frac{n_1 * N_2}{2 * n_2}$
Aufwand (E)	$E = D * V$

6.1.2 Bewertung durch *MTFLOW*-spezifische Metriken

Damit die Ergebnisse der automatisierten Modelltransformationen bewertet werden können, wird eine zusätzliche Metrik eingeführt. Diese Metrik basiert auf der Anzahl der XMI-Elemente, die durch die eingesetzten VTL-Vorlagen bei den verschiedenen Transformationen generiert werden. Das Maß *XEPR(i) (XMI Element Production Rate für ein gegebenes System i)* ist durch die folgende Formel definiert: *XEPR(i) = x(i) / v(i)*, wobei *x(i)* die gesamte Anzahl der generierten XMI-Elemente und *v(i)* die Anzahl der Benutzer-Angaben ist, die bei der Spezifikation des Systems vom Benutzer definiert bzw. ausgewählt werden.

6.2 Analyse der VTL-Vorlagen

In diesem Abschnitt werden die VTL-Vorlagen analysiert, die bei der Konfiguration eines Versionierungssystems verwendet werden. In *MTFLOW* werden vier Vorlagen implementiert, die zu der konkreten Transformation in jedem Konfigurationsschritt dienen. Jede Vorlage enthält Platzhalter für die Benutzer-definierten Parameter, die in jedem Konfigurationsschritt definiert bzw. ausgewählt werden. Aus diesen Parametern wird dann durch diese Vorlage ein XMI-Differenz-Dokument erzeugt, das in den nächsten Konfigurationsschritten mit Hilfe von anderen Vorlagen nach dem gleichen Prinzip erweitert werden kann. Die vier VTL-Vorlagen wurden mit Hilfe des

Analysewerkzeuges [Vir03] analysiert, das bei der automatischen Analyse von Java-Programmen verwendet werden kann. Um auch die Komplexität der Vorlagen mit diesem Werkzeug analysieren zu können, musste der VTL-Code zunächst in äquivalenten Java-Code umgewandelt werden. Abbildung 6.1 zeigt ein Beispiel solcher Umwandlung. In Abbildung 6.2 sind die Ergebnisse der Messungen dargestellt. Gemessen wurden die Dateigröße, die Anzahl der in den Vorlagen enthaltenen Textzeilen, Referenzen (d. h. die Anzahl der Vorlagen-Zugriffe auf Modelldaten über den Velocity-Kontext), Fallunterscheidungen, Schleifen. Die Anzahl der Anweisungen, die zyklomatische Komplexität sowie der Halstead-Aufwand der Vorlagen wurden mit Hilfe des Analysewerkzeuges ermittelt. Die Anzahl der in den Vorlagen enthaltenen XMI-Elemente wurde auch berechnet.

Abbildung 6.1 Abbildung von VTL-Code auf Java-Code

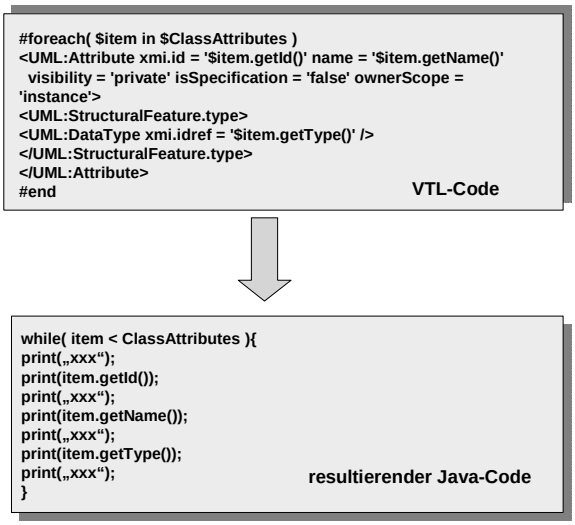


Abbildung 6.2 Ergebnisse der Analyse der VTL-Vorlagen

Vorlage	Dateigröße (in Bytes)	Textzeilen	Referenzen	Fallunter-scheidungen	Schleifen	Anweisungen	Zyklomatische Komplexität	Halstead-Aufwand	Anzahl der XMI-Elemente
A	8461	158	71	10	3	287	13	340.346,88	65
B	27467	421	280	16	8	753	24	1.633.699,31	213
C	87349	1014	972	57	0	2838	57	34.892.117,71	416
D	35955	512	469	26	10	1060	36	4.663.955,00	234

Die Tabelle in Abbildung 6.2 zeigt, dass die Größen der Vorlagen differieren und sowohl deren Komplexität als auch die Anzahl der Referenzen, Fallunterscheidungen, Anweisungen und XMI-Elemente mit der Größe ansteigen. Abbildung 6.3 zeigt die Verteilung der zyklomatischen Komplexität innerhalb der Vorlagen. Im Anhang A sind weitere Diagramme für die übrigen Metriken dargestellt. Aus dieser Abbildung ist zu erkennen, dass die Vorlage C die höchste Komplexität

besitzt. Im Vergleich zu den anderen Vorlagen hat diese Vorlage einen wesentlich höheren Halstead-Aufwand-Wert. Das liegt daran, dass die Vorlage C mehr Eingabedaten verarbeitet. In dieser Vorlage wird nämlich 972 mal (13,6 mal so hoch wie in der Vorlage A) auf die Modelldaten über den Velocity-Kontext zugegriffen. Das kann damit begründet werden, dass im Konfigurationsschritt C mehr Einstellungen als in den anderen Konfigurationsschritten möglich sind. In diesem Konfigurationsschritt kann der Benutzer die Beziehungen zwischen den Objekttypen festlegen und u. a. entscheiden, ob die Beziehung gleitende Beziehungsende enthält oder auch welche Operationen propagieren.
Zwischen Umfang der Vorlagen und deren Komplexität besteht eine annähernd lineare Abhängigkeit. Abbildung 6.4 zeigt den Zusammenhang zwischen der zyklomatischen Komplexität bzw. Halstead-Aufwand und der Zeilenanzahl der Vorlagen. Eine lineare Abhängigkeit ist auch zu erkennen, wenn anstatt der Komplexität die Anzahl der Referenzen, Fallunterscheidungen, Anweisungen oder XMI-Elemente auf der Y-Achse aufgetragen werden. Die resultierenden Diagramme befinden sich im Anhang A.

Abbildung 6.3 Zeilenanzahl und zyklomatische Komplexität der Vorlagen

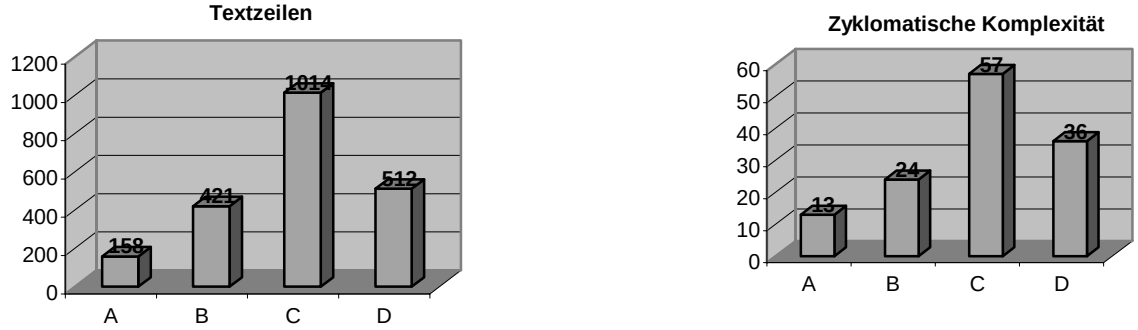
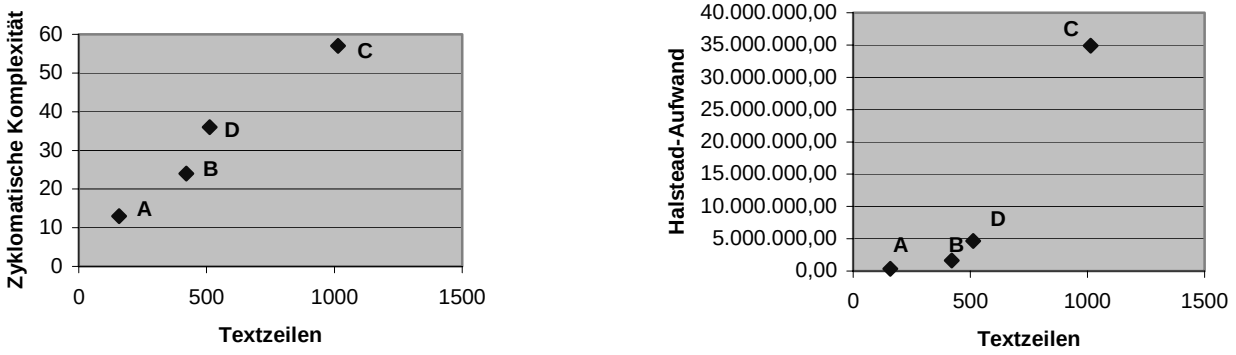


Abbildung 6.4 Korrelation zwischen Umfang und Komplexität der Vorlagen



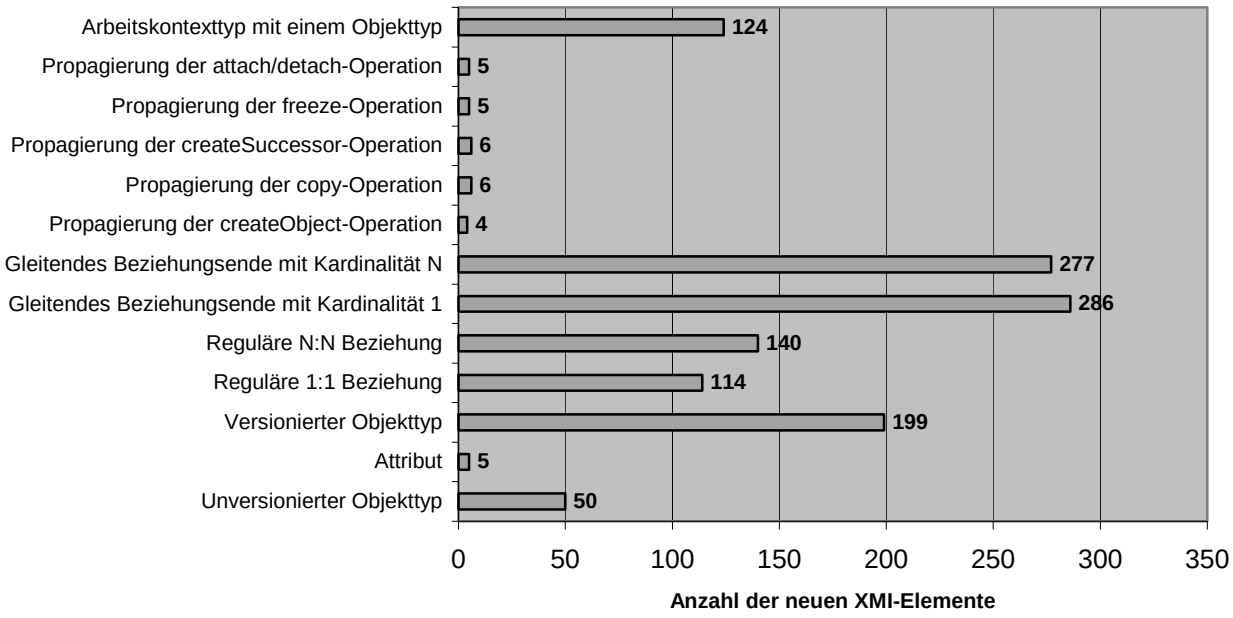
6.3 Analyse der Transformationsergebnisse

In diesem Abschnitt werden die Ergebnisse der automatisierten Transformationen analysiert. Dabei soll die Anzahl der verschiedenen generierten XMI-Elemente ermittelt werden und wird versucht, einen Zusammenhang zwischen dieser Anzahl und der Komplexität der Vorlagen zu finden. Mit Hilfe der am Anfang dieses Kapitels eingeführten Metrik nämlich *XEPR* soll die wichtige Frage beantwortet werden, welcher Zusammenhang bei der Spezifikation eines Software-Systems zwischen der Anzahl der generierten XMI-Elemente und dem Aufwand der Spezifikation dieses Systems besteht.

6.3.1 Beiträge einzelner Features

Im Informationsmodell eines Versionierungssystems befinden sich Objekttypen, die sowohl unversioniert als auch versioniert definiert werden können, Attribute, Operationen, Beziehungstypen und Arbeitskontexttypen. Die Beziehungsenden können gleitend definiert werden und die Operationen können propagieren. Diese Spezifikationselemente können die Generierung von XMI-Elementen unterschiedlich beeinflussen. In diesem Abschnitt soll geklärt werden, welche Anteile an der Generierung von XMI-Elementen auf die einzelnen Spezifikationselemente entfallen, d. h. wieviel ein Spezifikationselement zur Generierung von XMI-Elementen beiträgt. Dieser Beitrag wird ermittelt, indem entsprechende Elemente zu einer bestehenden Spezifikation hinzugefügt werden und die Differenz zwischen der Anzahl der generierten XMI-Elemente vor und nach der erweiterten Spezifikation gemessen wird. Abbildung 6.5 zeigt die graphische Darstellung der Ergebnisse dieser Messungen für verschiedenen Spezifikationselemente.

Abbildung 6.5 Beiträge einzelner Features



Die Abbildung 6.5 zeigt, dass die verschiedenen Spezifikationselemente die Generierung von XMI-Elementen unterschiedlich beeinflussen. Das Einfügen eines einzelnen Objekttyps ohne Attribute zum Informationsmodell bewirkt die Vergrößerung der Anzahl der XMI-Elemente mit 50 neuen Elementen. Auf jedes Attribut entfallen nur 5 zusätzliche Elemente. Das zeigt, dass die Aufnahme zusätzlicher Attribute keinen wesentlichen Einfluss auf die Anzahl der XMI-Elemente haben. Versionierte Objekttypen beanspruchen 3 mal so viel XMI-Elemente als unversionierte. Das liegt daran, dass bei versionierten Objekttypen mehrere Operationen zur Verwaltung der Versionen aufgenommen werden. Nach den Objekttypen werden die Anteile der Beziehungstypen aufgeführt, wobei zwischen regulären Beziehungen und Beziehungen mit gleitenden Enden unterschieden wird. Die Kardinalitäten der Beziehungsenden ist zudem noch ein Unterscheidungsfaktor. Für die Hinzufügung einer regulären 1:1 Beziehung wurde ein Zuwachs von 114 XMI-Elementen ermittelt. Einen größeren Einfluss haben die regulären N:N Beziehungstypen. Die Definition eines Beziehungsendes als gleitend beeinflusst wesentlich die Anzahl der generierten XMI-Elemente. Die Werte erhöhen sich um 277 zusätzliche Elemente bei einem gleitenden Beziehungsende mit Kardinalität N. Die Kardinalität 1 vergrößert diese Zahl noch mit 9 Elementen, weil beim Einfügen bzw. Löschen von Beziehungen, darauf geachtet werden muss, dass die Kardinalitätsbedingungen der Beziehungsenden nicht verletzt werden. Für die Hinzufügung eines Arbeitskontexttyps mit nur einem Objekttyp erhöht sich die Zahl mit 124 neuen Elementen. Diese Erhöhung wird hauptsächlich durch die Definition der Attach-/Detach-Methoden, der Methode *activate* und der Methoden, die die Objekttypen im Arbeitskontext zurückliefern, verursacht. Die Propagierung der Operationen verursachen hingegen nur eine minimale Vergrößerung.

6.3.2 Wirkung der Transformationen

Anhand des Informationssystems, das bereits aus dem letzten Kapitel bekannt ist, wird in diesem Abschnitt die Wirkung der durch Vorlagen automatisierten Transformationen auf die Anzahl der verschiedenen generierten XMI-Elemente untersucht. Der Zusammenhang zwischen der gesamten Anzahl der generierten XMI-Elemente in jedem Konfigurationsschritt und der Komplexität der Vorlagen wird hier auch diskutiert.

