Diplomarbeit

Werkzeuge für die Migrationsunterstützung von Anwendungen auf neue Rahmenwerksversionen

Andreas Havenstein
Martin-Luther-Straße 27
20459 Hamburg
Matrikelnummer 4626012

Februar 2003

Erstbetreuung: Prof. Dr. Heinz Züllighoven

Zweitbetreuung: Dr. Thorsten Altenkirch

Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Erklärung
Ich versichere hiermit, diese Arbeit selbständig und unter ausschließlicher Zuhilfenahme der in der Arbeit aufgeführten Hilfsmittel erstellt zu haben.

Hamburg, den

Andreas Havenstein

Martin-Luther-Straße 27
20459 Hamburg

Erstbetreuung:

Prof. Dr. Heinz Züllighoven

Fachbereich Informatik
Arbeitsbereich Softwaretechnik
Universität Hamburg
Vogt-Kölln-Straße 30
22527 Hamburg

Zweitbetreuung:

Dr. Thorsten Altenkirch

School of Computer Science & IT
University of Nottingham

Inhaltsverzeichnis

1. Einleitung und Überblick

Motivation

Seit über 10 Jahren wird der Werkzeug-Material-Ansatz angewendet und weiterentwickelt. Der Werkzeug-Material-Ansatz (siehe [Züllighoven 98]) ist ein Methodenrahmen für den Softwareentwicklungsprozess. Er zeigt einen Weg auf, wie unter Verwendung bestimmter Entwicklungsdokumente und Metaphern Software von hoher Gebrauchsqualität erstellt werden kann. Eine zentrale Bedeutung kommt dabei den Metaphern Werkzeug, Automat und Material zu, deren Kürzel (WAM) dem Methodenrahmen den Namen geben. Diese Metaphern geben eine Sichtweise auf die Dinge vor, die im Anwendungsbereich vorgefunden werden und die in der Software modelliert werden.

An der Universität Hamburg wurde am Arbeitsbereich Softwaretechnik des Fachbereichs Informatik ein Rahmenwerk (engl. framework) entwickelt, das die Konstruktion einer Anwendung nach dem WAM-Ansatz unterstützt. Im Design einer WAM-Anwendung finden sich die im Anwendungsbereich identifizierten Werkzeuge und Materialien wieder. Das Rahmenwerk enthält Konzepte und Implementationen für die Konstruktion von Software-Werkzeugen und –Materialien. Das Rahmenwerk wurde zunächst in der Forschung und Lehre eingesetzt. Die Erkenntnisse von Studien- und Diplomarbeiten flossen in das Rahmenwerk ein, so dass das Rahmenwerk kontinuierlich um neue Aspekte erweitert wurde.

1999 war das Rahmenwerk in der Entwicklung soweit vorangeschritten, dass es seine Praxistauglichkeit im professionellen Einsatz beweisen sollte. Von der „universitären Spielwiese“ kommend fand es nun seinen Einsatz in konkreten Projektkontexten. Daraus ergaben sich neue Impulse und Ideen, die zu kleineren bis größeren Umstrukturierungen des Frameworks führten.

Seitdem werden in einem halbjährigen Releasezyklus neue Rahmenwerksversionen veröffentlicht, für die bereits vorhandene Komponenten verbessert und neu entwickelte Komponenten hinzugefügt werden.

Die Umstrukturierungen am Rahmenwerk führen in vielen Fällen auch zu Änderungen der äußeren, von Anwendungen verwendeten Rahmenwerksschnittstelle. Anwendungen, die auf der älteren Version des Rahmenwerkes basieren, können ohne Anpassung auf diese neue Schnittstelle die neue Version nicht verwenden. Damit Anwendungen von der um neue Funktionalität erweiterten oder fehlerbereinigten neuen Rahmenwerksversion profitieren können, müssen die Anwendungsentwickler alle Rahmenwerks-Verwendungsstellen in der Anwendung überprüfen und gegebenenfalls auf die Struktur des neuen Rahmenwerkes anpassen.

Für die Anwendungsentwickler entsteht ein nicht unerheblicher Aufwand, um Anwendungen auf die neue Rahmenwerksversion zu migrieren.

Zielsetzung

In dieser Arbeit soll geklärt werden, wie der Aufwand der Migration von Anwendungen auf neue Rahmenwerksversionen reduziert werden kann. Das Ziel ist die Konstruktion eines Werkzeuges, das den Anwendungsentwickler durch den Migrationsprozess leitet und Migrationsschritte automatisiert durchführen kann.

Vorgehen

Als Gegenstand für die folgenden Untersuchungen wird ein Beispiel-Miniaturrahmenwerk definiert, das als Grundlage für eine Beispielanwendung dient.

Stefan Roock hat sich im Rahmen seiner Dissertation mit der Kategorisierung von Änderungen an Softwaresystemen beschäftigt und eine Anzahl Änderungstypen identifiziert (siehe [Roock 03]). Diese Änderungstypen werden exemplarisch auf das Beispielrahmenwerk angewendet. Durch die Änderungen am Rahmenwerk lässt sich die auf der alten Rahmenwerksschnittstelle basierende Anwendung mit der neuen Rahmenwerksversion nicht mehr verwenden. Der zur Anwendungsmigration nötige Aufwand wird analysiert.

Es wird dann ein Blick auf verwandte Forschungsbereiche geworfen, die sich mit der Wandlung und Migration von Softwaresystemen befassen. Ein bewährter Ansatz für die Migration von Legacy-Systemen und Datenbanken ist die inkrementelle Migration über eine Vermittlerschicht. Die Vermittlerschicht kapselt das geänderte System und bietet der Anwendung die Schnittstelle des unveränderten Altsystems. Dadurch wird eine schrittweise Migration auf das Neusystem ermöglicht. Es wird untersucht, ob dieser Ansatz auf die Migrationsunterstützung einer rahmenwerksbasierten Anwendung übertragbar ist. Dazu wird die Praktikabilität einer Rahmenwerks-Adapterschicht zur Kapselung einer geänderten Rahmenwerksversion analysiert.

Schließlich wird aufgezeigt, wie ein Migrationswerkzeug konstruiert werden kann, das den Prozess der Anwendungsmigration steuert und unterstützt. Damit ein Werkzeug notwendige Migrationsschritte identifizieren und ausführen kann, müssen Rahmenwerksänderungen für das Werkzeug erkennbar sein. Stefan Roock hat eine Methode zur Änderungsdokumentation im Quellcode entwickelt, die anhand der am Beispielrahmenwerk vorgenommenen Modifikationen demonstriert wird. Die Konstruktion eines Werkzeuges wird beschrieben, das auf Basis dieser Änderungsdokumentation den Migrationsprozess leiten kann. Abschließend wird anhand des Rahmenwerks- und Anwendungsbeispiels ausgewertet, inwieweit das Werkzeug den Migrationsprozess unterstützen kann.

Gliederung der Arbeit

In Kapitel 2 werden die Grundlagen für die Untersuchungen dieser Arbeit dargestellt. Der WAM-Ansatz wird näher beleuchtet, und die Definition eines Rahmenwerkes wird gegeben. Als Beispiel wird dazu das JWAM-Rahmenwerk vorgestellt, das zur Erstellung von Anwendungen basierend auf den WAM-Konzepten verwendet werden kann. Zur Vereinfachung der Darstellungen in den folgenden Abschnitten wird ein Miniaturausschnitt des Rahmenwerkes definiert, das JWAMMicro-Rahmenwerk. Dieses Miniaturrahmenwerk wird als durchgehendes Beispiel für die folgenden Kapitel verwendet. Es werden allgemeine Änderungsklassifikationen beschrieben und die konkret für die neue JWAMMicro-Version vorgenommenen Änderungen und Erweiterungen vorgestellt. Aus den Änderungen am Rahmenwerk resultiert die Notwendigkeit, auf der alten Rahmenwerksversion basierende Anwendungen zu migrieren. Der Migrationsaufwand wird analysiert und es wird untersucht, welche Migrationsstrategien und Werkzeuge in verwandten Forschungsfeldern entwickelt worden sind.

In Kapitel 3 wird untersucht, ob ein inkrementeller Migrationsprozess über die Verwendung einer Rahmenwerks-Adapterschicht realisiert werden kann. Zur Verwendung von alten Anwendungen mit einer neuen Rahmenwerksversion wird die Erstellung einer Zwischenschicht beschrieben, die als Vermittler zwischen alter Rahmenwerksschnittstelle und neuem Rahmenwerk fungiert.

In Kapitel 4 werden die Grundlagen zur Dokumentation von Rahmenwerksänderungen vorgestellt. Anhand des JWAMMicro-Beispielrahmenwerks werden die Dokumentationsmöglichkeiten veranschaulicht.

In Kapitel 5 wird die Migrationslösung der werkzeugunterstützten Anpassung von Anwendungen beschrieben. Im Detail wird dargestellt, wie ein entsprechendes Migrationswerkzeug konstruiert werden kann.

Sprachliche und grafische Konventionen

Alle formalen Darstellungen in dieser Arbeit, wie Klassen-, Objekt- und Interaktionsdiagramme, entsprechen dem UML-Standard, wie er in [Jacobson et. al. 99] und [Booch et al. 98] beschrieben ist.

Diese Arbeit ist im JWAM-Kontext entstanden. Die Sprache, in der Klassen- und Methodennamen im JWAM-Kontext verfasst sind, ist englisch. Ich habe im Rahmen dieser Arbeit die Namen nicht künstlich eingedeutscht, gebe aber die deutsche Übersetzung an, wo es hilfreich ist.

Danksagung

Ich möchte an dieser Stelle allen Menschen danken, die mir bei der Erstellung dieser Arbeit behilflich waren.
Ich danke Heinz Züllighoven für viele hilfreiche Anregungen und für die Erstbetreuung dieser Arbeit. Ich danke Thorsten Altenkirch, der die Zweitbetreuung dieser Arbeit übernommen hat.
Ein großer Dank gebührt Stefan Roock, der mir die Anregung zu dieser Arbeit gab. Stefan Roock leistete einen großen Beitrag zu dem Erstdesign des Migrationswerkzeugs und stand mir auch in der folgenden Zeit mit Rat und Unterstützung zur Seite.
Ich danke Björn Ostermann und meiner Familie für das Korrekturlesen.
Außerdem danke ich Sebastian Sanitz, der mich kurze Zeit bei dem Redesign des Migrationswerkzeuges begleitet hat.

2. Evolution von Rahmenwerken und Migration von Anwendungen

Das Ziel dieses Kapitels ist, die Grundlagen der Untersuchungen der folgenden Kapitel darzustellen. Dazu wird zunächst der Begriff des Rahmenwerkes definiert.
Am Beispiel des JWAM-Rahmenwerkes wird eine konkrete Rahmenwerksarchitektur beschrieben, die für eine möglichst einfache Veranschaulichung der in den folgenden Kapiteln erläuterten Probleme auf eine Minimalversion, das JWAMMicro-Rahmenwerk, reduziert wird. Auf dem JWAMMicro-Rahmenwerk basierend wird eine kleine Beispielanwendung vorgestellt.
Die an einem Rahmenwerk möglichen Änderungen zwischen zwei Versionen werden definiert und an dem Beispielrahmenwerk demonstriert. Die für die Beispielanwendung daraus resultierende Migrationsnotwendigkeit wird erläutert.
Nach einem Blick auf die Migrationsstrategien und Werkzeuge verwandter Forschungsbereiche wird abschließend dargestellt, welche prinzipiell möglichen Lösungen für die Zusammenführung von geändertem Rahmenwerk und alter Anwendung existieren.

2.1. Rahmenwerke

2.1.1. Definition Rahmenwerk

Um dem Begriff des Rahmenwerkes (engl. framework) näher zu kommen, sollen drei Definitionen helfen:

> *„Ein Framework besteht aus einer Menge von zusammenhängenden Klassen, die einen wiederverwendbaren Entwurf für eine bestimmte Art von Software darstellen.“* ([Deutsch 89])

> *„Das Framework bestimmt die Architektur ihrer Anwendung. Es definiert die Struktur im großen, ..., die Zusammenarbeit der Klassen und Objekte sowie den Kontrollfluß.“* ([Gamma et al. 96], S. 37)

> *„Ein Anwendungsrahmenwerk (Application Framework) ist für die Entwicklung von Anwendungssoftware in einem fachlich eingegrenzten Anwendungsbereich gedacht. Es gibt die softwaretechnische Architektur und die fachlichen Abstraktionen in Form einer generischen Lösung vor.“* ([Züllighoven 98], S. 119)

Diesen Definitionen gemeinsam ist der Aspekt der Wiederverwendung von Entwurfsentscheidungen unter Nutzung einer durch das Rahmenwerk vorgegebenen Architektur. In einem Rahmenwerk sind architektonische Lösungen für einen bestimmten Anwendungskontext enthalten. Das Rahmenwerk bildet das Gerüst, auf dem die zu entwickelnden konkreten Anwendungen basieren.
Dabei nutzt eine Rahmenwerksanwendung das Rahmenwerk nicht nur wie die Funktionen einer Klassenbibliothek, vielmehr wird die Anwendung in die Kontrollflüsse des Rahmenwerkes eingebettet. Der Kontrollfluss geht nicht von der Anwendung aus, sondern das Rahmenwerk ruft Methoden der Anwendungsklassen. Diese Umkehrung des Kontrollflusses wird auch als Hollywood-Prinzip („Don't call us; we'll call you!“) oder Greyhound-Prinzip ("Leave the driving to us.") bezeichnet.

Während es bei einer Klassenbibliothek um die Wiederverwendung von Code geht, spielt bei einem Rahmenwerk im Wesentlichen die Wiederverwendung von Entwurfsentscheidungen eine Rolle (siehe [Gamma et al. 96], S.37).

Bei der Konstruktion von Rahmenwerken lassen sich grundsätzlich zwei Extrema unterscheiden: Blackbox- und Whitebox-Rahmenwerke (siehe [Johnson & Foote 88]).

Ein Blackbox-Rahmenwerk stellt sich als ein geschlossenes System mit einer definierten Schnittstelle dar. Bei der Anwendungsentwicklung werden Rahmenwerksobjekte erzeugt, die mit anderen Rahmenwerksobjekten konfiguriert werden können. Der Kontrollfluss der Blackbox-Rahmenwerksobjekte lässt sich nicht direkt beeinflussen, über die Konfigurationsobjekte kann aber das Verhalten angepasst werden. Die Kopplung und Konfiguration der Rahmenwerksobjekte geschieht dynamisch zur Laufzeit.

Ein Whitebox-Rahmenwerk ist eine offengelegte Klassenstruktur, in die die Anwendungsklassen über Vererbung eingebettet werden. Der Kontrollfluss wird vom Rahmenwerk vorgegeben, über die Subklassenbildung besteht prinzipiell eine größere Flexibilität als bei der Benutzung eines Blackbox-Rahmenwerkes. Die Kopplung von Anwendungsklassen an Rahmenwerksklassen ist statisch und stellt damit eine engere Kopplung dar als im Falle der Blackbox-Rahmenwerksnutzung.

Häufig findet man aber Rahmenwerke, deren Verwendung eine Mischform von Black- und Whitebox-Nutzung darstellt. Bestimmte Teile eines Rahmenwerkes sind blackboxartig nutzbar, von anderen Teilen werden in der Anwendung Subklassen gebildet, es liegt also eine Whiteboxnutzung vor. Bei einer solchen Mischnutzung spricht man auch von Greybox-Rahmenwerk (siehe [Earles 00]).

2.1.2. Der WAM-Ansatz und das JWAM-Rahmenwerk

Als Beispiel für ein konkretes Rahmenwerk wird in diesem Abschnitt der WAM-Ansatz und das JWAM-Rahmenwerk vorgestellt. Das JWAM-Rahmenwerk ist ein Rahmenwerk zur Erstellung interaktiver Anwendungen nach dem WAM-Ansatz. Weder das JWAM noch der WAM-Ansatz stehen im Mittelpunkt dieser Arbeit, aber das Problem der Migration von Anwendungen auf neue Rahmenwerksversionen hat sich aus dem Einsatz des JWAM-Rahmenwerkes in konkreten Projekten ergeben.

Der WAM-Ansatz beschreibt eine Sichtweise auf die Softwareentwicklung, die geprägt ist von bestimmten Leitbildern und Metaphern. Eine detaillierte Beschreibung wird von Züllighoven gegeben (siehe [Züllighoven 98]). Die zentralen Metaphern des WAM-Ansatzes sind das Werkzeug, der Automat und das Material, woraus sich die Abkürzung WAM ergibt. Anhand dieser Metaphern und dem Leitbild des „Arbeitsplatzes für qualifizierte und eigenverantwortliche Tätigkeit“ wird im WAM-Ansatz beschrieben, wie Anwendungen konstruiert werden können, in denen sich die im Anwendungsbereich identifizierten Konzepte wiederfinden. Bei der Analyse werden also relevante Werkzeuge und Materialien im Anwendungsbereich identifiziert, die in der Software modelliert werden, so dass eine Strukturähnlichkeit zwischen dem fachlichen Begriffsgebäude des Anwendungsbereiches und der Softwarearchitektur hergestellt wird.
Der WAM-Ansatz ist nicht nur eine Modellierungstechnik, sondern beschreibt einen Methodenrahmen mit einer Anzahl von Dokumenttypen und Entwurfsmustern und definiert einen

evolutionären Entwicklungsprozess. Diese Aspekte stehen jedoch nicht im Fokus dieser Arbeit.

Der Grundstein für den WAM-Ansatz wurde Ende der 80er Jahre in einem großen Bankenprojekt gelegt, die weitere Ausgestaltung wurde seit 1991 maßgeblich von Züllighoven an der Universität Hamburg vorangetrieben. Im Rahmen von Studien- und Diplomarbeiten wurde zunächst ein objektorientiertes Rahmenwerk in der Sprache C++ zur Konstruktionsunterstützung von WAM-Anwendungen erstellt, das dann in Java (daher der Name JWAM) reimplementiert und stetig erweitert wurde.
1999 war das JWAM-Rahmenwerk dann soweit konsolidiert, dass es in professionellen Projekten eingesetzt werden konnte.
Im Rahmenwerk finden sich die Konzepte Werkzeug, Automat und Material als Rumpfklassen wieder. Bei der Anwendungsentwicklung werden damit im Anwendungsbereich vorgefundene Artefakte als Werkzeuge und Materialien im WAM-Sinne modelliert und in die Rahmenwerksstruktur eingebettet.
In der Architektur ist die Unterstützung von arbeitsplatzübergreifender kooperativer Arbeit vorgesehen.
Die folgende Abbildung gibt einen Überblick über die Architektur des JWAM-Rahmenwerkes.

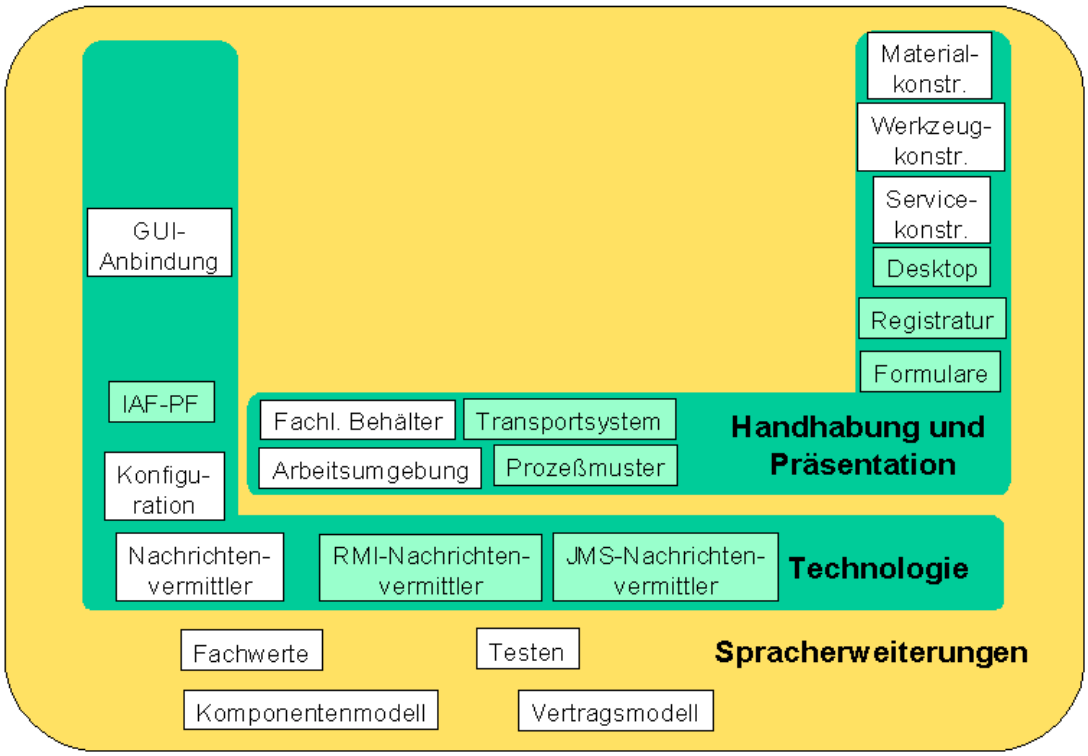


Abbildung 2-1: JWAM-Schichtenarchitektur

Die Rahmenwerksstruktur ist in Schichten unterteilt, die jeweils bestimmte softwaretechnische Komponenten und Konzepte zusammenfassen. Die Schichtenbildung und Verwendung gehorcht dabei folgenden Regeln (siehe auch die Beschreibung der WAM-Modellarchitektur in [Züllighoven 98], S.358ff.):

- *„Eine Schicht organisiert die softwaretechnischen Komponenten einer Modellarchitektur zu einer fachlich und technisch motivierten Entwurfs- und Konstruktionseinheit.“* ([Züllighoven 98], S. 352)

- Die Schichten sind hierarchisch angeordnet. *„Komponenten einer Schicht dürfen nur Komponenten derselben oder einer beliebigen tieferen objektorientierten Schicht verwenden.“* ([Züllighoven 98], S. 365)

Durch den hierarchischen Aufbau des Rahmenwerkes wird sichergestellt, dass Abhängigkeiten immer nur in eine Richtung bestehen: Klassen oberer Schichten verwenden Klassen in unteren Schichten durch Benutztbeziehungen oder indem sie Klassen unterer Schichten beerben.

Das JWAM-Rahmenwerk gliedert sich in folgende Schichten (beginnend von der untersten Schicht):

Spracherweiterungsschicht
In der Spracherweiterungsschicht sind Komponenten zusammengefasst, die konzeptionell eigentlich in der Programmiersprache vorhanden sein sollten. Da die Programmiersprache Java aber beispielsweise keine Sprachelemente zur Programmierung von Fachwerten vorsieht, müssen diese Konzepte nachprogrammiert werden. Gleiches gilt für eine vollständige Implementierung des Vertragsmodells nach Meyer (siehe [Meyer 91]), das im JWAM als eigenes Sub-Framework realisiert und in der Spracherweiterungsschicht angesiedelt worden ist.

Technologieschicht
In der Technologieschicht befinden sich Komponenten, die Technologieanschlüsse kapseln und unter einer anwendungsnäheren Schnittstelle anbieten.
Zwei Beispiele aus dem aktuellen JWAM-Rahmenwerk sollen dies verdeutlichen:

- Das Package `de.jwam.technology.messagebroker` enthält eine Nachrichtenvermittler-Komponente (engl. message broker). Im Sinne des WAM-Ansatzes ist der Nachrichtenvermittler eine Komponente, um kooperatives Arbeiten an verteilten Arbeitsplätzen zu ermöglichen. In die Anwendungsklassen soll aber kein Wissen über die technische Infrastruktur einfließen. Der Nachrichtenvermittler besitzt lediglich eine fachliche Schnittstelle für den Empfang und das Verschicken von Nachrichten. Ob die Nachrichtenverteilung über RMI (Remote Method Invocation) oder aber über eine JMS-Lösung (Java Messaging Services) realisiert wird, ist für die Anwendungsprogrammierung verborgen.[1]
- Das Package `de.jwam.technology.guimanagement` beschreibt eine Schnittstelle für die Verwaltung von Elementen einer Benutzungsoberfläche. In Anwendungsklassen wird auf GUI-Elemente über die Schnittstelle des GUI-Kontextes zugegriffen. Welches konkrete GUI-Toolkit von dem GUI-Kontext verwaltet wird, ist für die Konstruktion eines Werkzeuges nicht sichtbar und unerheblich.

Diese Beispiele zeigen, dass in der Technologieschicht von der konkreten Technologie abstrahiert Schnittstellen angeboten werden. Die Anwendungsklassen werden von der verwendeten Technologie entkoppelt, die Technologie ist dadurch auf einfache Weise austauschbar.
Die Technologieschicht kennt nur die Spracherweiterungsschicht, Elemente der Handhabungs- und Präsentationsschicht darf sie gemäß der Schichtenbildungsregel nicht verwenden.

