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

RepGen - Eine Generierungsplattform für Versionierungsdienste

Diplomarbeit

von
Christian Gebauer

Betreuer:
univ. dipl. inz. Jernej Kovse

Februar 2004

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

Kaiserslautern, den 9. Februar 2004

Christian Gebauer

Inhaltsverzeichnis

Kapitel 1 Einleitung

Die Forschungsdisziplin des Software-Engineerings befasst sich unter anderem mit der Entwicklung von Methoden und Techniken für eine effiziente Konzeption und Implementierung möglichst fehlerfreier Softwaresysteme. Im Bezug auf die Implementierung von Softwaresystemen steigt die Effizienz, je mehr Implementierungskomponenten wiederverwendet werden können. Allerdings ist es in der Praxis nicht einfach, die Wiederverwendungsquote beliebig zu erhöhen. Zwar existieren viele Technologien, die den Softwareentwickler bei dieser Aufgabe unterstützen, angefangen von Betriebssystem- und Compilermechanismen für die Auslagerung von Programmteilen in Bibliotheken, bis hin zu modernen *Komponentenarchitekturen*. Bestehen bleiben aber folgende Probleme, die durch reine Komponententechnologien prinzipbedingt nicht gelöst werden können:

- Während der Entwurfsphase müssen mögliche wiederverwendbare Komponenten manuell gesucht und im Entwurf berücksichtigt werden.
- Aufgefundene Komponenten müssen angepasst werden, falls die Anforderungen, für die sie entwickelt worden sind, nicht mit den Anforderungen des neu zu erstellenden Systems kompatibel sind. Solche Aufwände können vermieden werden, indem wiederzuverwendende Komponente durch einen flexiblen Entwurf für den Einsatz in unterschiedlichen Umgebungen vorbereitet werden, dadurch steigt allerdings auch der Aufwand für die Bereitstellung der wiederzuverwendenden Komponenten.
- Die Anpassung und die Zusammenstellung der wiederverwendeten Komponenten zu einem lauffähigen Softwaresystem erfolgt typischerweise manuell, weshalb menschliche Fehler auftreten können.

Wünschenswert wäre deshalb ein automatisierter Prozess, der aus einer gegebenen Systemspezifikation eine vollständige Implementierung gewinnt, indem er vorgefertigte Komponenten auswählt, anpasst und zusammensetzt. Der *Software-Produktlinienansatz* in Verbindung mit *Softwaregeneratoren* bietet für bestimmte Anwendungsfelder die Möglichkeit, dieses Ziel zu erreichen.

Software-Produktlinien sind ein *domänenbasierter* Softwareentwicklungsansatz, der sich mit der Entwicklung einer Menge von Softwaresystemen befasst, die gemeinsame Anforderungen und Implementierungsbestandteile besitzen. Bei der Entwicklung einer Produktlinie wird die Gesamtheit der Systeme betrachtet, anstatt die Produktlinienmitglieder als separate Einzelsysteme zu entwickeln. Diese Herangehensweise hat Auswirkungen auf den gesamten Entwicklungsprozess, angefangen von einer domänenbasierten Anforderungsanalyse bis hin zur Durchführung der Implementierung. Dort können Generatoren eingesetzt werden, die eine domänenspezifische Modelldarstellung eines Produktlinienmitglieds in eine Implementierung kompilieren.

Der domänenbasierte Blickwinkel der Produktlinienentwicklung ist entscheidend für die Effizienz der Wiederverwendung in ihrem Kontext. Durch die Einschränkung des Entwurfsraums auf eine bestimmte Anwendungsdomäne kann einerseits eine Spezifikationssprache mit einem wesentlichen höheren Abstraktionsgrad als eine universale Programmiersprache entwickelt werden, die eine kompakte und leicht verständliche Spezifikation eines Produktlinienmitglieds ermöglicht. Andererseits kann innerhalb einer Anwendungsdomäne leichter entschieden werden, welche Softwarekomponenten für eine Wiederverwendung aufbereitet werden sollten, bzw. welche Funktionalität in einem Softwaregenerator berücksichtigt werden sollten.

In dieser Arbeit wird die Anwendung dieser Methoden auf eine spezielle Variante von datenbankgestützen Informationssystemen, sogenannten *Versionierungssystemen*, untersucht. Hauptmerkmal eines Versionierungssystems ist die versionierte Speicherung der durch das System verwalteten Daten. Systembenutzer können durch die Schnittstellen des Versionierungssystems nicht nur den aktuellen Stand ihrer Daten einsehen und bearbeiten, sondern auch zurückliegende Versionen anfordern. Solche Systeme werden beispielsweise zur Speicherung von Modellierungsdaten und von Quellcode während der Softwareentwicklung eingesetzt.

Die Anwendungsdomäne der Versionierungssysteme hat besonderes Potential für eine ökonomische Implementierung des Produktlinienansatzes, da sie aufgrund umfangreicher Erfahrungen mit der Entwicklung von Versionierungssystemen klar definiert und abgegrenzt werden kann. Ausserdem beinhaltet sie relativ zu ihrer Größe viele Variationsmöglichkeiten, die es erlauben, eine Vielzahl verschiedener Softwaresystemen innerhalb einer Produktlinie für Versionierungssystem zu entwickeln. Dadurch können sich die hohen anfänglichen Entwicklungskosten einer Produktlinie auszahlen.

Ziel dieser Arbeit ist es zunächst, die festen und variablen Eigenschaften von Versionierungssystemen zu analysieren und darauf aufbauend eine passende Spezifikationssprache als Spracherweiterung der UML zu entwickeln. Anschließend wird ein Generator entwickelt, der diese Spezifikationssprache interpretiert. Dieser Generator erzeugt Implementierungen von Versionierungssystemen, die auf Middleware-Technologien basieren. Besonderes Augenmerk wird bei der Gestaltung dieser Implementierung auf die Semantik der Beziehungen im Datenmodell der Versionierungssysteme gelegt. Ausserdem soll versucht werden, die Möglichkeiten des generativen Ansatz auszunutzen und eine Implementierung zu generieren, die wesentlich stärker an den Eigenschaften eines individuellen Versionierungssystems ausgerichtet ist, als eine typische generische Implementierung. Die Ergebnisse dieser Bemühungen werden im letzten Teil der Arbeit experimentell untersucht. Dabei werden unter anderem die Leistungseigenschaften der generierten und einer vergleichbaren generischen Implementierung von Versionierungssystem verglichen.

Im folgenden Kapitel 2 werden zunächst einige Verfahren und Technologien vorgestellt, auf denen diese Arbeit aufbaut. Dabei handelt es sich einerseits um verschiedene Ansätze zur Wiederverwendung von Softwareprodukten, insbesondere um den hier verwendeten Software-Produktlinienansatz und den verwandten Ansatz der Modellgetriebenen Architekturen (MDA). Andererseits wird die Anwendungsdomäne der Versionierungssysteme vorgestellt, auf die der Produktlinienansatz in dieser Arbeit angewendet wird.

Im Kapitel 3 werden die Merkmale von Versionierungssystemen, die im Kapitel 2 informell vorgestellt wurden, mit Hilfe eines Verfahrens zur domänenbasierten Anforderungsanalyse untersucht. Ergebnis ist ein sogenanntes Merkmalmodell, das die festen und variablen

Eigenschaften von Versionierungssystemen auflistet und die Abhängigkeiten zwischen den Merkmalen darstellt. Ausgehend von diesem Ergebnis wird eine Spezifikationssprache für Versionierungssysteme in der Form einer Spracherweiterung für die UML entwickelt.

Das zentrale Kapitel 4 beschreibt die Abbildung der domänenspezifischen Spezifikationen, die mit Hilfe der entwickelten Spezifikationssprache formuliert werden, auf Middleware-basierte Implementierungen. Diese Abbildung wird durch einen Softwaregenerator automatisiert, der die UML-Modelle analysiert und mit Hilfe von Vorlagen aus den relevanten Modellbestandteile die Dateien der Implementierung erzeugt. In diesem Kapitel wird sowohl auf die Realisierung des Generators eingegangen, als auch auf das Design und die Implementierung der generierten Versionierungssysteme.

In den nachfolgenden Kapitel 5 werden die entwickelten Generatorvorlagen und auch die generierten Softwaresystem mit Softwaremetriken untersucht. Anhand der gewonnen Daten wird die Qualität der Vorlagen in Bezug auf Wartbarkeit und Entwicklungsaufwand beurteilt. Ausserdem wird untersucht, welcher Zusammenhang zwischen dem Implementierungsumfang eines Versionierungssystems und dem Umfang seiner Spezifikation besteht.

Die Leistungseigenschaften der generierten Softwaresysteme werden in Kapitel 6 betrachtet. Anhand eines Benchmarks wird die Verarbeitungsgeschwindigkeit eines Beispielsystems getestet. Die Ergebnisse dieser Tests werden mit Testergebnissen für eine alternativen Implementierung des Beispielsystems verglichen. Diese alternative Implementierung wurde ohne Unterstützung eines Generators entwickelt und verwendet stattdessen ein generisches Framework.

Kapitel 7 fasst abschließend die Ergebnisse dieser Arbeit zusammen und erläutert mögliche weitere Forschungsthemen im Zusammenhang mit der Entwicklung von Generatorvorlagen und der Integration verschiedener Versionierungssysteme.

Kapitel 2 Eine Produktlinie für Versionierungssysteme

Seit Jahrzehnten wird versucht, im Software Engineering den Übergang von der manuellen Fertigung von Einzelstücken zur mehr oder weniger automatisierten Serienfertigung nachzuvollziehen, der in vielen konventionellen Industriezweigen schon vor über hundert Jahren erreicht wurde [CE00]. Dahinter steht die Erwartung, die Produktivitäts- und Qualitätssteigerungen, die in diesen Industriezweigen realisiert werden konnten, auf die Herstellung von Software übertragen zu können. Die möglichen Qualitätssteigerungen resultieren hauptsächlich daraus, dass im Zuge der Umstellung auf eine Serienfertigung ein kontrollierter Prozess eingeführt wird, der Qualitätssicherungsmaßnahmen vorschreibt. Die Durchführung dieser Maßnahmen liegt also nicht mehr in der alleinigen Verantwortung der beteiligten Entwickler, wodurch ein stabiles Qualitätsniveau und eine systematische Steigerung der Qualität ermöglicht wird.

Produktivitätssteigerungen können ebenfalls durch verbesserte Methoden und Werkzeuge erreicht werden, besonders wirkungsvoll ist jedoch in diesem Zusammenhang die Wiederverwendung von bestehenden Arbeitsergebnissen. Potentiell kommen alle Produkte der Softwareentwicklung für eine Wiederverwendung in Frage. In erster Linie natürlich der ausführbare Quellcode, entweder in der Form vollständiger Programme oder in der Form von Teilstücken, also z.B. Funktionen, Klassen oder Frameworks. Aber auch Entwurfsdokumente, Anforderungsbeschreibungen, Benutzerdokumentationen oder Testdaten können wiederverwendet werden.

2.1 Varianten der Wiederverwendung

Wiederverwendung bezieht sich im einfachsten Fall auf den reinen Quellcode eines Softwaresystems. Im kleinen Maßstab funktioniert dies seit langer Zeit, etwa durch den Einsatz von Funktions- bzw. Klassenbibliotheken. Diese Bibliotheken enthalten in der Regel sehr elementare und kompakt zu beschreibende Funktionalität, wie etwa mathematische Funktionen oder Containerklassen. Ein typisches Beispiel ist die Standard-Template-Library (STL), die in der Spezifikation [ISO98] der Programmiersprache C++ enthalten ist. Diese Bibliothek stellt häufig benötigte elementare Datenstrukturen wie z.B. verkette Listen und Vektoren zur Verfügung. Zu diesen Datenstrukturen werden außerdem passende Algorithmen, etwa für Suche und Sortierung, angeboten.

Solche Bibliotheken besitzen typischerweise eine recht generische Funktionalität, deshalb muss die eigentliche Anwendungslogik in der Regel auf herkömmlichen Wege implemen-

tiert werden, die wiederverwendeten Elemente dienen also nur als Infrastruktur. Solange aber umfangreiche individuelle Softwareentwicklung nötig ist, kann der Herstellungsprozess nur im geringen Umfang automatisiert werden. Wenn die angestrebten Rationalisierungsziele erreicht werden sollen, werden größere „Bausteine“, sogenannte *Softwarekomponenten*, benötigt, die nach einem Baukastenprinzip zusammengesetzt werden können. Diese Softwarekomponenten implementieren jeweils einen Teil der Anwendungsanforderungen. Durch eine geeignete Kombination von Einzelkomponenten können deshalb prinzipiell vollständige Softwaresysteme ohne individuelle Implementierungsarbeiten erstellt werden. Der Begriff der Softwarekomponente ist sehr abstrakt, da die konkreten Ausprägungen und Eigenschaften von Softwarekomponenten stark von der verwendeten Technologie und Konstruktionsmethodik abhängig ist. Deshalb ist keine allgemeingültige Definition oder Klassifikation möglich [CE00].

Leider lassen sich die im kleinen Maßstab gut funktionierenden Wiederverwendungsmechanismen nicht einfach für einen komponentenorientierten Ansatz übernehmen. Aufgrund ihres größeren Funktionsumfangs implementieren wiederzuverwendende Softwarekomponenten in der Regel Anforderungen die anwendungsspezifisch sind und damit großen Variationen unterliegen. Die direkte Konsequenz ist, dass häufig keine passenden Komponenten gefunden werden, bzw. gefundene Komponenten aufwendig angepasst werden müssen. Natürlich ist es möglich, diesem Problem entgegenzuwirken, indem schon beim ursprünglichen Entwurf der fraglichen Komponenten die eventuelle spätere Wiederverwendung beachtet wird. Dann können entsprechende Variationsmöglichkeiten vorsehen werden und auf diese Weise die spätere Anpassung der Komponente erleichtert werden. Nur wie soll zum Entwurfszeitpunkt entschieden werden, welche Variationsmöglichkeiten in Zukunft benötigt werden bzw. ob die Komponente überhaupt in Zukunft wiederverwendet wird? Diese Entscheidungen können eigentlich nur dann korrekt getroffen werden, wenn die Anforderungen der zukünftigen Softwaresysteme bekannt sind. Da dies in der Realität nur selten zutrifft, werden die möglichen Vorteile der Wiederverwendung von Softwarekomponenten durch die erhöhten Aufwände bei der Bereitstellung und der Verwendung von Komponenten gemindert oder sogar aufgehoben.

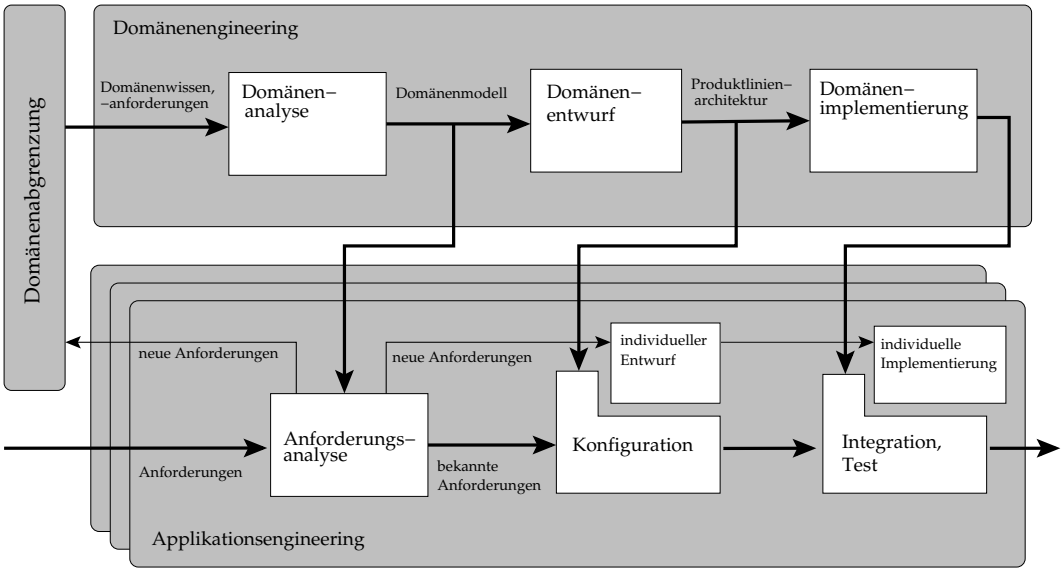
2.2 Software-Produktlinien

Software-Produktlinien bieten eine Möglichkeit, diese grundsätzliche Problematik in bestimmten Fällen zu lösen. Ihre Anwendung setzt vorraus, dass die Softwareentwicklungsorganisation nicht nur einzelne, sehr unterschiedliche Systeme entwickeln will, sondern Ansammlungen ähnlicher Systeme, sogenannte *Produktlinien*. Die Mitglieder einer Produktlinie haben ausgesuchte gemeinsame Eigenschaften und befriedigen damit die Bedürfnisse eines bestimmten Marktes oder Marktsegments [Wit96]. Da die Mitglieder einer Produktlinie typischerweise eine große Anzahl gemeinsamer Anforderungen besitzen, können sie als *Systemfamilie* entwickelt werden. Dies bedeutet, dass übereinstimmende Anforderungen in standardisierten, wiederverwendeten Softwarekomponenten implementiert sind [Wit96]. Insbesondere besitzen die Mitglieder einer Systemfamilie eine gemeinsame *Softwarearchitektur*, die die Einbettung der übrigen wiederzuverwendenden Komponenten erleichtert.

2.2.1 Die Phasen der Produktlinienentwicklung

Die Tätigkeiten, die während der Softwarentwicklung auftreten, werden bei der Entwicklung von Software-Produktlinien in zwei logische Phasen aufgeteilt: *Domänenengineering* (Domain Engineering) und *Applikationsengineering* (Application Engineering) [CE00]. Die Abbildung 2.1 gibt einen Überblick über das Zusammenspiel der in diesen beiden Phasen enthaltenden Vorgänge. Die dargestellten Abhängigkeiten zwischen den einzelnen Vorgängen sollen dabei nicht unbedingt den zeitlichen Ablauf der Entwicklung beschreiben, sondern sind in erster Linie inhaltlich begründet. Der Produktlinienansatz ist also a priori nicht an eine bestimmte Softwareentwicklungsmethode gebunden.

Abbildung 2.1 Schematische Darstellung der Produktlinienentwicklung (basierend auf [CE00])



Domänenengineering

Das Domänenengineering beschäftigt sich mit der Entwicklung aller *wiederverwendbaren Artefakte*, die innerhalb der Produktlinie benötigt werden. Bei diesen Artefakten handelt es sich nicht nur um reine Softwarekomponenten, also Quellcode, oder um die schon erwähnten Softwarearchitekturen, sondern auch um alle möglichen anderen Arten von Arbeitsergebnissen und Entwicklungswissen. Dazu können beispielsweise auch wiederverwendbare Anforderungen, Quellcodegeneratoren oder domänenspezifische Spezifikationssprachen gehören.

Um den Bedarf an wiederverwendbaren Artefakten möglichst genau prognostizieren zu können, ist dem eigentlichen Entwicklungsprozess die *Domänenabgrenzung* (Scoping) vorgelagert. Während dieser Phase wird die Anwendungsdomäne definiert, die durch die Mitglieder der Produktlinie abgedeckt werden soll. Die Abgrenzung soll fundierte Entscheidungen ermöglichen:

1. Ist es ökonomisch und technisch sinnvoll, ein bestimmtes Produkt innerhalb der Produktlinie zu entwickeln, oder sollte zu diesem Zweck eine separate Produktlinie eingerichtet werden?

2. Soll eine bestimmte Anforderung innerhalb des Domänenengineering oder innerhalb des Applikationsengineering berücksichtigt werden?

Die korrekte Beantwortung dieser Fragen ist eine wichtige Voraussetzung für den wirtschaftlichen Erfolg der Produktlinie.

Bei den Anwendungsdomänen ist zwischen *vertikalen* und *horizontalen* Domänen zu unterscheiden [SCK+96]. Vertikale Domänen sind als Gruppen von vollständigen Softwaresystemen definiert, die gewisse externe Anwenderanforderungen erfüllen. Ein Beispiel für eine vertikale Domäne sind Tabellenkalkulationsprogramme in ihren verschiedenen Ausprägungen. Horizontale Domänen beschreiben nur Teilaspekte von Softwaresystemen. Dabei kann es sich um abgeschlossene Subsysteme handeln (encapsulated domains) oder auch um Aspekte die über das gesamte System verteilt sind (diffused domain). Typische Beispiele für den erstgenannten Typ sind Datenbanksysteme und Anwendungsserver.

Anhand der Domänendefinition können nun im nächsten Schritt, der *Domänenanalyse*, die Anforderungen aller Beteiligten ermittelt und analysiert werden. Das Ergebnis ist ein *Domänenmodell,* das alle festen und variablen Eigenschaften der Domäne beschreibt. Insbesondere werden die Abhängigkeiten zwischen den variablen Eigenschaften festgehalten, um feststellen zu können, welche der möglichen Kombinationen sinnvoll und zulässig sind.

Das ermittelte Domänenmodell dient als Grundlage zum Entwurf der zentralen Produktlinienarchitektur im nächsten Abschnitt des Domänenengineering, dem *Domänenentwurf*. Die Architektur einer Produktlinie ist ihr wichtigstes wiederverwendbares Artefakt. Sie bildet die Grundlage für den Entwurf der teilnehmenden Systeme und muss deshalb entsprechend flexibel konstruiert werden.

Während der *Domänenimplementierung* werden schließlich alle Softwarekomponenten implementiert, für die während der vorrangegangen Analyse- und Entwurfsschritte festgestellt wurde, dass sie für eine Wiederverwendung in Betracht kommen. Dies sind typischerweise Komponenten, die feste bzw. nur wenig variable Anforderungen der Domäne umsetzten.

Applikationsengineering

Wie aus der Abbildung 2.1 hervorgeht, fließen in die eigentliche Entwicklung konkreter Softwaresysteme, dem *Applikationsengineering*, neben den reinen Softwarekomponenten auch alle anderen Ergebnisse des Domain Engineering ein. Das Domänenmodell beispielsweise dient als Grundlage für die individuelle Anforderungsanalyse. Hierbei werden die Anforderungen des Kunden daraufhin untersucht, ob sie bereits bekannt und in der Produktlinie berücksichtigt sind. Falls dies nicht der Fall ist, müssen die neuen Anforderungen entweder in die Produktlinie aufgenommen werden, was eine neue Iteration des Domänenengineering Prozesses nötig macht, oder sie werden individuell für das neue Softwaresystem implementiert. Die erste Variante wird typischerweise dann gewählt, wenn abzusehen ist, dass die neuartige Anforderung in Zukunft bei anderen Projekten ebenfalls auftreten wird. Alle identifizierten Anforderungen dienen bei den folgenden Entwurfs- und Implementierungschritten als Konfigurationsdaten für die Instantiierung der wiederverwendbaren Artefakte. Anhand dieser Konfigurationsdaten werden die passenden Artefakte ausgewählt und gegebenfalls angepasst. Auf die neuartigen Anforderungen wird durch Erweiterungen der Produktlinienarchitektur um neue Entwurfselemente und durch die Implementierung der zugehörigen, nicht wiederverwendeten Softwarekomponenten reagiert.

2.2.2 Zusammenstellung von Produktlinienmitgliedern

Im letzten Abschnitt war die Rede davon, dass während der Erstellung eines neuen Softwaresystems Komponenten aus der Menge der verfügbaren Komponenten ausgewählt und angepasst werden müssen. Wie dies geschieht, hängt stark von der Beschaffenheit der Softwarearchitektur und ihrer Komponenten ab, deshalb wird während des Domänenentwurfs ein *Produktionsplan* definiert, der diesen Vorgang festlegt. Es sind dabei folgende Varianten mit steigendem Automatisierungsgrad denkbar [Coh99]:

- *Manuelle Zusammenstellung der Komponenten.* Im einfachsten Fall erhält der Entwickler Dokumentation zur Architektur und zu den wiederverwendbaren Komponenten, z.B. in Form von UML-Modellen oder textuellen Beschreibungen. Mit Hilfe dieser Dokumente kann er die Architektur verstehen und in der neuen Anwendung implementieren. Die Anpassung der zur Verfügung stehenden Komponenten erfolgt durch konventionelle Programmiertechniken. Beispielsweise kann bedingte Kompilierung eingesetzt werden, indem also Kompilierdirektiven konfiguriert werden, die der Komponentendokumentation entnommen werden können. Dieser Prozess kann durch ein Beispielsystem unterstützt werden, das eine lauffähige Minimalkonfiguration eines Produktlinienmitglieds darstellt. Dieses Beispiel kann zum Verständnis der Softwarearchitektur beitragen und als Ausgangspunkt für die Implementierung genutzt werden.
- *Werkzeugunterstützung für Auswahl und Anpassung*. Die Arbeit des Anwendungsentwicklers kann durch Werkzeuge erleichtert werden. Diese ermöglichen beispielsweise eine komfortable Suche im Komponentenkatalog oder generieren Teilbereiche des Softwaresystems.
- *Automatische Zusammenstellung*. Der höchstmöglichste Automatisierungsgrad ist erreicht, wenn ein einziger Generator aus einer im passenden Format vorliegenden Konfigurationsspezifikation das gesamte Softwaresystem erstellen kann. Manuelle Programmierung ist nur noch für die Erfüllung von Anforderungen erforderlich, die noch nicht in der Produktlinie berücksichtigt worden sind.

Ein generativer Ansatz bietet viele Vorteile gegenüber einem traditionellen Entwicklungsprozess. Das wichtigste Argument für den Einsatz eines Generators ist natürlich die Beschleunigung der Implementierungsphase. Im Idealfall kann aus der Spezifikation direkt die Implementierung gewonnen werden. Außerdem können auch die Aufwendungen in der Integrations- und Testphase verringert werden. Die Integration von Komponenten aus verschiedenen Quellen entfällt bei der generativen Programmierung vollständig, und auch das Testen wird vereinfacht. Einerseits kann die Funktionsfähigkeit der Software prinzipiell schon durch Tests während der Entwicklung des Generators sichergestellt werden, andererseits kann der Generator neben dem reinen Programmcode auch Testdaten und Testcode generieren, die einen automatisierten Test der Anwendung ermöglichen.

Aber die generative Programmierung bietet auch die Chance, qualitativ höherwertige Programme zu erzeugen, als dies bei einer komponentenbasierten Entwicklung möglich wäre, die die gleichen Variationsmöglichkeiten beachten müsste. Ein Generator hat bei der Anpassung des Quellcodes sehr große Freiheiten. So muss er z.B. einen Konfigurationsaspekt auf hohen Abstraktionsniveau nicht unbedingt auf ein einzelnes Quellcodemodul abbilden, sondern kann an allen Modulen Veränderungen vornehmen. Im Gegensatz dazu sehen z.B. Frameworks Veränderungen nur an einzelnen vorgesehen Punkten vor, etwa durch die Ableitung von Klassen und der Überladung von Methoden. Bedingt durch diese starre Struktur muss eine sehr generische Softwarearchitektur verwendet werden, die aufwendiger zu im-

plementieren ist und schlechtere Leistungseigenschaften bietet, als eine generierte und damit gut angepasste Architektur.

Generatoren können zusätzlich auf Quellcodeebene wesentlich wirksamere Optimierungen durchführen als dies später durch den Compiler geschehen kann, da dem Generator zusätzliche Informationen auf höheren Abstraktionsebenen zur Verfügung stellen. Beispielsweise kann es mit Hilfe von Domänenwissen möglich sein, bestimmte aufwendige Ausdrücke durch Konstanten zu ersetzen, die während der Generierung berechnet werden. Oder es können in Abhängigkeit zur gewählten Konfiguration generische Algorithmen durch speziell optimierte Algorithmen ersetzt werden.

Leider ist eine vollständige Generierung nicht in allen Fällen sinnvoll durchführbar. Die Anwendungsdomäne sollte nämlich gut verstanden sein, um eine formale Beschreibung der Konfiguration von Produktlinienmitgliedern, beispielsweise in der Form einer domänenspezifischen Programmiersprache, zu ermöglichen. Außerdem steigt mit der Anzahl der nötigen Variationspunkte und der zulässigen Konfigurationskombinationen der Aufwand für die Implementierung und den Test des Generators. Die Komplexität der Domäne sollte also in einem sinnvollen Verhältnis zum Nutzen des Generators, also der Anzahl der tatsächlich benötigten Softwaresysteme, stehen. Die in dieser Arbeit als Beispielanwendung gewählte Domäne der Versionierungssysteme erfüllt diese Ansprüche. Versionierungssemantiken sind ausgiebig theoretisch untersucht worden und es existieren auch viele Referenzimplementierungen, es ist deshalb eher unwahrscheinlich, das im Laufe der Lebenszeit der Produktlinie völlig neue und inkompatible Anforderungen an diese Systeme entstehen. Außerdem können mit einer überschaubaren Anzahl von Variationspunkten sehr viele sinnvolle Systeme generiert werden.

2.3 Modellgetriebene Architekturen (MDA)

Modellgetriebene Architekturen stellen formale Modelle in den Mittelpunkt des Softwareentwicklungsprozesses. Dadurch soll eine klare Trennung zwischen der eigentlichen Anwendungsfunktionalität und von der zugrundeliegenden Technologieplattform abhängigen Implementierungsdetails erreicht werden. Diese Idee ist die logische Fortsetzung der durch die Einführung von höheren Programmiersprachen erreichten Abstraktion von einer konkreten Hardwarearchitektur [MM03]. Genauso wie ein Hochsprachenprogramm durch erneute Übersetzung im Prinzip auf allen Hardwarearchitekturen lauffähig ist, für die ein entsprechender Compiler verfügbar ist, sollen plattformunabhängig spezifizierte Softwaresysteme nach entsprechenden Transformationsschritten in verschiedene Softwareumgebungen eingebunden werden können. Der konkrete Anlass für die OMG die *Model Driven Architecture* (MDA) [MM03] zu entwickeln, war die Erkenntnis, dass sich in absehbarer Zeit kein Middlewarestandard dauerhaft durchsetzten wird. Dafür gibt es verschiedene Gründe, etwa dass prinzipiell universal einsetzbare Middleware wie z.B. CORBA [OMG02a] nicht in allen Spezialfällen die optimale Lösung darstellt. Deshalb besteht ein großer Bedarf, in betrieblichen Informationssystemen mehrere Middlewarearchitekturen parallel zu unterstützen.

2.3.1 Modellierungstechniken

Die MDA-Methode schreibt keine speziellen Modellierungstechniken vor, es wird aber der Einsatz von OMG-Modellierungstechnologien, insbesondere von MOF [OMG02b] und

UML [OMG03a] propagiert. Angesichts der weiten Verbreitung dieser Standards und der reichlich vorhandenen Werkzeugunterstützung erscheint dies auch sinnvoll.

Im Unterschied zu anderen Modellierungsaufgaben ist es bei der Anwendung der MDA sehr wichtig, nicht nur die Struktur eines Systems, sondern auch seine Semantik vollständig formal zu beschreiben. Dies war in UML bisher nicht direkt möglich, seit der Version 1.5 [OMG03a] der UML-Spezifikation enthält der Standard aber die nötige Funktionalität um ausführbare UML-Modelle (Executable UML) konstruieren zu können. Zu diesem Zweck wurden sogenannte *Action Languages* definiert, die auf der Ebene der Modellobjekte operieren und die Spezifikation von Methoden und Funktionen erlauben ohne auf traditionelle Implementierungssprachen zurückgreifen zu müssen, die zwangsläufig plattformabhängig sind. Die Action Languages haben eine textuelle Repräsentation wie jede andere normale Programmiersprache, aber ihre Semantik wird intern auf UML-Objekte aus dem neugeschaffenen *Action* Package abgebildet, die Modellierung ist deswegen vollständig formal.

Die Action Languages sind universal einsetzbar und führen deshalb zu umfangreichen und aufwendig zu erstellenden Modellen. Deshalb erscheint es sinnvoll, wo immer möglich, auf eine domänenspezifische Darstellung der Semantik zurückzugreifen, die eine wesentlich kompaktere Darstellung ermöglicht, weil die eigentliche Definition der Semantik im Domänenmodell enthalten ist. Diese kompakte Darstellung ist einfacher zu verstehen und zu analysieren als eine umfangreichere generische Darstellung. Im Rahmen der UML-Technologien ergeben sich drei Möglichkeiten zur Realisierung von domänenspezifischer Modellierung:

- *Definition eines eigenen MOF-Metamodells.* Die größten Freiheiten bei der Gestaltung einer domänenspezifischen Modellierungssprache bestehen, wenn auf der Basis des MOF-Metamodells ein eigener Ersatz für das UML-Metamodell erstellt wird. Dieser Ansatz wurde beispielsweise von der OMG für das *Common Warehouse Metamodel (CWM)* [OMG03b] gewählt. Die Verwendung dieses Ansatzes ist nur dann sinnvoll, falls große Teile des UML-Metamodells nicht benötigt werden, ansonsten müssten diese nämlich im neu geschaffenen Metamodell nachgebildet werden. Ein großer Nachteil ist weiterhin, dass die resultierenden Modelle nicht mit UML-basierten Werkzeugen bearbeitet werden können.

- *Erweiterung des UML-Metamodells (heavyweight extensions).* In dieser Variante wird das Metamodell ebenfalls verändert, allerdings wird das UML-Metamodell als Basis verwendet und um neuere Elemente erweitert. Auch hier besteht das Problem, dass UML-basierte Werkzeuge mit den Erweiterungen nicht umgehen können.

- *UML-Profile (lightweight extensions).* Eine Alternative zur Veränderung des Metamodells sind die nativen Erweiterungsmöglichkeiten die im UML-Metamodell definiert sind, die sogenannten *Profile*. Sie ermöglichen es, sogenannte *Stereotypes* zu definieren, die die Semantik eines Elementtyps verändern. Sobald einem Modellelement dieses Typs der Stereotype zugeordnet wird, übernimmt er dessen Semantik. Beispielsweise könnten mit einem UML-Profil für J2EE-Anwendungen [JCP01] Klassen als Entity- bzw. Sessionbeans gekennzeichnet werden.

Die verschiedenen domänenspezifischen Erweiterungsmöglichkeiten und insbesondere die UML-Profile werden im nächsten Kapitel noch detaillierter behandelt werden.

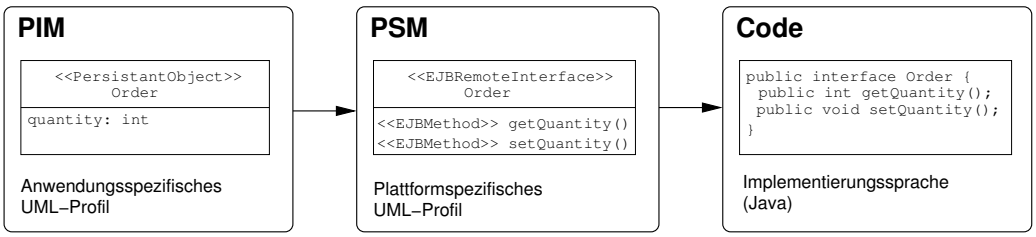
2.3.2 Modelltypen

Die MDA benutzt mehrere verschiedene Modelltypen auf verschiedenen Abstraktionsstufen. Unter Abstraktion wird in diesem Zusammenhang das Weglassen von Teilaspekten des modellierten Systems, die für die aktuelle Modellierungsaufgabe nicht relevant sind, verstanden. Während der Konstruktion eines Softwaresystems werden Modelle mit absteigenden Abstraktionsgrad sukzessive konstruiert bzw. von bestehenden Modellen mit höherem Abstraktionsgrad abgeleitet. Wie bei allen Entwurfsvorgängen üblich, sollte dabei die vertikale Verfolgbarkeit erhalten bleiben, d.h. dass z.B. zu einer modellierten Benutzeranforderung die Module auf niedrigeren Modellebenen identifiziert werden können, die diese Anforderung implementieren.

Verarbeitungsunabhängige Modelle (CIM)

Verarbeitungsunabhängige Modelle (computation independent model, CIM) enthalten keine Informationen über die Struktur des beschriebenen Systems, sondern spezifizieren die Umgebung und die Anforderungen an das Systems. Das CIM wird während der Anforderungsanalyse erstellt und enthält beispielsweise Use-Case- und Sequenzdiagramme. Um die Verifikation der Anforderungen und die Kommunikation zwischen Softwareentwicklern und Anwendern zu erleichtern, sind das verwendete Vokabular und die Darstellungsformen so ausgelegt, dass sie von Anwendern und anderen Domänenexperten verstanden werden können.

Abbildung 2.2 Darstellung eins Entwurfsobjekts in den verschiedenen Modelltypen



Plattformunabhängige Modelle (PIM)

Plattformunabhängige Modelle (platform independent model, PIM) stellen die Struktur und die Semantik des Softwaresystems da, soweit sie nicht von der zugrundeliegenden Technologieplattform abhängig ist, es werden also alle Elemente modelliert, die bei einem Plattformwechsel nicht verändert werden. Natürlich kann ein System nur sehr unvollständig spezifiziert werden, wenn über seine Schnittstellen keine Aussagen gemacht werden können. Deshalb wird hilfsweise eine virtuelle Maschine bzw. Plattform definiert, die die Dienste (z.B. Kommunikations- und Namensdienste) und Strukturen der zugrundeliegenden realen Plattform abstrahiert. Das resultierende PIM ist nur noch in sofern plattformunabhängig, als das es auf allen Plattformen instantiiert werden kann, für die die Konstrukte der virtuellen Maschine auf konkrete Konstrukte abgebildet werden können.

Zur Modellierung werden auf dieser Abstraktionsebene in der Regel eine formale Sprache wie z.B. UML verwendet, die die oben erwähnten Erweiterungen aufweisen kann. Die Auswahl der Erweiterungen wird zum einen durch die Anwendungsdomäne und zum anderen

durch die gewählte virtuelle Plattform beeinflusst. Abbildung 2.2 enthält ein Beispiel für ein Entwurfsobjekt aus einem PIM. In diesem Beispiel wurde ein UML-Profil verwendet, mit dem u.A. Objekte gekennzeichnet werden können, die persistent gespeichert werden sollen.

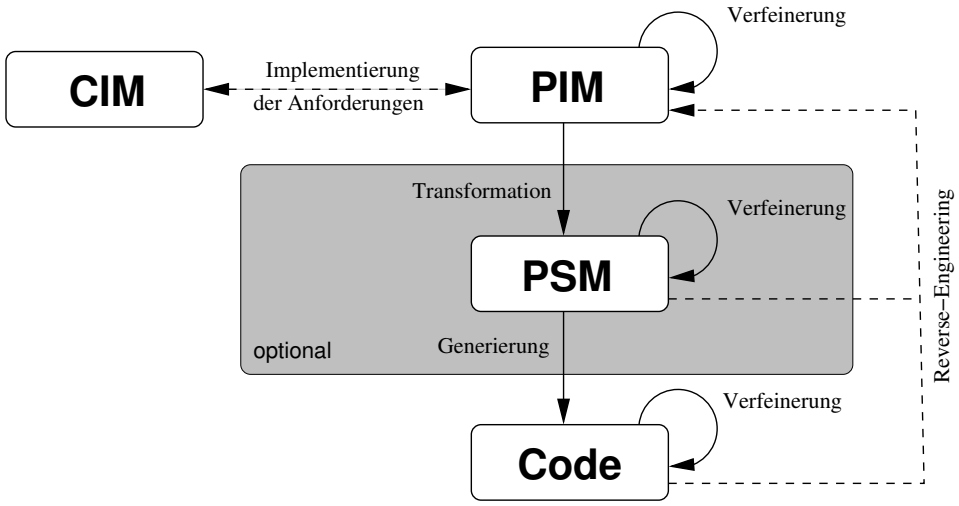
Plattformspezifische Modelle (PSM)

Am Ende des Softwareentwicklungsprozesses stehen *plattformspezifische Modelle* (platform specific model, PSM), die eine vollständige Darstellung des Systems enthalten. Diese Modelle übernehmen alle Eigenschaften der plattformunabhängigen Modelle von denen sie abgeleitet wurden und ergänzen sie um Implementierungsdetails. Zur Darstellung dieser Details werden typischerweise plattformspezifische Erweiterungen der Modellierungssprache verwendet, z.B. entsprechende UML-Profile. Passende UML-Profile sind für viele Plattformen verfügbar, z.B. für CORBA [OMG02c] oder J2EE [JCP01]. Im Beispiel aus Abbildung 2.2 verwendet das PSM ein UML-Profil für J2EE, mit dem das Entwurfsobjekt als Remoteinterface für ein Entity-Bean gekennzeichnet wird. Durch seine Nähe zur Implementierungsplattform kann das PSM in der Regel automatisch in Quellcode der Implementierungssprache übersetzt werden.

2.3.3 Abbildungen und Transformationen

Der Entwicklungsprozess der sich aus der Anwendung der MDA ergibt, besteht fast ausschließlich aus Modelltransformationen, die unterschiedlich stark automatisierbar sind. Abbildung 2.3 enthält einen Überblick über die Abfolge der Transformationsschritte, die nun erläutert werden.

Abbildung 2.3 Überblick über den MDA-Prozess



CIM nach PIM

Die Entwicklung startet nach der Anforderungsanalyse mit einem CIM, das in ein initiales PIM umgesetzt werden muss. Dieser sehr kreative Vorgang kann kaum durch Werkzeuge automatisiert werden. Allerdings kann die Arbeit des Entwicklers durch Entwurfsmuster [GHJV95] erleichtert werden, die bewährte Entwurfslösungen charakterisieren.

PIM nach PIM

Nach dem ein erstes PIM definiert worden ist, muss dieses in der Regel noch mehrmals verfeinert werden, um insbesondere eine vollständige und korrekte Spezifikation der Semantik zu erreichen. Auch diese Abbildungen können manuell unter Benutzung eines Modelleditors vorgenommen werden. Sinnvoller ist allerdings, die manuelle Eingabe von Detailinformationen zu vermeiden und die Modelle über die Anwendung von Transformationen schrittweise zu erweitern. Diese Transformationen können generisch definiert werden und durch einen Konfigurator an das aktuelle Modell angepasst werden. Dieser Konfigurator kann dem Anwender auch die korrekte Anwendungsreihenfolge der Transformationen aufzeigen.

PIM nach PSM

Die Anpassung der plattformunabhängigen Modelle an die speziellen Gegebenheiten einer Technologieplattform spielt eine zentrale Rolle in der MDA-Methode, deshalb sollte diese Transformation weitestgehend zu automatisiert werden. Aber selbst wenn dieser Schritt manuell erledigt werden muss, bietet die MDA-Methode Vorteile:

- Die strikte Trennung der PIM und PSM Modelle bleibt erhalten. Die Modelle sind deshalb einfacher zu verstehen als ein Modell, das beide Aspekte vermischt.
- Die bei der Plattformanpassung getroffenen Entwurfsentscheidungen werden dokumentiert und bleiben nachvollziehbar.

Falls für die angestrebte Zielplattform ein Generator existiert, der die Transformation zwischen PIM und PSM übernehmen kann, muss der Entwickler diesen unter Umständen bei der Interpretation des PIM unterstützen. Dazu fügt er Markierungen in das PIM ein, die in einem UML-Profil definiert sind. Anhand dieser Markierungen erkennt der Generator, welche Abbildung er auf ein Element des PIM anwenden muss und erzeugt ein Element oder auch mehrere im PSM. Komplexere Abbildungen können auch ganze Gruppen von Elementen des PIM in entsprechend komplexe Strukturen im PSM umwandeln.

Falls der verwendete Generator so mächtig ist, dass er ein PSM generieren kann, das keine weiteren Verfeinerungen benötigt, kann auf die plattformspezifische Modellierung sogar ganz verzichtet werden und sofort der Quellcode des Softwaresystems ausgegeben werden. In diesem Fall wird das PSM nur noch zu Dokumentationszwecken erstellt.

PSM nach PSM

Auch die gewonnen PSM-Modelle können bei Bedarf noch weiter verfeinert werden, um fehlende Implementierungsdetails zu ergänzen. Der Umfang der notwendigen Änderungen ist abhängig von der Leistung des verwendeten Generators.

Reverse-Engineering

Es kann auch angebracht sein, den Entwicklungsprozess umzukehren, um zu einem im Quellcode vorhandenen Softwaresystem ein Modell zu erhalten bzw. aus einem PSM das zugehörige PIM zurückzugewinnen. Dieser Vorgang wird als *Reverse-Engineering* bezeichnet und ist wie die Abbildung eines CIM auf ein PIM ein sehr kreativer Prozess, da die nicht mehr vorhandenen Metainformationen rekonstruiert werden müssen. Es sind jedoch Werkzeuge denkbar, die den Quellcode bzw. das PSM heuristisch untersuchen und im Dialog mit dem Entwickler ein Modell auf höherer Abstraktionsebene erzeugen.

2.4 Versionierungssysteme

Wie bereits erwähnt, wird in dieser Arbeit die Domäne der Versionierungssysteme als Beispielanwendung zur Demonstration der Fähigkeiten des entwickelten Generators für Produktlinienmitglieder verwendet. In diesem Abschnitt werden nun die grundlegenden Konzepte dieser Domäne beschrieben. Auf dieser Beschreibung aufbauend, wird dann im nächsten Kapitel ein Domänenmodell für Versionierungssysteme entwickelt.

2.4.1 Zielsetzung

Versionierungssysteme, auch Repositories genannt, basieren auf Datenbanksystemen bzw. Datenverwaltungssystemen anderer Art und stellen Versionierungssemantiken zur Verfügung. Versionierung soll dazu dienen, die Evolution von Datenobjekten zu kontrollieren und zu unterstützen. Zu diesem Zweck wird nicht nur der aktuelle Zustand der Datenstrukturen gespeichert, sondern auch Zwischenzustände, die die Entwicklung dokumentieren. Außerdem ist es möglich, mehrere verschiedene Varianten einer Datenstruktur zu verwalten und parallel weiterzuentwickeln. Versionierungssysteme ermöglichen den Zugriff auf jede der aufgezeichneten Versionen und die Navigation in der Abfolge der Versionen.

2.4.2 Typische Anwendungsgebiete

Versionierungstechniken können überall dort sinnvoll angewendet werden, wo

- Daten, wie z.B. UML-Modelle, Schnittstellendefinitionen oder Textdokumente in mehreren Bearbeitungsschritten über einen längeren Zeitraum erstellt werden,
- häufig auf ältere Versionen der Daten zugegriffen werden muss, bzw. verschiedene Datenversionen parallel verwaltet werden müssen,
- Daten von mehreren kooperierenden Systembenutzern bearbeitet werden.

Diese Bedingungen treffen beispielsweise für die folgenden Anwendungen zu:

- *Engineering Repositories*. Die erste Anwendungsdomäne von Versionierungssystemen war der rechnerunterstützte Entwurf (CAD) in den klassischen Ingenieursdisziplinen [Kat90]. Da bei diesen Entwurfsvorgängen sehr umfangreiche und komplexe Daten anfallen, reichten die Fähigkeiten von einfachen Dateisystemen und herkömmlichen Datenbanken für die Speicherung dieser Daten nicht aus. Deshalb wurden sogenannte Engineering Repositories entwickelt, die sowohl mit komplexeren Datenstrukturen umgehen können, als auch Versionierungssemantiken bieten. Später wurde diese Technologie in anderen Bereichen wiederverwendet, z.B. im Rahmen einer ingenieurmäßigen Softwareentwicklung für die Archivierung von Modellen, Quellcode und sonstigen Dokumenten die während der Entwicklung entstehen.
- *Content-Management-Systeme*. Eine neuere Entwicklung sind Redaktionssysteme, sogenannte Content-Management-Systeme (CMS), die beispielsweise die Erstellung und Pflege komplexer Internetinhalte durch ein großes Redaktionsteam ermöglichen. Hierbei werden die klassischen Versionierungssemantiken angepasst und erweitert, um den typischen Arbeitsablauf einer Redaktion zu unterstützen. Eine Anforderung ist es etwa, zum Zweck der Qualitätssicherung sicherzustellen, dass keine Inhalte fahrlässig ohne eine Abnahme durch einen zweiten Redakteur veröffentlicht werden können. Die Versionierungsseman-

tik kann dies sicherstellen, indem in das Lebenszyklusmodell der versionierten Objekte das Vier-Augen-Prinzip bzw. die Abnahme durch den Chefredakteur integriert wird.

- *Grid Computing.* Ein *Computational Grid* [FK98] ist ein verlässliche, offene, flexible und dezentral organisierte Hard- und Softwareinfrastruktur, die den Benutzern Speicher- und Rechenkapazitäten auf den teilnehmenden Rechnern transparent und nach Bedarf zur Verfügung stellt. Systeme dieses Typs werden beispielsweise benutzt um Hochleistungsrechner verschiedener Organisationen miteinander zu vernetzen und auf diese Weise die Rechenkapazitäten effizienter zu nutzen. Typischerweise werden mit diesen vernetzen Systemen wissenschaftliche Analyse- und Simulationsberechnungen durchgeführt. Da im Laufe der Zeit die Ergebnisse der Berechnungen optimieren werden sollen, bzw. sich die Voraus- und Zielsetzungen der Berechnungen ändern, ändern sich auch die verwendeten Daten und Algorithmen ständig. Deshalb ist ein Bedarf für Versionierungsdienste innerhalb des Computational Grids gegeben [KH03]. Mit Hilfe dieser Dienste kann jederzeit auf alte Versionen der Daten, Ergebnisse und Algorithmen zurückgegriffen werden, und es kann insbesondere genau nachvollzogen werden, welche Kombination bei vergangenen Berechnungen verwendet wurde.

2.4.3 Objektorientierte Versionierungssysteme

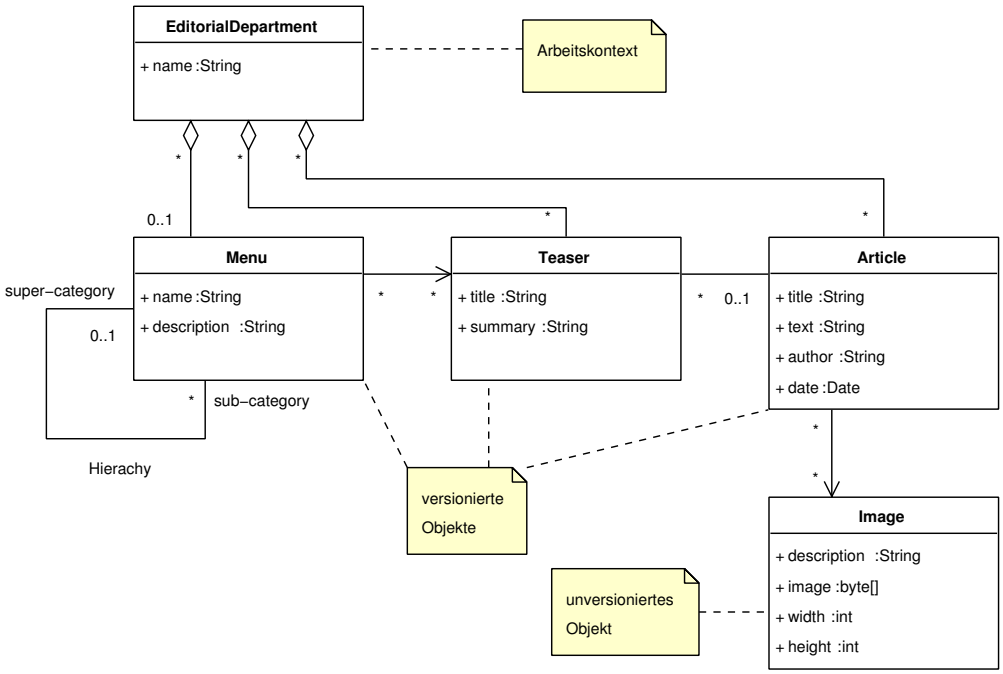
Im folgenden werden die wichtigsten Eigenschaften *objektorientierter Versionierungssysteme* vorgestellt, wie sie von Bernstein [Ber98] beschrieben worden sind. Objektorientierte Versionierungssysteme zeichnen sich durch ein *Informationsmodell* aus, dass speziell auf die zu speichernden Daten zugeschnitten ist. Ein solches Modell besteht im Wesentlichen aus einer beliebigen Anzahl von versionierten bzw. unversionierten *Objekttypen* mit ihren *Attributen* und *Beziehungen* zu anderen Objekttypen.

Ein Objekt ist die kleinste einzeln versionierte Informationseinheit in einem Versionierungssystem. Hierbei ist zu beachten, dass im Gegensatz zu objektorientierten Datenbanken oder Programmiersprachen, ein Objekt auch eine Aggregation komplexer Datenstrukturen sein kann. Die aggregierten Datenobjekte haben aus der Sicht des Versionierungssystems keine eigenständige Identität.

Wie bereits erwähnt, können Versionierungssysteme abweichend vom Normalfall auch unversionierte Objekttypen enthalten. Diese Objekttypen enthalten dann keine versionierungsspezifischen Attribute und es können keine Methoden der Versionsverwaltung auf sie angewandt werden. Diese Option ist eine Optimierung für den Fall, dass bestimmte zusätzliche Daten im Versionierungssystem gespeichert werden sollen, die in einer engen Beziehung mit den eigentlichen versionierten Daten stehen, selbst aber nach ihrer erstmaligen Erstellung nicht verändert werden, bzw. alte Versionen dieser Daten nicht benötigt werden.

Abbildung 2.4 zeigt ein Informationsmodell, das Grundlage für ein sehr einfaches Content-Management-System sein könnte und im folgenden als laufendes Beispiel dienen soll. Ein Versionierungssystem mit diesem Informationsmodell kann die Arbeitsergebnisse einer Nachrichtenredaktion speichern. Die gespeicherten Daten werden dann in einem Webserver zur Darstellung einer Nachrichtenseite verwendet. Der wichtigste Objekttyp ist `Article`, der die Inhalte eines Nachrichtenartikels speichert. Da der Text des Artikels vor seiner Veröffentlichung typischerweise mehrmals korrigiert wird, ist dieser Objekttyp versionierbar. Ein Artikel kann auch ein Foto enthalten, das in einem Objekt vom Typ `Image` gespeichert wird. Um Speicherplatz in der zugrundeliegenden Datenbank zu speichern, wurde für

Abbildung 2.4 Informationsmodell eines Versionierungssystems



diesen Objekttyp auf eine Versionierung verzichtet. Die Struktur der Nachrichtenseite wird durch `Menu`-Objekte bestimmt, die jeweils eine Übersichtsseite repräsentieren. Diese enthalten mehrere sogenannte `Teaser`, die eine kurze Zusammenfassung einer Nachricht und einen Verweis auf den ausführlichen Artikel enthalten. Um eine Navigation innerhalb der Nachrichtenseiten zu ermöglichen, können in ein `Menu`-Objekt andere Menüs als Unterkategorien eingetragen werden.

Objekte haben neben den im Informationsmodell vorgegeben Attributen noch weitere systembedingte Attribute, die zur Verwaltung notwendig sind. Das wichtigste Attribut ist der Primärschlüssel, der jedes Objekt eindeutig identifiziert. Dieser Schlüssel wird aus mehreren Attributen gebildet:

- `objectId` - Diese Nummer identifiziert ein Objekt eindeutig. Allerdings kann über diese Nummer alleine nicht zwischen den verschiedenen Versionen eines Objekts unterschieden werden.
- `versionId` - Diese Nummer identifiziert die konkrete Version eines versionierten Objekts. Unversionierte Objekttypen besitzen dieses Attribut nicht.
- `globalId` - Die `globalId` ist eine Kombination der beiden obenstehenden Nummern. Sie ist dadurch für jede Version jeder Objektinstanz eindeutig und kann beispielsweise als Primärschlüssel im dem Datenbankschema dienen, das zur Speicherung der versionierten Daten benutzt wird.

Durch diese Konstruktion haben nicht nur Objekte eine eigenständige Identität, sondern auch die unterschiedlichen Versionen werden als eigenständige Objekte behandelt. Diese Dualität ist ein weiterer Unterschied eines objektorientierten Versionierungssystems zu anderen objektorientierten Systemen.

Das Versionierungssystem bietet eine Schnittstelle (API) an, über die nach gespeicherten Objekten gesucht werden kann und die Daten eines gefundenen Objekts ausgelesen und verändert werden können. Im Falles eines objektorientierten Versionierungssystems ist diese Schnittstelle idealerweise ebenfalls objektorientiert, z.B. durch eine Implementierung die auf J2EE [Sun01] basiert. Falls eine prozedurale Schnittstelle gewählt wird, muss bei allen Funktionen die `globalId` als zusätzlicher Parameter mitgeführt werden, um eine Identifizierung des aktuellen Objekts zu ermöglichen.

Natürlich sind auch andere Arten von Informationsmodellen denkbar. So bieten einfachere Versionierungssysteme wie sie z.B. Rational ClearCase [IBM03a] oder CVS [CVS03], die im Bereich des Software Engineering eingesetzt werden, gar kein anwendungsspezifisches Informationsmodell, sondern versionieren beliebige Dateien innerhalb einer Dateisystemstruktur. Dieser Ansatz ist aber nur für sehr einfach strukturierte Daten geeignet und wird deshalb hier nicht weiter behandelt.

2.4.4 Versionsverwaltung

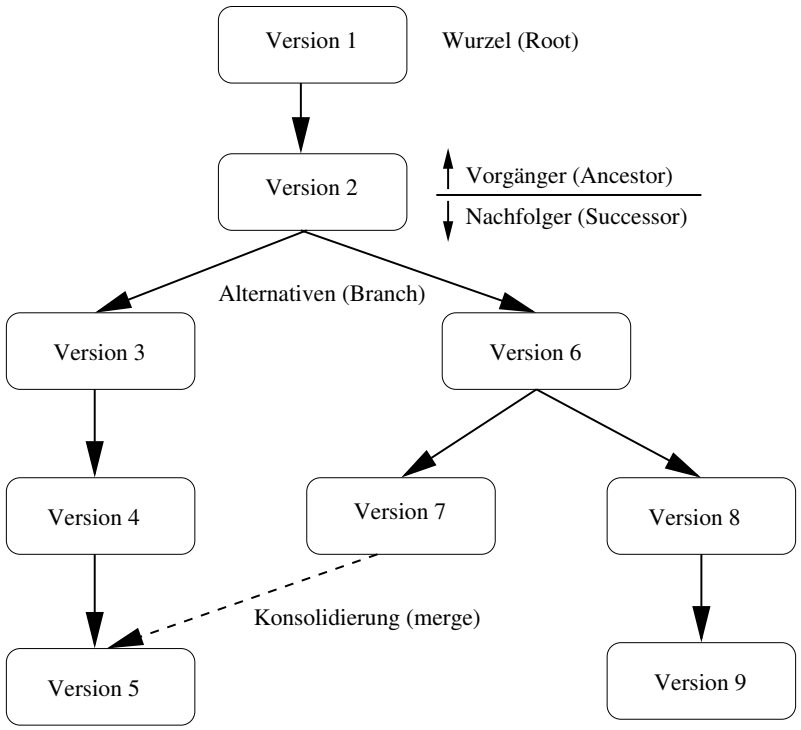
Eine *Version* ist eine semantisch bedeutungsvolle Momentaufnahme des Zustands eines durch einen Versionierungssystem verwalteten Objekts [Kat90]. Typischerweise wird der Zeitpunkt der Momentaufnahme durch den Anwender des Systems vorgegeben, die Version hat damit also eine definierte Bedeutung, sie ist beispielsweise das Ergebnis eines in sich abgeschlossenen Arbeitsschritts. Der Anwender hat auch die Möglichkeit, diesen Arbeitsschritt und die vorgenommenen Änderungen informell zu beschreiben, indem er die Version sinnvoll benennt bzw. eine längere textuelle Beschreibung anfügt.