Die Anzahl aller generierten XMI-Elemente wurden für dieses Beispiel gemessen. Das schließt auch die XMI-Elemente ein, die für die verschiedenen OAL-Konstrukte definiert worden sind. Im Anhang B sind die Messergebnisse zu finden. Wir beschränken uns hier aber nur auf die bedeutendesten XMI-Elemente: *UML:Class*, *UML:Attribute*, *UML:Association* und *OAL:Method*. In Abbildung 6.6 sind die Anteile dieser Elemente dargestellt. Die in Abbildung 6.6 dargestellten Diagramme zeigen, welche dieser XMI-Elemente, mit welcher Anzahl und in welchem Konfigurationsschritt generiert werden. Im Konfigurationsschritt A werden keine *UML:Association*-Elemente generiert. Das ist auch logisch so, weil die Beziehungstypen erst im Konfigurationsschritt C definiert werden können. Im Konfigurationsschritt A werden hauptsächlich Objekttypen und Attribute definiert. Im Konfigurationsschritt B sind mehr *OAL:Method*-Elemente generiert. Das liegt daran, dass in diesem Konfigurationsschritt mehrere Operationen zur Verwaltung der Versionen aufgenommen werden (z. B. *freeze*, *createSuccessor, getPredecessor* etc.). *UML:Association*-Elemente werden auch generiert, da die reflexiven Beziehungen, die dem Zugriff einer Objektversion auf ihren Nachfolger- bzw. Vorgänger-Versionen dienen, in diesem Konfigurationsschritt definiert werden. Im Konfigurationsschritt C werden hauptsächlich *OAL:Method*- und *UML:Association*-Elemente generiert. In diesem Konfigurationsschritt werden neben den Beziehungstypen Methoden definiert, die entweder der Navigation entlang der

Beziehungen oder der Manipulation dieser Beziehungen dienen. Im Konfigurationsschritt D werden mehr *OAL:Method*-Elemente generiert. In diesem Konfigurationsschritt werden nämlich verschiedene Methoden definiert wie beispielsweise die Attach-/Detach-Methoden. In Abbildung 6.7 sind die Anzahl der generierten XMI-Elemente und die Anzahl der in den Vorlagen enthaltenen XMI-Elemente dargestellt. Der größte Teil der XMI-Elemente wird im Konfigurationsschritt C generiert. Der Konfigurationsschritt A trägt am wenigsten zur Generierung von XMI-Elementen bei. Schon mit einem einfachen Informationsmodell, das nur drei Klassen und drei Beziehungen enthält, ist der Faktor, der sich bei einem Vergleich der Anzahl der generierten XMI-Elemente mit der Anzahl der in den Vorlagen enthaltenen XMI-Elemente ergibt, mehr als 3 in den ersten drei Konfigurationsschritten und fast 2 im Konfigurationsschritt D. Das zeigt, dass sich die Entwicklung der Vorlagen bezogen auf die manuelle Spezifikation eines Software-Systems auf jeden Fall rentiert.

Abbildung 6.6 Anteil der wichtigsten XMI-Elemente

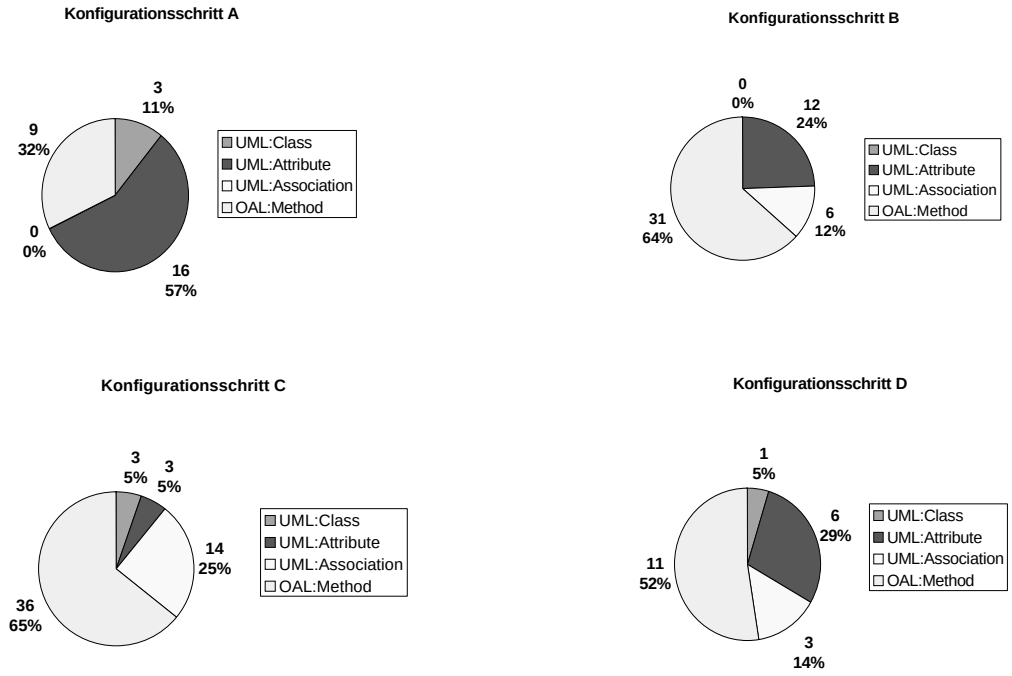
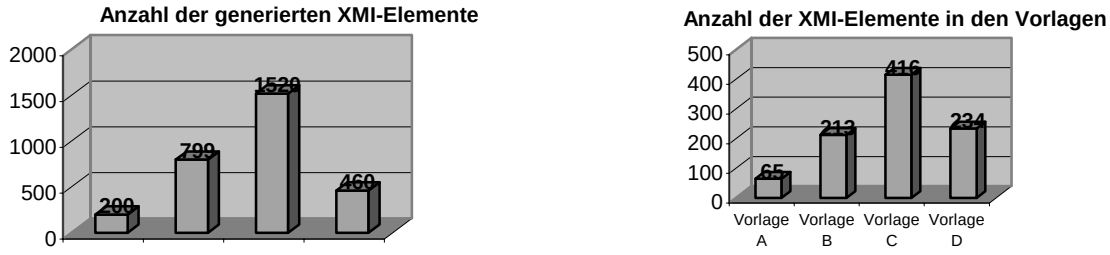
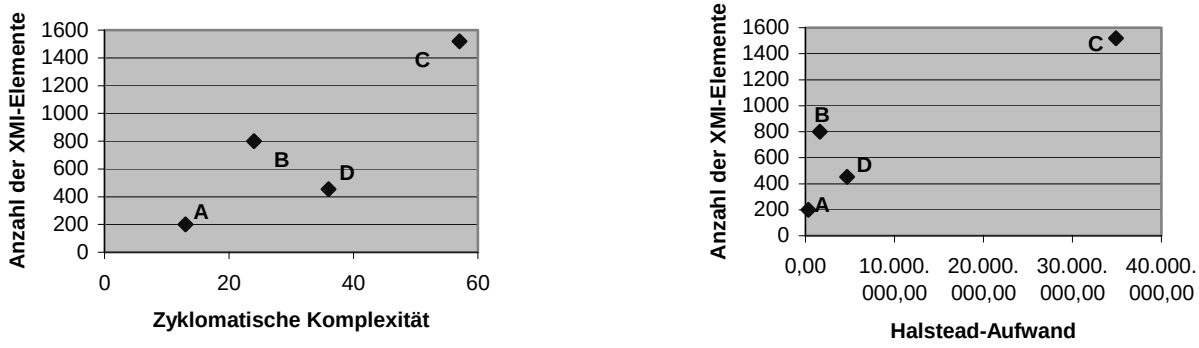


Abbildung 6.7 Anzahl der XMI-Elemente



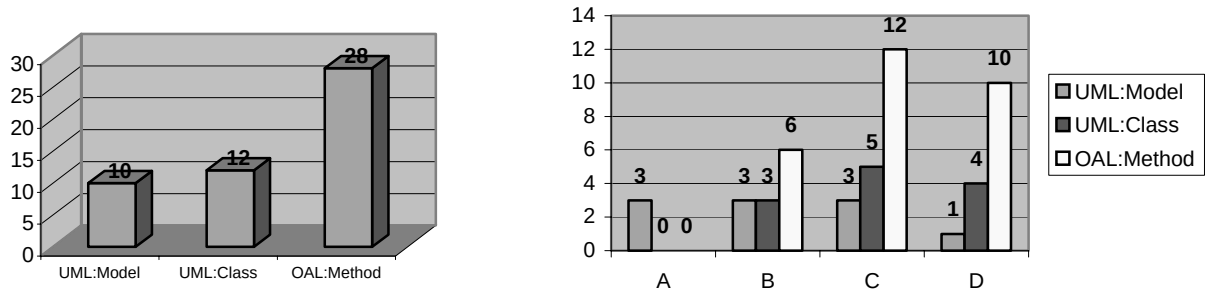
Nun wird der Zusammenhang zwischen der gesamten Anzahl der generierten XMI-Elemente in jedem Konfigurationsschritt und dem Aufwand der Vorlagen näher betrachtet. Zwischen der Anzahl der generierten XMI-Elemente und dem Aufwand der Vorlagen ist keine lineare Abhängigkeit erkennbar. Dieses wird in den Diagrammen aus Abbildung 6.8 veranschaulicht. In diesen Diagrammen ist auf der X-Achse die Zyklomatische Komplexität bzw. Halstead-Aufwand aufgetragen und auf der Y-Achse die Anzahl der generierten XMI-Elemente. Die Vorlage D ist zwar mit dem Faktor 1,5 komplexer als die Vorlage B, generiert allerdings weniger XMI-Elemente. Die größere zyklomatische Komplexität und der höhere Halstead-Aufwand der Vorlage D lassen sich dadurch begründen, dass einerseits diese Vorlage mehr Kontrollflüsse enthält und andererseits mehr Eingabedaten verarbeitet. Sowohl die Objekte im Arbeitskontext und die zwischen ihnen bestehenden Beziehungen als auch die Propagierung der Attach-/Detach-Operationen werden nämlich durch diese Vorlage verarbeitet. Der Grund, weshalb die Vorlage B mehr XMI-Elemente als die Vorlage D generiert, besteht darin, dass in einem Informationssystem meistens mehr versionierte Objekttypen als Arbeitskontexttypen definieret werden. Zudem trägt ein versionierter Objekttyp zur Generierung von XMI-Elementen mehr als ein Arbeitskontexttyp bei (Faktor 1,6).

Abbildung 6.8 Korrelationen zum Aufwand der Vorlagen



Die konkrete Transformation erweitert das Modell mit neuen Klassen und Beziehungen, die Klassen mit neuen Attributen und Methoden und die Methoden mit neuen Methoden-Konstrukten. Dabei werden die entsprechenden XMI-Elemente nämlich *UML:Model*, *UML:Class* und *OAL:Method* referenziert. Abbildung 6.9 zeigt wann und wie oft diese XMI-Elemente geändert werden. Die *OAL:Method*-Elemente werden am häufigsten geändert. Die Methoden *createObject* und *copy* werden zunächst im Konfigurationsschritt B mit neuen Anweisungen erweitert. Die Methoden *createObject*, *copy*, *createSuccessor* und *freeze* bzw. *attach* und *detach* werden im Konfigurationsschritt C bzw. D geändert und mit neuen Konstrukten erweitert, falls sie propagieren sollen. Die im Konfigurationsschritt A definierten Klassen können dann im Konfigurationsschritt B mit neuen Attributen wie *verId* und Methoden, die der Verwaltung der Versionshierarchie dienen wie *createSuccessor*, erweitert. Im Konfigurationsschritt C kommen noch andere Methoden dazu, die hauptsächlich der Navigation entlang der Beziehungen dienen. Die Methode *attach* und *detach* werden im Konfigurationsschritt D den Klassen hinzugefügt. Das Modell wird in den Konfigurationsschritt A und D mit Klassen und in B und C mit Beziehungen erweitert.

Abbildung 6.9 Anzahl der durch die Transformation geänderten XMI-Elemente



6.3.3 Vergleich veschiedener Informationsmodelle

In diesem Abschnitt soll hauptsächlich untersucht werden, welcher Zusammenhang bei der Spezifikation eines Software-Systems zwischen der Anzahl der generierten XMI-Elemente und dem Aufwand der Spezifikation dieses Systems besteht. Dafür wurden drei Modelle als Beispiele von Informationssystemen genommen: Das UML Core Package, das CWM Relational Package und das MOF Core Package. Die Anzahl der generierten XMI-Elemente wurden auch für jedes dieser Informationssysteme gemessen. Die Messergebnisse können dem Anhang B entnommen werden. Die Bemerkungen und Erklärungen aus dem letzten Abschnitt bezüglich einerseits der Verteilung der Anzahl der generierten XMI-Elemente und andererseits dem Zusammenhang zwischen der gesamten Anzahl der generierten XMI-Elemente in jedem Konfigurationsschritt und dem Aufwand der Vorlagen gelten auch bei diesen Beispielen. Das Diagramm in Abbildung 6.9 zeigt für jedes Informationsmodell die Anzahl der generierten XMI-Elemente. Die Schwankungen innerhalb dieses Diagramms lassen sich mit der unterschiedlichen Anzahl der Attributen der Objekttypen und der definierten Beziehungstypen erklären. Dieses Diagramm ist daher nicht sehr aussagekräftig. Die Anzahl der Benutzer-Angaben werden in den nächsten zwei Abbildungen 6.10 und 6.11 mitbetrachtet. So kann der Zusammenhang zwischen der Anzahl der generierten

XMI-Elemente und dem Aufwand der Spezifikation eines Systems ermittelt werden. In Abbildungen 6.10 werden die *XEPR*-Werte für verschiedene Informationsmodelle dargestellt. Das Diagramm zeigt, dass diese *XEPR*-Werte eine ähnliche Tendenz folgen. Werden die Anzahl der Benutzer-Angaben auf die X-Achse aufgetragen und auf der Y-Achse die der generierten XMI-Elemente, ist eine lineare Abhängigkeit erkennbar. Das resultierende Diagramm ist in Abbildung 6.11 aufgeführt. Mit diesem Diagramm kann festgestellt werden, dass zwischen der Anzahl der generierten XMI-Elemente und dem Aufwand der Spezifikation eines Software-Systems eine sehr enge Beziehung besteht.

Abbildung 6.10 Anzahl der generierten XMI-Elemente

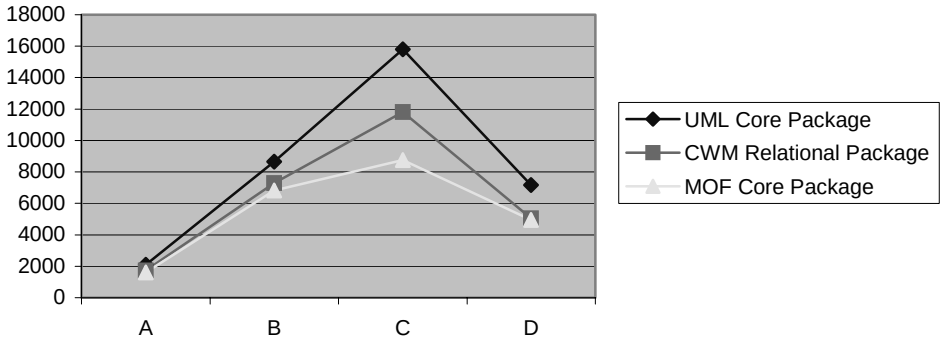


Abbildung 6.11 Anzahl der XMI-Elemente pro Benutzer-Angabe (*XEPR*)

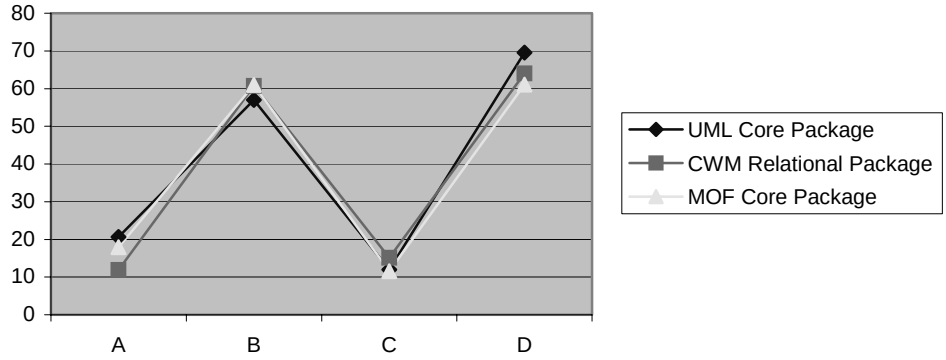
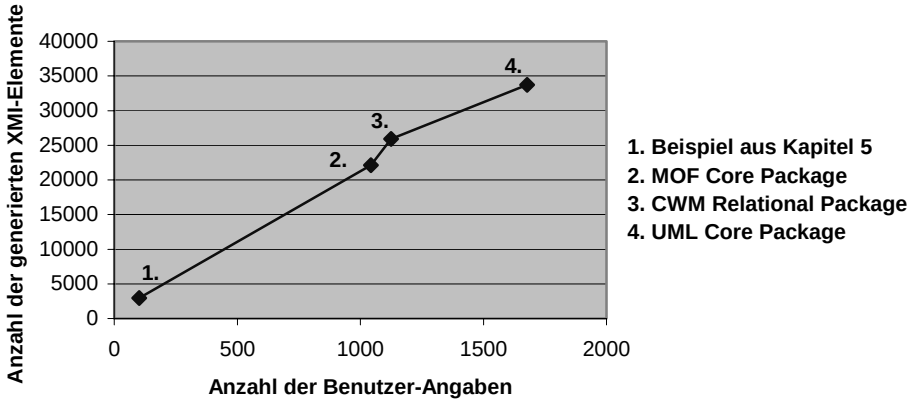


Abbildung 6.12 Korrelation zum Spezifikationsaufwand



Kapitel 7 Zusammenfassung und Ausblick

In diesem Kapitel werden die wichtigsten Erkenntnisse dieser Arbeit abschließend zusammengefasst und ein Ausblick auf weitere mögliche Aspekte und Forschungsfelder gegeben.