Handhabungs- und Präsentationsschicht
In dieser Schicht befinden sich Schnittstellenvorgaben oder Standardimplementationen für Metaphern und Konzepte im Sinne der WAM-Analyse.

[1] RMI und JMS sind Technologien zur Kommunikation zwischen getrennten Prozessräumen. Sie sind Bestandteil der Java-Entwicklerdistributionen von Sun (siehe [Sun J2SE 02], [Sun J2EE 02]).

Zur Verdeutlichung werden einige der Packages in der Handhabungs- und Präsentationsschicht vorgestellt:

- Package `de.jwam.handling.thing`: Hier befindet sich eine allgemeine Schnittstelle und eine Standardimplementation für das Konzept eines Gegenstandes (engl. thing). Diese „Dinge“ können Werkzeuge, Automaten oder Materialien sein. Ihnen gemeinsam ist, dass man sie benennen kann und sie eine Identität besitzen.
- Package `de.jwam.handling.toolconstruction`: Für die Implementation von Werkzeugen sind hier Standardimplementationen vorhanden.
- Package `de.jwam.handling.environment`: Die Metapher der Arbeitsumgebung (engl. environment) wird hier durch die `Environment`-Klasse ausgefüllt. Im Environment liegen alle Dinge eines JWAM-Arbeitsplatzes.

Im Rahmenwerk finden sich keine anwendungsspezifischen Ausprägungen dieser Metaphern.

Anwendungen, die mit dem JWAM-Rahmenwerk entwickelt werden, werden in die JWAM-Architektur eingebettet. In der Abbildung 2-1 sind die oberen Schichten des Rahmenwerks U-förmig dargestellt, um diese Einbettung zu verdeutlichen. Bei der Anwendungsentwicklung wird die Handhabungs- und Präsentationsschicht whitebox-artig zur Ableitung konkreter Anwendungsklassen genutzt. Die abgeleiteten Werkzeuge werden im Environment, also der Arbeitsumgebung von JWAM, verwaltet und in den Kontrollfluss eingebunden.

2.1.3. Das JWAMMicro-Rahmenwerk

Die im Abschnitt 2.1.2 beschriebenen Teile des JWAM-Rahmenwerkes stellen nur einen kleinen Ausschnitt der im Rahmenwerk vorhandenen Klassen und Konzepte dar. Die zum jetzigen Zeitpunkt aktuelle JWAM-Version 1.7 besteht aus über 1300 Klassen und über 600 Testklassen. Vor dem Hintergrund dieser mächtigen und umfangreichen Klassenstruktur stellt sich dann bei jedem Release einer neuen Version die Frage, an wieviel Stellen Änderungen am Rahmenwerk vorgenommen wurden, die in den jeweiligen konkreten Anwendungen nachgezogen werden müssen.
Da sich die Änderungen und die Änderungsauswirkungen kategorisieren lassen, ist es nicht notwendig, verschiedene Versionen des kompletten JWAM-Rahmenwerkes als Untersuchungsgegenstand zu verwenden. Die Änderungskategorien lassen sich anhand einer überschaubaren Untermenge der Rahmenwerksklassen analysieren.
Aus diesem Grund wird ein Minimalrahmenwerk definiert, in dem sich sämtliche Probleme der Rahmenwerksevolution und Anwendungsmigration wiederfinden. Das JWAMMicro-Rahmenwerk ist eine auf wenige Kernklassen reduzierte Version des JWAM-Rahmenwerkes.
Die im Folgenden vorgestellte Sammlung von Klassen bildet sicher kein für den Praxiseinsatz ausreichendes Anwendungsrahmenwerk, als Untersuchungsobjekt erfüllt sie aber ihren Zweck.

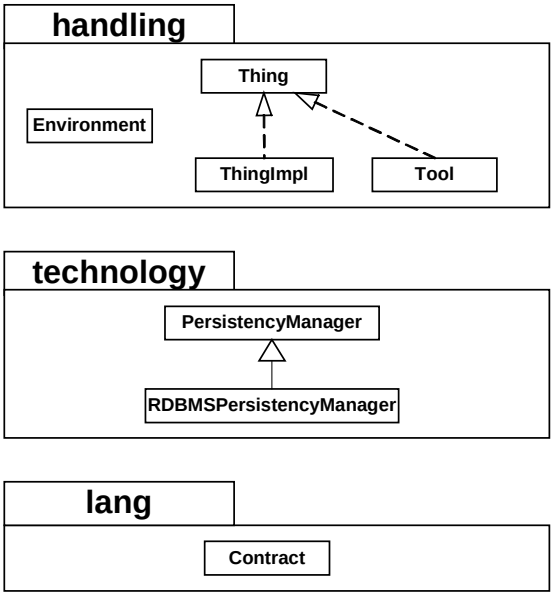
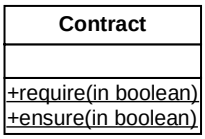


Abbildung 2-2: JWAMMicro-Rahmenwerk

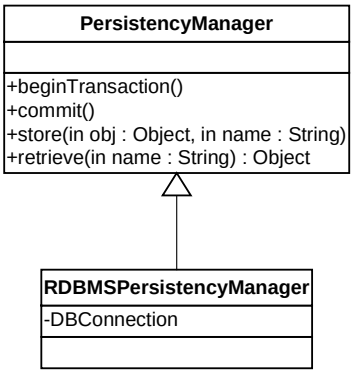
Die Abbildung 2-2 gibt einen Überblick über alle im JWAMMicro enthaltenen Packages und Klassen. Wie im JWAM-Rahmenwerk ist auch diese reduzierte Version in die drei bereits beschriebenen Schichten strukturiert, deren Klassen und Schnittstellen jetzt im Einzelnen vorgestellt werden.

Package de.jwammicro.lang



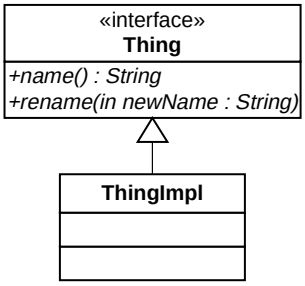
Die Klasse `Contract` in der Spracherweiterungsschicht bietet die Möglichkeit, Verträge gemäß des von Meyer beschriebenen Vertragsmodells zu formulieren (siehe [Meyer 91]). Vor- und Nachbedingungen können über die Methoden `require` und `ensure` im Anwendungscode explizit beschrieben werden.

Package de.jwammicro.technology

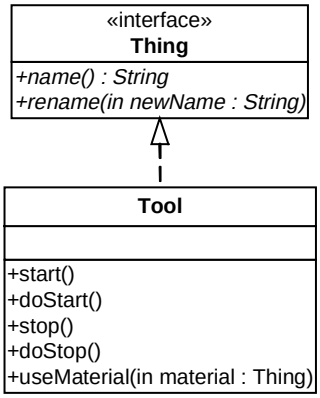


Die Klasse `PersistencyManager` in der Technologieschicht spezifiziert ein Protokoll, um auf einfache Weise Materialien auf einem Persistenzmedium zu speichern und sie wieder zu laden. Als Zugriffsschlüssel wird der Name eines Materials verwendet.
Über die Methoden `beginTransaction` und `commit` können Transaktionen geklammert werden.
In der Unterklasse `RDBMSPersistencyManager` wird eine Implementation basierend auf einer relationalen Datenbank angeboten.

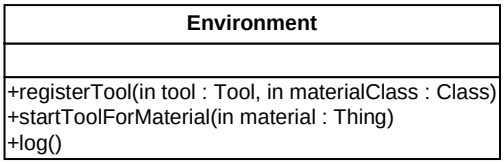
Package de.jwammicro.handling
Die Kernmetaphern des WAM-Ansatzes finden sich im Handling-Package wieder.



Wie schon oben für das JWAM-Rahmenwerk beschrieben, sind Materialien und Werkzeuge Ausprägungen von greifbaren Dingen, die benannt werden können und die eine Identität besitzen. Softwaretechnisch werden diese Gegenstände durch Ableitung von der Standardimplementation `ThingImpl` oder durch Implementation der `Thing`-Schnittstelle umgesetzt.



Die `Tool`-Klasse ist die Oberklasse für alle mit dem Rahmenwerk modellierten Werkzeuge. Durch die `start`- und `stop`-Methoden wird ein vom Rahmenwerk definiertes Protokoll vorgegeben. Das Starten und Stoppen eines Werkzeugs wird über das `Environment` gesteuert. Von der `Tool`-Klasse abgeleitete Werkzeugklassen müssen die Methoden `doStart` und `doStop` werkzeugspezifisch implementieren, die von den `start`- und `stop`-Methoden der `Tool`-Klasse gerufen werden (Entwurfsmuster Template Method, siehe [Gamma at al. 96]).



Das durch die Klasse `Environment` realisierte Umgebungskonzept dient zur Verwaltung der in einem JWAMMicro-System verwendeten Werkzeuge. Werkzeuge können für bestimmte Materialklassen an der Umgebung registriert und gestartet werden.
Zusätzlich besteht die Möglichkeit, über die Umgebung Ereignisse in einem Logbuch festzuhalten.

Die Klassen in der beschriebenen Package-Struktur bilden das Rahmenwerk in der Version 1.0.

> **Begriff Version:**
> Eine neue Version wird von den Entwicklern des Rahmenwerkes immer dann vergeben, wenn diese veröffentlicht (den Anwendungsentwicklern zugänglich gemacht) wird. Versionen werden so benannt, dass ihre chronologische Reihenfolge aus dem Namen abgeleitet werden kann (angelehnt an [Roock 03], Kapitel 6.2).

2.1.4. JWAMMicro-Beispielanwendung

Mit den beschriebenen Klassen des JWAMMicro-Rahmenwerks der Version 1 kann nun eine einfache Anwendung konstruiert werden, anhand derer später die Auswirkungen von Rahmenwerksänderungen analysiert werden können.
Die Beispielanwendung realisiert ein Kontowerkzeug (engl. account tool) zur Bearbeitung von Konten (engl. accounts). Die folgende Abbildung zeigt einen Ausschnitt der Vererbungs- und Benutzt-Beziehungen der Anwendungs- und Rahmenwerksklassen
.

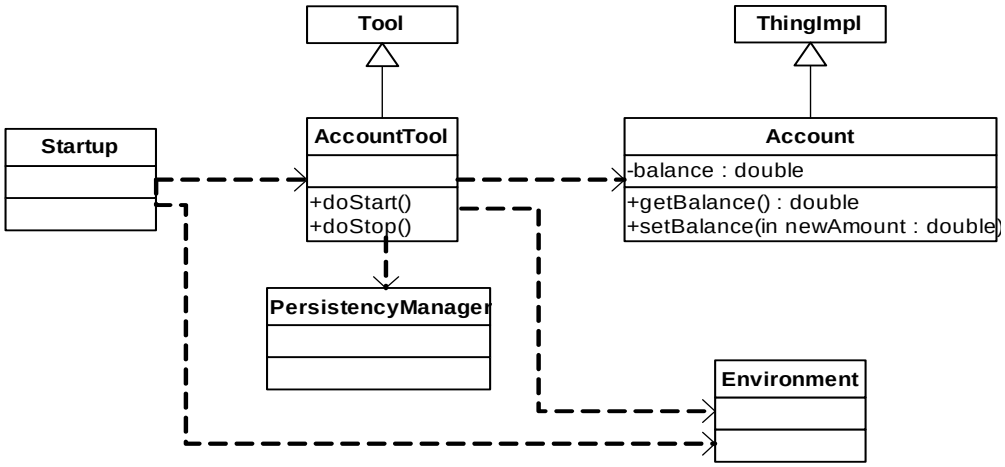


Abbildung 2-2: Klassendiagramm AccountTool

Die Klasse `Startup` registriert das Werkzeug `AccountTool` für `Account`-Materialien an der Umgebung (Environment) und startet über die Umgebung dann die Bearbeitung eines konkreten Materials.
Der konkrete Implementationscode der Anwendungsklassen findet sich im Anhang A.1.

2.2. Rahmenwerks-Evolution

Im letzten Abschnitt wurde das Rahmenwerk JWAMMicro Version 1 mit einer darauf basierenden Anwendung vorgestellt. In diesem Kapitel wird nun aufgezeigt, welche Modifikationen prinzipiell an einem Rahmenwerk vorgenommen werden können und welcher Migrationsbedarf daraus entsteht. Modifikationen können atomarer oder zusammengesetzter Natur sein und lassen sich bezüglich der Kompatibilität zu bestehenden Anwendungen in verschiedene Klassen einteilen. Die beschriebenen Modifikationen werden durch Änderungen am JWAMMicro-Rahmenwerk beispielhaft verdeutlicht, das damit zur Version 2 reift.

2.2.1. Evolution

Die Entwicklung von Software kann als ein fortdauernder Lernprozess verstanden werden, bei dem die Entwickler mit der Zeit immer besser die Anforderungen des Zielsystems erkennen. Auch Rahmenwerke werden nicht einmalig vollständig durchgeplant und implementiert, sondern wachsen in einem inkrementellen und iterativen Prozess.
Die Erfahrungen, die bei der Verwendung des Rahmenwerkes gesammelt wurden, können in der jeweils nächsten Iteration in das Rahmenwerk einfließen. Das Ziel der daraus resultierenden Rahmenwerksreorganisationen ist immer, den Wert des Rahmenwerkes zu steigern. Konkrete Ziele können sein (siehe auch [Foote 92]):

- höhere Stabilität
- Beseitigung von Fehlern
- größere Performanz
- Hinzufügen neuer Funktionalität
- Verschlankung durch Entfernen unbenötigter Funktionalität
- Vereinfachung zur besseren Verständlichkeit

Das JWAM-Rahmenwerk wird seit 1999 mit den Techniken des Extreme Programming (abgekürzt XP) weiterentwickelt. Der XP-Entwicklungsprozess, der unter anderem von Kent Beck beschriebenen worden ist (siehe [Beck 00]), stellt eine konkrete Ausprägung eines inkrementellen und iterativen Vorgehens dar.
Eine komplette Einführung in den XP-Entwicklungsprozess zu geben, würde den Rahmen dieser Arbeit sprengen. Dennoch sollen kurz wichtige Techniken erwähnt werden.

- **Testen:**
  Für alle Komponenten eines Software-Systems sollten automatisiert ausführbare Testklassen vorhanden sein. Als Faustregel gilt, dass jede Operation in einem Test erfasst wird. Mit einem Testwerkzeug kann „auf Knopfdruck“ die Funktionalität des gesamten Systems geprüft werden.
  Die Abdeckung des Rahmenwerkes mit Testklassen ist unter anderem die Basis dafür, dass Änderungen am Rahmenwerk durchgeführt werden können, ohne Fehler oder unerwünschte Seiteneffekte an anderen Stellen zur Folge zu haben. Alle Tests müssen vor der Änderung und nach der Änderung erfolgreich durchlaufen und sichern damit die Funktion des Systems zu.

- **Einfaches Design:**
  Softwaresysteme sollten möglichst einfach konstruiert sein. Einfache Entwürfe sind schneller umzusetzen und leichter zu verstehen.
  Auf Rahmenwerke bezogen bedeutet das vor allem, den Bau von „Technik auf Vorrat“ zu vermeiden. Es soll also möglichst nichts konstruiert werden, was aktuell nicht benötigt

wird und nur spekulativ einen Nutzen in kommenden Zeiten besitzt. Im JWAM-Kontext wird deshalb nach dem Leitsatz verfahren: „Erst verwenden, dann wiederverwenden." Für das Rahmenwerk werden also nur Anforderungen umgesetzt, die in konkreten Anwendungskontexten entstanden sind (siehe [Lippert et al. 02], S.180f).

- **Refactoring**
  Refactoring ist die Umstrukturierung eines Softwaresystems, bei der die bestehende Funktionalität erhalten bleibt. Notwendig wird ein Refactoring beispielsweise, wenn die Struktur eines Softwaresystems schwer verständlich oder unflexibel ist. Ziel ist es, eine bessere Struktur im Sinne der oben beschriebenen Wertsteigerung eines Softwaresystems zu erhalten. Die fortlaufende Designverbesserung über Refactorings ist ein wichtiger Teil des XP-Entwicklungsprozesses.

Die Anwendung der XP-Techniken im Rahmen der JWAM-Entwicklung wurde auch als XF, als *Extreme Frameworking*, bezeichnet (siehe [Roock 02]).
Das Verändern und Umstrukturieren des Frameworks ist also nicht eine aus der Not geborene Technik der Fehlerbeseitigung, sondern ist ein Teil des Entwicklungsprozesses.

2.2.2. Rahmenwerks-Modifikationen

Nachdem im letzten Abschnitt die Motivation für Rahmenwerksänderungen erläutert wurde, soll nun geklärt werden, welche Änderungen prinzipiell vorgenommen werden können und wie sich diese Änderungen kategorisieren lassen.
Zur Verdeutlichung werden Änderungen am JWAMMicro-Rahmenwerk beschrieben, die insgesamt zu einer Version 2 des Rahmenwerkes führen.

Opdyke war der Erste, der in seiner Arbeit „Refactoring Object Oriented Frameworks" (siehe [Opdyke 92]) detailliert Refactorings und ihre Auswirkungen untersuchte. Opdyke identifizierte 23 mögliche einfache Refactorings, die an C++-Softwaresystemen durchführbar sind. Opdyke unterteilte diese Refactorings in 4 Kategorien:

- Erzeugung einer Programmiereinheit
- Löschen einer Programmiereinheit
- Ändern einer Programmiereinheit
- Verschieben eines Attributes

Er beschreibt, wie die Refactorings am Code durchgeführt werden können, so dass die Semantik des gesamten Systems erhalten bleibt. Vorraussetzung dafür ist, dass zum Zeitpunkt der Rahmenwerks-Modifikation der gesamte Applikationscode sichtbar und veränderbar ist. Diese Vorraussetzung ist für ein Rahmenwerk im seltensten Fall erfüllt.
Martin Fowler unterscheidet deshalb „public interfaces" und „published interfaces" (siehe [Fowler 99]). Ein „public interface" ist eine softwaretechnisch als „öffentlich" bezeichnete Schnittstelle.
Ein „published interface" geht darüber hinaus. Es bezeichnet eine Schnittstelle, die veröffentlicht und in die „äußere Welt" zur Benutzung getragen worden ist. Der Ersteller der Schnittstelle hat nach der Veröffentlichung keinen Zugriff auf Code, der die Schnittstelle verwendet.
Ein Rahmenwerksrelease stellt beispielsweise eine veröffentlichte Schnittstelle dar. Das veröffentlichte Rahmenwerk kann weltweit von Anwendungsentwicklern verwendet werden, der Anwendungscode ist für die Rahmenwerkshersteller nicht sichtbar. Änderungen an der Rahmenwerksschnittstelle können zu Konflikten mit den bestehenden Anwendungsklassen führen.

Fowler stellte später einen Katalog von Refactorings vor, in denen Refactorings aufgabenorientiert geordnet zusammengetragen worden waren (siehe [Fowler 00]). Fowler legt den Begriff des Refactorings wie folgt aus:

> *„Refactoring (noun): a change made to the internal structure of software to make it easier to understand and cheaper to modify without changing its observable behaviour."* ([Fowler 00], S.53)

Stefan Roock hat sich im Rahmen seiner Dissertation mit den unterschiedlichen Definitionen des Refactoring-Begriffes befasst und festgestellt, dass weder die Beschreibung der Refactorings von Opdyke, noch die Refactoring-Definition von Fowler für die Untersuchung der Rahmenwerksänderungen geeignet ist. Bei der Rahmenwerksentwicklung steht zum Zeitpunkt des Refactorings nicht der Code aller darauf basierender Anwendungen zur Verfügung, wie von Opdyke gefordert. Und Rahmenwerksänderungen wie das Umbenennen von Klassen und Operationen führen zu einer nach außen sichtbaren Strukturänderung, was der Definition von Fowler widerspricht.
Roock verwendet in seiner Arbeit deshalb den allgemeineren Begriff der Modifikation für alle Änderungen am Rahmenwerk:

> **Begriff : Modifikation**
> *„Eine Modifikation an einer Verwendungseinheit beschreibt eine Änderung an ihrer Schnittstelle oder Implementation, die wieder zu einer syntaktisch korrekten Verwendungseinheit führt."* ([Roock 03], Kapitel 6.2)

Roock verwendet dabei den Begriff Verwendungseinheit als Oberbegriff für Rahmenwerke, Klassenbibliotheken und Komponenten, also für softwaretechnische Einheiten, die wiederverwendet werden können. Bei der Untersuchung von Rahmenwerksänderungen hat Roock eine Anzahl atomarer Modifikationen identifiziert, die er als Basismodifikationen bezeichnet.

> **Begriff : Basismodifikation**
> *„Eine Basismodifikation ist eine atomare Modifikation, die nicht weiter in Submodifikationen aufgeteilt werden kann."* ([Roock 03], Kapitel 6.2)

Basismodifikationen beziehen sich immer auf Klassen, Schnittstellen, Operationen und Attribute, die im Folgenden als Elemente bezeichnet werden.

Basismodifikationen sind:

- **Erzeuge Element.** Beispiel: Einer Schnittstelle wird eine neue Methode hinzugefügt
- **Lösche Element.** Beispiel: Eine Klasse wird gelöscht.
- **Ersetze Elementname.** Beispiel: Eine Operation wird umbenannt.
- **Ersetze Referenzliste.** Eine Referenzliste gibt alle Elemente an, die von einem Element referenziert werden. Eine Klassendefinition referenziert beispielsweise andere Klassenelemente, mit denen sie in einer Vererbungs- oder Implementationsbeziehung steht. Ein Methodenelement referenziert beispielsweise Klassenelemente, die in der Parameterliste enthalten sind. Wird an einer solchen Referenzliste eine Änderung vorgenommen, so wird diese Modifikation als Referenzlistenersetzung bezeichnet.
- **Ersetze Elementmodifier.** Beispiel: Einer Methodendefinition wird der Modifier `final` hinzugefügt, so dass die Methode nicht in Subklassen redefinierbar ist.
- **Ersetze Elementwert.** Beispiel: Eine Methodenimplementation wird durch eine andere Implementation ersetzt.

- **Ersetze Elementvertrag.** Diese Modifikation bedeutet, dass an den Verträgen einer Operation Änderungen vorgenommen werden.
- **Ersetze Kontext.** Elemente können geschachtelt definiert werden. Der Kontext eines Elementes beschreibt, in welcher Hülle sich ein Element befindet. Eine Klasse kann beispielsweise Operationen enthalten. Der Kontext einer Operation ist die Klasse. Wird eine Methode von einer Klasse in eine andere verschoben, so ändert sich ihr Kontext.

Nach der Durchführung einer Rahmenwerks-Modifikation ist das Rahmenwerk laut Modifikations-Definition in einem syntaktisch korrekten Zustand. Ganz anders kann das für die Klassen einer Anwendung aussehen, die auf dem Rahmenwerk basieren. Die Rahmenwerksmodifikation kann zu einem Struktur- oder Verhaltenskonflikt führen.

> **Begriff: Strukturkonflikt**
> *„Ein Strukturkonflikt zwischen Anwendung und Rahmenwerk tritt auf, wenn die Anwendung auf Typen, Objekte oder Operationen des Rahmenwerkes referenziert, diese jedoch nicht mehr oder nicht mehr in der erwarteten Art und Weise existieren.“* ([Roock 03], Kapitel 6.3)

> **Begriff: Verhaltenskonflikt**
> *„Ein Verhaltenskonflikt zwischen Anwendung und Rahmenwerk tritt auf, wenn die Strukturen von Anwendung und Rahmenwerk kompatibel zueinander sind, zur Laufzeit jedoch Fehler auftreten, die durch falsche Annahmen der Anwendung über die Dynamik des Rahmenwerkes begründet sind.“* ([Roock 03], Kapitel 6.3)

Rahmenwerksmodifikationen können Kompatibilitätsklassen zugeordnet werden, die angeben, ob durch die Modifikation Struktur- oder Verhaltenskonflikte zwischen Anwendungs- und Rahmenwerksklassen auftreten können.

Ohne die Unterstützung von speziellen Migrationswerkzeugen lassen sich Rahmenwerksänderungen in zwei Kompatibilitätsklassen einteilen:

- **sicher-kompatibel**:
  Auf dem Rahmenwerk basierende Anwendungen brauchen nach der Rahmenwerksmodifikation nicht angepasst werden.
- **potenziell-inkompatibel**:
  Rahmenwerksänderungen dieser Gruppe können potenziell zu Struktur- oder Verhaltenskonflikten führen.

In der Arbeit von Stefan Roock werden einerseits theoretisch die Auswirkungen von Modifikationen untersucht, andererseits werden Rahmenwerksänderungen aus der Praxis analysiert. Hintergrund für die praktischen Untersuchungen sind dabei Anforderungen, die in der JWAM-Entwicklung aufgetreten sind. Roock hat anhand der JWAM-Anforderungen immer wieder auftretende Modifikationen identifiziert, die sich jeweils als eine Basismodifikation oder als Kombination von Basismodifikationen beschreiben lassen.