Alternativ ist auch denkbar, dass implizit durch jede Änderung des versionierten Objekts eine neue Version erzeugt wird. Dieses Vorgehen führt allerdings zu einer unüberschaubar hohen Anzahl an Versionen und die vorgenommenen Änderungen können im Nachhinein nur noch schwer nachvollzogen werden.

Die initiale Version eines Objekts wird gleichzeitig mit der Erzeugung des Objekts angelegt. Alle Nachfolgeversionen werden mit Hilfe des Versionierungsystems als Nachfolger einer bestehenden Version definiert. Dadurch entsteht eine „stammt-ab-von“ Beziehung (is-derived-from) zwischen *Vorgänger* (Ancestor) und *Nachfolger* (Successor). Diese Abstammung wird als Eigenschaft der neu erstellten Version gespeichert. Das Abstammungsverhältnis der Versionen ergibt den *Versionsgraph* des Objekts. Ein solcher Graph ist in Abbildung 2.5 dargestellt. In diesem Versionsgraph ist Version „1“ die *Wurzel* (Root) des Versionsgraphen. Sie wurde beim Erzeugen einer neuen Objektinstanz, das durch einen Aufruf der Methode `createObject` erfolgt, automatisch erzeugt.

Ein Nachfolger wird mit Hilfe der Methode `createSuccessor` erstellt. Dabei werden die Attributwerte der Vorgängerversion in der Nachfolgeversion übernommen und können anschließend verändert werden. Die Version „2“ hat zwei Nachfolger und zwar die Versionen „3“ und „6“, die durch wiederholte Aufrufe von *createSuccessor* entstanden sind. Die beiden Nachfolger werden *Alternativen* genannt, da sie alternative Implementierungen des selben Objekts darstellen können, die z.B. auf unterschiedliche Randbedingungen hin optimiert worden sind. Häufiger entstehen Alternativen allerdings durch nebenläufige Entwicklungsstränge, etwa durch Wartung und gleichzeitige Weiterentwicklung von Softwaresystemen. Durch diese konkurrierenden Weiterentwicklungen entstehen *Zweige* (Branches) im Versionsgraph.

Abbildung 2.5 Versionsgraph



Manche Versionierungssysteme ermöglichen eine *Konsolidierung* des Versionen durch die Zusammenführung (merge) verschiedener Zweige des Versionsgraphen. Dies ist in Abbildung 2.5 bei der Version „5“ geschehen, dort wurde nach der Erzeugung der Version „5“ als Nachfolger der Version „4“, die Methode `merge` benutzt, um die Version „7“ mit der Version „5“ zusammenzuführen. Dadurch hat die Version „5“ zwei Vorgänger, nämlich „4“ und „7“. Neben der formalen Vereinigung der Zweige durch die Veränderung der Abstammungsverhältnisse müssen natürlich auch die Objektinhalte abgeglichen werden. Wie dies zu geschehen hat, hängt von der Semantik des Objekttyps ab, ist also abhängig von der Anwendungsdomäne. Unter Umständen ist es möglich, diesen Vorgang zu automatisieren, deshalb sollte das Versionierungssystem eine Möglichkeit bieten, das Verhalten der `merge`-Methode anzupassen. Falls diese Vorrausetzungen nicht gegeben sind, muss der Anwender die notwendigen Anpassungen manuell vornehmen.

Um unkompliziert auf zurückliegende Objektversionen zugreifen zu können, muss der Benutzer eines Versionierungssystems im Versionsgraph navigieren können. Über die gespeicherten Abstammungsverhältnisse ist es einfach möglich, entsprechende Methoden zu implementieren:

- `getAllSuccessors` liefert die direkten, durch die `createSuccessor`-Methode angelegten Nachfolger der aktuellen Version.
- `getAllSuccessors` liefert alle Nachfolger der aktuellen Version. Dies schließt auch alle Nachfolger ein, die nicht direkt über die `createSuccessor`-Methode erstellt worden sind, sondern mittels der `merge`-Methode zugeordnet worden sind.
- `getAncestor` liefert den Vorgänger der aktuellen Version.

- `getAllAncestors` liefert alle Vorgänger der aktuellen Version. In dieser Menge ist analog zu `getAllSuccessors` nicht nur die ursprüngliche Vorgängerversion enthalten, sondern auch alle durch Konsolidierungsvorgänge hinzugekommenen Vorgänger.
- `getAlternatives` liefert alle Alternativen zur aktuellen Version, also alle Versionen, die vom gleichen Vorgänger abgeleitet wurden.
- `getRoot` liefert die Wurzel des Versionsgraphen.

Versionierte Objekte besitzen neben ihrer `versionId` noch weitere Attribute, die den Zustand der Version beschreiben:

- Ein gebräuchliches Attribut ist eine textuelle Beschreibung der Version, bzw. der Arbeitsschritte die zu dieser Version geführt haben. Beispielsweise ermöglicht RCS/CVS [Tic85] die Eingabe einer Beschreibung bei der Erstellung einer neuen Version (commit).
- Weiterhin ist es möglich, verschiedene Zeitpunkte zu einer Version aufzuzeichnen, etwa das Datum der Versionserstellung oder das Datum der letzten Änderung.
- Da die Versionierung den Entwicklungsprozess eines Objekts dokumentieren soll, ist es sinnvoll, dass zurückliegende Versionen ab einem bestimmten Zeitpunkt nicht mehr verändert werden können. Zu diesem Zweck verwalten viele Versionierungssysteme ein Attribut `frozen`. Sobald dieses mittels der `freeze` Operation gesetzt wurde, verhindert das Versionierungssystem alle weiteren Schreibzugriffe auf die betroffene Objektversion. Die `freeze` Operation kann sowohl explizit durch den Anwender aufgerufen werden, als auch implizit bei der Durchführung anderer Operationen, wie z.B. `createSuccessor`.
- Bedingt durch das oben genannte Attribut `frozen` gibt es neben dem initialen Zustand noch den Zustand „frozen“ im *Lebenszyklus* einer Objektversion. Für viele Anwendungsfälle wäre es aber sinnvoll, die Zustandsmaschine die hinter diesem Lebenszyklusmodell steht, um weitere Zustände zu erweitern. Beispielsweise könnte das Lebenszyklusmodell zum Informationsmodell aus Abbildung 2.4 die Zustände „korrigiert“ und „freigegeben“ enthalten, die kennzeichnen, dass ein Artikel Korrektur gelesen wurde bzw. eine Freigabe für die Veröffentlichung erteilt wurde. Um die zusätzlichen Zustände zu kennzeichnen, müssen entsprechenende Attribute in die Objekttypen eingefügt werden.

2.4.5 Beziehungen zwischen Objekten

Wie bereits erwähnt, können Informationsmodelle von Versionierungssystemen *Beziehungstypen* enthalten, die Objekttypen miteinander verbinden. Die Informationen über bestehende Beziehungen werden wie die Attributdaten durch das Versionierungssystem verwaltet und über Methoden dem Benutzer zugängig gemacht. Die Navigation über eine Beziehung kann dabei entweder *direkt* oder *indirekt* ablaufen. Bei einer direkten Navigation kann der Benutzer über den Aufruf von entsprechenden Zugriffsmethoden direkt vom Ausgangsobjekt zum Zielobjekt gelangen. Im indirekten Fall erhält der Benutzer zunächst Zugriff auf ein Beziehungsobjekt, von dem er dann auf die *Beziehungsenden* und damit auf das Zielobjekt zugreifen kann. Manche Versionierungssysteme unterstützen Informationsmodell die den Beziehungen eigene Attribute zuordnen. Auf die Attributwerte kann dann der Benutzer mit Hilfe der indirekten Navigation zugreifen.

Weitere Spezifikationsmöglichkeiten beziehen sich auf die *Kardinalität* der Beziehungsenden und der erlaubten *Navigationsrichtungen*. Beziehungsenden können ein einziges oder eine Menge von Objekten referenzieren. Versionierungssysteme können 1:1, 1:n und n:m

Beziehungen verwalten. Falls für eine Beziehung das Informationsmodell die Navigation nur für eine Richtung vorsieht, muss das Versionierungssystem die benötigten Informationen und Zugriffsmethoden nur beim Ausgangsobjekt der Beziehung anbieten, dadurch kann die Datenverwaltung vereinfacht werden.

Beziehungen zwischen versionierten Objekten

Das Zielobjekt einer Beziehung zu bestimmen ist trivial, sofern der Zielobjekttyp nicht versioniert ist. Sobald er aber versioniert ist, stellt sich die semantisch wichtige Frage, welche Objektversion referenziert werden soll. Die einfachste Lösung ist, die Version entweder genau zu fixieren, dann führt die Navigation immer zu dieser Version, auch wenn in der Zwischenzeit neue Versionen angelegt wurden, oder aber die Version gar nicht zu spezifizieren, dann muss der Benutzer selbst entscheiden, welche Version er aus dem Versionsgraph des Zielobjekts verwenden möchte. Diese Minimallösung kann sehr effizient implementiert werden, ist aber semantisch nicht sehr ausdrucksstark. Beispielsweise wäre es wünschenswert, wenn die in Abbildung 2.4 enthalten Beziehung zwischen `Teaser` und `Article` immer die letzte freigegebene Version des Artikels referenzieren würde. Deshalb bieten manche Versionierungssysteme, z.B. das Microsoft Repository [BBC+99], Beziehungstypen, für deren Beziehungsenden der Benutzer detailliert spezifizieren kann, welche Objektversionen referenziert werden dürfen. Diese Beziehungsenden werden auch als *gleitende Beziehungsenden* (floating relationship ends) bezeichnet. Die zugelassenen Objektversionen werden durch den Benutzer in die *Kandidatenmenge* (candidate version collection) eingefügt. Dabei ist zu beachten, dass die Kardinalität des Beziehungsendes nicht davon beeinflusst wird, ob es gleitend ist oder nicht. Ein einwertiges gleitendes Beziehungsende besitzt nur eine einzige Kandidatenmenge, mehrwertige Enden hingegen jeweils eine Kandidatenmenge für jedes referenzierte Objekt. Die *Resolution*, die bestimmt welche Version aus der Kandidatenmenge tatsächlich verwendet wird, erfolgt wenn eine Zielobjekt durch den Benutzer erfragt wird. Für die Resolution gibt es mehrere Verfahren, zwischen denen der Benutzer auswählen kann indem er die zugeordnete Zugriffsmethode auswählt, bzw. entsprechende Parameter übergibt:

- *Regelbasierte Auswahl* - Das Versionierungssystem kann die Auswahl anhand von Regeln vornehmen, die durch den Hersteller des Versionierungssystems vorgegeben worden sind oder im Informationsmodell definiert werden. Ein typisches Beispiel ist eine Regel, die immer die neuste Version eines Objekts auswählt.

- *Explizite Festlegung (Pinning)* - Der Benutzer kann auch eine Voreinstellung (Defaultwert) für die zu verwendende Objektversion festlegen, die bei einer Anfrage ohne weitere Angaben verwendet wird. Die Festlegung der Voreinstellung wird als *Pinning* bezeichnet, die Aufhebung der selben als *Unpinning* [BBC+99].

- *Keine Vorauswahl* - Der Benutzer hat natürlich auch die Möglichkeit, die gesamte Kandidatenmenge vom Versionierungssystem anzufordern und dann selbst eine Auswahl zu treffen.

Propagierung von Zustandsänderungen

In bestimmten Fällen kann es sinnvoll sein, dass die Anwendung von Operationen wie z.B. `freeze` oder `createSuccessor` nicht nur Auswirkungen auf das Objekt hat, auf das die Operation eigentlich angewendet wurde, sondern auch auf Objekte die über Beziehungen mit diesem Objekt verbunden sind. Im Falle des Informationsmodells aus Abbildung 2.4

könnte beispielsweise beim Anlegen einer neuen Version eines Artikels automatisch eine neue Version des zugehörigen Teasers angelegt werden, da dieser ja voraussichtlich ebenfalls geändert werden muss. Welche Operationen über eine Beziehungen propagiert werden, wird im Informationsmodell spezifiziert.

2.4.6 Arbeitskontexte

Ein *Arbeitskontext* (Workspace), auch *Konfiguration* genannt, ist ein versioniertes Objekt, das andere Objekte aggregiert, die beispielsweise zum Zuständigkeitsbereich eines bestimmten Benutzers gehören. Da immer nur eine einzige Version eines Objekts in einen Arbeitskontext aufgenommen werden kann, stellt er eine Verbindung zwischen seiner eigenen Version und jeweils einer Versionen der enthaltenen Objekte her [Kat90].

Zwischen dem Arbeitskontext und den enthaltenen Objektversionen besteht eine *Attachment*-Beziehung. Das Versionierungssystem stellt die die Methoden `attach` und `detach` zur Verfügung, mit denen Versionen in den Arbeitskontext eingefügt bzw. entfernt werden können.

Welche Objekttypen in einem Arbeitskontext aufgenommen werden können, wird im Informationsmodell spezifiziert. Das Beispiel aus Abbildung 2.4 enthält auch einen Arbeitskontext, nämlich `EditorialDepartment`. Die Instanzen dieses Typs enthalten jeweils die Arbeitsergebnisse einer Redaktion, also die Menüseiten, Teaser und Artikel für die sie verantwortlich ist.

Im Informationsmodell wird auch spezifiziert, ob die Attachment-Beziehung exklusiv ist, also ob Objekte gleichzeitig in mehreren Arbeitskontexten enthalten sein dürfen, oder nicht. Da Arbeitskontexte reguläre Objekte sind, können sie versioniert werden und auch ihrerseits in andere Arbeitskontext aufgenommen werden.

Die Tatsache, das zu jedem Objekt höchstens eine Version in einem Arbeitskontext enthalten ist, kann genutzt werden, um einen versionsfreien Zugriff auf die Objekte zu ermöglichen. Solange der Benutzer sich in einem Arbeitskontext befindet, sieht er bei einer Suche nur die eingefügten Objektversionen und gelangt bei der Verfolgung einer Beziehung automatisch zu einer Version des Zielobjekts die ebenfalls im gleichen Arbeitskontext liegt. Dabei werden die sonstigen Auswahlregeln, bzw. die explizite Festlegung (Pinning) einer Version nicht beachtet. Aus der Sicht des Benutzers gibt es also innerhalb des Arbeitskontextes keinen Unterschied zwischen versionierten und unversionierten Objekten.

2.4.7 Transaktionsunterstützung

Da Versionierungsdienste in aller Regel auf Datenbanken aufbauen, können sie sich für alle durch den Benutzer veranlassten Operationen der durch Datenbanksysteme gebotenen *Transaktionsunterstützung* bedienen. Diese Transaktion haben die *ACID-Eigenschaften*. Dadurch kann unter anderem kontrolliert werden, zu welchem Zeitpunkt die in der Transaktion vorgenommen Änderungen für andere Anwender sichtbar werden. Ausserdem wird sichergestellt, dass die versionierten Objekte nach Beendigung der Transaktion die Integritätsbedingungen des Informationsmodells erfüllen.

Es sollte aber nicht nur einen Transaktionsschutz für elementare Operationen geben, sondern auch für länger andauernde Arbeitsvorgänge des Benutzers, die aus vielen elementaren Ope-

rationen bestehen. Bei einem solchen Arbeitsvorgang ist es beispielsweise nicht erwünscht, dass ein andere Benutzer die vorgenommen Änderungen unwissentlich überschreibt. Ein solcher Arbeitsvorgang könnte z.B. die Eingabe eines neuen Artikels durch einen Redakteur sein.

Herkömmliche ACID-Transaktionen können nicht verwendet werden, da die Transaktion für die den gesamten Arbeitsvorgang offen gehalten werden muss. Es würden daher auch die notwendigen Datenbanksperren sehr lange aufrecht erhalten, dies ist in einem Mehrbenutzersystem nicht akzeptabel. Versionierungssysteme bieten deshalb einen bewusst etwas schwächeren Schutz für langanhaltende Arbeitsvorgänge. Um diesen Schutz zu aktivieren, wird die Methode `checkout` auf den relevanten Objektversionen aufgerufen. Die so gesperrte Objektversion kann nun nur noch von dem Benutzer manipuliert werden, der die Sperre aktiviert hat. 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. Ist der Arbeitsvorgang beendet, hebt der Benutzer seine Sperren mit Hilfe der Methode `checkin` wieder auf.

Kapitel 3

Definition von Merkmalmodell und Spezifikationssprache

Nachdem im letzten Kapitel die einzelnen Phasen der Produktlinienentwicklung vorgestellt wurden, wird nun die Domänenanalyse für die Domäne der Versionierungssysteme durchgeführt. Ziel ist es, zunächst ein Merkmalmodell der Domäne zu erstellen. In einem Merkmalmodell werden die Merkmale, also 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.

Aus diesem Merkmalmodell wird eine Spezifikationssprache entwickelt, mit deren Hilfe die Konfiguration von Versionierungssystemen, die Mitglieder der Produktlinie sind, festgelegt werden kann. Für die Spezifikation domänenspezifischer Aspekte der Konfiguration wird der Sprachumfang von UML mittels eines UML-Profils erweitert, um auf diese Weise eine domänenspezifische Spezifikationssprache für Versionierungssysteme zu erhalten.

3.1 Merkmalanalyse

Die *Merkmalanalyse* (feature analysis) wurde ursprünglich durch die Feature-Oriented Domain Analysis (FODA) Methode [KCH+90] eingeführt. Die FODA-Methode besteht aus zwei Phasen: Während der *Kontextanalyse* (context analysis) werden die Grenzen der zu analysierenden Domäne festgelegt, es wird also die Domänenabgrenzung (scoping) durchgeführt. Anschließend wird mit verschiedenen Verfahren ein Domänenmodell erstellt, also die Domänenanalyse durchgeführt, die aus dem idealisierten Vorgehensmodell bekannt ist, das im letzten Kapitel vorgestellt wurde. Zu diesen Verfahren gehören, neben der hier angewendeten Merkmalanalyse, auch eine Analyse des vorhandenen Domänenwissens (information analysis), das beispielsweise aus für die Domäne relevanten wissenschaftlichen Erkenntnissen und Theorien und Erfahrungen mit der Entwicklung und dem Betrieb von Vorgängersystem bestehen kann. Dieses Domänenwissen wird in Entity-Relationship- bzw. objektorientierten Modellen festgehalten. Weiterhin wird das erwünschte Verhalten von in der Domäne enthalten Systemen untersucht (operational analysis). Dabei werden u.A. die Gemeinsamkeiten und Unterschiede in den Daten- und Kontrollflüssen dieser Systeme dokumentiert.

Da die wichtigsten Begriffe und die grundlegenden Konzepte der Domäne der Versionierungssysteme durch die Einführung bereits bekannt sind, wird an dieser Stelle auf die Durchführung der Domänenabgrenzung verzichtet und auch aus der Domänenanalysephase der FODA-Methode wird mit der Merkmalanalyse nur ein einzelner Baustein herausgegriffen.

Ein *Merkmal* (feature) eines Softwaresystems kann verschiedenartig definiert werden. Einerseits als eine Eigenschaft eines Systems, die für den Endbenutzer sichtbar ist [KCH+90], andererseits als eine unterscheidbare Eigenschaft einer Systemkomponente, die für einen beliebigen Interessenten von Bedeutung ist [CE00]. Im Folgenden werden wir die zweite, weiter gefasste Definition verwenden, da wir auch Unterschiede in der Implementierung von Softwaresystemen untersuchen wollen, die nicht direkt für den Endbenutzer einsehbar sind.

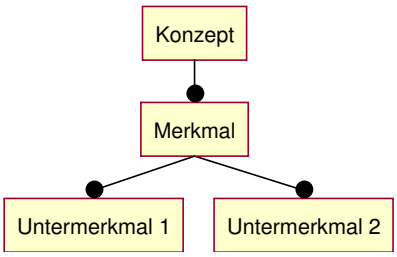
Wie bereits angedeutet, kategorisiert ein *Merkmalmodell* (feature model) die Merkmale eines Softwaresystems, bzw. einer Produktlinie. Es besteht aus folgenden Elementen [CE00]:

- *Merkmaldiagramme,* die eine hierarchische Aufschlüsselung der Merkmale enthalten und gleichzeitig dokumentieren, welche Merkmale feste bzw. optionale Bestandteile des Systems sind und welche Merkmale Alternativen bilden.
- *Kompositionsregeln,* die Abhängigkeiten zwischen den Merkmalen dokumentieren, die innerhalb der Hierarchie des Merkmaldiagramms nicht dargestellt werden können. Dadurch wird geklärt, welche Merkmalkombinationen zulässig sind und welche nicht.
- *Merkmaldefinitionen,* die die Merkmale in informeller Art und Weise näher beschreiben, beispielsweise durch Verweise auf die entsprechenden Anforderungsdokumente oder durch die Beschreibung der Semantik einer Funktionalität.
- *Auswahlrichtlinien,* die bei der Auswahl von optionalen bzw. alternativen Merkmalen helfen, indem sie aufzeigen, unter welchen Umständen ein Merkmal ausgewählt werden sollte.

3.1.1 Merkmaldiagramme

Czarnecki [CE00] definiert eine Notation für Merkmaldiagramme, die ausdrucksstärker ist als die ursprünglich durch die FODA-Methode eingeführte Notation. Diese erweiterte Notation wird in den weiteren Darstellungen verwendet und hier kurz erklärt.

Abbildung 3.1 Ein Merkmaldiagramm mit drei Merkmalen



Ein Merkmaldiagramm besteht aus Knoten und Kanten, die einen Baum bilden. Das Konzept, das durch das Diagramm beschrieben werden soll, wird durch den Wurzelknoten des Baums symbolisiert. Die Unterknoten der Wurzel bilden die Merkmale, die das Konzept charakterisieren. Um die Eigenschaften des Konzepts detaillierter darzustellen, können die

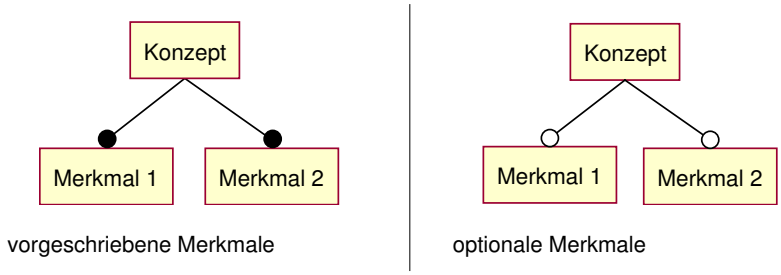
Merkmale ihrerseits beliebig viele Ebenen mit Untermerkmalen besitzen. Abbildung 3.1 zeigt ein Beispieldiagramm mit insgesamt drei Merkmalen. Dabei besitzt das beschriebene Konzept ein direktes Merkmal, das in zwei Untermerkmale aufgegliedert wird, die also indirekte Merkmale des Konzepts sind.

Merkmaldiagramme beschreiben ein Konzept nicht im Bezug auf ein konkretes Softwaresystem, sondern im Kontext einer Klasse von Systemen, typischerweise einer Produktlinie. Die Merkmale eines einzelnen System ergeben sich durch eine Auswahl aus der Gesamtheit der im Baum aufgelisteten Merkmale. Welche Auswahlen zulässig sind, ist vom Typ der (Unter-)Merkmale und von den zusätzlich definierten Kompositionsregeln abhängig.

Die Notation der Merkmaldiagramme unterscheidet, insbesondere durch die Gestalt der Verbindungskanten, zwischen verschiedenen Merkmaltypen, die in den folgenden Absätzen vorgestellt werden:

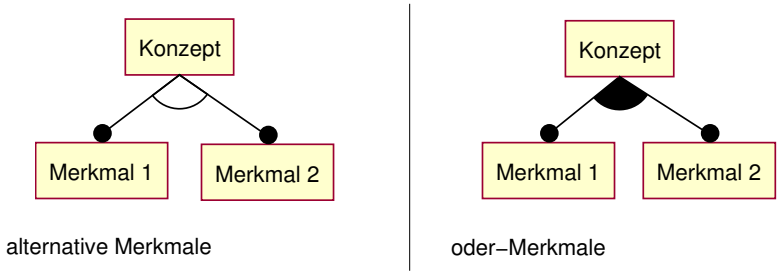
- Ein *vorgeschriebenes Merkmal* (mandatory feature) wird durch einen ausgefüllten Kreis an der eingehenden Kante gekennzeichnet, die es mit seinem übergeordneten Merkmal bzw. Konzept verbindet. Diese Kennzeichnung ist in Abbildung 3.2 zu erkennen, wo die beiden Merkmale 1 und 2 auf der linken Seite als vorgeschriebene Merkmale gekennzeichnet wurden. Die Semantik eines vorgeschriebenen Merkmals ist, dass es in eine Instanz des beschriebenen Konzepts aufgenommen werden muss, sofern das direkt übergeordnete Merkmal aufgenommen wurde. Ein solches Merkmal steht also für eine unveränderbare Eigenschaft einer Klasse von Systemen, bzw. im Fall eines Untermerkmals für eine unveränderbaren Eigenschaft des übergeordneten Merkmals.

Abbildung 3.2 Vorgeschriebene und optionale Merkmale



- Das Gegenstück zu den vorgeschriebenen Merkmalen bilden die *optionalen Merkmale*, die alle die Eigenschaften beschreiben, die nicht alle Instanzen eines Konzepts besitzen müssen, sondern nach Bedarf ausgewählt werden können. Wie im rechten Diagramm in Abbildung 3.2 zu sehen ist, werden optionale Merkmale durch einen nicht ausgefüllten Kreis gekennzeichnet. Wenn bei der Instantiierung eines Konzepts ein Merkmal ausgewählt wird, das optionale Untermerkmale besitzt, kann frei entschieden werden, welche der optionalen Merkmale mit aufgenommen werden. In Abbildung 3.2 sind also vier Varianten denkbar: Entweder wird keins der beiden Merkmale ausgewählt, oder aber beide Merkmale gemeinsam, oder auch nur ein einzelnes Merkmal. Bei der Auswahl müssen natürlich die außerhalb des Merkmaldiagramms durch die Kompositionsregeln definierten Abhängigkeiten zwischen den Merkmalen beachtet werden.
- Die wichtigsten Abhängigkeiten zwischen den Merkmalen können und sollten allerdings auch direkt im Merkmaldiagramm enthalten sein. Dazu dienen die in Abbildung 3.3

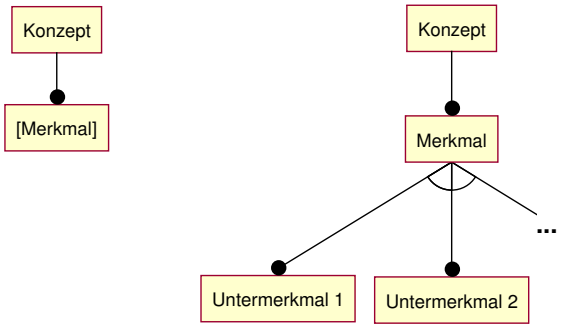
Abbildung 3.3 Darstellung von Varianten



gezeigten Notationen zur Darstellung von Varianten. Eine Möglichkeit besteht darin, Merkmale, die sich gegenseitig ausschließen, in einer Gruppe zusammenzufassen. Solche Merkmale werden *alternative Merkmale* genannt und durch einen Bogen zwischen ihren eingehenden Kanten gekennzeichnet. Aus einer Gruppe von alternativen Merkmalen muss bei der Instantiierung des Konzepts ein einziges Merkmal ausgewählt werden.

- Ein ausgefüllter Bogen zwischen den eingehenden Kanten einer Gruppe von Merkmalen kennzeichnet *oder-Merkmale*. Aus einer solchen Gruppe muss *mindestens* ein Merkmal ausgewählt, es werden also ähnliche Merkmale zusammengefasst, die beliebig kombiniert werden können.

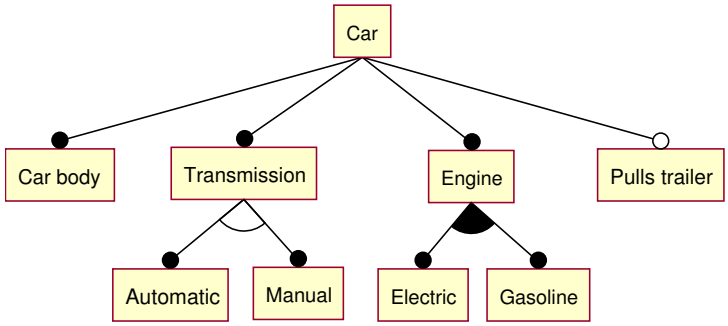
Abbildung 3.4 Darstellung von Erweiterungspunkten



- Eine weitere Variationsmöglichkeit, die mit Merkmaldiagrammen dargestellt werden kann, sind *Erweiterungspunkte*, also Merkmale, deren Semantik nicht vordefiniert ist, sondern während der Erstellung eines neuen Systems ergänzt werden kann bzw. muss. Wie aus Abbildung 3.4 ersichtlich ist, gibt es zwei Möglichkeiten, Erweiterungspunkte zu definieren. Einerseits kann ein Merkmal durch eckige Klammern als *freier Parameter* gekennzeichnet werden. Anderseits können mit dem Auslassungszeichen „...“ Bereiche des Diagramms gekennzeichnet werden, in denen nach Belieben neue Merkmale eingefügt werden können. Die Merkmaldefinition freier Parameter spezifiziert typischerweise, durch welche Elemente der Platzhalter ersetzt werden muss. Demgegenüber gelten bei der zweiten Variante von Erweiterungspunkten keine wirklichen Einschränkungen für die Erweiterungen.

Abbildung 3.5 enthält ein Beispieldiagramm, das die meisten der vorgestellten Notationselemente enthält. Das Diagramm beschreibt die Merkmale eines Autos. Alle Autos haben

Abbildung 3.5 Beispiel eines Merkmaldiagramms (entnommen aus [CE00])



eine Karosserie, einen Motor und ein Getriebe. Die Variationsmöglichkeiten sind, dass es manuelle und automatische Getriebe gibt, manche Autos Anhängerkupplungen besitzen und außerdem verschiedene Antriebskonzepte angeboten werden (Benzinmotor, Elektromotor, Hybridantrieb). Aufgrund dieser drei Variationspunkte können 2*2*3=12 verschiedene Konfigurationen des Konzepts „Auto“ gebildet werden.

3.1.2 Darstellung struktureller Aspekte

Die Notation für Merkmaldiagramme wurde mit Zielsetzung entworfen, Konfigurationsaspekte möglichst effizient und unabhängig von anderen Aspekten des Modells darstellen. Bedingt dadurch sind Merkmaldiagramme nicht geeignet, strukturelle Eigenschaften eines Systems darzustellen. Insbesondere sind die Kanten zwischen Merkmalen und Untermerkmalen nicht als „besteht-aus“- bzw. „ist-Teil-von“-Beziehungen im strukturellen Sinne zu verstehen [CE00]. Der Sachverhalt, dass im Informationsmodell eines Versionierungssystems das Konzept „Beziehungstyp“ zwei oder mehr genau bestimmte Instanzen des Konzepts „Objekttyp“ verbindet, kann also nicht darstellt werden, sondern nur, dass ein Objekttyp über Beziehungsenden und Beziehungen überhaupt mit anderen Objekten verbunden sein kann. Auch schon die einfache Tatsache, dass ein Informationsmodell beliebig viele verschiedene Instanzen des Konzepts „Objekttyp“ enthalten kann, ist nicht sauber darstellbar.

Für die Konstruktion eines Systems in der Domäne der Versionierungssysteme ist allerdings nicht nur die Konfiguration der in Merkmalmodellen beschriebenen Konzepte relevant, sondern auch die Struktur des Systems, die durch das Informationsmodell des Versionierungssystems definiert wird. Diese beiden Dimensionen lassen sich durch die folgenden Fragen charakterisieren:

1. *Welche* Datentypen werden gespeichert?
2. *Wie* werden die Daten verwaltet?

Merkmaldiagramme können die zweite Frage beantworten helfen, da durch eine Auswahl aus einem Merkmaldiagramm eine korrekte Konfiguration für einen Teilbaustein des Systems, beispielsweise ein einzelnes Beziehungsende, gewonnen werden kann. Es erscheint aufgrund der oben genannten Eigenschaften aber nicht sinnvoll zu sein, sie zur Analyse der Informationsmodelle zu verwenden. Deshalb wird an dieser Stelle auf eine formale Darstellung des Aufbaus von Informationsmodellen verzichtet und stattdessen nur die in der informellen Darstellung aus dem letzten Kapitel enthaltenen Konzepte untersucht und in

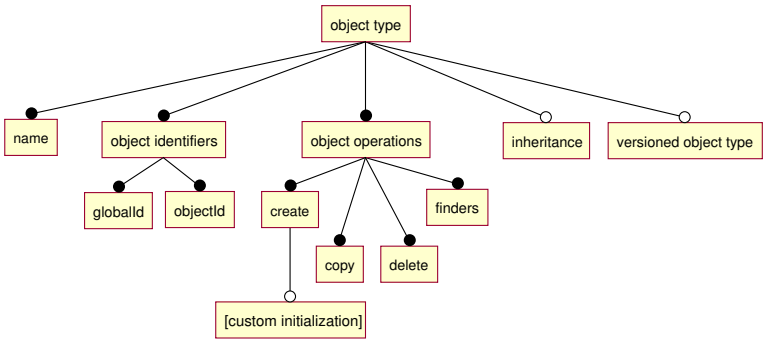
Merkmaldiagrammen dargestellt. Im nächsten Abschnitt wird dann eine formale Darstellung eingeführt werden, indem die Informationsmodelle auf UML-Klassendiagramme abgebildet werden. Die Auswahl aus den Variationspunkten der in diesem Abschnitt beschriebenen Konzepte hingegen wird über UML-Profile dargestellte werden, wodurch sich auf dieser Ebene eine geschlossene formale Darstellung ergibt.

3.2 Ein Merkmalmodell für Versionierungssysteme

Die im letzten Abschnitt vorgestellte Diagrammdarstellung wird nun angewendet, um eine Übersicht über die gemeinsamen und individuellen Merkmale von Versionierungssystemen zu gewinnen.

Die wichtigsten Bausteine eines Versionierungssystems sind die aus dem letzten Kapitel bekannten Komponenten des Informationsmodells: *Objekttypen*, die eine Einheit zur Speicherung und Versionierung von Daten definieren, die in Objekten enthaltenen *Attributtypen,* die die zu speichernden Datentypen definieren. Und schließlich auch Beziehungsenden-Typen und Beziehungstypen, die zusammen die Beziehungen innerhalb des Informationsmodell definieren. Jede dieser Komponenten wird in einem separaten Merkmaldiagramm analysiert.

Abbildung 3.6 Merkmaldiagramm für Objekttypen



Das Konzept eines Objekttyps hat eine sehr umfangreiche Semantik, deshalb kann in der Abbildung 3.6 nicht das gesamte zugehörige Merkmaldiagramm gezeigt werden. Das Diagramm wird in den Abbildungen 3.7 und 3.8 vorgesetzt, dabei werden Untermerkmale des Hauptdiagramms als Konzeptknoten für die Unterdiagramme verwendet und weiter aufgegliedert.

Zunächst sollen nun die in Abbildung 3.6 dargestellten Merkmale von Objekttypen erläutert werden:

- *name* - Jeder Objekttyp muss einen Namen haben, über den er innerhalb und außerhalb des Versionierungssystems identifiziert werden kann. Diese Identifizierung und somit auch die Namen der Objekttypen müssen eindeutig sein. Verwendet wird der Name einerseits innerhalb des Systems, er kann beispielsweise für die Bennennung der zum Objekttyp gehörenden Datenbanktabellen verwendet werden. Andererseits ist er aber auch für den Benutzer des Versionierungssystems sichtbar, da die Schnittstellen, die den Zugriff auf Instanzen des Objekttyps ermöglichen nach dem Objekttyp benannt werden, um eine klare Zuordnung zu ermöglichen.

- *object identifiers* - Zur Verwaltung der Objektinstanzen muss jeder Objekttyp die Standardattribute *objectId* und *globalId* besitzen. Wie im letzten Kapitel erläutert wurde, können über diese Attribute Objektinstanzen bzw. auch einzelne Versionen dieser Instanzen eindeutig identifiziert werden. Die Schlüsselwerte sind für den Anwender über die Schnittstelle des Versionierungsystems einsehbar und werden als Operationsparameter verwendet, um zu spezifizieren, auf welche Objektinstanz die Operation angewendet werden soll. Zusätzlich zu diesen Standardattributen werden natürlich im Informationsmodell zusätzliche Attribute für jeden Objekttyp definiert, die die eigentlichen Daten speichern. Die Eigenschaften dieser Attribute werden in Abbildung 3.9 dargestellt.
- *object operations* - Zur Erzeugung und Entfernung von Objektinstanzen muss das Versionierungssystem die Operationen *create*, *copy* und *delete* implementieren. Außerdem werden Operationen zum Auffinden von Objektinstanzen benötigt (*finders*). Ein optionaler Erweiterungspunkt im Zusammenhang mit der create-Operation ist, dass eine spezielle Initialisierungslogik spezifiziert werden kann (*custom initialization*), die die Attribute einer neuen Objektinstanz mit anderen Werten als den typabhängigen Standardwerten belegt.
- *inheritance* - Ein weiterer Variabilitätspunkt ist, dass Versionierungssysteme optional Vererbungsbeziehungen innerhalb des Informationsmodells unterstützten können. Falls dieses Merkmal ausgewählt wird, können Objekttypen von anderen Objekttypen abgeleitet werden und dabei deren Attribute und Beziehungen erben.
- *versioned object type* - Optional kann ein Objekttyp Versionierung unterstützen.

Abbildung 3.7 Unterdiagramm für versionierte Objekttypen

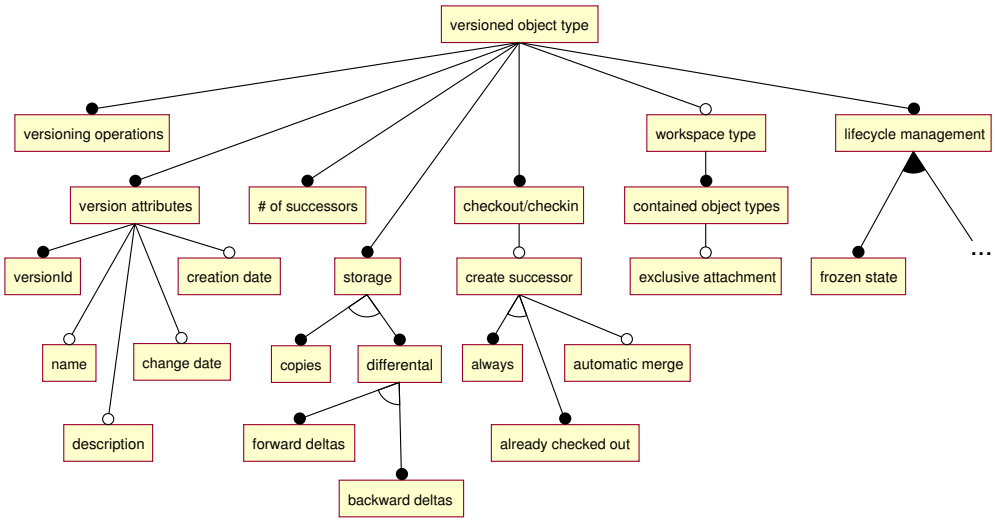


Abbildung 3.7 enthält ein Unterdiagramm, das die Eigenschaften versionierter Objekttypen aufgliedert. Das bereits im Diagramm 3.6 enthaltene Merkmal *versioned object type* wird darin um die folgenden Untermerkmale ergänzt:

- *versioning operations* - Ein Versionierungssystem bietet Operationen zur Versionsverwaltung an. Die Untermerkmale dieses Merkmals wurden in ein separates Unterdiagramm (Abbildung 3.8) ausgegliedert.

- *version attributes* - Jeder versionierte Objekttyp muss einen zusätzlichen Schlüssel (*versionId)* besitzen, mit dem einzelne Objektversion identifiziert werden können. Auch die Werte dieses Schlüssels werden über die Systemschnittstelle sichtbar. Die übrigen optionalen Attribute (*name, description, change date, creation date*) können bei Bedarf zusätzliche Informationen über die Versionen speichern. Name und Beschreibung einer Version werden dabei durch den Benutzer eingegeben, die Zeitpunkte der Versionserstellung und der letzten Änderung können hingegen automatisch durch das Versionierungssystem aufgezeichnet werden.

- *# of successors* - Dieses Merkmal legt die maximal mögliche Anzahl von Nachfolgern einer Version fest. Damit beeinflusst es die Gestalt des Versionsgraphen, beispielsweise erzwingt ein Limit von zwei Nachfolgern, dass der Graph die Gestalt eines binären Baums annimmt.

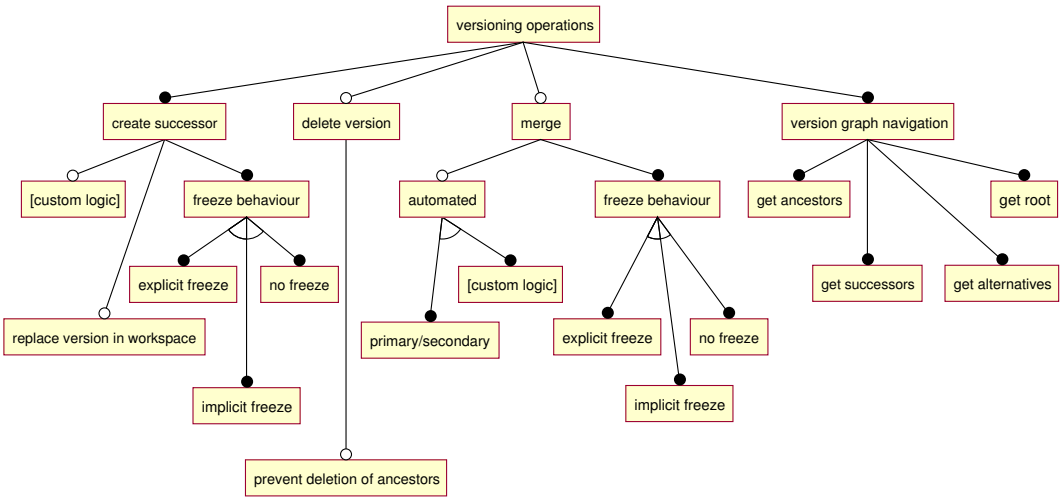
- *storage* - Die Methode zur Speicherung der Attributwerte, die einer Objektversion zugeordnet sind, ist ein weiterer Variationspunkt dieser Domäne. Im Wesentlichen gibt es zwei Alternativen: Entweder es werden für jede Version vollständige Kopien aller Attributwerte gespeichert (*copy*), oder aber die Attributwerte werden nur für eine einzige Version vollständig gespeichert und stattdessen für alle anderen Versionen die Unterschiede der Attributwerte aufgezeichnet (*differental*). Um im Fall der differentiellen Speicherung einen Attributwert zu ermitteln, wird ausgehend von der vollständig gespeicherten Version der Versionsgraph durchlaufen und dabei inkrementell alle relevanten Änderungen auf den Attributwert angewendet. Die Anlage von Kopien erlaubt einen schnellen Datenzugriff, verschwendet aber Speicherkapazität durch die redundanten Speicherung identischer Attributwerte. Eine differentielle Speicherung eliminiert diese Redundanz, erfordert aber eine Berechnung zur Ermittlung eines konkreten Attributwerts, verlangsamt also den Datenzugriff. Die differentielle Speicherung kann entweder die Unterschiede zur Vorgängerversion (*forward deltas*) oder die Unterschiede zur Nachfolgerversion (*backward deltas*) aufzeichnen. Durch diese Variationsmöglichkeit kann beeinflusst werden, wie sich der Berechnungsaufwand auf die verschiedenen Operationen verteilt. Forward Deltas sind einfach zu verwalten, weil bei der Erstellung einer neuen Version nur die Differenzen zur letzten Version berechnet und gespeichert werden müssen, dafür ist aber auch der Zugriff auf die neueste Objektversion sehr aufwendig. Die Verwaltung von Backward Deltas hingegen ist aufwändiger, dafür kann aber auf die neueste Objektversion sehr effizient zugegriffen werden, da sie intakt gespeichert wird.

- *checkout/checkin* - Ein Versionierungssystem stellt die Operationen checkout und checkin zur Verfügung, die einen einfachen Transaktionsschutz für langandauernde Bearbeitungsvorgänge des Benutzers bieten. Mit Hilfe der checkout-Operation wird eine Objektversion an einen Arbeitskontext gebunden. Die Systemschnittstelle eines Versionierungssystem ermöglicht den Systembenutzern einen Arbeitskontext als Arbeitsumgebung auszuwählen. Änderungen an einer gebundenen Version dürfen nur noch durch Benutzer vorgenommen werden, die den Arbeitskontext gewählt haben, an den die Version gebunden wurde. Die Bindung der Version kann nach der Beendigung des Bearbeitungsvorgangs mit Hilfe der checkin-Operation wieder aufgehoben werden, die Version kann dann wieder aus beliebigen Arbeitskontexten heraus manipuliert werden. Um die parallele Bearbeitung eines Objekts durch mehrere Benutzer zu vereinfachen, kann beim checkout automatisch eine neue Objektversion angelegt werden (*create successor*). Falls diese Option gewählt wird, kann die neue Version entweder in jedem Fall (*always*) angelegt werden, oder nur falls die Zielversion bereits gesperrt wurde (*already checked out*). Optional können die auf

diese Weise entstehenden Zweige des Versionsgraphen auch wieder automatisch konsolidiert werden (*automatic merge*). Dieses Merkmal kann nur ausgewählt werden, wenn das Versionierungssystem die merge-Operation zur Verfügung stellt (siehe Abbildung 3.8).

- *workspace type* - Ein Untertyp der versionierten Objekttypen sind Arbeitskontexte, die Objektversionen anderer Objekttypen (*contained object types*) aggregieren. Die Semantik der Arbeitskontexte wurde bereits im letzten Kapitel dargestellt. Relevante Konfigurationsaspekte sind in diesem Zusammenhang die Auswahl der Objekttypen die in einem Arbeitskontext enthalten sein dürfen und die Entscheidung, ob die Attachment-Beziehung zwischen Arbeitskontext und Objektversionen exklusiv ist (*exclusive attachment*).
- *lifecycle management* - Dieses Merkmal beschreibt die Verwaltung von Versionszuständen durch das Versionierungssystem. Standardmäßig existiert der Zustand *frozen*, der die Version für weitere Veränderungen sperrt. Der Übergang zum Zustand frozen wird durch den Aufruf der *freeze*-Operation ausgelöst. Der in Diagramm 3.7 gezeigte Erweiterungspunkt soll verdeutlichen, dass auch weitere zusätzliche Versionzustände denkbar sind, die aber an dieser Stelle nicht näher spezifiziert werden können.

Abbildung 3.8 Unterdiagramm für die Operationen versionierter Objekttypen



Für versionierte Objekttypen muss ein Versionierungssystem Operationen zur Verfügung stellen, die dem Anwender ermöglichen, neue Versionen anzulegen und optional auch wieder zu löschen. Außerdem sind Operationen vorhanden, die den Zugriff auf beliebige alte Objektversionen ermöglichen. Diese Operationen werden in Abbildung 3.8 analysiert.

- *create successor* - Diese Operation erzeugt eine neue Version eines Objekts. Standardmäßig werden dabei alle Attributwerte des Vorgängers für die neue Version übernommen. Ein optionaler Erweiterungspunkt ist, eine Logik zu spezifizieren, die die Attributwerte der neuen Version berechnet (*custom logic*). Ebenfalls optional ist die Möglichkeit, durch die createSuccessor-Operation automatisch die alte Objektversion in allen Arbeitskontexten, in denen sie enthalten war, durch die neue Objektversion zu ersetzten (*replace version in workspace*). Ein weiterer Variationspunkt ist die Behandlung des frozen Zustands (*freeze behaviour*). Es kann entweder vom Benutzer verlangt werden, dass er die Objektversion manuell vor dem Aufruf der createSuccessor-Operation einfriert (*explicit*

freeze), die Operation kann die Version selbst einfrieren (*implicit freeze*), oder den frozen-Zustand vollständig ignorieren (*no freeze*).

- *delete version* - Manche Versionierungssysteme bieten die Möglichkeit, Versionen aus dem Versiongraphen eines Objekts zu löschen. Optional kann dabei sichergestellt werden, dass nur Versionen ohne Nachfolger gelöscht werden dürfen (*prevent delete for ancestors*).
- *merge* - Die merge-Operation zur Konsolidierung von Versionen ist optional und wird nicht von jedem Versionierungssystem unterstützt. Die Bestimmung der richtigen Attributwerte für die zusammengeführte Objektversion wird standardmäßig dem Benutzer überlassen, der die Attributwerte manuell verändern muss. Optional ist es aber auch möglich, diesen Vorgang zu automatisieren (*automated*). Für eine automatisierte Berechnung kann zwischen einer prioritätenbasierten Berechnung (*primary/secondary*) und einer frei definierbaren Berechnungsmethode (*custom logic*) gewählt werden. Bei einer Berechnung die auf Prioritäten basiert, werden alle Attributwerte des primären Vorgängers auf den Nachfolger übertragen, nur für Fall, dass ein Attributwert nicht belegt ist, wird der entsprechende Wert des sekundären Vorgängers verwendet. Für das Verhalten der Operation im Bezug auf den frozen-Zustand gibt es die gleichen Alternativen wie bei der createSuccessor-Operation.
- *version graph navigation* - Versionierungssysteme stellen die im letzten Kapitel beschriebenen Operationen zur Navigation im Versionsgraphen eines versionierten Objekts zur Verfügung (*get successors, get ancestors, get alternatives, get root*).

Abbildung 3.9 Merkmaldiagramm für Attributtypen

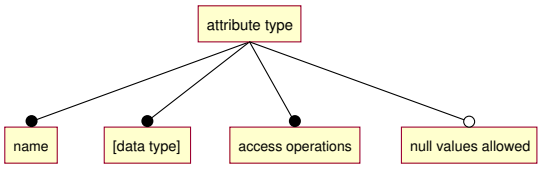
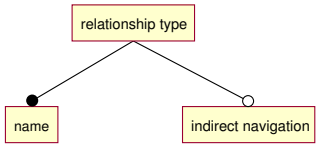


Abbildung 3.9 zeigt die Merkmale der Attributtypen, die sowohl Objekttypen als auch Beziehungstypen zugeordnet werden können. Ein Attributtyp muss, wie ein Objekttyp, einen eindeutigen Namen besitzen, der Bestandteil des Namens für die Zugriffsoperationen (*access operations*) wird. Der Platzhalter *data type* kann durch einen beliebigen Datentyp ersetzt werden, der durch die verwendete Technologieplattform unterstützt wird. Variabel in diesem Zusammenhang ist die Behandlung von null-Werten (*null values allowed*).

Abbildung 3.10 Merkmaldiagramm für Beziehungstypen



Auch für die Beziehungstypen, die im Informationsmodell eines Versionierungssystems Objekttypen verbinden, kann ein Merkmaldiagramm ermittelt werden, das Abbildung 3.10

zu entnehmen ist. Ein Beziehungstyp besitzt wie jedes Element eines Informationsmodells einen eindeutigen Namen (*name*). Das optionale Merkmal *indirect navigation* bestimmt, ob für die Benutzer des Systems Beziehungen als eigenständige Objekte sichtbar werden. Das Merkmal umfasst Zugriffsoperationen, die alle zu einem Objekt gehörigen Beziehungsobjekte bzw. alle Beziehungsobjekte eines bestimmten Typs zurücklieferen. Für die ermittelten Beziehungsobjekte können mit weiteren Operationen die verbundenen Objekte und die Attributwerte der eventuell vorhandenen Beziehungsattribute abgerufen werden. Zu beachten ist, dass eine indirekte Navigation möglich sein muss, falls ein Beziehungstyp Attribute besitzt, da ansonsten kein Zugriff auf die Attributwerte möglich wäre.

Abbildung 3.11 Merkmaldiagramm für Beziehungsenden

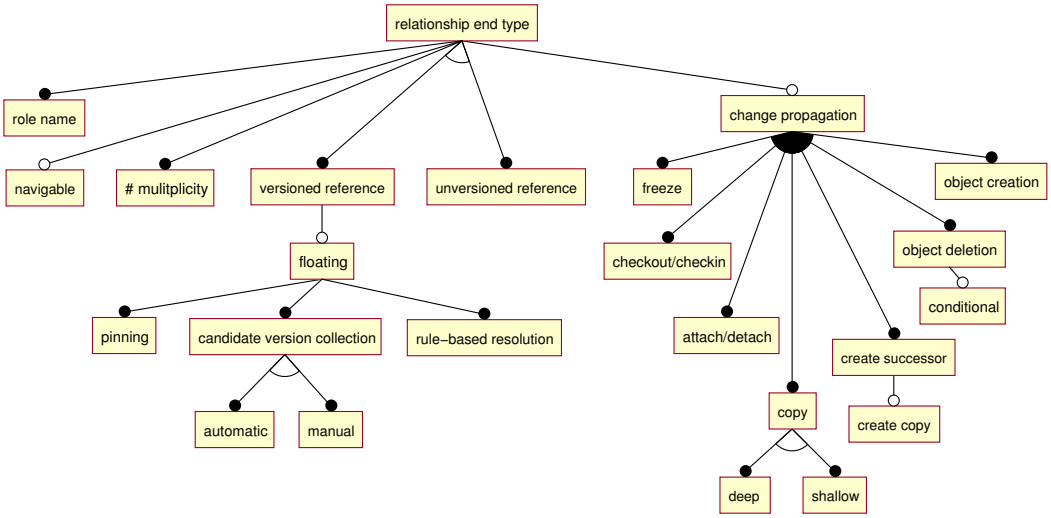


Abbildung 3.11 enthält das Merkmaldiagramm zu den Beziehungsenden des Informationsmodells. Das Konzept der Beziehungsenden wird in die folgenden Merkmale aufgegliedert:

- *role name* - Ein Beziehungsende besitzt einen Namen, der die semantische Bedeutung des zugeordneten Objekts in der Beziehung verdeutlicht. Dieser Name wird für die Benennung von Navigationsoperationen verwendet.
- *navigable* - Falls dieses Merkmal ausgewählt wird, kann der Benutzer von diesem Beziehungsende aus zum gegenübergesetzten Ende gelangen.
- *# multiplicity* - Dieses Merkmal bestimmt die maximale Kardinalität des Beziehungsendes. Das Merkmal bestimmt also, ob dieses Beziehungsende nur mit einem Objekt verbunden werden kann, oder mit mehreren. Eine Einschränkung der minimalen Kardinalität ist nicht vorgesehen, diese beträgt also immer null.
- *versioned reference / unversioned reference* - Wie bereits im letzten Kapitel dargelegt wurde, müssen besondere Vorkehrungen getroffen werden, wenn ein Beziehungsende mit einem versionierten Objekt verbunden ist. In diesem Fall besteht die Möglichkeit, gleitende Beziehungsenden zu verwenden (*floating*). Gleitende Beziehungsenden erfordern die Verwaltung von Kandidatenmengen (*candidate version collection*) und vorausgewählten Objektversionen (*pinning*). Die Inhalte der Kandidatenmengen können entweder durch den Benutzer bestimmt werden (*manual*), oder automatisch durch das Versionierungssystem generiert werden (*automatic*). Bei der automatischen Verwaltung werden stets alle

Versionen eines Objekts in die Kandidatenmenge aufgenommen. Zur Auswahl einer passenden Zielversion aus der Kandidatenmenge kann eine Operation benutzt werden (*rule based resolution*), die nach folgenden Regeln vorgeht: Falls ein aktueller Arbeitskontext existiert, wird nur die Objektversion betrachtet, die in diesem Arbeitskontext liegt. Anderenfalls wird zunächst überprüft, ob eine vorausgewählte Version existiert, ansonsten wird die zeitlich neueste Objektversion verwendet.

- *change propagation* - Falls dieses Merkmal Anwendung findet, können viele Arten von Zustandsänderungen über Beziehungen von Objekt zu Objekt propagiert werden. Besonders interessant ist in diesem Zusammenhang die Propagierung bei der Erzeugung oder Löschung eines Objekts (*object creation, object deletion, createSuccessor, copy*). Beim Anlegen eines neuen Objekts (*object creation*), können automatisch Beziehungen und die darüber verknüpften Objekte angelegt werden. Umgekehrt können verknüpfte Objekte beim Löschen eines Objekts (*object deletion*) automatisch mit gelöscht werden. Falls das Untermerkmal *conditional* mit ausgewählt wird, wird die automatische Löschung nur durchgeführt, falls ein Objekt nicht auch noch an anderen Beziehungen beteiligt ist. Bei der Propagierung der *createSuccessor*-Operation gibt es die Option, von den verknüpften Objekten Kopien anzulegen (*create copy*), anstatt einfach nur neue Objektversionen zu erzeugen. Beim Kopieren eines Objekts (*copy*) gibt es schließlich die Alternative, entweder flache (*shallow*) oder tiefe (*deep*) Kopien zu erzeugen. Im Fall von flachen Kopien werden nur die Beziehungen eines Objekts kopiert und die Objektkopie teilt sich die verknüpften Objekte mit dem Original. Im Fall von tiefen Kopien werden auch alle verknüpften Objekte kopiert und entsprechende Beziehungen angelegt. Bei allen Propagierungsoptionen ist zu beachten, dass alle an der Beziehung beteiligten Objekttypen in der Lage sein müssen, die propagierende Operation durchzuführen. Beispielsweise müssen alle Objekttypen mit den gleichen Arbeitskontexttypen verbunden sein, falls die attach/detach Operationen propagiert werden sollen.

3.3 UML-Profile

Wie bereits angekündigt, soll nun eine formale Spezifikationssprache definiert werden, die die im letzten Abschnitt ermittelte Variabilität ausdrücken kann. Mit dieser Sprache können Versionierungssysteme vollständig spezifizieren werden, sodass ein Generator ohne weitere Benutzereingriffe die Spezifikation in ein lauffähiges Softwaresystem umsetzen kann.

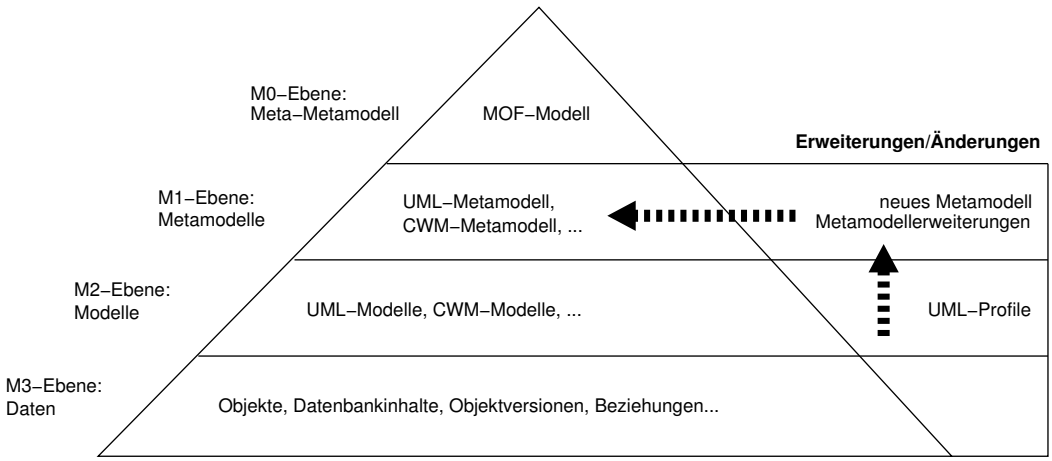
Da bisher noch keine Festlegung auf eine konkrete Implementierungstechnologie vorgenommen wurde, erscheint es sinnvoll, auch eine plattformunabhängige Spezifikationssprache zu verwenden. Wie bei den PIM des MDA-Ansatzes entsteht dadurch der Vorteil, dass bei späteren Änderungen an der Implementierung der Produktlinie die Spezifikationen der einzelnen Produktlinienmitglieder nicht angepasst werden müssen. Diese Anforderung wird von der UML erfüllt, außerdem kann auf ein große Anzahl von Werkzeugen zur Erzeugung und Verarbeitung von UML-Modelle zurückgegriffen werden: Beliebige UML-Modellierungsprogramme sollen für den Entwurf der Systemspezifikationen verwendet werden, die dann über die standardisierte XMI-Schnittstelle [OMG03c] an den Generator übertragen werden sollen.