7.1 Zusammenfassung

In dieser Diplomarbeit wird ein Werkzeug vorgestellt, dass in enger Beziehung zu dem Produktlinienansatz bzw. der generativen Programmierung und dem verwandten Ansatz der MDA steht. Im Bezug auf der generativen Programmierung bietet *MTFLOW* eine Umgebung, in der alle Schritte, die der automatisierten Spezifikation eines Systems in einer Software-Produktlinie dienen, vorgenommen werden können. Die Spezifikation erfolgt dabei nicht mehr direkt über eine domänenspezifische Sprache, mit deren Komplexität der Benutzer überfordert werden kann, sondern mit Hilfe eines Konfigurators, der dem Benutzer ein einfaches Mittel zur Verfügung stellen soll, seine Anforderung leicht zu formulieren und sein erwünschtes System bis ins Detail schrittweise und korrekt zu spezifizieren. Anhand dieser Spezifikation erstellt ein Generator das gewünschte Produkt. In *MTFLOW* erfolgt die Spezifikation eines Software-Systems mit Hilfe von UML und OAL, so wird die Mühe erspart, einen Generator zu finden, der aus einer gegebenen DSL Code in einer bestimmten Implementierungssprache erzeugen soll. In unserem Fall braucht der Generator keine semantische Analyse der Spezifikation durchzuführen. Er braucht auch nicht die DSL des Systems zu verstehen. Seine Aufgabe wird so stark vereinfacht, er hat nämlich nur die Aufgabe eines Übersetzers, der aus UML und OAL den Implementierungscode erzeugen soll.

MTFLOW steht auch in Beziehung zu MDA. Der Ansatz der MDA besteht darin, die Funktionalität eines Software-Systems zunächst unter Abstraktion von den technischen Details der Realisierung in Modellform zu entwerfen. Nach Verfeinerungen auf dieser Ebene der plattformunabhängigen Modelle (PIM) kann die Funktionalität auf ein plattformspezifisches Modell (PSM) umgesetzt werden, das Bezug auf technische Details der Zielplattform nimmt. In MDA spielen die Modelltransformationen eine wesentliche Rolle. *MTFLOW* unterstützt die PIM-zu-PIM Transformation und verwendet dieser Art der Modelltransformationen bei der Spezifikation von Software-Systemen. Die Spezifikation erfolgt dabei in mehreren Konfigurationsschritten mit Hilfe aufeinanderfolgender Modelltransformationen, in welchen XMI und VTL als Techniken eingesetzt werden. XMI enthält Mechanismen, die Differenzen zwischen zwei Modellen beschreiben können. In jedem Konfigurationsschritt entsteht mit Hilfe einer VTL-Vorlage ein

XMI-Differenz-Dokument, das die in diesem Konfigurationsschritt definierten Anwendungsdaten durch entsprechende XMI-Elemente beschreibt. Dieses XMI-Differenz-Dokument wird dann mit dem alten XMI-Modell-Dokument zusammengeführt, das aus dem vorherigen Konfigurationsschritt entstanden ist. Das nach einem Konfigurationsschritt erzeugte XMI-Model-Dokument enthält eine XMI-Darstellung des Modells, das bis zu diesem Konfigurationsschritt das System spezifiziert. Dieses XMI-Modell-Dokument wird im nächsten Konfigurationsschritt nach dem gleichen Modelltransformations-Prinzip erweitert. Mit diesem iterativen Verfahren bekommt der Benutzer nach dem letzten Konfigurationsschritt ein in XMI beschriebenes Modell seines erwünschten Systems. Diese Transformationen erfolgen aber nicht willkürlich, sondern werden durch ein Workflow-Modell kontrolliert, das den Ablauf der Transformationen vorschreibt. MTFLOW ist ein generischer Ansatz und beschränkt sich nicht nur auf eine Software-Produktlinie. Dieser Ansatz spielt nämlich die Rolle einer Workflow-Engine, die ein Workflow-Modell instantiiert und die Prozess-Instanzen ausführt. So können mehrere Software-Produktlinien unterstützt werden, indem einfach andere Workflow-Modell definiert werden sollen.

Die Beurteilung dieses XMI-basierten Ansatzes, der die automatisierten Modelltransformationen durch VTL-Vorlagen unterstützt, ergab, dass sich der in die Entwicklung der Vorlagen investierte Aufwand bezogen auf den der manuellen Spezifikation eines Software-Systems auszahlen würde. Aus dem Vergleich verschiedener Informationsmodelle konnte festgestellt werden, dass zwischen der Anzahl der generierten XMI-Elemente und dem Aufwand der Spezifikation eines Software-Systems eine lineare Abhängigkeit besteht.

7.2 Ausblick

Während der Erstellung dieser Diplomarbeit sind Fragestellungen entstanden, die sich als weiterführende Forschungsthemen anbieten. Für die Zukunft sind einige Erweiterungen an *MTFLOW* vorstellbar, die teilweise neue Forschungsfelder eröffnen.

Es existiert keine Entwicklungsumgebung, die die Entwicklung der VTL-Vorlagen unterstützen soll. Zur Erstellung einer Vorlage muss zur Zeit ein herkömmlicher Texteditor verwendet werden, der keine spezielle Unterstützung bietet. Es wäre sehr hilfreich ein Werkzeug zu haben, das einerseits den Syntax der VTL-Programme interpretieren und andererseits beim Auffinden von Fehlern in den Vorlagen durch einen internen Debugger helfen kann.

Für die Spezifikation eines Systems in einer Software-Produktlinie, werden in *MTFLOW* ein domänenspezifisches Workflow-Modell und eine Menge von domänenspezifischen Vorlagen, die für die konkrete Transformation zuständig sind, benötigt. Das Workflow-Modell und die Vorlagen sollen die domänenspezifische Konfigurationssprache ersetzen. Viele Software-Hersteller verkaufen Produkte, die eigentlich zu einer Produktlinie gehören können, da sie aus einer gemeinsamen Menge von Komponenten bestehen und auf einer gemeinsamen Software-Architektur basieren. Aus einer Analyse der bereits vorhandenen Systeme können ein Workflow-Modell und *MTFLOW*-Vorlagen erzeugt werden, auf Basis deren *MTFLOW* für die Entwicklung weiterer Systeme aus der Produktlinie verwendet werden kann.

Zur Zeit besteht in *MTFLOW* keine direkte Verbindung zur Domänen-Analyse. Die Phase der Domänen-Analyse ist in der generativen Programmierung wichtig. In dieser Phase werden hauptsächlich Merkmalmodelle definiert. In einem Merkmalmodell werden die Merkmale, d. h. die strukturellen und funktionalen Eigenschaften der untersuchten Systeme kategorisiert und dabei

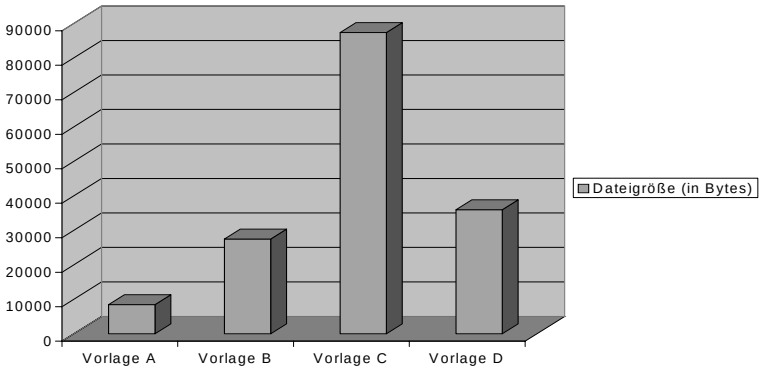
insbesondere zwischen festen und variablen Merkmalen unterschieden. Außerdem werden die möglichen Erweiterungspunkte der Produktlinie identifiziert. Bei Merkmalmodellen werden u. A. die Gemeinsamkeiten und Unterschiede in den Daten- und Kontrollflüssen der Systeme dokumentiert. Teil der Merkmalmodelle sind Merkmaldiagramme. Dabei werden die Merkmale des Softwaresystems hierarchisch dargestellt. Sie dokumentieren, welche Merkmale feste bzw. optionale Bestandteile des Systems sind und welche Merkmale Alternativen bilden. FODA [KCH+90] ist eine Notation für Merkmaldiagramme. In [CE00] wurde auch eine Notation definiert, die auf FODA basiert. Diese Notationen können als Basis für das Workflow-Modell und die Vorlagen benutzt werden. Die Funktionalität von *MTFLOW* kann erweitert werden, indem die Merkmaldiagramme analysiert und daraus ein Workflow-Modell und die nötigen Transformations-Vorlagen definiert werden.

Bei der Analyse der Ergebnisse der automatisierten Transformationen werden Metriken benutzt, die den Zusammenhang einerseits zwischen der Komplexität der Vorlagen und der Anzahl der generierten XMI-Elemente und andererseits zwischen dieser Anzahl und dem Aufwand der Spezifikation eines Software-Systems erklären sollen. Es könnte auch von Bedeutung sein, den Aufwand der Spezifikation eines Systems mit Hilfe von *MTFLOW* mit dem der manuellen Modellierung dieses Systems zu vergleichen. Die eingeführte Metrik nämlich *XEPR* betrachtet nur die Anzahl der generierten XMI-Elemente. Es hat sich festgestellt, dass die verschiedenen Spezifikationselemente (Objekttyp, Attribute, Beziehungstyp...) die Generierung von XMI-Elementen unterschiedlich beeinflussen und unterschiedliche XMI-Elemente erzeugen. Daher wäre auch sinnvoll andere Metriken einzuführen, die die Gewichte der verschiedenen XMI-Elemente miteinbeziehen. Die empirische Bestimmung der Gewichte sollte auf einer Analyse verschiedener Produktlinie basieren.

Anhang A Ergebnisse der Vorlagenbewertung

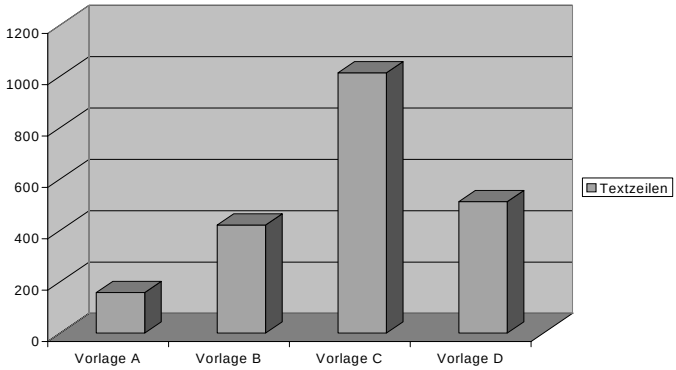
A.1 Grafische Darstellung der Messergebnisse

Abbildung 1 Dateigröße der Vorlagen (in Bytes)