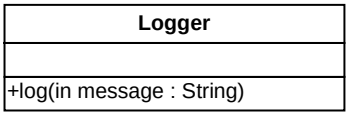
Im Folgenden werden die zehn wichtigsten von Roock in der Praxis vorgefundenen Modifikationen beispielhaft auf das JWAMMicro-Rahmenwerk angewendet, und es werden jeweils die Auswirkungen hinsichtlich der Kompatibilität auf die `AccountTool`-Anwendung analysiert.
Dazu wird die Modifikation benannt, es wird ein konkretes Beispiel am JWAMMicro-Rahmenwerk erläutert und die dafür identifizierte Kompatibilitätsklasse wird angegeben. Um den Bezug zu den oben genannten Basismodifikationen herzustellen, werden zusätzlich die

Namen der Basismodifikationen angegeben, in die sich die Modifikation zerlegen lässt. Für eine genauere Analyse der Zusammenhänge von Basismodifikationen und zusammengesetzten Modifikationen wird auf die Arbeit von Roock verwiesen (siehe [Roock 03], Kapitel 9.3). Die in den folgenden Beispielen genannten Kompatibilitätsklassen gelten jeweils nur für das gegebene Beispiel und nicht für jede denkbare Modifikation der jeweiligen Art. Eine ausführliche Diskussion jedes Modifikationstyps und der möglichen Sonderfälle findet sich in der Arbeit von Roock (siehe [Roock 03], Kapitel 6 und 9).

2.2.2.1. Modifikation: Klasse erzeugen

Beispiel: Der Technologieschicht `de.jwammicro.technology` wird eine neue Klasse `Logger` hinzugefügt, die für das Protokollieren von Log-Einträgen zuständig ist.

JWAMMicro V2.0

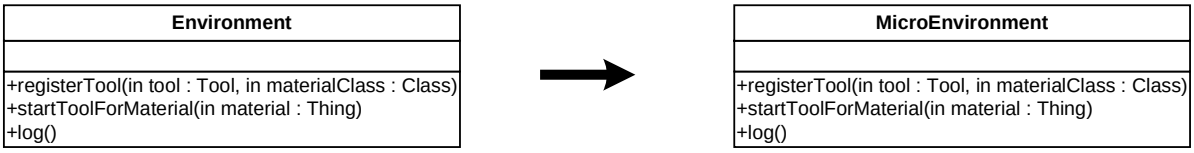


Das Hinzufügen der neuen Klasse hat keine Auswirkungen auf die `AccountTool`-Applikation, die Modifikation ist also kompatibel zur Anwendung.
Kompatibilität: kompatibel
Basismodifikationen: Erzeuge Element

2.2.2.2. Modifikation: Interface oder Klasse umbenennen

Beispiel: Die Klasse `Environment` in der Handhabungsschicht wird umbenannt nach `MicroEnvironment`, um deutlich zu machen, dass es sich um eine in der Funktionalität reduzierte Version eines JWAM-Environments handelt.

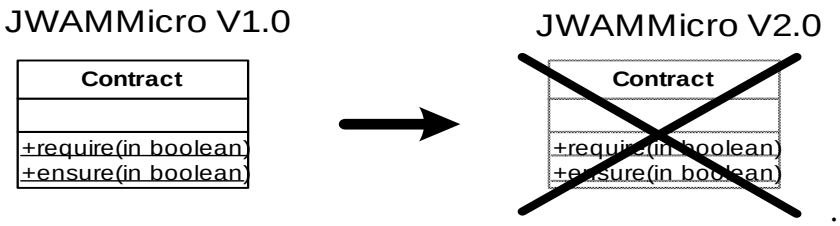
JWAMMicro V1.0 JWAMMicro V2.0



Da in der `Startup`-Klasse der Anwendung das Environment unter dem alten Namen referenziert wird, lässt sich die Anwendung mit der modifizierten Rahmenwerksversion nicht mehr kompilieren. Die Modifikation ist somit inkompatibel zur Anwendung.
Kompatibilität: inkompatibel
Basismodifikationen: Ersetze Elementname

2.2.2.3. Modifikation: Interface oder Klasse löschen

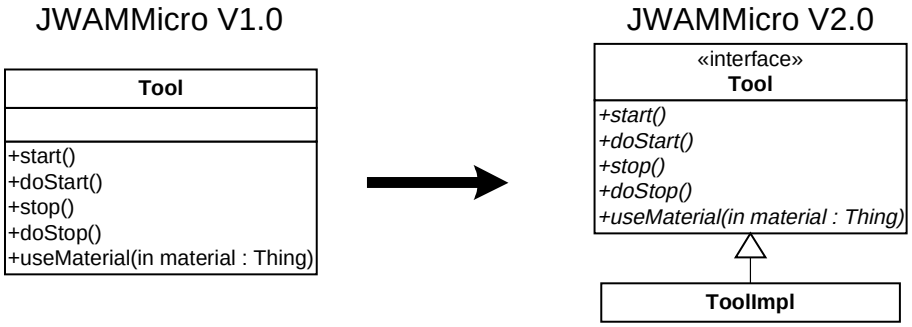
Beispiel: Die `Contract`-Klasse aus der Spracherweiterungsschicht wird gelöscht, da ab der Java-Sprachspezifikation Version 1.4 eine rudimentäre Möglichkeit für die Formulierung von Verträgen über das Schlüsselwort `assert` gegeben ist.



Da die Klasse `Contract` in der `Account`-Klasse verwendet wird, lässt sich die Account-Tool-Anwendung mit der neuen Rahmenwerksversion nicht vollständig kompilieren.
Kompatibilität: inkompatibel
Basismodifikationen: Lösche Element

2.2.2.4. Modifikation: Interface oder Klasse auftrennen

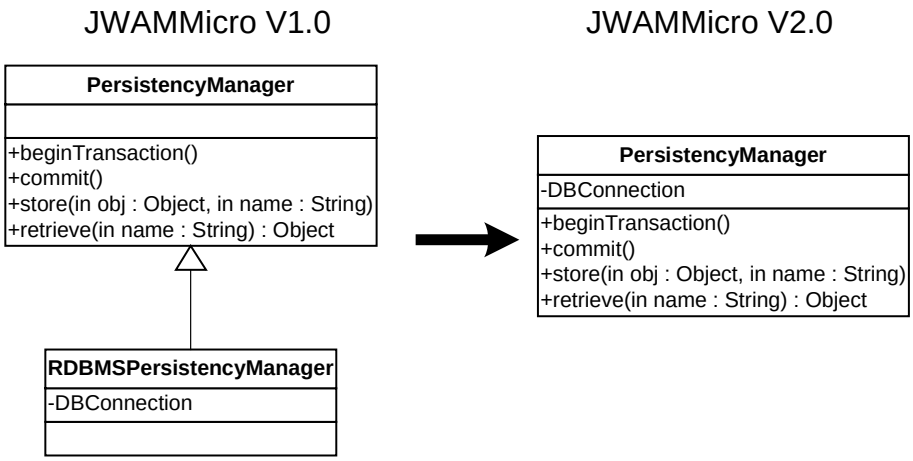
Beispiel: Die Klasse `Tool` wird aufgetrennt in das Interface `Tool` und die implementierende Klasse `ToolImpl`. Genau wie bei der Trennung von `Thing`-Schnittstelle und `ThingImpl`-Standardimplementation wird hier jetzt eine Trennung von Schnittstellenbeschreibung und Implementation vorgenommen.



Je nach Verwendungszusammenhang sollte in der Anwendung die Schnittstelle oder die implementierende Klasse verwendet werden. Bei der blackboxartigen Verwendung von `Tool`-Objekten kann die Schnittstelle verwendet werden. Für die Objekterzeugung und für die Subklassenbildung ist die Verwendung der Implementationsklasse zwingend erforderlich, die im Prozess der Auftrennungsmodifikation in `ThingImpl` umbenannt wurde. Da in den Anwendungsklassen die alte `Tool`-Klasse referenziert wird, in der jetzt lediglich die Schnittstelle beschrieben wird, lässt sich nicht die gesamte Anwendung kompilieren.
Es handelt sich um eine inkompatible Modifikation.
Kompatibilität: inkompatibel
Basismodifikationen: Erzeuge Element, Ersetze Elementname, Ersetze Referenzliste

2.2.2.5. Modifikation: Interface oder Klasse zusammenfassen

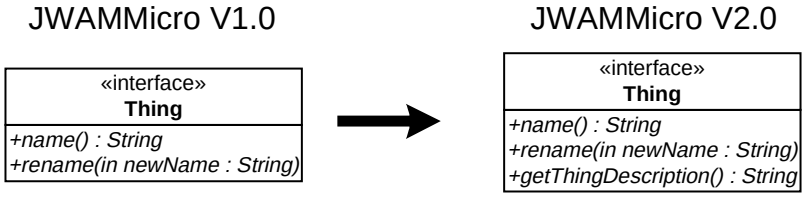
Beispiel: Die Klasse `PersistencyManager`, die grundsätzliches Transaktionshandling bereitstellt, und die `RDBMSPersistencyManager`-Subklasse, die Funktionalität zur Speicherung von Daten in einer relationalen Datenbank hinzufügt, sollen zur Vereinfachung des Rahmenwerkes zusammengefasst werden.



Das Zusammenfassen zweier Klassen zu einer impliziert das Löschen einer Klasse. Dadurch ergibt sich die gleiche Kompatibilität wie beim oben beschriebenen Löschen einer Klasse: die Modifikation ist inkompatibel.
Kompatibilität: inkompatibel
Basismodifikationen: Ersetze Elementname, Lösche Element, Ersetze Referenzliste

2.2.2.6. Modifikation: Operation erzeugen

Beispiel: Wie auch in der JWAM-Rahmenwerksklasse `Thing` soll nun in der Micro-Version jedes `Thing` zusätzlich zu einem Namen eine Klartextbeschreibung seiner selbst geben können.
Das Interface `Thing` wird um eine Methode `getThingDescription` erweitert.

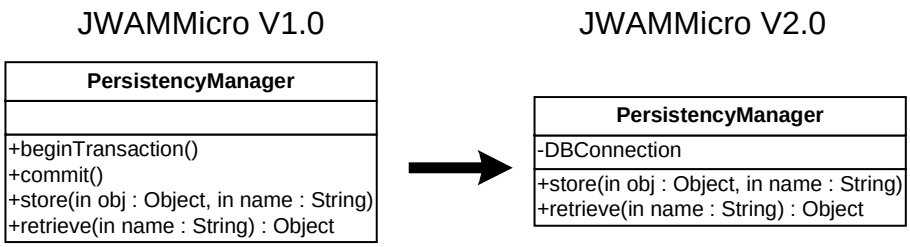


Jede Anwendungsklasse, die die `Thing`-Schnittstelle implementiert, muss die neue Methode implementieren. Ohne diese Implementation ist beispielsweise die Klasse `Account` nicht mehr kompilierbar.
Kompatibilität: inkompatibel
Basismodifikationen: Erzeuge Element

2.2.2.7. Modifikation: Operation löschen

Beispiel: Im Zuge der oben bereits begonnenen Vereinfachung des JWAMMicro-Persistenzmanagements werden die Operationen zur Transaktionsklammerung, `beginTransaction` und `commit`, gelöscht. Die Persistenzoperationen `store` und `retrieve` enthalten eine *Autocommit*[2]-Semantik.

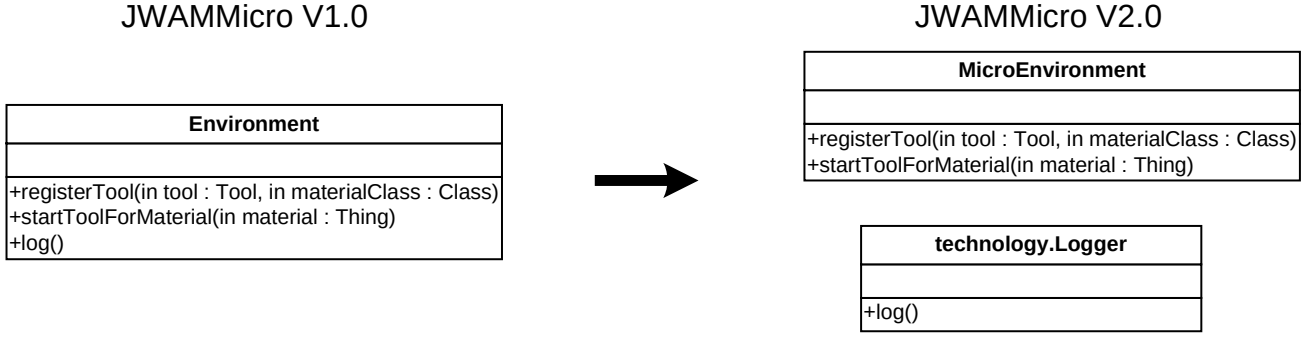
[2] Jede der Persistenzoperationen klammert implizit eine atomare Transaktion.



Die Verwendung der gelöschten Methode in der `AccountTool`-Klasse führt zur Nichtkompilierbarkeit der Anwendung. Die semantische Änderung der Persistenzmethoden wird nicht weiter berücksichtigt.
Kompatibilität: inkompatibel
Basismodifikationen: Lösche Element

2.2.2.8. Modifikation: Operation verschieben

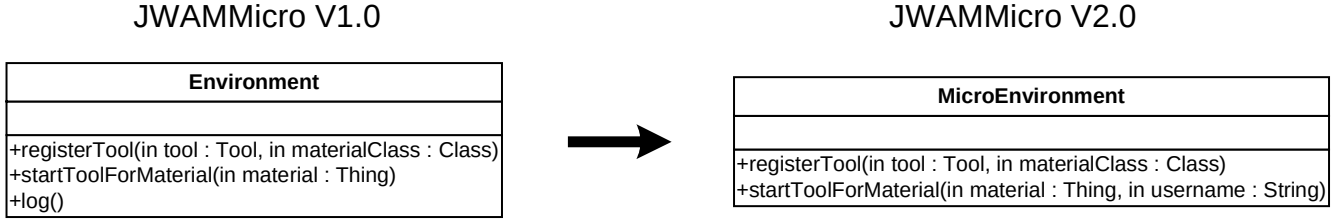
Beispiel: Die `log`-Operation am Environment wird in die neue `Logger`-Klasse in der Technologieschicht verschoben.



Die `Startup`-Klasse der Anwendung lässt sich nicht mehr kompilieren, da die `log`-Methode an einem Objekt der `Environment`-Klasse gerufen wird. Es handelt sich also um eine inkompatible Modifikation.
Kompatibilität: inkompatibel
Basismodifikationen: Ersetze Kontext

2.2.2.9. Modifikation: Parameterliste einer Operation ändern

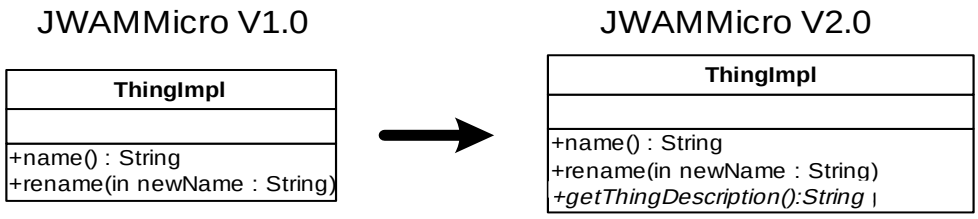
Beispiel: Das Rahmenwerk wird um ein Benutzerkonzept erweitert. Die Methode `startToolForMaterial` wird um einen Parameter erweitert, über den der Benutzer des Werkzeuges beim Starten mitgegeben wird. Das Werkzeug könnte beispielsweise danach entscheiden, welche Aktionen für den Benutzer erlaubt sind.



Der Methodenaufruf in der `Startup`-Klasse muss um einen Benutzernamen erweitert werden. Die `Startup`-Klasse lässt sich so also nicht mehr kompilieren. Die Modifikation ist inkompatibel.
Kompatibilität: inkompatibel
Basismodifikationen: Ersetze Referenzliste, Ersetze Elementwert

2.2.2.10. Modifikation: Modifier einer Klasse oder einer Operation ändern

Beispiel: Die `ThingImpl`-Klassendefinition wird über den Modifier *abstract* als abstrakte Klasse deklariert. Die in der oben beschriebenen Schnittstellenerweiterung von `Thing` (Abschnitt 2.2.2.6) deklarierte Methode `getThingDescription` wird nicht mit einer Standardimplementation versehen.



Die von der nun abstrakten Klasse erbende Materialklasse `Account` muss die abstrakte Methode implementieren und ist somit unmodifiziert nicht übersetzbar.
Kompatibilität: inkompatibel
Basismodifikationen: Ersetze Elementmodifier

2.2.2.11. Zusammenfassung

Nachdem alle beschriebenen Modifikationen durchgeführt worden sind, wird für das daraus resultierende JWAMMicro-Rahmenwerk die Versionsnummer 2 vergeben.

In der folgenden Tabelle sind die Modifikationen und ihre auf die Anwendungsklassen bezogene Kompatibilität zusammengefasst:

Modifikation	im Beispiel erreichte Kompatibilitätsklasse
Klasse erzeugen	kompatibel
Interface oder Klasse umbenennen	inkompatibel
Interface oder Klasse löschen	inkompatibel
Interface oder Klasse auftrennen	inkompatibel
Interface oder Klasse zusammenfassen	inkompatibel
Operation erzeugen	inkompatibel
Operation löschen	inkompatibel
Operation verschieben	inkompatibel
Parameterliste einer Operation ändern	inkompatibel
Modifier einer Klasse oder einer Operation ändern	inkompatibel

Ergebnis

Als Ergebnis der Modifikationen kann festgehalten werden, dass fast alle der Modifikationen zu einem Strukturkonflikt geführt haben und somit inkompatibel zu den Anwendungsklassen sind.

2.3. Migration

Die im letzten Abschnitt beschriebene große Anzahl inkompatibler Modifikationen verdeutlicht den Aufwand, der zur Anpassung einer Anwendung auf eine neue Rahmenwerksversion nötig ist.
Software-Entwicklungsumgebungen unterstützen die Modifizierung komplexer Softwaresysteme, indem nötige Änderungen an allen betroffenen Stellen automatisch vollzogen werden. Die Opensource-Entwicklungsumgebungen Eclipse[3] oder Idea[4] von IntelliJ beispielsweise ermöglichen Modifikationen wie Namensänderungen von Klassen oder Methoden und modifizieren alle Stellen im Code, an denen die betroffene Klasse oder Methode referenziert wird. An allen Stellen im Anwendungscode wird also der alte Name durch den neuen ersetzt. So wird eine ursprünglich potenziell-inkompatible Modifikation über diese Automatisierung zu einer kompatiblen Modifikation.
Vorraussetzung dafür ist aber, dass zum Zeitpunkt der Modifikation der gesamte Applikationscode für das Modifikationswerkzeug sichtbar und veränderbar ist, in der Entwicklungsumgebung also der das Rahmenwerk verwendende Code zugreifbar ist.
Diese Vorraussetzung ist für ein Rahmenwerk im seltensten Fall erfüllt, da es sich bei der Rahmenwerksschnittstelle um ein „published interface“ handelt.
Solange ein Zugriff auf den gesamten Code möglich ist, der gegen ein „public interface“ programmiert worden ist, ist ein Schnittstellen-Refactoring kein größeres Problem. Die Refactoring-Werkzeuge einer IDE können die Änderungen in den Anwendungsklassen unterstützen oder automatisieren.
Dagegen sind Refactorings an einem „published interface“ für die davon abhängigen Anwendungen problematisch, da die Anwendungen nicht im Raum des Rahmenwerkes liegen und somit für die Refactoring-Werkzeuge zum Zeitpunkt des Refactorings nicht sichtbar sind.

Im Mittelpunkt dieser Arbeit steht die Untersuchung der Migrationsunterstützung für Anwendungen basierend auf veröffentlichten Schnittstellen.
Gegenstand der Untersuchungen ist also eine alte Anwendung und eine neue Rahmenwerksversion, die zu der gegen die alte Rahmenwerksschnittstelle programmierten Anwendung nicht kompatibel ist.

2.3.1. Migrationsstrategien verwandter Forschungsfelder

Bevor die Lösungsansätze für den Rahmenwerkskontext vorgestellt werden, wird aufgezeigt, welche Mittel und Strategien der Migrationsunterstützung in verwandten Forschungs- und Anwendungsbereichen existieren. Es wird dargestellt, welche Werkzeuge verfügbar sind und wo die Berührungspunkte zu dieser Arbeit liegen.

2.3.1.1. Change Impact Analysis

Auf dem Gebiet der *Change Impact Analysis* wird erforscht, welche Auswirkungen Änderungen an Softwarebestandteilen auf andere Komponenten haben. Eine kleine Änderung an einer Quelldatei kann zu weitreichenden Auswirkungen in einem Softwaresystem führen. Im Kontext der Rahmenwerksentwicklung ist es interessant zu analysieren, wie "teuer" Änderungen am Rahmenwerk sind, wie groß also der Migrationsaufwand für davon abhängige Softwarebestandteile ist.
Grundsätzlich lassen sich zwei Analysearten unterscheiden: statische Analyse und dynamische Analyse.

[3] Eclipse Tool Platform, http://www.eclipse.org
[4] IntelliJ Idea, http://www.intellij.com/idea

Der statischen Analyse liegen die statischen Abhängigkeiten eines Softwaresystems zugrunde. Die statischen Abhängigkeiten lassen sich über die syntaktische Analyse aller zum System gehörenden Quelldateien ermitteln. Nach der Erstellung der zugehörenden Symboltabelle und der Analyse aller Querbezüge (engl. *cross referencing*) können für jedes Element die Auswirkungen einer Änderung benannt werden. Änderungen an einer Klasse können zu einem Strukturkonflikt in davon abhängigen Klassen führen. Löst man diesen Konflikt durch eine Anpassung der abhängigen Klasse, so können hieraus wieder neue Strukturkonflikte in wiederum davon abhängigen Klassen resultieren.
Eine kleine Änderung kann so zu weitreichenden Auswirkungen im gesamten System führen. Dieser Effekt wird *Ripple-Effekt* genannt (siehe [Yau et al. 85]).
Mit dem Werkzeug *Ripples 2* können die Auswirkungen von Änderungen visualisiert werden (siehe [Rajlich 97]). Wird über das Werkzeug eine Quelldatei verändert, so werden alle Stellen benannt und markiert, die von der Änderung betroffen sind. Das Werkzeug kann zum einen zur Abschätzung des Folgeaufwandes einer Änderung eingesetzt werden, zum anderen kann es den Entwickler beim Anpassungsvorgang leiten.
Eine andere Art der Abhängigkeitsermittlung ist die dynamische Analyse. Dabei wird zur Laufzeit eines Systems protokolliert, welche Abhängigkeiten unter den Komponenten bestehen und wie die Komponenten untereinander kommunizieren. Vahdat und Anderson haben den Begriff des *Transparent Result Caching* (TREC) eingeführt (siehe [Vahdat & Anderson 98]). In einem TREC-System werden die zu untersuchenden Komponenten zur Laufzeit von einem Trace-Modul beobachtet, das die Verwendungs-Beziehungen, das Aufrufverhalten und beteiligte Dateien protokolliert. Übertragen auf den Kontext der rahmenwerksbasierten Anwendungsentwicklung in Java könnte eine modifizierte Virtual Machine die Laufzeitbeziehungen von Klassen aufzeichnen. Eine Analyse dieser Daten würde dann Aufschluss über die Verwendungshäufigkeit von Klassen und Methoden liefern. Ein Vorteil der dynamischen Analyse gegenüber der statischen Analyse ist die Fähigkeit, auch dynamisch über Core Reflection verwendete Klassen und Methoden bei der Analyse zu erfassen.
Die beschriebenen Werkzeuge für die statische und dynamische Analyse sind hilfreich bei der Untersuchung der Auswirkung von Änderungen. Dazu müssen im Moment der Analyse alle beteiligten Komponenten verfügbar sein. Da bei der rahmenwerksbasierten Anwendungsentwicklung diese Vorraussetzung nicht gegeben ist, lassen sich die Techniken und Werkzeuge nicht direkt auf das Problemfeld dieser Arbeit übertragen.

2.3.1.2. Software Configuration Management

Der Bereich des Software Configuration Managements (SCM) umfasst das Verwalten von Änderungen an Softwaresystemen und unterstützt damit den Prozess der Evolution von Software:

> *„SCM, like CM, is defined as the discipline of identifying the configuration of a system at discrete points in time for purposes of systematically controlling changes to this configuration and maintaining the integrity and traceability of this configuration throughout the system life cycle."* ([Bersov et al. 80])

Die Geschichte der SCM-Systeme reicht zurück bis in die 70er Jahre. Marc Rochkind entwikkelte das *Source Code Control System* (SCCS), dass er 1975 veröffentlichte. Später folgten weitere Systeme wie das *Revision Control System* (RCS), PVCS, CVS oder ClearCase.
Moderne SCM-Systeme erfüllen folgende Aufgaben (siehe [Frühauf & Zeller 99]):

- Der gemeinschaftliche Zugriff eines Projektteams auf Quelldateien eines Softwaresystems wird ermöglicht.