Die zu definierende Spezifikationssprache soll *domänenspezifisch* sein, d.h. sie soll ein angepasstes Vokabular besitzen und Abstraktionsmöglichkeiten bieten, die über die von universal einsetzbaren Sprachen hinausgehen. Dadurch können Versionierungsysteme rein deklarativ

beschrieben werden und es kann auf eine imperative Beschreibung verzichtet werden. Es kann also beispielsweise ein Objekttyp als Arbeitskontexttyp gekennzeichnet werden, ohne dass explizit definiert werden muss, worin sich die Semantik eines Arbeitskontextes von der Semantik eine normalen Objekts unterscheidet. Auf diese Weise kann eine sehr kompakte Darstellung erreicht werden, die leicht zu verstehen und zu validieren ist. Da die UML per se kein spezielles Vokabular für die Domäne der Versionierungssysteme anbietet, muss nun überlegt werden, in welcher Form die Sprache erweitert werden muss.

Die entscheidenden strukturellen Merkmale eines Versionierungssystems in Form seines Informationsmodells lassen sich mit Hilfe von UML-Klassendiagrammen darstellen, indem die Objekt- und Beziehungstypen des Informationsmodells mit Klassen und Assoziationen eines Klassendiagramms identifiziert werden. Diese Vorgehensweise wurde bereits in Abbildung 2.4 angewandt, die ein Klassendiagramm mit dem Informationsmodell eines Content Management Systems enthält. Es fehlt aber eine Möglichkeit, zwischen den einzelnen Untertypen, etwa zwischen versionierten und unversionierten Objekttypen, zu unterscheiden. Außerdem muss auch eine Möglichkeit gefunden werden, die gewählte Konfiguration für die in den Merkmaldiagrammen dokumentierten Variationsmöglichkeiten im UML-Modell aufzuzeichnen. Die Lösung für beide Probleme wird ein UML-Profil sein, das es erlaubt, zusätzliche Markierungen in UML-Klassendiagramme einzubringen, die die detaillierte Semantik des Informationsmodells in einer Form dokumentieren, die sowohl für einen menschlichen Betrachter als auch für den Generator verständlich ist.

Abbildung 3.12 Die MOF-Metamodellarchitektur und ihre Erweiterungs- und Änderungsmöglichkeiten



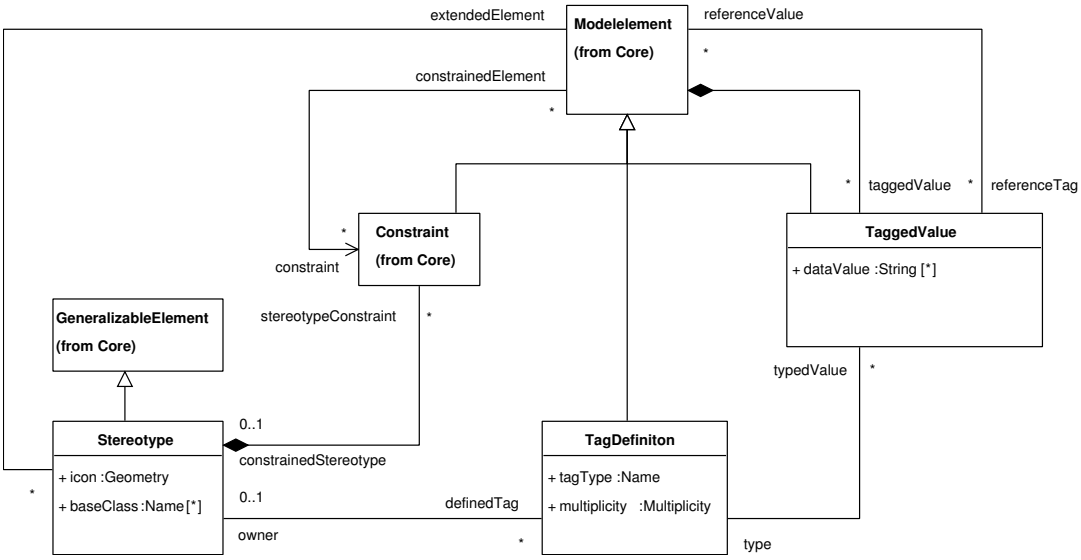
Im Abschnitt 2.3.1 wurden die grundsätzlichen Möglichkeiten zur Einführung neuer Sprachelemente in der UML bzw. zur Schaffung einer Alternative zur UML bereits vorgestellt:

1. Es kann ein vollständig neues Metamodell als Ersatz für das UML-Metamodell entwickelt werden. Dieser Ansatz bietet die größtmögliche Freiheit bei der Definition eine domänenspezifischen Sprache, da im Unterschied zu einer Erweiterung des UML-Metamodells nicht darauf geachtet werden muss, das neu eingefügte Modellelemente mit den bereits vorhandenen Elementen semantisch kompatibel sind. Diesem Vorteil wird durch den Nachteil erkauft, das UML-Werkzeuge ohne weitere Modifikationen nicht eingesetzt werden können.

2. Das UML-Metamodell kann um zusätzliche Modellelemente erweitert werden. Beispielsweise können die Modellelemente für Objekt- und Beziehungstypen von den Elementen Class bzw. Association der UML-Klassendiagramme abgeleitet werden. Diese Erweiterungen können entweder regulär auf der Metamodellebene vorgenommen werden (heavyweight extensions) oder durch UML-Profile auf der Modellebene (lightweight extensions).

Abbildung 3.12 zeigt die drei Ansätze im Kontext der vierschichtigen MOF-Metamodellarchitektur. Die Anwendung von UML-Profilen bewirkt wie die anderen Ansätze eine Veränderung des Metamodells, da die Profile aber auf der Modellebene M2 definiert werden und die Metamodellebene M1 nur implizit verändern, wird von *virtuellen Metamodellen* gesprochen. Virtuelle Metamodelle unterliegen der Einschränkung, dass ihrer Semantik nicht der Semantik des ursprünglichen Metamodells wiedersprechen darf, sie dürfen also nur reine Erweiterungen der Semantik definieren. Diese Einschränkung ermöglicht es andererseits aber auch, dass die erweiterten Modelle ohne die Kenntnis des zugehörigen Profils verarbeitet werden können. Dies ermöglicht den Einsatz von Standardwerkzeugen und ist der Grund warum ein UML-Profil für die Definition der Spezifikationssprache verwendet werden wird.

Abbildung 3.13 Das Extensions-Mechanisms-Package des UML-Metamodells



UML-Profile sind Teil des UML-Standards [OMG03a]. Die nötigen Konstrukte sind Teil des UML-Metamodells und werden im Subpackage „Extension Mechanisms“ des Core-Package definiert. Abbildung 3.13 zeigt die Inhalte des Extension-Mechanisms-Package.

Der Elementtyp *Stereotype*

Stereotype ist das wichtigste Element des Extension-Mechanism-Package. Mit diesem Element kann die Semantik der Instanzen beliebiger Metamodell-Elemente verändert bzw. verfeinert werden. Ein Stereotyp kann auf Modellelemente all derjenigen Typen angewendet werden, die im seinem Attribut *baseClass* enthalten sind. Ein Stereotyp, der beispielsweise für das Metamodellelement *Classifier* definiert wurde, darf auf Instanzen von *Classifier* und allen seinen Untertypen, also z.B. *Class* angewendet werden. Die Anwendung geschieht, in-

dem ein Stereotype-Element über die Beziehung *extendedElement* mit einer *Modelelement*-Element verknüpft wird. Auf Modellelemente können beliebig viele Stereotypes gleichzeitig angewendet werden, dabei wird die Semantik der Elemente in mehrerer Hinsicht verändert:

- Durch Anwendung eines Stereotypes kann das betroffene Element von anderen Elementen des gleichen Typs unterschieden werden, auch wenn diese ansonsten identische Attributwerte besitzen. Es wird also der Typ des Elements genauer bestimmt. Die semantische Bedeutung eines durch ein UML-Profil neu geschaffenen Subtyps ergibt sich aus dem Kontext der Anwendungdomäne, für die das Profil geschaffen wurde. So ist beispielsweise durch die in diesem Kapitel durchgeführte Merkmalanalyse ermittelt worden, worin die semantischen Unterschiede zwischen versionierten und unversionierten Objekttypen bestehen. Ein UML-Profil für Versionierungsdienste ermöglicht durch die Einführung entsprechende Stereotypes, die Unterscheidung zwischen UML-Klassen die versionierte bzw. unversionierte Objekttypen verkörpern.
- Um die semantischen Unterschiede zu verdeutlichen, kann ein Stereotype eine geänderte graphische Darstellung der betroffenen Modellelemente spezifizieren. Dazu dient das Attribute *icon*, das ein Bild enthält, das von einem UML-Modellierungsprogramm anstelle des Standardsymbols in der Diagrammdarstellung verwendet werden kann.
- Falls die Änderung der Semantik formal mit Hilfe des UML-Constraint-Mechanismus spezifiziert werden kann, ist es möglich, innerhalb eines UML-Profils *Constraint*-Elemente zu definieren, die beispielsweise OCL-Ausdrücke enthalten. Diese werden über die *stereotypeConstraint*-Beziehung mit einem Stereotype verbunden. Dies hat zur Folge, dass der Constraint auf alle Modellelemente angewendet wird, auf die der Stereotype angewendet wird. Diese Constraints haben also die gleiche Wirkung wie die Constraints, die mit den nativen UML-Metamodellelementen verknüpft sind.
- Stereotypes ermöglichen es außerdem, für den virtuellen Elementtyp, den sie schaffen, zusätzliche Attribute zu definieren. Dies geschieht durch *TagDefinition*-Elemente, die Name und Typ des zusätzlichen Attributs definieren. Für gekennzeichnete Elemente können Werte für diese Attribute definieren werden, indem sie mit *TaggedValue*-Elementen verbunden werden, die den Tag Definitions des Stereotypes entsprechen.

Da das Metamodellelement Stereotype vom Metamodellelement *GeneralizableElement* abgeleitet ist, können Stereotypes auch eine Vererbungshierarchie bilden, wobei die abgeleiteten Stereotypes analog zu abgeleiteten Klassen alle Eigenschaften ihrer Vorgänger erben.

Die Elementtypen *TagDefinition* und *TaggedValue*

Wie bereits gesagt, dienen die *TagDefinition*- und *TaggedValue*-Elemente dazu, für den durch einen Stereotype geschaffenen virtuellen Elementtyp Tagged Definitions bzw. Tag Values zu definieren, die als zusätzliche virtuelle Attribute verwendet werden können. Ein TagDefinition-Element besitzt die folgenden Attribute:

- *name* (geerbt von ModelElement) - Der Name des virtuellen Attributs.
- *multiplicity* - Die Kardinalität gibt an, wieviele Datenwerte ein Tagged Value besitzen kann, der der Tag Definition entspricht. Eine Kardinalität von „0..1“ bedeutet, dass nicht jedes mit dem Stereotype markierte Modellelement einen Tagged Value dieses Typs besitzen muss.
- *tagType* - Dieses Attribut enthält den Namen des Datentyps des virtuellen Attributs. Als Datentyp können alle UML-Datentypen und auch beliebige andere UML-Elementtypen verwendet werden.

Bei der Anwendung eines Stereotypes werden *TaggedValue*-Elemente, die über die *typedValue*-Beziehung eine Referenz auf die Tag Definition besitzen. Der Attributwert kann entweder im Attribut *dataValue* als String abgelegt werden oder bei einem getypten Tag über die Beziehung *referenceValue* durch ein weiteres Modellelement definiert werden.

Ein UML-Profil wird innerhalb eines mit dem speziellen Stereotype „profile“ markierten Package definiert, das die zugehörigen Stereotype- und TagDefinition-Elemente enthält. Typischerweise wird ein Profil nicht individuell für ein einzelnes Modell erstellt, sondern es wird auf standardisierte Profile zurückgegriffen, die nicht systemspezifisch, sondern domänenspezifisch konzipiert sind. Eine solche Standardisierung kann entweder durch den Hersteller eines Softwareprodukts erfolgen, der dieses Profil verarbeitet, wie z.B. der Generator, der im Rahmen dieser Arbeit entwickelt wurde. Oder aber ein Standard wird durch ein herstellerübergreifendes Komitee definiert, dies ist beispielsweise bei der Abbildung zwischen UML und EJB der Fall, die im Rahmen des Java Community Process entwickelt wird [JCP01].

Zur Verdeutlichung des Erweiterungsmechanismus soll nun dargestellt werden, wie mit einem Profil Klassen eines Klassendiagramms als persistent gekennzeichnet werden können und gleichzeitig spezifiziert werden kann, in welcher Datenbank die Objekte gespeichert werden sollen. Dazu definiert der Anwender einen Stereotype „persistent“ für das Metamodellelement „Class“, den er den betroffenen Klasse zuweist. Zum Stereotype wird eine Tag Definition mit dem Namen „databaseUrl“ und dem Datentyp „String“ angelegt. An die markierten Klassen kann nun ein Tagged Value angefügt werden, der die URL der Datenbank enthält.

Ein Werkzeug kann nun zwischen regulären Klassen und solchen, deren Objektinstanzen persistent gespeichert werden sollen, unterscheiden. Für die mit dem Stereotype „persistent“ markierten Klassen könnte das Werkzeug ein Datenbankschema erzeugen, dass die Attribute der markierten Klassen berücksichtigt. Weiterhin könnte Quellcode in einer beliebigen Programmiersprache generiert werden, der Objektinstanzen in die Datenbank aus- und einlagert. Für die Generierung dieses Codes muss der Generator wissen, wie auf die zugehörige Datenbank programmatisch zugegriffen werden kann. Diese Zusatzinformation erhält aus den Tagged Values vom Typ „databaseUrl“ die an die persistenten Klassen angefügt worden sind.

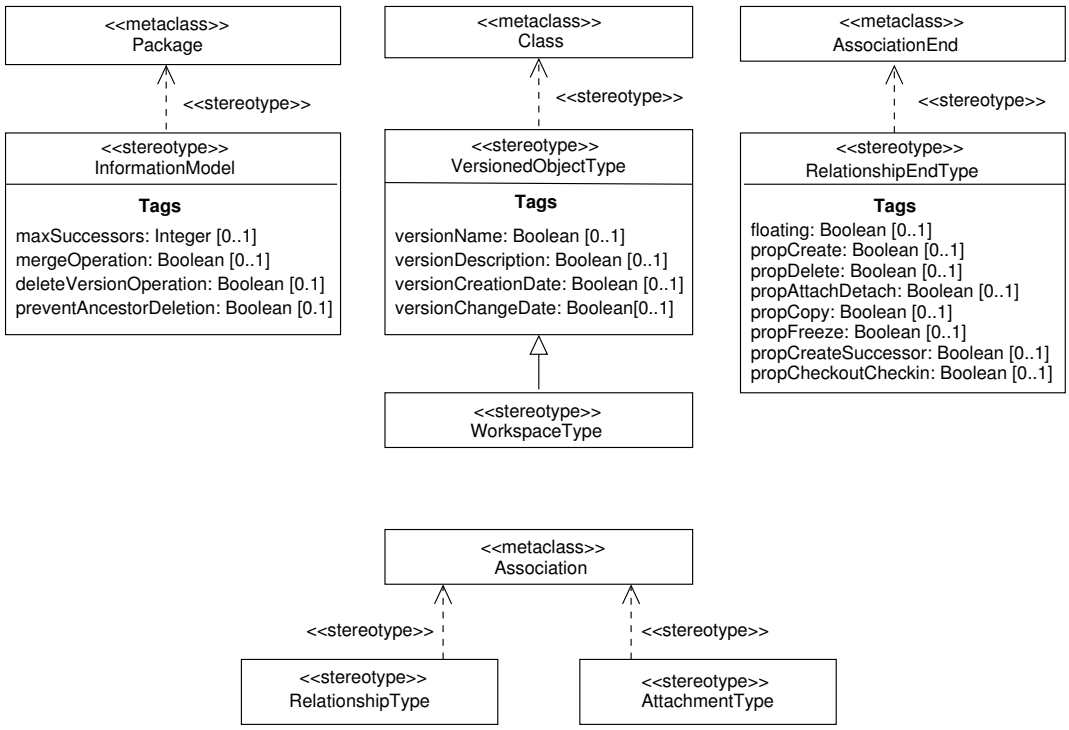
3.4 Ein UML-Profil für Versionierungssysteme

Die Abbildung 3.14 zeigt ein UML-Profil, das die Definition eines Informationsmodells für ein Versionierungssystem ermöglicht und einen Großteil der während der Merkmalanalyse bestimmten Variationsmöglichkeiten bietet. Dieses Profil enthält alle Variationmöglichkeiten, die durch die im nächsten Kapitel beschriebenen Implementierung eines Generators für Versionierungssystem unterstützt werden.

InformationModel

Dieser Stereotype kennzeichnet das Package, das das Informationsmodell enthält. Diese Kennzeichnung ermöglicht es einem Generator, das richtige Package in einem umfangreichen UML-Modell selbstständig zu finden. Außerdem definiert dieser Stereotype einige Attribute, die globale Konfigurationsoptionen steuern:

Abbildung 3.14 UML-Profil für Versionierungssysteme



- *maxSuccessors* - Mittels dieses Attribut wird festgelegt, wieviele Nachfolger eine Objektversion besitzen darf. Falls dieses optionale Attribut nicht vorhanden ist, ist eine unbegrenzte Anzahl an Nachfolgeversionen zugelassen.

- *mergeOperation* - Dieses Attribut bestimmt, ob verschiedene Zweige eines Versionsgraphen mit Hilfe der merge-Operation zusammengeführt werden können. Diese Option hat einen gewissen Einfluss auf die Komplexität des aus der Spezifikation erzeugten Versionierungssystems, da für die Implementierung dieser Funktionalität zusätzliche Daten zu jedem einzelnen Objekt gespeichert und verarbeitet werden müssen. Die merge-Operation und die notwendigen Datenstrukturen werden nur generiert, wenn dieses Attribut vorhanden ist und den Wert „true“ hat. Wie die anderen boolschen Attribute dieses Profils hat es eine Kardinalität von „0..1“, d.h. es muss nicht in jedem durch den Stereotyp gekennzeichneten Modellelement vorhanden sein. Falls das Attribut fehlt, wird „false“ als Standardwert verwendet.

- *deleteVersionOperation* - Hiermit wird spezifiziert, ob der Benutzer einmal erzeugte Objektversionen wieder löschen kann.

- *preventAncestorDeletion* - Dieser Tagged Value beeinflusst die Semantik mit der letzten Option. Falls er auf „true“ gesetzt wird, lässt das resultierende Versionierungssystem die Löschung einer Version nur dann zu, wenn für sie noch keine Nachfolgerversionen existieren.

VersionedObjectType

Alle Klassen, die in einem als Informationsmodell gekennzeichneten Package enthalten sind, werden implizit als Objekttypen angesehen. Die Daten ihrer Objektinstanzen werden also persistent und gegebenenfalls auch versioniert gespeichert.

Alle versionierten Objekttypen benötigen eine gesonderte Kennzeichnung durch den Stereotype *VersionedObjectTyp*e. Da versionierte Objekttypen eine wesentlich komplexere Semantik als unversionierte Objekttypen haben, definiert dieser Stereotype einige Attribute, die das Verhalten eines spezifischen versionierten Objekttyps genauer beschreiben:

- *versionName, versionDescription, versionCreationDate, versionChangeDate* - Diese Attribute bestimmen, welche zusätzlichen Informationen über die Versionen eines versionierten Objekts gespeichert werden. Wenn die entsprechenden Tagged Values auf „true“ gesetzt werden, wird ein Versionsname und eine Beschreibung gespeichert, über die der Benutzer die in der Version vorgenommenen Änderungen beschreiben kann. Die Tagged Values *versionCreationDate* bzw. *versionChangeDate* bewirkten eine automatische Aufzeichnung des Zeitpunkts der Versionserzeugung bzw. der letzten Änderung.

WorkspaceType

Da Arbeitskontexte selbst versionierte Objekte sind, wird der Stereotype *WorkspaceType*, der Arbeitskontexte kennzeichnet, vom Stereotype VersionedObjectType abgeleitet. Für Arbeitskontexte können also die gleichen Konfigurationsoptionen wie für versionierte Objekte verwendet werden.

Die Attachment-Beziehungen zu den Objekttypen, die im Arbeitskontext gespeichert werden können, werden mittels regulärer Aggregationsbeziehungen im UML-Modell modelliert.

RelationshipType und AttachmentType

Die beiden Stereotypes *RelationshipType* und *AttachmentType* kennzeichnen Assoziationen innerhalb eines UML-Klassendiagramms als Beziehungstypen zwischen Objekttypen bzw. als Attachment-Beziehung zwischen einem Arbeitskontexttyp und einem Objekttyp. Diese beiden Stereotypes bieten keine weiteren Variationsmöglichkeiten in der Form von Tagged Values. Ihr Einsatz ist aber analog zum Stereotype *WorkspaceType* notwendig, da der Generator ansonsten nicht zwischen den beiden verschiedenen Beziehungstypen unterscheiden kann.

RelationshipEndType

Dieser Stereotype kennzeichnet Assoziationsenden, die mit Objekttypen verbunden sind. Er spezifiziert einige Konfigurationsoptionen, die die Semantik der Beziehung beeinflussen:

- *floating* - Dieses Attribut darf nur benutzt werden, wenn der referenzierte Objekttyp versioniert wird. In diesem Fall bewirkt es, dass ein gleitendes Beziehungende erzeugt wird, dass also das Beziehungsende mehr als eine Objektversion referenzieren kann.
- *propCreate, propDelete, propAttachDetach, propCopy, propFreeze, propCreateSuccessor, propCheckoutCheckin* - Wie bereits erwähnt, können über Beziehungen verschiedene Operationen propagiert werden. Diese sieben Attribute spezifizieren, ob die dazugehörigen Operationen über dieses Beziehungsende weiterverbreitet werden oder nicht. Standardmäßig wird keine Operation weiterverbreitet.

3.4.1 Anwendung des UML-Profils

Im letzten Kapitel wurde in Abbildung 2.4 ein UML-Modell mit dem Informationsmodell eines einfachen Content-Management-Systems vorgestellt. Dort wurden Kommentare verwendet, um die die Bedeutung der einzelnen Klassen und Beziehungen anzuzeigen. Mit Hilfe des UML-Profils für Versionierungssysteme können die Modellelemente nun formal gekennzeichnet werden. Auf diese Weise können sie durch einen Generator identifiziert werden und in entsprechende Implementierungskonstrukte umgesetzt werden.

Abbildung 3.15 Anwendung des UML-Profils auf das Modell des CMS

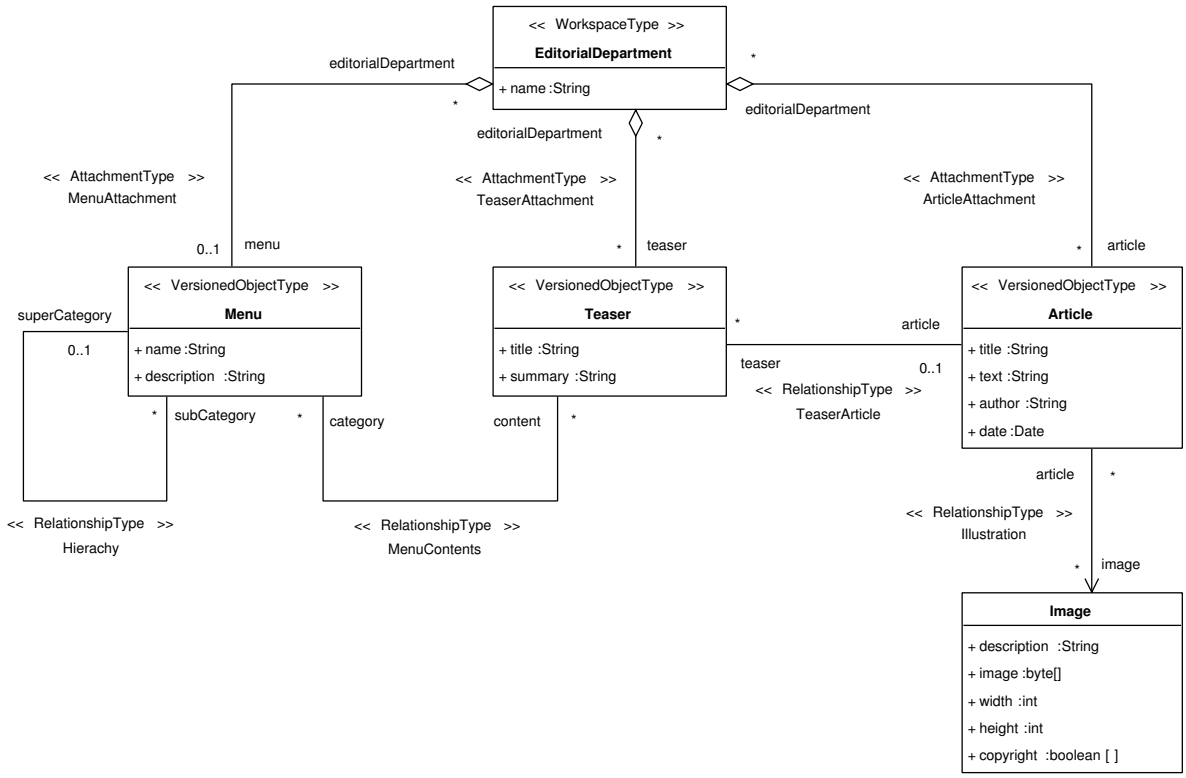


Abbildung 3.15 zeigt das UML-Modell nach Anwendung des UML-Profils für Versionierungssysteme. Prinzipbedingt sind einige Elemente des UML-Modells nicht im Diagramm sichtbar, insbesondere die eingefügten Tagged Values und das Package in dem die Klassen des Diagramms enthalten sind. Diese Elemente können der folgenden tabellarischen Aufstellung entnommen werden:

Package: cmsDemo

Stereotype:	<<InformationModel>>
Tagged Values:	mergeOperation = „true“ deleteVersionOperation = „true“ preventAncestorDeletion = „true“ maxSuccessors = „10“

In diesem Package sind alle Objekttypen unseres Informationsmodells enthalten, dementsprechend wurde es mit dem Stereotype *InformationModel* gekennzeichnet, damit der Generator das Informationsmodell problemlos auffinden kann. Außerdem werden an dieser Stelle über Tagged Values einige globale Konfigurationen vorgenommen, beispielsweise wird spezifiziert, dass das zu generierende Versionierungsystem merge- und deleteVersion-Operationen für versionierte Objekttypen besitzen soll.

Class: EditorialDepartment

Stereotype:	<<WorkspaceType>>
Tagged Values:	versionName = „true“

Der Objekttyp *EditioralDepartment* wird mit dem Stereotype *WorkspaceType* als Arbeitskontext gekennzeichnet. Die zugehörigen Beziehungen zu den enthaltenen Objekttypen werden weiter unten aufgeführt.

Wie alle anderen versionierten Objekttypen dieses Informationsmodells soll für die Versionen des Objekttyps EditorialDepartment ein Name verwaltet werden, der durch den Systembenutzer eingeben wird, deshalb besitzt EditorialDepartment den Tagged Value *versionName*.

Class: Menu

Stereotype:	<<VersionedObjectType>>
Tagged Values:	versionName = „true“

Der *Menu*-Objekttyp wird durch den Stereotype *VersionedObjectType* als versionierter Objekttyp gekennzeichnet.

Class: Teaser

Stereotype:	<<VersionedObjectType>>
Tagged Values:	versionName = „true“

Class: Article

Stereotype:	<<VersionedObjectType>>
Tagged Values:	versionName = „true“
	versionCreationDate = „true“
	versionChangeDate = „true“

Auch der Objekttyp *Article* soll durch das Versionierungssystem versioniert verwaltet werden. Für diesen Objekttyp soll das System den Zeitpunkt der Objekterstellung und den Zeitpunkt der letzten Änderung automatisch aufzeichen, deshalb wurden die Tagged Values vom Typ *versionCreationDate* bzw. *versionChangeDate* eingefügt.

Class: Image

Die Klasse *Image* besitzt keinen Stereotype und wird deshalb implizit als unversionierter Objekttyp angesehen, da sie in einem mit dem Stereotype *InformationModel* gekennzeichneten Package definiert wird.

Association: MenuAttachment, TeaserAttachment, ArticleAttachment

Mit diesen drei Assoziationen wird spezifiziert, dass die Objekttypen *Menu, Teaser* und *Article* an den Arbeitskontexttyp gebunden sind. Um diese speziellen Beziehungstyp von regulären Beziehungen im Informationmodell zu unterscheiden, wurden alle drei Assoziation mit dem Stereotype *AttachmentType* gekennzeichnet. Die Assoziationsenden dieser Beziehun-

gen wurden nicht durch Stereotypes bzw. Tagged Values gekennzeichnet, da das verwendete Profil keine Konfigurationmöglichkeiten für die Enden dieses Beziehungstyps vorsieht.

Association: Hierachy

Association:	Stereotype:	<<RelationshipType>>
superCategory	Stereotype:	<<RelationshipEndType>>
	Tagged Values:	floating = „true“
subCategory	Stereotype:	<<RelationshipEndType>>
	Tagged Values:	floating = „true“
		propDelete = „true“

Die Assoziation *Hierachy* ist eine reflexive Beziehung des versionierten Objekttyps *Menu*, über die ein Menüobjekt (*superCategory*) mit seinen Untermenüs (*subCategory*) verbunden ist. Diese Beziehung ist über den Stereotype *RelationshipType* an der Assoziation und durch den Stereotype *RelationshipEndType* an den Assoziationsenden als reguläre Beziehung gekennzeichnet. Da Menu ein versionierter Objekttyp ist, sollen beide Beziehungsenden gleitend sein, dies wird durch den Tagged Value *floating* spezifiziert. Zusätzlich wird noch spezifiziert, dass das über das Beziehungsende subCategory die delete-Operation propagiert werden soll. Dies bewirkt, dass bei der Löschung eines Menüobjekts automatisch auch alle vorhanden Untermenüs gelöscht werden.

Association: MenuContents

Association:	Stereotype:	<<RelationshipType>>
category	Stereotype:	<<RelationshipEndType>>
	Tagged Values:	floating = „true“
content	Stereotype:	<<RelationshipEndType>>
	Tagged Values:	floating = „true“

Diese Beziehung verbindet die versionierten Objekttypen *Menu* (*category*) und *Teaser* (*content*) und speichert damit die Inhalte der Menüseiten. Die Beziehungsenden dieser Beziehung sind ebenfalls gleitend.

Association: TeaserArticle

Association:	Stereotype:	<<RelationshipType>>
teaser	Stereotype:	<<RelationshipEndType>>
	Tagged Values:	floating = „true“
		propCreate = „true“
		propDelete = „true“
		propFreeze = „true“
		propAttachDetach = „true“
		propCopy = „true“
		propCreateSuccessor = „true“
		propCheckoutCheckin = „true“
article	Stereotype:	<<RelationshipEndType>>
	Tagged Values:	floating = „true“

Zwischen den den Objekttypen *Article* und *Teaser* existiert ein weiterer Beziehungstyp mit gleitenden Beziehungsenden. Da ein Teaser ein semantisch sehr enges Verhältnis zu dem Artikel hat, den er beschreibt, wird vom Objekttyp Artikel aus alle propagierungsfähigen Operation auf die verbunden Teaserobjekte propagiert. Beispielsweise wird beim Kopieren

eines Artikels automatisch auch Kopien aller vorhanden Teaser angelegt und mit dem neuen Artikelobjekt verbunden.

Association: Illustration

Association:	Stereotype:	<<RelationshipType>>
article	Stereotype:	<<RelationshipEndType>>
image	Stereotype:	<<RelationshipEndType>>
	Tagged Values:	propDelete = „true“
		propCopy = „true“

Der Beziehungstyp *Illustration* der Artikel mit den zugehörigen Bildern verbindet, hat ebenfalls den Charakter einer Aggregationsbeziehung, deshalb werden auch hier bei der Löschung eines Artikel bzw. bei der Anlegung einer Artikelkopie die Operation zu die verbunden *Image*-Objekte propagiert. Da Image ein unversionierter Objekttyp ist, darf hier der Tagged Value *floating* nicht verwendet werden, und auch die Propagierung von versionierungsspezifischen Operation wie *freeze* oder *createSuccessor* ist nicht zulässig.

Kapitel 4

Ein Generator für Middleware-basierte Versionierungssysteme

Die Vorzüge eines domänenbasierten Ansatzes für Analyse und Design kommen erst dann voll zur Geltung, wenn die Umsetzung der plattformunabhängigen Systemspezifikation in die plattformspezifische Implementierung durch einen Generator weitesgehend automatisiert wird.

Durch die Abbildung auf eine allgemeine Programmiersprache wird die Struktur der plattformunabhängigen Spezifikation verändert, wodurch ein Großteil der durch die domänenspezifischen Darstellung gewonnen Abstraktionen verloren geht. Dieser Effekt entsteht durch die Hinzufügung von Implementierungsdetails und wird verstärkt durch die Implementierung von Systemaspekten, die nicht einer einzelnen Komponente zuzuordnen sind, wie beispielsweise der Fehlerbehandlung. Auch durch komponentenübergreifende Optimierungen wird die zugrundeliegende Struktur des Systems verfälscht. Im resultierende Programmcode ist also die Struktur der Systemspezifikation nur noch bedingt erkennbar. Deshalb ist eine manuelle Implementierung aller Produktlinienmitglieder aufwändig und fehlerträchtig.

Frameworks und andere Komponententechnologien können dazu beitragen, den Aufwand für die Implementierung eines Systems durch die Wiederverwendung von Teilen des Systemdesigns und der Implementierung zu senken. Allerdings bieten diese Technologien prinzipbedingt nur eine schlechte Unterstützung bei der Implementierung von komponentenübergreifenden Systemaspekten. Es müssen entweder Komponentenversionen für alle gültigen Merkmalkombinationen entwickelt werden, was einen erheblichen Entwicklungs- und Wartungsaufwand erzeugt, oder es müssen die variablen Systemanteile gekapselt und ausgegliedert werden, was zu einer weniger leistungsfähigen Implementierung führt.

Deshalb sind Generatoren vorzuziehen, da sie ebenfalls die Wiederverwendung von Design und Implementierung ermöglichen, gleichzeitig aber auch umfangreiche Anpassungen an die Systemspezifikation durchführen können.

Ein Generator ist ein Transformationssystem, das eine gegebene Spezifikation in eine Implementierung umwandelt. Hierbei führt ein Generator folgende Arbeitsschritte durch [CE00]:

1. Er liest die Spezifikation ein und wandelt sie in eine interne Darstellung um. Falls die Spezifikation als Text vorliegt, muss der Generator, wie ein gewöhnlicher Compiler, eine Syntaxanalyse durchführen um den Inhalt der Spezifikation zu erschließen. Bei diesem Vorgang überprüft der Generator auch die Korrektheit der Spezifikation.

2. Nachdem die Korrektheit der Spezifikation geprüft wurde, wird sie durch den Generator vervollständigt. Ergänzt werden dabei sowohl plattformspezifische Aspekte, darunter fallen beispielsweise die Deploymentdeskriptoren einer J2EE-Anwendung, als auch standardisierte Implementierungsaspekte, die in allen Systemen der Anwendungsdomäne enthalten sind und die deshalb in der abstrahierten Darstellungsform der domänenspezifischen Spezifikationssprache nicht explizit modelliert werden. In der Domäne der Versionierungssysteme müssen beispielsweise die Deklaration und die Implementierung der Versionierungsoperationen für alle versionierten Objekttypen ergänzt werden.

3. Die vervollständigte Spezifikation kann optional in weiteren Transformationsschritten optimiert werden. Der Schwerpunkt liegt hierbei auf *domänenspezifischen Optimierungen*, die nur mit der Kenntnis der gesamten Spezifikation bzw. von Domänenwissen durchgeführt werden können und daher für einen nachgeschalteten herkömmlichen Compiler nicht ersichtlich sind. Dabei kann es sich beispielsweise um die gezielte Auswahl eines Algorithmus für eine Operation abhängig von der Konfiguration anderer Modellelemente handeln. Ein herkömmlicher Compiler könnte nur die Durchführung der Operation optimieren, aber nicht den Algorithmus selbst verändern.

4. Abschließend wird die vollständige Systemspezifikation, die nun im internen Datenformat des Generators vorliegt, in das gewünschte Ausgabeformat umgewandelt. Dies kann entweder, wie bei einem herkömmlichen Compiler, direkt ausführbarer Byte- bzw. Maschinencode sein oder aber Programmcode in einer höheren, anwendungsunabhängigen Programmiersprache. Dieser Programmcode kann dann durch den Entwickler entweder weiter manuell verändert werden, oder direkt durch einen entsprechenden Compiler weiterverarbeitet werden.

Diese vier dargestellten Verarbeitungsschritte müssen nicht zwingend getrennt voneinander und in dieser Reihenfolge durchgeführt werden. Abhängig von der Art der durchgeführten Transformationen können einerseits mehrere Arbeitsschritte zusammengefasst werden, oder es kann andererseits erforderlich sein, Arbeitsschritte mehrmals iterativ durchzuführen.

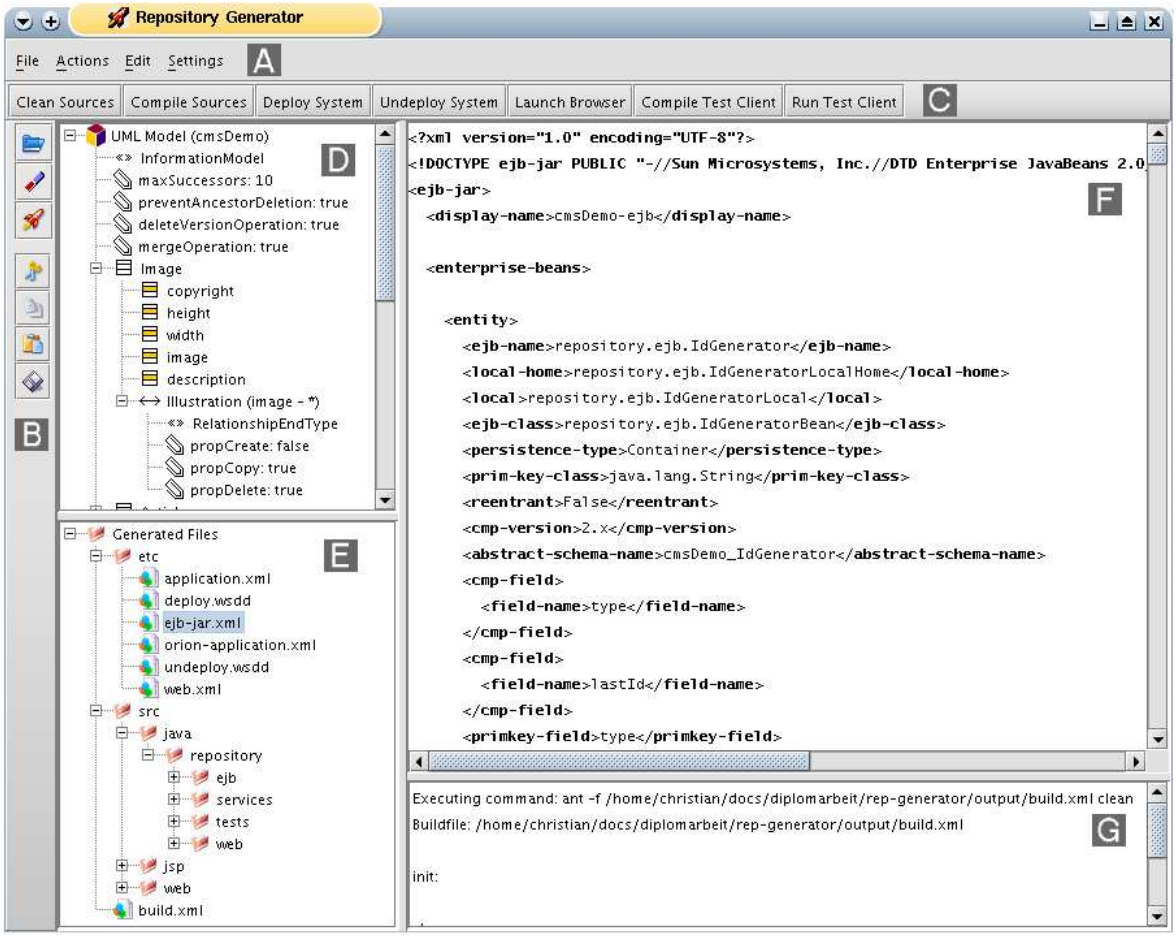
In diesem Kapitel wird zunächst der Generator vorgestellt, der im Rahmen dieser Arbeit entwickelt wurde. Dabei wird insbesondere auf den gewählten vorlagenbasierten Generierungsansatz und auf die Aufbereitung der Modelldaten eingegangen. Für diesen Generator wurden *Vorlagen* entwickelt, die die im letzten Kapitel gewonnene domänenspezifische Darstellung eins Versionierungssystems in eine Middleware-basierte Implementierung umsetzen. Hierzu wird die gewählte Anwendungsarchitektur und ihre Umsetzung auf der Basis von J2EE und Webservices beschrieben.

4.1 Aufbau und Funktionsprinzip von *RepGen*

Der Generator *RepGen* bietet dem Softwareentwickler eine Umgebung, in der alle Arbeitsschritte, die eine domänenspezifischen Spezifikation eines Softwaresystems in eine lauffähige Implementierung überführen, vorgenommen werden können.

Die Spezifikation des Softwaresystems wird nicht innerhalb von *RepGen* entwickelt, sondern von einem externen UML-Modellierungsprogramm übernommen. Zum Transfer der Modelldaten wird das durch die OMG standardisierte XMI-Format [OMG02d] verwendet, da dieses von allen gängigen Modellierungsprogrammen erzeugt werden kann. Der XMI-Standard definiert für MOF-konforme Metamodelle, und damit auch für das hier relevante

Abbildung 4.1 Die grafische Benutzeroberfläche von *RepGen*



UML-Metamodell, eine Abbildung auf ein XML-Schema. Modelle, die diesen Metamodellen entsprechen, können mit Hilfe des im Standard definierten Verfahrens auf XML-Dokumente abgebildet werden und aus diesen auch wieder rekonstruiert werden.

Der in die Entwicklungsumgebung integrierte vorlagenbasierte Generator ist vielfältig anpassbar. Einerseits kann er durch den Austausch der verwendeten Vorlagen an verschiedene Anwendungdomänen und darauf bezogenen Spezifikationssprachen angepasst werden. Andererseits werten die Vorlagen zusätzlich zu den Modelldaten Konfigurationswerte aus, die plattformspezifische Information enthalten, wie beispielsweise die Installationspfade benötigter Bibliotheken und Hilfprogramme. Sowohl die Auswahl der passenden Generatorvorlagen als auch die Festlegung der Konfigurationswerte kann der Entwickler innerhalb der Entwicklungsumgebung vornehmen.

Auch über den eigentlichen Generierungsprozess hinaus unterstützt *RepGen* den Entwickler. Die Ergebnisse der Generierung können innerhalb der Entwicklungsumgebung begutachtet werden. Falls die Ergebnisse nicht den Wünschen des Entwicklers entsprechen, kann er entscheiden, ob er die Generierung mit einer geänderten Spezifikation oder einer angepassten Generatorkonfiguration wiederholt, oder ob er die generierten Dateien manuell mit Hilfe

des integrierten Editors verändert (forward engineering). Auch die anschließende Übersetzung, die Installation und der Test des Softwaresystems kann direkt innerhalb von *RepGen* durchgeführt werden.

Gliederung der Benutzeroberfläche

Abbildung 4.1 enthält ein Bildschirmfoto der Benutzeroberfläche von *RepGen*, in dem die einzelnen Elemente der der Oberfläche durch Buchstaben markiert worden sind.

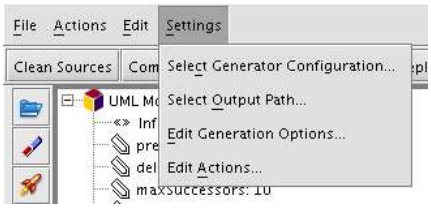
Über die Menüleiste (**A**) und die vertikale Symbolleiste (**B**) steuert der Benutzer den Generierungsprozess und den integrierten Quellcode-Editor. Über die horizontale Symbolleiste (**C**) können die integrierten Aktionen zur Übersetzung, Installation und zum Test des generierten Softwaresystems gestartet werden.

Das Hauptfenster von RegGen ist in vier Regionen aufgeteilt, die verschiedene Sichten auf die verarbeitete Spezifikation und das erzeugte Softwaresystem zeigen. Die Region (**D**) visualisiert das geladene UML-Modell. Anhand der Auflistung der für die Generierung relevanten Modellelemente kann der Benutzer einerseits überprüfen, dass er die richtige Version der Systemspezifikation geladen hat und andererseits das Verhalten des Generators durch einen Vergleich der Spezifikation mit den generierten Dokumenten nachvollziehen. Die Anzeige umfasst die UML-Klassen mit ihren Attributen und Methoden sowie die Assoziationen zwischen diesen Klassen. Außerdem werden zu allen Modellelementen die zugeordneten Stereotypes und Tagged Values angezeigt, da diese einen starken Einfluss auf den Inhalt der generierten Dateien haben. In Abbildung 4.1 ist beispielsweise zu erkennen, dass das Assoziationsende „image“ der Assoziation „Illustration“ den Stereotype „RelationshipEndType“ besitzt und der verknüpfte Tagged Value vom Typ „propCreate“ den Wert „false“ hat.

Die Region (**E**) enthält die Liste der durch den Generator erzeugten Dateien. Wenn der Benutzer eine der aufgelisteten Dateien auswählt, wird sie im integrierten Editor (**F**) geöffnet. Auf diese Weise kann der Benutzer die Ergebnisse der Generierung direkt begutachten und bei Bedarf manuell verändern.

Unterhalb des Editors befindet sich eine Region (**G**), in dem der Benutzer über die Ausgaben der über die Symbolleiste (**C**) gestarteten Aktionen informiert wird. Er kann diesem Feld also beispielsweise entnehmen, welche Fehler der aufgerufene externe Compiler im Quellcode des generierten Softwaresystem gefunden hat.

Abbildung 4.2 Das „Settings“ Menü



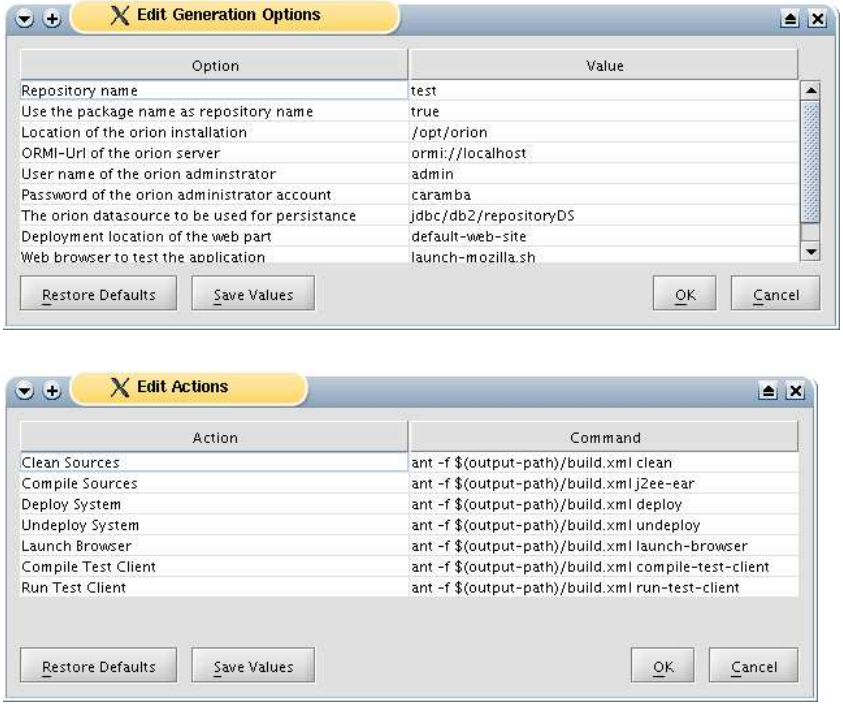
Konfiguration des Generators

Die Funktionen zur Konfiguration des Generators sind im Menü „Settings“ enthalten, das in Abbildung 4.2 zu sehen ist. Über den Menüpunkt „Select Generator Configuration...“ kann

der Benutzer eine Generatorkonfiguration auswählen. Die Generatorkonfiguration bestimmt, welche Vorlagen zur Generierung verwendet werden. Die Wahl der Generatorkonfiguration beeinflusst ausserdem, welche plattformspezifischen Konfigurationswerte durch die Vorlagen ausgewertet werden und welche Aktionen zur Übersetzung, Installation und zum Test der generierten Systems zur Verfügung stehen.

Die plattformspezifischen Konfigurationswerte und Aktionen können über die Menüpunkte „Edit Generation Options...“ bzw. „Edit Actions...“ an die Gegebenheiten der lokalen Softwareumgebung anpasst werden. Abbildung 4.3 enthält Bildschirmfotos der Dialoge, in denen dieses Einstellungen vorgenommen werden.

Abbildung 4.3 Konfigurationsdialoge



4.1.1 Vorlagenbasierte Generierung

Für die Generierung wurde von uns der erprobte Ansatz der *vorlagenbasierten Generierung* gewählt. Die von uns verwendete Variante dieses Ansatzes wird in [SVB02] beschrieben und in dieser Form in vielen Werkzeugen zur Codegenerierung erfolgreich angewendet.

Die vorlagenbasierte Generierung arbeitet mit *Vorlagen*, die die statischen Anteile der zu generierenden Dokumente beinhalten. Zur Festlegung der dynamischen Anteile werden die Vorlagen mit den Konstrukten einer Skriptsprache angereichert, die durch den Generator ausgewertet werden. Mit Hilfe der Skriptsprache können Platzhalter definiert werden, die der Generator bei der Verarbeitung der Vorlage durch tatsächliche Modelldaten ersetzt. Es können aber auch komplexere Kontrollstrukturen, wie etwa Fallunterscheidungen und Schleifen, verwendet werden, wodurch beliebig komplexe Transformationen der Modelldaten realisiert werden können. Die Ausgabe des Generators setzt sich also aus den statischen Anteilen der Vorlage und den Ergebnissen der Skriptsprachen-Anweisungen zusammen.

Die vorlagenbasierte Generierung bietet einige Vorteile gegenüber anderen Generierungsansätzen, wie etwa einer ad-hoc Implementierung mit Hilfe der Stringverarbeitungsroutinen einer beliebigen Programmiersprache:

- Da die Vorlagen die statischen Anteile der generierten Dokumente im Klartext enthalten, ist die *Struktur* der generierten Dokumente aus den Vorlagen gut erkennbar. Die Vorlagen sind deshalb leicht verständlich und können vergleichsweise einfach gewartet und weiterentwickelt werden.
- Bei einer geeigneten Aufbereitung der Modelldaten vor der Anwendung der Vorlagen kann die Verarbeitungslogik innerhalb der Vorlagen auf ein Minimum reduziert werden. Auf diese Weise entsteht, dem MVC-Muster [KP88] entsprechend, eine klare Trennung zwischen der Steuerung der Generierungsprozesses und der Datendarstellung. Diese Trennung erhöht die Wartbarkeit der Vorlagen zusätzlich.
- Da der in den Vorlagen enthaltene Skriptsprachen-Code durch den Generator zur Laufzeit interpretiert wird, bedürfen Änderungen an den Vorlagen keiner erneuten Übersetzung des gesamten Generatorprogramms. Die vorlagenbasierte Generierung unterstützt also kurze Zyklen bei der Entwicklung der Vorlagen.

Velocity

Für die Implementierung von *RepGen* wurde die Vorlagenverarbeitung nicht neu entwickelt, sondern die durch das Apache-Projekt entwickelte *Velocity Template Engine* [Apa03a] verwendet. Velocity besteht aus einer Skriptsprache (Velocity Template Language - VTL) mit einem kompakten und gut lesbaren Syntax und einem passenden Prozessor für Vorlagen, die in VTL geschrieben sind.

Abbildung 4.4 Vorlagenverarbeitung durch Velocity

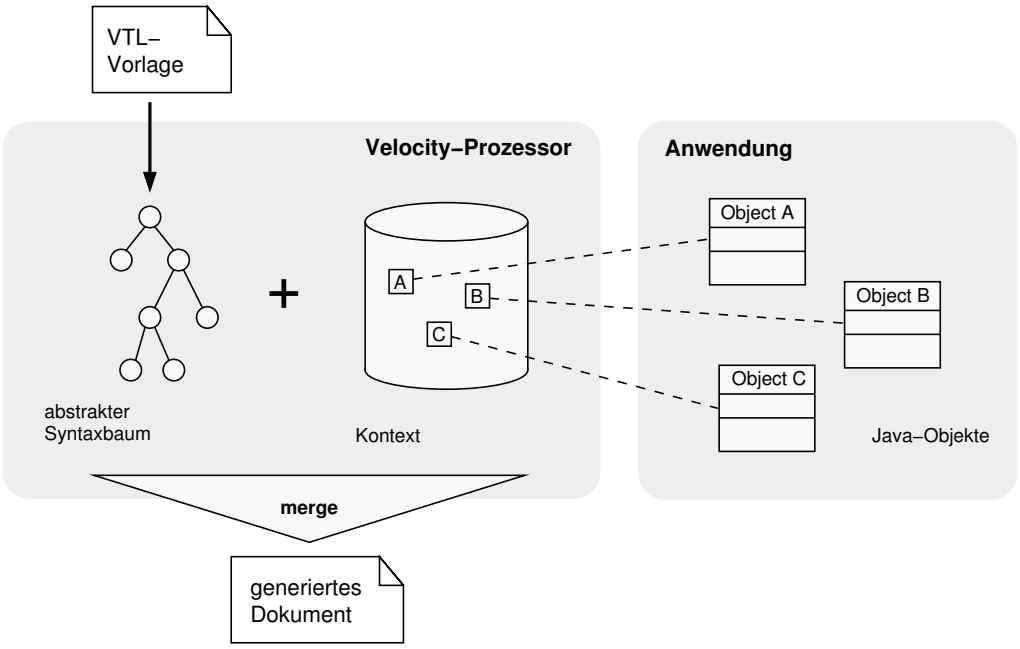


Abbildung 4.4 zeigt schematisch die Vorlagenverarbeitung durch Velocity. Nach der Initialisierung des Prozessors wird auf Veranlassung der Anwendung zunächst eine Vorlage geladen und das darin enthaltende *Metaprogramm* in der Skriptsprache VTL einer Syntaxanalyse unterzogen. Die Vorlage kann dabei aus dem Dateisystem gelesen werden oder aber auch die Anwendung dynamisch erzeugt werden. Ergebnis der Syntaxanalyse ist ein *abstrakter Syntaxbaum* des Metaprogramms, der die Grundlage für die weitere Verarbeitung der Vorlage durch Velocity bildet.

Typischerweise referenziert ein Metaprogramm Anwendungsdaten, um sie in die generierten Dokumente zu integrieren. Deshalb muss dem Velocity-Prozessor vor der Ausführung eines Metaprogramms mitgeteilt werden, welchen konkreten Anwendungsdaten den im Metaprogramms enthaltenen Referenzen zugeordnet werden sollen. Dazu dient der *Kontext*, der eine bidirektionale Schnittstelle zwischen dem Metaprogramm und der Anwendung bildet. Die Anwendung kann im Kontext Java-Objekte unter einem Schlüsselwort ablegen. Diese Java-Objekte kapseln die Anwendungsdaten, die für die Verarbeitung der Vorlage benötigt werden. Im Fall von *RepGen* werden also die für diese Anwendung der Vorlage relevanten Modelldaten im Kontext abgelegt. Der Prozessor kann nun während der Ausführung des Metaprogramms Referenzen auflösen, indem er überprüft, ob im Kontext ein Objekt unter dem Namen der Referenz abgelegt wurde. Wenn dies der Fall ist, kann das Metaprogramm über die Schnittstelle des Objekts an die Anwendungsdaten gelangen. Auch umgekehrt können im Metaprogramm neue Java-Objekte erzeugt werden und im Kontext abgelegt werden. Nach der Ausführung des Metaprogramms kann die Anwendung auf diese Weise Information über die Verarbeitung der Vorlage erhalten.

Die Ausführung des Metaprogramms wird durch Velocity als *merge* bezeichnet, da sie den Syntaxbaum mit den im Kontext enthaltenen Anwendungsdaten zum generierten Dokument vereinigt.

Der Syntax der von Velocity verwendeten Skriptsprache VTL soll nun anhand der in Beispiel 4.1 enthaltenen Vorlage erläutert werden. Diese Vorlage generiert eines Java-Interfaces, das für jeden Objekttyp eines Versionierungssystems generiert werden muss. Wie bereits erwähnt wurde, ist die Struktur der generierten Schnittstelle aus dem Quellcode der Vorlage direkt ersichtlich. Die Referenzen auf im Kontext befindliche Modelldaten werden durch die Notation `${Name.Attribut/Methode}` gekennzeichnet. Diese Platzhalter werden bei der Verarbeitung der Vorlage durch die tatsächlichen Daten ersetzt. Im Fall dieser Vorlage beziehen sich alle Referenzen auf die UML-Klasse, die die Spezifikation des Objekttyps bildet. Ein entsprechendes Datenobjekt wird durch die Generatorsteuerung unter dem Namen `class` im Velocity-Kontext abgelegt.

Die Vorlage enthält neben einfachen Platzhaltern für Modelldaten auch mehrere Fallunterscheidungen und eine Schleife, die durch die Schlüsselwörter `#if`, `#foreach` und `#end` definiert werden. Durch die erste Fallunterscheidung wird erreicht, dass nur für versionierte Objekttypen die Definition der Methode `create(int objectId)` eingefügt wird. Versionierte Objekttypen sind daran zu erkennen, dass die UML-Klasse den Stereotype `VersionedObjectType` oder aber den Stereotype `WorkspaceType` besitzt, deshalb überprüft die Fallunterscheidung mit Hilfe der Methode `hasStereotype`, ob einer dieser Stereotypes vorhanden ist. Die Schleife am Ende der Vorlage iteriert über alle Assoziationsenden der UML-Klasse, die sie von der Methode `associationEnds` erhält. Dadurch wird erreicht, dass für alle Arbeitskontexttypen in denen ein Objekttyp enthalten ist, entsprechende Suchmethoden in die Schnittstelle des Objekttyps eingefügt werden.