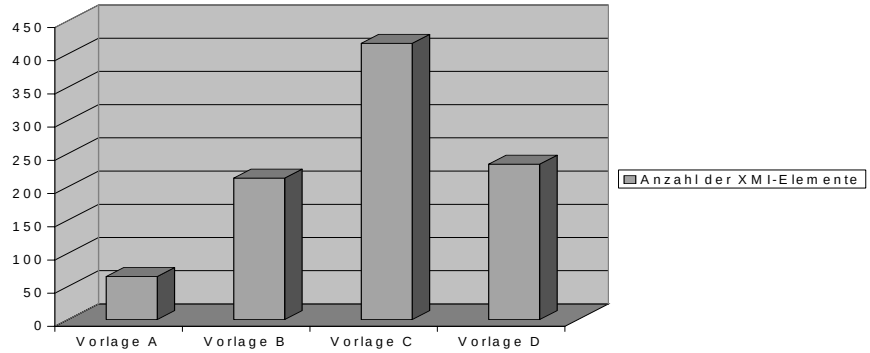
❑

Abbildung 2 Zeilenanzahl der Vorlagen



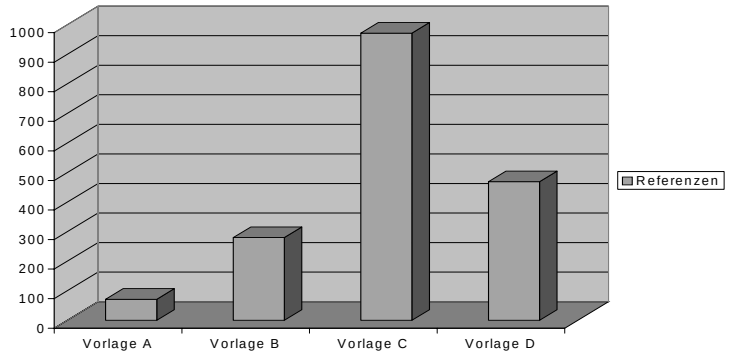
❑

Abbildung 3 Anzahl der XMI-Elemente in den Vorlagen



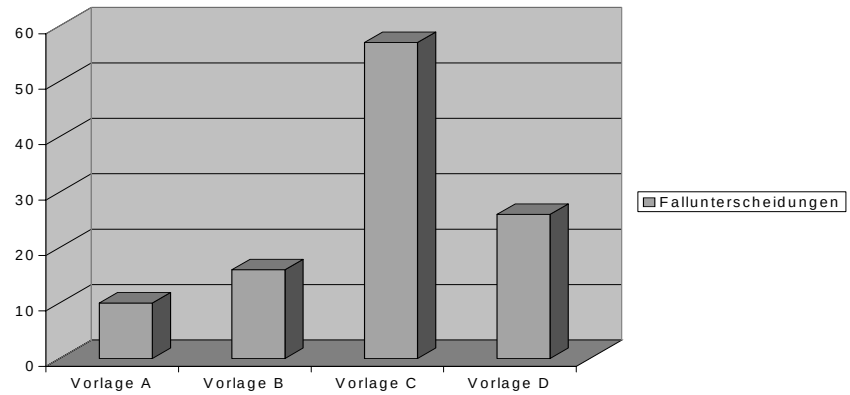
❑

Abbildung 4 Anzahl der Referenzen in den Vorlagen



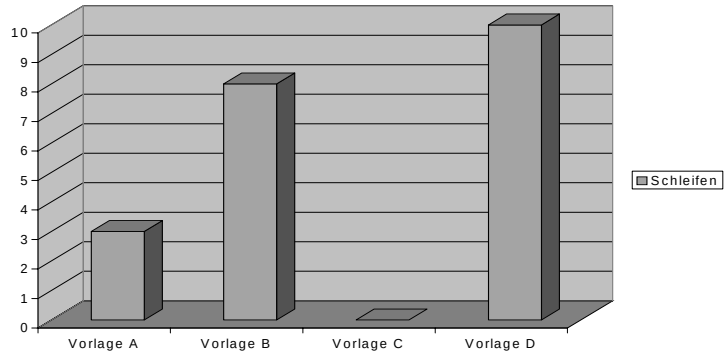
❑

Abbildung 5 Anzahl der Fallunterscheidungen in den Vorlagen



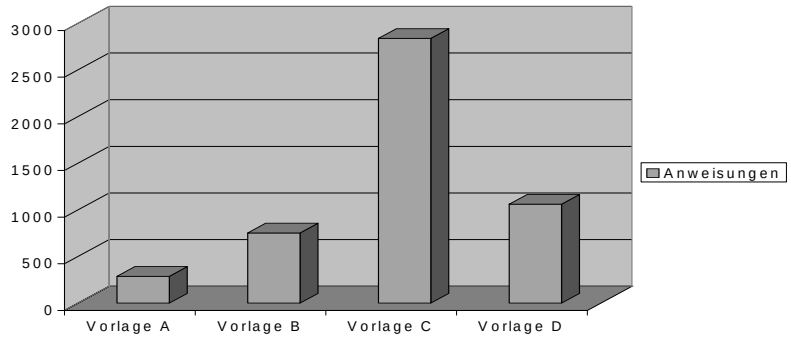
❑

Abbildung 6 Anzahl der Schleifen in den Vorlagen



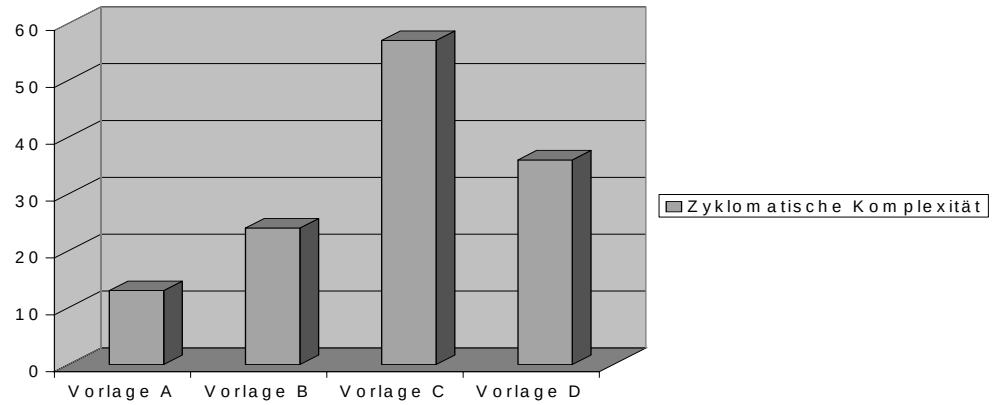
❑

Abbildung 7 Anzahl der Anweisungen in den Vorlagen



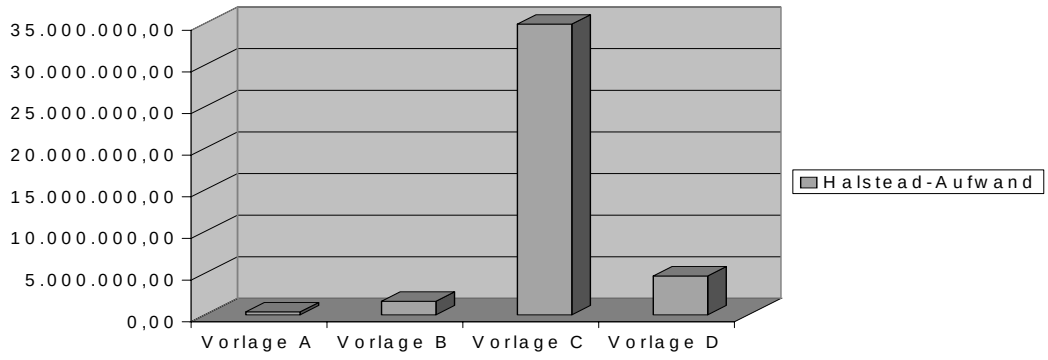
❑

Abbildung 8 Zyklomatische Komplexität der Vorlagen



❑

Abbildung 9 Halstead-Aufwand der Vorlagen

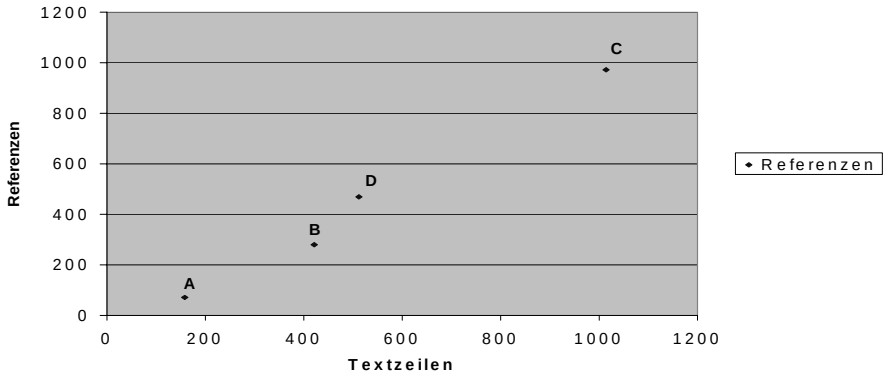


❑

A.2 Korrelationen

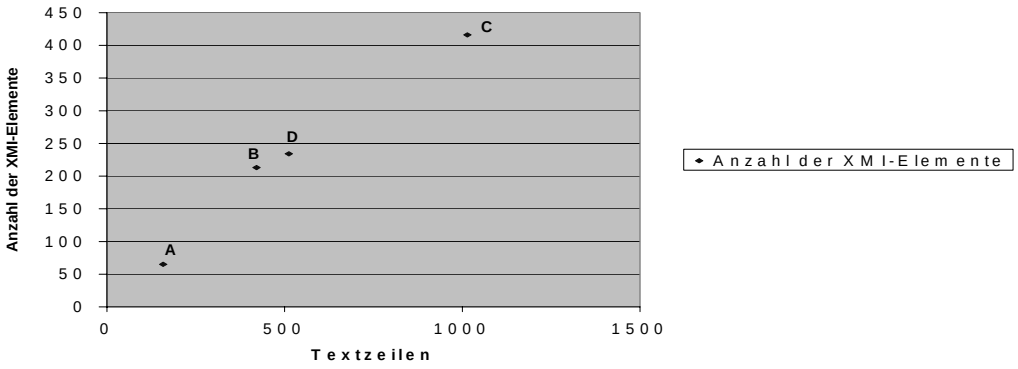
A.2.1 Korrelationen zum Umfang der Vorlagen