- Über entsprechende Werkzeuge lassen sich Änderungen zwischen verschiedenen Versionen von Quelldateien verfolgen.
- Ein kompletter Versionsstand kann markiert werden und ist damit zu einem späteren Zeitpunkt immer wieder aus dem SCM rekonstruierbar.
- Von einem Versionsstand ausgehend können Verzweigungen definiert werden, an denen separat Änderungen vorgenommen werden können. Zu einem späteren Zeitpunkt können der Verzweigungs- und der Hauptstrang wieder zu einem gemeinsamen Strang verschmolzen werden. In der Abbildung 2-3 ist ein typischer Einsatzkontext dargestellt. Nachdem Version 1.1 veröffentlicht wurde, wird ein Parallelzweig erzeugt, in dem Fehlerkorrekturen an der Version 1.1 durchgeführt und veröffentlicht werden können. Gleichzeitig werden aber weitere Revisionen im Hauptzweig erstellt. Durch geeignete Merge-Werkzeuge können die Fehlerkorrekturen zurück in die Hauptstrang-Revisionen einfließen.

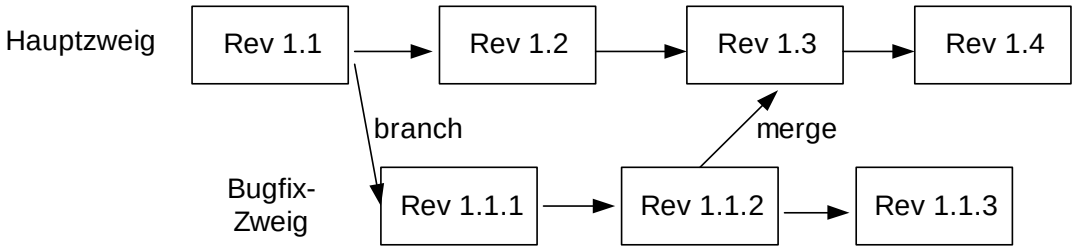


Abbildung 2-3: Beispiel für Branch- und Mergevorgang eines SCM-Systems

Überträgt man den Gedanken der parallelen Weiterentwicklung und des Rückflusses von Änderungen auf die rahmenwerksbasierte Anwendungsentwicklung, so gelangt man zu dem in Abbildung 2-4 dargestellten Szenario.

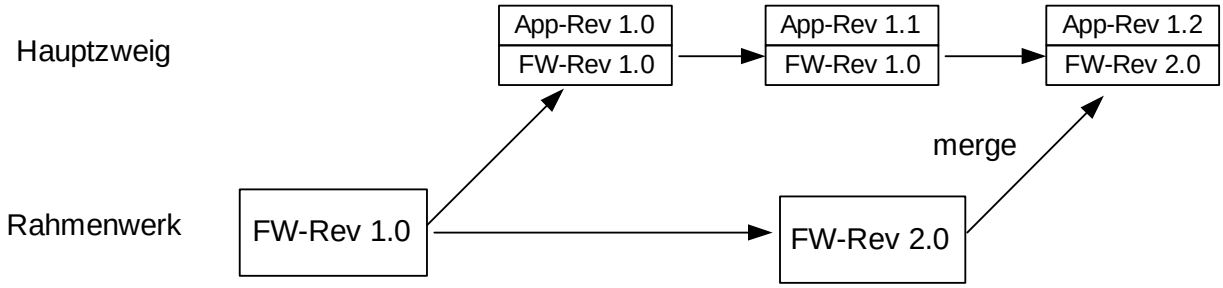


Abbildung 2-4: Mergevorgang übertragen auf rahmenwerksbasierte Anwendung

Die rahmenwerksbasierte Anwendung wird, initial auf der Rahmenwerksrevision 1.0 basierend, im Hauptzweig weiterentwickelt. Parallel dazu wird die Rahmenwerksentwicklung fortgesetzt. Die Änderungen, die zwischen Rahmenwerksversion 1.0 und 2.0 vorgenommen wurden, sollen über die Merge-Werkzeuge in den Hauptzweig eingepflegt werden.

Frühauf und Zeller haben untersucht, wie weit entwickelt heutige SCM-Systeme in den Bereichen Team- und Prozessunterstützung für die Softwareentwicklung sind (siehe [Frühauf & Zeller 99]). Dabei haben sie auch analysiert, wie effektiv das automatische Mergen von Zweigen unterstützt wird. Als Ergebnis haben sie zusammengefasst: „Conflict Resolution: Effectiveness low, Research Potential high“ (siehe [Frühauf & Zeller 99], S. 5, Table 1).
Die Schwäche aller Merge-Werkzeuge der verbreiteten SCM-Systeme ist, dass sie lediglich morphologisches Mergen auf Einzeldateiebene unterstützen. Morphologisches Mergen bedeutet, dass das Merge-Werkzeug versucht, sich gleichende Textblöcke der beiden zu ver-

schmelzenden Dateien zu finden. Sind gleiche Blöcke identifiziert, so wird ermittelt, was um diese unveränderten Textblöcke geschehen ist. Wurden in beiden zu mergenden Revisionen Änderungen an der gleichen Ursprungszeile vorgenommen, so liegt ein Konflikt vor, der vom Merge-Werkzeug zwar angezeigt, aber nicht automatisch behoben werden kann.
Durch das rein morphologische Abgleichen von Textblöcken ist nicht sichergestellt, dass die Ergebnisdatei syntaktisch korrekt ist, da durch das Mergen Textblöcke hintereinander gehängt werden können, die syntaktisch nicht nahtlos zueinander passen.
Wissenschaftliche Ansätze, die darüber hinausgehen, sind das syntaktische und semantische Mergen:
Beim syntaktischen Mergen werden die zu verschmelzenden Quelltextrevisionen syntaktisch analysiert und in abstrakte Syntaxbäume transformiert. Beim Mergen werden dann die Knoten und Kanten der Syntaxbäume gemäß der Sprachregeln verschmolzen. So ist sichergestellt, dass die aus dem Zielbaum generierte Ergebnisdatei syntaktisch wohlgeformt ist.
Das semantische Mergen geht darüber hinaus. Zusätzlich zu der syntaktischen Analyse kann für die zu verschmelzenden Dateien jeweils ein Control Flow Graph (CFG) und ein Program Dependence Graph (PDG) erstellt werden (siehe [Morgenthaler 97]). Beim Mergen wird berücksichtigt, dass im Zielgraph keine unsinnigen oder unerlaubten Kontrollflüsse möglich sind.
Syntaktisches und semantisches Mergen sind Gegenstand der wissenschaftlichen Forschung ohne ausreichende Werkzeugunterstützung (siehe auch [Conradi & Westfechtel 99]).

Die Analyse der heute verfügbaren Merge-Werkzeuge hat gezeigt, dass bereits das automatische morphologische Mergen von SCM-Systemen sehr unzureichend unterstützt wird und das Potential von syntaktischem oder semantischem Mergen so gut wie gar nicht ausgeschöpft wird (siehe [Frühauf & Zeller 99]).
Für die oben geschilderte Idee des Rahmenwerk-Mergens in den Hauptzweig, bei der zusätzlich zum Mergen des Rahmenwerks auch noch die Auswirkungen auf die Anwendungsklassen berücksichtigt werden müssen, steht also bisher keine ausreichende Werkzeugunterstützung zur Verfügung.

Ein weitergehender Ansatz wurde von Van der Hoek formuliert (siehe [Van der Hoek et al. 2001]). Während die verbreiteten SCM-Systeme sich auf die versionierte Verwaltung von Einzeldateien beschränken, können in der von van der Hoek prototypisch erstellten Entwicklungsumgebung *Mae* (Managing Architectural Evolution) zusätzlich komplette Architekturen versioniert werden. In einer Architecture Definition Language (ADL) wird die Architektur eines Softwaresystems beschrieben. Es können Schnittstellen definiert werden, die von Komponenten erfüllt werden. Zu Schnittstellen und Komponenten können Subtypen definiert werden. Über Konnektoren werden Benutztbeziehungen von Komponenten untereinander festgelegt. Eine mit diesen Ausdrucksmitteln beschriebene Architektur kann im Mae-System versioniert verwaltet werden. Im Mae-System wird auf diese Weise die Evolution der Architektur festgehalten. Wie beim Vergleichen von Quelldateirevisionen in gewöhnlichen SCM-Systemen können verschiedene Architekturrevisionen im Mae-System verglichen werden.

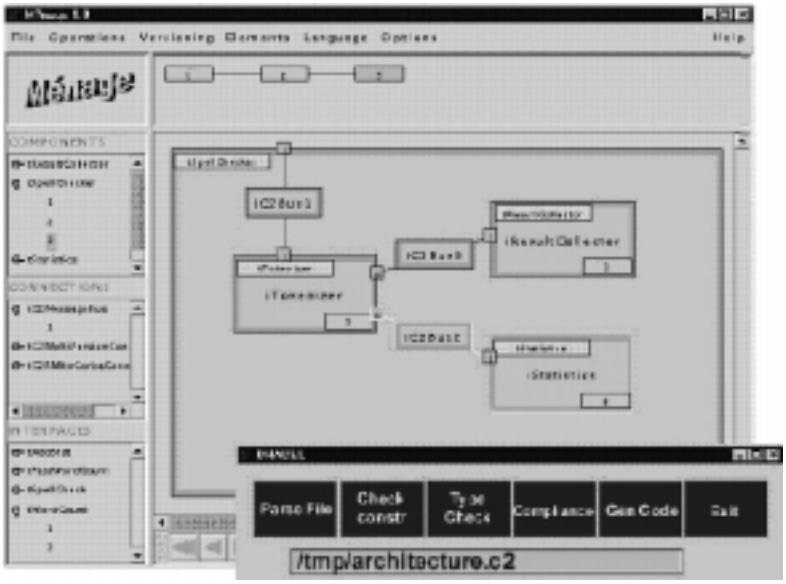


Abbildung 2-5: Mae-Entwicklungsumgebung

Das Mae-Environment-Werkzeug (siehe Abbildung 2-5) visualisiert die Architekturrevisionen und kann Unterschiede zwischen Architekturrevisionen verdeutlichen. Darüber hinaus kann aus einem Architektur-Delta einer alten und einer neueren Revision mit wenigen manuellen Eingriffen ein Patch generiert werden (Automated Change Script Generation, siehe [Van der Hoek et al. 2001, S. 8), mit dem zur Laufzeit ein der alten Architektur entsprechendes System auf die neue Architektur gewandelt wird.
Die Beschreibung der Eigenschaften des Mae-Systems erweckt den Eindruck, die Migrationsproblematik einer rahmenwerksbasierten Anwendung lösen zu können. Für das Rahmenwerk müsste eine Architekturbeschreibung gepflegt werden. Zu beliebigen Zeitpunkten könnten dann über die aus Architektur-Deltas generierten Patches auch die Anwendungen migriert werden.
Die Architekturbeschreibungen, die im Mae-System verwaltet werden, sind aber nicht detailliert genug, um die Abhängigkeiten und Änderungen eines Whitebox-Rahmenwerkes zu berücksichtigen. Das Mae-System arbeitet mit Architekturbeschreibungen auf Ebene von Komponenten, feingranulare Änderungen wie die Umbenennung von Methoden werden vom Mae-System nicht berücksichtigt. Damit ist das System für die Migrationsunterstützung von JWAM-Anwendungen nicht verwendbar.

2.3.1.3. Software-Transformation

Der Forschungsbereich der Software-Transformation befasst sich mit der Migration von Quell- oder Binärcode auf Zielsprachen oder Zielplattformen mittels Werkzeugen. Während bei den SCM-Systemen Quelldateien morphologisch in neue Quelldatei-Revisionen transformiert werden, liegt den Werkzeugen zur Software-Transformation in der Regel eine syntaktische Transformation zu Grunde. Terekhov beschreibt den Prozess einer Sprach-Transformation als ein dreistufiges Vorgehen (Abbildung 2-6, siehe [Terekhov & Verhoef 00]):

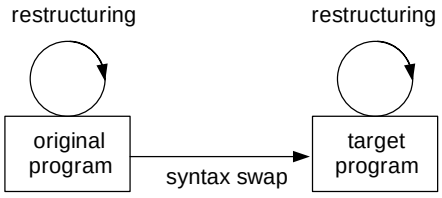


Abbildung 2-6: Sprach-Transformationsprozess aus [Terekhov & Verhoef 98], Figure 2

Zuerst wird der Quellcode in der Quellsprache umstrukturiert, so dass er zwar syntaktisch korrekt bleibt, aber bestimmte Eigenschaften der Zielsprache erfüllt. Beispielsweise würde bei einer COBOL-nach-C – Wandlung eine Startoperation mit dem Namen `main` benannt werden (vgl. [Terekhov & Verhoef 00], S.18).
Im zweiten Schritt wird dann der „syntax swap“, also die syntaktische Wandlung, durchgeführt. Dabei wird im Regelfall beim Parsen der Quelldatei ein abstrakter Syntaxbaum (engl. abstract syntax tree, AST) erzeugt. Der AST wird dann nach im Transformationswerkzeug definierten Transformationsregeln so umgewandelt, dass er der Syntaxstruktur der Zielsprache entspricht. Zum Abschluss der syntaktischen Wandlung wird die textuelle Repräsentation des gewandelten Syntaxbaumes in eine Zieldatei geschrieben.
Im Zuge der syntaktischen Wandlung wurden gegebenenfalls Hilfskonstrukte in der Zielsprache erzeugt, die in der Zielsprache fehlende Eigenarten der Quellsprache simulieren. Diese unschönen und häufig schlecht lesbaren Konstrukte müssen im dritten und abschließenden Schritt im Zuge von Restrukturierungen ersetzt werden.
Die Schwierigkeit bei der Entwicklung von Transformationswerkzeugen liegt in der Erfassung der Transformationsregeln. Eine mögliche Vorgehensweise ist, zunächst manuell Quellcode zu transformieren und dabei Heuristiken über die Transformationsstrategien zu gewinnen. In einem zweiten Schritt werden dann die gewonnenen Erkenntnisse als Regeln in das Transformationswerkzeug eingefügt. Wird mit dem Werkzeug Code transformiert, so muss der automatisiert transformierte Code dem manuell transformierten Code gleichen (siehe [Kontogiannis et al. 98, S. 10]).
Beispiele für existierende Transformationswerkzeuge zeigen, welche Schwierigkeiten mit einer automatisierbaren Transformation verbunden sind:
Die Werkzeuge *f2c* und *p2c* können automatisiert Fortran- oder Pascal-Quellcode in C-Quellcode transformieren. Der erzeugte C-Code ist syntaktisch korrekt und lässt sich direkt übersetzen, aber für einen Menschen les- und wartbar ist der transformierte Code nicht mehr (siehe [Feldman et al. 93]).
Wegen der vielen möglichen Probleme und Schwierigkeiten bei der Software-Transformation hat Terekhov als ein Ergebnis seiner Untersuchungen festgehalten: “*Easy conversion is an oxymoron.*“ (siehe [Terekhov & Verhoef 00]).

Die Berührungspunkte zu dieser Arbeit liegen darin, dass auch bei der Migration einer rahmenwerksbasierten Anwendung Quellcode von einer Quell- auf eine Zielplattform übertragen werden muss. Während aber bei der oben beschriebenen Software-Transformation einzelner Quelldateien jede einzelne Datei nach statisch im Werkzeug kodierten Regeln transformiert werden kann, müssen bei der Anwendungsmigration dateiübergreifende Abhängigkeiten berücksichtigt werden. Die Transformation einer Anwendungsklasse kann nicht allein aufgrund der statischen Struktur des zugehörigen AST durchgeführt werden, da Namensräume und Vererbungsbeziehungen, also Abhängigkeiten zu anderen Klassen, berücksichtigt werden müssen.

Überhaupt erscheint es aufwändig, ein Anwendungs-Transformationswerkzeug zu erstellen, da bei jeder Rahmenwerksänderung entsprechende Transformationsregeln in das Werkzeug eingepflegt werden müssen.
Die bisherigen Erkenntnisse im Bereich der Software Transformation reichen also nicht aus, das Problem der rahmenwerksbasierten Anwendungsmigration zu lösen.

2.3.1.4. Migration von Legacy-Systemen

In vielen großen Organisationen existieren veraltete Informationssysteme. Die Applikationen sind in veralteten Programmiersprachen geschrieben und die Daten werden in veralteten Datenbanksystemen gehalten. Als Legacy-Systeme werden alte Informationssysteme bezeichnet, die sich aufgrund ihrer Architektur evolutionären Veränderungen widersetzen (siehe [Brodie & Stonebraker 95]). Insbesondere fehlende Kapselung der Systembestandteile führt zu monolithischen und kaum wartbaren Systemen. Irgendwann ist der Zeitpunkt erreicht, an dem die Wartungskosten so hoch sind, dass eine Migration auf moderne Programmiersprachen und moderne Datenbank-Management-Systeme notwendig wird.
Bei der Migration von Legacy-Systemen müssen verschiedene Aspekte berücksichtigt werden. Applikationscode, der in alten Programmiersprachen wie COBOL verfasst ist, muss in moderne Programmiersprachen transformiert werden. Dazu können Software-Transformationswerkzeuge eingesetzt werden, wie im letzten Abschnitt beschrieben. Die Daten müssen aus dem alten Informationssystem in ein modernes Datenbank-Managementsystem übertragen werden. Auch hierfür existieren Werkzeuge, auf die im nächsten Kapitel kurz eingegangen wird.
Da in einem Legacy-System die verschiedensten Produkte im Einsatz sein können, kann kein allgemeingültiger Migrations-Werkzeugsatz benannt werden. Brodie und Stonebraker haben in ihren Untersuchungen festgestellt: *„Although legacy IS re-engineering and migration are frequently discussed, there are few, if any, effective migration methods, tools, or techniques."* (siehe [Brodie & Stonebraker 93], S. 11). Sie haben aber grundsätzliche Strategien aufgezeigt, mit denen eine Migration mit kontrollierbarem Risiko durchführbar ist (siehe [Brodie & Stonebraker 95]).
Für eine Migration lassen sich zwei gegensätzliche Strategien benennen: *Cold Turkey* versus *Chicken Little* (siehe [Brodie & Stonebraker 95]).

- *Cold Turkey* (umgangssprachlich für „abrupt") bedeutet, dass das Altsystem auf einen Schlag durch ein neu entwickeltes System ersetzt wird.
- *Chicken Little* steht für eine sehr vorsichtige, „ängstliche" Strategie. Bei dieser Strategie wird das Altsystem Stück für Stück in ein modernes System transformiert. Das Altsystem wandelt sich in einem fließenden Übergang in das neue System.

Die Erfahrung hat gezeigt, dass der *Cold-Turkey*-Ansatz mit zu vielen Risiken behaftet ist. Altsysteme sind häufig nicht ausreichend dokumentiert. Ein auf Basis der lückenhaften Dokumentation entwickeltes Neusystem birgt die Gefahr, undokumentierte Abhängigkeiten nicht zu berücksichtigen und stellt somit ein hohes Projektrisiko dar. Im Allgemeinen wird deshalb die *Chicken-Little*-Strategie mit ihrem iterativen und inkrementellen Ansatz favorisiert (siehe [Kontogiannis 98] und [Stevens 98]).
Brodie und Stonebraker haben abstrahiert von konkreten Hostsystemen und Programmiersprachen beschrieben, wie die *Chicken-Little* - Strategie umgesetzt werden kann, um eine fließende Transformation des Legacy-Systems zu erreichen.

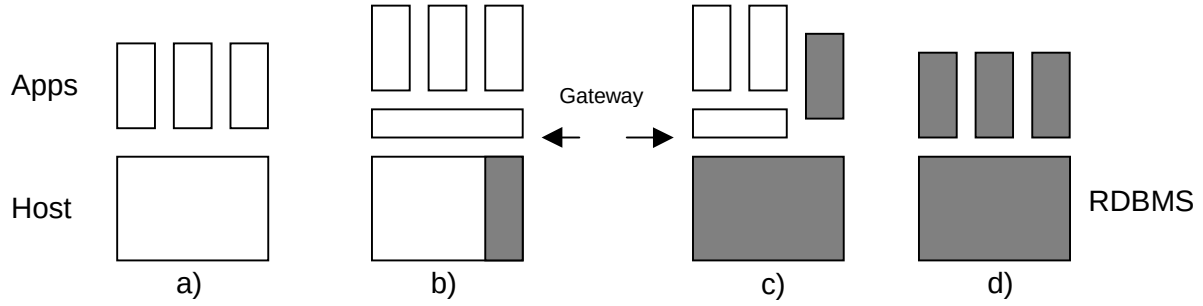


Abbildung 2-7: Inkrementeller Migrationsprozess über Gateways

In der Abbildung 2-7 a) ist das Legacy-System vor der Migration dargestellt. Es gibt mehrere Applikationen, die auf einem Hostsystem basieren, das die Daten und Operationen verwaltet. Nach dem *Cold-Turkey*-Ansatz würde der Host jetzt gegen ein Datenbank-Managementsystem ausgetauscht werden und die Anwendungen müssten schlagartig umstrukturiert werden. Nach dem *Chicken-Little*-Ansatz läuft das Legacy-System weiter und wird in kleinen Schritten ausgetauscht. Möglich wird diese inkrementelle Migration durch die Einführung einer Host-Kapsel, dem Gateway. Das Gateway bietet den Anwendungen die Schnittstelle des Legacy-Systems und leitet Anfragen in der Anfangsphase unverändert an den Host weiter.
In der in Abbildung 2-7 b) dargestellten ersten Migrationsphase wird neben den Host ein modernes Datenbanksystem gestellt und eine kleine Teilmenge der Daten (in der Abbildung dunkel dargestellt) auf die Datenbank transferiert. Das Gateway sorgt dafür, dass Anfragen, die die migrierten Daten betreffen, nicht an den Host gestellt werden, sondern auf die Datenbank umgelenkt werden. So können in kleinen Schritten alle Daten migriert werden. Die neue Datenbank wird immer weiter in das laufende System eingebunden, so dass Probleme sehr früh erkannt werden können.
Sind alle Daten migriert, so kann das Hostsystem abgeschaltet werden. Das Gateway bietet den Applikationen zwar noch immer die Schnittstelle des Hostsystems an, intern werden aber alle Anfragen auf das neue Datenbanksystem umgelenkt. Nun können schrittweise die alten Anwendungen auf eine direkte Verwendung des neuen Datenbank-Managementsystems umgestellt werden. Abbildung 2-7 c) zeigt, dass eine erste Anwendung (in der Abbildung dunkel dargestellt) bereits auf die direkte Datenbankverwendung umgestellt wurde, das Gateway also umgangen wird. Je mehr Anwendungen migriert werden, umso kleiner ist die verwendete Funktionalität des Gateways. Ist endlich die komplette Migration aller Anwendungen und Daten abgeschlossen, so kann das Gateway entfernt werden. (Abbildung 2-7 d)).
Das beschriebene Gateway unterstützt die „Forward Migration Method“ (siehe [Brodie & Stonebraker 93], S. 12). Das Gateway bietet die Schnittstelle des alten Systems an und vermittelt auf ein neues System. Der umgekehrte Ansatz wird als „Reverse Migration Method“ bezeichnet. Das Gateway bietet dann den Anwendungen die Schnittstelle des zukünftigen Systems und vermittelt die Anfragen und Aufrufe an das noch laufende alte Hostsystem. Neue Anwendungen können dadurch gleich gegen die Schnittstelle des neuen Systems programmiert werden, während noch das alte System läuft. Nach dem Abschalten des Altsystems muss nur das Reverse-Gateway entfernt werden, die Anwendungen brauchen bei der Aktivierung des neuen Systems nicht mehr angepasst werden. Auch Mischformen von Reverse- und Forward-Gateways sind denkbar, um den inkrementellen Migrationsprozess zu unterstützen.

Die beschriebenen *Chicken-Little*- und *Cold-Turkey*-Strategien lassen sich auf die Problematik der rahmenwerksbasierten Anwendungsmigration übertragen. Die Anwendung der *Cold-Turkey*-Strategie würde bedeuten, die alte Rahmenwerksversion abrupt durch eine neue Ver-

sion zu ersetzen. Die Anwendung wäre solange nicht lauffähig, bis alle daraus resultierenden Konflikte in den Anwendungsklassen beseitigt worden sind.
Überträgt man die im Legacy-Bereich bewährte *Chicken-Little*-Strategie auf die Rahmenwerksproblematik, so müsste eine Rahmenwerks-Gatewayschicht konstruiert werden, hinter der ein Rahmenwerkswechsel gekapselt werden kann. Im Kapitel 3 wird untersucht, ob der Gateway-Ansatz für die Migration von rahmenwerksbasierten Anwendungen umsetzbar ist.