Beispiel 4.1 Beispiel einer Velocity-Vorlage

```
package repository.ejb;

import java.lang.Integer;
import java.util.*;
import javax.ejb.*;

/**
 * local home interface for the entity bean that
 * stores objects of the type ${class.name}
 */
public interface ${class.name}LocalHome extends EJBLocalHome {

  // create a new object of the type ${class.name}
  public ${class.name}Local create() throws CreateException;

#if ( $class.hasStereotype("VersionedObjectType") ||
      $class.hasStereotype("WorkspaceType") )
  // create a new object version for the given objectId
  public ${class.name}Local create(int objectId) throws CreateException;
#end

  // locate existing objects...
  public ${class.name}Local findByPrimaryKey(Integer primaryKey)
                            throws FinderException;
  public Collection findAll() throws FinderException;
  public Collection findByObjectId(int objectId) throws FinderException;

#if ( !$class.hasStereotype("WorkspaceType") )
#foreach ( $associationEnd in $class.associationEnds )
#if ( $associationEnd.association.hasStereotype("AttachmentType") )
  public Collection findAllInWorkspace${associationEnd.oppositeEnd.nameUpperCase}
                    (int wsGlobalId) throws FinderException;
  public Collection findObjectInWorkspace${associationEnd.oppositeEnd.nameUpperCase}
                    (int wsGlobalId, int objectId) throws FinderException;
#end
#end
#end
}
```

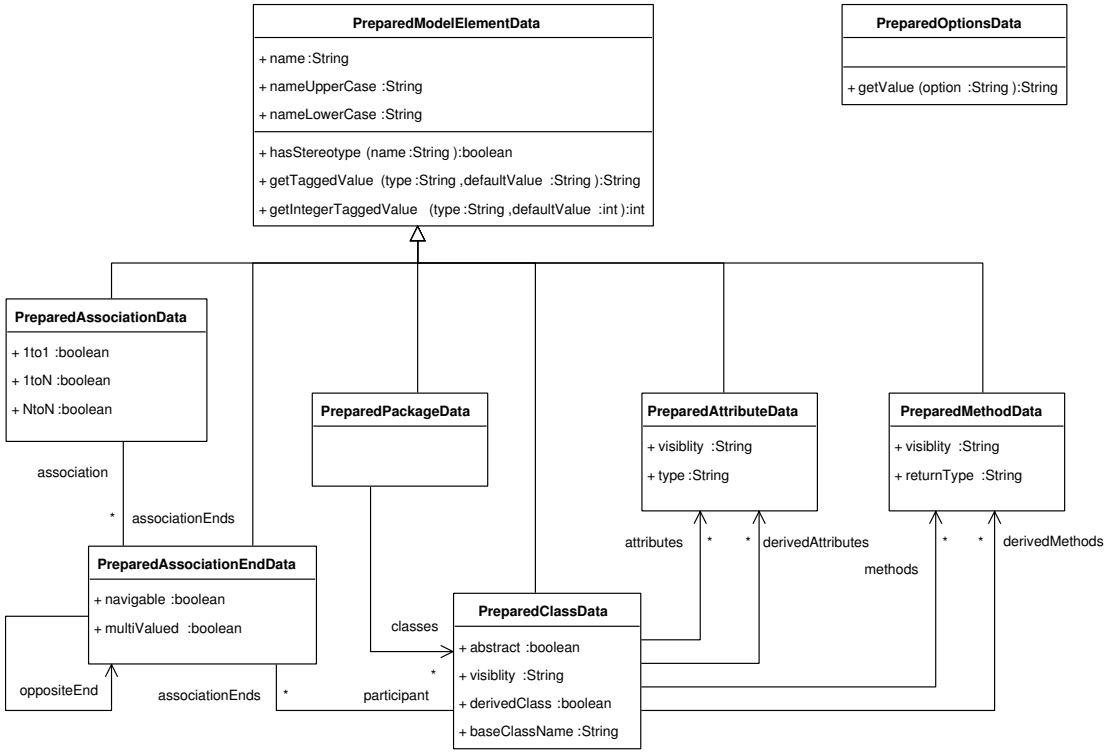
4.1.2 Aufbereitung der Modelldaten

Das zu verarbeitende UML-Modell muss aus einem XMI-Dokument eingelesen werden und der Vorlagenverarbeitung mittels einer geeigneten Schnittstelle zur Verfügung gestellt werden. Für diese sehr generische Funktionalität bietet es sich an, eine bestehende Softwarekomponente wiederzuverwenden. Die Entscheidung fiel auf eine Implementierung des durch den Java Community Process definierten Java Metadata Interface (JMI) [JCP02]. Diese Schnittstelle soll Anwendungen einen einfachen Zugriff auf Metadaten zu den von ihnen verarbeiteten Daten geben und auch den Austausch von Metadaten zwischen Anwendungen unterstützen. Zu diesem Zweck wurde die MOF-Spezifikation auf Java abgebildet. Das bedeutet, dass für MOF-konforme Metamodelle sowohl eine generische als auch eine metamodellspezifische Java-Schnittstelle definiert wird. Ausserdem wird der Im- und Export von Metadaten mit Hilfe von XMI-Dokumenten definiert. Eine Implementierung der JMI-Spezifikation ermöglicht einer Anwendung, MOF-Modelle einzulesen, diese zu verändern, neue Modelle zu erstellen und diese auch wieder abzuspeichern. Diese Funktionalität wird

typischerweise durch ein *in-memory Repository* realisiert, das die Modelldaten während der Bearbeitung durch die Anwendung im Hauptspeicher verwaltet. Es existieren bereits mehrere Implementierungen der JMI-Spezifikation. Für *RepGen* wurde das *Novosoft Metadata Framework* [Nov02] ausgewählt. Es beinhaltet bereits eine vorbereitete Bibliothek für das UML 1.4 Metamodell (NS-UML).

Die Schnittstelle der NS-UML-Bibliothek besteht, der JMI-Spezifikation entsprechend, aus Java-Beans die jeweils die im UML-Metamodell definierten Attribute eines Modellelements verwalten. Auch die Vererbungshierarchie und die Beziehungen zwischen den einzelnen Modellelemente wurde aus dem UML-Metamodell übernommen. Die Beans bilden deshalb einen Objektgraphen, in dem mit Hilfe der entsprechenden Bean-Methoden navigiert werden kann.

Abbildung 4.5 Zugriffsschicht für die Modelldaten



Um die Logik der Vorlagen möglichst einfach zu halten, soll diesen aber nicht der gesamte Objektgraph über den Velocity-Kontext zur Verfügung gestellt werden, sondern nur ein für sie relevanter Ausschnitt. Ausserdem soll die UML-Daten für die Ausgaben in den Vorlage zunächst aufbereitet werden. Dazu dient die in Abbildung 4.5 dargestellte Zugriffsschicht. Diese Klassen kapseln sowohl die Inhalte, als auch die Beziehungsstruktur der NS-UML-Klassen, die Vorlagen kommen also mit der eigentlichen NS-UML-Bibliothek nicht mehr direkt in Kontakt. Die zusätzliche Zugriffsschicht bietet folgende Vorteile:

- Das UML-Metamodell und somit auch die Schnittstelle der Zugriffsschicht kann vereinfacht werden, indem Teile des Metamodells entfernt werden, die für die Vorlagen nicht relevant sind. Die im Rahmen dieser Arbeit entwickelten Vorlagen verwenden beispielsweise im Wesentlichen nur eine Ausschnitt des UML-Core-Packages.

- Ein Objekt der Zugriffsschicht kann die Daten mehrerer verknüpfter NS-UML-Objekte kapseln. Dadurch muss in der Vorlage nicht mehr explizit zwischen diesen Objekten navigiert werden. Beispielsweise greift die in Abbildung 4.5 dargestellte Klasse `PreparedAssociationEndData` nicht nur auf Modellelemente des Typs `AssociationEnd` zu, sondern zusätzlich auch auf verknüpfte Elemente der Typen `Multiplicity`, `MultiplicityRange`, `Association`, `Stereotype`, `TagDefinition` und `TaggedValue`. Die Tabelle 4.1 enthält eine Aufstellung der durch die Zugriffsschicht gekapselten Klassen des UML-Metamodells. Die Navigation in den Modelldaten kann ausserdem durch zusätzliche Beziehungen vereinfacht werden, die nicht Teil des UML-Metamodells sind. Diese verkürzen die Navigation zwischen Modellelementen, deren Elementtypen im Metamodell nur in einer indirekten Beziehung stehen.
- Die Zugriffsschicht kann neben reinen Zugriffsmethoden für die Modelldaten auch Methoden anbieten, die diese Daten für die Ausgabe in den Vorlagen bzw. für darin enthaltene Fallunterscheidungen speziell aufbereiten.

Tabelle 4.1 Aufstellung der durch die Zugriffsschicht gekapselten UML-Klassen

Klasse der Zugriffschicht	Klasse des UML-Metamodells
PerparedAssociationData	Association, Stereotype, TagDefinition, TaggedValue
PerparedAssociationEndData	AssociationEnd, Multiplicity, MultiplicityRange, Association, Stereotype, TagDefinition, TaggedValue
PreparedAttributeData	Attribute, Classifier, Namespace, Stereotype, TagDefinition, TaggedValue
PreparedClassData	Class, Generalization, Stereotype, TagDefinition, TaggedValue
PreparedMethodData	Method, Operation, Parameter, Classifier, Namespace, Stereotype, TagDefinition, TaggedValue
PreparedModelElementData	ModelElement, Stereotype, TagDefinition, TaggedValue
PreparedPackageData	Package, Stereotype, TagDefinition, TaggedValue

Nachfolgend sollen exemplarisch einige der Methoden aus der Zugriffsschicht vorgestellt werden, die sich nicht direkt aus dem UML-Metamodell ergeben, sondern die Daten des gekapselten UML-Elements und verknüpfter Elemente aufbereiten und damit die Logik der Vorlagen vereinfachen:

- *PreparedModelElementData.hasStereotype(String name)* - Die Zuordnung eines Stereotypes zu einem Modellelement verändert dessen Semantik. Deshalb enthalten Vorlagen häufig Fallunterscheidungen um abhängig vom zugeordneten Stereotype die passende Implementierung auszugeben. Diese Methode überprüft, ob dem Modellelement ein Stereotype mit dem übergebenen Namen zugeordnet ist. Dazu iteriert die Methode über die verknüpften Stereotype-Elemente und vergleicht deren Namen mit dem Eingabeparameter.
- *PreparedModelElementData.getTaggedValue(String type, String defaultValue)* - Auch die einem Element zugeordneten Tagged Values müssen häufig in den Vorlagen ausgewertet werden. Diese Methode ermöglicht es den Vorlagen, mit einem einzigen Methodenaufruf die Belegung eines Tagged Values in Erfahrung zu bringen. Dazu muss für die assoziierten Tagged Value-Elemente überprüft werden, ob sie den richtigen Typ haben, indem der

Name ihrer Tag Definition mit dem übergebenen `type`-String verglichen wird. Falls ein passender Tagged Value vorhanden ist, liefert die Methode dessen Wert zurück, ansonsten verwendet sie den durch die Vorlage übergeben Standardwert.

- *PreparedAssociationData: is1to1(), is1toN(), isNtoN()* - Für die Generierung des Quellcodes zur Verwaltung der Beziehungen innerhalb eines Versionierungssystems muss häufig zwischen den drei Fällen einer 1:1, 1:n oder n:m Beziehung unterschieden werden. Für diese Entscheidung müssen die Kardinalitäten der Assoziationsenden überprüft werden, die der Assoziation zugeordnet sind. Die drei Methoden der Klasse `PreparedAssociationData` kapseln die notwendige Logik und ermöglichen so eine kompakte Darstellung der Fallunterscheidung im VTL-Quellcode der Vorlage.
- *PreparedAssociationEndData.getOppositeEnd()* - Diese Methode ermöglicht die direkte Navigation von einem Ende einer Assoziation zum gegenüberliegenden Ende. Diese Navigation muss durch die Vorlagen häufig durchgeführt werden, da beispielsweise bei der Generierung von Zugriffsmethoden für Beziehungen der Name des gegenüberliegenden Assoziationsendes bekannt sein muss. Das UML-Metamodell sieht aber nur eine Beziehung zwischen den Assoziationsenden und der Assoziation vor, ohne die Hilfe der Methode `getOppositeEnd` müsste also zunächst vom Assoziationende zur zugehörigen Assoziation navigiert werden und von dort zum gegenüberliegenden Assoziationsende.

Eine Sonderrolle spielt die in Abbildung 4.5 ebenfalls sichtbare Klasse `PreparedOptionsData`. Diese Klasse kapselt keine Modelldaten, sondern ermöglicht den Zugriff auf die bereits beschriebenen plattformspezifischen Konfigurationswerte. Dies geschieht über die Methode `getValue(String name)`, die zu dem gegebenen Optionsnamen den durch den Benutzer konfigurierten Wert zurückliefert.

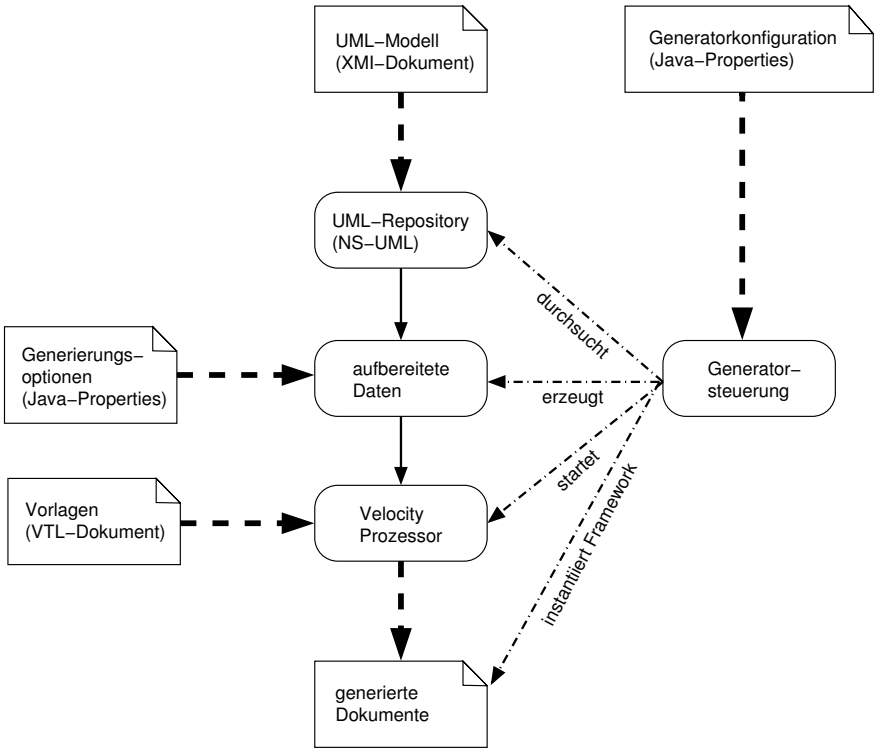
4.1.3 Generatorsteuerung

Der Ablauf der Generierung muss gesteuert werden. Die Abbildung 4.6 gibt einen Überblick über die Aufgaben der Generatorsteuerung. Da *RepGen* ein generischer Softwaregenerator ist, ist sein Verhalten abhängig von dem Inhalt der Generatorkonfiguration, die einem Java-Properties-Dokument definiert wird. Vor dem Beginn des eigentlichen Generierungsvorgangs instantiiert die Steuerung zunächst ein in der Konfiguration spezifiziertes *Framework*. Dabei legt sie durch das Entpacken eines Dateiarchivs die Verzeichnisstruktur des zukünftigen Softwaresystems an und kopiert Dateien in diese Verzeichnisstruktur, die nicht generiert werden müssen, da sie für alle Produktlinienmitglieder identisch sind. Anschließend werden die übrigen Dateien des Softwaresystems generiert, indem die in der Konfiguration definierten *Generierungsziele* abgearbeitet werden.

Ein Generierungsziel legt fest, mit welchen Modellinformationen und welcher Vorlage eine Datei des Softwaresystems erzeugt werden soll. Deshalb besteht der erste Arbeitsschritt der Steuerung darin, im UML-Modell nach den Modellelementen zu suchen, die den Kriterien des Generierungsziels entsprechen. Wenn diese gefunden worden sind, werden sie für die Verarbeitung in der Vorlage aufbereitet, indem die Objekte der UML-Bibliothek in Objekten der im letzten Abschnitt beschrieben Zugriffsschicht gekapselt werden.

Wenn die Steuerung alle nötigen Daten gesammelt und aufbereitet hat, legt sie die Objekte im Kontext des Velocity-Prozessors ab und startet den Prozessor mit der passenden Vorlage. Die Ausgabe des Prozessors wird unter dem durch das Generierungsziel spezifizierten Dateinamen abgespeichert.

Abbildung 4.6 Schematische Darstellung des Generierungsprozess

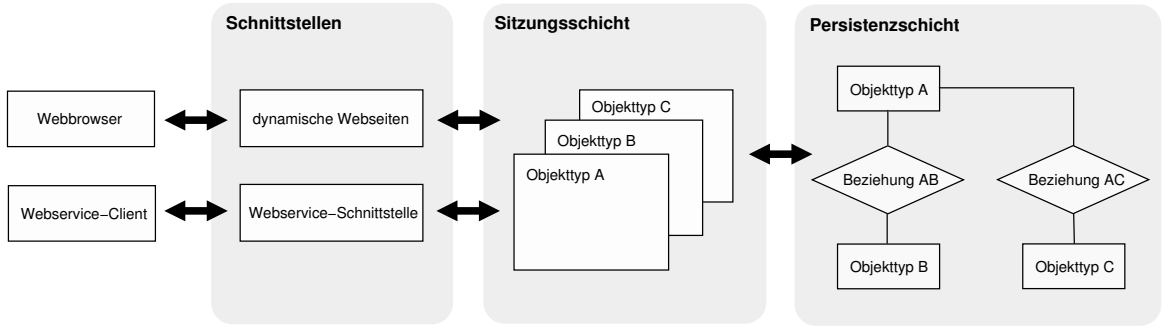


4.2 Generierung Middleware-basierter Versionierungssysteme

Der in diesem Kapitel beschriebene Generator erlaubt es nun, den Quellcode von Versionierungssystemen zu generieren, die mit Hilfe der in Kapitel 3 entwickelten domänenspezifischen Spezifikationssprache modelliert worden sind. Die Zielsetzung war, ein Middleware-basierte Implementierung zu entwickeln, die über webbasierte Systemschnittstellen verfügt und deren Design die Optimierungsmöglichkeiten ausnutzt, die der generative Ansatz im Vergleich zu einer generischen Komponentenlösung bietet. Dabei handelt es sich hauptsächlich um größere strukturelle Optimierungen, die mehrere Komponenten der Implementierung betreffen und deshalb nur schwer manuell durchführbar sind.

Abbildung 4.7 zeigt die gefundene Systemarchitektur. Die Architektur teilt das Gesamtsystem in drei Schichten auf. Die Persistenzschicht ist für die Verwaltung der Daten zuständig. Hier werden die Attributwerte der Objekttypen und die Versionierungs- und Beziehungsdaten in einem speziell an das Informationsmodell des Versionierungssystems angepassten Datenbankschema verwaltet. Diese Schicht bietet eine objektorientierte Schnittstelle, die der übergeordneten Sitzungsschicht den Zugriff auf die gespeicherten Objekte und Beziehungen ermöglicht. Die Komponenten der Persistenzschicht enthalten aber keine Anwendungslogik, den diese wird exklusiv in der Sitzungsschicht implementiert. Die Sitzungsschicht besteht aus Komponenten, die jeweils die Schnittstelle für einen Objekttyp des Informationsmodelle implementieren. Diese Komponenten haben eine Vielzahl von Funktionen:

Abbildung 4.7 Systemarchitektur der generierten Versionierungssysteme



- Über ihre Schnittstelle können Objekte des ihnen zugeordneten Objekttyps erzeugt, aufgefunden und gelöscht werden. Weiterhin ermöglichen die Komponenten den Zugriff auf die Attributwerte und die Beziehungen der Objekte.
- Sie implementieren die im Merkmalmodell definierte Semantik der Versionierungssysteme und überwachen die Einhaltung der im Informationsmodell festgelegten Konsistenzbedingungen.
- Sie bilden die objektorientierte Sicht der Persistenzschicht auf eine prozedurale Schnittstelle ab. Die vorgesehen webbasierten Systemschnittstellen können prinzipbedingt nicht objektorientiert ausgelegt werden, deshalb ist es sinnvoll, diese Abbildung an zentraler Stelle vorzunehmen.
- Die Komponenten verwalten außerdem den Sitzungskontext der Clients. Dies ermöglicht beispielsweise die Implementierung von Transaktionen, die sich über mehrere Aufrufe der Sitzungskomponenten durch den Client erstrecken können.

Die äußerste Schicht der generierten Versionierungssysteme bilden die webbasierten Systemschnittstellen. Diese bestehen einerseits aus dynamischen Webseiten, die einem menschlichen Benutzer den Zugriff auf das Versionierungssystem mit Hilfe eines Webbrowsers ermöglichen. Andererseits wird externen Softwaresystemen die Möglichkeit geboten, auf die Daten des Versionierungssystems über eine Webservice-Schnittstelle zuzugreifen.

Als Middlewaretechnologie für die Implementierung der Versionierungssysteme wurde die J2EE-Plattform [Sun01] ausgewählt. J2EE bietet für alle Aspekte der Implementierung, insbesondere im Bereich der Persistenzschicht, adäquate Unterstützung. Aus der Sammlung der Technologien, die die J2EE-Plattform bilden, wurden Enterprise JavaBeans (EJB) [JCP03c] für die Implementierung der Sitzungs- sowie der Persistenzschicht ausgewählt. Die webbasierten Systemschnittstellen wurden auf der Basis von Java ServerPages (JSP) [JCP03a] bzw. Java Servlets [JCP03b] implementiert.

Eine J2EE-Anwendung läuft innerhalb einer durch einen *Anwendungsserver* zur Verfügung gestellten Laufzeitumgebung. Für die Entwicklung und den Test der Generierungsvorlagen wurde der Orion Application Server [Iro03] verwendet. Als Datenbanksystem für die Speicherung der Versionierungsdaten wurde die DB2 Universal Database [IBM03b] ausgewählt.

Da die Implementierung der Versionierungssysteme keine properitären Erweiterungen des J2EE-Standards verwendet, sind die generierten Anwendungen prinzipiell auch auf anderen

Anwendungsservern einsetzbar. Dafür muss lediglich der Installationsprozess der Anwendung an die neue Softwareumgebung angepasst werden.

Auf folgenden Seiten werden nun die einzelnen Bestandteile der Systemarchitektur detaillierter beschrieben. Dabei werden insbesondere die Unterschiede des Entwurfs im Vergleich zu einer generischen Implementierung erläutert.

4.2.1 Persistenzschicht

Die Grundlage für die gesamte Funktionalität eines Versionierungssystems bildet die versionierte Speicherung der Objektdaten. Sie wird im Falle der durch unsere Vorlagen erzeugten Systeme durch ein Datenbanksystem übernommen, das ein systemspezifisches Datenbankschema besitzt. Passend zu diesem Datenbankschema bietet die Persistenzschicht eine objektorientierte Schnittstelle, die die Datenbankzugriffe kapselt.

Diese Schnittstelle wurde mit Entity-EJBs realisiert. Entity-EJBs sind J2EE-Komponenten, die ihre Attributwerte persistent verwalten. Eine Instanz eines Entity-EJBs repräsentiert typischerweise ein einzelnes Datenbanktupel und wird dementsprechend auch über einen Primärschlüsselwert identifiziert. Entity-EJBs ermöglichen einen sehr effizienten Zugriff auf die Versionierungsdaten, da sie aus der Datenbank erhaltene Daten zwischenspeichern. Auf diese Weise können viele Anfragen an die Persistenzschicht ohne Datenbankzugriff bearbeitet werden.

Auch die Erzeugung neuer EJB-Instanzen ist sehr effizient implementiert. Diese werden in einem Pool vorgehalten und erst bei Bedarf mit einem Datenbanktupel assoziiert. Wenn eine Instanz nicht mehr benötigt wird, wird diese Assoziation aufgehoben und die EJB-Instanz wieder in den Pool der verfügbaren Instanzen aufgenommen. Dadurch müssen nicht für jeden Datenbankzugriff neue Java-Objekte erzeugt werden und somit sind auch Anfragen mit sehr großen Ergebnismengen problemlos durchführbar.

Die Speicherung der Attributwerte im verbundenen Datenbanksystem kann entweder individuell durch die Implementierung des Entity-Beans erfolgen (Bean Managed Persistance, BMP) oder aber durch den Anwendungsserver koordiniert werden (Container Managed Persistence, CMP). Die generierten Versionierungssysteme verwenden CMP, deshalb muss der Generator nur eine Definition des abstrakten Datenbankschemas in der Form des Deployment-Deskriptors und die Schnittstellen der Entity-EJBs generieren. Die Implementierung der Entity-EJBs hingegen wird durch den Anwendungsserver bei der Installation der Anwendung automatisch ergänzt. Gleiches gilt für die Verwaltung der Beziehungen zwischen Entity-EJBs. Hierfür verwendet die Implementierung Container Managed Relationships (CMR), weshalb auch in diesem Fall nur die Definition der Beziehungen im Deployment-Deskriptor und die entsprechenden Zugriffsmethoden generiert werden müssen.

Verwaltung von Objektversionen

Da bei einem generativen Ansatz keine Rücksicht auf die Wiederverwendbarkeit des Quellcodes der Zugriffsschicht genommen werden muss, ergibt sich die Struktur des Datenbankschemas direkt aus der Struktur des Informationsmodells. Für jeden Objekttyp wird eine separate Datenbanktabelle angelegt, in der alle Objekte in allen Versionen abgelegt werden. Diese Tabellen erhalten Felder für alle regulären Attribute des Objekttyps und zusätzlich, je nach der vorliegenden Konfiguration des Objekttyps, auch die im Merkmalmodell definier-

ten Versionierungsattribute wie beispielsweise den Versionsnamen. Die für die Tabellen notwendigen Schlüsselfelder wurden ebenfalls bereits im Merkmalmodell festgelegt. Das Feld `objectId` identifiziert alle Versionen eines Objekts, wohingegen das Feld `versionId` im Fall eines versionierten Objekts die einzelnen Versionen eines Objekts unterscheidet. Gemeinsam identifizieren sie eine Objektversion eindeutig. Diese Kombination ist im Feld `globalId` materialisiert, das als Primärschlüssel der Objekttabellen dient.

Bei einem generischen Ansatz wird das Datenbankschema hingegen soweit wie möglich generisch gestaltet, da für alle generischen Teile des Schemas keine systemspezifische Zugriffslogik implementiert werden muss. So wäre es beispielsweise denkbar, eine zentrale Tabelle einzuführen, die die gemeinsamen Attribute aller Objekttypen speichert. Diese würde die oben beschriebenen Schlüsselfelder besitzen und alle für die Versionierung benötigten Felder, die ja standardisiert sind. Zusätzlich würde sie ein Feld besitzen, das festhält, welchen Typ die jeweilige Objektversion hat. In dieser Tabelle würden für die Versionen aller Objekttypen Einträge eingefügt. Anhand der Typkennzeichnung und der `globalId` könnten dann die Attributwerte der Objektversion in einer gesonderten Tabellen gefunden werden, die für jeden Objekttyp eingerichtet wird. Konsequenz dieses Ansatzes ist, dass für jede Objektversion zwei Datenbanktupel verwaltet werden müssen, ausserdem kann die zentrale generische Objekttabelle einen sehr großen Umfang annehmen. Die Leistungsfähigkeit des resultierenden Versionierungssystems ist also potentiell geringer als die eines Versionierungssystems mit einem spezialisierten Datenbankschema.

Beispiel 4.2 Ausschnitt aus dem SQL-DDL-Schema zum CMS-Beispiel

```
create table Menu (globalId integer primary key,
                   objectId integer,
                   versionId integer,
                   versionName varchar(255),
                   frozen char(1),
                   name varchar(255),
                   description varchar(255),
                   checkoutEditorialDepartmentId integer,
                   superCategoryPinnedId integer,
                   superCategoryNewestId integer,
                   ancestorId integer)
create table Menu_mergedAncestors (globalId integer,
                                   targetId integer,
                                   primary key (globalId, targetId))
create table Menu_subCategory (globalId integer,
                               targetId integer,
                               primary key (globalId, targetId))
create table Menu_subCategoryPinned (globalId integer,
                                     targetId integer,
                                     primary key (globalId, targetId))
create table Menu_subCategoryNewest (globalId integer,
                                     targetId integer,
                                     primary key (globalId, targetId))
create table Menu_content (globalId integer,
                           targetId integer,
                           primary key (globalId, targetId))
create table Menu_contentPinned (globalId integer,
                                 targetId integer,
                                 primary key (globalId, targetId))
create table Menu_contentNewest (globalId integer,
                                 targetId integer,
                                 primary key (globalId, targetId))
```

Beispiel 4.2 zeigt einen Ausschnitt aus dem Datenbankschema, das für das bereits in den vorangegangen Kapiteln verwendete Beispiel eines Content-Management-Systems generiert wurde. Im Beispiel ist die Definition der Datenbanktabellen für den Objekttyp `Menu` zu sehen, wobei der SQL-DDL-Syntax verwendet wird. Wie beschrieben enthält die Tabelle `Menu` neben den drei Schlüsselfeldern `globalId`, `objectId` und `versionId` auch die Versionierungsattribute `versionName` und `frozen`, da für diesen Objekttyp in der Spezifikation sowohl spezifiziert wurde, dass er versioniert werden soll, als auch dass er ein Attribut für den Versionsnamen besitzen soll. Anschließend werden Felder für die regulären Attribute `name` und `description` definiert.

Um in der Sitzungsschicht die Versionierungssemantik implementieren zu können, müssen in der Persistenzschicht außer den Objektversionen auch die Beziehungen innerhalb des Versionsgraphen verwaltet werden. Zu diesem Zweck besitzt jeder versionierte Objekttyp eine Beziehung zwischen einer Objektversion und ihrer direkten Vorgängerversion, die durch die `createSuccessor`-Methode angelegt wird. Da jede Objektversion nur eine direkte Vorgängerversion besitzt, besitzt diese Beziehung eine Kardinalität von 1:n. Sie kann deshalb durch ein zusätzliches Feld `ancestorId` implementiert werden, das die `globalId` der Vorgängerversion als Fremdschlüssel enthält.

In die Schnittstellen der Entity-Komponenten werden für ein solches Beziehungsfeld spezielle Zugriffsmethoden eingefügt, die eine direkte Navigation von einer Objektinstanz zur nächsten für beide Richtungen der Beziehung unterstützt. Dabei kommt der Aufrufer nicht mit Werten der Fremdschlüssel in Berührung, sondern die Persistenzschicht führt für ihn die notwendigen Datenbankanfragen durch und liefert ihm die passende Objektinstanz zurück. Zusätzlich bietet die Persistenzschicht auch Methoden, die Beziehungen anlegen und entfernen. Diese Funktionalität der Persistenzschicht erleichtert die Implementierung der Anwendungslogik in der Sitzungschicht erheblich, da der Datenbestand praktisch wie ein normaler Objektgraph manipuliert werden kann ohne das Umsetzung der Beziehungen im Datenbankschema beachtet werden muss. Dieser Vorteil wird besonders deutlich bei n:m Beziehungen. Für diese muss nämlich eine separate Zwischentabelle eingeführt werden, in der für jede Beziehung ein Tupel mit den Primärschlüsseln der beteiligten Objekte eingefügt wird. Die Persistenzschicht ermöglicht auch in diesem Fall eine transparente Verwaltung der Beziehungen, die Zwischentabelle wird also im logischen Datenbankschema, das durch die Schnittstelle der Persistenzschicht definiert wird, nicht sichtbar.

Falls das Versionierungssystem die `merge`-Operation implementieren soll, muss eine zusätzliche Beziehung in das Datenbankschema eingefügt werden. Da einer Objektversionen beliebig viele Vorgängerversionen nachträglich zugeordnet werden können, ist dies eine n:m-Beziehung, die über eine separate Zwischentabelle realisiert werden muss. Es wäre alternativ auch möglich, die reguläre Beziehung zwischen Vorgänger- und Nachfolgerversion in eine n:m-Beziehung umzuwandeln, diese Lösung hat allerdings den Nachteil, dass nicht zwischen der ursprünglichen Vorgängerversion und den nachträglich hinzugefügten Vorgängerversionen unterschieden werden kann. In Beispiel 4.2 besitzt der Objekttyp `Menu` beide Beziehungen: In der Definition der Objekttabelle ist das Feld `ancestorId` für die Beziehung zur direkten Vorgängerversion enthalten und die Beziehung zu den nachträglich eingefügten Vorgängerversion wird durch die Zwischentabelle `Menu_mergedAncestors` realisiert.

Im Beispiel 4.3 ist ein weiterer Vorteil des generativen Ansatzes zu sehen. Die Beziehung zwischen einer Objektversion und ihren nachträglich eingefügten Vorgängerversionen wird

Beispiel 4.3 Bedingte Generierung der Beziehung zu nachträglich eingefügten Vorgängerversionen

```
if (!($package.getIntegerTaggedValue("maxSuccessors", -1) == 1) &&
      ($package.getTaggedValue("mergeOperation", "false") == "true"))
     <ejb-relation>
       <ejb-relation-name>
         ${class.name}-Ancestors-Successors-Merged
       </ejb-relation-name>
       ...
    </ejb-relation>
#end
```

nur dann generiert, wenn sie tatsächlich benötigt wird. Dazu wird geprüft, ob mehr als eine einzige Nachfolgeversion zulässig ist (`maxSuccessors`) und ob die merge-Operation implementiert werden soll (`mergeOperation`). Mit Hilfe der gleichen Fallunterscheidung wird auch innerhalb der Sitzungsschicht ein vereinfachter Algorithmus zu Navigation im Navigationgraphen ausgewählt, falls die zusätzliche Beziehung nicht benötigt wird. Eine generische Implementierung würde die Beziehung in jedem Fall enthalten müssen, da das System erst zu Laufzeit die Information erhält, ob die zusätzliche Beziehung benötigt wird.

Verwaltung von Arbeitskontexten

Für die Verwaltung der Arbeitskontexte in der Sitzungsschicht müssen ebenfalls Beziehungen in das Datenbankschema eingefügt werden. Einerseits muss gespeichert werden, zu welchem Arbeitskontext eine Objektversion zugeordnet ist. Dazu wird für Assoziation, die den Arbeitskontext mit einem enthaltenen Objekttypen verbindet und die in der UML-Spezifikation mit dem Stereotype `AttachmentType` gekennzeichnet wurde, eine Beziehung zwischen der Tabelle des Arbeitskontextes und der Tabelle der enthaltenen Objekttypen eingefügt. Die Kardinalität dieser Beziehungen ergibt sich aus der Kardinalität der UML-Assoziation. Anderseits muss auch gespeichert werden, an welchen Arbeitskontext eine Objektversion durch die `checkout`-Operation gebunden wurde. Hierfür wird zusätzlich zu jeder Attachment-Beziehung eine Checkout-Beziehung in das Datenbankschema eingefügt.

Verwaltung von Beziehungen

Die persistente Speicherung der regulären Beziehung des Informationsmodells ist unproblematisch, soweit es sich um Beziehung ohne gleitenden Beziehungsenden handelt. In diesem Fall kann die in der UML-Spezifikation mit dem Stereotype `RelationshipType` gekennzeichnete Assoziation einfach auf eine entsprechende Beziehung innerhalb des Datenbankschemas abgebildet werden. Sobald aber ein Beziehungsende mit Hilfe des UML-Profiles als gleitendes Beziehungsende markiert wurde, wird eine wesentlich komplexere Abbildung benötigt:

- Die Semantik eines gleitenden Beziehungsendes erfordert die Verwaltung von Kandidatenmengen. Daraus folgt, dass auch für ein einwertiges gleitendes Beziehungsende auf der Ebene der Datenbank mehrere Objektversionen gespeichert werden müssen. Die Kardinalität eines gleitenden Beziehungsendes ist also im Datenbankschema immer größer als 1.
- Für gleitende Beziehungsenden reicht es nicht aus, nur zu speichern, welche Objektversionen über die Beziehung verbunden sind. Zusätzlich muss auch die Information verwal-

tet werden, welche Objektversion aus der Kandidatenmenge aller verbundenen Versionen durch den Benutzer vorausgewählt (pinned) wurde. Dafür ist pro gleitendem Beziehungsende eine zusätzliche Beziehung zwischen den beteiligten Objekttabellen notwendig.

- Außerdem soll zu einer Kandidatenmenge auch die neuste Objektversion bzw. die neusten Objektversionen aufgezeichnet werden, um die regelbasierte Auswahl aus der Kandidatenmenge in der Sitzungsschicht zu beschleunigen. Zur Speicherung dieser Informationen ist für jedes gleitende Beziehungsende eine weitere Beziehung zwischen den beteiligten Objekttabellen notwendig.

Dies bedeutet, dass eine im Informationmodell definierte Beziehung zwischen versionierten Objekttypen, die zwei gleitende Beziehungsenden besitzt, auf insgesamt fünf Beziehungen im Datenbankschema abgebildet wird. Dies ist im laufenden CMS-Beispiel für die Beziehung `MenuContents` zwischen den Objekttypen `Menu` und `Teaser` der Fall. Im Beispiel 4.2 sind drei der dazugehörigen Zwischentabellen enthalten. Die Zwischentabelle `Menu_content` enthält alle in Beziehung stehenden Versionskombinationen, also die Kandidatenmenge der Beziehung. Die Zwischentabellen `Menu_contentPinned` und `Menu_contentNewest` enthalten aus Sicht des Objekttyps `Menu` alle vorausgewählten, bzw. neusten Objektversionen des Typs `Teaser`. Nicht sichtbar sind entsprechenden Tabellen für die Gegenrichtung, nämlich `Teaser_categoryPinned` und `Teaser_categoryNewest`.

Eine generische Lösung kommt mit einem wesentlich einfacheren Datenbankschema aus, da hier alle Beziehungen in gemeinsam genutzten Tabellen gespeichert werden können. Diese enthalten neben den beiden Fremdschlüsseln der beteiligten Objektversionen drei Felder zur Kennzeichnung des Beziehungstyps und der jeweiligen Beziehungsrollen der beteiligten Objektversionen. Auf diese Weise brauchen nur drei Tabellen angelegt werden. Eine für die Speicherung der eigentlichen Beziehungen, und die beiden anderen zur Verwaltung der Informationen über die neusten bzw. die vorausgewählten Objektversionen. Nachteil dieser Lösung ist, neben dem großen Umfang dieser zentralen Tabellen, der erhöhte Verarbeitungsaufwand durch die dynamische Typfestlegung.

4.2.2 Sitzungsschicht

Die Sitzungsschicht besteht aus je einer Softwarekomponente für jeden Objekttyp des Versionierungssystems. Diese Komponenten beinhalten jeweils die gesamte Anwendungslogik des Versionierungssystems, soweit sie sich auf den zugeordneten Objekttyp bezieht. Diese Anwendungslogik umfasst insbesondere die Lebenszyklusverwaltung von Objekten, die Versionsverwaltung im Fall von versionierten Objekten und die Verwaltung der Beziehungen für alle Beziehungstypen die mit dem zugeordneten Objekttyp verbunden sind.

Die Instanzen dieser Softwarekomponenten sind nicht an bestimmte Datenobjekte gebunden, sondern erlauben den Zugriff auf den gesamten Objektbestand. Sie besitzen aber trotzdem einen internen Zustand, denn sie verwalten den Zustand der Sitzung eines Systembenutzers und werden deshalb *Sitzungskomponenten* (session components) genannt. Eine Instanz einer Sitzungskomponente wird beim Beginn der Interaktion des Benutzers mit dem System erzeugt und bleibt für die Dauer der Sitzung diesem Besitzer fest zugeordnet. Nach dem Ende der Benutzersitzung wird die Instanz zerstört bzw. in einen neutralen Zustand zurückversetzt. Alle Informationen über den Zustand der Sitzung, die durch eine Sitzungskomponenten verwalteten werden, haben also nur für die Dauer der Sitzung Gültigkeit und werden

anschließend aufgegeben. Die Verwaltung der Komponenteninstanzen erfolgt dabei für den Systembenutzer transparent, da die beiden webbasierten Systemschnittstellen die Instanzen selbstständig anlegen, mit geeigneten technischen Maßnahmen dem Benutzer zuordnen und nach Beendigung der Interaktion selbstständig freigeben. Der Benutzer sieht dabei nur eine statische Schnittstelle und erhält keine direkte Kontrolle über die Instanz der Sitzungskomponente.

Implementiert wird die Sitzungsschicht im Rahmen des J2EE-Frameworks durch Session-EJBs. Da die Sitzungskomponenten den Zustand der Anwendersitzung verwalten müssen, werden sie im Deployment-Deskriptor der Anwendung als Stateful Session Beans deklariert. Diese Variante der Session-EJBs besitzt Attribute und damit einen Objektzustand. Der Lebenszyklus eines Session-EJBs entspricht dem eines Entity-EJBs. Der Anwendungsserver erzeugt beim Start des Systems einen Pool von EJB-Instanzen, die auf Anforderung mit einem Systembenutzer assoziiert werden. Sobald eine Instanz nicht mehr benötigt wird, versetzt der Server sie in einen neutralen Zustand zurück und fügt sie wieder in den Pool ein.

Im vorliegenden Entwicklungsstand der Produktlinie für Versionierungssysteme verwalten die Sitzungkomponenten zwei Informationen über den Zustand der Benutzersitzung, und zwar den aktuellen Arbeitskontext und den aktuellen Transaktionskontext.

Der Benutzer kann für jeden Arbeitskontext-Typ des Informationsmodells einen aktuellen Arbeitskontext definieren. Dazu wählt er aus der Menge der vorhandenen Arbeitskontext-Objekte ein passendes Objekt aus und übergibt es an die Sitzungskomponente. Dieser Schritt hat vielfältige Auswirkungen auf das Verhalten der Sitzungskomponente:

- Die Sessionkomponente fügt Objekte die sie erzeugt automatisch in den aktuellen Arbeitskontext ein. Dieses Verhalten betrifft die Operationen `createObject` und `copyObject`.
- Alle Operationen der Sessionkomponente, die zur Auffindung von Objekten dienen, liefern nur Objektversionen zurück, die sich im aktuellen Arbeitskontext befinden. Auch bei der Navigation über Beziehungen werden nur diejenigen verknüpften Objektversionen beachtet, die sich im aktuellen Arbeitskontext befinden. Dieses Verhalten ermöglicht eine versionsfreie Sicht auf die im Versionierungssystem gespeicherten Daten, da von versionierten Objekten jeweils nur maximal eine Version sichtbar ist.
- Die Information über den aktuellen Arbeitskontext des Benutzers wird auch benötigt, um zu überprüfen, ob der Benutzer eine durch die `checkout`-Operation gesperrte Objektversion verändern darf. Dies ist nur dann der Fall, wenn der Benutzer sich im dem Arbeitskontext befindet, in dem die Objektversion enthalten ist.

Die Verwaltung des Transaktionskontexts dient zur Implementierung einer erweiterten Transaktionssemantik. Standardmäßig veranlasst jeder atomare Aufruf einer Sitzungskomponente den Start einer neuen Transaktion. Diese bleibt während der Durchführung der aufgerufenen Operation geöffnet. Bei einem erfolgreichen Abschluss der Operation wird ein Commit der Transaktion ausgelöst und die vorgenommen Datenänderungen damit endgültig durchgeführt und für andere Systembenutzer sichtbar gemacht. Falls hingegen ein Fehler während der Abarbeitung der Operation auftritt, wird ein Rollback ausgeführt und die bereits getätigten Änderungen damit rückgängig gemacht.

Diese einfache Semantik ist nicht ausreichend für die Implementierung komplexer Anwendungslogik in einem Client des Versionierungssystems. Dafür ist es erforderlich, dass der Client Einfluss auf den Ablauf der Transaktionen nehmen kann. Einerseits sollte er die Set-

zung der Transaktionsgrenzen bestimmen können, um mehrere logisch zusammengehörende Aufrufe der Sitzungskomponente in einer Transaktion zusammenfassen zu können. Anderseits sollte die Entscheidung über den Erfolg einer Transaktion nicht allein durch Sitzungskomponente sondern mit Beteiligung des Client getroffen werden. Zu diesem Zweck bieten die Sitzungskomponenten die Operationen `beginTx`, `abortTx` und `commitTx` an. Die Operation `beginTx` veranlasst den Start einer neuen Transaktion und assoziiert sie mit der betreffenden Sitzungskomponente und damit auch mit der Sitzung des Benutzers. Alle folgenden Aufrufe der Sitzungskomponente werden im Kontext dieser Transaktion ausgeführt. Nach der Durchführung der notwendigen Änderungen kann der Client die Transaktion mit `abortTx` bzw. `commitTx` beenden, vorrausgesetzt die Transaktion wurde nicht aufgrund eines Fehlers bereits von der Sitzungskomponente beendet.

Beispiel 4.4 Die Schnittstelle einer Sitzungskomponente

```
public interface MenuAccess {

   // object creation, deletion and finders ========================
   public int createObject();
   public MenuValue createObject_Value();
   public int copyObject(int globalId);
   public MenuValue copyObject_Value(int globalId);
   public void deleteObject(int globalId);
   public int[] findAll();
   public MenuValue[] findAll_Value();
   public int[] findByObjectId(int objectId);
   public MenuValue[] findByObjectId_Value(int objectId);

   // versioning operations ========================================
   public int createSuccessor(int globalId);
   public MenuValue createSuccessor_Value(int globalId);
   public void deleteVersion(int globalId);
   public void merge(int globalId, int ancestorGlobalId);
   public int getRoot(int globalId);
   public MenuValue getRoot_Value(int globalId);
   public int[] getAncestors(int globalId);
   public MenuValue[] getAncestors_Value(int globalId);
   public int[] getSuccessors(int globalId);
   public MenuValue[] getSuccessors_Value(int globalId);
   public int[] getAlternatives(int globalId);
   public MenuValue[] getAlternatives_Value(int globalId);

   // attribute value access =======================================
   public MenuValue getValueObject(int globalId);
   public void setValueObject(int globalId, MenuValue valueObject);

   public java.lang.String getDescription(int globalId);
   public void setDescription(int globalId, java.lang.String newDescription);

   [...]
}
```

In der Einleitung dieses Abschnitts wurde bereits erwähnt, dass die Sitzungsschicht die objektorientierten Sicht auf die Objektdaten, die von der Persistenzschicht angeboten wird, auf eine prozedurale Schnittstelle abbildet. Diese Abbildung ist notwendig, da die Webservice-Schnittstelle aufgrund technologischer Einschränkungen eine prozedurale Form haben muss. Da auch die zweite Systemschnittstelle in der Form der dynamischen Webseiten mit einer prozeduralen Sitzungsschicht einfacher zu implementieren war, wurde entschieden, diese Abbildung in der Sitzungsschicht vorzunehmen.

Das Ergebnis der Abbildung ist in Beispiel 4.4 zu sehen. Das Beispiel enthält einen Ausschnitt aus der Schnittstelle der Sitzungskomponente für den Objekttyp `Menu`. Dieser Objekttyp ist in der aus Abbildung 3.15 bekannten Spezifikation eines Content-Management-Systems enthalten. Die Methoden der Schnittstelle sind größtenteils bereits aus dem Merkmalmodell für Versionierungssysteme bekannt. Entfernt wurden in diesem Beispiel ein Teil der Zugriffsmethoden für die Objektattribute, sowie alle Methoden die im Zusammenhang mit den Beziehungen zu anderen Objekttypen stehen.

Bei einer prozeduralen Schnittstelle kann eine Methode, wie beispielsweise `getDescription` aus dem obigen Beispiel, nicht aus dem Kontext eines Programmiersprachenobjekts erschließen, auf welches Datenobjekt sie sich beziehen soll. Deshalb wurde in die Parameterliste aller Methoden, die sich auf ein konkretes Datenobjekt beziehen, der zusätzliche Parameter `globalId` eingefügt. Dieser enthält den Primärschlüsselwert des Objekts, mit dem in der Persistenzschicht der zum Objekt gehörende Tabelleneintrag lokalisiert werden kann. Der Client kann die Primärschlüsselwerte der für ihn relevanten Objekte über verschiedene Methoden der Sitzungskomponente in Erfahrung bringen:

- Die Methoden zur Auffindung von Objekten wie etwa `findAll` und `findByObjectId` liefern den Primärschlüssel bzw. die Menge der gefunden Primärschlüssel zurück.
- Die Methoden zur Navigation von Beziehungen bzw. des Versionsgraphen geben ebenfalls die Primärschlüssel der verknüpften Objekte zurück. Im Beispiel 4.4 ist u.A. die Methode `getRoot` ein Vertreter dieser Navigationsmethoden.
- Auch bei der Erzeugung neuer Objekte bzw. Objektversionen mit Hilfe der Methoden `createObject`, `copyObject` und `createSuccessor` erhält der Client die Primärschlüssel der erzeugten Objektversionen.

Ein Client, dem der passende Primärschlüsselwert bekannt ist, kann die Attributwerte eines Objekts auslesen und verändern, bzw. andere Methoden auf das Objekt anwenden, die Informationen über das Objekt liefern oder den Zustand des Objekts verändern. Da im Falle eines entfernten Web-Clients jeder Methodenaufruf einen großen Kommunikationsaufwand mit sich bringt, wäre es allerdings sehr ineffizient, jeden einzelnen Attributwert eines Objekts in einem separaten Methodenaufruf zu übermitteln. Umgekehrt wäre auch eine Aktualisierung der Attributwerte sehr aufwendig, wenn jedes Attribut in einem separaten Methodenaufruf angesprochen würde. Deshalb wurde die Möglichkeit geschaffen, alle Daten eines Objekts gesammelt in einer Datenstruktur, die *Value Object* genannt wird, in einem einzigen Methodenaufruf vom Versionierungssystem zum Client und umgekehrt zu übermitteln. Dazu dienen die im Beispiel 4.4 sichtbaren Methoden `getValueObject` bzw. `setValueObject`. Um noch weitere Methodenaufrufe einzusparen, wurde für jede Methode, die einen Primärschlüsselwert als Rückgabewert liefert eine äquivalente Methode in die Schnittstelle eingefügt, die direkt die Objektdaten in der Form von Value-Objekten zurückliefert. Diese Methoden sind an der Namensendung „`_Value`“zu erkennen.

Bei der Ausführung der Operationen zur Manipulation der Objektdaten garantiert die Sitzungsschicht die Einhaltung einer Reihe von Konsistenzbedingungen, die im Merkmalmodell definiert worden sind. Dazu zählt unter anderem, dass keine Änderungen an Objektversionen vorgenommen werden dürfen, die für einen Schreibzugriff gesperrt worden sind. Eine Objektversion ist dann gesperrt, wenn entweder das `frozen`-Attribut gesetzt wurde, oder aber der Benutzer sich in einem fremden Arbeitskontext befindet, nachdem die Version über die `checkout`-Operation an ihren Arbeitskontext gebunden wurde. Je nach

Spezifikation muss die Sitzungskomponente nach Änderungen an einer Objektversion auch sicherstellen, dass das Datum der letzten Änderung dieser Version aktualisiert wird.

Ein weiterer wichtiger Aspekt der Anwendungslogik innerhalb der Sitzungskomponenten ist die Verwaltung von Beziehungen. In der Beschreibung der Persistenzschicht wurde bereits dargestellt, dass ein Beziehungstyp des Informationsmodells auf bis zu fünf Beziehungstabellen im Datenbankschema des Versionierungssystems abgebildet wird. Die Sitzungsschicht muss für die Implementierung der Navigationsmethoden diese Beziehungstabellen auswerten und bei Änderungen ihre Konsistenz erhalten. Beispielhaft sollen die Umsetzung von drei Aufgaben der Beziehungsverwaltung dargestellt werden:

- *Einfügen einer neuen Beziehung* - 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. Beispielsweise muss bei einem einwertigen, gleitenden Beziehungsende darauf geachtet werden, dass nur Objektversionen eines einzigen Objekts verknüpft werden. Falls die Kardinalitätsbedingungen inklusive der neuen Beziehung verletzt wären, werden diejenigen bestehenden Beziehungen entfernt, die eine Erfüllung der Bedingungen verhindern. Nun kann die neue Beziehung eingefügt werden. Dazu wird zunächst die reguläre Beziehung zwischen den Objekttypen angelegt, die die Kandidatenmenge definiert. Anschließend müssen die bestehenden Beziehungen zur Kennzeichnung der neusten Objektversionen in der Kandidatenmenge überprüft und gegebenfalls geändert werden.
- *Auffindung der relevanten Objektversionen im Rahmen der Navigation* - Die Methoden zur Navigation entlang einer Beziehung geben aus der zur Verfügung stehenden Kandidatenmenge pro Objekt nur eine einzige Objektversion zurück. Diese Auswahl wird durch die Anwendung der folgenden Regeln bewerkstelligt. Zunächst wird überprüft, ob im Sitzungszustand ein aktueller Arbeitskontext definiert 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. Falls zur Zeit kein aktueller Arbeitskontext existiert, werden die beiden zusätzlichen Beziehungen ausgewertet, die festlegen, welche die neusten Objektversionen der Kandidatenmenge sind und welche Objektversionen durch den Benutzer vorausgewählt (pinned) wurden. Dabei wird, falls vorhanden, die vorausgewählte Objektversion zurückgeliefert, ansonsten die neuste Objektversion. Beispiel 4.5 zeigt einen Ausschnitt aus der Vorlage zur Generierung der Sitzungkomponenten, in dem die Navigationslogik generiert wird.
- *Propagierung einer Operation zu verknüpften Objekten* - Im Rahmen der Spezifikation eines Versionierungssystems kann für jedes Beziehungsende bestimmt werden, welche Operationen propagiert werden sollen. Operationsprogagierung bedeutet, dass beim Aufruf der entsprechenden Methode der Sitzungskomponente diese Operation nicht nur auf die spezifizierte Objektversion angewendet wird, sondern auch auf alle anderen Objektversionen, die über dieses Beziehungsende mit der spezifizierten Version verbunden sind. Die Propagierung wird rekursiv durchgeführt. Zunächst wird die betreffende Operation auf dem lokalen Objekt durchgeführt. Dann werden mit Hilfe den oben beschriebenen Navigationsmethoden die für die Propagierung relevanten verbunden Objektversionen bestimmt. Für diese Objektversionen wird dann auf der zum Typ passenden Sitzungkomponente die Operation rekursiv aufgerufen. Diese Sitzungskomponente kann dann möglicherweise die Operation ebenfalls weiter propagieren. Um zu garantieren, dass die

Beispiel 4.5 Navigation entlang einer Beziehung

```
private ${associationEnd.oppositeEnd.participant.name}Local
        get${associationEnd.oppositeEnd.nameUpperCase}_Local(${class.name}Local object)
        throws FinderException
{
#if ( ($associationEnd.oppositeEnd.getTaggedValue("floating", "false") == "true") )
   Collection linkedObjects = object.get${associationEnd.oppositeEnd.nameUpperCase}();
   if (linkedObjects.size() == 0)
     return null;
#else
   ${associationEnd.oppositeEnd.participant.name}Local linkedObject =
                         object.get${associationEnd.oppositeEnd.nameUpperCase}();
   if (linkedObject == null)
     return null;
#end


#if ( !$class.hasStereotype("WorkspaceType") )
   #foreach ( $attAssociationEnd in $class.associationEnds )
     #if ( $attAssociationEnd.association.hasStereotype("AttachmentType") )
       #set( $relHasAttachment = false )
       #foreach ( $attOfRelAssociationEnd in
                  $associationEnd.oppositeEnd.participant.associationEnds )
         #if ( $attOfRelAssociationEnd.association.hasStereotype("AttachmentType") )
           #if ( $attOfRelAssociationEnd.oppositeEnd.name ==
                 $attAssociationEnd.oppositeEnd.name)
             #set( $relHasAttachment = true )
           #end
         #end
       #end
       #if ( $relHasAttachment == true )
         #if ( $associationEnd.oppositeEnd.getTaggedValue("floating", "false") == "true")
   if (mCurrent${attAssociationEnd.oppositeEnd.nameUpperCase} != null) {
     for (Iterator it = linkedObjects.iterator(); it.hasNext(); ) {
       ${associationEnd.oppositeEnd.participant.name}Local linkedObject =
                    (${associationEnd.oppositeEnd.participant.name}Local)it.next();
           #if ( $attAssociationEnd.oppositeEnd.isMultiValued() )
       if (linkedObject.get${attAssociationEnd.oppositeEnd.nameUpperCase}()
           .contains(mCurrent${attAssociationEnd.oppositeEnd.nameUpperCase}))
         return linkedObject;
           #else
     ${attAssociationEnd.oppositeEnd.participant.name}Local ws =
                        linkedObject.get${attAssociationEnd.oppositeEnd.nameUpperCase}();
       if ((ws != null) &&
           (ws.getGlobalId().equals(mCurrent${attAssociationEnd.oppositeEnd.nameUpperCase}
            .getGlobalId())))
         return linkedObject;
           #end
       }
       return null;
     }
           #else
     if (mCurrent${attAssociationEnd.oppositeEnd.nameUpperCase} != null) {
           #if ( $attAssociationEnd.oppositeEnd.isMultiValued() )
       if (linkedObject.get${attAssociationEnd.oppositeEnd.nameUpperCase}()
            .contains(mCurrent${attAssociationEnd.oppositeEnd.nameUpperCase}))
         return linkedObject;
           #else
       ${attAssociationEnd.oppositeEnd.participant.name}Local ws =
                        linkedObject.get${attAssociationEnd.oppositeEnd.nameUpperCase}();
       if ((ws != null) && (ws.getGlobalId()
           .equals(mCurrent${attAssociationEnd.oppositeEnd.nameUpperCase}.getGlobalId())))
         return linkedObject;
           #end
       else
         return null;
     }
           #end
         #end
       #end
     #end
   #end
#end


#if ( $associationEnd.oppositeEnd.getTaggedValue("floating", "false") == "true")
   ${associationEnd.oppositeEnd.participant.name}Local pinnedObject =
                        object.get${associationEnd.oppositeEnd.nameUpperCase}Pinned();
   if (pinnedObject != null)
     return pinnedObject;
   return object.get${associationEnd.oppositeEnd.nameUpperCase}Newest();
#else
   return linkedObject;
#end
}
```

Propagierung in jedem Fall terminiert, auch falls Zyklen in den Beziehungen vorhanden sind, wird beim Abstieg in die Rekursion eine Datenstruktur mitgeführt, die alle bereits besuchten Objektversionen festhält. Trifft eine Sitzungskomponente auf eine bereits besuchte Objektversion, wird die Rekursion abgebrochen.