Abbildung 10 Referenzen



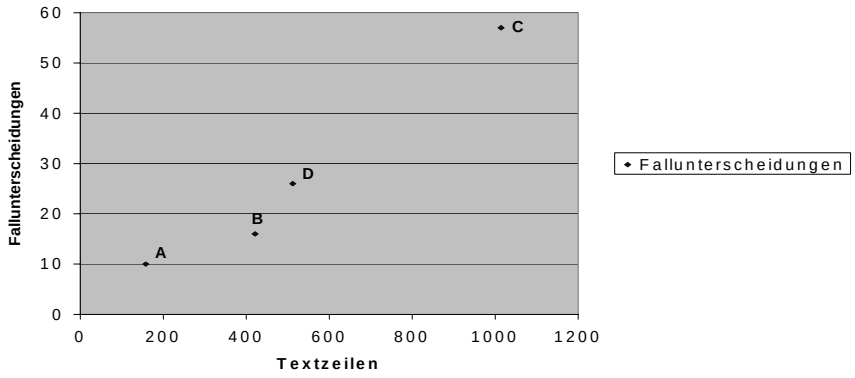
❑

Abbildung 11 Anzahl der XMI-Elemente



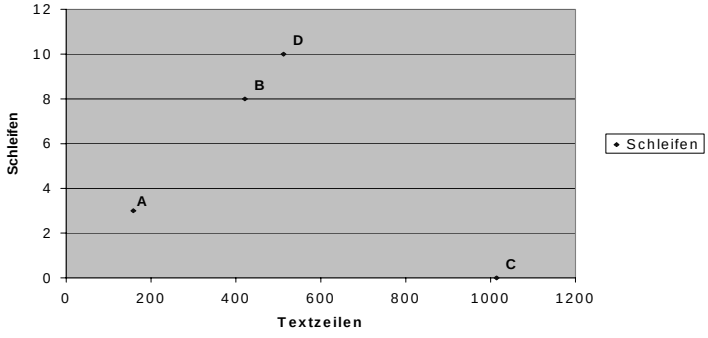
❑

Abbildung 12 Fallunterscheidungen



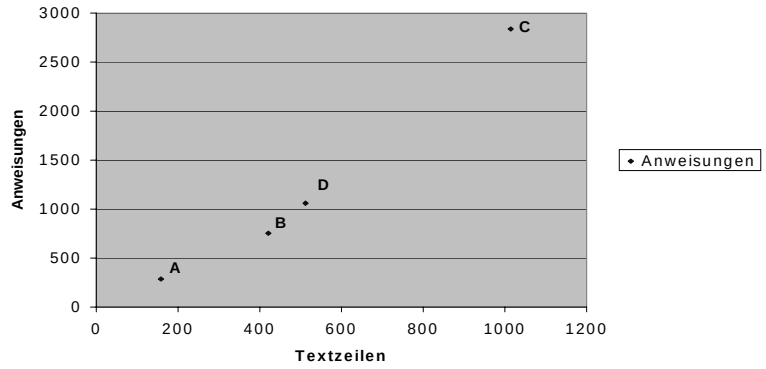
❑

Abbildung 13 Schleifen



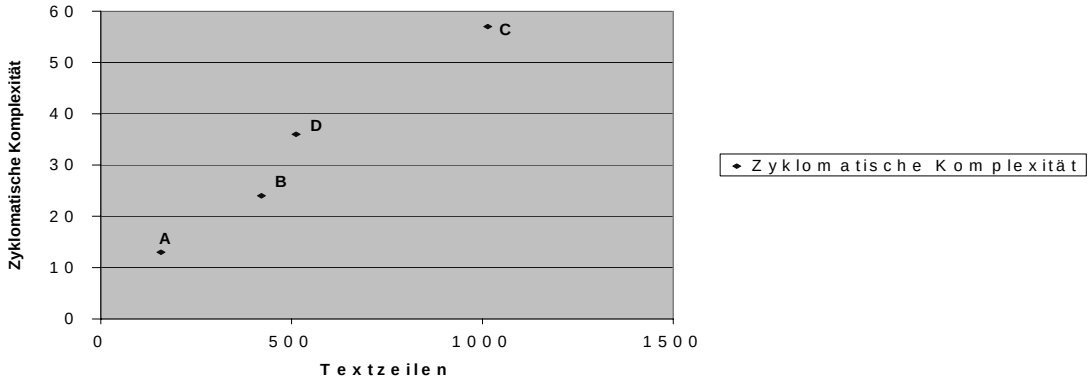
❑

Abbildung 14 Anweisungsanzahl



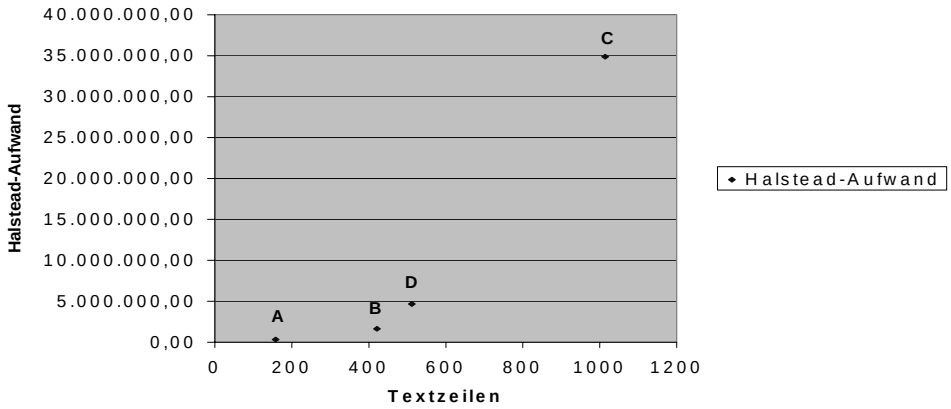
❑

Abbildung 15 Zyklomatische Komplexität



❑

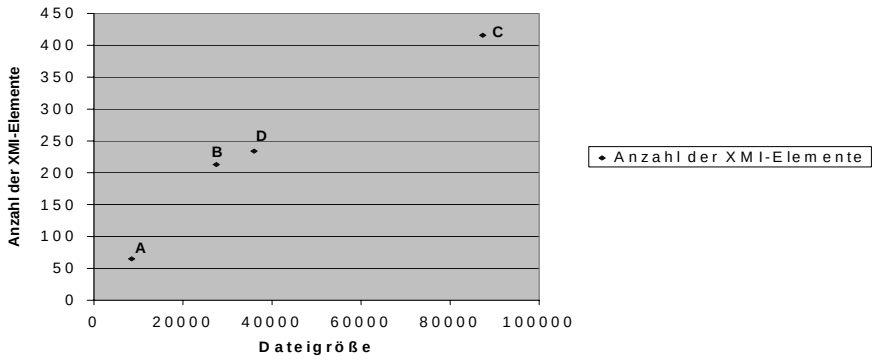
Abbildung 16 Halstead-Aufwand



❑

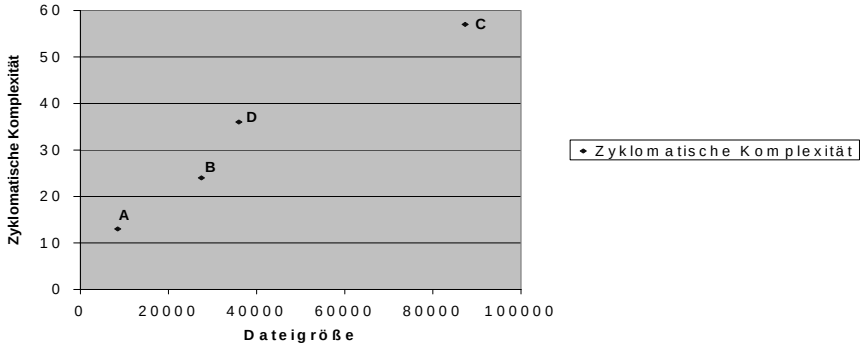
A.2.2 korrelationen zur Dateigröße

Abbildung 17 Anzahl der XMI-Elemente



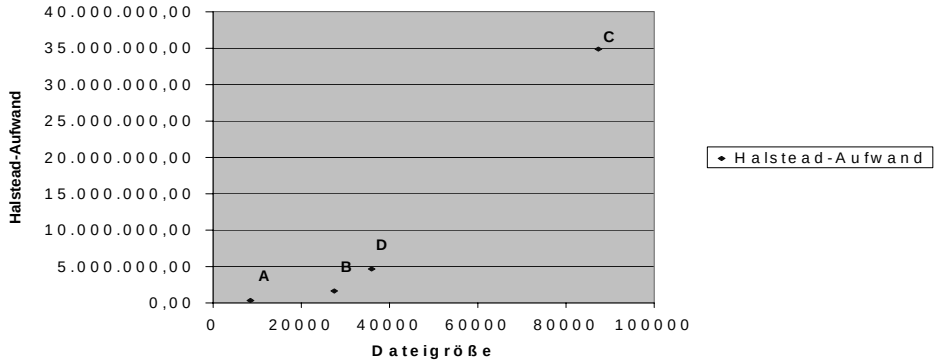
❑

Abbildung 18 Zyklomatische Komplexität



❑

Abbildung 19 Halstead-Aufwand



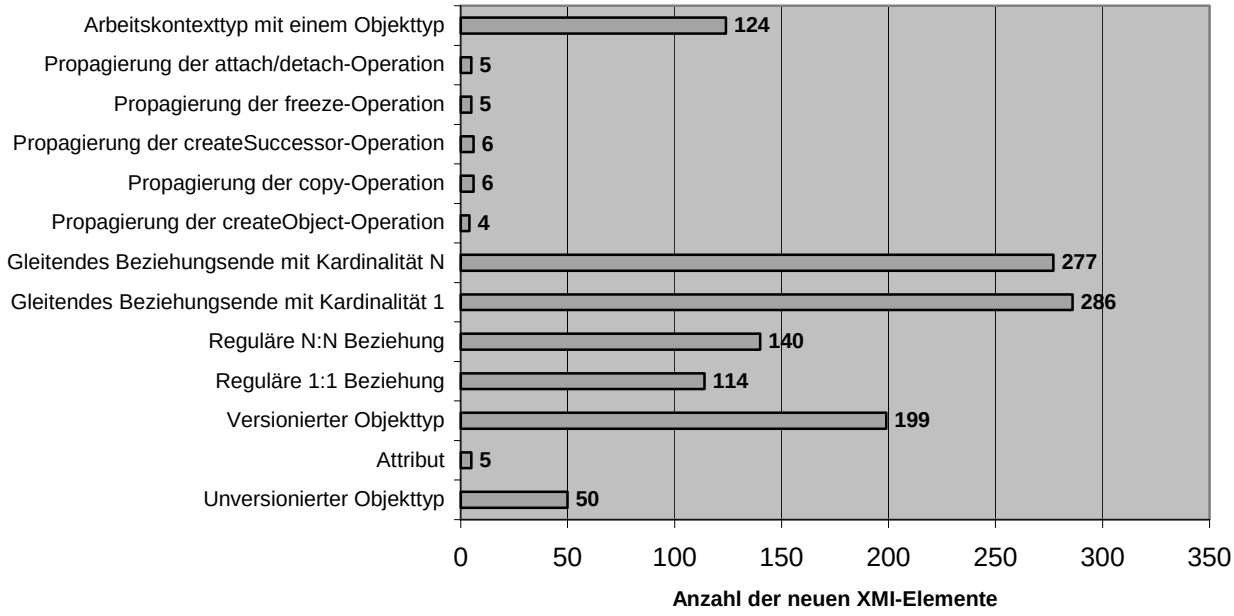
❑

Anhang B

Ergebnisse der Transformationen

B.1 Beiträge einzelner Features

Abbildung 1 Beiträge einzelner Features



❑

B.2 Graphische Darstellung der Messergebnisse

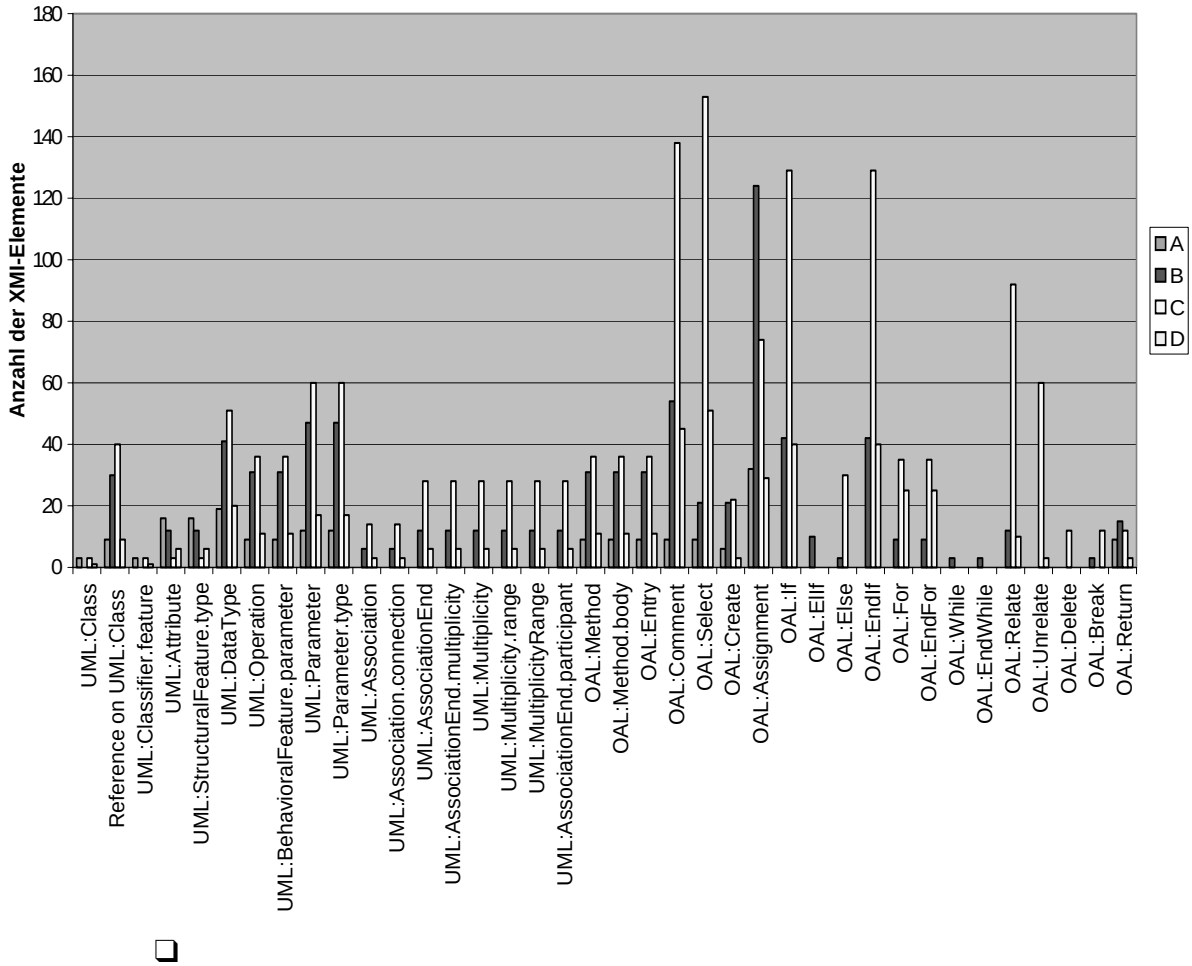
B.2.1 Informationsmodell aus Kapitel 5

Abbildung 2 Anzahl der generierten XMI-Elemente

XMI-Element	A	B	C	D
UML:Class	3		3	1
Reference on UML:Class	9	30	40	9
UML:Classifier.feature	3		3	1
UML:Attribute	16	12	3	6
UML:StructuralFeature.type	16	12	3	6
UML:DataType	19	41	51	20
UML:Operation	9	31	36	11
UML:BehavioralFeature.parameter	9	31	36	11
UML:Parameter	12	47	60	17
UML:Parameter.type	12	47	60	17
UML:Association		6	14	3
UML:Association.connection		6	14	3
UML:AssociationEnd		12	28	6
UML:AssociationEnd.multiplicity		12	28	6
UML:Multiplicity		12	28	6
UML:Multiplicity.range		12	28	6
UML:MultiplicityRange		12	28	6
UML:AssociationEnd.participant		12	28	6
OAL:Method	9	31	36	11
OAL:Method.body	9	31	36	11
OAL:Entry	9	31	36	11
OAL:Comment	9	54	138	45
OAL:Select	9	21	153	51
OAL:Create	6	21	22	3
OAL:Assignment	32	124	74	29
OAL:If		42	129	40
OAL:ElIf		10		
OAL:Else		3	30	
OAL:EndIf		42	129	40
OAL:For		9	35	25
OAL:EndFor		9	35	25
OAL:While		3		
OAL:EndWhile		3		
OAL:Relate		12	92	10
OAL:Unrelate			60	3
OAL:Delete			12	
OAL:Break		3		12
OAL:Return	9	15	12	3
	200	799	1520	460

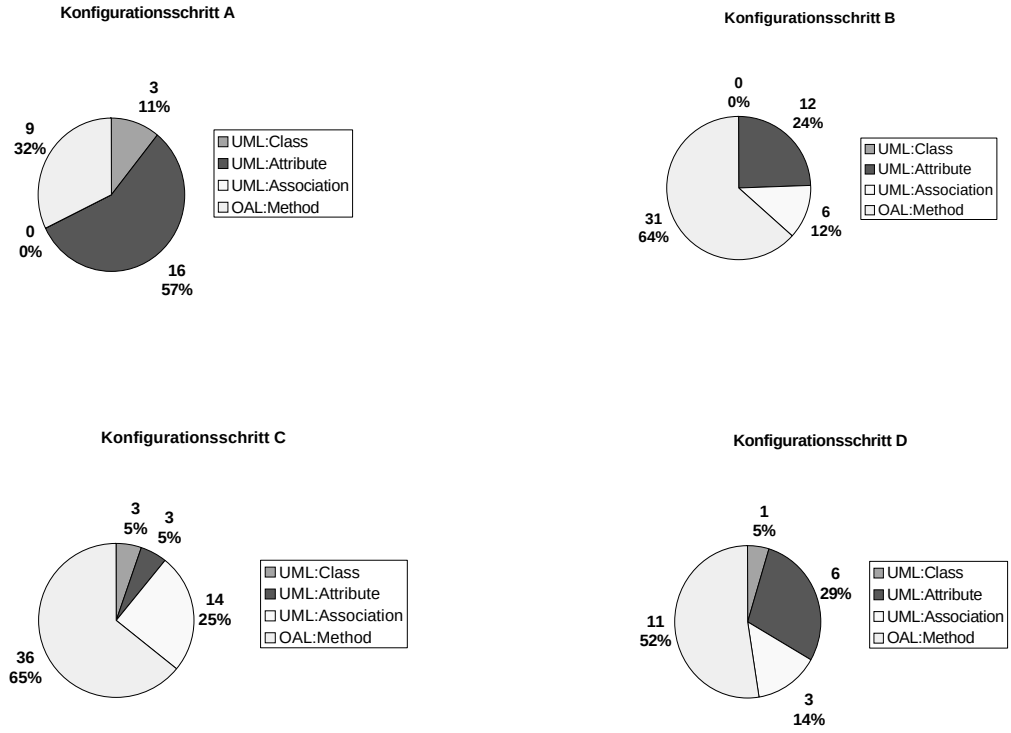
❑

Abbildung 3 Graphische Darstellung der Messergebnisse aus Abbildung 2



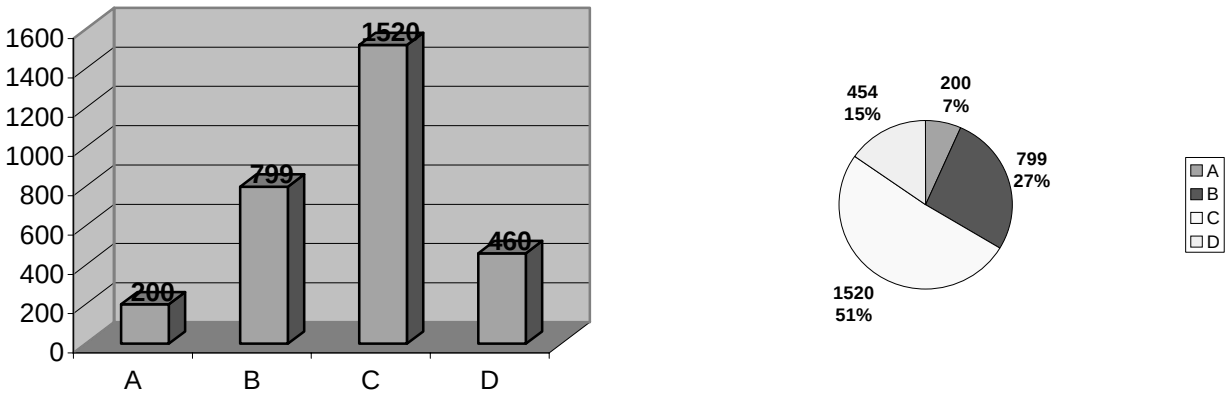
❑

Abbildung 4 Anteil der wichtigsten XMI-Elemente



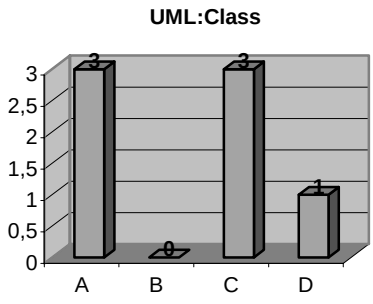
❑

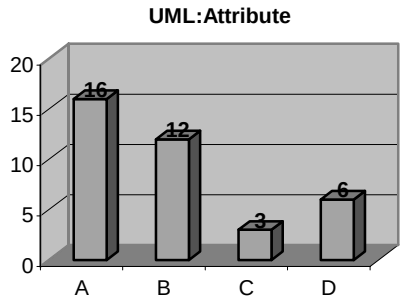
Abbildung 5 Anzahl aller XMI-Elemente



❑

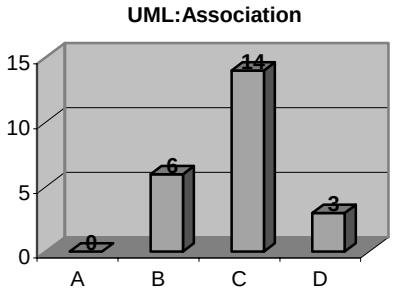
Abbildung 6 Anzahl der <UML:Class>- und <UML:Attribute>-Elemente

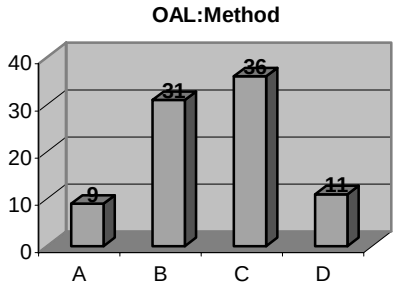




❑

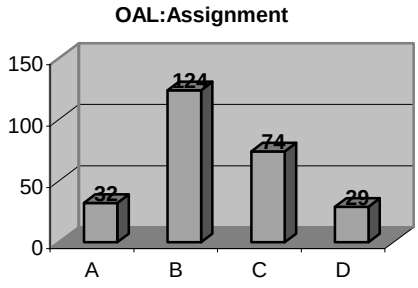
Abbildung 7 Anzahl der <UML:Association>- und <OAL:Method>-Elemente

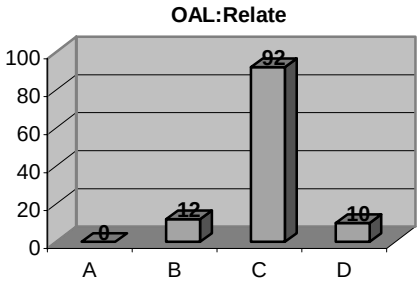




❑

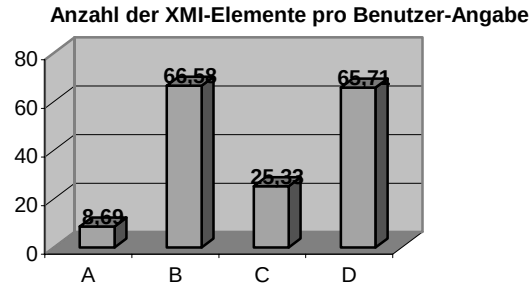
Abbildung 8 Anzahl der <OAL:Assignment>- und <OAL:Relate>-Elemente

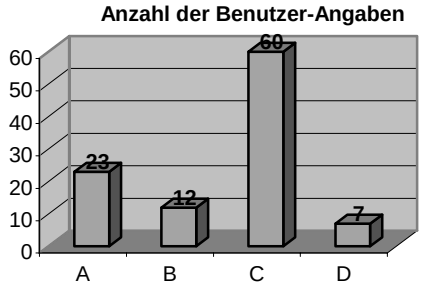




❑

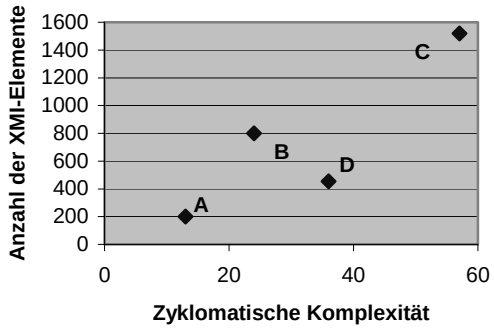
Abbildung 9 Anzahl der XMI-Elemente pro Benutzer-Angabe