2.3.1.5. Datenbank-Migration

Bei der Migration von einem Datenbank-System auf ein anderes müssen alle Daten und deren Beziehungen auf das Zielsystem transferiert werden. Wie im letzten Abschnitt beschrieben, birgt eine schlagartige, komplette Migration Risiken, die mit einem inkrementellen Ansatz reduziert werden können. Eine konkrete Ausprägung des *Chicken-Little*-Ansatzes für die Datenbankmigration ist das *Sybil Database Integration and Evolution Environment* (Abbildung 2-8, siehe [King et al. 97]).

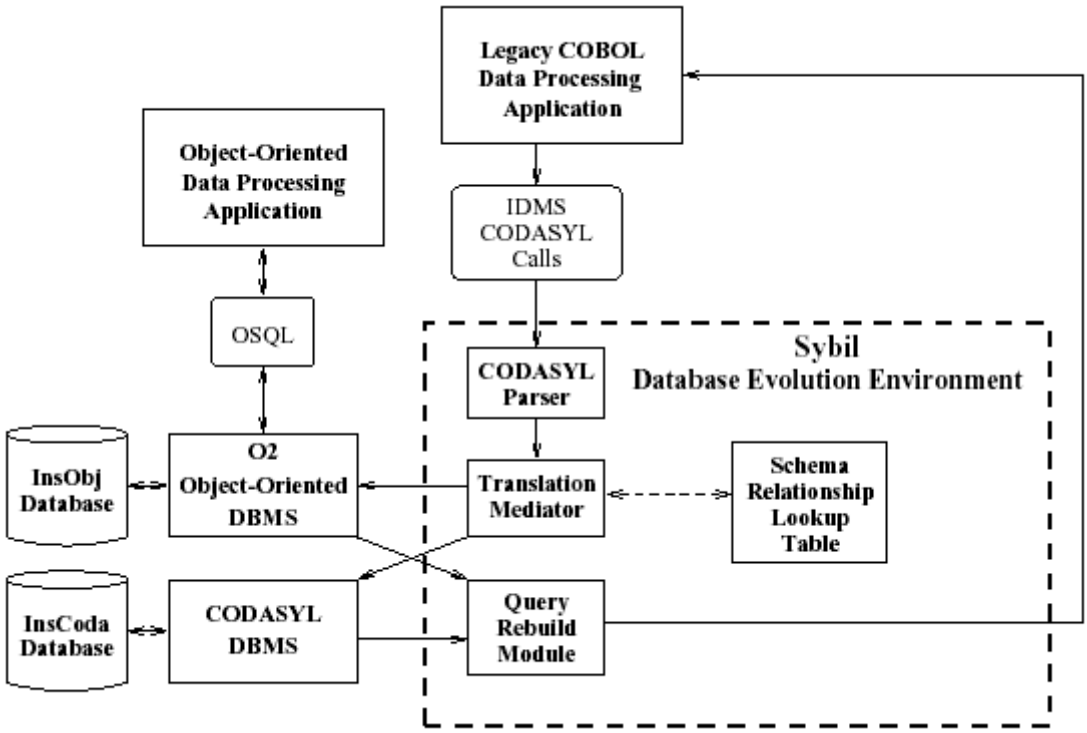


Abbildung 2-8: Abbildung aus [King et al. 97], S. 13

Wie im letzten Kapitel beschrieben, wird der Datenbestand schrittweise auf das neue Datenbanksystem übertragen. Sowohl die alte als auch die neue Datenbank sind während der gesamten Migrationsphase in das operative Gesamtsystem eingebunden. Die Rolle der Gateways nehmen im Sybil-Kontext die Mediatoren ein. Mediatoren kapseln den konkreten Datenbankzugriff und das dazu notwendige Datenbankprotokoll. Die Applikationen müssen während der Migrationsphase nicht angepasst werden, der Mediator vermittelt die Datenbankzugriffe an die richtige Datenbank. Über die Mediatoren wird also eine sanfte, inkrementelle Migration ermöglicht.
Im allgemeingültigen Sybil-Modell müssen große Teile der Mediatoren manuell auf die jeweiligen Quell- und Zieldatenbanken programmiert werden.
Für einige Kombinationen von Datenbanken und Legacy-Systemen gibt es aber auch bereits fertige Produkte, die eine Mediatorrolle einnehmen können. Die *Transparency Software* von

Computer Associates kann beispielsweise Anfragen an IMS-, VMS- oder DB2-Systeme transparent auf das Computer Associates - Datacom DBMS umlenken.
Für die direkte Migration von Daten aus einer Quell- in eine Zieldatenbank gibt es auch verschiedene Werkzeuge, die von den Entwicklern mit geeigneten Extraktions- und Transformationsregeln konfiguriert werden müssen. Von den Datenbank-Herstellern gibt es proprietäre Werkzeuge, mit denen von anderen Datenbanken auf die Datenbank des Herstellers migriert werden kann. Als Beispiel sei die *Oracle Migration Workbench* genannt (siehe [OMW 02]). Ein Beispiel für ein System, das zwischen vielen verschiedenen Datenbanken den direkten Migrationsprozess durchführen kann, ist *Chyfo* von Ispirer Systems (siehe [Chyfo 02]).

Im Datenbankbereich hat sich der Ansatz der inkrementellen Migration bewährt. Es existieren Werkzeuge zur Generierung der Vermittlerschicht und für die Durchführung des eigentlichen Transformationsprozesses. Die Kombination aus Vermittlerschicht und Transformationswerkzeugen hat sich im Datenbankkontext als praktikabel erwiesen und ermöglicht eine sanfte Migration.
Für den Rahmenwerkskontext gibt es keine vergleichbaren Produkte. Die Übertragung der Idee von Vermittlerschicht und Transformationswerkzeug auf die Anwendungsmigration wird in den folgenden Kapiteln untersucht.

2.3.1.6. Ergebnis

In den letzen Abschnitten wurden einige Bereiche untersucht, die sich mit der Evolution oder der Migration von Software und Daten befassen
Die *Change-Impact-Analysis* benennt die Auswirkung von Änderungen und benötigt dazu Zugriff auf alle Bestandteile des zu analysierenden Systems und ist deshalb nicht im Kontext von veröffentlichten Rahmenwerksschnittstellen nutzbar.
Der Bereich des *Software Configuration Management* beschäftigt sich mit der Versionierung von Softwarekomponenten. Die Werkzeuge und Systeme im SCM-Bereich bieten die Möglichkeit, parallele Entwicklungsstränge zu verwalten. Das Zusammenführen der Entwicklungsstränge ist zwar prinzipiell möglich, die Werkzeugunterstützung für diese Aufgabe ist aber sehr unzureichend.
Software-Transformation-Werkzeuge ermöglichen die Migration von Quelltexten auf neue Plattformen. Dabei werden aber nur statische Transformationen jeweils einzelner Quelltexte nach fest definierten Regeln durchgeführt, syntaktische Abhängigkeiten der Quelltexte untereinander werden nicht berücksichtigt.
Für die Migration von Legacy-Systemen und Datenbanken gibt es Werkzeuge und Umgebungen, die mit einigem Aufwand manuell programmiert und konfiguriert werden müssen. Die Strategie einer inkrementellen Migration reduziert signifikant die Risiken einer Migration.

Keines der untersuchten Werkzeuge oder Vorgehen stellt eine befriedigende Lösung für die Problemstellung der rahmenwerksbasierten Anwendungsmigration dar.
Grundsätzlich aber lassen sich zwei Strategien in den verschiedenen Bereichen identifizieren:

- Die direkte Manipulation und Migration von Quelldateien lässt sich über einen Transformationsprozess unter Verwendung eines abstrakten Syntaxbaumes realisieren.
  Software-Transformations-Werkzeuge migrieren Quellcode unter Verwendung von Syntaxbäumen auf neue Plattformen.
  Eine noch in der Erforschung befindliche neue Generation von Merge-Werkzeugen realisiert syntaktisches Mergen über abstrakte Syntaxbäume.
- Die Strategie einer inkrementellen Migration (*Chicken-Little*-Strategie) kann über die Verwendung einer Gateway- oder Adapterschicht umgesetzt werden. Im Bereich der Legacy-Systeme hat sich dieser Ansatz gegenüber des risikobehafteten *Cold-Turkey*-

Vorgehens bewährt. Insbesondere für den Datenbankkontext gibt es fertige Systeme, mit denen eine sanfte inkrementelle Migration realisiert werden kann.

2.3.2. Migrationsstrategien für eine rahmenwerksbasierte Anwendung

Bei der Untersuchung von Migrationsstrategien in verwandten Gebieten konnten grundsätzlich zwei Ansätze identifiziert werden: die direkte Migration und die inkrementelle Migration nach der Generierung einer Adapterschicht. Im Folgenden wird dargestellt, wie die Strategien auf den Kontext einer rahmenwerksbasierten Anwendungsmigration übertragen werden können.

Abbildung 2-9 veranschaulicht die Ausgangssituation. Eine Anwendung wird unter Benutzung des Rahmenwerkes in seiner ersten Version entwickelt. Die Abbildung verdeutlicht das Zusammenpassen von Rahmenwerk und Anwendung.

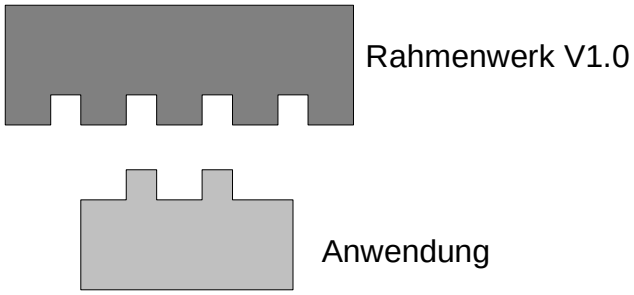


Abbildung 2-9: Rahmenwerk und Anwendung

Im Zuge der Rahmenwerksevolution wird später das Rahmenwerk in seiner zweiten Version veröffentlicht. An der Schnittstelle des Rahmenwerkes wurden Änderungen vorgenommen, die inkompatibel zu der von der Anwendung verwendeten Schnittstelle sind.
Rahmenwerk und Anwendung passen nicht mehr zusammen (Abbildung 2-10).

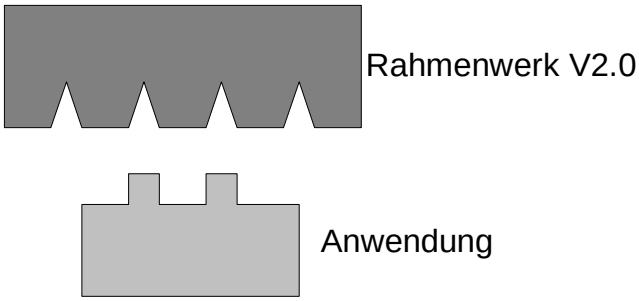


Abbildung 2-10: Neues Rahmenwerk und alte Anwendung

Zwei Ansätze können nun grundsätzlich verfolgt werden:

1) Direkte Anpassung der Anwendung an das geänderte Rahmenwerk.
   Wie in Abbildung 2-11 verdeutlicht, wird die Schnittstelle der Anwendung an die neue Rahmenwerksversion angepasst.
   Die Anwendungsentwickler müssen dafür die Stellen am Rahmenwerk identifizieren, die sich mit der Version 2 geändert haben. Für jede Änderung am Rahmenwerk muss analysiert werden, welche Stellen in Anwendungsklassen von dieser Änderung betroffen sind. An den betroffenen Stellen müssen dann die relevanten Modifikationen durchgeführt wer-

den. Es handelt sich also um eine nachträgliche Analyse des Ripple-Effektes (siehe Kapitel 2.3.1.1).
Am Ende des Migrationsprozesses passen Rahmenwerk und Anwendung wieder zusammen und das Verhalten der Anwendung entspricht dem Verhalten zum Ausgangszeitpunkt mit der Rahmenwerksversion 1.

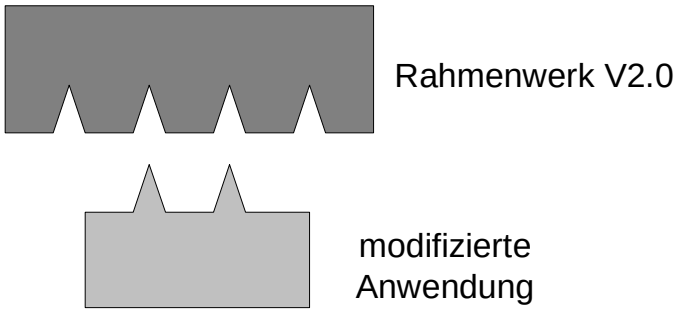


Abbildung 2-11: Neues Rahmenwerk und modifizierte Anwendung

2) Generierung einer Vermittlerschicht für Anwendungen.

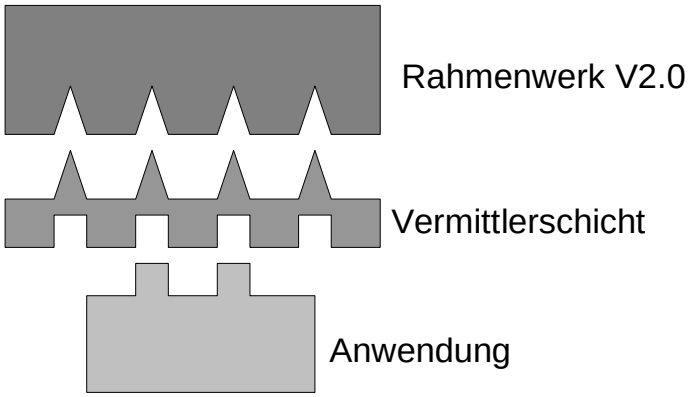


Abbildung 2-12: Anpassung über Vermittlerschicht

Abbildung 2-12 verdeutlicht eine Migrationslösung unter Verwendung einer Zwischenschicht. Zusammen mit der neuen Rahmenwerksversion wird eine Vermittlerschicht ausgeliefert, die auf Anwendungsseite eine Fassade der Rahmenwerksschnittstelle Version 1 darstellt. In der Zwischenschicht werden dann beispielsweise Methodenaufrufe oder Benutzt-Beziehungen auf die neue Schnittstelle der Rahmenwerksversion 2 umgeleitet.
Für alle auf der Rahmenwerksversion 1 basierenden Anwendungen muss also nur einmal eine allgemeingültige Vermittlerschicht generiert und ausgeliefert werden, damit alle alten Anwendungen mit der neuen Rahmenwerksversion verwendet werden können.
Im Laufe der Zeit kann die Anwendung dann Schritt für Schritt wie in der Lösung 1) oben beschrieben an die neue Rahmenwerksschnittstelle angepasst werden, bis die Vermittlerschicht nicht mehr benötigt wird. Rahmenwerk und Anwendung sehen dann aus wie in Abbildung 2-11 skizziert.
Mit der Adapterschicht kann die inkrementelle *Chicken-Little*-Strategie umgesetzt werden, wie in Abschnitt 2.3.1.4 beschrieben.

Die Praktikabilität und Umsetzbarkeit dieser beiden Ansätze wird in den folgenden Kapiteln untersucht.

2.4. Zusammenfassung

In diesem Kapitel wurde die grundlegende Problematik von Rahmenwerksevolution und daraus resultierender Migrationsnotwendigkeit für rahmenwerksbasierte Anwendungen dargestellt.
Der Begriff des Rahmenwerkes wurde definiert und an einem konkreten Rahmenwerk aus der Praxis, dem JWAM-Rahmenwerk, näher beleuchtet.
Für die weiteren Untersuchungen in dieser Arbeit wurde aus dem JWAM-Rahmenwerk eine reduzierte Minimalversion, das JWAMMicro-Rahmenwerk, entwickelt.
Der Begriff der Basismodifikation wurde definiert. Die darauf basierenden zehn wichtigsten von Roock identifizierten Rahmenwerks-Modifikationen wurden in Beispielen auf das JWAMMicro-Rahmenwerk angewendet und der daraus resultierende Migrationsaufwand ermittelt. Die Beispielmodifikationen bilden die Grundlage für die Untersuchungen der folgenden Kapitel.
Es wurde dargestellt, welche Migrationswerkzeuge und Strategien in verwandten Bereichen existieren und abschließend aufgezeigt, wie sich die Ansätze auf den Kontext der Migration von rahmenwerksbasierten Anwendungen übertragen lassen.

3. Rahmenwerks-Adapterschicht zur Migrationsunterstützung

Der Ansatz einer inkrementellen Migration über eine Zwischenschicht hat sich in anderen Bereichen wie der Legacy-System- und Datenbank-Migration bewährt. Im Folgenden wird untersucht, ob sich diese Strategie auf den Rahmenwerkskontext übertragen lässt und ob eine Adapterschicht den Migrationsprozess unterstützen kann.

Die Dienste der Vermittlerschicht sollen für die Anwendung transparent sein, im Idealfall muss der Anwendungscode nicht angepasst werden. Für die Anwendung soll es unerheblich sein, ob das ursprüngliche Rahmenwerk oder die Vermittlerschicht mit neuem Rahmenwerk verwendet wird. Die Fassade, unter der die Vermittlerschicht das Rahmenwerk anbietet, muss die komplette Schnittstelle und das Verhalten der alten Rahmenwerksversion bereitstellen.

Die beiden grundsätzlichen Möglichkeiten für Schnittstellenanpassungen sind von Gamma et al. als Adapter-Entwurfsmuster beschrieben worden (siehe [Gamma et al. 95]).

- Ein Klassenadapter verwendet Vererbung zur Schnittstellenanpassung.
- Ein Objektadapter verwendet Objektkomposition, um eine Schnittstelle an eine andere anzupassen.

Beide Muster beziehen sich auf die Anpassung der Schnittstelle einer einzelnen Klasse. Im Falle der Rahmenwerks-Schnittstellenanpassung handelt es sich aber um die Bereitstellung einer breiten, viele Klassen abdeckenden Schnittstelle.
In den folgenden Abschnitten wird untersucht, ob sich die beiden Ausprägungen des Adaptermusters auf komplette Rahmenwerksschnittstellen ausdehnen lassen.

3.1. Lösung über Klassenadapter

Die statische Struktur des Klassenadapter-Musters (siehe [Gamma et al. 96], S.174) ist dargestellt in Abbildung 3-1:

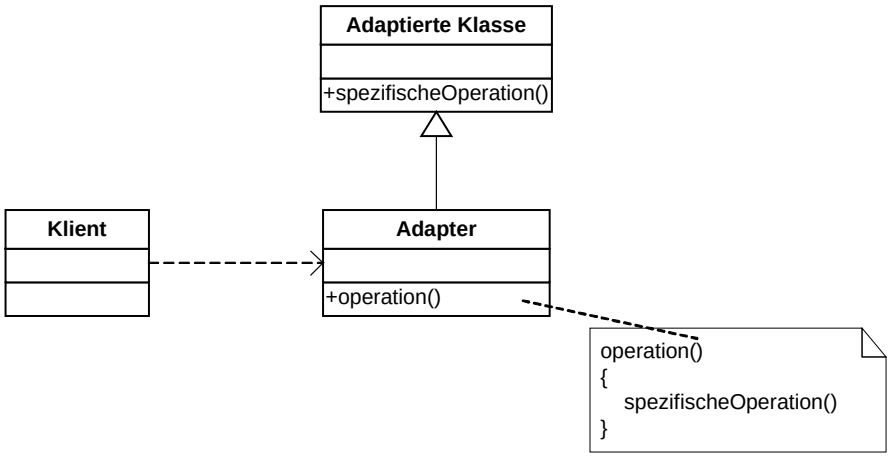


Abbildung 3-1: Klassenadapter-Muster

Der Adapter verfügt über die vom Klienten erwartete Schnittstelle mit der Methode `operation`. Beim Aufruf der Operation ruft der Adapter an seiner Oberklasse die konkret zu verwendende Methode der Oberklassenschnittstelle auf und sorgt somit dafür, dass unter der Adapterschnittstelle die zu adaptierende Klasse verwendet werden kann.

Dehnt man dieses Muster nun auf ein ganzes Rahmenwerk aus, so erhält man eine komplette Vermittlerebene, die zwischen alter und neuer Rahmenwerksversion vermittelt.
Die Anwendung verwendet dann nicht mehr direkt die Interfaces und Klassen des neuen Rahmenwerkes, sondern davon erbende Interfaces und Klassen, die für die Schnittstellenanpassung verantwortlich sind. Die Gesamtheit der Vermittler-Interfaces und -Klassen bildet die Fassade, die der alten Rahmenwerksschnittstelle entspricht.

3.1.1. Grundstruktur der Klassenadapter-Schnittstelle

Zunächst wird untersucht, welchen Randbedingungen ein Rahmenwerk genügen muss, damit die Generierung einer Klassenadapter-Schicht möglich ist.
Folgende Bedingungen müssen bei der Verwendung der Rahmenwerksfassade erfüllt sein:

- Es muss sichergestellt sein, dass Vererbungsbeziehungen, die innerhalb des Rahmenwerkes vorhanden und für die Anwendung sichtbar sind, auch bei der Verwendung der Adapterschicht weiterhin gültig sind.
- Die polymorphe Verwendung von Interfaces und Klassen des Rahmenwerkes in Anwendungsklassen muss auch unter Verwendung der Adapterklassen möglich sein.
- Vererbungsbeziehungen von Klassen aus dem Rahmenwerk und der Anwendung müssen auch bei der Verwendung der Adapterklassen gültig sein. Überschriebene Methoden in Anwendungsklassen müssen vom adaptierten Rahmenwerk und den Adapterklassen berücksichtigt werden.

Das von Gamma et al. beschriebene Adaptermuster bezieht sich nur auf die Adaptierung einer einzelnen Klasse. Die Berücksichtigung von Vererbungsbeziehungen bei der Adaptierung mehrerer Klassen erzeugt eine Reihe von Problemen, die im Folgenden erläutert werden.
Ein erster Versuch einer Rahmenwerks-Schnittstellenstruktur könnte aussehen, wie in Abbildung 3-2 dargestellt:

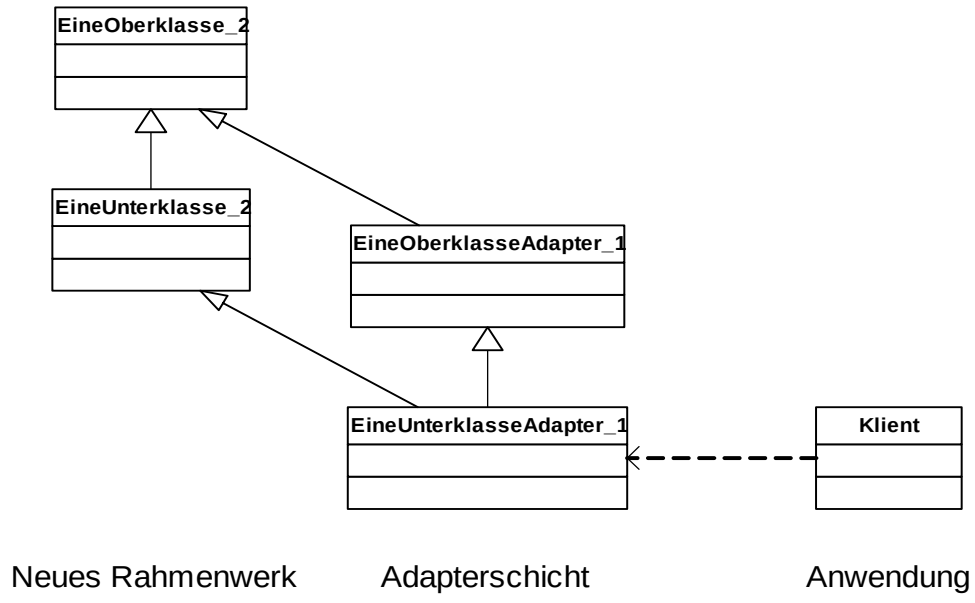


Abbildung 3-2: Rahmenwerksadapter mit Mehrfachvererbung

Intuitiv stellt die in Abbildung 3-2 aufgezeigte Struktur einen Rahmenwerksadapter dar, in dem die oben geforderte Bedingung nach Gültigkeit der alten Vererbungsbeziehungen erfüllt ist. Zur einfachen Unterscheidung von Rahmenwerksversionen ist den Klassennamen ein

Versions-Postfix nachgestellt. Die Adapterklassen, die die alte Rahmenwerksschnittstelle erfüllen müssen, haben das Postfix „_1“, die zu adaptierende neue Rahmenwerksversion das Postfix „_2“. Jede Adapterklasse beerbt gemäß des oben beschriebenen Klassenadaptermusters die zu adaptierende neue Rahmenwerksklasse und bietet in der Schnittstelle die Signaturen der ursprünglichen Rahmenwerksklasse an. Zusätzlich sind die Vererbungsbeziehungen der alten Rahmenwerksversion in den Vererbungsbeziehungen der Adapterklassen nachempfunden.
Diese Lösung ist in Programmiersprachen wie Java, in denen Implementations-Mehrfachvererbung unzulässig ist, nicht realisierbar. Die Klasse `EineUnterklasseAdapter_1` erbt sowohl von der zu adaptierenden Rahmenwerksklasse `EineUnterklasse_2`, als auch von der Klasse `EineOberklasseAdapter_1`. Für das JWAMMicro-Rahmenwerk ist diese Lösung also nicht möglich.
Das Problem kann durch die Verwendung von Interface-Vererbung statt Implementationsvererbung auf Seiten der Adapterklassen umgangen werden. Neben die Implementationsvererbung aus den neuen Rahmenwerksklassen wird eine komplette Vererbungsstruktur aus Adapterklassen-Interfaces gestellt, die die Vererbungsbeziehungen der ursprünglichen Rahmenwerksversion nachbilden.