Die Gestalt der Implementierung der Sitzungsschicht ist, wie im Fall der Persistenzschicht, stark davon abhängig, ob ein generischer oder generativer Ansatz gewählt wird. Im Fall des generischen Ansatz wird, etwa durch den Einsatz von Vererbungstechniken, der Anteil an dupliziertem Quellcode in den einzelnen Sitzungskomponenten möglichst gering gehalten, denn duplizierte Quellcodemodule behindern die manuelle Wartung des Versionierungssystems. Die wiederverwendeten Implementierungsanteile würden generisch ausgelegt, sodass sie an alle möglichen Spezifikationen durch einen geeigneten Konfigurationsmechanismus angepasst werden können. Im Fall des von uns verfolgten generativen Ansatzes hingegen kann eine Duplizierung von großen Teilen der Implementierung in Kauf genommen werden, da Softwarewartung über eine Weiterentwicklung der Generierungsvorlagen vorgenommen wird. Dies hat zur Folge, dass alle verwendeten Algorithmen schon während der Generierung auf die konkrete Spezifikation des betreffenden Versionierungssystems hin optimiert werden können.

4.2.3 Systemschnittstellen

Web-Schnittstelle

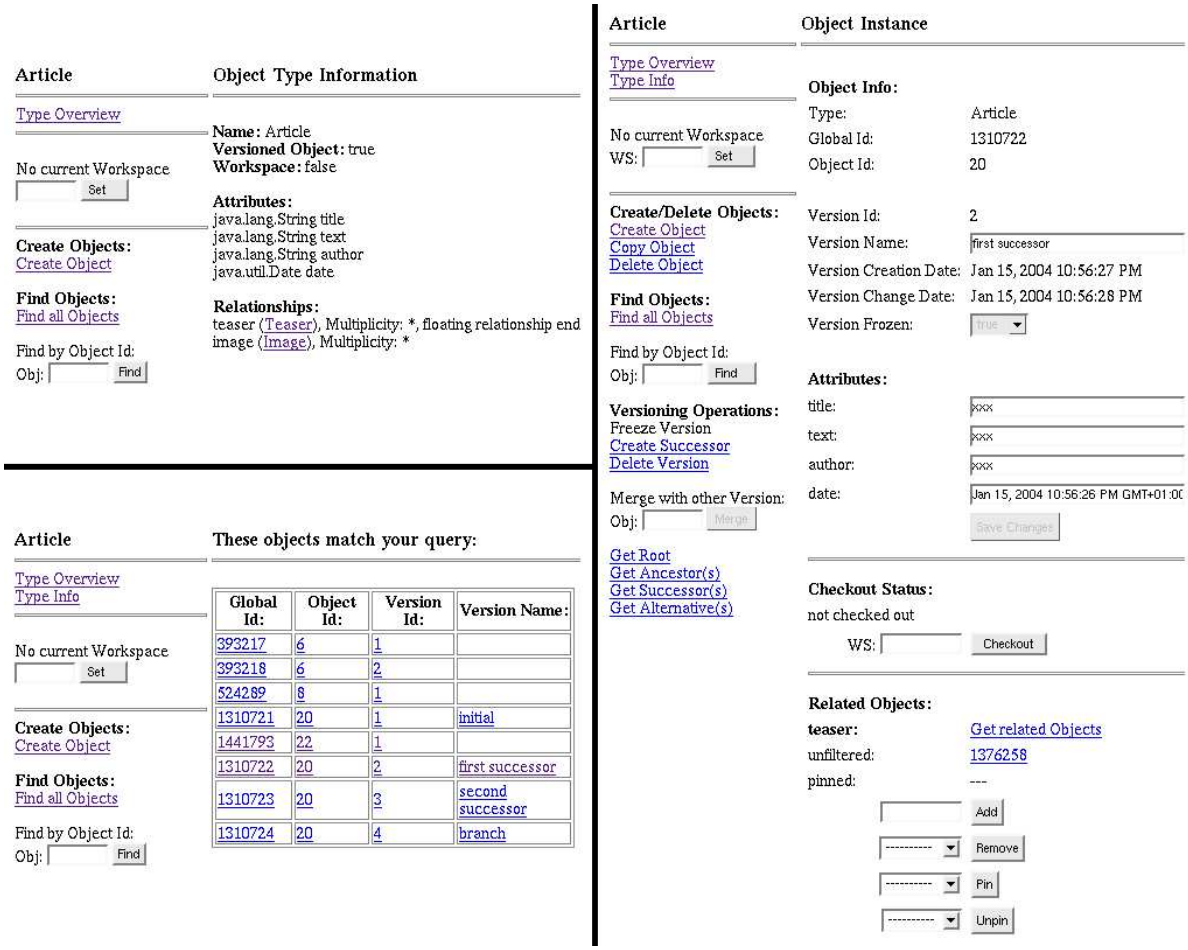
Die auf dynamischen Webseiten basierende Web-Schnittstelle der Versionierungssysteme soll es einem menschlichen Benutzer ermöglichen, über einen Webbrowsers mit dem Versionierungssystem zu interagieren. Dabei kann er sich sowohl über das zugrundeliegende Informationsmodell informieren, als auch Inhalte einsehen und editieren.

Ausgangspunkt der Benutzerinteraktion ist dabei eine Seite, die die im Informationsmodell enthaltenen Objekt- und Beziehungstypen auflistet. Von dieser Seite gelangt der Benutzer zu Informationsseiten über die einzelnen Objekttypen. Eine solche Typ-Informationsseite ist links-oben in der Abbildung 4.8 zu sehen. Alle diese Informationsseiten sind nicht dynamisch, sondern bestehen aus statischen HTML-Dokumenten, die durch den Generator erzeugt werden.

Ausgehend von einer Typ-Informationsseite kann der Benutzer auf bestehende Objekte dieses Typs zugreifen bzw. neue Objekte anlegen. Dazu wählt er die passende Operation aus dem linksseitigen Menü aus. Dieses Menü listet stets die in der jeweiligen Situation anwendbaren Operationen der zuständigen Sitzungskomponente auf. Das in der Abbildung 4.8 links-unten platzierte Bildschirmfoto zeigt die Ausgabe der `findAll`-Operation. Aus dieser Liste von Objektversionen kann der Benutzer eine Version auswählen und gelangt damit zu der Ansicht, die im rechten Bildschirmfoto dargestellt wird. In dieser Ansicht einer einzelnen Objektversion können auf der rechten Seite des Bildschirms die Attributwerte und Beziehungen der Objektversion eingesehen und verändert werden. Im Auswahlmenü auf der linken Bildschirmseite stehen nun zusätzlich die Versionierungsoperationen und die Operationen zum Löschen und Kopieren der Objekts zur Auswahl.

In der derzeitigen Implementierung bieten die Webseiten den Zugriff auf die vollständige Funktionalität der Sitzungskomponenten, und eignen sich damit zum Test der generierten Versionierungssysteme. Die Aufbereitung der Daten unterstützt die Funktion als

Abbildung 4.8 Bildschirmfotos der Webseiten

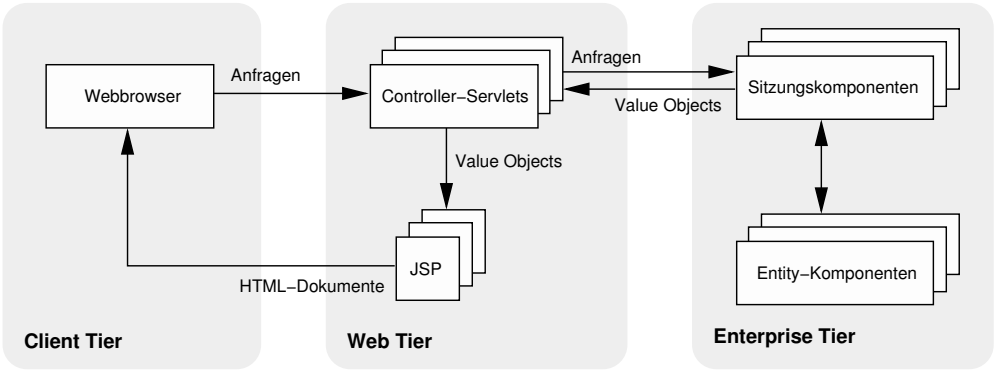


Testwerkzeug. Falls die Webschnittstelle hingegen durch Endanwender verwendet werden sollte, müsste die Ergonomie der Oberfläche durch eine Anpassung an die Bedürfnisse der verschiedenen Benutzergruppen verbessert werden. Im Bezug auf das Beispiel des Content-Management-Systems wären beispielsweise spezialisierte Oberflächen für Redakteure, Webdesigner und Systemadministratoren denkbar.

Die Abbildung 4.9 zeigt die technische Realisierung der Web-Schnittstelle auf der Basis von Servlets und JSP. Das Design dieser Lösung wurde gemäß des MVC-Musters [KP88] entwickelt. Dieses Muster sieht vor, dass die Steuerung der Anfrageverarbeitung getrennt von der eigentlichen Bearbeitung der Anfrage und der Darstellung der Anfrageergebnisse implementiert wird.

Die Webschnittstelle bearbeitet Anfragen, die vom Benutzer des Systems über seinen Browser an das System gestellt werden. Diese HTTP-Anfragen gelangen zunächst, abhängig vom Objekttyp, auf den sich die Anfrage bezieht, beim zuständigen Controller-Servlet. Jeder Objekttyp besitzt ein eigenes Controller-Servlet. Die Zuordnung der Anfragen zu den Servlets wird über den Deployment-Deskriptor des Servlet-Containers realisiert. Das Controller-Servlet überprüft zunächst, ob für diesen Benutzer bereits eine Instanz der zuständigen Sitzungskomponente erzeugt wurde. Falls dies nicht der Fall ist, wird eine neue Instanz erzeugt

Abbildung 4.9 Design der Webschnittstelle



und mit der HTTP-Sitzung assoziiert. Dann entscheidet das Servlet, welche Operationen durchgeführt werden müssen und ruft die entsprechenden Methoden der Sitzungskomponente auf. Als Rückgabewert erhält es ein Value-Object bzw. eine ganze Menge dieser Objekte. Diese Daten leitet es an die passende JSP-Vorlage weiter, die daraus ein HTML-Dokument erstellt, das dem Webbrowser als Antwort auf seine HTTP-Anfrage zurückgeliefert wird.

Webservice-Schnittstelle

Als *Webservice* wird ein im Inter- bzw. Intranet verfügbarer Dienst bezeichnet, mit dem potentielle Clients über ein XML-basiertes Nachrichtenprotokoll kommunizieren können. Als Nachrichtenprotokoll wird typischerweise SOAP [W3C03] verwendet, eine durch das World Wide Web Consortium entwickelte Spezifikation, die sowohl ein Protokoll für synchrone Kommunikation in der Form von Methodenaufrufen, als auch ein Protokoll für asynchrone nachrichtenbasierte Kommunikation spezifiziert.

Ein Methodenaufruf wird durch den Versand zweier XML-Dokumente mittels HTTP oder eines anderen Basisprotokolls realisiert. Das erste XML-Dokument mit der Anfrage des Clients enthält die Information, welche Methode aufgerufen werden soll und die durch den Client übergebenen Eingabeparameter. Als Parameter können dabei auch komplexe Datenstrukturen übermittelt werden, da die SOAP-Spezifikation geeignete Verfahren zur Abbildung dieser Datenstrukturen auf XML-Elemente definiert. Nach der Interpretation der Anfragenachricht durch den Server führt dieser die gewünschte Methode aus und schickt den Rückgabewert der Methode in einer weiteren Nachricht an den Client zurück.

Die Information darüber, welche Arten von Anfragen der Server verarbeiten kann, also die Schnittstellendefinition, wird mit Hilfe der Web Services Description Language (WSDL) [W3C01] dokumentiert. Diese Spezifikation definiert das Format für ein XML-Dokument, dass die angebotenen Methoden und ihre Ein- und Ausgabeparameter beschreibt.

Die Vorteile des Webservices-Ansatzes liegen darin, dass er die Plattformunabhängigkeit komplexerer Middlewarestandards wie beispielsweise CORBA [OMG02a] mit der einfachen Implementierung properitärer Ansätze wie beispielweise RMI [Sun03] verbindet. Die Plattformunabhängigkeit der Webservices begründet sich darin, dass jede Plattform, die die Analyse und Erzeugung von XML-Dokumenten unterstützt, mit einem Webservice kommunizieren kann bzw. selbst einen solchen Dienst anbieten kann. Die einfache Implemen-

tierung des Webservice-Ansatzes wird durch die mittlerweile gut ausgeprägte Werkzeugunterstützung garantiert. Für die clientseitige Nutzung eines Webservice stehen beispielsweise Generatoren zur Verfügung, die aus der WSDL-Beschreibung der angebotenen Dienste Stubs generieren. Diese Stubs kapseln die gesamte Abwicklung der Kommunikation mit dem Webservice und stellen eine native Programmiersprachenschnittstelle für die durch den Webservice angebotenen Methoden zur Verfügung.

Der Hauptnachteil XML-basierter Middleware liegt in den hohen Kommunikationskosten, die durch die Erstellung und Interpretation der verschickten XML-Nachrichten entstehen. Die im nativen Binärformat vorliegenden Daten müssen aufwendig in die textuelle XML-Darstellung konvertiert werden und beim Empfang wieder dekodiert werden. Außerdem sind die resultierenden XML-Dokumente größer als die Nachrichtenpakete eines äquivalenten binären Nachrichtenprotokolls, weshalb auch die Übertragung der XML-Nachrichten mehr Zeit in Anspruch nimmt als die Übertragung binären Nachrichten.

Zur Realisierung der Webservice-Schnittstelle wird das durch die Apache Software Foundation entwickelte Open-Source-Produkt Axis [Apa03b] verwendet. Die serverseitige Implementierung von Axis besteht im Wesentlichen aus einem Servlet, das die eingehenden HTTP-Anfragen entgegen nimmt, und einer Bibliothek, die die Anfrage analysiert und an die eigentliche Implementierung des Webservices weiterleitet. Axis integriert sich also in den Web-Tier der J2EE-Anwendung.

Es existieren verschiedene Varianten zur Realisierung eines Webservices mit Axis. In der von uns gewählten Variante wird der Webservice durch eine normale Java-Klasse definiert. In einem speziellen Deployment-Deskriptor wird Axis mitgeteilt, unter welchem Namen welche Java-Klasse verfügbar gemacht werden soll. Bei der Installation des Deployment-Deskriptors analysiert Axis die angegebene Java-Klasse und generiert eine WSDL-Beschreibung des neuen Webservice, in die alle öffentlichen Methoden der Java-Klasse aufgenommen werden. Nun kann der neue Webservice bereits benutzt werden. Wenn eine Methode des Webservices aufgerufen wird, erzeugt Axis eine Instanz der Klasse und ruft die entsprechende Java-Methode der Instanz auf. Der Rückgabewert der Methode wird an den Aufrufer des Webservices zurückgeliefert, ohne das im Quellcode der Java-Klasse dafür irgendwelche Vorkehrungen getroffen werden müssten.

Für die Objekttypen der Versionierungssysteme erstellt der Generator jeweils eine Java-Klasse, die die gleiche Schnittstelle wie die zum Objekttyp gehörende Sitzungskomponente besitzt. Die Implementierung dieser Klasse bedient sich des durch Axis zur Verfügung gestellten Kontext, um eine Referenz auf eine Instanz dieser Sitzungskomponente zu speichern und damit an die Sitzung des Anwenders zu binden. Da SOAP ein zustandsloses Protokoll ist, realisiert Axis diesen Kontext über einen Schlüsselwert, der in den Header der SOAP-Nachrichten eingefügt wird. Dieses Verfahren setzt vorraus, dass der Client Schlüsselwerte, die er über die Antworten des Servers erhält, in nachfolgende Anfragen selbstständig einzufügt. Die von Axis zur Verfügung gestellte Client-Implementierung besitzt diese Eigenschaft.

Da beim Entwurf der Schnittstellen der Sitzungskomponenten wurde bereits auf die technischen Restriktionen der Webservice-Technologie Rücksicht genommen wurde, besteht die restliche Implementierung der Java-Klasse aus einfachen Weiterleitungen der Methodenaufrufe an die Sitzungskomponente. Alle Ein- und Ausgabeparameter der Sitzungskomponente sind entweder elementare Datentypen oder einfache Datenstrukturen, die durch Axis automatisch in XML-Elemente umgewandelt werden können.

Verwendet wird die Webservice-Schnittstelle momentan durch eine Test-Suite, die der Generator passend zum Informationsmodell des Versionierungssystems generiert. Die enthaltenen Tests überprüfen für jeden Objekttyp, ob die zugehörige Sitzungskomponente korrekt implementiert wurde. Getestet wird beispielsweise das Verhalten der Versions- und Beziehungsverwaltung mit ausgesuchten korrekten und inkorrekten Anfragen an das Versionierungssystem. Die Test-Suite überprüft selbstständig, ob alle Tests korrekt abgearbeitet wurden und eignet sich deshalb zur entwicklungsbegleitenden Verifikation während der Wartung und Weiterentwicklung der Generierungsvorlagen.

4.2.4 Übersicht der erstellten Vorlagen

Zum Abschluss dieses Kapitels soll ein Überblick über alle Vorlagen gegeben werden, die für die Generierung der Versionierungssysteme benötigt werden.

`application-xml, orion-application-xml`

Diese Vorlagen generieren die globalen Deployment-Deskriptoren die zur Installation der J2EE-Anwendung im Anwendungsserver notwendig sind.

`build-xml`

Diese Vorlage erzeugt ein Skript, das die Übersetzung und die Installation der J2EE-Anwendung automatisiert.

`ejb-jar-xml`

Durch diese Vorlage wird der Deployment-Deskriptor erstellt, der die Enterprise-Beans der Sitzungs- und Persistenzschicht beschreibt.

`repObjAccessBean, repObjAccessLocalHome, repObjAccessLocalInterface, repObjAccessRemoteHome, repObjAccessRemoteInterface`

Diese Vorlagen werden für jeden Objekttyp des Informationsmodells instantiiert und erzeugen die zugehörige Sitzungskomponente. Im einzelnen werden die Implementierung der Sitzungskomponente (`repObjAccessBean`), deren lokale und globale Schnittstellen, sowie die Home-Interfaces zur Erzeugung von EJB-Instanzen generiert.

`repObjBean, repObjLocalHome, repObjLocalInterface, repObjRemoteHome, repObjRemoteInterface, repObjValueBean`

Entsprechend wird für jeden Objekttyp ein Entity-Komponente generiert, die zusammen die Persistenzschicht des Versionierungssystems bilden. Auch hier muss die Implementierung des EJB (`repObjBean`), dessen lokale und globale Schnittstellen, sowie die zugehörigen Home-Interfaces generiert werden. Ausserdem wird für jeden Objekttyp ein normales JavaBean erstellt, das die Attributwerte dieses Objekttyps speichert (`repObjValueBean`).

`web-xml`

Für die Komponenten der Web-Schnittstelle und der Webservice-Schnittstelle muss ebenfalls ein Deployment-Deskriptor generiert werden, um diese in die J2EE-Anwendung zu integrieren.

`controllerServlet`, `showIndex`, `showInstance`, `showList`, `showTypeInfo`

Diese Vorlagen erzeugen die Servlets, JSP und HTML-Dokumente der Web-Schnittstelle.

`deploy-wsdd`, `undeploy-wsdd`, `repObjService`, `testClient`

Diese Vorlagen generieren die Webservice-Schnittstelle des Versionierungssystems. Die Vorlage `testClient` generiert auf der Basis der Webservice-Schnittstelle eine Testsuite für das Versionierungssystem.

Kapitel 5 Bewertung durch Softwaremetriken

Die entwickelte Implementierung der Produktlinie für Versionierungssysteme soll nun nach verschiedenen Kriterien beurteilt werden. In diesem Kapitel werden *Softwaremetriken* eingesetzt, um den Quellcode der Generatorvorlagen und der durch sie erstellten Softwaresysteme zu untersuchen und damit Aussagen über die Qualität der Generatorvorlagen und die Effektivität des generativen Ansatzes zu gewinnen. Im folgenden Kapitel werden dann die Leistungseigenschaften der Implementierung experimentell bewertet.

Als Softwaremetriken werden Verfahren bezeichnet, die aus dem Quellcode bzw. der Spezifikation eines Softwaresystems Messwerte ableiten. Dazu kann beispielsweise die Kontrollstruktur eines Programms untersucht werden und die Anzahl und Verschachtelungstiefe der Fallunterscheidungen ermittelt werden. Aus diesen Messwerten werden auf der Basis empirischer Erkenntnisse bzw. theoretischer Überlegungen weiterführende Aussagen über die untersuchten Systeme abgeleitet. Beispielsweise kann aus Messwerten, die den Umfang eines Systementwurfs charakterisieren, eine Prognose über den nötigen Aufwand zur Implementierung dieses Entwurfs erstellt werden.

Im Rahmen dieser Arbeit sollen mit Softwaremetriken Antworten auf die folgenden Fragestellungen gefunden werden:

- Wie ist die Qualität der entwickelten Generatorvorlagen im Bezug auf ihre Wartbarkeit zu beurteilen?
- Aus welchen Anteilen setzt sich die Implementierung des Beispielsystems zusammen und welche Wiederverwendungsquoten werden im Falle des Beispielsystems erreicht?
- Welcher Zusammenhang besteht zwischen dem Umfang der generierten Implementierung eines Versionierungssystems und dem Umfang seiner Spezifikation?

5.1 Verwendete Softwaremetriken

An dieser Stelle sollen nun zunächst die in diesem Kapitel verwendeten Metriken vorgestellt werden. Aus der Vielzahl der verfügbaren Metriken wurden drei ausgewählt, die zur Abschätzung der Komplexität eines Programms dienen können bzw. zur Abschätzung des bei der Erstellung des Programms angefallenen Entwicklungsaufwands.

5.1.1 Anzahl der Anweisungen

Eine der einfachsten Metriken ist die *Anzahl der Quellcodezeilen* (Lines of Code, LOC) eines Systems, wobei nur nicht-leere Zeilen gezählt werden, die keine Kommentare enthalten. Dieser Messwert ist einfach zu ermitteln, hat aber auch nur beschränkte Aussagekraft. Unter der Annahme, dass sich der durchschnittliche Aufwand für die Erstellung einer Quellcodezeile ermitteln lässt, kann aus der Anzahl der Quellcodezeilen auf den Entwicklungsaufwand des Systems geschlossen werden. Die Anzahl der Quellcodezeilen kann auch als Maß der Komplexität eines Systems angesehen werden, wenn voraussetzt wird, dass größere Softwaresysteme eine größere Komplexität besitzen. Alle diese Annahmen sind nur beim Vergleich sehr ähnlich strukturierter Systeme gerechtfertigt. Bei den hier vorgenommen Messungen wird eine leicht abgewandelte Form dieser Metrik verwendet, es werden nämlich durch eine Syntaxanalyse die im Quellcode enthalten *Anweisungen* gezählt, dardurch wird erreicht, dass der Messwert nicht von der Formatierung des Quellcodes abhängig ist.

5.1.2 Zyklomatische Komplexität

Das von McCabe [McC76] definierte Maß der *zyklomatischen Komplexität* bewertet die Gestalt des Kontrollflussgraphen eines Programms, indem sie die Anzahl der unabhängigen Wege durch den Graphen angibt. Sie ist definiert durch die Formel $MCC - V = |E| - |N| + 2$, wobei E die Menge der im Graphen enthaltenen Kanten (edges) bezeichnet und N die Knotenmenge (nodes). Je höher diese Zahl ist, desto komplexer ist die Kontrollstruktur des untersuchten Systems. Programme mit hoher zyklomatischer Komplexität lassen sich schlechter testen, da mehr Testfälle konstruiert werden müssen, um alle Kontrollpfade abzudecken. Komplexe Programme sind auch schwerer verständlich als einfach strukturierte Programme und erzeugen deshalb höhere Aufwendung in der Wartung und Entwicklung.

5.1.3 Halstead-Aufwand

Die Komplexität eines Programms liegt nicht nur in der Struktur des Kontrollgraphen, sondern auch in der Anzahl der zu verarbeitenden Eingabedaten und der Anzahl der dazu verwendeten Operationen. Diesen Aspekt untersuchen die durch Halstead [Hal77] definierten Metriken. Sie basieren auf den folgenden Eigenschaften des Programmcodes:

$n1$	Anzahl der unterscheidbaren Operatoren
$n2$	Anzahl der unterscheidbaren Operanden
$N1$	Anzahl der insgesamt enthaltenen Operatoren
$N2$	Anzahl der insgesamt enthaltenen Operanden

Daraus werden die folgenden Metriken abgeleitet:

Volume (V)	$V = (N1 + N2) * log2(n1 + n2)$
Difficulty (D)	$D = \frac{n1 * N2}{2 * n2}$
Effort (E)	$E = D * V$

In den folgenden Messungen verwenden wir den *Halstead-Aufwand* als einen weiteren Indikator für den Implementierungsaufwand der untersuchten Softwaresysteme.

5.1.4 Durchführung der Messungen

Für die Ermittlung der Messwerte wurde ein Werkzeug [Vir03] verwendet, das Java-Programme automatisch analysiert. Um auch die in den Generatorvorlagen enthaltenen Metaprogramme mit diesem Werkzeug analysieren zu können, musste der VTL-Code zunächst in äquivalenten Java-Code umgewandelt werden. Im Beispiel 5.1 ist ein Ausschnitt aus einer Vorlage zu sehen, der die Methode `createObject` generiert. Darunter steht der Java-Code, in den dieser Ausschnitt umgewandelt wurde. Die statischen Anteile der Vorlage werden bei der Abbildung durch die Ausgabe eines konstanten Platzhalters (`print("xxx")`) ersetzt, deshalb werden die in der Beispielvorlage enthalten Java-Anweisungen durch die Abbildung entfernt. Die Kontrollstrukturen und die dynamischen Ausgaben der Vorlagen werden in äquivalente Java-Konstrukte umgewandelt. Das resultierende Java-Programm kann zwar u.a. aufgrund fehlender Variablendefinitionen nicht übersetzt werden, hat aber im Bezug auf die verwendeten Metriken die gleichen Eigenschaften wie das Metaprogramm.

Beispiel 5.1 Abbildung von VTL-Code auf Java-Code

```
public ${class.name}Local createObject() throws Exception
{
  ${class.name}Local newObject = mHomeInterface.create();
#if ( !$class.hasStereotype("WorkspaceType") )
#foreach ( $attAssociationEnd in $class.associationEnds )
#if ( $attAssociationEnd.association.hasStereotype("AttachmentType") )
  if (mCurrent${attAssociationEnd.oppositeEnd.nameUpperCase} != null)
    attachTo${attAssociationEnd.oppositeEnd.nameUpperCase}_Local(newObject,
              mCurrent${attAssociationEnd.oppositeEnd.nameUpperCase},
              new HashMap());
#end
#end
#end
  return newObject;
}
```

resultierender Java-Code:

```
print("xxx");
print(class.name);
print("xxx");
print(class.name);
print("xxx");
if (!class.hasStereotype("WorkspaceType") ) {
  while (attAssociationEnd < class.associationEnds) {
    if (attAssociationEnd.association.hasStereotype("AttachmentType")) {
      print("xxx");
      print(attAssociationEnd.oppositeEnd.nameUpperCase);
      print("xxx");
      print(attAssociationEnd.oppositeEnd.nameUpperCase);
      print("xxx");
    }
  }
}
print("xxx");
```

5.2 Analyse der Generatorvorlagen

Im ersten Schritt werden nun die Generatorvorlagen analysiert, mit denen die Versionierungssysteme der Produktlinie erzeugt werden. Die Tabelle 5.1 enthält die Messergebnisse für jede der 25 Generatorvorlagen. Die ersten beiden Spalten dieser Tabelle enthalten die Dateigröße und die enthaltenen Textzeilen der Vorlagen.

Tabelle 5.1 Ergebnisse der Analyse der Generatorvorlagen

Vorlage	Dateigröße (in Bytes)	Textzeilen	Referenzen	Fallunterscheidungen	Schleifen	Anweisungen	Zyklomatische Komplexität	Halstead-Aufwand
RepObjAccessBean	100405	2064	1102	198	39	2388	237	22.220.480,88
TestClient	29481	569	562	49	11	1187	60	9.094.331,40
ControllerServlet	25728	595	232	55	14	523	69	1.531.176,87
ShowInstance	18647	379	146	33	8	331	41	644.664,46
Ejb-jar-xml	16762	436	124	27	11	294	38	416.433,32
RepObjService	15918	470	139	21	4	308	25	653.819,54
Build-xml	12685	312	14	1	1	31	2	9.874,24
RepObjBean	12291	312	111	35	6	260	41	392.915,51
RepObjAccessLocalInterface	11068	184	115	21	4	260	25	457.099,50
RepObjAccessRemoteInterface	10631	175	95	21	4	220	25	319.980,50
Web-xml	8504	220	103	14	4	224	18	421.654,41
RepObjLocalInterface	5739	123	72	15	3	161	18	174.160,17
ShowList	4294	103	39	9	2	88	11	71.251,00
ShowTypeInfo	4215	113	46	14	5	117	19	77.007,93
RepObjValueBean	2927	133	26	10	2	66	12	33.692,75
RepObjRemoteInterface	2342	56	17	5	1	42	6	13.537,27
RepObjRemoteHome	1429	32	13	3	1	30	4	8.755,36
ShowIndex	1350	46	18	5	4	45	9	13.602,35
RepObjLocalHome	1291	31	13	3	1	30	4	8.755,36
Deploy-wsdd	1151	19	10	2	2	26	4	3.853,01
Application-xml	604	22	7	1	0	13	1	1.608,44
RepObjAccessRemoteHome	327	11	3	0	0	8	0	303,40
RepObjAccessLocalHome	293	10	3	0	0	8	0	303,40
Undeploy-wsdd	245	5	2	0	1	8	1	498,29
Orion-application-xml	123	3	1	0	0	2	0	8,00
(25 Vorlagen insgesamt)	*288450*	*6423*	*3013*	*542*	*128*	*6670*	*670*	*36.569.767,36*

Die folgenden Spalten beziehen sich nur auf die in den Vorlagen enthaltenen Metaprogramme. Zunächst wird unter „Referenzen“ die Anzahl der Zugriffe auf Modelldaten über den Kontext aufgeführt. Dazu wird sowohl die Ausgabe eines Wertes aus dem Modell in die generierte Datei als auch die Auswertung von Modelldaten für eine Fallunterscheidung (#if-

Anweisung) gezählt. Dass insgesamt an mehr als 3000 Stellen Modelldaten in die generierten Dateien einfließen, zeigt wie stark der Inhalt der generierten Dateien von der Spezifikation des Versionierungssystems abhängen. In den Spalten „Fallunterscheidungen“ und „Schleifen“ wird die Anzahl dieser Anweisungen im Metaprogramm der Vorlagen aufgeführt.

Die letzten drei Spalten enthalten die Messergebnisse, die das Werkzeug für die im letzten Abschnitt vorgestellten Softwaremetriken ermittelt hat. Bei den Anweisungszahlen fällt auf, dass sie höher sind, als die Summe der aufgeführten Referenzen, Fallunterscheidungen und Schleifen. Dies liegt daran, dass bei der verwendeten Abbildung des VTL-Codes auf Java-Code die statischen Anteile der Vorlagen in Ausgabeanweisungen umgewandelt worden sind, die in die Gesamtanzahl der Anweisungen eingehen. Die Anzahl der Anweisungen setzt sich also aus Anweisungen für dynamische und statischen Ausgaben sowie den Anweisungen zur Steuerung des Kontrollflusses zusammen. Bei einigen Vorlagen ist die Anzahl der Anweisungen auch höher als die Anzahl der Textzeilen. In diesen Fällen beinhaltet die Vorlage viele Zeilen, in denen mehreren Modellreferenzen enthalten sind und somit auf mehrere Ausgabeanweisungen abgebildet werden.

Abbildung 5.1 Größenverhältnisse der Generatorvorlagen (Textzeilen)

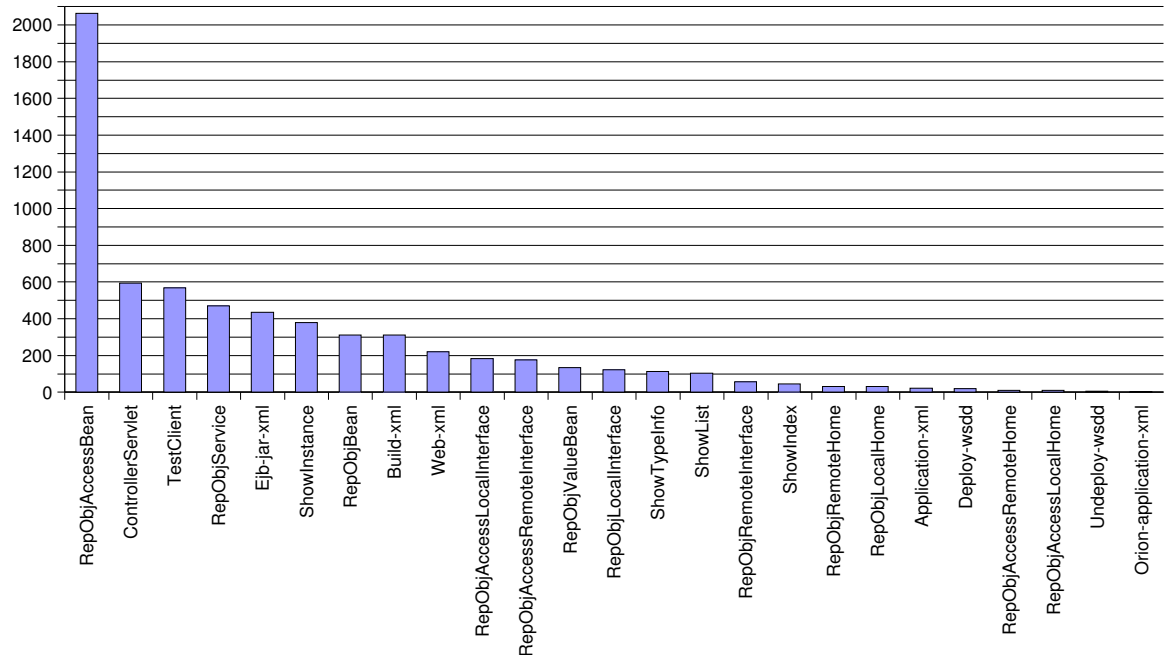
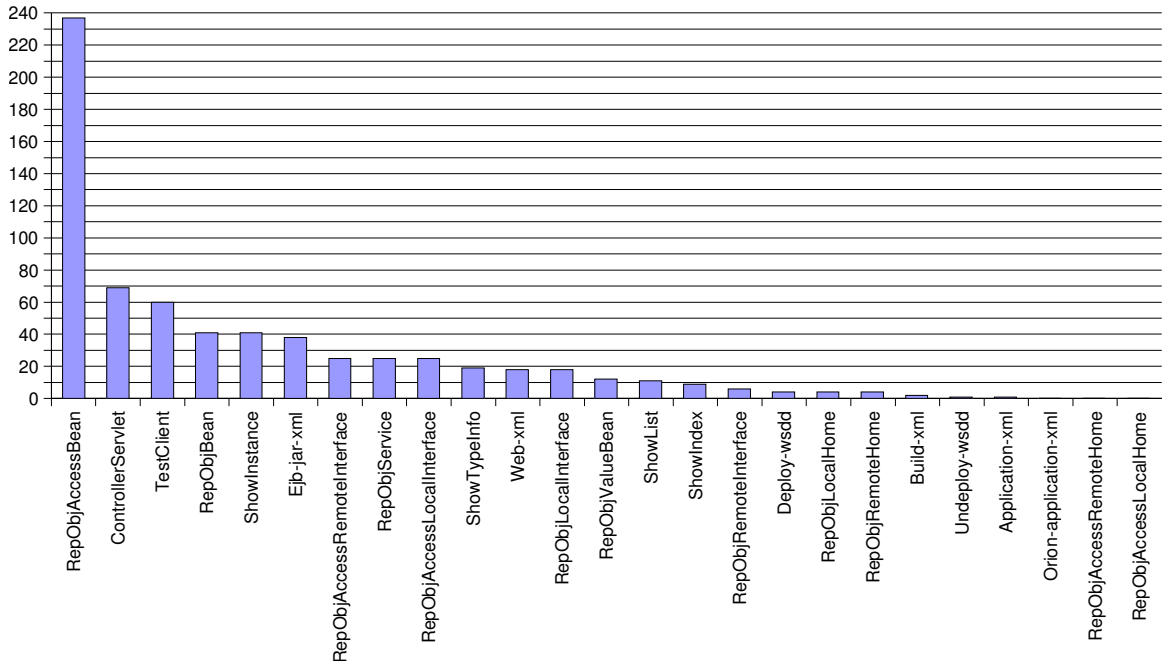


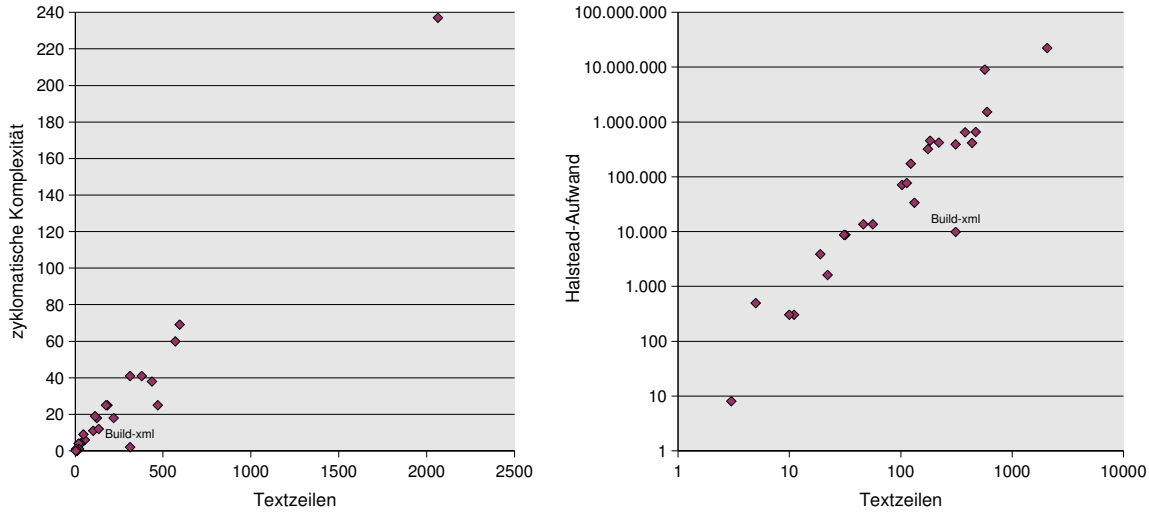
Abbildung 5.1 zeigt grafisch, dass die Größen der Vorlagen sehr stark differieren. Auf die Vorlage „RepObjAccessBean“, die die Implementierung der Sitzungskomponenten und damit ein Großteil der Anwendungslogik generiert, entfällt fast ein Drittel der gesamten Textzeilen. Aber auch der Umfang und die Komplexität der enthaltenen Metaprogramme differieren und steigen mit der Größe der Vorlagen an. Abbildung 5.2 zeigt die Verteilung der zyklomatischen Komplexität innerhalb der Vorlagen. Die Vorlagen mit den höchsten Komplexitätswerten belegten auch in Abbildung 5.1 die ersten Plätze. Im Anhang A.1 sind weitere Diagramme für die übrigen Metriken enthalten.

Abbildung 5.2 Zyklomatische Komplexität der Generatorvorlagen



Zwischen Umfang der Vorlage und der Komplexität des Metaprogramms besteht sogar ein annähernd lineare Abhängigkeit. Dieser Zusammenhang wird in Abbildung 5.3 im linken Diagramm visualisiert. In diesem Diagramm sind auf der X-Achse die Textzeilen der Vorlagen aufgetragen und auf der Y-Achse die ermittelte zyklomatische Komplexität. Jeder Diagrammeintrag entspricht einer Vorlage.

Abbildung 5.3 Korrelation zwischen Umfang und Komplexität der Vorlagen



Einige Einträge weichen von der erkennbaren Ursprungsgerade ab. Dies gilt beispielsweise für die Vorlage „Build-xml“, die das Skript zur Kompilierung und Installation des Versionierungssystems generiert. Diese Vorlage besteht fast ausschließlich aus statischen Bestandteilen, da das Skript nur an wenigen Stellen an die Spezifikation des Versionierungssystems angepasst werden muss. Dementsprechend besitzt diese Vorlage nur eine zyklomatische Komplexität von 2, obwohl sie die achtgrößte Vorlage ist.

Wird anstatt der Komplexität des Metaprogramms die Anzahl der Referenzen, Fallunterscheidungen oder Schleifen des Metaprogramms auf der Y-Achse aufgetragen, sind ebenfalls lineare Abhängigkeiten erkennbar. Die resultierenden Diagramme sind im Anhang A.2 aufgeführt. Für den Halstead-Aufwand ist allerdings keine lineare Abhängigkeit erkennbar. Da in diesen Messwert die Anzahl der unterscheidbaren Operatoren bzw. Operanden logarithmisch eingeht, wird eine Korrelation hier bei der Verwendung von logarithmischen Skalen sichtbar. Die Abbildung 5.3 enthält das resultierende Diagramm auf der rechten Seite.

Die gefundenen Abhängigkeiten zeigen, dass die Vorlagen jeweils einen Anteil der Generierungslogik enthalten, der ihrer Größe entspricht. Eine solche Verteilung ist auf den ersten Blick ungewöhnlich, da in Softwaresystemen üblicherweise die Anwendungslogik in zentralen Komponenten enthalten ist und andere Komponenten, etwa die Systemschnittstellen, eine geringere Komplexität besitzen. Da die Generatorvorlagen aber gerade die Generierungslogik kapseln und alle Infrastrukturkomponenten im Generator enthalten sind, der die Vorlagen ausführt, war eine solche Verteilung zu erwarten.

Die Konzentration der Komplexität auf wenige große Vorlagen ist negativ zu bewerten. Ab einem gewissen Umfang bzw. einer gewissen Komplexität sind die Vorlagen nur noch schwer verständlich und kaum vollständig testbar, da ihnen eine Strukturierung in Form von Unterprogrammen fehlt. Das Software Engineering Institute [Van00] bewertet beispielsweise Programme, deren zyklomatische Komplexität größer als 20 ist, als komplexe Programme die mit einem hohen Risiko behaftet sind. Programme deren Komplexität 50 überschreitet, werden als nicht testbar eingestuft. Deshalb sollte die Möglichkeit der VTL zur Inklusion von Untervorlagen in eine übergeordnete Vorlage genutzt werden, um mindestens die neun Vorlagen mit einer Komplexität größer als 20 auf mehrere Untervorlagen mit geringeren Komplexitäten aufzuteilen.

5.3 Quantitative Betrachtung der Generierungsergebnisse

Anhand des Beispielsystems, das bereits aus den letzten Kapiteln bekannt ist, wird nun der Umfang der Ausgaben untersucht, die durch die Generatorvorlagen getätigt werden. Dadurch soll sowohl ermittelt werden, wieviel für jede einzelne Vorlage ausgegeben wurde, als auch, wieviel für einzelne Elemente der Systemspezifikation ausgegeben wurde.

5.3.1 Ausgaben der Vorlagen

Die Tabelle 5.2 führt für jede Vorlage die Anzahl und die Größe der für diese Spezifikation generierten Dateien auf, wobei alle erzeugten Dateitypen berücksichtigt werden.

In der ersten Spalte wird zunächst erneut die Zeilenanzahl der Vorlagen angegeben, da diese Werte mit der Zeilenanzahl der generierten Dateien verglichen werden sollen. Die Spalte „Generierte Dateien“ gibt an, wie oft die Vorlage auf die Spezifikation angewendet wurde.

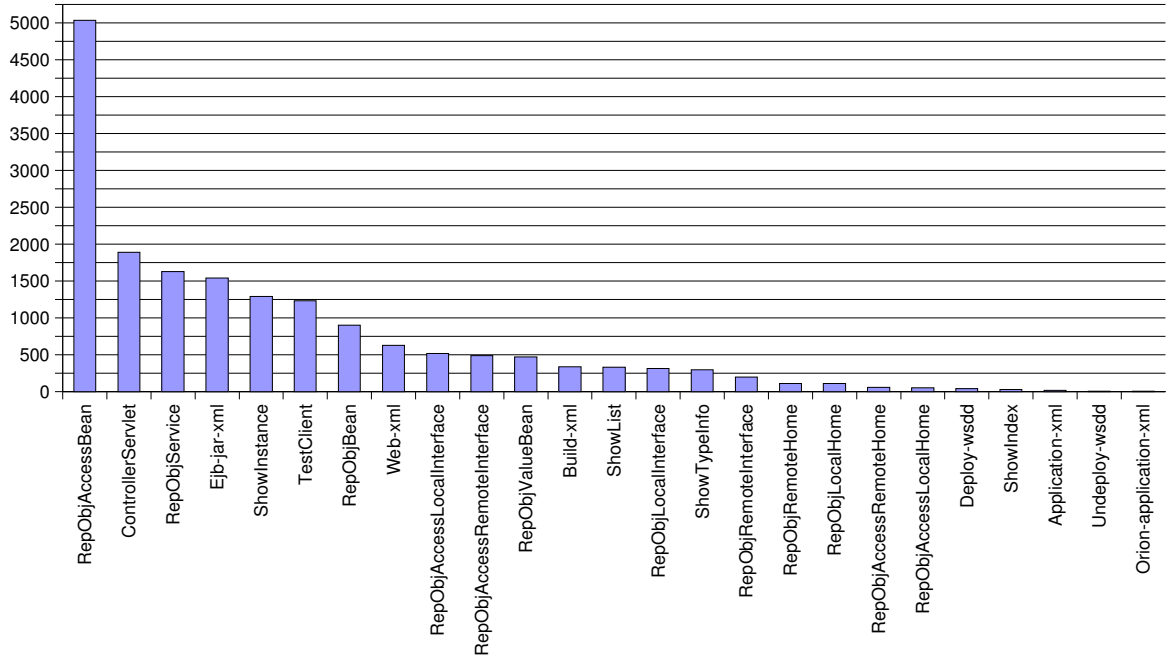
Tabelle 5.2 Generierungsergebnisse für das Beispielsystem

Vorlage	Textzeilen der Vorlage	Generierte Dateien	Textzeilen der generierten Dateien	Wiederverwendungs-faktor	Wiederverwendungs-faktor pro Datei
RepObjAccessBean	2064	5	5038	2,44	0,49
ControllerServlet	595	5	1892	3,18	0,64
TestClient	569	1	1233	2,17	2,17
RepObjService	470	5	1626	3,46	0,69
Ejb-jar-xml	436	1	1542	3,54	3,54
ShowInstance	379	5	1292	3,41	0,68
Build-xml	312	1	335	1,07	1,07
RepObjBean	312	5	899	2,88	0,58
Web-xml	220	1	626	2,85	2,85
RepObjAccessLocalInterface	184	5	519	2,82	0,56
RepObjAccessRemoteInterface	175	5	486	2,78	0,56
RepObjValueBean	133	5	468	3,52	0,70
RepObjLocalInterface	123	5	315	2,56	0,51
ShowTypeInfo	113	5	294	2,60	0,52
ShowList	103	5	331	3,21	0,64
RepObjRemoteInterface	56	5	195	3,48	0,70
ShowIndex	46	1	31	0,67	0,67
RepObjRemoteHome	32	5	113	3,53	0,71
RepObjLocalHome	31	5	108	3,48	0,70
Application-xml	22	1	15	0,68	0,68
Deploy-wsdd	19	1	41	2,16	2,16
RepObjAccessRemoteHome	11	5	55	5,00	1,00
RepObjAccessLocalHome	10	5	50	5,00	1,00
Undeploy-wsdd	5	1	7	1,40	1,40
Orion-application-xml	3	1	3	1,00	1,00
	6423	*89*	*17514*	*2,73*	*1,05*

Hier ist zwischen solchen Vorlagen zu unterscheiden, die nur einmal auf das gesamte UML-Modell angewendet werden, und solchen, die für jeden der fünf Objekttypen instantiiert werden. Dementsprechend werden eine bzw. fünf Dateien ausgegeben. Insgesamt werden für die Beispielspezifikation 89 Dateien generiert.

In der folgenden Spalte werden die Gesamtgrößen der generierten Dateien aufgeführt. Abbildung 5.4 zeigt diese Werte in einem Diagramm. Wenig überraschend ist, dass die größten Vorlagen auch die größten Ausgaben erzeugen, wie aus einem Vergleich mit der Abbildung 5.1 bzw. durch einen Vergleich der entsprechenden Spalten in der Tabelle 5.2 deutlich wird.

Abbildung 5.4 Umfang der generierten Dateien (Textzeilen)



Anhand der Größe der Ausgaben kann der Wiederverwendungsfaktor, der Quotient aus Ausgabegröße und Vorlagengröße, errechnet werden. Ein Wert von 1,0 bedeutet, dass die generierten Dateien insgesamt den gleichen Umfang haben wie die Generatorvorlagen. Bei kleineren Werten wurde eine Vorlage entwickelt, die einen größeren Umfang besitzt als die generierten Dateien. In diesen Fällen hat sich also die Entwicklung der Vorlage bezogen auf die Entwicklung eines einzigen Systems nicht rentiert. Diese Situation kann eintreten, wenn die Vorlage einen sehr hohen Anteil an VTL-Code enthält oder sehr viele alternative Implementierungen, von denen während der Generierung nur ein Variante ausgewählt wird. Bei einem Wiederverwendungsfaktor größer als 1,0 werden insgesamt mehr Textzeilen ausgeben, als in der Vorlage enthalten sind.

Abbildung 5.5 zeigt grafisch, welche Wiederverwendungsfaktoren die einzelnen Vorlagen erreichen. Nur bei zwei Vorlagen liegt der Faktor unter 1,0 und im Durchschnitt wird ein Wiederverwendungsfaktor von 2,73 erreicht. Diese Werte gelten für die einmalige Anwendung des Generators zur Generierung des Content-Management-Systems. Mit jedem weiteren generierten Versionierungssystem steigt der Wiederverwendungsfaktor und die aufwendige Entwicklung der Generatorvorlagen zahlt sich aus.

Interessant ist auch, den Wiederverwendungfaktor für jede einzelne Anwendung der Vorlage zu berechnen. Die entsprechenden Werte sind in der letzten Tabellenspalte der Abbildung 5.2 aufgeführt und in der Abbildung 5.6 visualisiert. Bei dieser Betrachtungsweise ergibt sich ein völlig anderes Bild. Bei der Generierung einer einzelnen Datei erreichen die meisten Vorlagen einen Wiederverwendungsfaktor, der deutlich kleiner als 1,0 ist. Die fünf Vorlagen mit den größten Faktoren verarbeiten die Informationen der gesamten Spezifikation und enthalten große Textabschnitte, die für jedes einzelne Spezifikationselement in die Ausgabedatei kopiert werden.

Abbildung 5.5 Wiederverwendungfaktoren für alle Anwendungen der Vorlagen

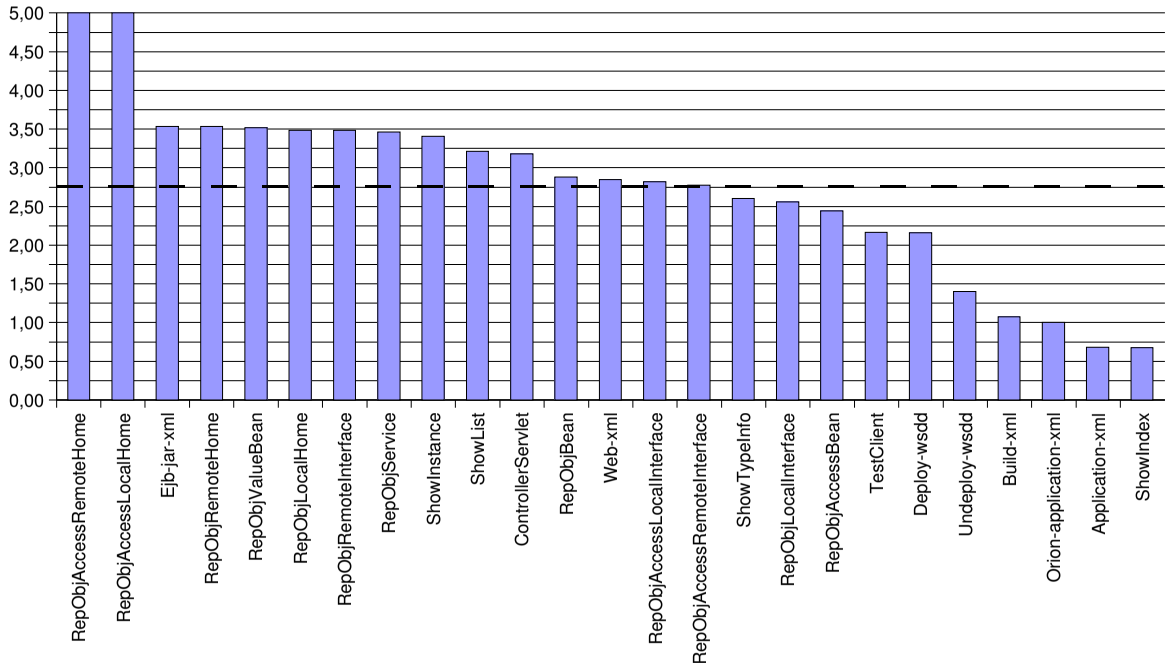
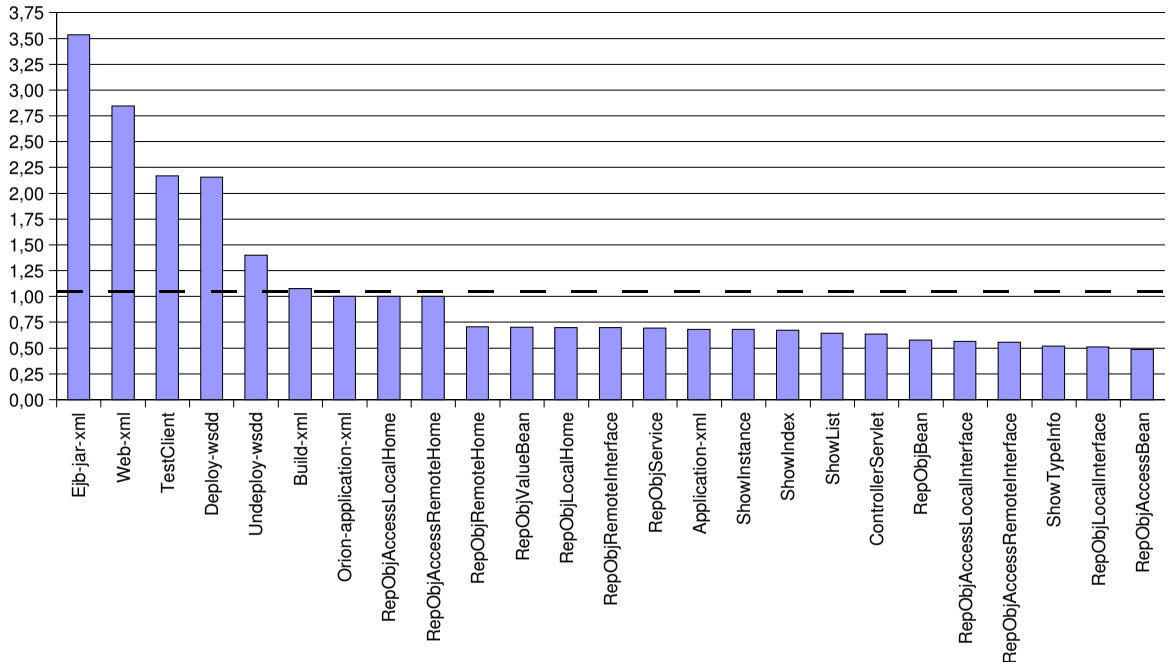


Abbildung 5.6 Wiederverwendungfaktoren für eine einzelne Anwendung der Vorlagen



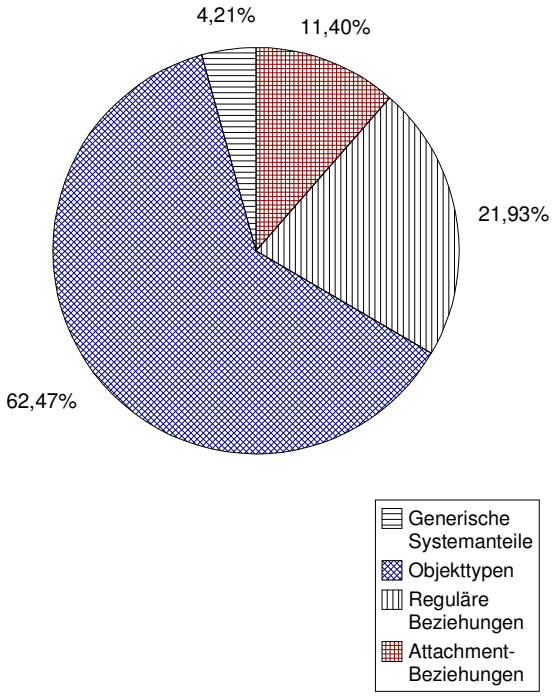
5.3.2 Zusammensetzung der generierten Implementierung

Nun soll geklärt werden, welchen Anteile an der Implementierung auf die einzelnen Objekt- und Beziehungstypen des Content-Management-Systems entfallen. Dazu wurden alle Datei-

en des Systems manuell aufgeteilt, indem generische Bestandteile von solchen Bestandteilen getrennt wurden, die einem konkreten Objekt- bzw. Beziehungstyp zugeordnet werden konnten.

Abbildung 5.7 Bestandteile der Implementierung des Content-Management-Systems

Systembestandteil	Textzeilen	Prozentualer Anteil
Generische Systemanteile	750	4,21%
Objekttypen	11139	62,47%
Article	2559	14,35%
EditorialDepartment	2372	13,30%
Image	1583	8,88%
Menu	2325	13,04%
Teaser	2300	12,90%
Reguläre Beziehungen	3910	21,93%
Hierachy	1149	6,44%
MenuContents	1268	7,11%
TeaserArticle	1178	6,61%
Illustration	315	1,77%
Attachment-Beziehungen	2032	11,40%
MenuAttachment	657	3,68%
TeaserAttachment	674	3,78%
ArticleAttachment	701	3,93%



Die Abbildung 5.7 enthält das Ergebnis dieser Zerlegung. Auf generische Bestandteile, die in jedem System unabhängig von der zugrunde liegenden Spezifikation enthalten sind, entfällt nur ein geringer Anteil von 4,21%. Den Großteil der Implementierung, nämlich 62,47%, befasst sich hingegen mit den fünf Objekttypen. Die Tabelle führt auch die auf die individuellen Objekttypen entfallenden Anteile auf. Dabei fällt auf, dass der unversionierte Objekttyp `Image` eine wesentlich kleinere Implementierung besitzt als die vier übrigen versionierten Objekttypen. Während die versionierten Objekttypen im Durchschnitt einen Implementierungsumfang von 2389 Textzeilen haben, entfallen auf `Image` mit 1583 Zeilen nur 2/3 dieses Umfangs. Die Schwankungen innerhalb der versionierten Objekttypen lassen sich mit der unterschiedlichen Attributanzahl dieser Objekttypen erklären.

Nach den Objekttypen werden in der Tabelle die Anteile der Beziehungstypen aufgeführt, wobei zwischen regulären Beziehungen und Attachment-Beziehungen unterschieden wird. Attachment-Beziehungen verbinden Arbeitskontexttypen mit den enthaltenen Objekttypen und besitzen eine einfachere Implementierung als die mit gleitenden Beziehungsenden ausgestatteten regulären Beziehungstypen des Informationsmodells. Deren Implementierung hat mit durchschnittlich 1198 Zeilen fast den doppelten Umfang. Eine Ausnahme bildet der Beziehungstyp `Illustration`, der die Objekttypen `Article` und `Image` verbindet. Dieser Beziehungstyp kann nur in Richtung des Objekttyps `Image` navigiert werden und

besitzt auch keine gleitenden Beziehungsenden, da `Image` nicht versioniert wird. Dementsprechend fällt seine Implementierung mit 315 Textzeilen sehr kompakt aus.