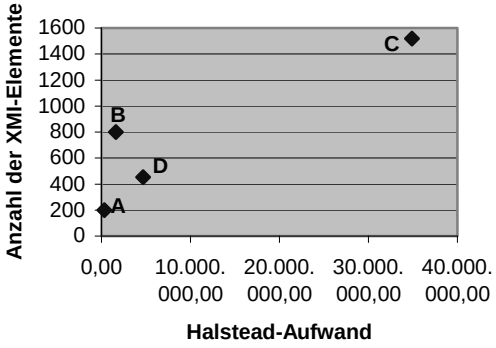




❑

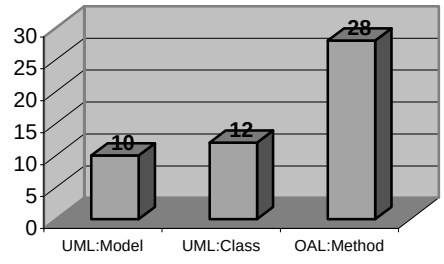
Abbildung 10 Korrelationen zum Aufwand der Vorlagen

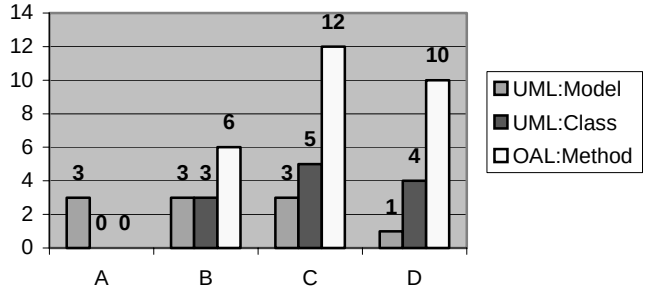




❑

Abbildung 11 Anzahl der durch die Transformation geänderten XMI-Elemente





❑

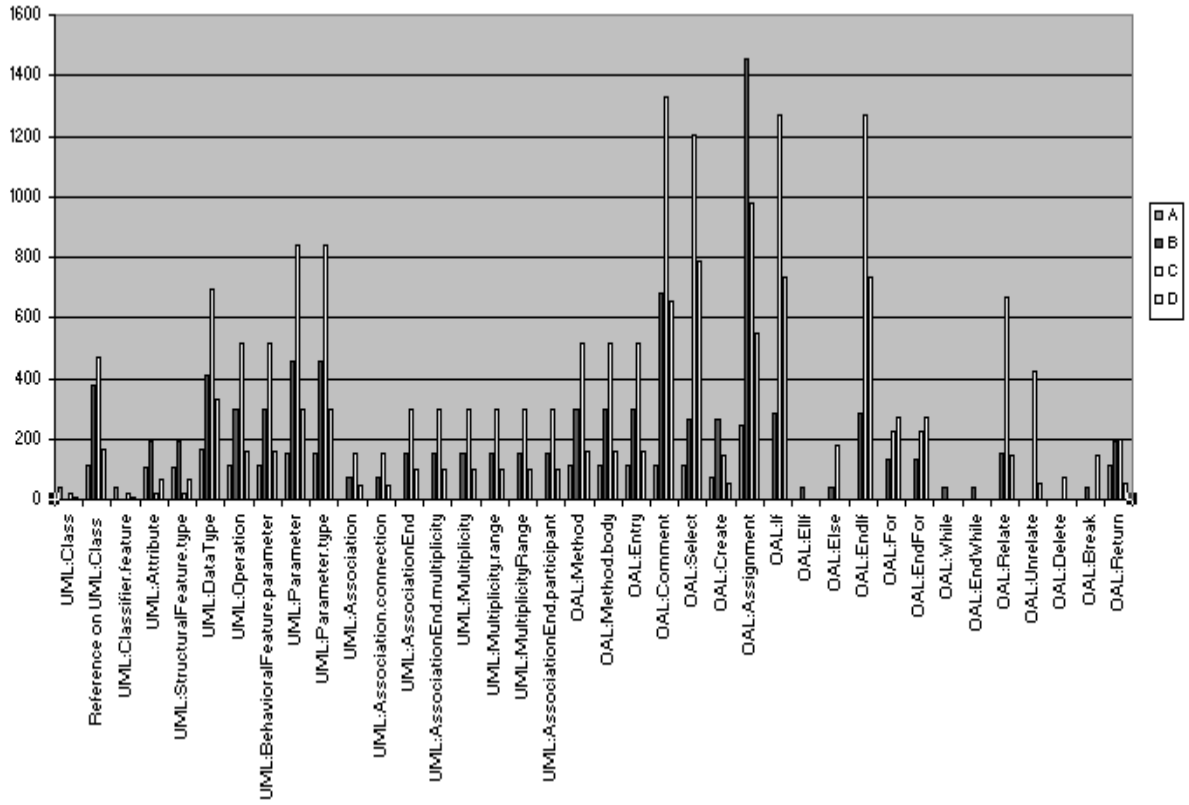
B.2.2 UML Core Package als Informationsmodell

Abbildung 12 Anzahl der generierten XMI-Elemente

XMI-Element	A	B	C	D
UML:Class	38		21	5
Reference on UML:Class	114	380	468	168
UML:Classifier.feature	38		21	5
UML:Attribute	108	191	21	64
UML:StructuralFeature.type	108	191	21	64
UML:DataType	164	412	694	330
UML:Operation	114	298	517	157
UML:BehavioralFeature.parameter	114	298	517	157
UML:Parameter	152	454	840	300
UML:Parameter.type	152	454	840	300
UML:Association		76	150	49
UML:Association.connection		76	150	49
UML:AssociationEnd		152	300	98
UML:AssociationEnd.multiplicity		152	300	98
UML:Multiplicity		152	300	98
UML:Multiplicity.range		152	300	98
UML:MultiplicityRange		152	300	98
UML:AssociationEnd.participant		152	300	98
OAL:Method	114	298	517	157
OAL:Method.body	114	298	517	157
OAL:Entry	114	298	517	157
OAL:Comment	114	684	1331	652
OAL:Select	114	266	1202	790
OAL:Create	76	266	146	56
OAL:Assignment	244	1454	981	548
OAL:If		282	1269	732
OAL:ElIf		37		
OAL:Else		38	180	
OAL:EndIf		282	1269	732
OAL:For		129	223	272
OAL:EndFor		129	223	272
OAL:While		38		
OAL:EndWhile		38		
OAL:Relate		152	665	148
OAL:Unrelate			420	56
OAL:Delete			72	
OAL:Break		38		144
OAL:Return	114	190	199	56
	2106	8659	15791	7165

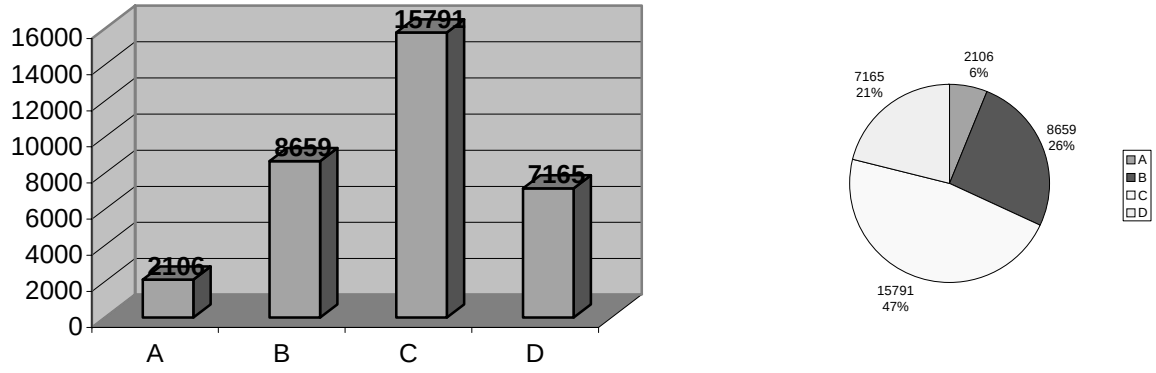
❑

Abbildung 13 Graphische Darstellung der Messergebnisse aus Abbildung 12



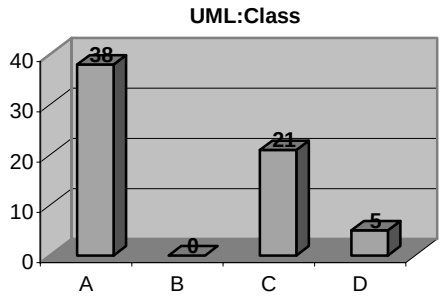
❑

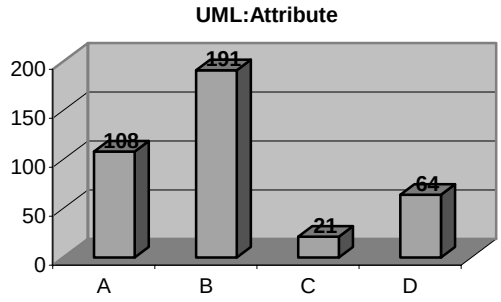
Abbildung 14 Anzahl aller XMI-Elemente



❑

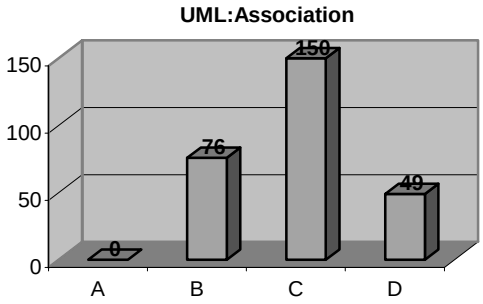
Abbildung 15 Anzahl der <UML:Class>- und <UML:Attribute>-Elemente

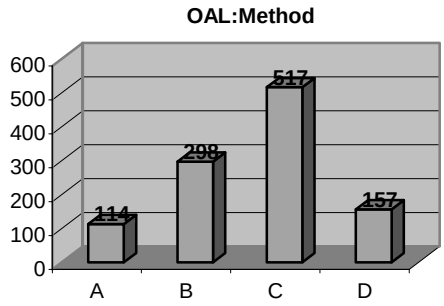




❑

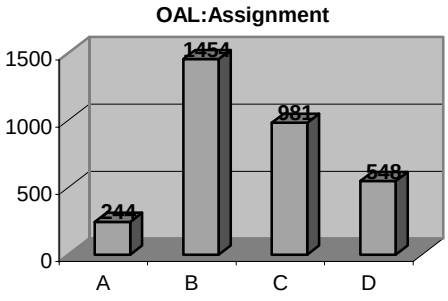
Abbildung 16 Anzahl der <UML:Association>- und <OAL:Method>-Elemente

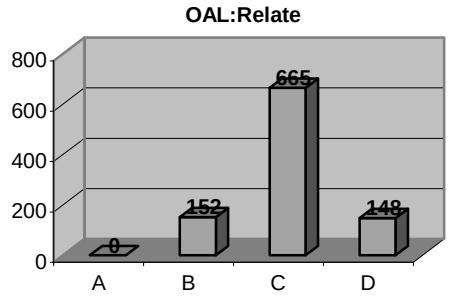




❑

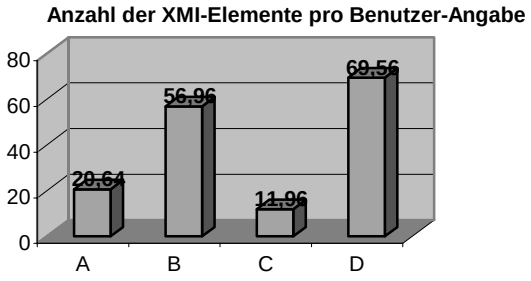
Abbildung 17 Anzahl der <OAL:Assignment>- und <OAL:Relate>-Elemente

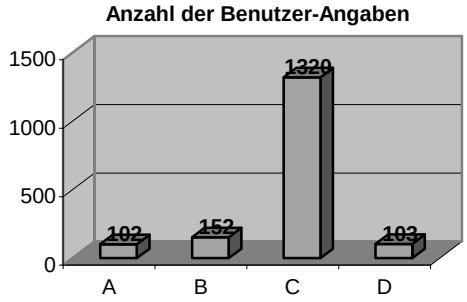




❑

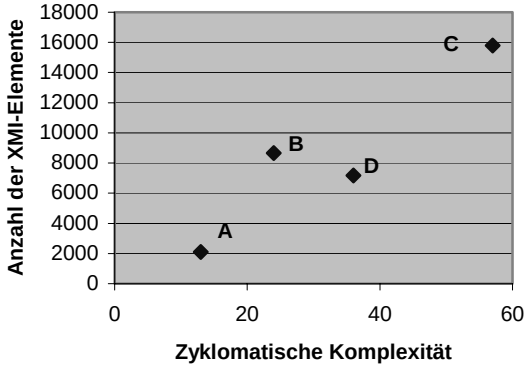
Abbildung 18 Anzahl der XMI-Elemente pro Benutzer-Angabe

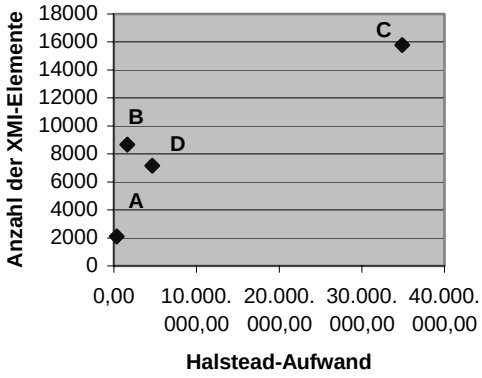




❑

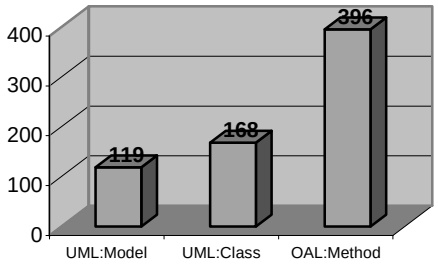
Abbildung 19 Korrelationen zum Aufwand der Vorlagen

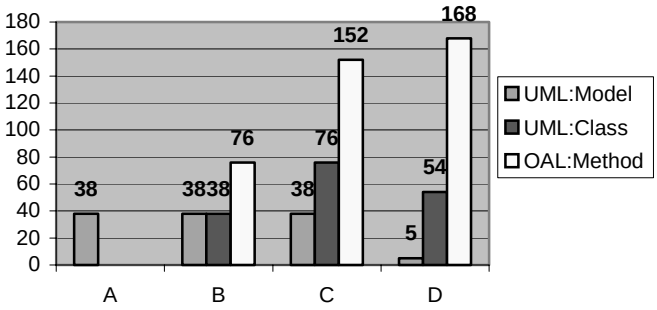




❑

Abbildung 20 Anzahl der durch die Transformation geänderten XMI-Elemente





❑

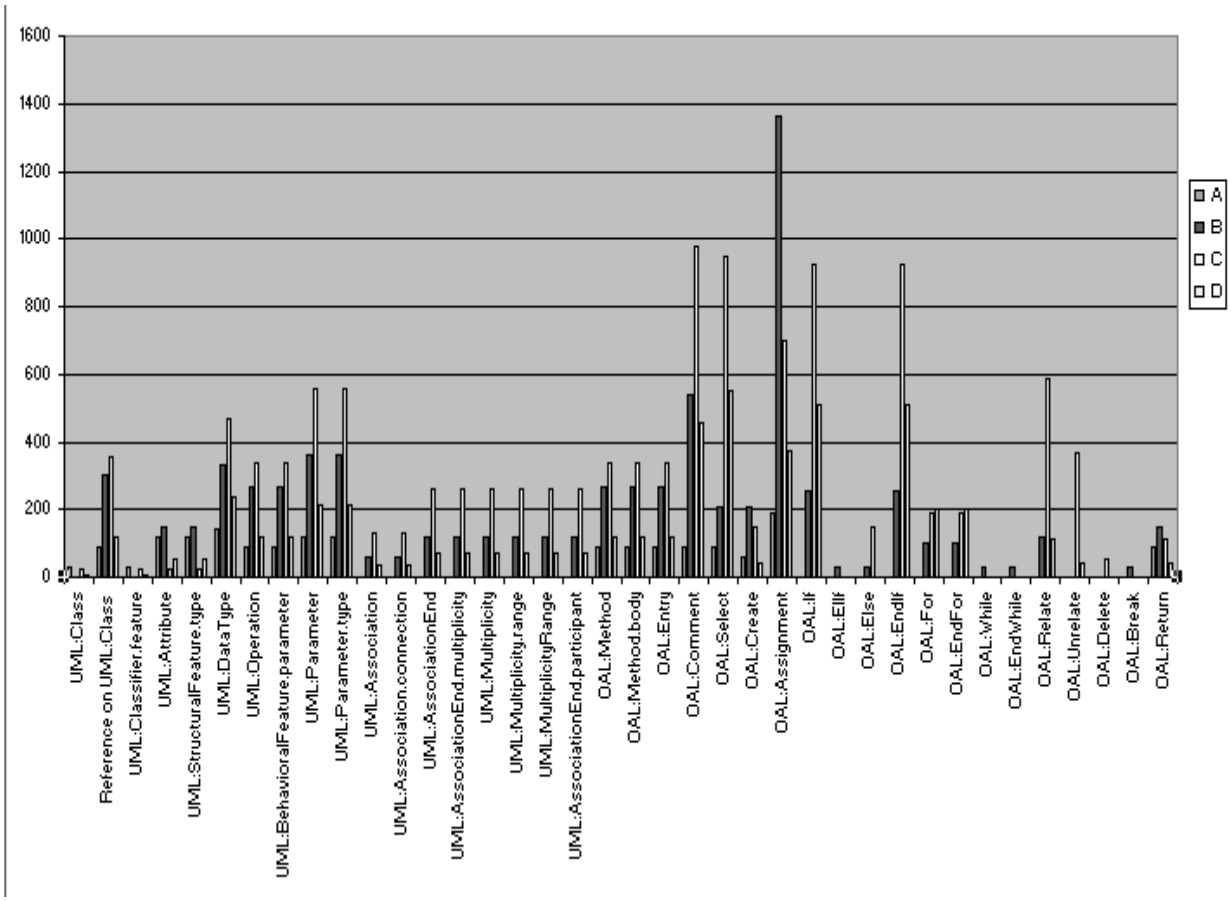
B.2.3 CWM Relational Package als Informationsmodell