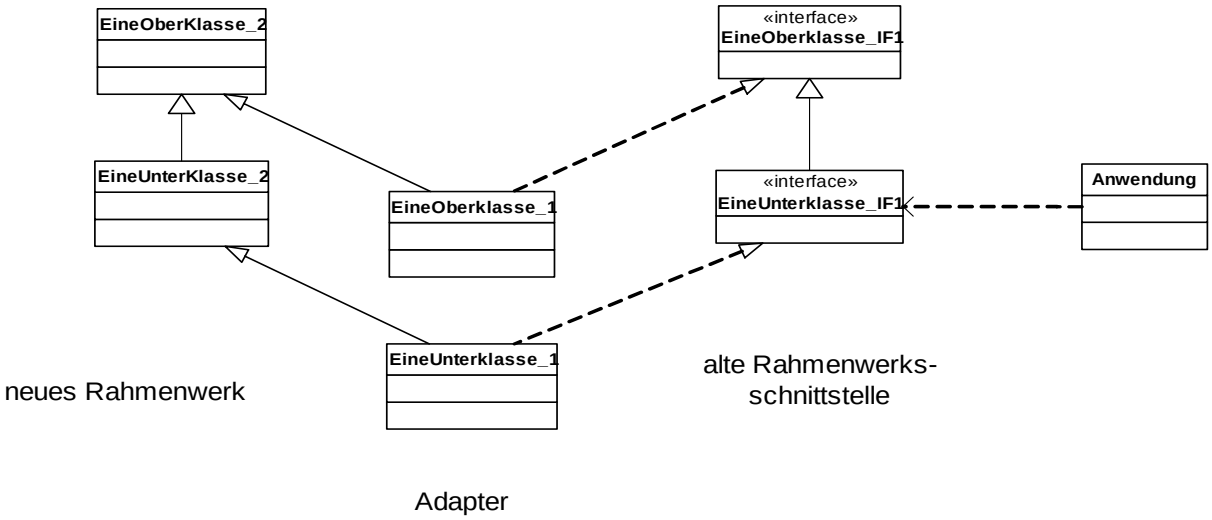


Abbildung 3-3: Rahmenwerksadapter mit Interfacevererbung

In Abbildung 3-3 ist dargestellt, wie die Struktur aus neuen Rahmenwerksklassen, Adapterklassen und der alten Rahmenwerksschnittstelle aussieht. Adapterklassen erben von den zu adaptierenden Rahmenwerksklassen und implementieren die Schnittstellen des alten Rahmenwerkes. Die Schnittstellen sind gekennzeichnet durch das Suffix „_IF“, gefolgt von der Versionsnummer. Die Anwendung ist ausschließlich gegen die Schnittstellen des alten Rahmenwerkes programmiert.
Das Problem der Mehrfachvererbung ist damit gelöst. Adapterklassen sind über Implementationsvererbung in die Klassenhierarchie der neuen Rahmenwerksversion eingebunden, über die Interface-Vererbung erfüllen sie die Schnittstellen und Vererbungsbeziehungen des alten Rahmenwerks.

Diese Lösung hat aber auch zwei unangenehme Konsequenzen:

- Zu jeder Klasse des Rahmenwerkes muss explizit ein Interface definiert worden sein. Diese Interfaces müssen die Vererbungsbeziehungen der Klassen widerspiegeln.
- Bei der Anwendungsprogrammierung muss als Richtlinie beachtet werden, Deklarationen und Signaturen nur unter Verwendung der Rahmenwerks-Interfaces zu formulieren. Kon-

krete Rahmenwerks-Implementationsklassen dürfen nur noch bei der Erzeugung von Objekten Verwendung finden. Diese Einschränkung ist notwendig, da die Adapterklassen nicht voneinander erben. Im obigen Beispiel ist `EineUnterklasse_1` keine Subklasse von `EineOberklasse_1` und kann somit nicht polymorph verwendet werden. Dagegen lassen sich Objekte, die den Typ `EineUnterklasse_IF1` erfüllen, polymorph in Kontexten von `EineOberklasse_IF1` verwenden.

3.1.2. Implementation von fachlichen Rahmenwerks-Interfaces

Im vorhergehenden Abschnitt wurde deutlich, dass jede Implementationsklasse über eine die Schnittstelle beschreibende Interface-Klasse verfügen muss, damit eine Adapterschicht generiert werden kann. Die Interface-Klassen sind eine rein technisch bedingte Notwendigkeit. Zusätzlich zu diesen technisch motivierten Schnittstellen gibt es aber auch fachlich motivierte Schnittstellenklassen, die von einer konkreten Implementation abstrahiert bestimmte Aspekte oder Protokolle definieren. In der ersten Version des JWAMMicro-Rahmenwerkes gibt es beispielsweise das Interface `Thing`, das abstrakt den Umgang mit Gegenständen nach dem WAM-Leitbild beschreibt. Rahmenwerksklassen wie die Klasse `ThingImpl`, die das Interface implementieren und von Anwendungsklassen verwendet werden, verhalten sich in der beschriebenen Weise über die Adapter syntaktisch und semantisch korrekt.
Wenn aber eine Anwendungsklasse ein Rahmenwerks-Interface direkt implementiert, ist die syntaktisch und semantisch korrekte Einbindung in das Rahmenwerk nicht mehr gegeben.

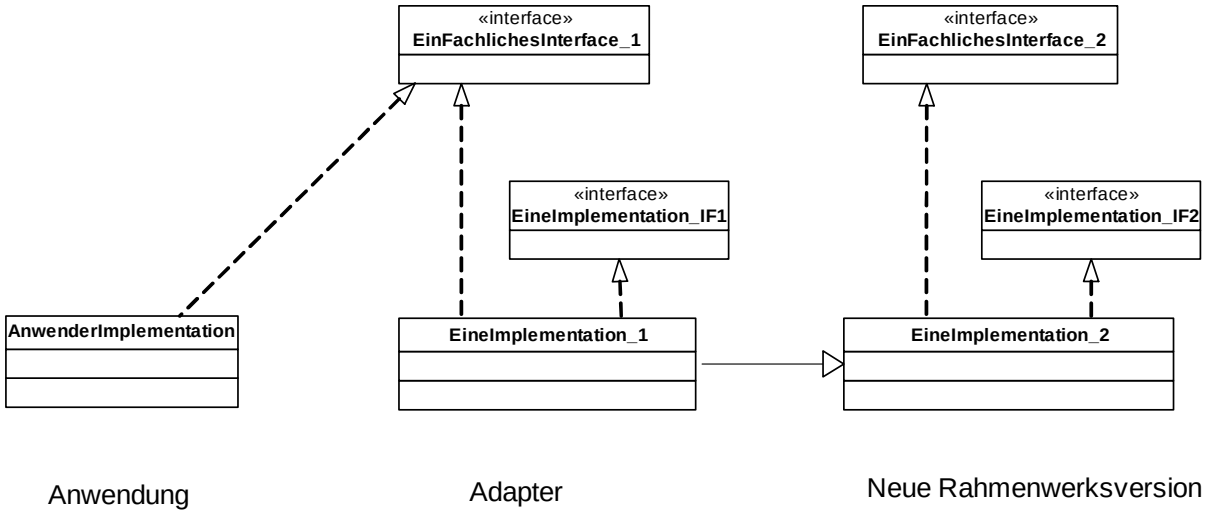


Abbildung 3-4: Klassendiagramm Unzulässige Interfaceimplementation

In Abbildung 3-4 implementiert die Klasse `AnwenderImplementation` das Interface `EinFachlichesInterface_1`. Im Kontext der neuen Rahmenwerksversion lässt sich diese Klasse nicht verwenden, da kein Bezug zum neuen Interface-Typ `EinFachlichesInterface_2` besteht. Die Folgerung hieraus ist also, Anwendungsklassen dürfen Rahmenwerks-Interfaces niemals direkt implementieren. Sollen Anwendungsklassen eine bestimmte Rahmenwerksschnittstelle implementieren, so müssen sie von Standardimplementationsklassen des entsprechenden Interfaces erben, in der die Adaptermechanismen enthalten sind. Für alle fachlich motivierten Interfaces müssen also auch Standardimplementationen vorhanden sein. Würde die Anwendungsklasse in der Abbildung von der Implementationsklasse `EineImplementation_1` erben, so würde sie das fachliche Interface erfüllen und wäre korrekt mit der neuen Rahmenwerksversion verwendbar.

Die Notwendigkeit, von Standardimplementationsklassen zu Erben, ist die größte Einschränkung, die bei der Architektur eines adaptierbaren Rahmenwerkes berücksichtigt werden muss. Da in Java keine Mehrfachvererbung von Implementationsklassen möglich ist, bietet multiple Interface-Vererbung die einzige Möglichkeit, Klassen aus mehreren Typen zu definieren. Jede von der Anwendung genutzte Kombination von Rahmenwerks-Interfaces muss aber nach dem oben beschriebenen Muster eine Standardimplementation im Rahmenwerk besitzen. Der Rahmenwerksentwickler wird dadurch gezwungen, jede nur erdenkliche Kombination von Interfacenutzungen vorherzusehen und entsprechende Kombinations-Implementationsklassen anzubieten.

3.1.3. Auswertung der Klassenadapter-Lösung

In den vorangehenden Abschnitten wurde gezeigt, dass eine Adapterschicht nur generierbar ist, wenn das Rahmenwerk eine bestimmte Struktur besitzt und bei der Entwicklung der Anwendungsklassen einige Regeln eingehalten werden.

Vorraussetzungen für die Rahmenwerksstruktur:

- Für jede Implementationsklasse des Rahmenwerkes muss explizit eine technisch notwendige Interfaceklasse definiert werden. Vererbungsbeziehungen unter den Implementationsklassen müssen sich auch in der zugehörigen Interface-Hierarchie widerspiegeln.
- Fachlich motivierte Interfaces müssen immer mit Standard-Implementationen versehen werden. Für jede sinnvolle Kombination von Interfaces müssen auch entsprechende Standardimplementationsklassen vorhanden sein.

Regeln für die Anwendungskonstruktion:

- In den Anwendungsklassen darf nur gegen die Interface-Klassenhierarchie des Rahmenwerkes programmiert werden. Die Interface-Klassenhierarchie spiegelt die Vererbungshierarchie der Rahmenwerks-Implementationsklassen wider. Wird später anstelle des Rahmenwerkes eine Adapterschicht verwendet, so entsprechen die Adapterklassen nicht mehr der Rahmenwerksvererbungshierarchie, implementieren aber die zugehörigen Schnittstellen entsprechend der Interface-Vererbungshierarchie des Rahmenwerkes.
- Anwendungsklassen dürfen keine Interface-Klassen des Rahmenwerkes direkt implementieren, sondern müssen immer von im Rahmenwerk vorgegebenen Standard-Implementationen erben.

Die Einschränkungen sind so groß und führen zu so komplexen Strukturen, dass die Generierung einer Klassenadapter-Rahmenwerkskapsel nicht praktikabel ist.

3.2. Lösung über Objektadapter

In diesem Abschnitt wird in knapper Form eine Adapterlösung unter Verwendung von Rahmenwerks-Objektadaptern untersucht.

Die Struktur des Objektadaptermusters (siehe [Gamma et al. 96], S.174) ist in der folgenden Abbildung dargestellt:

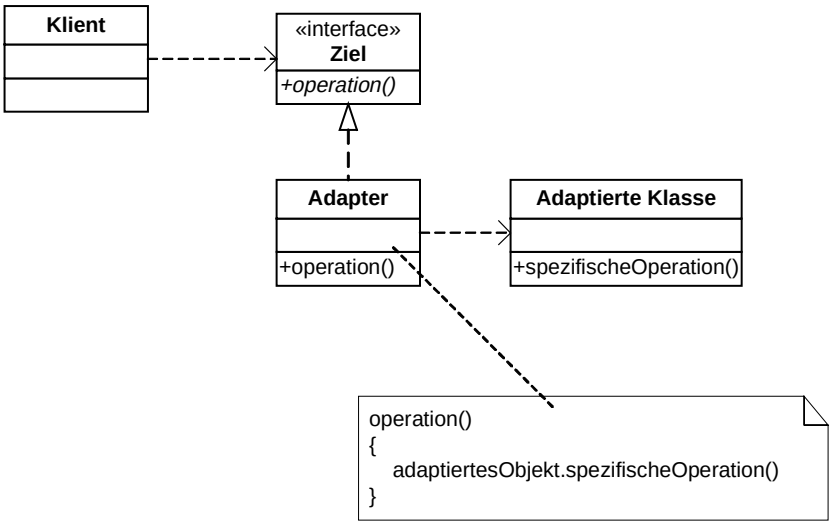


Abbildung 3-5: Klassendiagramm: Objektadapter

Die Klientenklasse erwartet eine bestimmte Schnittstelle an der Zielklasse. Der Adapter sorgt dafür, dass Operationsaufrufe an Objekte der zu adaptierenden Klasse unter der zu adaptierenden Schnittstelle umgesetzt werden. Jedes Adapterobjekt verfügt über ein Objekt der zu adaptierenden Klasse.
Ruft beispielsweise ein Klienten-Objekt die Methode `operation` am Adapter, so delegiert der Adapter an ein Objekt der zu adaptierenden Klasse und ruft dort die Methode `spezifischeOperation`.
Der große Nachteil bei der Verwendung von Objektadaptern ist, dass das Verhalten der zu adaptierenden Klasse nicht überschreibbar ist. Die Aufrufe können immer nur von Seiten des Klienten über den Adapter an die Objekte der adaptierten Klassen kommen, die adaptierten Objekte kennen ihre Adapter nicht. Der Kontrollfluss in den adaptierten Objekten kann im Gegensatz zur Klassenadapterlösung nicht beeinflusst werden. Die Adaption von whiteboxartigen Rahmenwerken ist somit nicht mit Objektadaptern möglich, da das Überschreiben von Rahmenwerksklassen und damit die Beeinflussung des Verhaltens von Rahmenwerksklassen eine wesentliche Eigenschaft von Whitebox-Rahmenwerken ist.
Eine Rahmenwerkskapselung über Objektadapter eignet sich also nur für die Adaption von Blackbox-Rahmenwerken. Damit ist klar, dass nicht für das ganze JWAMMicro-Rahmenwerk eine Objektadapter-Fassade definiert werden kann. Die Whitebox-Bestandteile wie die Klassen `Tool` oder `Thing`, die in ihrem Verhalten von Anwendungsklassen erweitert werden müssen, sind nicht adaptierbar. Lediglich die Blackbox-Bestandteile wie die Klassen `Environment` oder `PersistencyHandler` können adaptiert werden.

Ergebnis:

Die Kapselung eines Rahmenwerkes über eine Objektadapter-Schicht ist für Whitebox-Komponenten eines Rahmenwerkes nicht möglich, da in den Anwendungen redefinierte Methoden nicht in den Kontrollfluss des Rahmenwerkes eingebettet werden.

3.3. Zusammenfassung

In diesem Kapitel wurde untersucht, wie das Muster eines Objekt- oder Klassenadapters ausgedehnt werden kann auf die Kapselung eines ganzen Rahmenwerkes zur Unterstützung eines inkrementellen Migrationsansatzes.
Die Rahmenwerkskapselung über Klassenadapter ist prinzipiell nur möglich, wenn bei der Rahmenwerks- und Anwendungskonstruktion einige Regeln eingehalten werden, die zu komplexen Strukturen führen. Insbesondere die anwendungsseitig nicht erlaubte direkte Implementation von fachlichen Rahmenwerks-Interfaces stellt eine große Einschränkung dar.
Die Verwendung von Objektadaptern ist bei Whitebox-Rahmenwerken nicht möglich, da Klassen des Anwendungsbereiches nicht korrekt in den Kontrollfluss des Rahmenwerkes eingebunden werden können.

Als Ergebnis kann festgehalten werden, dass weder der klassenadapterbasierte noch der objektadapterbasierte Ansatz für die Unterstützung eines inkrementellen Migrationsprozesses nutzbar ist.

4. Vergegenständlichung von Historien im Quelltext zur Migrationsunterstützung

In diesem Kapitel werden die Grundlagen für die direkte Anpassung einer Anwendung an eine neue Rahmenwerksschnittstelle beschrieben.
Damit ein Werkzeug entsprechende Anpassungen durchführen kann, müssen Rahmenwerksänderungen dokumentiert sein, aus denen die benötigten Migrationsschritte ableitbar sind.
Dazu wird eine Möglichkeit der Dokumentation von Rahmenwerksänderungen über Markierungen im Quellcode vorgestellt.
Abschließend wird dargestellt, wie die im Kapitel 2.2.2 aufgezählten Änderungen am JWAMMicro-Rahmenwerk mit den vorgestellten Tags im Quellcode dokumentiert werden können.

4.1. Vergegenständlichung von Historien im Quelltext

Im Zuge der Weiterentwicklung eines Rahmenwerkes werden zwischen zwei Rahmenwerksversionen viele kleine Änderungen und Erweiterungen vorgenommen. Stefan Roock definiert für die Gesamtheit aller Änderungen den Begriff der Historie:

> **Begriff: Historie**
> „*Die geordnete Menge aller Modifikationen zwischen zwei Versionen einer Verwendungseinheit heißt Historie*.“ ([Roock 03], Kapitel 6.2)

Stefan Roock gebraucht dabei den Begriff Verwendungseinheit als Oberbegriff für Rahmenwerke, Klassenbibliotheken und Komponenten, also für softwaretechnische Einheiten, die wiederverwendet werden können.
Für die Migration von Anwendungen auf neue Rahmenwerksversionen ist es notwendig, dass die Historie dokumentiert ist, da nur aus einer Änderungsdokumentation die Migrationsschritte abgeleitet werden können.
Für eine manuelle Migration könnte von den Rahmenwerksentwicklern beispielsweise eine informelle Migrationsanleitung erstellt werden, in denen die wesentlichen Veränderungen der neuen Rahmenwerksversion dargestellt sind und in der die notwendigen Anpassungen für die Anwendungsklassen skizziert werden.
Für eine werkzeuggestützte Migration müssen die Modifikationsinformationen formalisiert dokumentiert sein, damit das Migrationswerkzeug die an den Anwendungsklassen durchzuführenden Migrationsschritte daraus ableiten kann.
Eine Lösung der formalisierten Dokumentation hat Stefan Roock in seiner Dissertation beschrieben. Er greift dafür die Idee der Quellcode-Dokumentations-Markierungen (engl. tags) auf, die ein Teil der Java-Sprachspezifikation sind und als Annotationen in den Quellcode eingefügt werden. Dokumentations-Tags dienen für die Java-Entwickler zum einen als Markierung bestimmter Erläuterungen, die von JavaDoc, einem Dokumentationserzeugungswerkzeug interpretiert und in formatierte Dokumentation umgesetzt werden. Zum anderen gibt es mit dem Deprecated-Tag auch eine Markierung, die Klassen- oder Operationsdefinitionen als veraltet kennzeichnet. Bei der Verwendung einer derart gekennzeichneten Definition wird vom Compiler eine Warnung generiert.
Diese in der Sprache bereits vorhandenen Dokumentations-Tags werden um weitere Tags für die Historiendokumentation erweitert, die von einem Migrationswerkzeug ausgewertet werden können.

Die Rahmenwerksänderungen direkt im Quellcode der Klassen zu verwalten ist ganz im Sinne des von Meyer beschriebenen Single-Source-Prinzips (siehe [Meyer 97]), das fordert, dass möglichst alle eine Klasse betreffenden Informationen konsistent an einer Stelle gehalten werden.

Roock hat für die Änderungsdokumentation vier neue Tags eingeführt, die sich drei Aufgabenbereichen zuordnen lassen:

- **Rückblick in die Vergangenheit über das *Past*-Tag**
  Über den Rückblick in die Vergangenheit können die auf den vorgenommenen Modifikationen beruhenden Unterschiede erkannt werden. Dazu wird das Past-Tag eingeführt. Das Past-Tag annotiert Klassen- und Methodendefinitionen mit der Signatur der ursprünglichen Version. Direkt nach der Veröffentlichung einer Rahmenwerksversion wird die komplette Rahmenwerksschnittstelle durch Erzeugung der Past-Tags aller Klassen- und Methodendefinitionen konserviert. Die Past-Tags können automatisiert mit einem einfachen Werkzeug generiert werden, das Klassen- und Methodendefinitionen in ein Past-Tag im Kommentar kopiert. Werden im Rahmen der Historie Änderungen an den Definitionen vorgenommen, so kann über den Vergleich von aktueller Definition und der im Past-Tag konservierten Ausgangsdefinition eine Modifikation erkannt werden.

- **Nähere Beschreibung von Modifikationen über *Default*- und *Paramdef*-Tag**
  Zur näheren Beschreibung von Modifikationen kann es nötig sein, zusätzlich zu den über das Past-Tag zu erkennenden Unterschieden weitere Modifikations-Informationen zu dokumentieren. Zur Dokumentation von Default-Parametern oder -Implementationen, die in der neuen Version benötigt werden, wird das Paramdef- und Default-Tag eingeführt.

- **Vorausschau in die Zukunft über *Future*-Tag**
  Um auf zukünftige Rahmenwerks-Modifikationen hinzuweisen, wird das Future-Tag eingeführt. Das Future-Tag erweitert die Funktionalität des Java-Deprecated-Tags. Der Rahmenwerksentwickler kann damit nicht nur zukünftig gelöschte Definitionen kennzeichnen, er kann auch auf in der nächsten Version geänderte Definitionen hinweisen.

Im Folgenden wird an die Darstellungen von Roock (siehe [Roock 03], Kapitel 7) angelehnt in knapper Form erläutert, wie sich mit Hilfe dieser Meta- oder Historisierungs-Tags Veränderungen am Rahmenwerk dokumentieren lassen.

4.1.1. Dokumentation von Basismodifikationen

Im Kapitel 2.2.2 wurde der Begriff der Basismodifikation definiert. Basismodifikationen sind die atomaren Modifikationen, die an Verwendungseinheiten vorgenommen werden können. Bevor die Dokumentation der zusammengesetzten JWAMMicro-Modifikationen erläutert wird, wird zunächst dargestellt, mit welchen Tags die Basismodifikationen im Rahmenwerkscode dokumentiert werden können. Diese Darstellung beschränkt sich auf grundlegende Basismodifikationen. Eine komplette formale Analyse und eine Diskussion der möglichen Sonderfälle und ihrer Dokumentation findet sich in der Arbeit von Roock ([Roock 03], Kapitel 7.3).

Basismodifikation: Lösche Element

Über das Setzen des Future-Tags mit dem Wert „#UNDEFINED“ wird das gekennzeichnete Element als in der nächsten Version nicht mehr vorhanden markiert. Genau

wie bei dem Java-Deprecated-Tag wird das markierte Element nicht sofort entfernt. Die Anwendungsentwickler haben bis zur nächsten Version Zeit, entsprechende Anpassungen an den Anwendungsklassen vorzunehmen.

Basismodifikation: Erzeuge Element

Die Erzeugung neuer Elemente kann mit dem Since-Tag gekennzeichnet werden, das als Wert die Nummer der neuen Rahmenwerksversion trägt. Das Since-Tag ist ein Teil des Java-Dokumentationsstandards.
Handelt es sich bei dem neu erzeugten Element um eine abstrakte Operation, so kann mit dem Default-Tag eine Defaultimplementation für die neue Operation definiert werden.

Basismodifikation: Ersetze Elementname

Wird ein Klassen- oder Operationsname geändert, so kann der ursprüngliche Name im Past-Tag konserviert werden, das damit einen vergleichenden Blick in die Vergangenheit ermöglicht.

Basismodifikation: Ersetze Referenzliste

Die Referenzliste eines Elementes ist eine Menge von anderen Elementen, zu denen eine Verwendungsbeziehung besteht. Die Referenzliste eines Klassen-Elementes umfasst die beerbten Klassen und implementierten Interfaces. Die Referenzliste einer Operation umfasst Parameter- und Rückgabetypen.
Werden Modifikationen an einer solchen Referenzliste eines Elementes vorgenommen, so können die Änderungen über ein Past-Tag dokumentiert werden, indem im Past-Tag nicht nur der Name, sondern auch die ursprüngliche Referenzliste dokumentiert wird.
Wird einer Operation ein zusätzlicher Parameter hinzugefügt, so kann über das Paramdef-Tag ein Defaultwert für diesen neuen Parameter dokumentiert werden.

Basismodifikation: Ersetze Elementmodifier

Über das Past-Tag kann die ursprüngliche Konstellation der Element-Modifier dokumentiert werden.