Die in Abbildung 5.7 sichtbaren Verhältnisse zeigen, dass es möglich ist, den Umfang der Systemimplementierung aus dem Umfang der Systemspezifikation zu errechnen. Dazu muss allerdings exakter bekannt sein, wieviel ein Spezifikationselement in den verschiedenen zulässigen Konfigurationen zum Umfang der Implementierung beiträgt.

Diese Werte können ermittelt werden, indem entsprechende Elemente zu einer bestehenden Spezifikation hinzugefügt werden und die Differenz zwischen dem Umfang der ursprünglichen und der erweiterten Implementierung gemessen wird. Die Tabelle 5.3 zeigt die Ergebnisse dieser Messungen für einige ausgewählte Konfigurationen von Spezifikationselementen. Zusätzlich zur Ermittlung der Umfangsänderungen wurden auch die bekannten Metriken (Anweisungsanzahl, zyklomatische Komplexität, Halstead-Aufwand) auf die in den generierten Softwaresystemen enthaltenen Java-Klassen angewendet. Die ermittelten Differenzen dieser Messwerte sind ebenfalls in der Tabelle 5.3 enthalten.

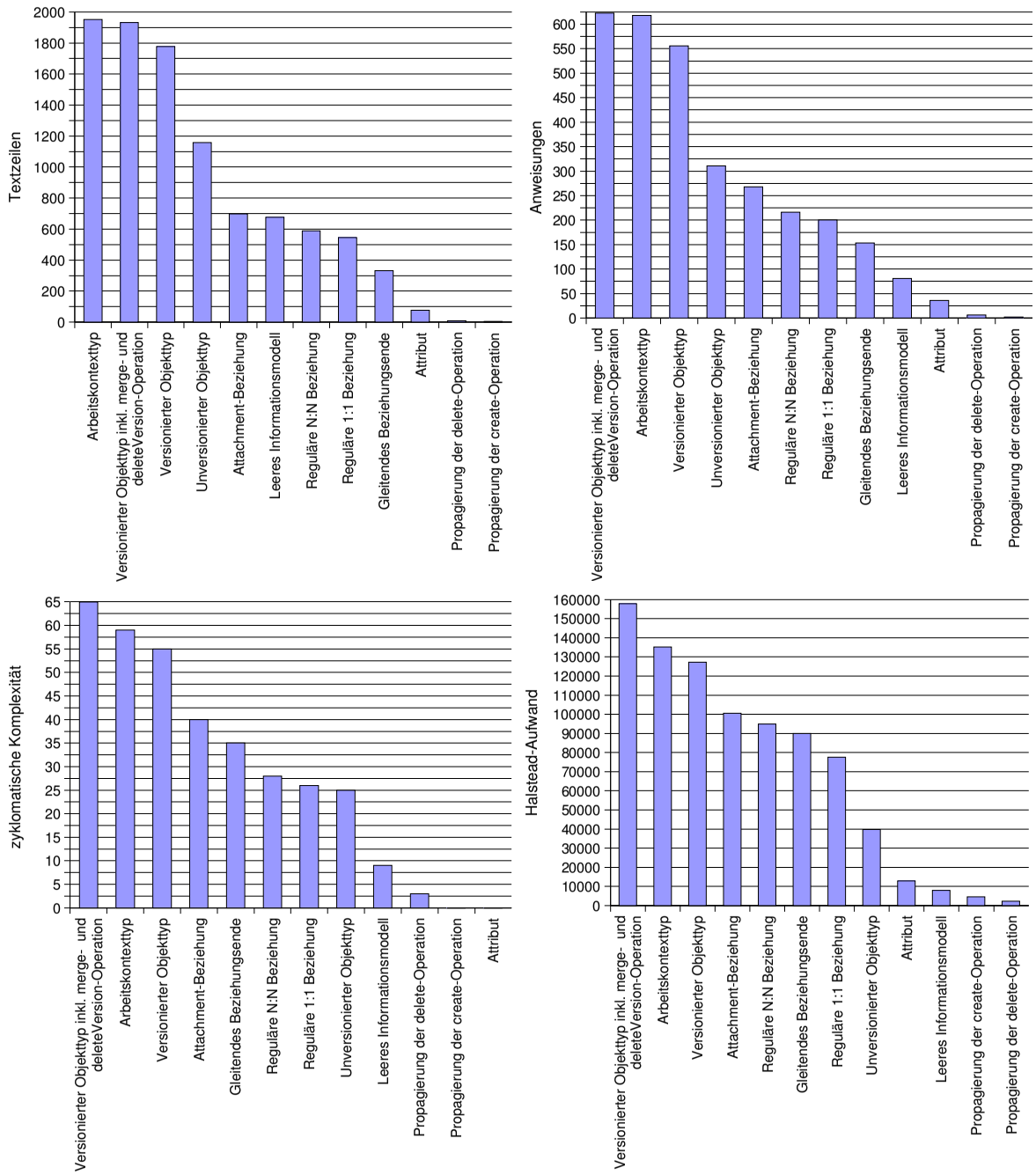
Tabelle 5.3 Implementierungsumfang verschiedener Spezifikationselemente

Spezifikationselement	Textzeilen	Anweisungen	Zyklomatische Komplexität	Halstead-Aufwand
Leeres Informationsmodell	677	81	9	7932,75
Unversionierter Objekttyp	1159	311	25	39807,96
Versionierter Objekttyp	1778	556	55	127241,85
Versionierter Objekttyp inkl. merge- und deleteVersion-Operation	1932	623	65	157922,10
Arbeitskontexttyp	1951	618	59	135228,55
Attribut	76	36	0	12923,27
Attachment-Beziehung	697	268	40	100544,21
Reguläre 1:1 Beziehung	545	200	26	77519,26
Reguläre N:N Beziehung	589	216	28	94867,00
Gleitendes Beziehungsende	331	153	35	90017,46
Propagierung der delete-Operation	9	6	3	2352,63
Propagierung der create-Operation	5	2	0	4545,50

In der ersten Zeile der Tabelle sind Umfang und Komplexität eines Versionierungssystems mit einem leeren Informationsmodell aufgeführt. Die angegebenen Messwerte beziehen sich also auf die Systemanteile, die in jedem System der Produktlinie enthalten sind. Der Umfang bzw. die Komplexität eines konkreten Systems kann errechnet werden, indem zu den Werten der ersten Tabellenzeile für jedes Element der Systemspezifikation die zugehörigen Werte aus den weiteren Tabellenzeilen hinzuaddiert werden.

In den nächsten vier Zeilen der Tabelle werden die Beträge aufgeführt, um die sich eine Systemimplementierung vergrößert, wenn ein einzelner Objekttyp ohne Attribute in die Spezifikation eingefügt wird. Wie bereits bei der Analyse des Content-Management-Systems festgestellt wurde, beanspruchen versionierte Objekttypen wesentlich mehr Textzeilen als unversionierte Typen. In der grafischen Darstellung der Messergebnisse in Abbildung 5.8 wird deutlich, dass dieser Unterschied noch stärker ausgeprägt ist im Bezug auf die Komplexitäten und die Halstead-Aufwände.

Abbildung 5.8 Grafische Darstellung der Messergebnisse aus Abbildung 5.3



Aber auch die über das UML-Profil festgelegte globale Konfiguration des Systems beeinflusst die Größe und Komplexität der Implementierung eines einzelnen Objekttyps. Die in der vierten Zeile enthaltenen Werte zeigen, dass die optionalen merge- und deleteVersion-Operationen die Implementierung eines versionierten Objekttyps von 1778 auf 1932 Textzeilen vergrößern und deren zyklomatische Komplexität von 55 auf 65 erhöhen. Ein weitere Einflussgröße auf die Implementierung eines Objekttyps sind dessen Attribute. Auf jedes Attribut entfallen aber nur 76 zusätzliche Textzeilen, die Aufnahme zusätzlicher Attribute

in ein bestehendes Informationsmodell vergrößert also die Systemimplementierung nur unwesentlich. Die zyklomatische Komplexität wird durch die Attributsanzahl überhaupt nicht beeinflusst.

Größeren Einfluss haben hingegen die Beziehungstypen. Für die Hinzufügung einer Attachment-Beziehung zu einem bestehenden Informationsmodell wurde ein Zuwachs von rund 700 Zeilen ermittelt und damit wurden auch in diesem Punkt die Ergebnisse aus Abbildung 5.7 bestätigt. Für reguläre Beziehungen wurde zunächst ermittelt, wie viele Textzeilen durch 1:1- bzw. N:M-Beziehungen ohne gleitende Beziehungsenden oder sonstige Merkmale verursacht werden. Diese Werte erhöhen sich um 331 Zeilen für jedes gleitende Beziehungsende. Andere Merkmale der Beziehungsenden, wie beispielsweise die Propagierung der delete- bzw. create-Operation verursachen hingegen nur minimale Vergrößerungen der Implementierung.

Die Betrachtung der Komplexitäts- und Aufwandswerte ergibt, dass die Beziehungstypen insgesamt einen höheren Einfluss auf die Komplexität eines Versionierungssystems haben können als die Objekttypen. Die Implementierung einer N:M-Beziehung mit zwei gleitenden Beziehungsenden besitzt nämlich eine zyklomatische Komplexität von 98, wohingegen auf einen versionierten Objekttyp nur ein Wert von 65 entfällt.

Kapitel 6 Leistungsuntersuchungen

In diesem Kapitel sollen die Leistungseigenschaften der entwickelten generativen Implementierung der Versionierungssysteme anhand mehrerer Leistungstests untersucht und mit einer alternativen Implementierung verglichen werden, die auf der Basis eines generischen Frameworks entwickelt wurde.

Bevor die Ergebnisse der Leistungsmessungen vorgestellt und interpretiert werden, wird nun zunächst die generische Implementierung beschrieben und die Unterschiede zur generierten Implementierung zusammengefasst. Anschließend wird der für die Messungen verwendete Leistungstest und die Systemumgebung beschrieben.

6.1 Eine generische Implementierung der Produktlinie

Parallel zur Entwicklung des Generators und der zugehörigen Vorlagen wurde ein weiterer Ansatz zur Implementierung der Produktlinie für Versionierungssysteme verfolgt. Um eine Vergleichsgrundlage für die Bewertung des generativen Lösung zur Verfügung zu stellen, wurde hier auf den Einsatz eines Generators verzichtet und stattdessen die Produktlinie auf der Basis eines Frameworks realisiert. Dieses Framework enthält eine generische Implementierung der im Merkmalmodell definierten Funktionalitäten eines Versionierungssystems. Für die Realisierung eines Versionierungssystems im Rahmen der Produktlinie werden die dem Informationsmodell entsprechenden Schnittstellen entwickelt und das Framework entsprechend dem gewünschten Verhalten konfiguriert. Dieses geschieht teilweise zur Compilezeit und teilweise zur Laufzeit durch die Speicherung von Konfigurationseinstellungen in der Datenbank. Der entscheidende Unterschied zum generativen Ansatz ist, dass auf eine Optimierung der Implementierung auf ein spezielles Informationsmodell und die damit verbundene Duplizierung von Implementierungsanteilen verzichtet wird. Der Framework-Ansatz entspricht den von kommerziellen Herstellern üblicherweise gewählten Lösungen für Versionierungsysteme.

Die auf der Basis des Frameworks entwickelten Versionierungssysteme bieten die gleichen Systemschnittstellen wie die generierten Versionierungsdienste an und besitzen ebenfalls eine dreischichtige Systemarchitektur. Allerdings sind die Sitzungs- und Persistenzschicht hier, den Bedürfnissen eine generischen Implementierung entsprechend, anders strukturiert.

Auf der Ebene der Persistenzschicht besteht der wichtigste Unterschied darin, dass alle Beziehungen und die Versionierungsinformationen getrennt von den Objektdaten in generischen Tabellen des Datenbankschemas verwaltet werden. Dies ermöglicht in der Sitzungsschicht eine generische Implementierung der Operationen zur Beziehungs- und Versionsver-

waltung. Diese generische Implementierung enthält dabei die Informationen über die Konfiguration der Beziehungstypen und der gewünschte Versionierungssemantik aus Tabellen des Datenbankschemas, die zur Laufzeit mit den Spezifikationsinformationen gefüllt werden. Konsequenz dieses Entwurfs ist, dass die Datenhaltung weniger effizient ist, als im Fall der generierten Implementierung, da die Gegebenheiten des Informationsmodells, insbesondere die Multiplizitäten der Beziehungen, nicht für Optimierungen ausgenutzt werden können. So werden beispielsweise auch für unversionierte Objekttypen Versionsinformation verwaltet und auch solche Beziehungstypen, die mit Hilfe von einfachen Fremdschlüsselattributen implementiert werden könnten, werden über Zwischentabellen realisiert.

Abbildung 6.1 Datenbanktabellen zur generischen Beziehungsverwaltung

```
create table genericvs_Relationship (
  relId varchar(255) not null primary key,
  origGlobalId integer default null,
  origObjectId integer default null,
  origVersionId integer default null,
  destGlobalId integer default null,
  destObjectId integer default null,
  destVersionId integer default null,
  destRole varchar(255) default null)

create table genericvs_PinningSetting (
  pinningSettingId varchar(255) not null primary key,
  origGlobalId integer default null,
  origObjectId integer default null,
  origVersionId integer default null,
  destGlobalId integer default null,
  destObjectId integer default null,
  destVersionId integer default null,
  destRole varchar(255) default null)

create table genericvs_RelEnd (
  relEndId varchar(255) not null primary key,
  origObjectType varchar(255) default null,
  destObjectType varchar(255) default null,
  destRole varchar(255) default null,
  propCreate char(1),
  propDelete char(1),
  propFreeze char(1),
  propAttachDetach char(1),
  propCopy char(1),
  propCreateSuccessor char(1),
  propCheckoutCheckin char(1))
```

Abbildung 6.1 zeigt die generischen Tabellen, die für die Implementierung regulärer Beziehungstypen verwendet werden. Die Tabelle `genericvs_Relationship` enthält einen Eintrag für jedes Paar von Objektversionen, das miteinander verbunden ist, also die Kandidatenmengen aller Beziehungen. Um die verschiedenen Kandidatenmengen unterscheiden zu können, kennzeichnet das Attribut `destRole` die Rolle von einem der beteiligten Beziehungsenden. Während der Navigation einer Beziehung werden also nur die Tabelleneinträge mit der passenden Typkennzeichnung beachtet.

Die Vorauswahl von Objektversionen in den Kandidatenmengen wird über die Tabelle `genericvs_PinningSetting` realisiert, die einen Eintrag für jede vorausgewählte Version enthält. Auch hier wird über das Attribut `destRole` gekennzeichnet, auf welche Rolle sich ein Tabelleneintrag bezieht. Auf eine explizite Speicherung der neusten Objekt-

versionen einer Kandidatenmenge wurde in der generischen Implementierung verzichtet, stattdessen wird diese Information bei Bedarf während der Durchführung der Navigation ermittelt.

Die Informationen über die im Informationsmodell vorhanden Beziehungstypen und Beziehungsenden und deren Kardinalität erhält das Framework durch die Implementierung der Sitzungskomponenten zur Compilezeit. Diese Konfiguration ist im Beispiel 6.1 am Beispiel der Beziehung `Illustration` zwischen den Objekttypen `Article` und `Image` zu sehen.

Beispiel 6.1 Framework-basierte Implementierung der Sitzungskomponenten

```
public class ArticleAccessBean extends VSObjectAccessBean {
  [...]
  public java.lang.String getTitle(int globalId) throws FinderException {
    return (String)getProperty(globalId, "title");
  }

  public void setTitle(int globalId, java.lang.String newTitle)
                      throws FinderException, EJBException {
    setProperty(globalId, "title", newTitle);
  }

  [...]

  public int[] getImage(int globalId) throws Exception {
    return getObject(globalId, "image");
  }

  public void addImage(int globalId, int otherId)
                      throws FinderException, EJBException {
    addObject(globalId, null, '*', otherId, "image", '*', "Image");
  }

  public void removeImage(int globalId, int otherId)
                        throws FinderException, EJBException {
    removeObject(globalId, "article", otherId, "image");
  }
  [...]
}
```

Die Informationen über die vorgesehene Propagierung von Operation über die Beziehungen erhält das Framework hingegen zur Laufzeit aus der Tabelle `genericvs_RelEnd`. Diese Tabelle enthält boolsche Attribute, die für jedes Beziehungsende definieren, ob ein bestimmte Operation propagiert werden soll. Bei der Ausführung einer Operation muss das Framework also stets für alle mit dem Objekttyp verbunden Beziehungstypen überprüfen, ob die Tabelle `genericvs_RelEnd` eine Propagierung der Operation vorsieht.

Die Speicherung der Attachment-Beziehungen erfolgt in ähnlichen zentralen Tabellen, die jeweils die Primärschlüssel der beteiligten Objektversionen enthalten und eine Typkennzeichnung um die Einträge verschiedenere Beziehungstypen unterscheiden zu können.

Ein Datenbankschema für ein konkretes Versionierungssystem entsteht aus dem generischen Schema dadurch, dass für jeden Objekttyp eine Tabelle eingefügt wird, die die Attribute des Objekttyps aufnimmt.

Die Sitzungsschicht besteht weiterhin aus Sitzungskomponenten für jeden Objekttyp des Versionierungssystems. Allerdings erben diese ihre Implementierung von einer generischen Superklasse des Frameworks. Dies reduziert den systemspezifische Teil der Sitzungsschicht

auf die Implementierung von Wrapper-Klassen, die an dem Informationsmodell entsprechende Zugriffsmethoden für Attribute und Beziehungen bieten, indem sie die entsprechenden Anfragen in Aufrufe der generischen Zugriffsmethoden der generischen Implementierung umsetzen. Das Beispiel 6.1 zeigt einen Ausschnitt einer solchen Wrapper-Klasse für den bekannten Objekttyp `Article`.

Tabelle 6.1 Vergleich von generischer und generierter Implementierung

	generische Implementierung	**generierte Implementierung**
versionierte Objekttypen	ja	ja
unversionierte Objekttypen	nein	ja
Versionsverwaltung	Speicherung des Versionsgraphen in zentralen Tabellen.	Speicherung des Versionsgraphen über Fremdschlüsselattribute der Objekttabellen bzw. gesonderte Zwischentabellen für jeden Objekttyp.
Verwaltung regulärer Beziehungen	Speicherung der Kandidatenmengen und der vorausgewählten Versionen in zentralen Tabellen. Die Zuordnung zu den Beziehungstypen erfolgt über eine Typkennung.	Speicherung der Kandidatenmengen und der vorausgewählten Versionen über Fremdschlüsselattribute der Objekttabellen bzw. gesonderte Zwischentabellen für jeden Objekttyp. Gesonderte Vorhaltung der neusten Objektversionen für eine beschleunigte Navigation.
gleitende Beziehungsenden	ja	ja
nicht-gleitende Beziehungsenden	nein	ja
Operations-propagierung	Die Propagierung erfolgt dynamisch anhand der in der Datenbank abgelegten Konfigurationsoperationen.	Die Propagierung ist fest vorgeben durch entsprechende Logik in den Sitzungskomponenten der an der Beziehung beteiligten Objekttypen.
Verwaltung von Attachment-Beziehungen	Speicherung der Attachment-Beziehungen und der zugehörigen checkout-Informationen in zentralen Tabellen. Die Zuordnung zu den Beziehungstypen erfolgt über eine Typkennung.	Speicherung der Attachment-Beziehungen und der zugehörigen checkout-Informationen über Fremdschlüsselattribute der Objekttabellen bzw. gesonderte Zwischentabellen für jeden Objekttyp.
Implementierung der Sitzungs-komponenten	Sitzungskomponenten erben ihre Anwendungslogik von Superklassen des Frameworks, wo diese mit geigneten Mitteln (u.A. der Reflection-API) implementiert ist.	Duplizierung und Anpassung der Anwendungslogik durch den Generator.

In der Tabelle 6.1 werden die wichtigsten Unterschiede der durch den generativen Ansatz und den Framework-Ansatz erzeugten Systemimplementierungen abschließend zusammengefasst.

6.2 Beschreibung des Leistungstests

Die im Rahmen dieser Arbeit durchgeführten Leistungsuntersuchungen basieren auf der Ausführung eines standardisierten Leistungstests in der Form eines Client-Programms für das aus den vorrangegangenen Kapiteln bekannte Content-Management-System. Dieser Client simuliert die Zugriffe auf ein CMS durch die verschiedenen Benutzergruppen. Zunächst legt er einige initiale Objektdaten an, die die Inhalte einer Nachrichten-Webseite wiederspiegeln. Anschließend simuliert er abwechselnd Lesezugriffe auf diese Daten durch Nutzer der Webseite und Veränderungen der Webseite durch Redakteure, die Objekte verändern, löschen und neu anlegen. Damit wird versucht, ein realistisches Nutzungsprofil zu nachzubilden.

Der Client kann sowohl die Webservice-Schnittstelle der zu testenden Versionierungssysteme ansteuern, als auch direkt auf deren Sitzungskomponenten zugreifen. Für jede aufgerufene Operation protokolliert er die Ausführungszeit in einer Log-Datei, die anschließend durch ein Analyseprogramm aufbereitet wird. Von den ermittelten Ausführungszeiten wird dabei ein Zeitbetrag abgezogen, der für den Aufruf einer speziellen leeren Operation gemessen wurde. Ziel ist es, die reinen Rechenzeiten zu ermitteln. Die als Referenz verwendete leere Operation der Sitzungskomponenten enthält keinerlei Anweisungen oder Parameter, die Ausführungszeit dieser Operation entspricht deshalb der Kommunikationszeit, die für jeden beliebigen Aufruf der Sitzungsschicht anfällt. Für die verwendete Testumgebung wurde ein Wert von 186 μs ermittelt.

Insgesamt führt ein Durchlauf dieses Leistungstests 192148 Operationen aus. Dabei werden 976 unterschiedliche Objekte angelegt. Nach der Beendigung des Durchlaufs verbleiben insgesamt 1164 Objektversionen im Versionierungsystem. Falls nicht anders angegeben, wurden die Messungen stets auf neu installierten Versionierungssystemen ausgeführt, in denen noch keine Objekte angelegt wurden.

6.3 Beschreibung der Testumgebung

Als Systemumgebung für die Leistungsmessungen wurde ein Linux-Server verwendet, dessen Hard- und Softwarekonfiguration in den nachfolgenden Tabellen dargestellt ist.

Hardwarekonfiguration des Testsystems	
Prozessoren:	2x Intel Pentium III, 866 MHz Taktfrequenz
Hauptspeicher:	1 Gb
Massenspeicher:	2x 80 Gb IDE-Platten, gespiegelt (RAID)

Softwarekonfiguration des Testsystems	
Prozessoren:	SuSE-Linux 9.0
Betriebssystemkern:	Linux 2.4.21
Java-Laufzeitumgebung:	Sun-SDK 1.4.2
Anwendungsserver:	Ironflare Orion Server 2.0.2
Datenbanksystem:	IBM DB2 8.1.0

Es wurde durch die Abschaltung aller nicht benötigten Systemdienste die notwendigen Maßnahmen getroffen, um störende Einflüsse auf den Messprozess zu minimieren.

6.4 Auswertung der Messergebnisse

In diesem Abschnitt werden nun die vier durchgeführten Leistungsmessungen erläutert und deren Ergebnisse interpretiert.

6.4.1 Test der generierten Implementierung

In einem ersten Schritt wurde das Testprogramm auf die durch den Softwaregenerator generierte Implementierung angewendet. Getestet wurde dabei der direkte Zugriff auf die Sitzungschicht des Versionierungssystems.

Tabelle 6.2 Ergebnisse für den Objekttyp Article

Operation	Operations-aufrufe	Gesamte Rechenzeit (µs)	Minimale Rechenzeit (µs)	Maximale Rechenzeit (µs)	Durchschnittliche Rechenzeit (µs)	Durchschnittliche Abweichung (µs)	Median der Rechenzeit (µs)	Durchschnittliche Abweichung (µs)
deleteObject	36	3674157	53360	219779	102059,9	29645,4	109109,0	28746,5
createObject	64	5148170	52964	1182896	80440,2	40072,6	56468,0	25858,1
createSuccessor	840	46482822	41890	825755	55336,7	9677,7	50260,0	8338,4
findAll	16	825749	8377	109572	51609,3	12134,9	49938,5	11883,7
attachToEditorialDepartment	168	3379288	15423	54002	20114,8	2311,6	19671,5	2281,1
freeze	504	3837941	5818	36803	7615,0	1065,0	7200,0	915,3
merge	168	1080878	5563	36733	6433,8	835,9	6007,0	621,2
checkoutToEditorialDepartment	684	3696088	4240	57412	5403,6	1421,3	4626,0	961,3
checkinFromEditorialDepartment	684	3287560	4125	34145	4806,4	609,7	4508,0	476,8
setValueObject	1716	6078087	3202	48223	3542,0	404,1	3320,0	250,1
addImage	1148	3955977	2138	30091	3446,0	614,0	3246,0	518,9
setDate	64	221191	2469	22319	3456,1	1006,9	2797,5	908,0
setTitle	1844	4361869	2123	28586	2365,4	240,8	2237,0	160,4
setText	1780	3401956	1741	25902	1911,2	197,2	1800,0	131,1
getValueObject	3108	4311109	903	44512	1387,1	370,8	1262,0	298,9
getSuccessors	3264	6567178	657	28453	2012,0	1516,3	936,0	1248,6
getTeaser	268	287921	800	6648	1074,3	295,5	896,0	196,2
getImage	3276	4472907	719	59887	1365,4	685,5	861,0	576,0
getAncestors	3264	4584953	708	26376	1404,7	734,6	803,0	638,7
getAlternatives	3264	2659427	635	25154	814,8	139,1	741,0	95,9
getRoot	3432	1974651	344	72042	575,4	96,5	532,0	81,7
getTitle	4824	2115643	146	24719	438,6	112,7	456,0	110,0
isFrozen	3264	1034852	52	23766	317,1	122,3	366,0	100,2
setCurrentEditorialDepartment	1756	551678	161	2525	314,2	163,9	181,0	143,7
unsetCurrentEditorialDepartment	1802	202602	10	1390	112,4	13,1	105,0	10,8
	41238	*118194654*						

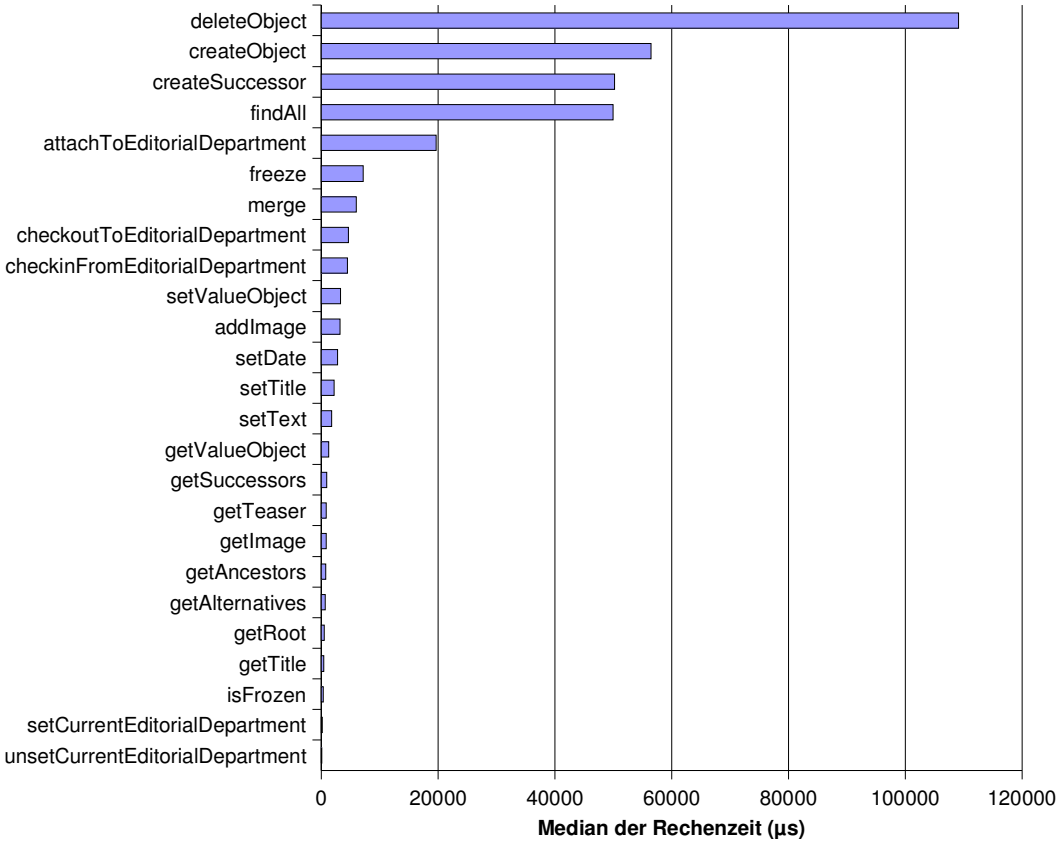
Durch diese Messung sollte geklärt werden, welche Rechenzeiten auf die einzelnen Operationen der Sitzungskomponenten entfallen, und zwar sowohl im Bezug auf einen einzelnen Aufruf als auch auf die Gesamtzeit aller Aufrufe. Ausserdem dienen die in dieser Messung ermittelten Werte als Referenz für die nachfolgenden Messungen, die jeweils Vergleiche zwischen verschiedenen Durchläufen des Testprogramms beinhalten.

In der Tabelle 6.2 sind die Messergebnisse für alle Operationen des Objekttyps `Article` aufgeführt. Die Messergebnisse für alle anderen Objekttypen sind im Anhang B.1 enthalten.

Die erste Spalte dieser Tabelle enthält die Anzahl der Aufrufe, die auf die einzelnen Operationen entfallen sind. Aus den Zahlen geht hervor, dass der Schwerpunkt des Test auf dem Auslesen von Objektdaten und der Navigation von Beziehungen bzw. des Versionsgraphen liegt. Diese Schwerpunktsetzung basiert auf der Annahme, dass Daten typischerweise häufiger gelesen als geschrieben werden, die zumindestens für Content-Management-Systeme zutrifft.

In der zweiten Spalte werden die akkumulierten Rechenzeiten für alle Aufrufe einer Operation aufgeführt. Die häufig genutzten Operationen zum Lesen von Daten erreichen hier vergleichsweise hohe Werte obwohl einzelne Aufrufe dieser Operationen im Vergleich zu anderen Operationen schnell bearbeitet werden.

Abbildung 6.2 Grafische Darstellung der Rechenzeiten für den Objekttyp Article



In den folgenden Spalten der Tabelle werden die Rechenzeiten für einen einzelnen Operationsaufruf analysiert. Aufgeführt wird neben dem Durchschnitt und dem Mittelwert der

Rechenzeit auch die Abweichungen der Messwerte von diesen Werten. Hierbei zeigt sich, dass der Mittelwert besser geeignet ist, da er durch aus Einschwingvorgängen und Störungen resultierenden Extremwerte nicht beeinflusst wird. In Abbildung 6.2 sind die Mittelwerte der Rechenzeiten grafisch dargestellt. Das Diagramm verdeutlicht, dass die Operationen zum Anlegen und Löschen von Objekten bzw. Objektversionen die höchsten Einzelzeiten erreichen. Teuer ist auch die Operation `findAll`, die den gesamten Inhalt einer Objekttabelle zurückliefert. Diese Tabelle wächst während der Durchführung des Leistungstest stark an und dementsprechen auch die Ausführungszeiten von `findAll`. Die im Anhang B.1 enthalten Diagramme für die anderen Objekttypen des Systems zeigen ein ähnliches Bild.

6.4.2 Wiederholte Durchführung des Leistungstests

In der nächsten Messung wurde untersucht, wie sich die Rechenzeiten verändern, wenn der Leistungstest mehrfach hintereinander ausgeführt wird, und somit die Objekt- und Beziehungstabellen des Versionierungssystems bereits gefüllt sind.

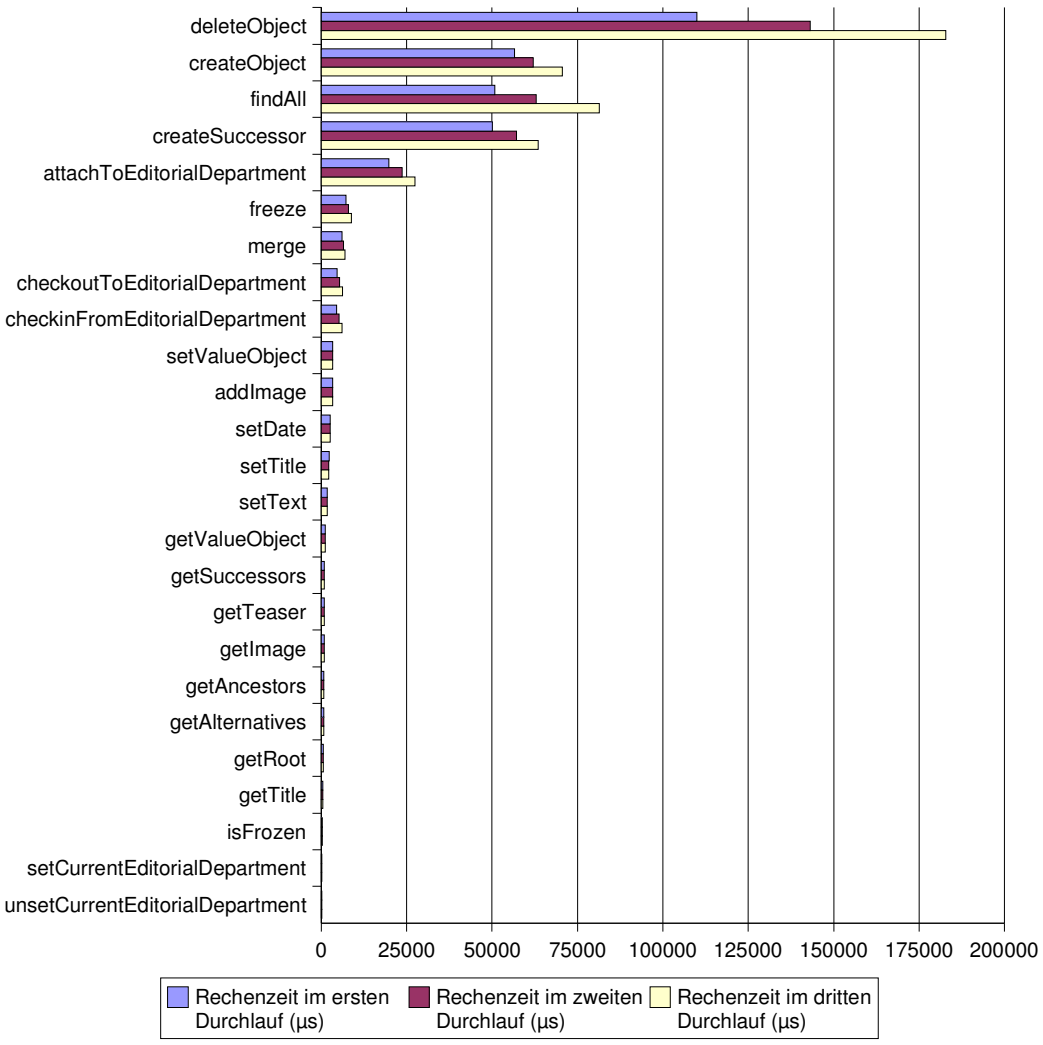
Tabelle 6.3 Ergebnisse für den Objekttyp Article

Operation	Rechenzeit im ersten Durchlauf (µs)	Rechenzeit im zweiten Durchlauf (µs)	Rechenzeit im dritten Durchlauf (µs)
deleteObject	110001,0	143125,5	182755,5
createObject	56530,0	61968,0	70594,5
findAll	50742,0	62914,0	81352,5
createSuccessor	50004,5	57180,0	63517,0
attachToEditorialDepartment	19748,0	23724,0	27451,5
freeze	7197,5	7946,5	8753,5
merge	6016,5	6567,0	6952,0
checkoutToEditorialDepartment	4624,0	5378,0	6190,5
checkinFromEditorialDepartment	4479,5	5249,0	6038,0
setValueObject	3307,0	3292,0	3304,0
addImage	3277,0	3259,0	3267,0
setDate	2619,5	2554,5	2557,0
setTitle	2253,0	2246,0	2243,0
setText	1801,0	1794,0	1796,0
getValueObject	1242,0	1220,0	1232,0
getSuccessors	925,0	923,0	921,0
getTeaser	894,0	880,0	876,0
getImage	856,0	812,0	809,0
getAncestors	796,0	789,0	787,0
getAlternatives	732,0	723,0	723,0
getRoot	534,0	529,0	528,0
getTitle	454,0	424,0	424,0
isFrozen	362,0	366,0	367,0
setCurrentEditorialDepartment	182,0	181,0	183,0
unsetCurrentEditorialDepartment	101,0	101,0	101,0

Zu erwarten war, dass die Zeiten insbesondere für die `findAll`-Operation stark ansteigen, da diese mit jedem Durchlauf des Leistungstests mehr Objekte auffinden und zurückgeben muss. Die in Tabelle 6.3 enthaltenen Messwerte bestätigen diese Erwartung. Auch die anderen aufwendigen Operationen zum Anlegen und Entfernen von Objekten, die den Cache des Anwendungsservers nicht ausnutzen können, werden während der drei in der Tabelle dokumentierten Testdurchläufe deutlich langsamer. Diese Operationen müssten optimiert werden, um ein akzeptables Leistungsverhalten des Versionierungssystems für große Datenmengen zu erreichen.

Die Mehrzahl der Operationen hingegen zeigen keine Verlangsamung. Dies liegt daran, dass im zweiten und dritten Durchlauf des Leistungstest abgesehen von der `findAll`-Operation nicht auf Objekte zugegriffen wird, die in den vorangegangenen Durchläufen angelegt wurden. Dieser Umstand ermöglicht es dem Anwendungsserver, die Objekte des aktuellen Durchlaufs im Cache zu halten, wodurch die Ausführungszeiten von Operationen wie beispielsweise `getTitle` konstant bleiben.

Abbildung 6.3 Grafische Darstellung der Rechenzeiten für den Objekttyp Article



In Abbildung 6.3 wird die Verlangsamung der einzelnen Operationen grafisch dargestellt. Die Messergebnisse und Diagramme für die anderen Objekttypen des Content-Management-Systems sind im Anhang B.2 enthalten.

6.4.3 Test der Webservice-Schnittstelle

Im Zuge der Beschreibung der webbasierten Systemschnittstellen im Abschnitt 4.2.3 wurde bereits erwähnt, dass die Webservice-Schnittstellen der generierten Versionierungssysteme systembedingt relativ schlechte Leistungseigenschaften besitzen. Dies liegt daran, dass sie sich eines Kommunikationsprotokolls bedienen, das auf dem Versand von XML-Dokumenten basiert, die aufwendig erzeugt und analysiert werden müssen.

In dieser Messung soll überprüft werden, wie groß diese Verlangsamung ausfällt im Vergleich zum direkten Zugriff auf die Sitzungskomponenten, wozu ein binäres Protokoll verwendet wird. Deshalb wurde für beide Zugriffsmethoden die gesamte Ausführungszeit inklusive der anfallenden Kommunikationskosten bestimmt. Tabelle 6.4 enthält diese Zeiten für den Objekttyp `Image`, die entsprechenden Tabellen für die übrigen Objekttypen können im Anhang B.3 eingesehen werden.

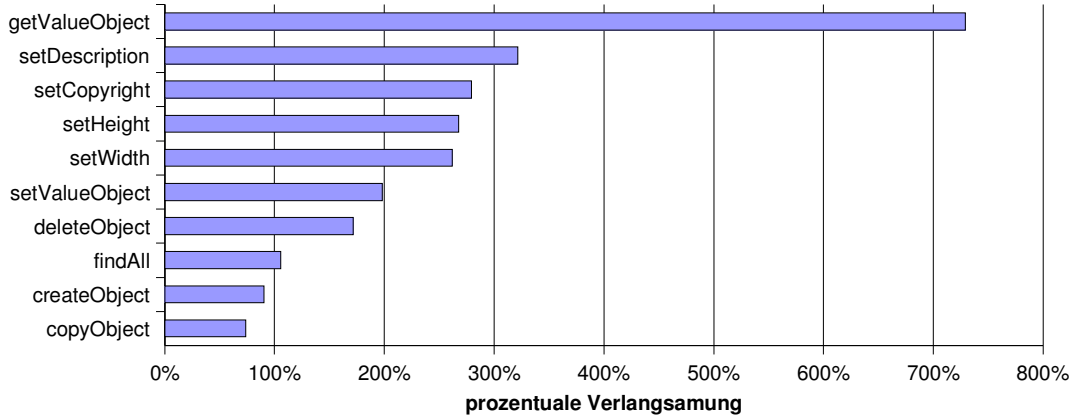
Tabelle 6.4 Ergebnisse für den Objekttyp Image

Operation	Aufrufdauer für nativen Client (µs)	Aufrufdauer für Webservice-Client (µs)	Verlangsamung	prozentuale Verlangsamung
findAll	10764,0	22128,0	11364,0	105,6%
setCopyright	2794,0	10602,5	7808,5	279,5%
setWidth	2975,5	10765,5	7790,0	261,8%
setHeight	2884,0	10606,5	7722,5	267,8%
setDescription	2399,5	10117,5	7718,0	321,7%
createObject	8130,5	15487,5	7357,0	90,5%
getValueObject	938,5	7780,0	6841,5	729,0%
setValueObject	3151,0	9393,5	6242,5	198,1%
copyObject	7879,5	13672,0	5792,5	73,5%
deleteObject	3306,5	8981,0	5674,5	171,6%

Neben den absoluten Ausführungszeiten wird in den letzten beiden Spalten der Tabelle die relative Verlangsamung der Ausführung durch die Verwendung der Webservice- anstatt der nativen Schnittstelle aufgeführt. Bei der Betrachtung der Zahlen fällt auf, das für die `findAll`-Operation eine deutlich höhere Verlangsamung als für die übrigen Operationen eintritt. Dies ist damit zu erklären, dass zur Übermittlung der umfangreichen Objekt- bzw. Schlüsselliste, die die `findAll`-Operation zurückliefert, sehr große XML-Dokumente erzeugt und analysiert werden müssen. Die anderen getesteten Operationen liefern hingegen nur Einzelwerte zurück.

Interessant ist auch, die Verlangsamung als prozentualen Wert relativ zur absoluten Ausführungszeit der Operation zu betrachten. Diese Werte werden in Abbildung 6.4 grafisch dargestellt. Operationen mit einer kurzen Ausführungszeit, wie etwa `getValueObject`, werden durch die hohen Kommunikationskosten der Webservice-Schnittstelle um vielfaches ihrer eigentlichen Rechenzeit verlangsamt.

Abbildung 6.4 Grafische Darstellung der prozentualen Verlangsamung für den Objekttyp Image



6.4.4 Vergleich von generischer und generierter Implementierung

Zum Abschluss der Leistungsbetrachtungen soll die im Rahmen dieser Arbeit entwickelte Implementierung der Versionierungssysteme mit der bereits vorgestellten generischen Implementierung verglichen werden.

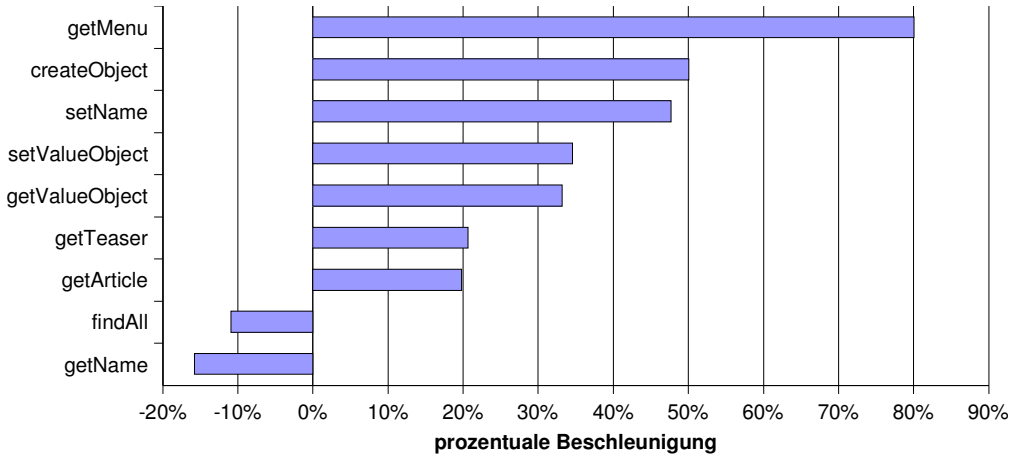
Zu erwarten war, dass die generierte Implementierung aufgrund der vielfältigen Anpassungen, die in Tabelle 6.1 dokumentiert worden sind, deutlich bessere Leistungseigenschaften besitzt. Diese Erwartung wurde durch die Messungen bestätigt. Insgesamt benötigt die generierte Implementierung eine Rechenzeit von 370,86 Sekunden für einen Durchlauf des Leistungstest, wohingegen die generische Implementierung 505,3 Sekunden für die Durchführung der gleichen Operationen benötigt. Dies bedeutet eine prozentuale Beschleunigung der Verarbeitung von immerhin 26,6%.

Auch bei der Betrachtung der Einzelergebnisse kann diese Beschleunigung nachvollzogen werden. Die abgebildete Tabelle 6.5 zeigt die Beschleunigungswerte für den Objekttyp `EditorialDepartment`.

Tabelle 6.5 Ergebnisse für den Objekttyp EditorialDepartment

Operation	Rechenzeit der generischen Implementierung (µs)	Rechenzeit der generierten Implementierung (µs)	Beschleunigung	prozentuale Beschleunigung
getMenu	2423,0	484,0	1939,0	80,0%
createObject	31342,0	15656,0	15686,0	50,0%
setName	3906,5	2044,0	1862,5	47,7%
setValueObject	4581,5	2998,5	1583,0	34,6%
getValueObject	6540,0	4368,5	2171,5	33,2%
getTeaser	2989,0	2372,0	617,0	20,6%
getArticle	2869,5	2300,5	569,0	19,8%
findAll	2115,0	2346,0	-231,0	-10,9%
getName	489,0	566,0	-77,0	-15,7%

Abbildung 6.5 Grafische Darstellung der Beschleunigung für den Objekttyp EditorialDepartment



Die für diesen Objekttyp errechneten prozentuale Beschleunigungswerte werden in Abbildung 6.5 visualisiert. Die Werte lassen sich durch die Optimierungen der generierten Implementierung erklären. Im Fall der Operation `getMenu` benötigt die generierte Implementierung beispielsweise keine Zwischentabelle für die Speicherung der zugehörigen Attachment-Beziehung. Stattdessen wird diese Beziehung durch ein Fremdschlüsselattribut realisiert, wodurch die Navigationslogik auf die Rückgabe eines einfachen Attributwerts reduziert wird.

Ein anderer Grund ist, dass das Attribut `frozen` einer Objektversion durch die generische Implementierung getrennt von den Objektdaten in einer zentralen Versionstabelle verwaltet wird. Dadurch müssen die Operationen `setName`, `setValueObject` und `getValueObject`, die alle dieses Attribut abfragen müssen, nicht nur auf die eigentliche Objekttabelle zugreifen, sondern auch auf die zentrale Versionstabelle, die sehr viele Einträge enthalten kann. Dementsprechend benötigen auch diese Operationen deutlich mehr Rechenzeit im Fall der generischen Implementierung.

Die Messergebnisse für die übrigen Objekttypen können dem Anhang B.4 entnommen werden.

Kapitel 7 Zusammenfassung und Ausblick

In diesem Kapitel werden die in dieser Arbeit gewonnen Erkenntnisse zusammengefasst und ein Ausblick auf weitere mögliche Forschungsthemen gegeben.

7.1 Zusammenfassung

Ausgangspunkt dieser Diplomarbeit war eine Analyse der Domäne der Versionierungssysteme mit Hilfe des Verfahrens der Merkmalanalyse. Die dort gewonnenen Erkenntnisse über die Anwendungsdomäne wurden in eine Spezifikationssprache für Versionierungssysteme umgesetzt. Diese wurde als Erweiterung der UML durch ein UML-Profil realisiert. Diese Technologiewahl erwies sich als sinnvoll. Einerseits sind Klassendiagramme mit den Ergänzungen des UML-Profils gut geeignet für die Definition des Informationsmodells eines Versionierungssystems, das den Hauptanteil der Systemspezifikation ausmacht. Andererseits erlaubten die für die UML verfügbaren Werkzeuge eine einfache Anbindung des entwickelten Generators *RepGen* an ein Modellierungsprogramm.

Dieser Generator unterstützt den Entwickler eines Versionierungssystems in allen Arbeitsschritten, die sich an die Definition der Systemspezifikation mit Hilfe eines Modellierungswerkzeugs anschließen. Als Generierungsverfahren wurde dabei die vorlagenbasierte Generierung gewählt. Dieses Verfahren ermöglichte eine einfache Definition der Generierungslogik, da es die Modellaufbereitung von der Ausgabe der Modelldaten in generierte Dateien trennt. Da alle domänenspezifischen Funktionen von *RepGen* in die Generatorvorlagen integriert wurden, kann *RepGen* durch einen Austausch der Generatorvorlagen für die Umsetzung weiterer Produktlinien wiederverwendet werden.

Die Beurteilung der Generatorvorlagen durch Softwaremetriken ergab, dass diese umstrukturiert werden sollten, um ihre Wartung und Weiterentwicklung zu erleichtern. Die erreichten Wiederverwendungsfaktoren im Falle des Bespielsystems zeigen aber auch, dass der in die Entwicklung der Vorlagen investierte Aufwand sich nach wenigen entwickelten Systemen auszahlen würde.

Die entwickelten Generatorvorlagen bilden die Systemspezifikation eines Versionierungssystems auf eine Middleware-basierte Implementierung ab. Bei der Konzeption der Implementierung wurde darauf geachtet, dass das verwendete Datenbankschema und auch die Anwendungslogik des Systems weitesgehend an die Eigenschaften der individuellen Systemspezifikation anzupassen. Dabei wurde der Vorteil des generativen Ansatzes ausgenutzt, dass

wiederverwendeter Quellcode durch den Generator beliebig dupliziert und verändert werden kann, ohne dass die durch manuelle Duplikation von Quellcode hervorgerufenen Probleme bei der Softwarewartung auftreten. Die abschließend durchgeführten Leistungsuntersuchungen ergaben schließlich auch die Bestätigung, dass die vorgenommen Optimierungen und Anpassungen einem deutlichen Leistungsgewinn gegenüber einer generischen Implementierung bewirken können.

7.2 Ausblick

Während der Erstellung dieser Diplomarbeit sind Fragestellungen entstanden, die sich als weiterführende Forschungsthemen anbieten. Eine Auswahl dieser Themen wird nun vorgestellt.

7.2.1 Übergang von Domänenanalyse zu Domänenimplementierung

Das in dieser Arbeit verwendete UML-Profil für Versionierungssysteme bildet zusammen mit dem Generator die Implementierung der Produktlinie für Versionierungssysteme. Das mit dem UML-Profil erweiterte UML-Metamodell ermöglicht die Modellierung von in der Produktlinie enthaltenen Systemen. Modelliert werden sowohl strukturelle Aspekte der Systeme, als auch die gewünschte Konfiguration der Produktlinienkonzepte. Grundlage für die Definition des UML-Profils waren die Ergebnisse der Domänenanalyse. Dort wurden die Konzepte der Anwendungdomäne und ihre Merkmale identifiziert und die Abhängigkeiten der Merkmale analysiert. Aufgrund der Einschränkungen der verwendeten Analysemethode konnte aber während der Domänenanalyse keine formale Darstellung der strukturellen Aspekte der Anwendungsdomäne gewonnen werden. Da diese aber in der Domäne der Versionierungssysteme, wie auch in anderen auf Datenbanken aufbauenden Domänen, einen entscheidenden Teil der Systemspezifikation bilden, konnte das Profil nicht automatisch aus den Ergebnissen der Domänenanalyse abgeleitet werden, sondern wurde in einem kreativen Prozess erstellt.

Es wäre interessant zu untersuchen, ob durch die Verwendung eines anderen Analyseverfahren, das auch strukturelle Aspekte der Domäne berücksichtigt, eine automatische Abbildung auf ein UML-Profil möglich wird. Dabei wäre einerseits zu klären, welche Anforderungen ein geeignetes Analyseverfahren zu erfüllen hätte, andererseits müsste ein Format zur Definition der notwendigen Abbildungsregeln gefunden werden. Ein denkbarer Ansatz wäre, bereits während der Domänenanalyse UML zur Modellierung der strukturellen Eigenschaften der Domäne zu verwenden und die gewonnene Darstellung anschließend bei der Definition des UML-Profils um die Konfiguration der Domänenkonzepte zu ergänzen.

7.2.2 Werkzeugunterstützung für die Vorlagenerstellung

Während der Entwicklung der Generatorvorlagen wurde deutlich, dass die Werkzeugunterstützung dieser Tätigkeit unzureichend ist und verbessert werden sollte. Zur Zeit muss zur Erstellung einer Generatorvorlage ein herkömmlicher Texteditor verwendet werden, der nur für die Erstellung der statischen Anteile einer Vorlage, also für die in das Metaprogramm eingebetteten Java- oder XML-Codefragmente, spezielle Unterstützung bietet. Für diese Fragmente analysiert der Editor den Programmsyntax und unterstützt den Entwickler

unter anderem durch eine farbige Markierung des Programmsyntax und durch Funktionen zur automatischen Eingabenvervollständigung.

Durch die Analyse der Vorlagen durch Softwaremetriken und durch die gegebenen Beispiele wurde deutlich, dass die enthaltenen Metaprogramme einen beträchtlichen Umfang erreichen können. Deshalb sollte eine Entwicklungsumgebung für Generatorvorlagen zusätzlich auch den Syntax der Metaprogramme interpretieren können. Dadurch wäre es möglich, die Kontrollstrukturen des Metaprogramms grafisch so hervorzuheben, dass sie gut von den Kontrollstrukturen des generierten Programms zu unterscheiden sind. Sehr hilfreich wäre auch eine Eingabenvervollständigung für die Schlüsselwörter der VTL und für die über den Velocity-Kontext zugänglichen Java-Objekte.

Ein weiter Problempunkt ist das Auffinden von Fehlern in den Generatorvorlagen. Hierbei könnte ein vollwertiger Debugger helfen, der eine schrittweise interaktive Ausführung der Vorlagen erlaubt und einen Einblick in den Inhalte des Velocity-Kontext bietet. Derzeit kann eine Vorlage zwar mit Debug-Ausgaben instrumentiert werden, letztlich muss ein Fehler aber durch die Erstellung geeigneter Testdaten und eine Interpretation der Ausgaben aufwendig manuell nachvollzogen werden. Denkbar wäre auch, die Ausgaben des Generators insofern rückverfolgbar zu machen, sodass die Entwicklungsumgebung zu einem gegebenen Abschnitt der generierten Dateien darstellen kann, welche UML-Elemente der Spezifikation in die Generierung dieses Abschnitts eingegangen sind.

Auch die Durchführung von Qualitätssicherungsmassnahmen ist derzeit sehr aufwendig. Die im Rahmen dieser Arbeit entwickelten Vorlagen erstellen für die durch sie generierten Versionierungssysteme ein an deren Spezifikation angepasstes Testprogramm, dass einen automatisierten Selbsttest des installierten Systems ermöglicht. Für eine umfassende Qualitätsicherung müsste allerdings auch überprüft werden, ob die Vorlagen für jede zulässige Spezifikation ein funktionierendes Softwaresystem ausgeben. Diese Tests sind nur stichprobenartig durchgeführt worden, da sie ohne Werkzeugunterstützung recht aufwendig sind. Hilfreich wäre ein Testfall-Generator, der eine große Anzahl unterschiedlicher Spezifikationen erzeugt und dazu passend ein Mechanismus, der diese Testfälle automatisiert anwendet. Dieser Mechanismus müsste den Generator starten, das erzeugte Softwaresystem übersetzen, es installieren und anschließend den Selbsttest des Systems ausführen. Wenn keine Fehler bei der Übersetzung, der Installation und dem Selbsttest des erzeugten Versionierungssystems auftauchen, kann angenommen werden, dass eine Testspezifikation durch die Vorlagen korrekt verarbeitet wurde.

7.2.3 Föderierte Versionierungssysteme

Die in dieser Arbeit konzipierte Implementierung der generierten Versionierungssysteme beinhaltet eine Webservice-Schnittstelle. Über diese Schnittstelle können sich Anwendungsprogramme mit einem Versionierungssystem verbinden, die die dort verwalteten Daten verarbeiten. Diese Schnittstelle kann aber auch genutzt werden, um mehrere Versionierungssysteme miteinander zu vernetzen.

Ein *föderiertes Versionierungssystem* umfasst mehrere Einzelsysteme, die durch ein übergeordnetes Versionierungssystem zu einem virtuellen Gesamtsystem integriert werden. Das Informationsmodell des übergeordneten Systems bietet eine *integrierte Sicht* auf die Daten der teilnehmenden Systeme. In ihm sind einerseits die Objekt- und Beziehungstypen aller teilnehmenden Systeme enthalten, andererseits enthält es zusätzliche Beziehungstypen, die

die Objekttypen der Teilnehmersysteme verbinden. Die zusätzlich eingefügten Beziehungen werden durch das übergeordnete Versionierungssystem selbst verwaltet, alle Operationen hingegen, die sich auf Typen der Teilnehmersysteme beziehen, werden an diese weitergeleitet.

Das Hauptanwendungsgebiet eines solchen Systems liegt darin, bestehende Versionierungssysteme in eine neue Systemumgebung zu integrieren. Oftmals ist es aufgrund von Anwendungsprogrammen, die für die bestehenden Systeme entwickelt wurden, nicht möglich bzw. ökonomisch nicht sinnvoll, alte Versionierungssysteme vollständig durch erweiterte Nachfolgesysteme abzulösen. Stattdessen können ihre Daten in ein föderiertes Versionierungssystem aufgenommen und dort mit den nötigen Erweiterungen versehen werden. Auf diese Weise können bestehende Anwendungsprogramme weiterhin verwendet werden, da sich das Informationsmodell des Altsystems nicht verändert.

Wie auch schon die einfachen Versionierungssysteme könnten auch die übergeordnete Systeme im Rahmen einer Produktlinie durch *RepGen* generiert werden. Dazu müsste die Spezifikationssprache um zusätzliche Konstrukte erweitert werden, um die Definition des integrierten Informationsmodells zu ermöglichen.

Sehr interessant wäre es auch, *Wrapper* für Versionierungssysteme zu schaffen, die nicht in unserer Produktlinie enthalten sind. Dazu zählen beispielsweise dateibasierte Versionierungssysteme wie etwas CVS [CVS03], oder auch objektorientierte Versionierungssysteme kommerzieller Hersteller. Diese Wrapper statten Fremdsysteme mit einer Schnittstelle aus, die ein äquivalentes Produktliniensystem besitzen würde. Problematisch ist in diesem Zusammenhang vor allem die Definition von äquivalenten Informationsmodellen für nicht objektorientierte Fremdsysteme und die Emulation von Versionierungssemantiken, die von Fremdsystemen nicht angeboten werden.