Abbildung 21 Anzahl der generierten XMI-Elemente

XMI-Element	A	B	C	D
UML:Class	30		21	5
Reference on UML:Class	90	300	356	120
UML:Classifier.feature	30		21	5
UML:Attribute	118	150	21	52
UML:StructuralFeature.type	118	150	21	52
UML:DataType	140	330	470	240
UML:Operation	90	268	336	121
UML:BehavioralFeature.parameter	90	268	336	121
UML:Parameter	120	360	560	216
UML:Parameter.type	120	360	560	216
UML:Association		60	131	37
UML:Association.connection		60	131	37
UML:AssociationEnd		120	262	74
UML:AssociationEnd.multiplicity		120	262	74
UML:Multiplicity		120	262	74
UML:Multiplicity.range		120	262	74
UML:MultiplicityRange		120	262	74
UML:AssociationEnd.participant		120	262	74
OAL:Method	90	268	336	121
OAL:Method.body	90	268	336	121
OAL:Entry	90	268	336	121
OAL:Comment	90	540	980	454
OAL:Select	90	210	948	550
OAL:Create	60	210	148	40
OAL:Assignment	190	1360	697	374
OAL:If		256	922	510
OAL:ElIf		30		
OAL:Else		30	150	
OAL:EndIf		256	922	510
OAL:For		101	190	200
OAL:EndFor		101	190	200
OAL:While		30		
OAL:EndWhile		30		
OAL:Relate		120	584	112
OAL:Unrelate			368	40
OAL:Delete			56	
OAL:Break		30		
OAL:Return	90	150	112	40
	1736	7284	11811	5059

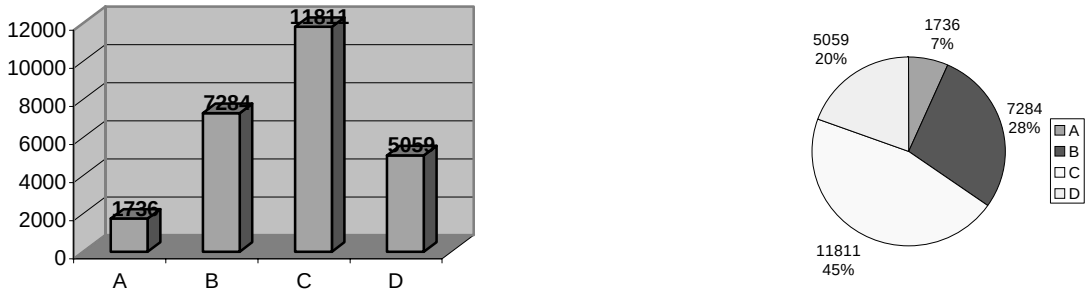
❑

Abbildung 22 Graphische Darstellung der Messergebnisse aus Abbildung 21



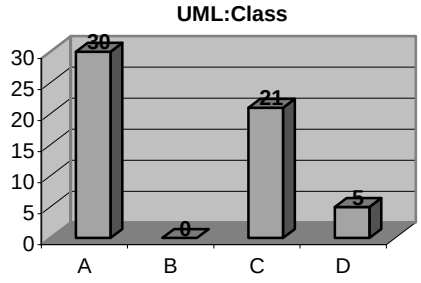
❑

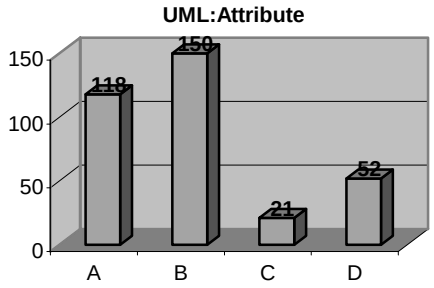
Abbildung 23 Anzahl aller XMI-Elemente



❑

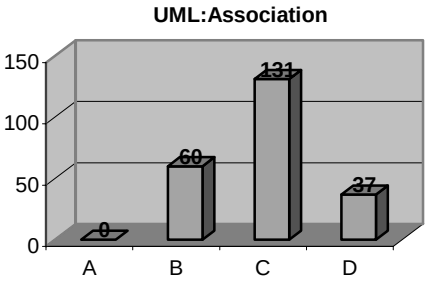
Abbildung 24 Anzahl der <UML:Class>- und <UML:Attribute>-Elemente

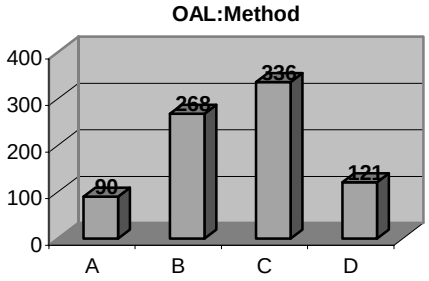




❑

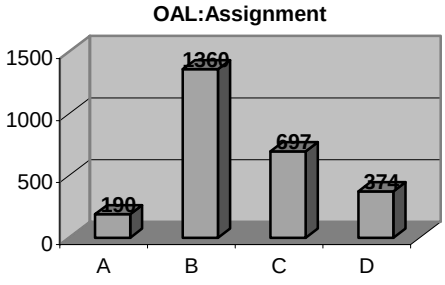
Abbildung 25 Anzahl der <UML:Association>- und <OAL:Method>-Elemente

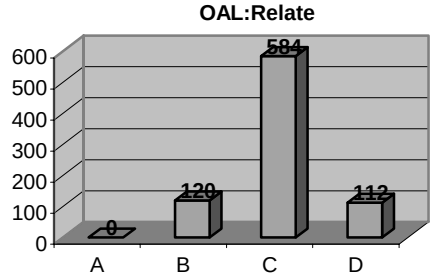




❑

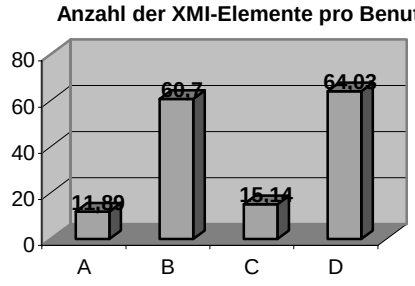
Abbildung 26 Anzahl der <OAL:Assignment>- und <OAL:Relate>-Elemente

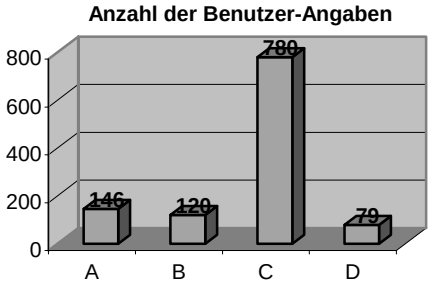




❑

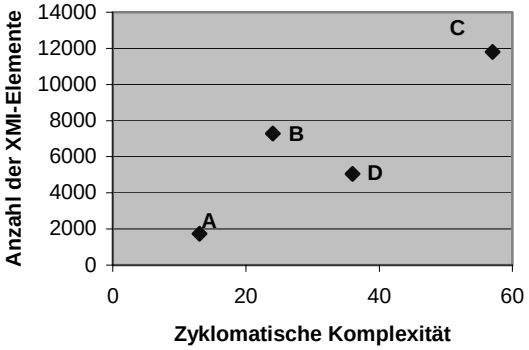
Abbildung 27 Anzahl der XMI-Elemente pro Benutzer-Angabe

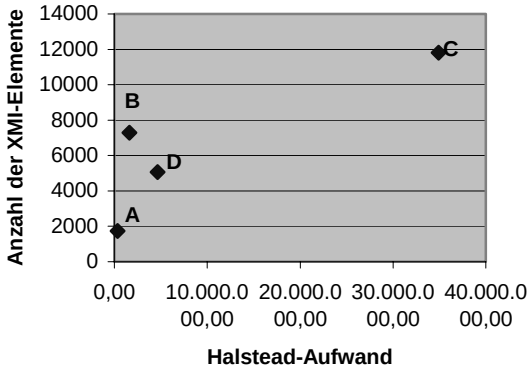




❑

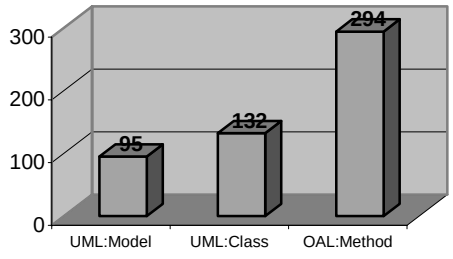
Abbildung 28 Korrelationen zum Aufwand der Vorlagen

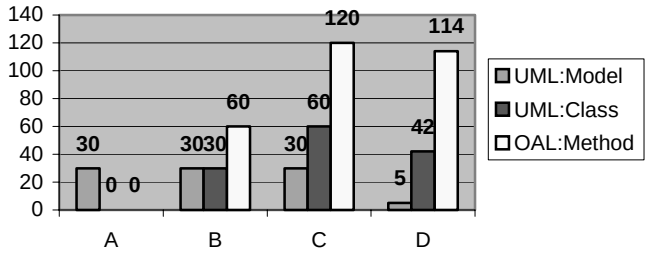




❑

Abbildung 29 Anzahl der durch die Transformation geänderten XMI-Elemente





❑

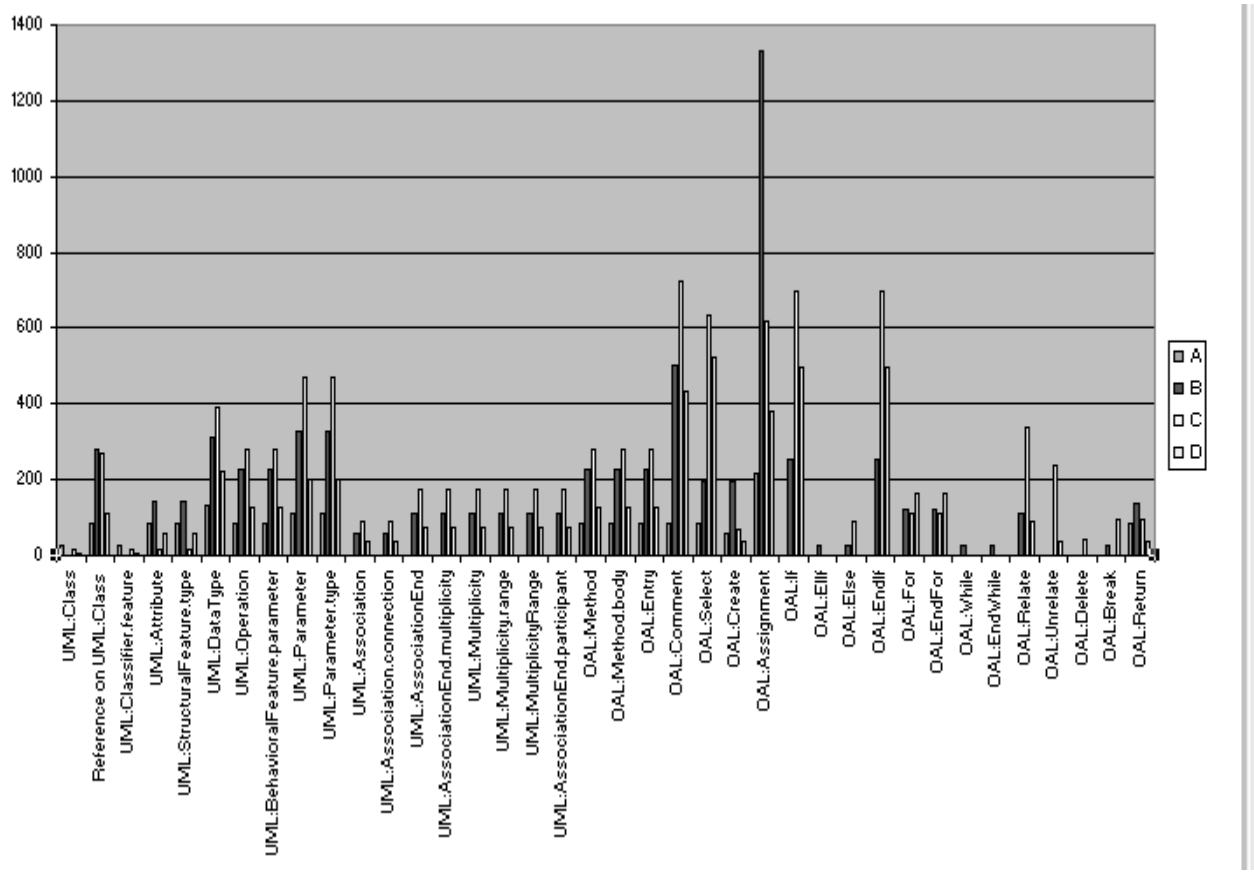
B.2.4 MOF Core Package als Informationsmodell

Abbildung 30 Anzahl der generierten XMI-Elemente

XMI-Element	A	B	C	D
UML:Class	28		14	7
Reference on UML:Class	84	280	270	111
UML:Classifier.feature	28		14	7
UML:Attribute	87	141	14	58
UML:StructuralFeature.type	87	141	14	58
UML:DataType	134	312	390	220
UML:Operation	84	227	282	125
UML:BehavioralFeature.parameter	84	227	282	125
UML:Parameter	112	330	470	199
UML:Parameter.type	112	330	470	199
UML:Association		56	88	37
UML:Association.connection		56	88	37
UML:AssociationEnd		112	176	74
UML:AssociationEnd.multiplicity		112	176	74
UML:Multiplicity		112	176	74
UML:Multiplicity.range		112	176	74
UML:MultiplicityRange		112	176	74
UML:AssociationEnd.participant		112	176	74
OAL:Method	84	227	282	125
OAL:Method.body	84	227	282	125
OAL:Entry	84	227	282	125
OAL:Comment	84	504	726	433
OAL:Select	84	196	633	523
OAL:Create	56	196	69	37
OAL:Assignment	214	1330	618	382
OAL:If		254	698	494
OAL:ElIf		28		
OAL:Else		28	90	
OAL:EndIf		254	698	494
OAL:For		124	110	162
OAL:EndFor		124	110	162
OAL:While		28		
OAL:EndWhile		28		
OAL:Relate		112	339	88
OAL:Unrelate			238	37
OAL:Delete			40	
OAL:Break		28		94
OAL:Return	84	140	94	37
	1614	6827	8761	4945

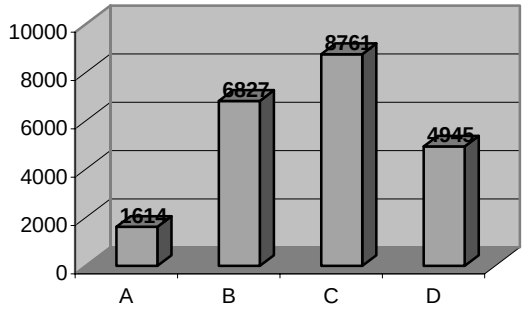
❑

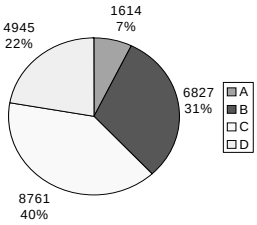
Abbildung 31 Graphische Darstellung der Messergebnisse aus Abbildung 30