Basismodifikation: Ändere Vertrag

Mit Hilfe des Past-Tags lässt sich die Beschreibung des Vertrages vor der Modifikation dokumentieren. Der ursprüngliche Vertrag muss vor der Veränderung in ein Past-Tag kopiert werden.

Basismodifikation: Ersetze Elementwert

Ändert der Rahmenwerksentwickler den Wert eines Attributes, so kann der ursprüngliche Attributwert in einem Past-Tag dokumentiert werden. Soll die Implementation einer Methode geändert werden, so kann vor der Implementationsmodifikation der Methodencode komplett in ein Past-Tag kopiert werden. Bei längeren Methoden führt diese Art der Modifikationsdokumentation zu sehr großen Tags.

Basismodifikation: Ändere Kontext

Die Änderung des Kontextes eines Elementes lässt sich dokumentieren, indem der ursprüngliche Kontext in einem Past-Tag erfasst wird. Der Kontext einer Methode ist die Klasse, in der sie definiert ist. Der Kontext einer Klasse ist das Package, in dem sie definiert ist.

Mit Hilfe der Tags Past, Since, Default, Paramdef und Future können also alle Basismodifikationen dokumentiert werden. Diese formale Dokumentation von Modifikationen kann von einem Werkzeug erkannt werden, das den Quellcode analysiert.
Ein Migrationswerkzeug kann daraus Informationen gewinnen über die ursprüngliche Struktur des Rahmenwerkes, die vorgenommenen Änderungen und die daraus resultierende, aktuelle Struktur. Mit diesen Informationen kann das Werkzeug geeignete Anpassungen an dem Anwendungscode durchführen, der von der ursprünglichen Struktur abhängig ist.

Im Abschnitt 2.2.2 (Rahmenwerks-Modifikationen) wurden für Modifikationen am Rahmenwerk zwei Kompatibilitätsklassen definiert.

- Eine Änderung ist **sicher-kompatibel**, wenn die vorgenommene Modifikation zu keinem Konflikt mit bestehenden Anwendungen führen kann.
- Eine Änderung ist **potenziell-inkompatibel**, wenn die vorgenommene Modifikation potenziell dazu führen kann, dass verwendende Anwendungsklassen nicht mehr kompilierbar sind oder ein semantisch anderes Verhalten aufweisen können.

Mit einem geeigneten Migrationswerkzeug und der Historiendokumentation im Rahmenwerk ist es nun möglich, einen Teil der potenziell inkompatiblen Rahmenwerksmodifikationen automatisiert in den verwendenden Anwendungsklassen anzupassen.
Ist ein solches Migrationswerkzeug vorhanden, so lassen sich zusätzliche Kompatibilitätsklassen für Modifikationen definieren (siehe [Roock 03], Kapitel 7):

- **Automatisierbare Modifikationen**
  Rahmenwerksmodifikationen sind automatisierbare Modifikationen, wenn die resultierenden Konflikte in den Anwendungsklassen vollautomatisch von einem Migrationswerkzeug aufgelöst werden können.
- **Semi-Automatisierbare Modifikationen**
  Modifikationen am Rahmenwerk, die zu verschiedenen möglichen automatisiert durchführbaren Migrationsschritten in Anwendungsklassen führen, werden als semi-automatisierbar bezeichnet. Das Migrationswerkzeug bietet dem Benutzer die Alternativen zur Auswahl an. Der Werkzeugbenutzer wählt manuell einen Migrationsschritt, der dann automatisiert durchgeführt wird.
- **Aufgeschoben potenziell-inkompatible Modifikationen**
  Eine aufgeschoben potenziell-inkompatible Modifikation ist die Ankündigung einer potenziell-inkompatiblen Modifikation. Die eigentliche Modifikation wird erst in der nächsten Version des Rahmenwerkes umgesetzt. Bis dahin haben die Anwendungsentwickler Zeit, Anpassungen am Anwendungscode vorzunehmen.

4.2. Historisierungsdokumentation am JWAMMicro-Beispiel

Um die Verwendung der Historisierungstags zu illustrieren, werden nun wieder die Modifikationen aufgegriffen, die am ursprünglichen JWAMMicro-Rahmenwerk vorgenommen wurden (siehe Kapitel 2.2.2). Für jede der Modifikationen wird an Quellcodeausschnitten beschrieben, wie die Änderungen im Rahmenwerks-Quellcode unter Verwendung der Historisierungs-Tags dokumentiert werden können. Durch Verwendung eines geeigneten Migrationswerkzeuges können durch Auswertung der Dokumentations-Tags die ursprünglich inkompatiblen Modifikationen zum Teil höhere Kompatibilitätsklassen erreichen. Dabei werden nicht alle möglichen Randfälle betrachtet, eine ausführliche Untersuchung unter Einbeziehung möglicher Randfälle ist von Roock beschrieben worden (siehe [Roock 03]).

4.2.1. Modifikation: Klasse erzeugen

Beispiel: Die Klasse `Logger` wird erzeugt.
Da diese Modifikation sicher-kompatibel ist, ist eigentlich keine die Kompatibilitätsklasse steigernde Dokumentation notwendig. Das Migrationswerkzeug benötigt für sicher-kompatible Änderungen keine Dokumentation, da keine Migrationsschritte durchzuführen sind. Um dennoch eine vollständige Historisierungsdokumentation zu erhalten, kann das Since-Tag eingesetzt werden, das als Wert die neue Rahmenwerks-Versionsnummer erhält.

```
/**
 * @since 2.0
 */
public class Logger
{
  ...
}
```

➔ **Kompatibilität**: sicher kompatibel

4.2.2. Modifikation: Interface oder Klasse umbenennen

Beispiel: Die Klasse `Environment` wird umbenannt nach `MicroEnvironment`.
Die Modifikation wird über das Past-Tag, das als Wert die ursprüngliche Klassen-Signatur enthält, dokumentiert.

```
/**
 * @past public class de.jwammicro.handling:Environment
 */
public class MicroEnvironment
{
  ...
}
```

Ein Migrationswerkzeug kann über den Vergleich von Past-Tag und aktueller Klassendefinition erkennen, dass sich der Klassenname geändert hat. In den zu migrierenden Anwendungsklassen kann diese Änderung vom Werkzeug automatisiert nachvollzogen werden.

➔ **Kompatibilität**: automatisierbar

4.2.3. Modifikation: Interface oder Klasse löschen

Beispiel: Die `Contract`-Klasse wird gelöscht.
Über das Future-Tag wird die Klasse als in der Zukunft nicht mehr vorhanden gekennzeichnet.

```
/**
 * @future #undefined
 */
public class Contract
{

  /**
```

```
   * @future #undefined
   */
  public static void require(boolean b)
  {
    if (!b) throw new RuntimeException();
  }

  /**
   * @future #undefined
   */
  public static void ensure(boolean b)
  {
    if (!b) throw new RuntimeException();
  }
}
```

Durch die Verwendung des Future-Tags wird das Löschen der Klasse und der enthaltenen Methoden angekündigt. Das Migrationswerkzeug kann durch Erkennen des „#UNDEFINED“-Wertes den Anwendungsentwickler auf die zukünftige Löschung der Klasse hinweisen.
Die Entwickler haben bis zum Einsatz der nächsten Rahmenwerksversion Zeit, die Anwendungsklassen umzustellen.

➔ **Kompatibilität**: aufgeschoben potenziell-inkompatibel

4.2.4. Modifikation: Interface oder Klasse auftrennen

Beispiel: Die Klasse `Tool` wird aufgetrennt in die implementierende Klasse `ToolImpl` und das Interface `Tool`.
Beide Klassen der neuen Struktur verweisen über das Past-Tag auf ihre ursprüngliche Ausgangsklasse.

Interface `Tool`:

```
/**
 * @past public class de.jwammicro.handling:Tool
 */
public interface Tool
{

  /**
   * @past public void de.jwammicro.handling:Tool#start()
   */
  public void start();
  ...
}
```

Klasse `ToolImpl`:

```
/**
 * @past public class de.jwammicro.handling:Tool
 */
public class ToolImpl
```

```
{

  /**
   * @past public de.jwammicro.handling:Tool#Tool()
   */
  public ToolImpl()
  {
  }
  ...
}
```

Das Migrationswerkzeug kann an den Past-Tags der Klassen- und Interface-Definitionen erkennen, dass beide die gleiche Ursprungsklasse haben. Bei der werkzeuggestützten Migration können dem Benutzer dann beide Alternativen für jede Verwendung in den Anwendungsklassen angeboten werden. Der Benutzer wählt an den Verwendungsstellen jeweils entweder die Interface- oder die Implementationsklasse aus. Im Quellcode der Anwendungsklasse wird daraufhin eine entsprechende Anpassung automatisiert vorgenommen.

➔ **Kompatibilität**: semi-automatisierbar

4.2.5. Modifikation: Interface oder Klasse zusammenfassen

Beispiel: Die Funktionalität der Klassen `PersistencyManager` und `RDBMSPersistency-Manager` werden zusammengefasst.
Die Definition der neuen gemeinsamen Klasse verweist über zwei Past-Tags auf ihre beiden Ursprungsklassen.

```
/**
 *@past public class de.jwammicro.technology:RDBMSPersistencyManager
      implements PersistencyManager
 *@past public class de.jwammicro.technology:PersistencyManager
 */
public class PersistencyManager
{

  /**
   *
   * @past public void RDBMSPersistencyManager#store(Object o)
   * @past public void PersistencyManager#store(Object o)
   */
  public void store(Object o)
  {
    System.out.println(" Storing to DB");
  }
  ...
}
```

An allen Stellen in Anwendungsklassen, an denen bisher die ursprüngliche Oberklasse `Per-sistencyManager` verwendet wurde, ist nichts zu ändern. An Stellen, an denen die bisherige RDBMS-Subklasse referenziert wurde, kann das Migrationswerkzeug durch Umbenennen der Referenz automatisiert die Klasse an die neue Benennung anpassen.

➔ **Kompatibilität**: automatisierbar

4.2.6. Modifikation: Operation erzeugen

Beispiel: Dem Interface `Thing` wird eine neue Methode `getThingDescription` hinzugefügt. Über das Since-Tag wird die hinzugefügte Methodendefinition markiert, und über das Default-Tag wird eine Standardimplementation dieser neuen Methode definiert.

```
public interface Thing
{
 ...

 /**
   * @since 2.0
   * @default return toString();
   */
  public String getThingDescription();
}
```

Das Migrationswerkzeug kann allen Klassen, die dieses Interface implementieren, automatisiert die neue Methode mit der gegebenen Standardimplementation hinzufügen.

➔ **Kompatibilität**: automatisierbar

4.2.7. Modifikation: Operation löschen

Beispiel: Die `commit`-Operation in der `PersistencyManager`-Klasse soll entfernt werden. Über das Future-Tag wird die Operation als in der Zukunft nicht mehr vorhanden gekennzeichnet.

```
  /**
   * @future #undefined
   */
  public void commit()
  {
    ...
  }
```

Es gilt das Gleiche wie beim Löschen einer Klasse. Durch die Verwendung des Future-Tags wird das Löschen der Methode nicht durchgeführt, sondern nur angekündigt.

➔ **Kompatibilität**: aufgeschoben potenziell-inkompatibel

4.2.8. Modifikation: Operation verschieben

Beispiel: Die `log`-Operation wird verschoben in die `Logger`-Klasse.
Über das Past-Tag der Operation, in der auch der Kontext der Methodendefinition beschrieben ist, wird der ursprüngliche Kontext der Operation dokumentiert.

```
public class Logger
{
  /**
   * @past public static void
         de.jwammicro.handling:Environment#log(String message)
   */
  public static void log(String message)
```

```
  {
    ...
  }
}
```

In diesem Beispiel ist die Migration von Anwendungsklassen, die die `log`-Operation verwenden, automatisierbar. Das Migrationswerkzeug kann über das Past-Tag erkennen, dass die log-Methode ursprünglich in einem anderen Kontext, der Klasse `Environment`, enthalten war. Da es sich um eine Klassenmethode handelt, können alle Referenzen in Anwendungsklassen automatisiert auf die Verwendung der Operation im neuen Kontext geändert werden.

➔ **Kompatibilität**: automatisierbar

4.2.9. Modifikation: Parameterliste einer Operation ändern

Beispiel: Die Operation `startToolForMaterial` in der `Environment`-Klasse wird um einen String-Parameter erweitert, der den aktuellen Benutzer benennt.
Über das Past-Tag wird die ursprüngliche Signatur der Methode dokumentiert.
Über das Paramdef-Tag wird ein Standardwert für den neuen Parameter definiert.

```
/**
 * @past public void
       de.jwammicro.handling:Environment#startToolForMaterial(
         Thing material)
 * @paramdef username="unknown user"
 */
public void startToolForMaterial(Thing material, String username)
{
  ...
}
```

Das Migrationswerkzeug kann über das Past-Tag erkennen, dass sich die Signatur der Operation geändert hat. Automatisch kann das Werkzeug dann an allen Stellen in Anwendungsklassen, in denen die Operation verwendet wird, den zusätzlichen Parameter mit dem Defaultwert in den Methodenaufruf einfügen.

➔ **Kompatibilität**: automatisierbar

4.2.10. Modifikation: Modifier einer Klasse oder einer Operation ändern

Beispiel: Die Klasse `ThingImpl` ist zu einer abstrakten Klasse geworden, da sie die neue Operation `getThingDescription` nicht implementiert.
Im Past-Tag wird die ursprüngliche Modifier-Zusammensetzung der Klassendefinition dokumentiert.

```
/**
 * @past public class de.jwammicro.handling:ThingImpl
       implements Thing
 */
public abstract class ThingImpl implements Thing
{
  ...
}
```

Ein Migrationswerkzeug kann jetzt zwar anhand der abweichenden Modifier-Zusammensetzung von Past-Tag und aktueller Definition feststellen, dass die Klassendefinition modifiziert wurde; eine automatisierte Anpassung von Anwendungsklassen, die die Klasse `ThingImpl` referenzieren, ist aber nicht ohne weiteres möglich. Beispielsweise ist die direkte Erzeugung von `ThingImpl`-Objekten in Anwendungsklassen mit der neuen Rahmenwerksversion nicht mehr möglich. Das Migrationswerkzeug kann aber wenigstens bei der Migration helfen, indem es den Anwendungsentwickler auf diese kritischen Stellen hinweist.

➔ **Kompatibilität**: potenziell inkompatibel

Zum Abschluss wird noch eine Variante erläutert, die das Modifier-Beispiel wenigstens auf die Kompatibilitätsstufe semi-automatisierbar bringt.
Dazu benennen die Rahmenwerksentwickler vor der Modifikation die ursprüngliche `ThingImpl`-Klasse nach `Deprecated_ThingImpl` um und belassen diese Klasse temporär für eine Version im Rahmenwerk zusätzlich zu der modifizierten `Thing_Impl`-Klasse. Wegen der Implementationsbeziehung zum Interface `Thing` muss zusätzlich entweder auch `Thing` gedoppelt werden, oder die zusätzliche Methode `getThingDescription` wird in der `Deprecated_ThingImpl`-Klasse implementiert.

```
/**
 * @past public class de.jwammicro.handling:ThingImpl
       implements Thing
 * @future #undefined
 */
public class Deprecated_ThingImpl implements Deprecated_Thing
{
  ...
}
```

Über das Past-Tag erkennt das Migrationswerkzeug die Ursprungsklasse und kann auf den Strukturkonflikt hinweisen, der durch die Verwendung der `ThingImpl`-Klasse entsteht. Das Werkzeug kann dem Anwendungsentwickler die automatisiert durchführbare Umbenennung nach `Deprecated_ThingImpl` vorschlagen. Wird die `Deprecated_ThingImpl`-Variante gewählt, so kann das Migrationswerkzeug über das Future-Tag die zukünftige Löschung feststellen und den Anwendungsentwickler darauf aufmerksam machen, dass die Klasse in der nächsten Rahmenwerksversion nicht mehr vorhanden sein wird.

Für den Preis der zusätzlichen Klasse erreicht man eine höhere Kompatibilitätsstufe:

➔ **Kompatibilität**: semi-automatisierbar

Das letzte Beispiel verdeutlicht, dass die genannten zehn Modifikationen nicht immer automatisch einer Kompatibilitätsklasse zugeordnet werden können. Inkompatible Modifikationen können über den Umweg der Klassendoppelung und der Future-„#UNDEFINED“-Markierung zumindest auf die Stufe semi-automatisierbar gehoben werden.

4.3. Zusammenfassung

In diesem Kapitel wurde dargestellt, wie Rahmenwerksmodifikationen dokumentiert werden können. Dazu wurden die von Stefan Roock definierten Dokumentations-Tags vorgestellt, die eine Dokumentation direkt im Quellcode ermöglichen. Die Dokumentations-Tags schaffen die Voraussetzungen für einen historisierten Blick auf Klassen- und Methodendefinitionen.
Anhand einfacher Basismodifikationen wurde erläutert, wie prinzipiell die Historisierungstags zur Beschreibung der jeweiligen Basismodifikationen eingesetzt werden können.
Ein großer Vorteil der formalisierten Dokumentation über Tags ist, dass ein Werkzeug über Quellcodeanalyse die Modifikationsinformationen auslesen kann.
Ein Migrationswerkzeug kann mit diesen Informationen automatisiert Migrationsschritte in Anwendungsklassen durchführen. Die daraus ableitbare feinere Abstufung von Kompatibilitätsklassen wurde definiert.
Anschließend wurden die im Kapitel 2 beschriebenen Modifikationen des JWAMMicro-Beispiels aufgegriffen. Anhand der Modifikationen am JWAMMicro-Rahmenwerk wurde illustriert, wie aus Basismodifikationen zusammengesetzte, praxisorientierte Modifikationen im Code dokumentiert werden können und welche Kompatibilitätsklassen bei der Nutzung eines Migrationswerkzeuges erreicht werden könnten.

5. Konstruktion eines Migrationswerkzeuges

Nachdem im letzten Kapitel erläutert wurde, wie Rahmenwerksmodifikationen formalisiert dokumentiert werden können, wird in diesem Kapitel die Konstruktion eines Werkzeuges beschrieben, das aufgrund dieser Dokumentation die Migration unterstützen kann.
Dazu werden zunächst die Anforderungen an ein solches Werkzeug erfasst. Danach wird eine Infrastruktur zur Verwaltung und Analyse von Quellcode vorgestellt, auf der das Migrationswerkzeug basiert. Zum Abschluss wird dargestellt, welche Kompatibilitätsklassen für die Modifikationen unter Nutzung des Werkzeuges erreicht werden.

5.1. Anforderungen

Ein Werkzeug, das den Migrationsprozess von rahmenwerksbasierten Anwendungen unterstützen kann, muss verfügen über ein Wissen über die alte und neue Rahmenwerksversion und die am Rahmenwerk vorgenommenen Modifikationen. Zusätzlich muss das Werkzeug die Abhängigkeiten zwischen der Anwendung und der alten Rahmenwerksversion erkennen können.
Bezogen auf die im Kapitel 2.3.1 beschriebenen Migrationsstrategien verwandter Wissenschaftsgebiete muss das Werkzeug Aspekte verschiedener Bereiche erfüllen:
Aus dem Wissen um die alte und neue Rahmenwerksstruktur müssen auf syntaktischer Ebene Unterschiede erkannt werden, wie im Bereich der SCM-Systeme beschrieben (siehe Kapitel 2.3.1.2).
Diese durch Rahmenwerksmodifikationen hervorgerufenen Unterschiede können zu Konflikten mit bestehenden Anwendungsklassen führen. Die Auswirkungen der Rahmenwerksmodifikationen müssen analysiert werden, wie für den Bereich der Change-Impact-Analysis geschildert (siehe Kapitel 2.3.1.1).
Das Werkzeug muss aus den ermittelten Rahmenwerksmodifikationen Regeln ableiten, mit denen die Anwendungsklassen von der Verwendung des alten auf die Verwendung des neuen Rahmenwerkes transformiert werden können. Das Werkzeug erfüllt damit eine im Bereich der Software-Transformations-Werkzeuge beschriebene Aufgabe (siehe Kapitel 2.3.1.3).

Im Einzelnen lassen sich folgende Aufgaben zur Unterstützung des Migrationsprozesses identifizieren:

1. **Analyse von Rahmenwerkscode und Modifikationsdokumentation**
   Das Werkzeug muss den neuen Rahmenwerks-Quellcode analysieren und die Historisierungs-Tags interpretieren können. Über den Quellcode und die Historisierungs-Tags muss das Werkzeug eine Rahmenwerkssicht entwickeln, in der ein Rückblick auf die vergangene Version, die vollzogenen Modifikationen zur aktuellen Version und eine Vorausschau auf die kommende Rahmenwerksversion enthalten ist.
2. **Analyse von Anwendungsklassen**
   Das Werkzeug muss den Quellcode der zu migrierenden Anwendungsklassen analysieren können. Es muss daraus die Vererbungs- und Benutzt-Beziehungen von Anwendungsklassen untereinander und von Anwendungsklassen zu den Klassen des ursprünglichen Rahmenwerkes erkennen und einordnen können.
3. **Ermitteln notwendiger Migrationsschritte**
   Das Werkzeug verfügt nach den ersten beiden Anforderungen über das Wissen über die ursprüngliche Rahmenwerksversion, die aktuelle Rahmenwerksversion, die durchgeführten Modifikationen und über die Struktur der Anwendung, die auf der ursprünglichen

Rahmenwerksversion basiert. Mit diesem Wissen muss das Werkzeug in der Lage sein, die Stellen in der Anwendung zu identifizieren, an denen Migrationsschritte notwendig sind. Mit dem Wissen über die durchgeführten Modifikationen kann es Vorschläge zur Migration machen.

4. **Durchführung automatisierbarer Migrationsschritte**
   Einige der vom Werkzeug ermittelten Migrationsschritte sind automatisiert durchführbar. Das Werkzeug soll diese Schritte an den Anwendungsklassen automatisiert durchführen können.
5. **Unterstützung für die Migration von Modifikationen anderer Kompatibilitätsklassen**
   Für Migrationsschritte, die nicht vollständig automatisiert durchgeführt werden können, muss das Werkzeug dem mit der Migration betrauten Anwendungsentwickler Hilfestellung leisten können. Je nach Kompatibilitätsklasse der Modifikation soll es dem Anwendungsentwickler automatisiert durchführbare Migrationsalternativen anbieten oder ihm zumindest Hinweise auf manuell zu migrierende Stellen geben.

Diese Aufgaben lassen sich in drei Kategorien unterteilen: die Analyse von Quelltexten, das Ermitteln von Migrationsschritten und die Steuerung des eigentlichen Migrationsprozesses. Zwischen diesen Aufgabenbereichen bestehen Abhängigkeiten. Die Steuerung des Migrationsprozesses basiert auf der Ermittlung der Migrationsschritte. Migrationsschritte wiederum können erst nach der strukturellen Analyse von Rahmenwerks- und Anwendungsquelltexten identifiziert werden.
Das Migrationswerkzeug wurde deshalb in drei Schichten organisiert. Jede Schicht erfüllt genau einen der beschriebenen Aufgabenbereiche. In Abbildung 5-1 ist die Grobstruktur des Werkzeuges dargestellt.

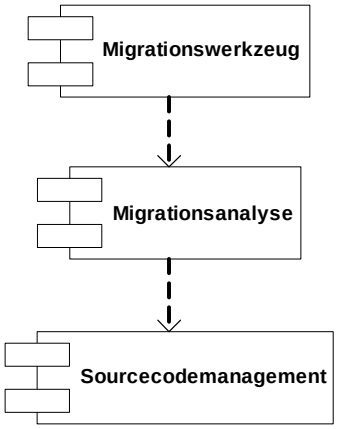


Abbildung 5-1: Grobstruktur des Migrationswerkzeuges

Als untere Schicht des Migrationswerkzeuges wird eine Sourcecodemanagement-Komponente benötigt. Die Sourcecodemanagement-Schicht ist zuständig für die Verwaltung und Analyse des für den Migrationsprozess benötigten Quellcodes, also für die Rahmenwerks- und Anwendungsklassen. Die Sourcecodemanagement-Schicht ist auch verantwortlich für die Sicht auf die vorangegangene und zukünftige Struktur des Rahmenwerkes, die durch die Historisierungs-Tags im Quellcode dokumentiert ist. Damit erfüllt sie die Anforderungen 1 und 2.
Zusätzlich muss das Sourcecodemanagement ermöglichen, Quellcode zu ändern und in die Quelldatei zurück zu schreiben. Damit leistet sie einen Beitrag zur Erfüllung der Anforderungen 4 und 5.