Die Wrapper würden es ermöglichen, die in Fremdsystemen gespeicherten Daten in das integrierte Informationsmodell eines föderierten Versionierungssystems einzubeziehen. So könnte beispielsweise ein Versionierungssystem, das Testdaten und -ergebnisse verwaltet, mit einem CVS-Server kombiniert werden, der den zu testenden Quellcode verwaltet. Im integrierten Informationsmodell könnte Beziehungen zwischen Testfällen und zugehörigen Quellcodedateien eingerichtet werden.

Anhang A

Ergebnisse der Vorlagenbewertung

A.1 Grafische Darstellung der Messergebnisse

Abbildung A.1 Dateigröße der Vorlagen

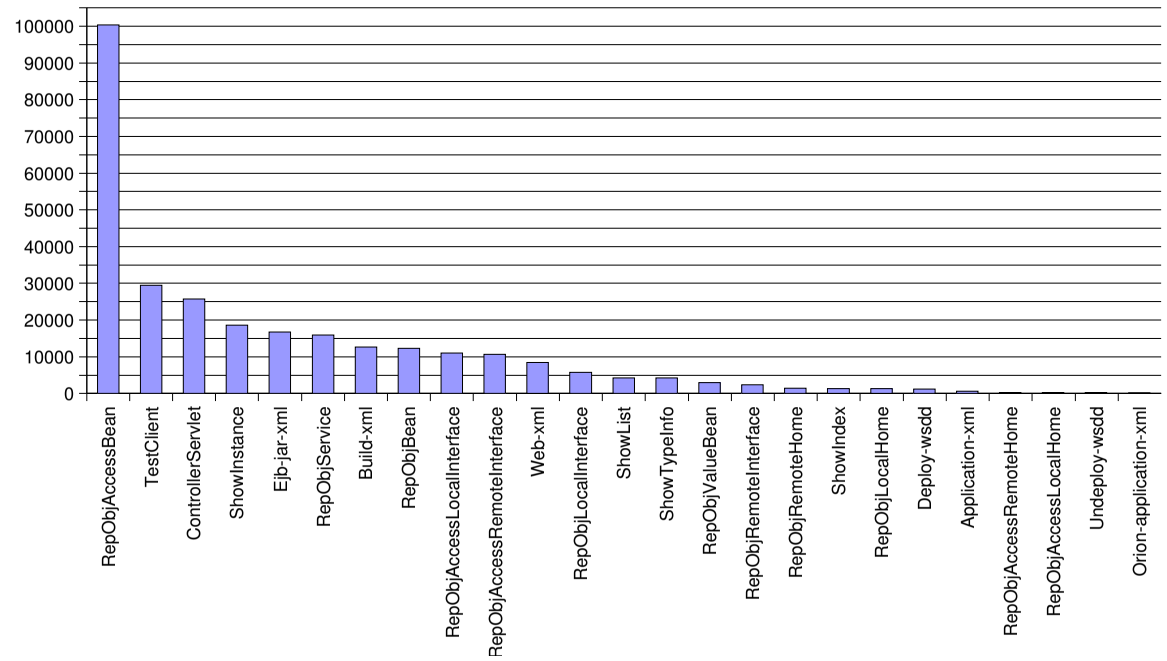


Abbildung A.2 Zeilenanzahl der Vorlagen

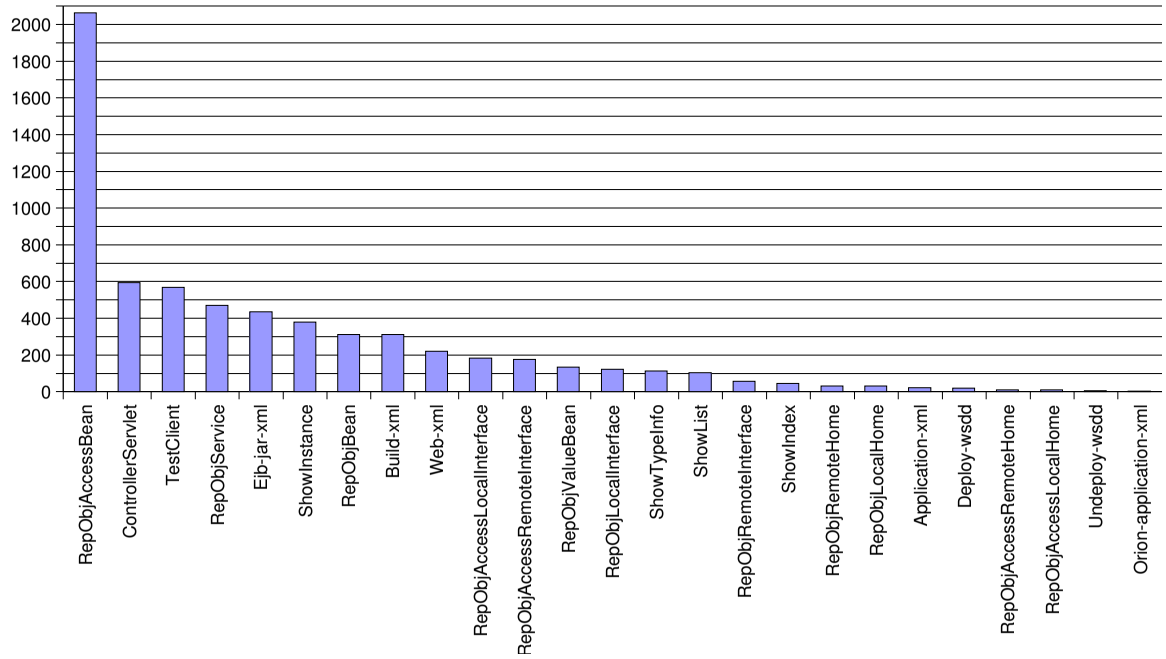


Abbildung A.3 Anzahl der in den Vorlagen enthalten Referenzen

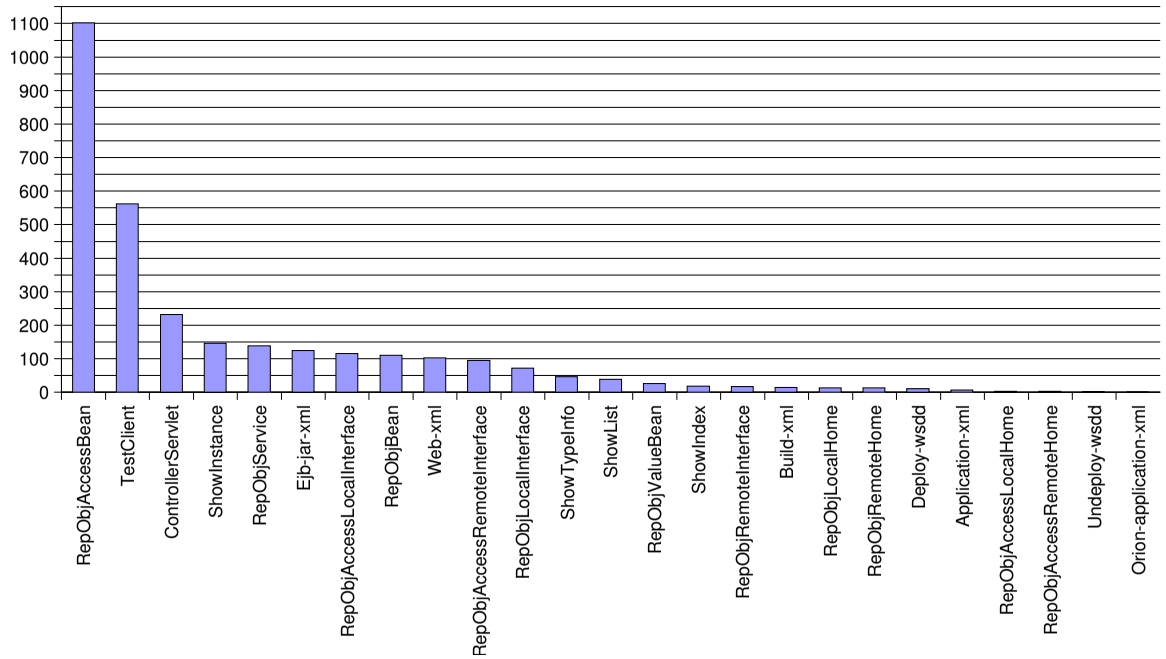


Abbildung A.4 Anzahl der in den Vorlagen enthalten Fallunterscheidungen

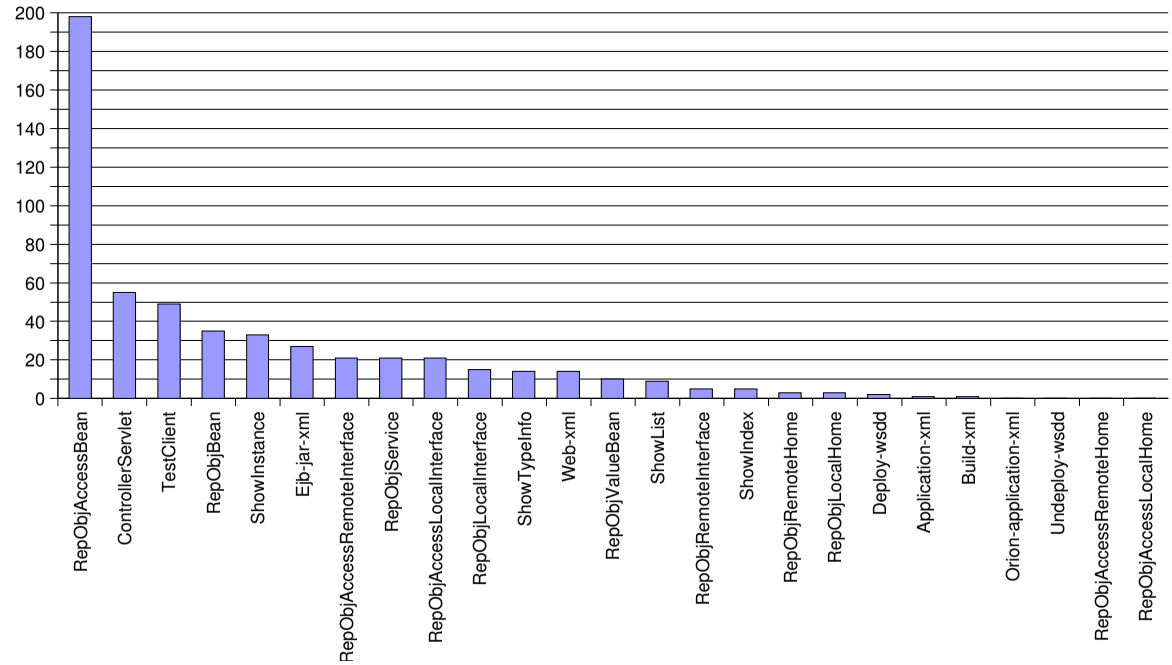


Abbildung A.5 Anzahl der in den Vorlagen enthalten Schleifen

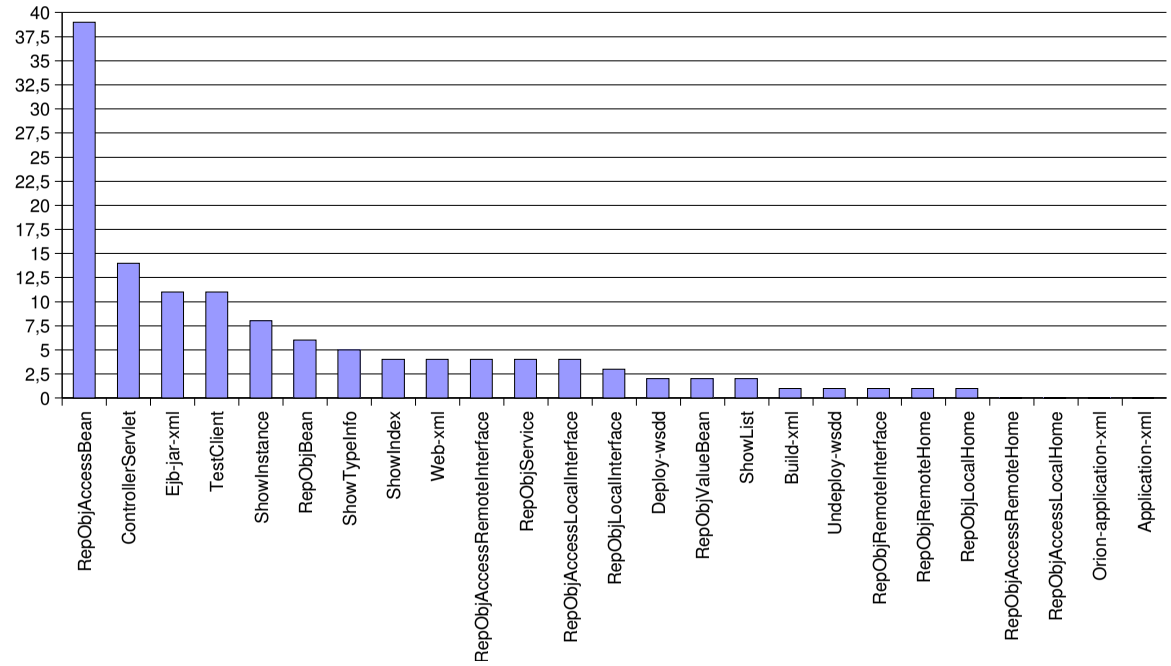


Abbildung A.6 Anzahl der in den Vorlagen enthalten Anweisungen

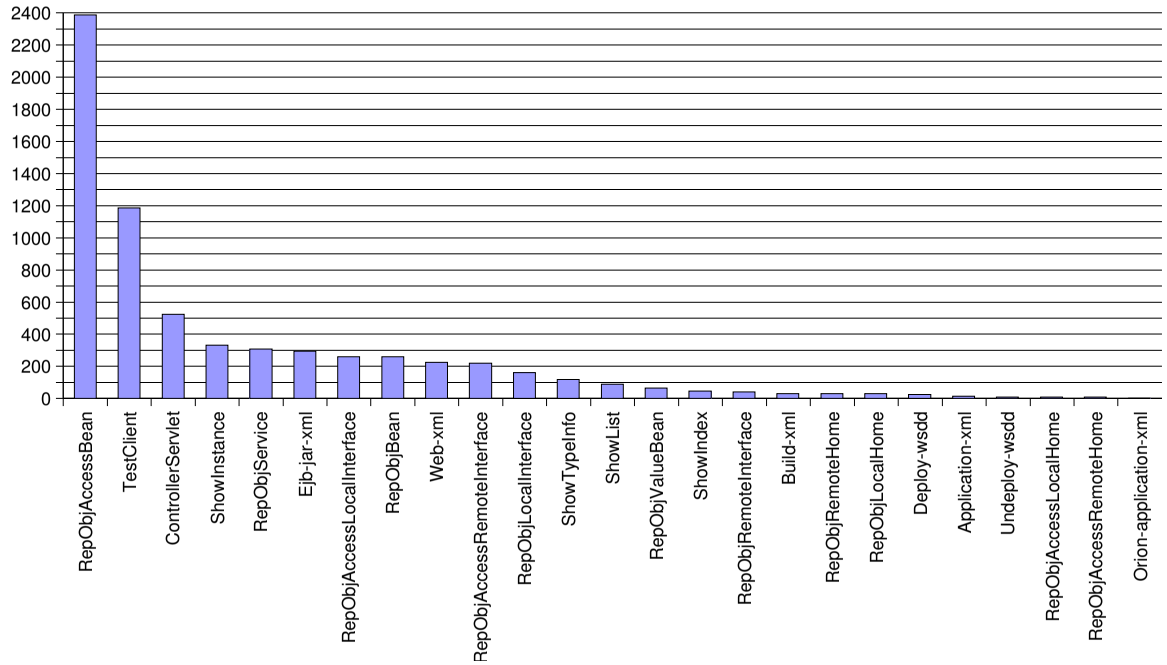


Abbildung A.7 Zyklomatische Komplexität der Vorlagen

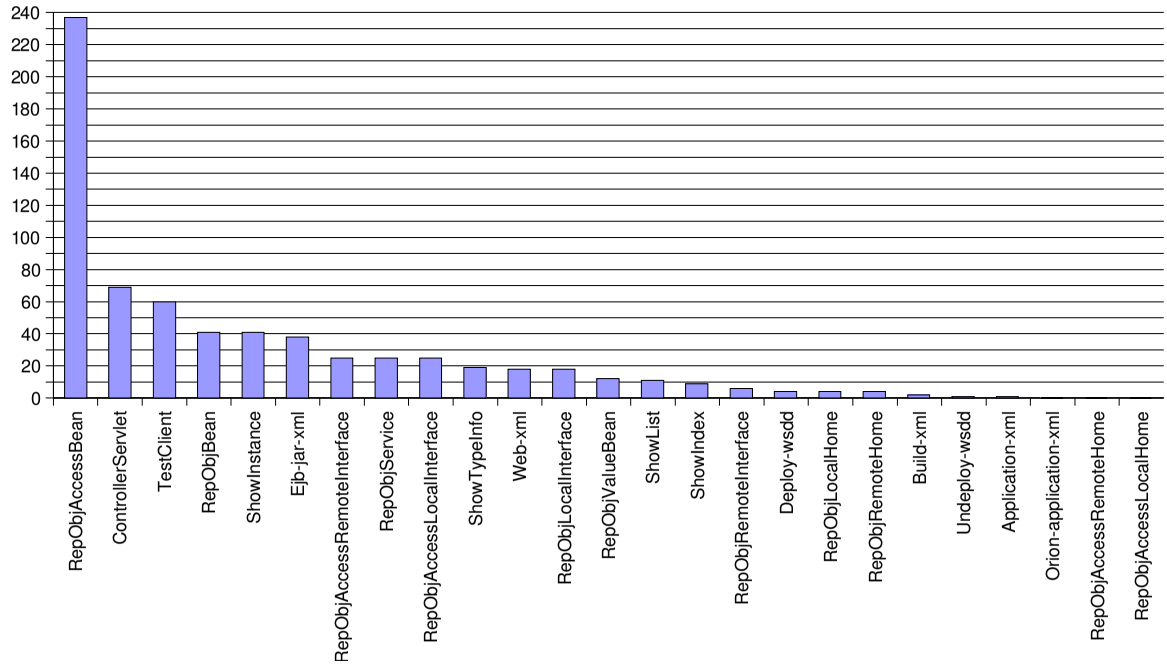
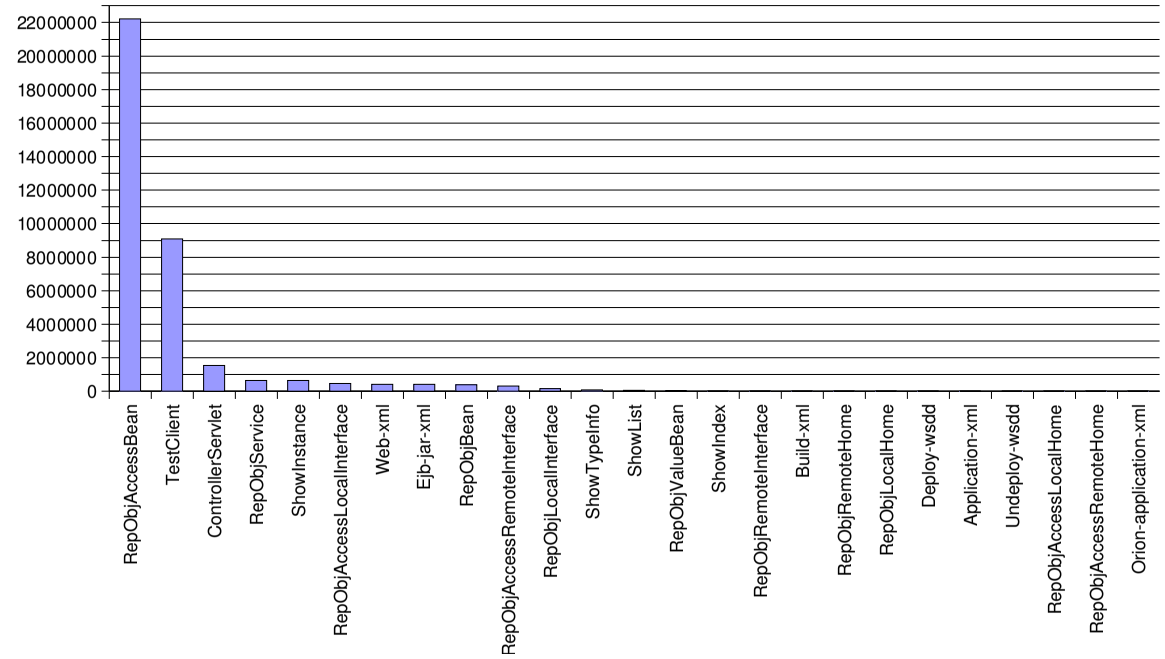


Abbildung A.8 Hallstead-Aufwand der Vorlagen



A.2 Korrelationen zum Umfang der Vorlagen

Abbildung A.9 Referenzen und Fallunterscheidungen

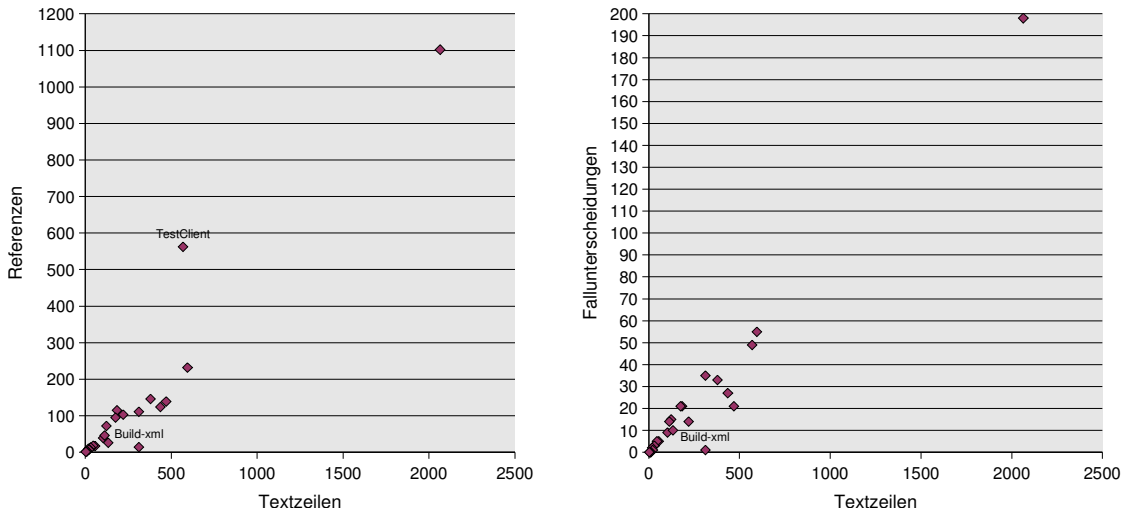


Abbildung A.10 Schleifen- und Anweisungsanzahl

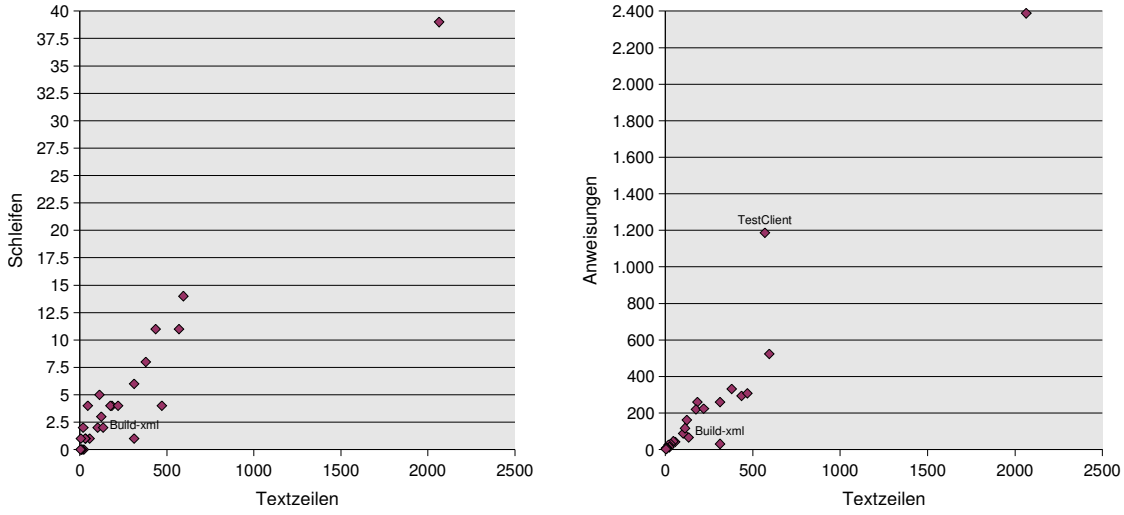
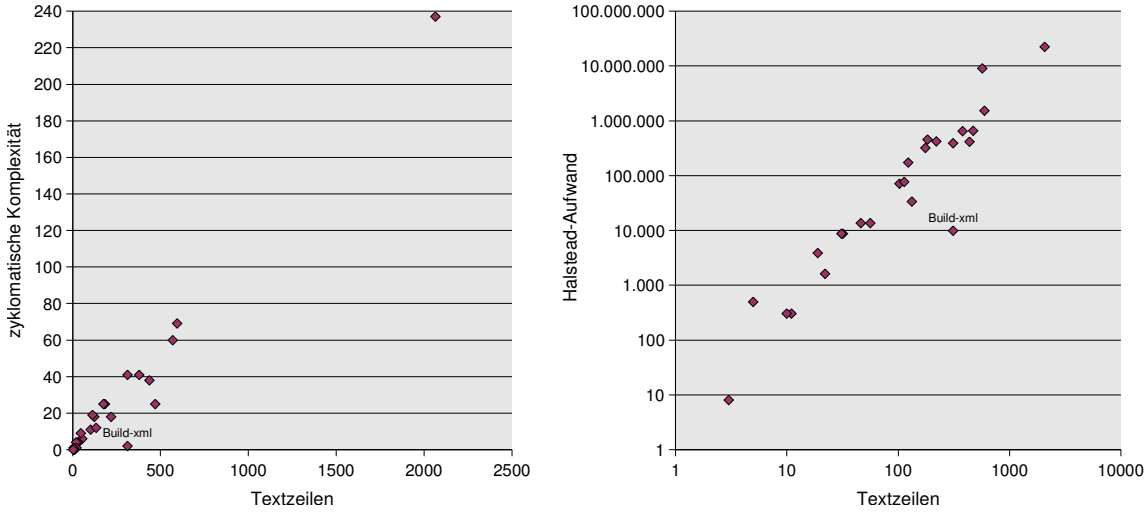


Abbildung A.11 Komplexität und Halstead-Aufwand



Anhang B Ergebnisse der Leistungsuntersuchungen

B.1 Test der generierten Implementierung

Tabelle B.1 Ergebnisse für den Objekttyp EditorialDepartment

Operation	Operations-aufrufe	Gesamte Rechenzeit (µs)	Minimale Rechenzeit (µs)	Maximale Rechenzeit (µs)	Durchschnittliche Rechenzeit (µs)	Durchschnittliche Abweichung (µs)	Median der Rechenzeit (µs)	Durchschnittliche Abweichung (µs)
createObject	4	498268	14539	452417	124567,0	163925,0	15656,0	109939,0
getValueObject	24	112466	984	10090	4686,1	1295,9	4368,5	1241,8
setValueObject	24	84359	2782	9626	3515,0	960,6	2998,5	658,0
getTeaser	48	140939	949	29499	2936,2	1315,5	2372,0	930,3
findAll	16	143911	2266	58539	8994,4	11372,3	2346,0	6689,3
getArticle	60	121777	801	5131	2029,6	650,7	2300,5	564,0
setName	28	73697	1881	8123	2632,0	996,6	2044,0	698,5
getName	24	14744	303	1859	614,3	155,4	566,0	142,3
getMenu	105	57253	409	1925	545,3	108,6	484,0	87,2
	333	*1247414*						

Abbildung B.1 Grafische Darstellung der Rechenzeiten für den Objekttyp EditorialDepartment

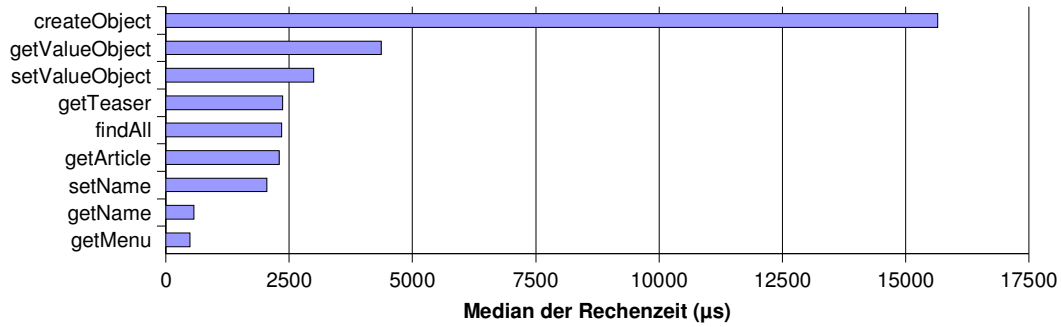


Tabelle B.2 Ergebnisse für den Objekttyp Article

Operation	Operations-aufrufe	Gesamte Rechenzeit (µs)	Minimale Rechenzeit (µs)	Maximale Rechenzeit (µs)	Durchschnittliche Rechenzeit (µs)	Durchschnittliche Abweichung (µs)	Median der Rechenzeit (µs)	Durchschnittliche Abweichung (µs)
deleteObject	36	3674157	53360	219779	102059,9	29645,4	109109,0	28746,5
createObject	64	5148170	52964	1182896	80440,2	40072,6	56468,0	25858,1
createSuccessor	840	46482822	41890	825755	55336,7	9677,7	50260,0	8338,4
findAll	16	825749	8377	109572	51609,3	12134,9	49938,5	11883,7
attachToEditorialDepartment	168	3379288	15423	54002	20114,8	2311,6	19671,5	2281,1
freeze	504	3837941	5818	36803	7615,0	1065,0	7200,0	915,3
merge	168	1080878	5563	36733	6433,8	835,9	6007,0	621,2
checkoutToEditorialDepartment	684	3696088	4240	57412	5403,6	1421,3	4626,0	961,3
checkinFromEditorialDepartment	684	3287560	4125	34145	4806,4	609,7	4508,0	476,8
setValueObject	1716	6078087	3202	48223	3542,0	404,1	3320,0	250,1
addImage	1148	3955977	2138	30091	3446,0	614,0	3246,0	518,9
setDate	64	221191	2469	22319	3456,1	1006,9	2797,5	908,0
setTitle	1844	4361869	2123	28586	2365,4	240,8	2237,0	160,4
setText	1780	3401956	1741	25902	1911,2	197,2	1800,0	131,1
getValueObject	3108	4311109	903	44512	1387,1	370,8	1262,0	298,9
getSuccessors	3264	6567178	657	28453	2012,0	1516,3	936,0	1248,6
getTeaser	268	287921	800	6648	1074,3	295,5	896,0	196,2
getImage	3276	4472907	719	59887	1365,4	685,5	861,0	576,0
getAncestors	3264	4584953	708	26376	1404,7	734,6	803,0	638,7
getAlternatives	3264	2659427	635	25154	814,8	139,1	741,0	95,9
getRoot	3432	1974651	344	72042	575,4	96,5	532,0	81,7
getTitle	4824	2115643	146	24719	438,6	112,7	456,0	110,0
isFrozen	3264	1034852	52	23766	317,1	122,3	366,0	100,2
setCurrentEditorialDepartment	1756	551678	161	2525	314,2	163,9	181,0	143,7
unsetCurrentEditorialDepartment	1802	202602	10	1390	112,4	13,1	105,0	10,8
	41238	*118194654*						

Abbildung B.2 Grafische Darstellung der Rechenzeiten für den Objekttyp Article

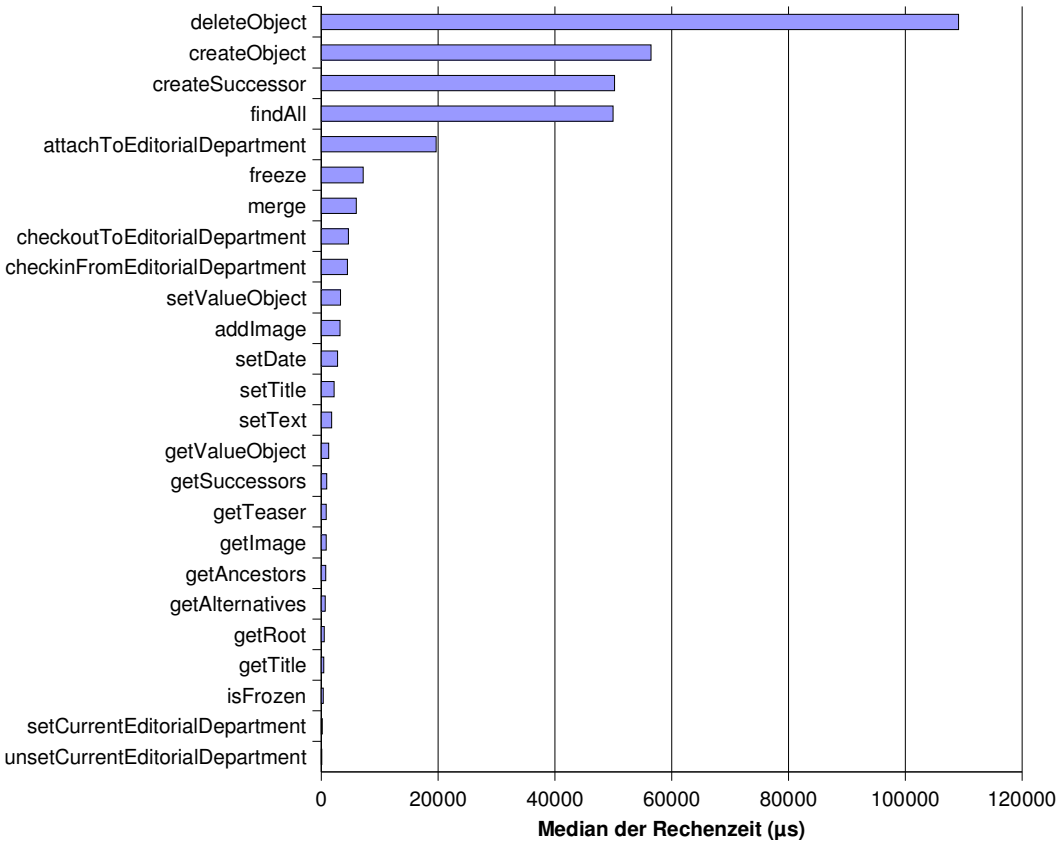


Tabelle B.3 Ergebnisse für den Objekttyp Image

Operation	Operations-aufrufe	Gesamte Rechenzeit (µs)	Minimale Rechenzeit (µs)	Maximale Rechenzeit (µs)	Durchschnittliche Rechenzeit (µs)	Durchschnittliche Abweichung (µs)	Median der Rechenzeit (µs)	Durchschnittliche Abweichung (µs)
findAll	16	171764	7001	15514	10735,3	1569,4	10578,0	1542,5
createObject	128	1445137	7383	211133	11290,1	5300,6	7944,5	3754,2
copyObject	684	5646854	7503	39009	8255,6	997,9	7693,5	611,8
deleteObject	684	2322465	2983	39035	3395,4	373,9	3120,5	324,0
setValueObject	684	2164340	2908	26979	3164,2	344,2	2965,0	214,1
setWidth	128	424177	2699	15858	3313,9	636,4	2789,5	572,3
setHeight	128	500606	2427	45824	3911,0	1980,2	2698,0	1409,4
setCopyright	128	437002	2452	29357	3414,1	1125,4	2608,0	915,5
setDescription	128	379140	2096	31550	2962,0	992,3	2213,5	812,2
getValueObject	2292	2227647	583	29693	971,9	420,4	752,5	315,5
	5000	*15719132*						

Abbildung B.3 Grafische Darstellung der Rechenzeiten für den Objekttyp Image

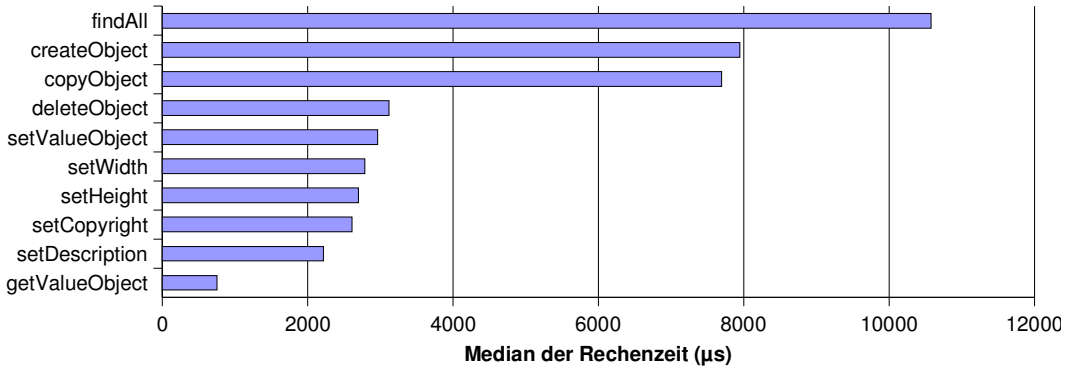


Tabelle B.4 Ergebnisse für den Objekttyp Menu

Operation	Operations-aufrufe	Gesamte Rechenzeit (µs)	Minimale Rechenzeit (µs)	Maximale Rechenzeit (µs)	Durchschnittliche Rechenzeit (µs)	Durchschnittliche Abweichung (µs)	Median der Rechenzeit (µs)	Durchschnittliche Abweichung (µs)
createObject	4	845448	27196	668249	211362,0	228443,5	75001,5	176199,0
deleteVersion	24	559215	20591	49299	23300,6	3605,0	21328,5	2366,5
createSuccessor	120	2287543	17095	68179	19062,9	2155,2	17616,5	1606,3
addContent	672	4973307	4681	45778	7400,8	2189,1	8367,0	2111,7
merge	24	138417	5398	9103	5767,4	486,3	5475,0	327,8
addSubCategory	21	148529	3641	35953	7072,8	3341,1	4474,0	2995,9
findAll	16	73794	3369	10223	4612,1	969,4	4293,5	861,8
pinContent	336	1304081	3393	25837	3881,2	585,5	3533,0	406,5
attachToEditorialDepartment	24	86678	3250	7275	3611,6	402,9	3384,5	287,1
freeze	72	204801	2565	4027	2844,5	153,4	2835,5	153,2
setValueObject	480	1456467	2743	27544	3034,3	413,4	2807,0	248,6
setName	484	1107745	2136	7152	2288,7	104,1	2244,0	89,2
checkoutToEditorialDepartment	252	651285	2159	24598	2584,5	630,0	2216,5	389,9
checkinFromEditorialDepartment	252	580814	2115	8103	2304,8	225,9	2174,5	150,5
setDescription	484	977390	1737	23550	2019,4	431,1	1787,0	249,3
getContent	192	237522	707	16052	1237,1	514,7	978,5	386,4
getValueObject	648	619800	619	18961	956,5	220,0	971,0	217,6
getSuccessors	1164	1566247	668	27809	1345,6	830,7	845,0	606,4
getSubCategory	192	226072	714	7374	1177,5	585,0	806,5	407,5
getAncestors	1164	1371128	736	25937	1177,9	573,6	800,0	401,2
getAlternatives	1164	1018744	660	25895	875,2	185,7	777,0	112,8
getSubCategory_Value	168	229413	681	24276	1365,6	841,0	739,0	647,6
getRoot	1188	832195	457	112726	700,5	280,7	554,0	174,5
setCurrentEditorialDepartment	40	21569	225	1871	539,2	165,8	480,5	148,8
getName	1356	521014	144	2934	384,2	150,4	461,0	128,5
isFrozen	1164	313142	60	1773	269,0	151,3	367,0	130,9
unsetCurrentEditorialDepartment	86	11844	6	5006	137,7	155,1	23,0	125,1
	11791	*22364204*						

Abbildung B.4 Grafische Darstellung der Rechenzeiten für den Objekttyp Menu

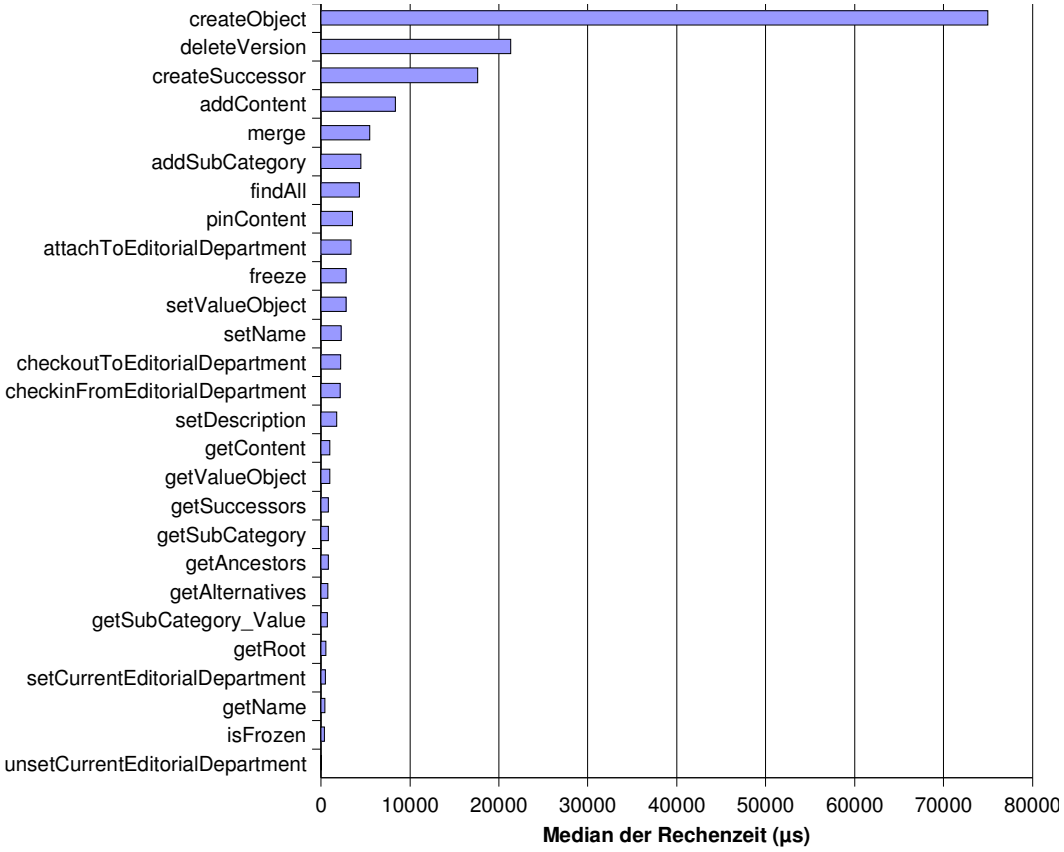
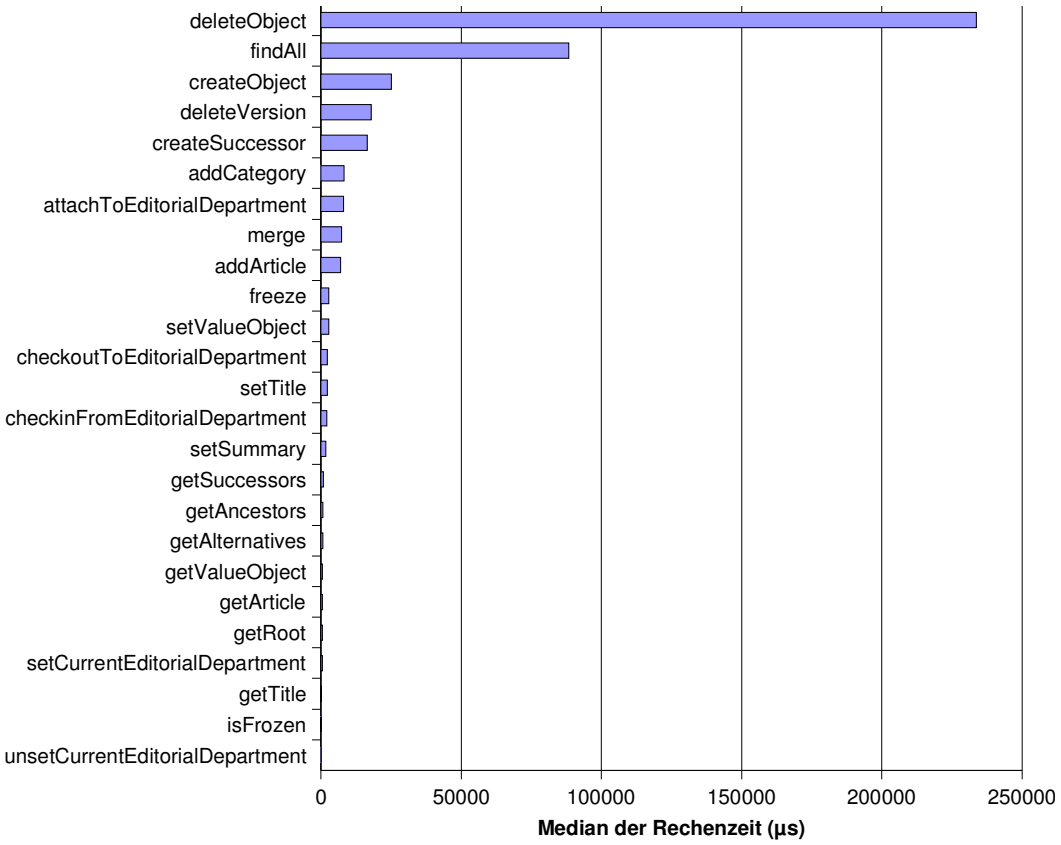


Tabelle B.5 Ergebnisse für den Objekttyp Teaser

Operation	Operations-aufrufe	Gesamte Rechenzeit (µs)	Minimale Rechenzeit (µs)	Maximale Rechenzeit (µs)	Durchschnittliche Rechenzeit (µs)	Durchschnittliche Abweichung (µs)	Median der Rechenzeit (µs)	Durchschnittliche Abweichung (µs)
deleteObject	72	16314411	114393	416010	226589,0	66398,7	233565,5	66000,3
findAll	16	1696021	7738	236906	106001,3	43281,0	88353,5	41121,9
createObject	64	1956542	24314	93392	30571,0	7851,8	25186,5	5730,5
deleteVersion	336	6437381	15841	70394	19158,9	2549,2	17951,5	2061,5
createSuccessor	1680	31143144	15620	141314	18537,6	3206,8	16491,0	2351,8
addCategory	128	1295466	7700	58982	10120,8	3202,0	8166,0	2299,4
attachToEditorialDepartment	336	2753273	6392	41612	8194,3	902,8	8069,0	893,8
merge	336	2567292	5665	41789	7640,8	1101,4	7318,5	1052,5
addArticle	400	3060751	6114	49105	7651,9	1196,0	7016,0	812,8
freeze	1008	3034928	2525	35131	3010,8	456,9	2793,5	337,3
setValueObject	6528	19142727	2675	46806	2932,4	292,6	2764,0	194,8
checkoutToEditorialDepartment	1368	3265805	2129	21284	2387,3	309,1	2205,0	207,7
setTitle	6656	15013692	2035	27084	2255,7	143,9	2179,0	110,5
checkinFromEditorialDepartment	1368	3158549	2038	77163	2308,9	306,2	2131,0	203,3
setSummary	6656	12160274	1656	46019	1827,0	178,8	1726,0	126,4
getSuccessors	15816	35796923	310	29498	2263,3	2062,8	859,5	1693,2
getAncestors	15816	20665936	607	29174	1306,6	734,0	705,0	640,9
getAlternatives	15816	11443871	233	43922	723,6	142,6	648,0	98,1
getValueObject	7920	5695325	475	43511	719,1	209,7	583,0	152,0
getArticle	1728	1095499	508	24620	634,0	137,8	555,0	98,9
getRoot	16152	9162221	1	628217	567,2	142,0	504,0	119,6
setCurrentEditorialDepartment	40	21815	260	1749	545,4	151,3	476,0	132,6
getTitle	17544	4093584	49	22542	233,3	158,7	111,0	144,4
isFrozen	15816	2306710	7	22246	145,8	141,7	43,0	128,1
unsetCurrentEditorialDepartment	86	7365	12	1360	85,6	74,7	27,0	66,7
	133686	*213289505*						

Abbildung B.5 Grafische Darstellung der Rechenzeiten für den Objekttyp Teaser



B.2 Wiederholte Durchführung des Leistungstests

Tabelle B.6 Ergebnisse für den Objekttyp EditorialDepartment

Operation	Rechenzeit im ersten Durchlauf (µs)	Rechenzeit im zweiten Durchlauf (µs)	Rechenzeit im dritten Durchlauf (µs)
createObject	43376,5	13254,0	13199,5
getValueObject	4630,5	4170,0	4091,0
setValueObject	3077,5	2852,5	2861,0
getTeaser	2388,0	2360,0	2317,5
findAll	2340,5	2326,5	2399,0
getArticle	2304,5	2294,0	2286,5
setName	2086,5	1936,0	1920,0
getName	572,0	516,5	523,5
getMenu	494,0	469,0	470,0

Abbildung B.6 Grafische Darstellung der Ergebnisse für den Objekttyp EditorialDepartment

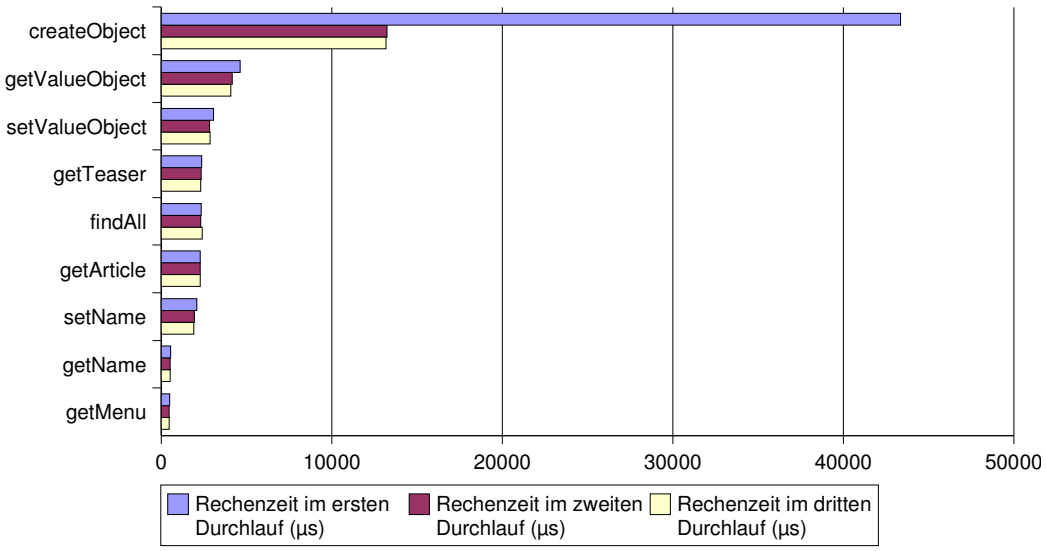


Tabelle B.7 Ergebnisse für den Objekttyp Article

Operation	Rechenzeit im ersten Durchlauf (µs)	Rechenzeit im zweiten Durchlauf (µs)	Rechenzeit im dritten Durchlauf (µs)
deleteObject	110001,0	143125,5	182755,5
createObject	56530,0	61968,0	70594,5
findAll	50742,0	62914,0	81352,5
createSuccessor	50004,5	57180,0	63517,0
attachToEditorialDepartment	19748,0	23724,0	27451,5
freeze	7197,5	7946,5	8753,5
merge	6016,5	6567,0	6952,0
checkoutToEditorialDepartment	4624,0	5378,0	6190,5
checkinFromEditorialDepartment	4479,5	5249,0	6038,0
setValueObject	3307,0	3292,0	3304,0
addImage	3277,0	3259,0	3267,0
setDate	2619,5	2554,5	2557,0
setTitle	2253,0	2246,0	2243,0
setText	1801,0	1794,0	1796,0
getValueObject	1242,0	1220,0	1232,0
getSuccessors	925,0	923,0	921,0
getTeaser	894,0	880,0	876,0
getImage	856,0	812,0	809,0
getAncestors	796,0	789,0	787,0
getAlternatives	732,0	723,0	723,0
getRoot	534,0	529,0	528,0
getTitle	454,0	424,0	424,0
isFrozen	362,0	366,0	367,0
setCurrentEditorialDepartment	182,0	181,0	183,0
unsetCurrentEditorialDepartment	101,0	101,0	101,0

Abbildung B.7 Grafische Darstellung der Ergebnisse für den Objekttyp Article

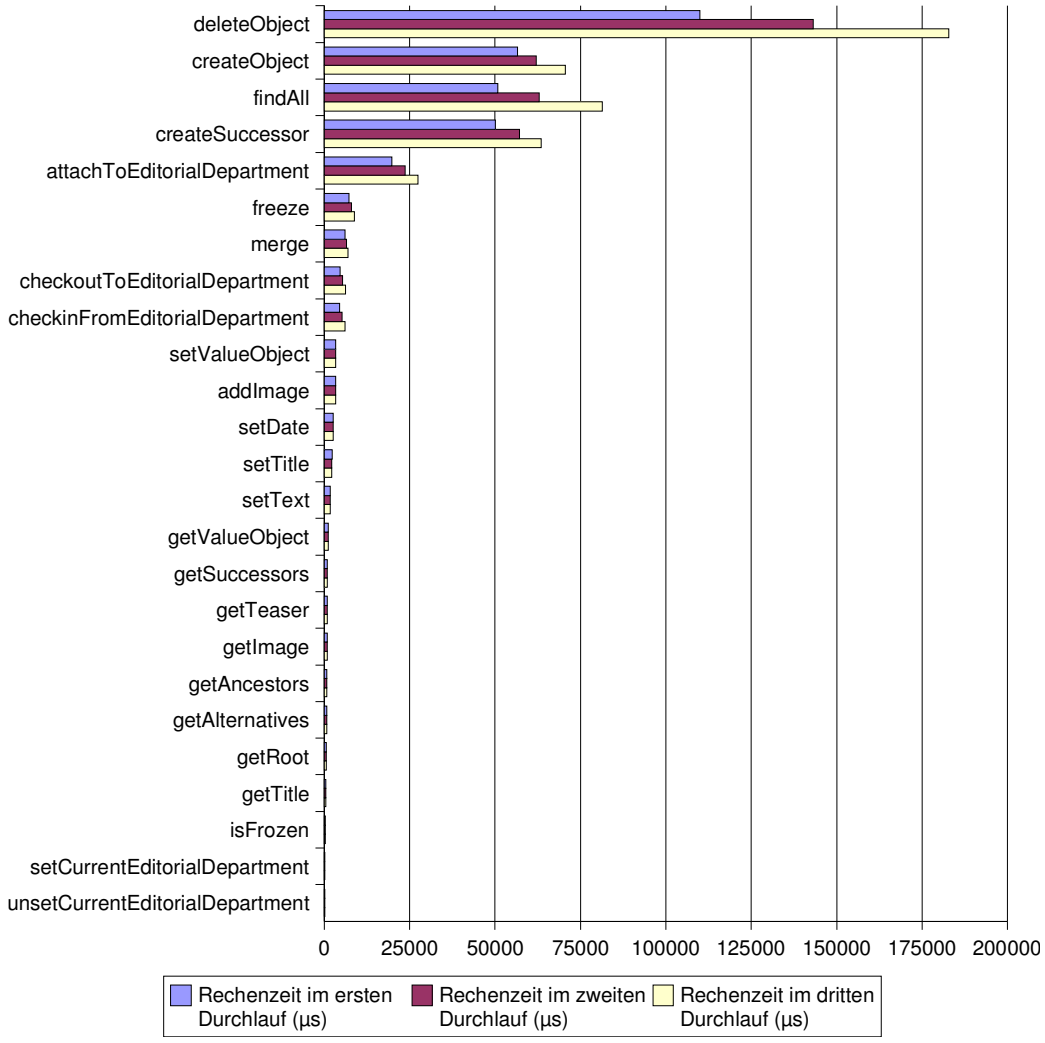


Tabelle B.8 Ergebnisse für den Objekttyp Image

Operation	Rechenzeit im ersten Durchlauf (µs)	Rechenzeit im zweiten Durchlauf (µs)	Rechenzeit im dritten Durchlauf (µs)
findAll	10548,0	18214,0	66548,5
createObject	8051,5	7492,5	7478,5
copyObject	7678,0	7672,0	7677,0
deleteObject	3143,0	3108,5	3099,0
setValueObject	2940,0	2935,0	2932,0
setWidth	2831,0	2759,0	2757,5
setHeight	2808,0	2514,5	2513,0
setCopyright	2623,0	2513,5	2513,0
setDescription	2336,0	2160,5	2168,5
getValueObject	762,0	670,0	671,0

Abbildung B.8 Grafische Darstellung der Ergebnisse für den Objekttyp Image

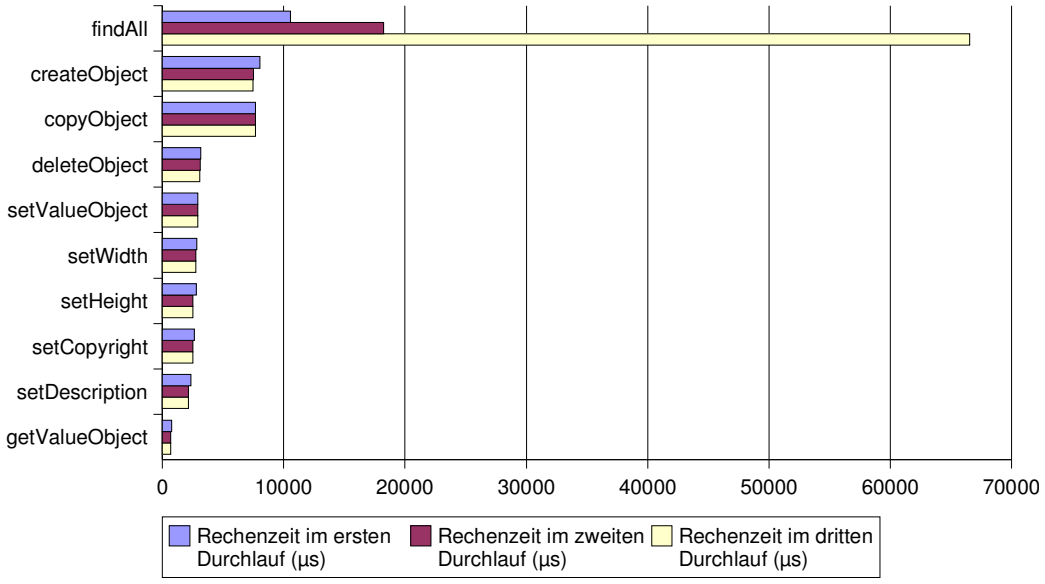


Tabelle B.9 Ergebnisse für den Objekttyp Menu

Operation	Rechenzeit im ersten Durchlauf (µs)	Rechenzeit im zweiten Durchlauf (µs)	Rechenzeit im dritten Durchlauf (µs)
createObject	43487,0	22882,5	23681,5
deleteVersion	21204,0	21903,5	22826,5
createSuccessor	17569,0	17983,0	18319,5
addContent	8342,0	8715,0	9110,5
merge	5515,0	5565,0	5821,5
addSubCategory	4468,0	4394,0	5346,0
findAll	4306,5	46604,0	50542,0
pinContent	3548,5	3503,0	3504,5
attachToEditorialDepartment	3429,0	3356,5	3353,5
freeze	2851,0	2793,0	2784,0
setValueObject	2808,0	2811,5	2810,0
setName	2258,5	2248,0	2252,0
checkoutToEditorialDepartment	2235,5	2233,0	2242,5
checkinFromEditorialDepartment	2192,0	2192,5	2195,0
setDescription	1788,5	1787,0	1788,0
getContent	984,0	869,0	867,5
getValueObject	960,0	962,0	959,0
getSuccessors	843,0	840,0	839,0
getSubCategory	794,5	771,0	771,5
getAncestors	791,0	786,0	782,0
getAlternatives	765,0	754,0	750,0
getSubCategory_Value	731,0	708,0	706,0
getRoot	557,0	549,0	550,0
setCurrentEditorialDepartment	491,0	472,0	467,5
getName	460,0	459,0	461,0
getSubCategory_Value	731,0	708,0	706,0
getRoot	557,0	549,0	550,0