❑

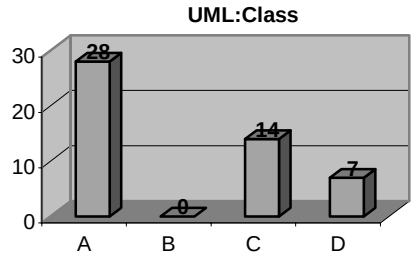
Abbildung 32 Anzahl aller XMI-Elemente

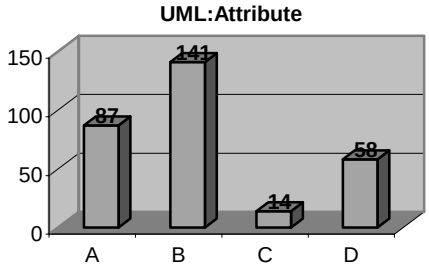




❑

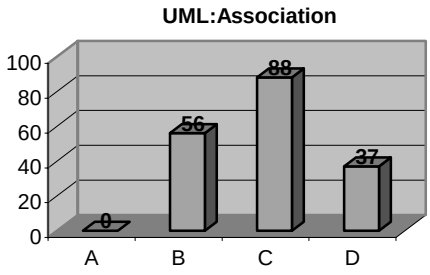
Abbildung 33 Anzahl der <UML:Class>- und <UML:Attribute>-Elemente

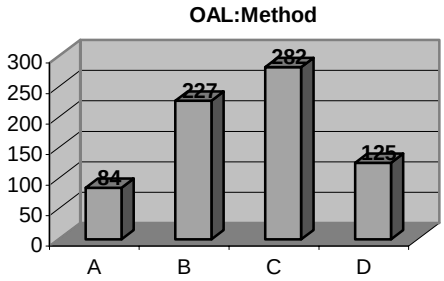




❑

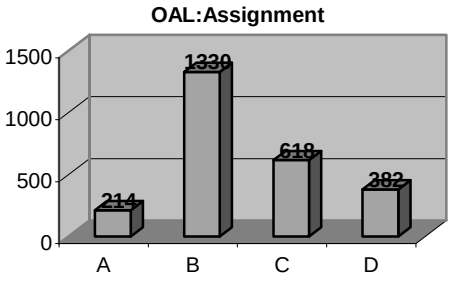
Abbildung 34 Anzahl der <UML:Association>- und <OAL:Method>-Elemente

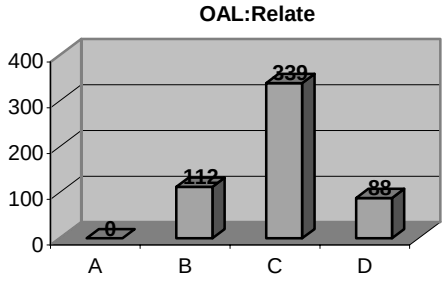




❑

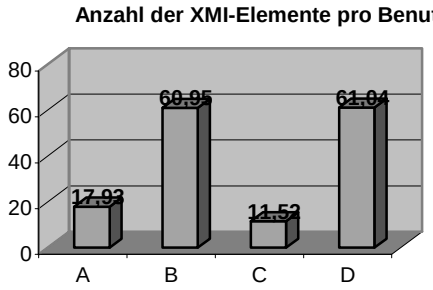
Abbildung 35 Anzahl der <OAL:Assignment>- und <OAL:Relate>-Elemente

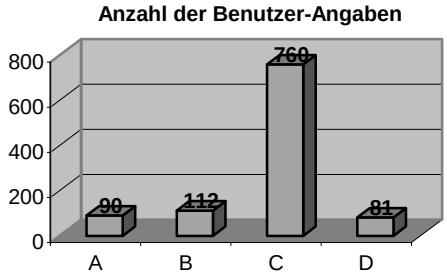




❑

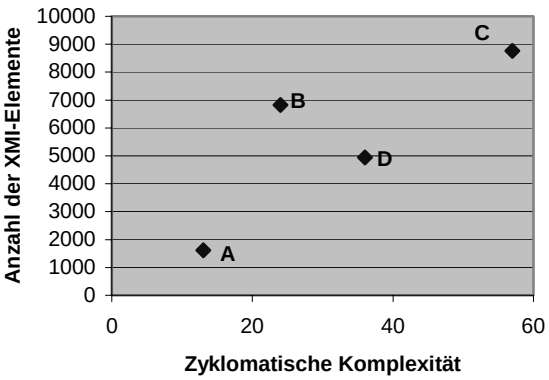
Abbildung 36 Anzahl der XMI-Elemente pro Benutzer-Angabe

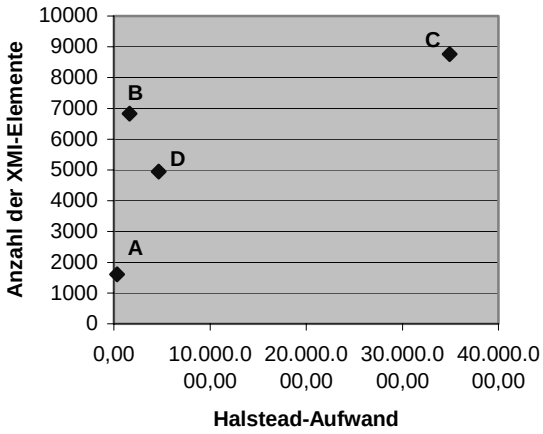




❑

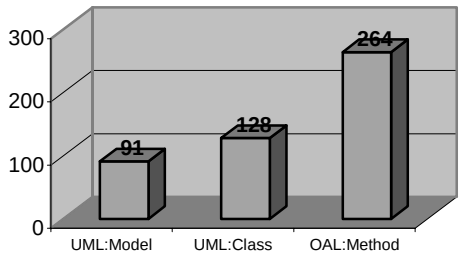
Abbildung 37 Korrelationen zum Aufwand der Vorlagen

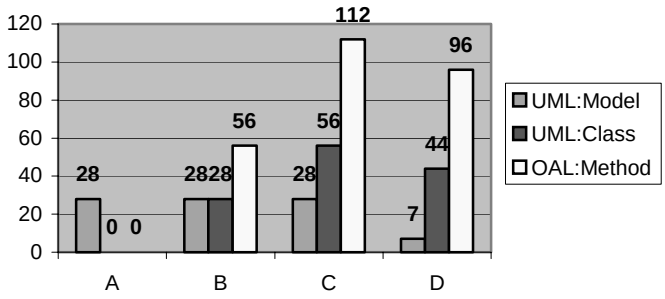




❑

Abbildung 38 Anzahl der durch die Transformation geänderten XMI-Elemente

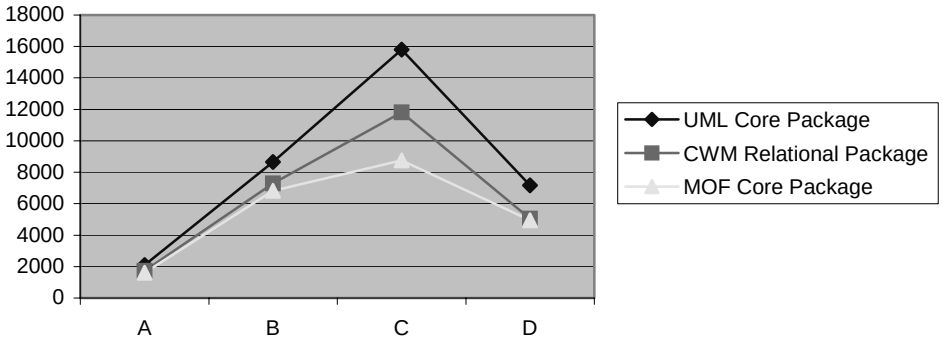




❑

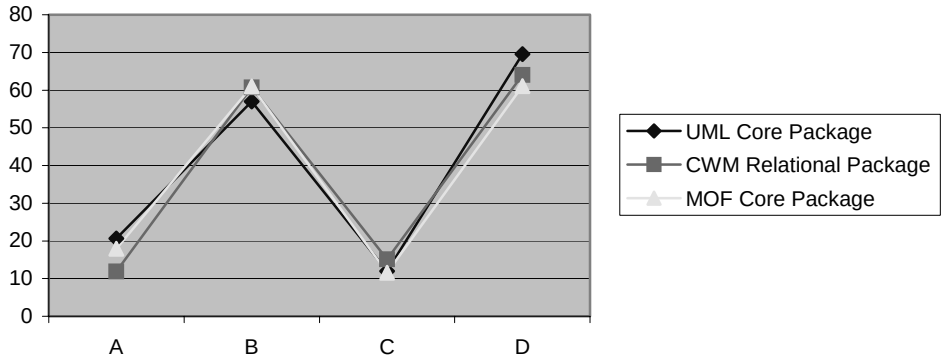
B.2.5 Vergleich

Abbildung 39 Anzahl der generierten XMI-Elemente



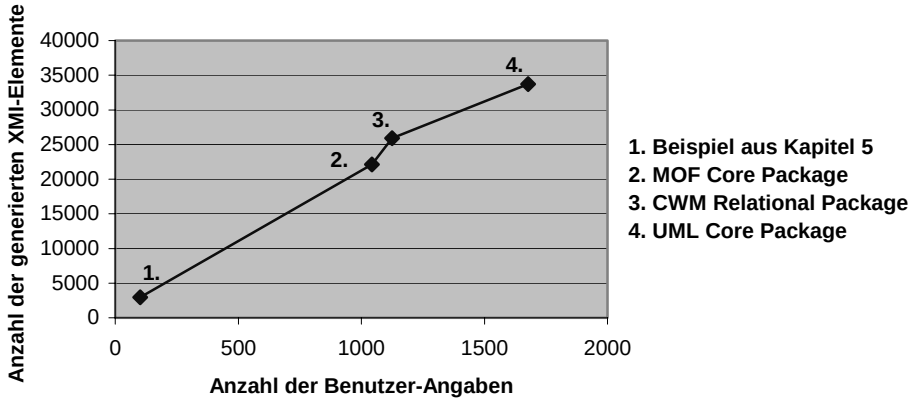
❑

Abbildung 40 Anzahl der XMI-Elemente pro Benutzer-Angabe



❑

Abbildung 41 Korrelation



❑

Literaturverzeichnis

[Apa03a] Apache Software Foundation:
Velocity User Guide, v1.3.1
verfügbar als http://jakarta.apache.org/velocity/user-guide.html
März 2003

[Apa03b] Apache Software Foundation:
Velocity Developer's Guide, v1.3.1
verfügbar als http://jakarta.apache.org/velocity/developer-guide.html
März 2003

[Apa03c] Apache Software Foundation:
VTL Reference Guide, v1.3.1
verfügbar als http://jakarta.apache.org/velocity/vtl-reference-guide.html
März 2003

[Ber98] Bernstein, P.A.:
Repositories and Object-Oriented Databases
SIGMOD Record 27:1 (1998), pp. 34-46

[BM03] Braun, P., Marschall, F.:
The Bi-directional Object-Oriented Transformation Language.
Technische Universität München, TUM-I0307
Mai 2003

[CE00] Czarnecki, K., Eisenecker, U.:
Generative programming: methods, tools and applications
Addison-Wesley, Boston
Juni 2000

[CE97] Czarnecki, K., Eisenecker, U.:
Generatives Programmieren
42. Internationales Wissenschaftliches Kolloquium. 22.-25.09.1997.
Conference Proceedings, Band 1. S. 3-9
Technische Universität Ilmenau, Thüringen
1997

[CE99] Czarnecki, K., Eisenecker, U.:
Vom Fließend - Generative Programmierung: Softwarefabriken bauen und nutzen
iX 7/1999 S. 142-147

[CH03] Czarnecki, K., Helson, S.:
Classification of Model Transformation Approaches
University of Waterloo, Canada
2003

[Cza01] Czarnecki, K.:
Generative Programmierung und die Entwicklung von Softwaresystemfamilien
daimler Chrysler AG
2001

[Gen03] Gentleware:
Poseidon for UML
http://www.gentleware.com/
2003

[Hal77] Halstead, M. H.:
Elements of Software Science
Elsevier North-Holland
1977

[IBM03] IBM Rational Software:
Rational ClearCase
http://www.rational.com/products/clearcase
2003

[JAM03] *Java Model Driven Architecture 0.2*
http://sourceforge.net/projects/jamda/
Mai 2003

[JCP01] Java Community Process:
UML/EJB Mapping Specification 1.0
verfügbar als http://www.jcp.org/aboutJava/communityprocess/review/jsr026
Juni 2001

[Kat90] Katz, R.:
Torward a Unified Framework for Version Modeling in Engineering Databases
ACM Computing Surveys 22:4 (1990)

[KCH+90] Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.:
Feature-Oriented Domain Analysis (FODA). Feasiblity study.
Technical Report CMU/SEI-90-TR-21,
Carnegie Mellon University, Software Engineering Institute
1990

[MB02] Mellor, S., Balcer, M.:
Executable UML: A foundation for Model Driven Architecture
Addison Wesley Professional,
Mai 2002

[McC76] McCabe, T.:
A Complexity Measure
IEEE Transactions on Software Engineering 2:4 (1976), pp. 308-320

[MM03] Miller, J., Mukerji, J.:
MDA Guide Version 1.0.1
http://www.omg.org/cgi-bin/doc?mda-guide
Juni 2003

[OMG01a] Architecture Board of the Object Management Group:
Model Driven Architecture - A Technical Perspective
verfügbar als http://www.omg.org/cgi-bin/doc?ormsc/2001-07-01
Juli 2001

[OMG01b] Object Management Group:
Common Warehouse Metamodel (CWM) Specification, Version 1.0
verfügbar als http://www.omg.org/cgi-bin/doc?formal/01-10-01
Oktober 2001

[OMG02a] Object Management Group:
XML Metadata Interchange (XMI) Specification, Version 1.2
verfügbar als http://www.omg.org/cgi-bin/doc?formal/2002-01-01
Januar 2002

[OMG02b] Object Management Group:
Meta Object Facility (MOF) Specification, Version 1.4
verfügbar als http://www.omg.org/cgi-bin/doc?formal/2002-04-03
April 2002

[OMG02c] Object Management Group:
UML Profile for CORBA Specification, Version 1.0
verfügbar als http://www.omg.org/cgi-bin/doc?formal/02-04-01
April 2002

[OMG03a] Object Management Group:
Unified Modeling Language (UML) Specification, Version 1.5
verfügbar als http://www.omg.org/cgi-bin/doc?formal/03-03-01
März 2003

[OPT03] *OptimalJ 3.0, User´s Guide*
http://www.compuware.com/products/optimalj/default.htm
2003

[Pro02] Project Technology Inc.:
Object Action Language Manual, Version 1.4
verfügbar als http://www.projtech.com/pdfs/bp/oal.pdf
Dezember 2002

[Pro03] Project Technology Inc.:
BridgePoint Development Suite
verfügbar als http://www.projtech.com/pdfs/xtuml/bridgepoint.pdf
Mai 2003

[QVT02] Object Management Group:
MOF 2.0 Query/Views/Transformations RFP
OMG Document: ad/2002-04-10
April 2002

[QVT03] QVT-Partners:
Revised Submission for MOF 2.0 Query/Views/Transformations RFP
OMG Document: ad/2003-08-08
August 2003

[Sti02] Stitzelberger, S.:
Generatives Programmieren und Software Produktlinien
http://www.iste.uni-stuttgart.de/ps/Lehre/HS_Produktlinien/ss-finale.pdf
Mai 2002

[WFMC04] Workflow Management Coalition
http://www.wfmc.org/
2004

[Wit96] Withey, J.:
Investment Analysis of Software Assets for Product Lines
Technical Report CMU/SEI-96-TR-010, Software Engineering Institute,
Carnegie Mellon University, Pittsburgh, Pennsylvania
November 1996

[XDE03] *Rational XDE*
http://www.rational.com/products/xde
2003

[XSL99] W3C:
XSL Transformations (XSLT) Version 1.0
November 1999