Abbildung B.9 Grafische Darstellung der Ergebnisse für den Objekttyp Menu

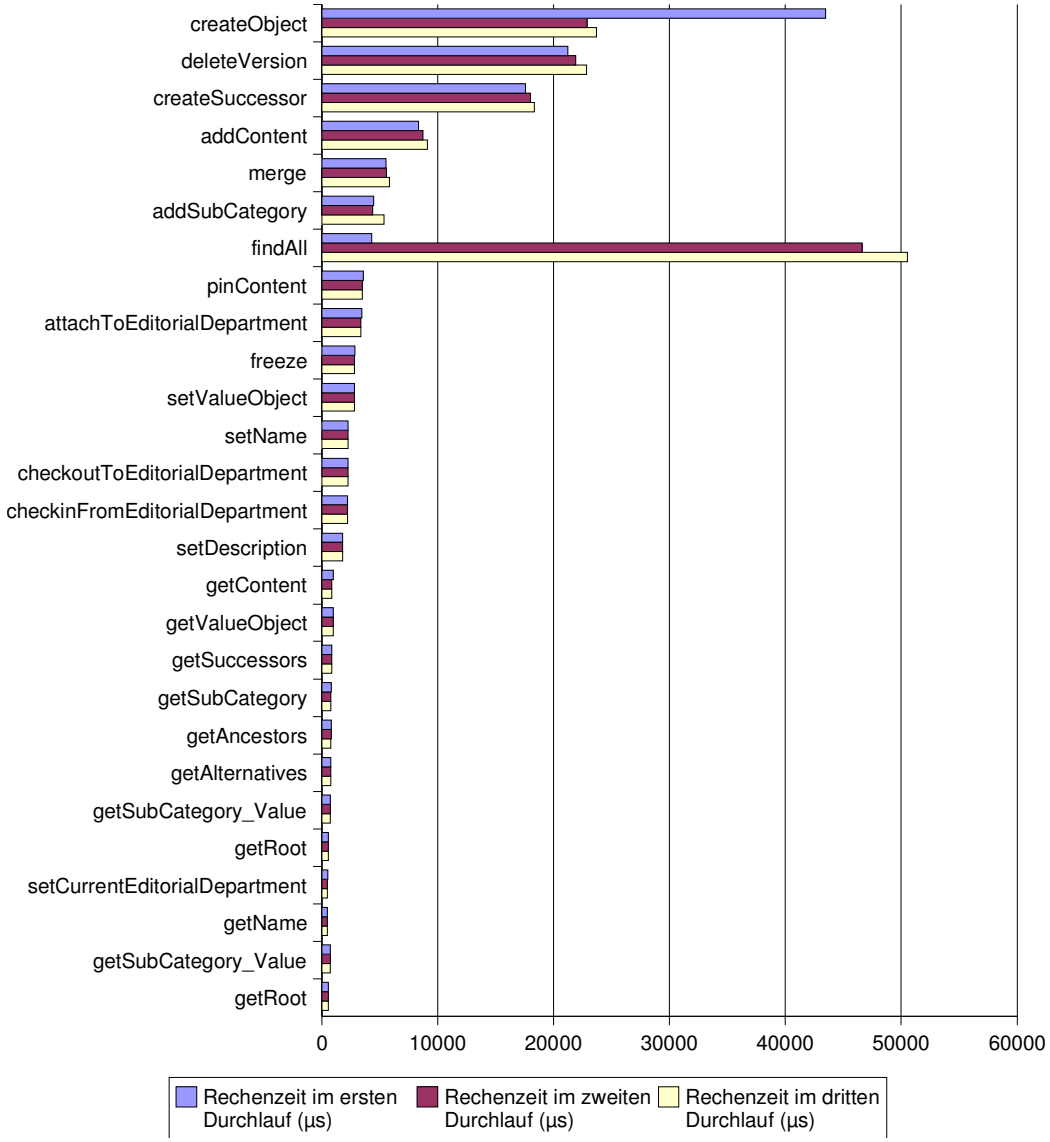
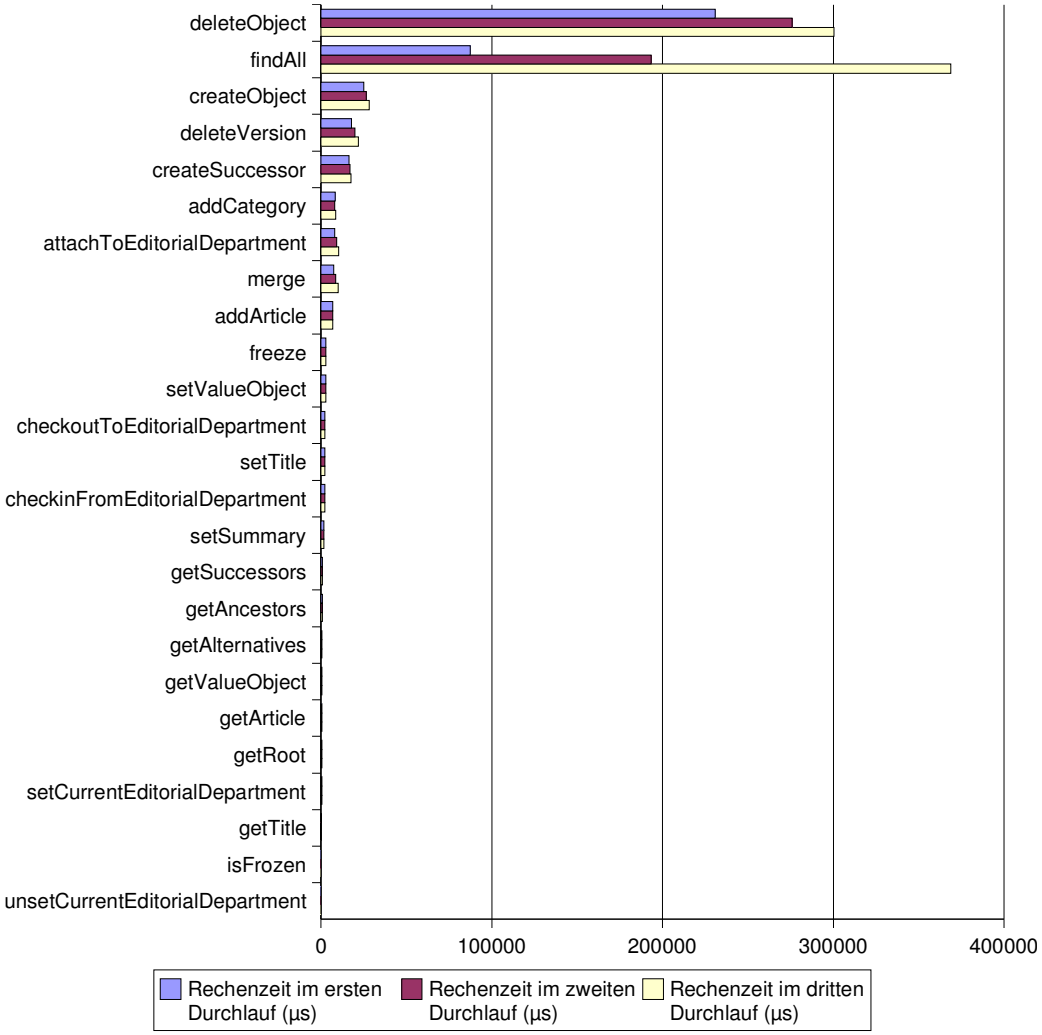


Tabelle B.10 Ergebnisse für den Objekttyp Teaser

Operation	Rechenzeit im ersten Durchlauf (µs)	Rechenzeit im zweiten Durchlauf (µs)	Rechenzeit im dritten Durchlauf (µs)
deleteObject	230836,0	275801,5	300378,0
findAll	87343,5	193283,0	368789,5
createObject	25179,5	26562,5	28262,5
deleteVersion	17995,0	19962,0	21933,5
createSuccessor	16461,0	17112,0	17715,0
addCategory	8191,0	8125,5	8550,0
attachToEditorialDepartment	8073,0	9288,0	10497,0
merge	7363,0	8722,5	10092,5
addArticle	7014,5	6942,0	6935,0
freeze	2802,5	2801,0	2800,0
setValueObject	2778,0	2743,0	2743,0
checkoutToEditorialDepartment	2226,0	2194,0	2195,0
setTitle	2209,0	2174,0	2172,0
checkinFromEditorialDepartment	2150,0	2115,0	2120,0
setSummary	1745,0	1706,0	1706,0
getSuccessors	850,0	849,0	847,0
getAncestors	697,0	685,0	683,0
getAlternatives	642,0	636,0	636,0
getValueObject	573,0	567,0	567,0
getArticle	560,0	539,0	539,0
getRoot	501,0	486,0	485,0
setCurrentEditorialDepartment	478,0	470,5	468,0
getTitle	118,0	96,0	96,0
isFrozen	29,0	19,0	17,0
unsetCurrentEditorialDepartment	26,5	23,0	20,0

Abbildung B.10 Grafische Darstellung der Ergebnisse für den Objekttyp Teaser



B.3 Test der Webservice-Schnittstelle

Tabelle B.11 Ergebnisse für den Objekttyp EditorialDepartment

Operation	Aufrufdauer für nativen Client (µs)	Aufrufdauer für Webservice-Client (µs)	Verlangsamung	prozentuale Verlangsamung
createObject	15842,0	29275,5	13433,5	84,8%
getTeaser	2558,0	9948,0	7390,0	288,9%
getArticle	2486,5	9231,5	6745,0	271,3%
findAll	2532,0	8769,5	6237,5	246,3%
setValueObject	3184,5	9131,0	5946,5	186,7%
getMenu	670,0	6482,0	5812,0	867,5%
getName	752,0	6407,0	5655,0	752,0%
setName	2230,0	7848,5	5618,5	252,0%
getValueObject	4554,5	9102,5	4548,0	99,9%

Abbildung B.11 Grafische Darstellung der Ergebnisse für den Objekttyp EditorialDepartment

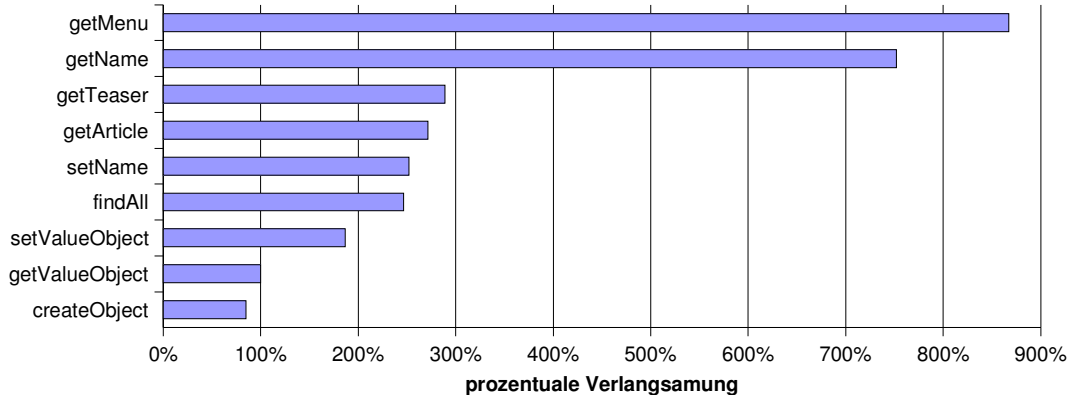


Tabelle B.12 Ergebnisse für den Objekttyp Article

Operation	Aufrufdauer für nativen Client (µs)	Aufrufdauer für Webservice-Client (µs)	Verlangsamung	prozentuale Verlangsamung
setDate	2983,5	11304,5	8321,0	278,9%
getSuccessors	1122,0	8905,5	7783,5	693,7%
getAncestors	989,0	8308,0	7319,0	740,0%
getValueObject	1448,0	8733,5	7285,5	503,1%
getImage	1047,0	8209,0	7162,0	684,0%
createSuccessor	50446,0	57196,0	6750,0	13,4%
attachToEditorialDepartment	19857,5	26599,5	6742,0	34,0%
setValueObject	3506,0	10188,0	6682,0	190,6%
createObject	56654,0	63289,0	6635,0	11,7%
getAlternatives	927,0	7329,0	6402,0	690,6%
getTeaser	1082,0	7195,0	6113,0	565,0%
freeze	7386,0	13207,0	5821,0	78,8%
isFrozen	552,0	6330,0	5778,0	1046,7%
checkoutToEditorialDepartment	4812,0	10555,0	5743,0	119,3%
setCurrentEditorialDepartment	367,0	6105,0	5738,0	1563,5%
checkinFromEditorialDepartment	4694,0	10371,0	5677,0	120,9%
getRoot	718,0	6391,0	5673,0	790,1%
getTitle	642,0	6313,0	5671,0	883,3%
merge	6193,0	11814,5	5621,5	90,8%
addImage	3432,0	9028,0	5596,0	163,1%
setText	1986,0	7579,0	5593,0	281,6%
setTitle	2423,0	7977,0	5554,0	229,2%
unsetCurrentEditorialDepartment	291,0	5377,0	5086,0	1747,8%
deleteObject	109295,0	113419,5	4124,5	3,8%
findAll	50124,5	38052,0	-12072,5	-24,1%

Abbildung B.12 Grafische Darstellung der Ergebnisse für den Objekttyp Article

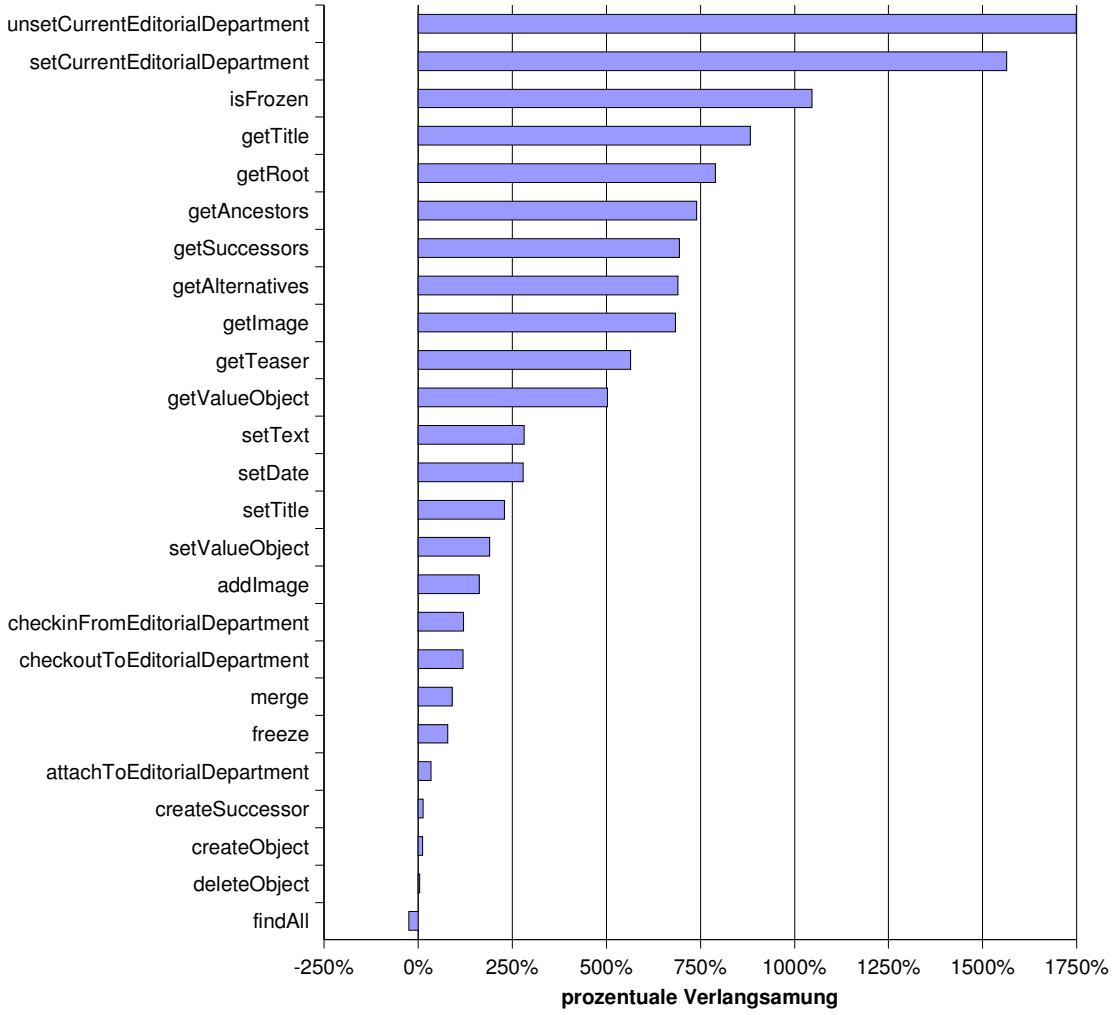


Tabelle B.13 Ergebnisse für den Objekttyp Image

Operation	Aufrufdauer für nativen Client (µs)	Aufrufdauer für Webservice-Client (µs)	Verlangsamung	prozentuale Verlangsamung
findAll	10764,0	22128,0	11364,0	105,6%
setCopyright	2794,0	10602,5	7808,5	279,5%
setWidth	2975,5	10765,5	7790,0	261,8%
setHeight	2884,0	10606,5	7722,5	267,8%
setDescription	2399,5	10117,5	7718,0	321,7%
createObject	8130,5	15487,5	7357,0	90,5%
getValueObject	938,5	7780,0	6841,5	729,0%
setValueObject	3151,0	9393,5	6242,5	198,1%
copyObject	7879,5	13672,0	5792,5	73,5%
deleteObject	3306,5	8981,0	5674,5	171,6%

Abbildung B.13 Grafische Darstellung der Ergebnisse für den Objekttyp Image

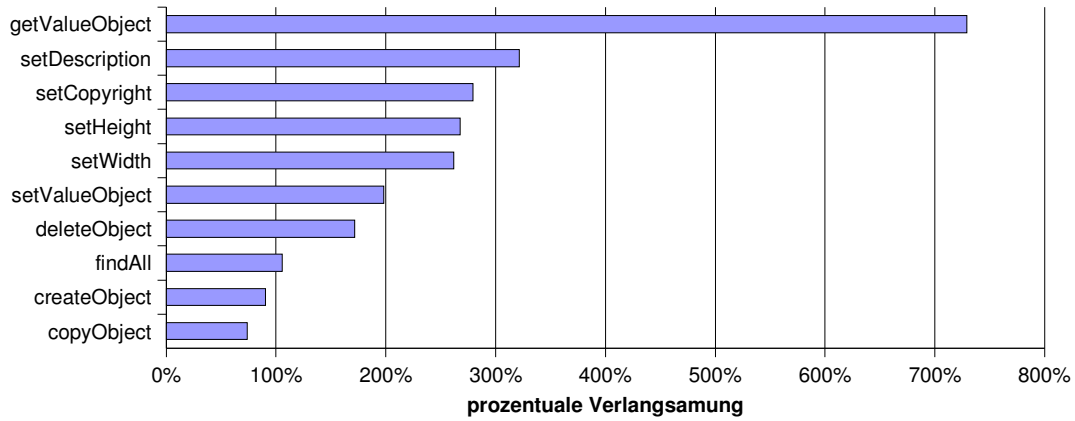


Tabelle B.14 Ergebnisse für den Objekttyp Menu

Operation	Aufrufdauer für nativen Client (µs)	Aufrufdauer für Webservice-Client (µs)	Verlangsamung	prozentuale Verlangsamung
findAll	4479,5	16599,0	12119,5	270,6%
addSubCategory	4660,0	12117,0	7457,0	160,0%
getContent	1164,5	8400,5	7236,0	621,4%
getValueObject	1157,0	8048,5	6891,5	595,6%
getSubCategory_Value	925,0	7435,5	6510,5	703,8%
getSuccessors	1031,0	7371,0	6340,0	614,9%
getAlternatives	963,0	7236,0	6273,0	651,4%
getSubCategory	992,5	7240,5	6248,0	629,5%
getAncestors	986,0	7212,0	6226,0	631,4%
createSuccessor	17802,5	24016,5	6214,0	34,9%
setValueObject	2993,0	8972,5	5979,5	199,8%
setCurrentEditorialDepartment	666,5	6485,5	5819,0	873,1%
deleteVersion	21514,5	27324,5	5810,0	27,0%
isFrozen	553,0	6321,0	5768,0	1043,0%
attachToEditorialDepartment	3570,5	9297,0	5726,5	160,4%
getRoot	740,0	6434,0	5694,0	769,5%
getName	647,0	6309,5	5662,5	875,2%
checkoutToEditorialDepartment	2402,5	8041,0	5638,5	234,7%
merge	5661,0	11296,5	5635,5	99,5%
addContent	8553,0	14183,0	5630,0	65,8%
setDescription	1973,0	7564,5	5591,5	283,4%
pinContent	3719,0	9294,0	5575,0	149,9%
setName	2430,0	8004,0	5574,0	229,4%
freeze	3021,5	8559,0	5537,5	183,3%
checkinFromEditorialDepartment	2360,5	7878,0	5517,5	233,7%
unsetCurrentEditorialDepartment	209,0	5394,5	5185,5	2481,1%
createObject	75187,5	44977,0	-30210,5	-40,2%

Abbildung B.14 Grafische Darstellung der Ergebnisse für den Objekttyp Menu

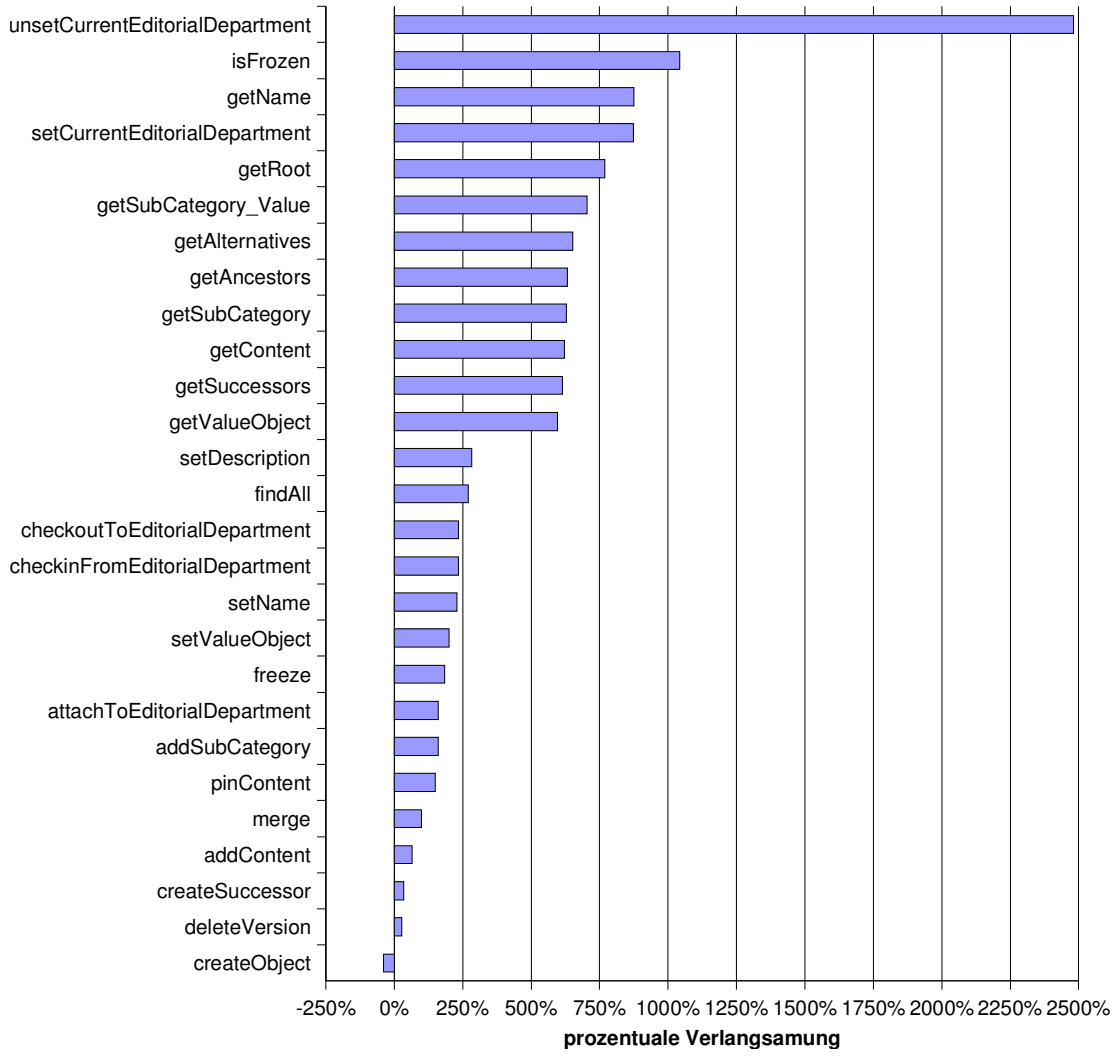
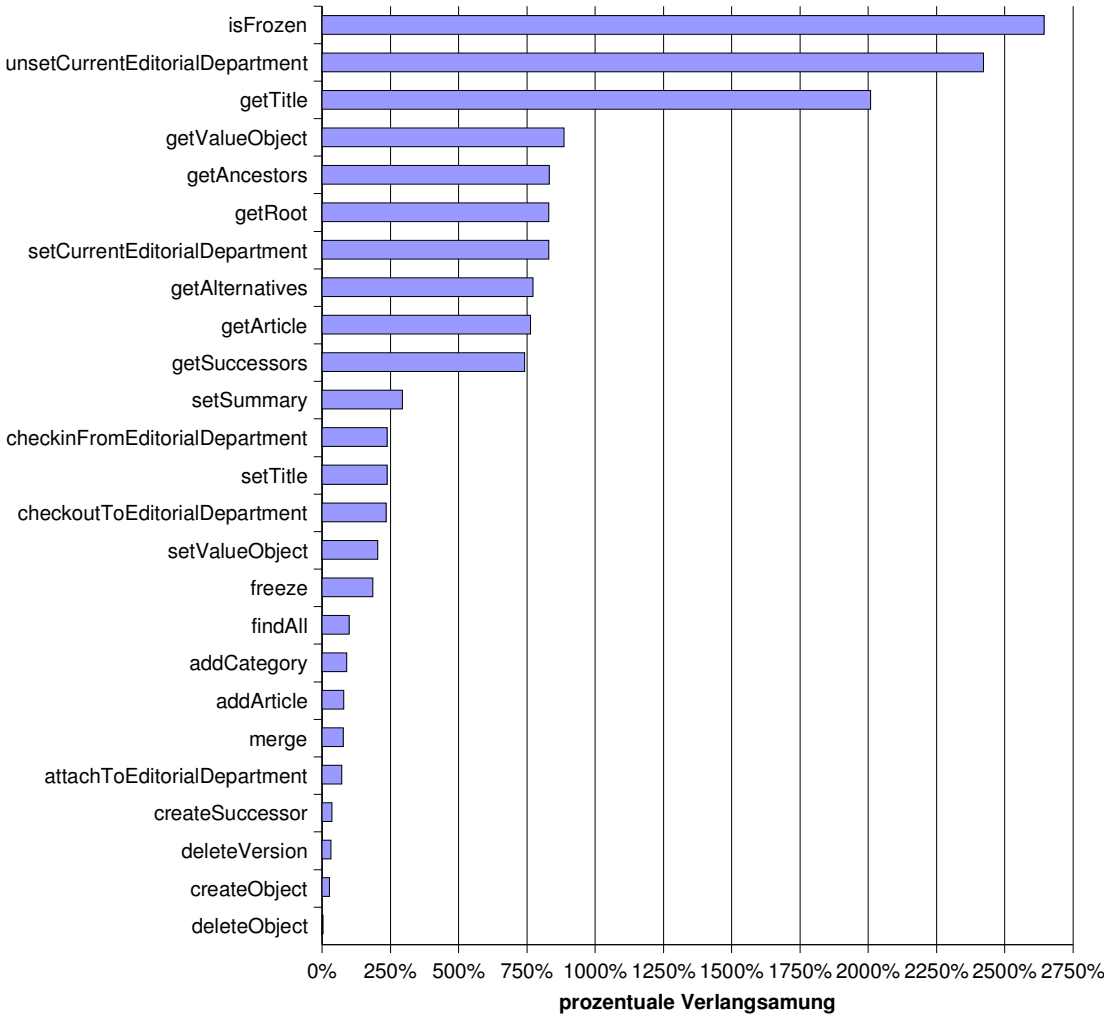


Tabelle B.15 Ergebnisse für den Objekttyp Teaser

Operation	Aufrufdauer für nativen Client (µs)	Aufrufdauer für Webservice-Client (µs)	Verlangsamung	prozentuale Verlangsamung
findAll	88539,5	175936,5	87397,0	98,7%
deleteObject	233751,5	242229,0	8477,5	3,6%
getSuccessors	1045,5	8798,5	7753,0	741,6%
addCategory	8352,0	15903,5	7551,5	90,4%
getAncestors	891,0	8302,0	7411,0	831,8%
createObject	25372,5	32280,0	6907,5	27,2%
getValueObject	769,0	7581,0	6812,0	885,8%
getAlternatives	834,0	7268,0	6434,0	771,5%
isFrozen	229,0	6283,0	6054,0	2643,7%
createSuccessor	16677,0	22714,0	6037,0	36,2%
setValueObject	2950,0	8969,0	6019,0	204,0%
getTitle	297,0	6264,5	5967,5	2009,3%
attachToEditorialDepartment	8255,0	14217,0	5962,0	72,2%
merge	7504,5	13340,5	5836,0	77,8%
deleteVersion	18137,5	23946,0	5808,5	32,0%
getRoot	690,0	6420,0	5730,0	830,4%
addArticle	7202,0	12886,5	5684,5	78,9%
getArticle	741,0	6392,0	5651,0	762,6%
setSummary	1912,0	7533,0	5621,0	294,0%
checkoutToEditorialDepartment	2391,0	8000,0	5609,0	234,6%
setTitle	2365,0	7964,0	5599,0	236,7%
freeze	2979,5	8520,0	5540,5	186,0%
checkinFromEditorialDepartment	2317,0	7830,0	5513,0	237,9%
setCurrentEditorialDepartment	662,0	6156,5	5494,5	830,0%
unsetCurrentEditorialDepartment	213,0	5371,5	5158,5	2421,8%

Abbildung B.15 Grafische Darstellung der Ergebnisse für den Objekttyp Teaser



B.4 Vergleich von generischer und generierter Implementierung

Tabelle B.16 Ergebnisse für den Objekttyp EditorialDepartment

Operation	Rechenzeit der generischen Implementierung (µs)	Rechenzeit der generierten Implementierung (µs)	Beschleunigung	prozentuale Beschleunigung
getMenu	2423,0	484,0	1939,0	80,0%
createObject	31342,0	15656,0	15686,0	50,0%
setName	3906,5	2044,0	1862,5	47,7%
setValueObject	4581,5	2998,5	1583,0	34,6%
getValueObject	6540,0	4368,5	2171,5	33,2%
getTeaser	2989,0	2372,0	617,0	20,6%
getArticle	2869,5	2300,5	569,0	19,8%
findAll	2115,0	2346,0	-231,0	-10,9%
getName	489,0	566,0	-77,0	-15,7%

Abbildung B.16 Grafische Darstellung der Ergebnisse für den Objekttyp EditorialDepartment

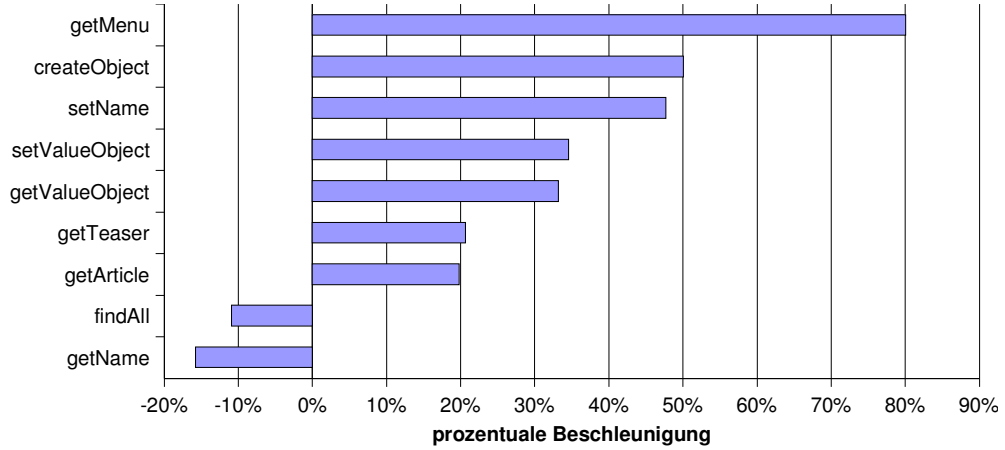


Tabelle B.17 Ergebnisse für den Objekttyp Article

Operation	Rechenzeit der generischen Implementierung (µs)	Rechenzeit der generierten Implementierung (µs)	Beschleunigung	prozentuale Beschleunigung
getAncestors	7567,0	861,0	6706,0	88,6%
getSuccessors	7753,0	896,0	6857,0	88,4%
checkinFromEditorialDepartment	21659,0	4626,0	17033,0	78,6%
attachToEditorialDepartment	18029,0	4508,0	13521,0	75,0%
findAll	16071,5	7200,0	8871,5	55,2%
setText	3347,5	1800,0	1547,5	46,2%
merge	305,0	181,0	124,0	40,7%
createSuccessor	182783,0	109109,0	73674,0	40,3%
setTitle	3641,0	2237,0	1404,0	38,6%
setCurrentEditorialDepartment	4345,5	2797,5	1548,0	35,6%
setDate	4990,0	3246,0	1744,0	34,9%
isFrozen	8969,5	6007,0	2962,5	33,0%
setValueObject	4784,5	3320,0	1464,5	30,6%
createObject	70247,0	50260,0	19987,0	28,5%
addImage	24386,5	19671,5	4715,0	19,3%
unsetCurrentEditorialDepartment	114,0	105,0	9,0	7,9%
checkoutToEditorialDepartment	61088,5	56468,0	4620,5	7,6%
deleteObject	48294,0	49938,5	-1644,5	-3,4%
freeze	697,0	741,0	-44,0	-6,3%
getAlternatives	750,0	803,0	-53,0	-7,1%
getImage	495,5	532,0	-36,5	-7,4%
getTitle	1165,5	1262,0	-96,5	-8,3%
getTeaser	326,0	456,0	-130,0	-39,9%
getRoot	641,0	936,0	-295,0	-46,0%
getValueObject	89,0	366,0	-277,0	-311,2%

Abbildung B.17 Grafische Darstellung der Ergebnisse für den Objekttyp Article

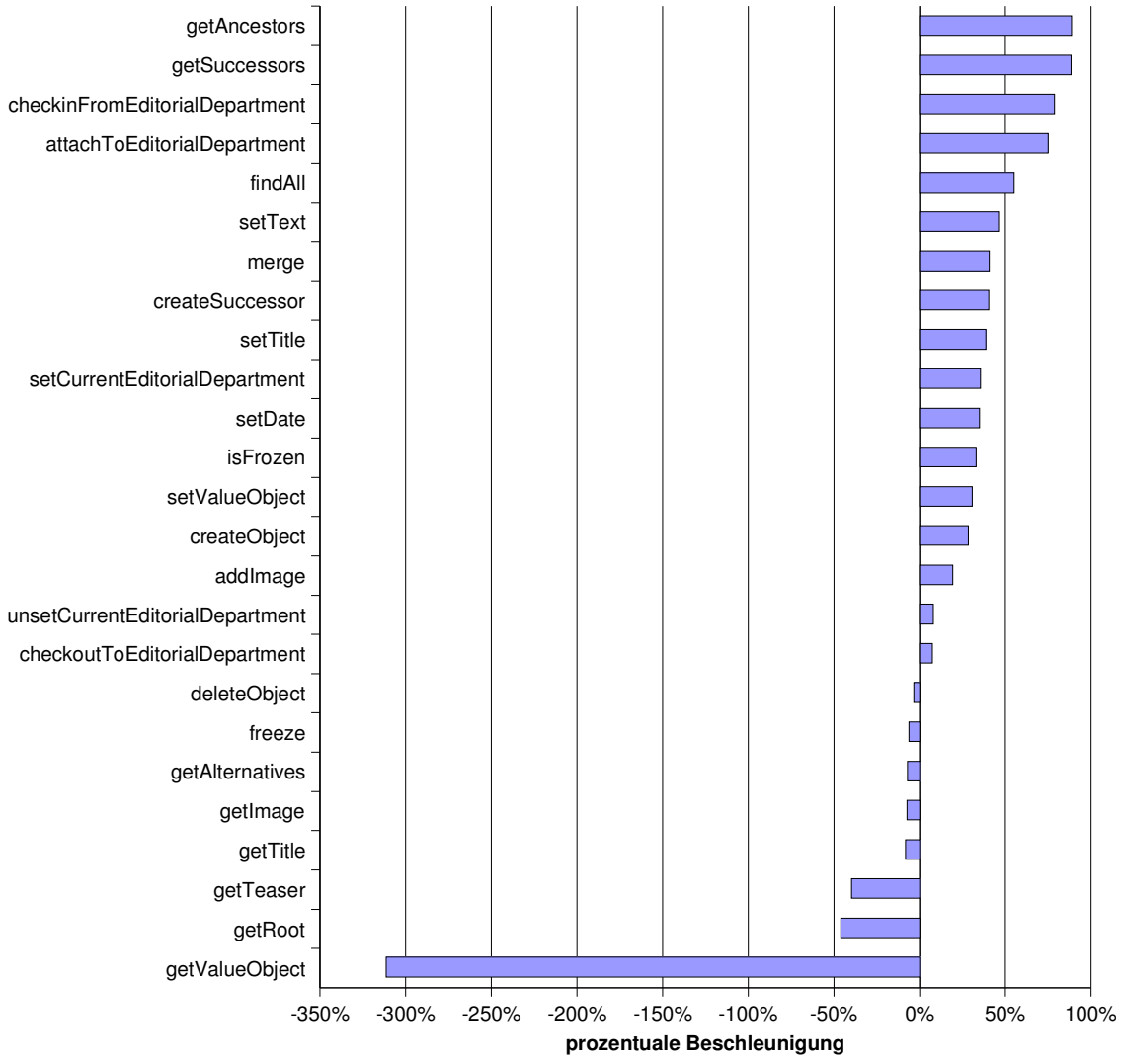


Tabelle B.18 Ergebnisse für den Objekttyp Image

Operation	Rechenzeit der generischen Implementierung (µs)	Rechenzeit der generierten Implementierung (µs)	Beschleunigung	prozentuale Beschleunigung
deleteObject	22903,5	3120,5	19783,0	86,4%
copyObject	18242,0	7693,5	10548,5	57,8%
createObject	18391,0	7944,5	10446,5	56,8%
setDescription	4230,5	2213,5	2017,0	47,7%
setCopyright	4481,0	2608,0	1873,0	41,8%
setValueObject	4537,5	2965,0	1572,5	34,7%
setHeight	4098,0	2698,0	1400,0	34,2%
setWidth	4097,5	2789,5	1308,0	31,9%
getValueObject	1080,5	752,5	328,0	30,4%
findAll	10445,0	10578,0	-133,0	-1,3%

Abbildung B.18 Grafische Darstellung der Ergebnisse für den Objekttyp Image

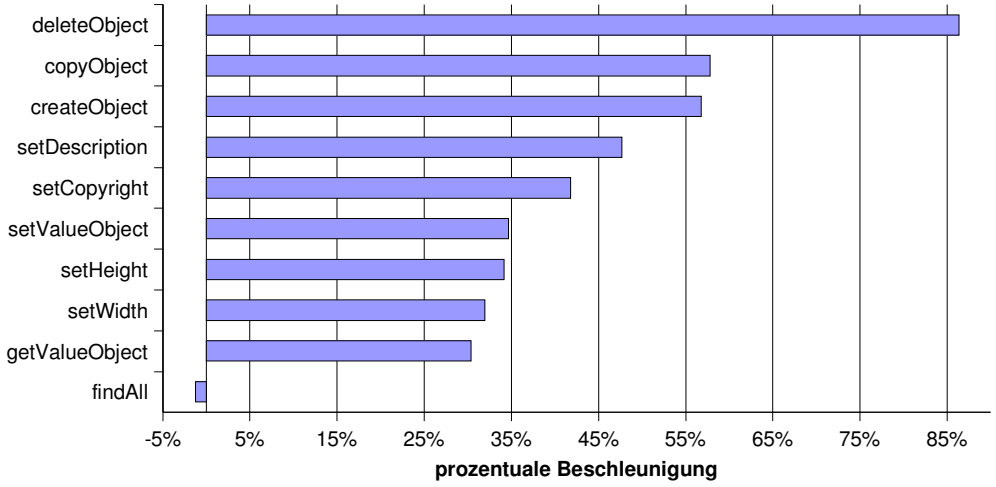


Tabelle B.19 Ergebnisse für den Objekttyp Menu

Operation	Rechenzeit der generischen Implementierung (µs)	Rechenzeit der generierten Implementierung (µs)	Beschleunigung	prozentuale Beschleunigung
getSubCategory_Value	7507,0	739,0	6768,0	90,2%
getSubCategory	7202,5	806,5	6396,0	88,8%
getContent	7448,0	978,5	6469,5	86,9%
addContent	17542,0	4474,0	13068,0	74,5%
checkinFromEditorialDepartment	7825,5	2216,5	5609,0	71,7%
addSubCategory	9280,0	3384,5	5895,5	63,5%
attachToEditorialDepartment	5267,0	2174,5	3092,5	58,7%
freeze	5364,5	2835,5	2529,0	47,1%
setDescription	3366,0	1787,0	1579,0	46,9%
pinContent	6353,5	3533,0	2820,5	44,4%
unsetCurrentEditorialDepartment	40,0	23,0	17,0	42,5%
setName	3766,0	2244,0	1522,0	40,4%
merge	8923,0	5475,0	3448,0	38,6%
setValueObject	4248,5	2807,0	1441,5	33,9%
getAncestors	738,0	800,0	-62,0	-8,4%
getRoot	505,0	554,0	-49,0	-9,7%
deleteVersion	3887,0	4293,5	-406,5	-10,5%
getAlternatives	693,0	777,0	-84,0	-12,1%
getValueObject	831,0	971,0	-140,0	-16,8%
createSuccessor	18105,5	21328,5	-3223,0	-17,8%
getSuccessors	649,0	845,0	-196,0	-30,2%
findAll	6369,0	8367,0	-1998,0	-31,4%
createObject	13302,5	17616,5	-4314,0	-32,4%
getName	326,0	461,0	-135,0	-41,4%
setCurrentEditorialDepartment	331,0	480,5	-149,5	-45,2%
checkoutToEditorialDepartment	26052,5	75001,5	-48949,0	-187,9%
isFrozen	89,0	367,0	-278,0	-312,4%

Abbildung B.19 Grafische Darstellung der Ergebnisse für den Objekttyp Menu

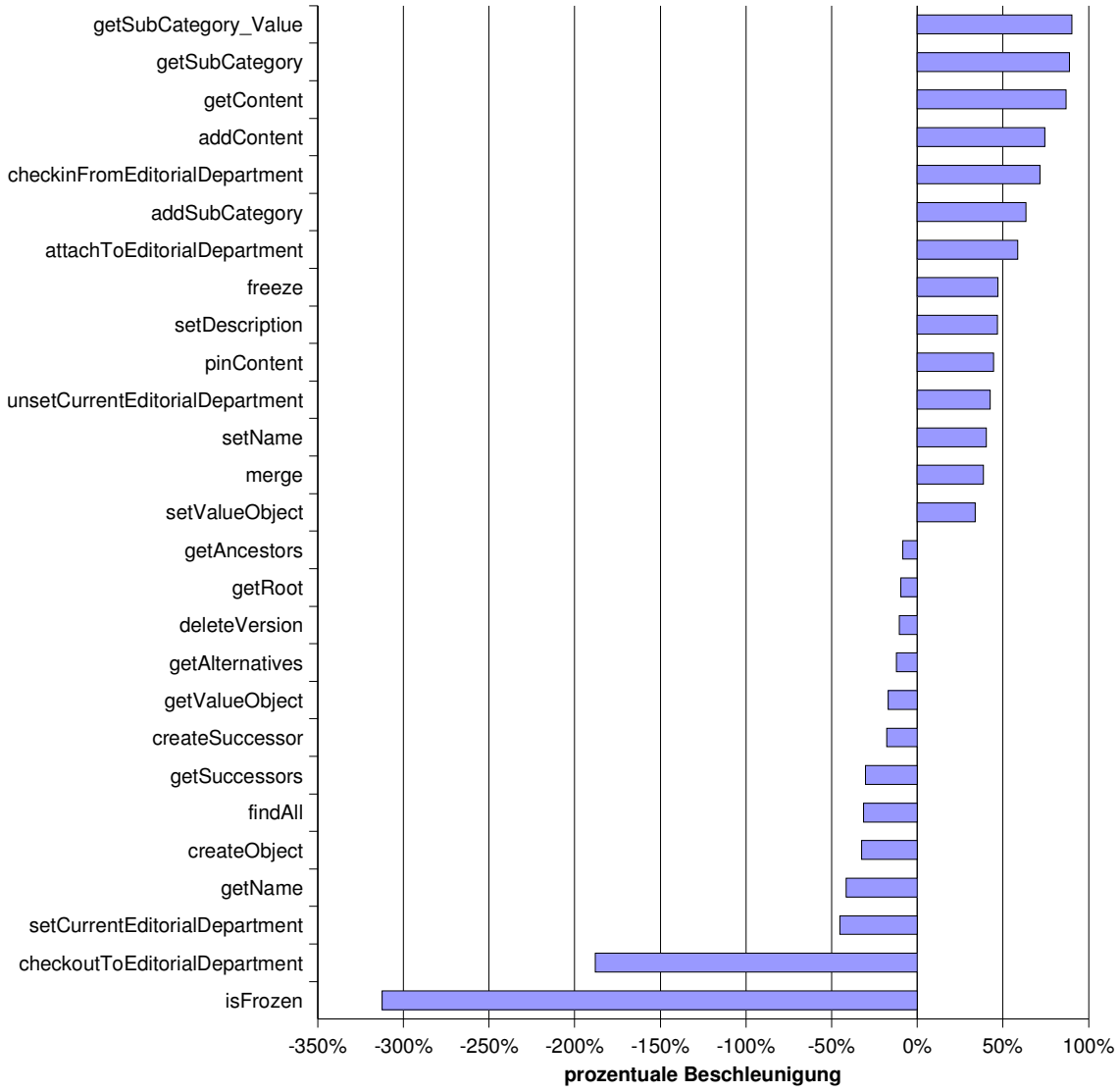
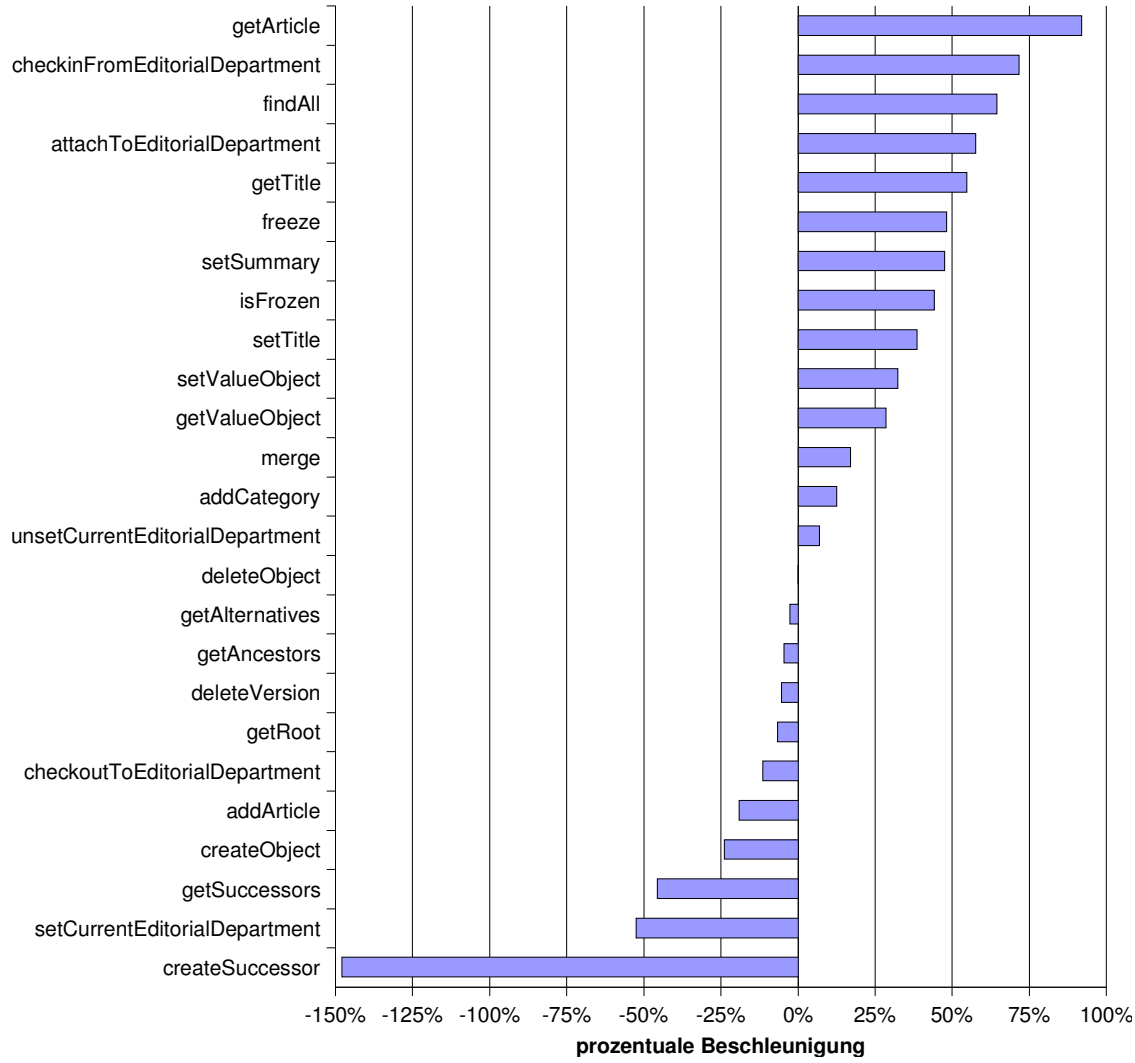


Tabelle B.20 Ergebnisse für den Objekttyp Teaser

Operation	Rechenzeit der generischen Implementierung (µs)	Rechenzeit der generierten Implementierung (µs)	Beschleunigung	prozentuale Beschleunigung
getArticle	6959,5	555,0	6404,5	92,0%
checkinFromEditorialDepartment	7784,0	2205,0	5579,0	71,7%
findAll	19745,5	7016,0	12729,5	64,5%
attachToEditorialDepartment	5027,0	2131,0	2896,0	57,6%
getTitle	245,0	111,0	134,0	54,7%
freeze	5384,0	2793,5	2590,5	48,1%
setSummary	3294,0	1726,0	1568,0	47,6%
isFrozen	77,0	43,0	34,0	44,2%
setTitle	3543,0	2179,0	1364,0	38,5%
setValueObject	4081,0	2764,0	1317,0	32,3%
getValueObject	815,0	583,0	232,0	28,5%
merge	8812,5	7318,5	1494,0	17,0%
addCategory	9229,0	8069,0	1160,0	12,6%
unsetCurrentEditorialDepartment	29,0	27,0	2,0	6,9%
deleteObject	17975,0	17951,5	23,5	0,1%
getAlternatives	631,0	648,0	-17,0	-2,7%
getAncestors	674,0	705,0	-31,0	-4,6%
deleteVersion	83841,5	88353,5	-4512,0	-5,4%
getRoot	473,0	504,0	-31,0	-6,6%
checkoutToEditorialDepartment	22606,0	25186,5	-2580,5	-11,4%
addArticle	6855,5	8166,0	-1310,5	-19,1%
createObject	13316,5	16491,0	-3174,5	-23,8%
getSuccessors	590,0	859,5	-269,5	-45,7%
setCurrentEditorialDepartment	312,0	476,0	-164,0	-52,6%
createSuccessor	94249,5	233565,5	-139316,0	-147,8%

Abbildung B.20 Grafische Darstellung der Ergebnisse für den Objekttyp Teaser



Literaturverzeichnis

[Apa03a] The Apache Jakarta Project (Apache Software Foundation):
Velocity Template Engine
http://jakarta.apache.org/velocity/
2003

[Apa03b] The Apache Web Services Project (Apache Software Foundation):
Axis
http://ws.apache.org/axis/
2003

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

[BBC+99] Bernstein, P.A., Bergstraesser, T., Carlson, J., Pal, S., Sanders, P., Shutt, D.:
Microsoft Repository Version 2 and the Open Information Model
Information Systems 24:2 (1999), pp. 71-98

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

[Coh99] Cohen, S.:
From Product Line Architectures to Products
ECOOP'99 Workshop on Object-Technology for Product-Line Architectures,
Lisbon, Portugal

[CVS03] Concurrent Versions System
The open standard for version control
http://www.cvshome.org/
2003

[FK98] Foster, I., Kesselmann, C.:
The Grid: Blueprint for a New Computing Infrastructure
Morgan Kaufmann
1998

[GHJV95] Gamma, E., Held, R., Johnson, R., Vlissides, J.:
Design Patterns - Elements of Reusable Object-Oriented Software
Addison-Wesley
1995

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

[IBM03a] IBM Rational Software:
Rational ClearCase
http://www.rational.com/products/clearcase/
2003

[IBM03b] IBM:
DB2 Universal Database
http://www.ibm.com/software/data/db2/udb/
2003

[ISO98] International Standards Organization:
Programming languages – C++
ISO/IEC 14882:1998(E)
September 1998.

[Iro03] IronFlare AB:
Orion Application Server
http://www.orionserver.com/
2003

[JCP01] Java Community Process:
UML/EJB Mapping Specification, Version 1.0
http://www.jcp.org/aboutJava/communityprocess/review/jsr026/
Mai 2001

[JCP02] Java Community Process:
Java Metadata Interface (JMI) Specification, Version 1.0
http://jcp.org/aboutJava/communityprocess/final/jsr040/
Juni 2002

[JCP03a] Java Community Process:
JavaServer Pages Specification, Version 2.0
http://jcp.org/aboutJava/communityprocess/final/jsr152/
November 2003

[JCP03b] Java Community Process:
Java Servlet Specification, Version 2.4
http://jcp.org/aboutJava/communityprocess/final/jsr154/
November 2003

[JCP03c] Java Community Process:
Enterprise JavaBeans Specification, Version 2.1
http://jcp.org/aboutJava/communityprocess/final/jsr153/
November 2003

[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

[KH03] Kovse, J., Härder, T.:
V-Grid - A Versioning Services Framework for the Grid
Proc. 3rd Int. Workshop Web-Datenbanken, Berliner XML-Tage 2003, Berlin
Oktober 2003

[KP88] Krasner, G., Pope, S.:
A Cookbook for using the Model-View-Controller User Interface Paradigm in Smalltalk-80
Journal of Object Oriented Programming 1:3 (1988), pp. 26-49

[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

[Nov02] Novosoft:
Novosoft Metadata Framework and UML Library
http://nsuml.sourceforge.net/
2002

[OMG02a] Object Managment Group:
Common Object Request Broker Architecture (CORBA) Specification, Version 3.0.2
http://www.omg.org/cgi-bin/doc?formal/02-12-02
Dezember 2002

[OMG02b] Object Management Group:
Meta-Object Facility (MOF) Specification, Version 1.4
http://www.omg.org/cgi-bin/doc?formal/02-04-03
April 2002

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

[OMG02d] Object Management Group:
OMG XML Metadata Interchange (XMI) Specification, Version 1.2
http://www.omg.org/cgi-bin/doc?formal/2002-01-01
Januar 2002

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

[OMG03b] Object Management Group:
Common Warehouse Metamodel (CWM) Specification, Version 1.1
http://www.omg.org/cgi-bin/doc?formal/03-03-02
März 2003

[OMG03c] Object Management Group:
XML Metadata Interchange (XMI) Specification, Version 2.0
http://www.omg.org/cgi-bin/doc?formal/2003-05-02
Mai 2003

[SCK+96] Simos, M., Creps, D., Klinger, C., Levine, L., Allemang, D.:
Organization Domain Modeling (ODM) Guidebook, Version 2.0
Informal Technical Report for STARS, STARS-VC-A025/001/00
Juni 1996

[SVB02] Sturm, T., von Voss, J., Boger, M.:
Generating Code from UML with Velocity Templates
UML 2002, pp. 150-161, Dresden

[Sun01] Sun Microsystems, Inc.:
Java 2 Platform Enterprise Edition Specification, v1.3
http://java.sun.com/j2ee/j2ee-1_3-fr-spec.pdf
August 2001

[Sun03] Sun Microsystems, Inc.:
Java Remote Method Invocation Specification
http://java.sun.com/j2se/1.4.2/docs/guide/rmi/spec/rmiTOC.html
2003

[Tic85] Tichy, W.:
RCS: a system for version control
Software-Practice & Experience 15:7 (1985), pp. 637–654

[Van00] VanDoren, E.:
Cyclomatic Complexity
Carnegie Mellon University, Software Engineering Institute,
Software Technology Roadmap
http://www.sei.cmu.edu/str/descriptions/cyclomatic.html
Juli 2000

[Vir03] Virtual Machinery
JHawk Product Overview
http://www.virtualmachinery.com/jhawkprod.htm
2003

[W3C01] World Wide Web Consortium (W3C):
Web Services Description Language (WSDL) 1.1
W3C Note, http://www.w3.org/TR/wsdl
März 2001

[W3C03] World Wide Web Consortium (W3C):
SOAP Version 1.2 Part 1: Messaging Framework
W3C Recommendation, http://www.w3.org/TR/SOAP
Juni 2003

[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