Die Migrationsanalyse-Schicht verwendet das Sourcecodemanagement als Basis. Die Migrationsanalyse-Schicht analysiert die über das Sourcecodemanagement verwaltete Struktur von ursprünglicher, aktueller und zukünftiger Rahmenwerksversion. Auf dieser Grundlage werden Modifikationen erkannt und je nach Kompatibilitätsklasse der Modifikation Hinweise für den Anwendungsentwickler generiert und automatisiert durchführbare Migrationsschritte identifiziert. Damit werden die Anforderungen 3 bis 5 erfüllt.
Gesteuert wird der Migrationsprozess von einem Migrationswerkzeug, das auf den Diensten der unteren Schichten basiert.
Im Folgenden wird die Modellierung und Konstruktion der einzelnen Schichten detailliert dargestellt.

5.2. Die Sourcecodemanagement-Schicht

Anforderungen
Die Sourcecodemanagement-Schicht ist die unterste Schicht im Werkzeugmodell. Sie stellt einen Dienst zum Zugriff auf die syntaktischen Informationen aller Quelldateien bereit.
Der Quellcode des Rahmenwerkes und der Anwendungsklassen muss verwaltet und analysiert werden. Der Zugriff auf Klassen- und Methodendefinitionen muss ermöglicht werden. Die in den Meta-Tags dokumentierten Historisierungsinformationen müssen zugreifbar sein. Und schließlich müssen Änderungen an der Quellcodestruktur durchführbar sein, die dann auch durch Zurückschreiben in die Quelldateien persistent gemacht werden können.

Umsetzung
Damit diese Aufgaben erfüllt werden können, müssen die zu verwaltenden Quelldateien eingelesen und die textuellen Informationen in eine Objektstruktur überführt werden. Dabei handelt es sich um einen sehr komplexen Vorgang, der in mehrere Stufen zerlegt werden kann. Das in Abbildung 5-2 dargestellte Kollaborationsdiagramm zeigt, mit welchen Komponenten und Strukturen die Analyse realisiert werden kann.

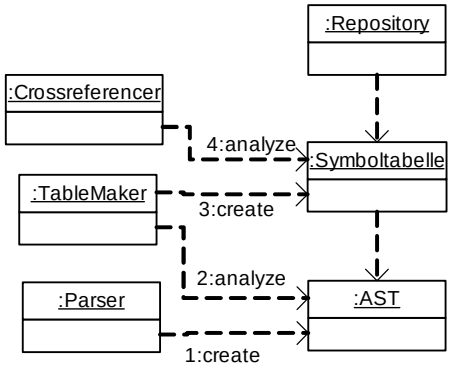


Abbildung 5-2: Kollaborationsdiagramm der Sourcecodemanagement-Schicht

In einem ersten Schritt wird der Zeichenstrom der Quelldateien von einem Parser analysiert und in einen abstrakten Syntaxbaum transformiert.
Auf dem abstrakten Syntaxbaum basierend wird eine Symboltabelle erstellt, die alle Symbole und Namensräume enthält.
Basierend auf den Symboltabellen-Informationen und dem abstrakten Syntaxbaum können dann alle Verwendungsbezüge analysiert werden (Crossreferencing).

Als zentrale Verwaltungsinstanz der Sourcecodemanagement-Schicht fungiert das Repository, das die Analyseprozesse und konkreten Zugriffsmechanismen kapselt und die syntaktischen Informationen des Gesamtsystems bereitstellt.
Die konkrete Realisierung dieses Entwurfs wird in den folgenden Abschnitten dargestellt.

5.2.1. Parsen der Quelldateien

In dem ersten Analyseschritt müssen die Quelldateien von einem Parser analysiert werden. Beim Parsen geht es darum, aus dem Zeichenstrom einer Quelldatei programmiersprachliche Konstrukte zu erkennen. Da es sehr aufwändig ist, einen Parser für komplexere Sprachen komplett von Hand zu programmieren, gibt es Werkzeuge, die eine Parsergenerierung unterstützen. Parser-Generatoren können für beliebige Sprachen Parser erzeugen. Zur Erzeugung eines Parsers muss der Generator mit lexikalischen Regeln und der Grammatik der zu parsenden Sprache konfiguriert werden.
Zwei Parser-Generatoren sind im Java-Kontext verbreitet: JavaCC und ANTLR.
JavaCC ist ein von Sun-Mitarbeitern entwickelter Parser-Generator (siehe [JavaCC 02]), der zwar frei verwendbar, aber nur als Closed-Source-Software verfügbar ist.
Um das Migrationswerkzeug zu realisieren wurde deshalb der Open-Source-Parsergenerator ANTLR (Abkürzung für Another Tool for Language Recognition) von Terence Parr (siehe [ANTLR 02]) gewählt.
Für die Sprache Java gibt es bereits eine ANTLR-konforme Grammatik, mit der ein Java-Parser generiert werden kann. Der ANTLR-generierte Java-Parser kann die programmiersprachlichen Konstrukte aus dem Zeichenstrom einer Quelldatei erkennen und in einen abstrakten Syntaxbaum überführen. Der Syntaxbaum repräsentiert die syntaktische Struktur der Java-Quelldatei.
Eine Besonderheit bei der Parser-Generierung im Kontext des Migrationswerkzeuges ist aber zu beachten: Der beim Parsen generierte Syntaxbaum enthält normalerweise nur syntaktisch relevante Knoten. Kommentare werden nicht in den Syntaxbaum aufgenommen, da sie auf syntaktischer Ebene bedeutungslos sind. Für das Migrationswerkzeug sind die Kommentare aber unentbehrlich, da sie die im Kapitel 4 beschriebenen Historieninformationen enthalten. Die Java-Grammatik musste deshalb für die Parser-Generierung um Kommentar-Strukturinformationen erweitert werden, die zu einer Aufnahme von Kommentaren in den Syntaxbaum führen.

Das Ergebnis des so realisierten Parser-Prozesses ist ein abstrakter Syntaxbaum, in dem für alle programmiersprachlichen Konstrukte auch die zugehörigen Kommentare enthalten sind.

5.2.2. Erstellung einer Symboltabelle

Nachdem die syntaktische Struktur der Quelldateien in einen Syntaxbaum übertragen wurde, können daraus sämtliche Symbolinformationen und Namensräume gewonnen werden. Dieser Vorgang ist ebenfalls ein komplexer Prozess, für den es bereits fertige Lösungen zur freien Verwendung gibt.
Für das Migrationswerkzeug wird ein Teil des Transmogrify-Paketes verwendet (siehe [Transmogrify 01]). Transmogrify ist ein in Java entwickeltes Refactoring-Rahmenwerk, das in verschiedene IDEs eingebettet werden kann und eine Reihe von Refactoring-Maßnahmen unterstützt. Basis des Transmogrify Refactoring-Frameworks ist ein analytischer Teil, Transmogrify-Analytics-Core oder auch *Thaumaturge* genannt. Dieser analytische Teil benötigt

einen von einem ANTLR-Parser generierten abstrakten Syntaxbaum und erstellt auf dieser Basis eine Symboltabelle.
Die dafür benötigten Kernklassen sind in der folgenden Darstellung abgebildet:

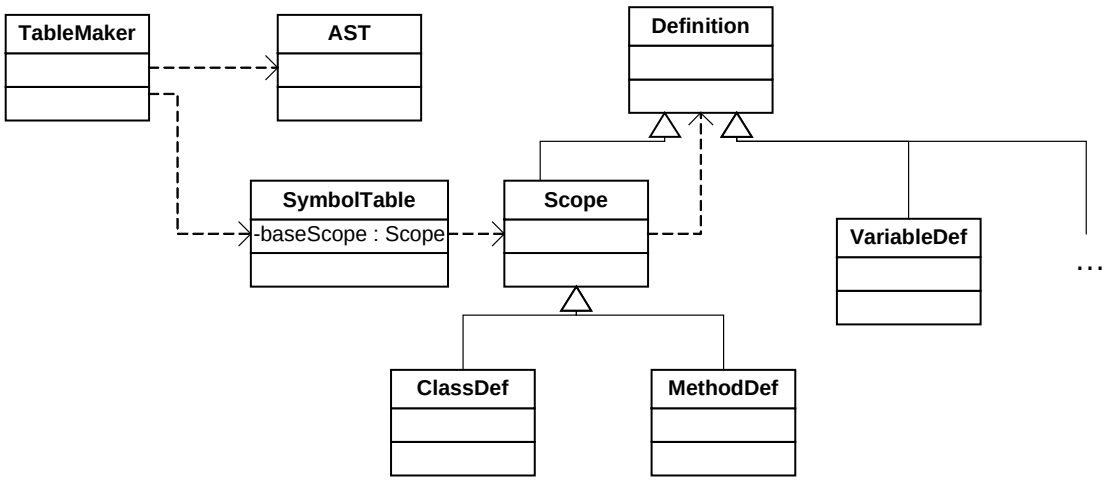


Abbildung 5-3: Transmogrify Kernklassen

Die Klasse `TableMaker` ist für das Traversieren des vom ANTLR-Parser generierten Syntaxbaumes verantwortlich. Beim Traversieren werden die gefundenen Namensräume in der Symboltabelle definiert und für programmiersprachliche Konstrukte wie Klassen, Methoden und Variablendeklarationen werden Definitionsobjekte in den Namensräumen erzeugt.
Nachdem der `TableMaker` den gesamten Syntaxbaum traversiert hat, ist eine komplette Namensraumstruktur erstellt.
Die in Abbildung 5-3 dargestellte Hierarchie der Definitions-Klassen, deren Exemplare in der Symboltabelle verwaltet werden, bildet das fachliche Vokabular der Sourcecodemanagement-Schicht und wird für die Migrationsanalyse in höheren Schichten verwendet.

5.2.3. Auflösung von Verwendungsbezügen

Im vorigen Schritt wurden in der Symboltabelle Informationen über die in allen Klassen definierten Symbole, also Klassendefinitionen, Methodendefinitionen und Variablendefinitionen, gesammelt. Für das Migrationswerkzeug ist es zusätzlich wichtig, wo die Definitionen verwendet werden. Insbesondere müssen die Abhängigkeiten zwischen den Anwendungs- und Rahmenwerksklassen für die Migrationsunterstützung identifiziert werden.
Die Zuordnung von Definitionen zu ihren Verwendungsstellen im Quellcode wird auch als *Crossreferencing* bezeichnet.
Gelöst wird diese Aufgabe durch die Klasse `Resolver` aus dem Transmogrify-Package. Der `Resolver` iteriert wieder über den gesamten Syntaxbaum und ermittelt für alle Bestandteile des Syntaxbaumes die Bezüge zu anderen Elementen.
Jede Definition in der Symboltabelle führt eine Liste von Referenzen. Werden im Syntaxbaum Bezüge zu Klassen- Methoden- oder Variablendefinitionen gefunden, so werden diese Stellen in die Referenzliste der jeweiligen Definition aufgenommen (siehe Abbildung 5-4).

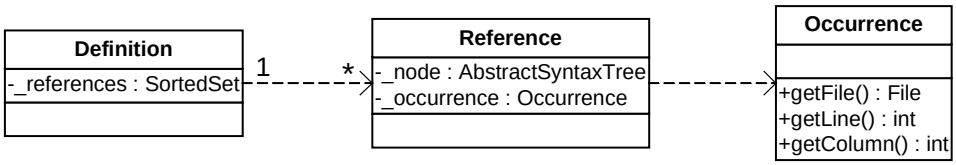


Abbildung 5-4: Klassendiagramm: Reference

Nach dem kompletten Resolver-Prozess kennt jede Definition alle Stellen im Quellcode, an denen sie referenziert wird.

5.2.4. Repository

In den vorhergehenden Abschnitten wurden die Basistechniken zum Einlesen und Analysieren von Quelltexten vorgestellt, mit denen die unteren Ebenen der Sourcecodemanagement-Schicht realisiert werden. In diesem Abschnitt wird das Konzept des Repositorys vorgestellt, das den vorhandenen Unterbau um eine wesentliche Fähigkeit erweitert: die Navigation in der Zeit.
Mit dem Repository werden auch die restlichen Anforderungen an das Sourcecode-Management umgesetzt. Während die bisher vorgestellten Basistechniken angepasste Produkte der Open-Source- und Forschungsgemeinde sind, ist das Repository eine im Rahmen dieser Arbeit entwickelte Erweiterung der Quellcode-Verwaltung.
Das Repository ist die zentrale Instanz für das Sourcecode-Management des Migrationswerkzeuges. Das Repository wird mit allen benötigten Rahmenwerks- und Anwendungs-Quelldateien bestückt und verfügt danach über das Wissen über alle Klassen- und Methodendefinitionen der Anwendung, des aktuellen Rahmenwerkes und der vorangegangenen Rahmenwerksversion. Zusätzlich kennt das Repository einen Teil der in der zukünftigen Rahmenwerksversion geänderten Schnittstelle. Die Schnittstelle des Repositorys abstrahiert weitgehend von den konkret verwendeten Quellcode-Parser- und Analysetechnologien.
Der Schnittstellenentwurf in Abbildung 5-5 verdeutlicht den Umgang mit dem Repository.

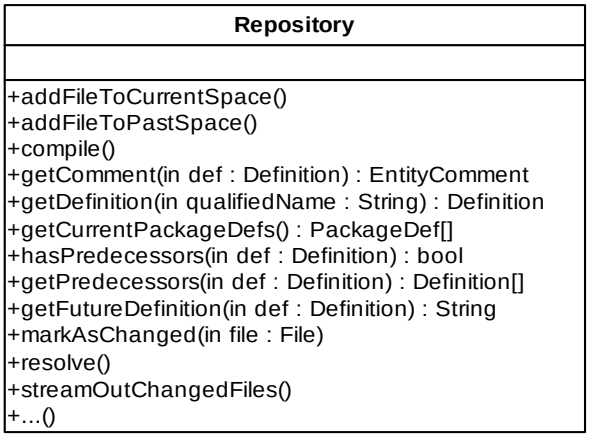


Abbildung 5-5: Repository-Schnittstelle

Mit den Operationen `addFileToCurrentSpace` und `addFileToPastSpace` ist es möglich, Quelldateien zum Bestand des vom Repository verwalteten Sourcecodes hinzuzufügen. Zur Unterscheidung der Namensräume von alter, neuer und zukünftiger Rahmenwerksversion wird definiert:

Begriff: PastSpace
Der *PastSpace* ist der Namensraum, der durch die ursprüngliche Rahmenwerksversion und die darauf basierende Anwendung gebildet wird.

Begriff: CurrentSpace
Der *CurrentSpace* ist der Namensraum der neuen Rahmenwerksversion.

Begriff: FutureSpace
Der *FutureSpace* ist der Namensraum der zukünftigen Rahmenwerksversion.

Sind dem Repository alle benötigten Rahmenwerks- und Anwendungsklassen bekanntgegeben worden, so können über Aufruf der `resolve`-Methode die Prozesse des Parsens, der Symboltabellengenerierung und des Auflösens aller Verwendungsbezüge angestoßen werden. Das Repository kapselt dabei die dahinter stehenden konkreten Prozesse.
Am Beispiel des JWAMMicro-Technology-Packages soll die Aufgabe des Repositorys verdeutlicht werden. Im oberen Teil der Abbildung 5-6 sind die Quellen einiger Klassen des Technology-Packages dargestellt.
Dem Repository wird der Quellcode der Klassen `PersistencyManager` und `RDBMSPersistencyManager` aus dem alten Rahmenwerk übergeben. In der neuen Rahmenwerksversion ist die Funktionalität beider Klassen vereint in der Klasse `PersistencyManager`. Auch diese Klasse wird dem Repository übergeben.
Im Repository wird über die Quellcode-Analyse eine versionsübergreifende Symbol- und Namensraumstruktur entwickelt. In der Abbildung 5-6 sind im unteren Bereich die Namensräume der Klassen- und Methodendefinitionen dargestellt, die aus den Quellen gewonnen wurden. Dem Namensraum des FutureSpace liegt kein Quellcode zugrunde, er wird über die Analyse der Future-Tags der Klassen im CurrentSpace gebildet. Die im Beispiel durchgestrichene Definition der `commit`-Methode im FutureSpace zeigt, dass die Methode in der nächsten Version nicht mehr vorhanden sein wird.

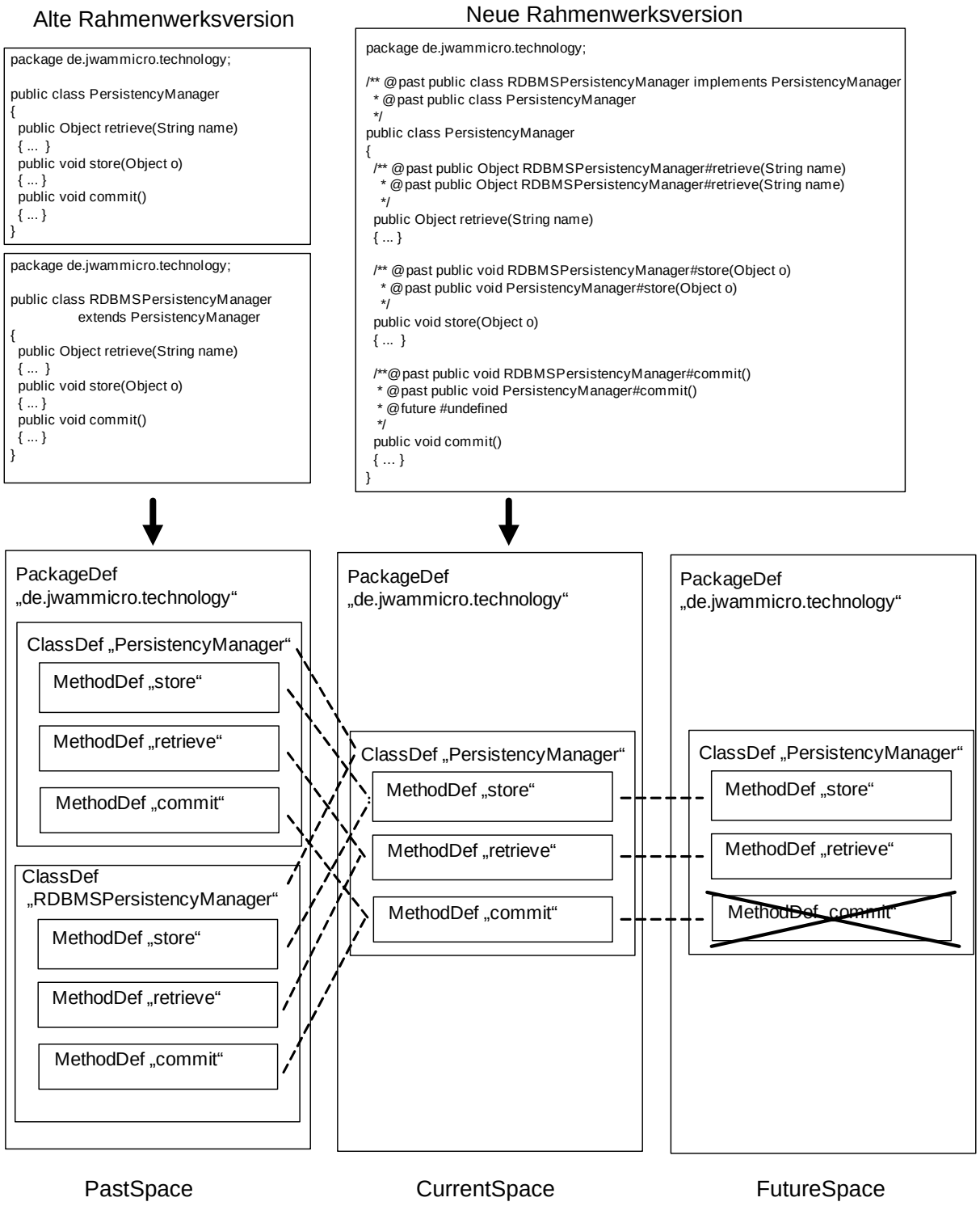


Abbildung 5-6: Repository - Beispielkonfiguration

Die besondere Fähigkeit des Repositorys liegt in der Möglichkeit, zu jeder Definition deren Vorgänger- oder Nachfolgerversion zu ermitteln. Das Repository stellt die Verbindung zwischen den Definitionen der alten, neuen und zukünftigen Rahmenwerksversion her.
Über die Methoden `getPredecessors(Definition)` und `getFutureDefiniti-on(Definition)` kann für den Definitions-Parameter zwischen den Rahmenwerksversionen navigiert werden (siehe Abbildung 5-7). Das Repository realisiert damit eine um eine Zeit-Dimension erweiterte Symboltabelle. Damit ermöglicht das Repository höheren Schichten,

Unterschiede zwischen den Rahmenwerksversionen zu erkennen und daraus notwendige Migrationsschritte zu ermitteln.

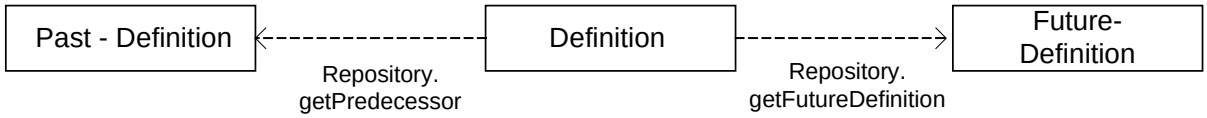


Abbildung 5-7: Navigation zwischen Definitionsversionen

Umsetzung
Die interne Struktur des Repositorys ist im folgenden Objektdiagramm dargestellt:

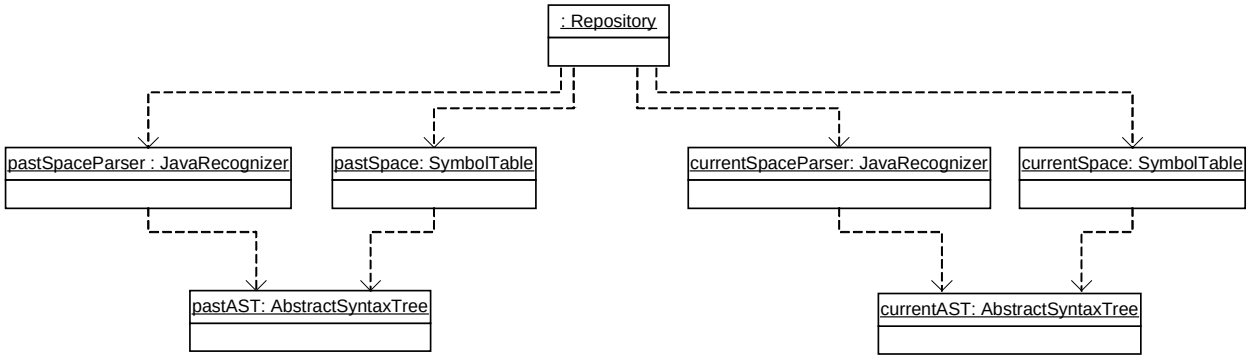
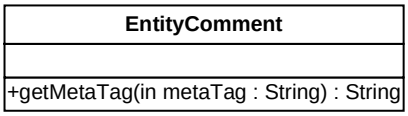


Abbildung 5-8: Objektdiagramm: Repository

Das Repository verwendet intern zwei ANTLR-generierte Parser, die jeweils für den Aufbau des PastSpace- und des CurrentSpace-Syntaxbaumes zuständig sind. Der PastSpace muss mit den Quelltexten der Anwendung und der alten Rahmenwerksversion gefüllt werden, der CurrentSpace mit der neuen Rahmenwerksversion. Nachdem alle Quelltexte geparst wurden, werden zwei Transmogrify-Symboltabellen getrennt über die beiden generierten Syntaxbäume erzeugt. Der PastSpace und der CurrentSpace sind zwei komplett getrennte Namensräume. Das Repository als zentrale Verwaltungsinstanz führt diese getrennten Räume zusammen und erlaubt den Blick in die Vergangenheit und Zukunft von Definitionen.
Die Zeitnavigation wird vom Repository über Auswertung der Past- und Future-Tags der Definitionen umgesetzt. Da alle Klassen- und Methodendefinitionen in den Symboltabellen ihren definierenden Knoten im Syntaxbaum kennen, kann im Syntaxbaum nach dem zum Definitionsknoten zugehörigen Kommentarknoten gesucht werden. Das Auslesen des Kommentars ist in der Methode `getComment(Definition)` gekapselt. Die Operation `getComment` liefert ein Objekt vom Typ `EntityComment`, das den Kommentartext kapselt.



Über die Methode `getMetaTag` können an einem `EntityComment` die Werte der Tags im Kommentar erfragt werden. Soll beispielsweise der Vorgänger einer CurrentSpace-Definition ermittelt werden, so wird am EntityComment-Objekt der Definition nach einem Past-Tag gesucht. Ist ein Past-Tag vorhanden, so kann über den Tag-Wert der Name und die Struktur der

zeitlich vorhergehenden Definition ausgelesen werden. Über die Symboltabelle des PastSpace kann mit diesen Informationen die Vorgängerdefinition ermittelt werden.

Zur Verwaltung des Quellcodes gehört auch die Aufgabe, geänderte Definitionen oder Quelltexte durch Zurückschreiben in Quelldateien persistent zu machen. Die Änderungen müssen dazu in den Syntaxbäumen vollzogen worden sein. Geänderte Quelltexte können am Repository über die Operation `markAsChanged` als verändert markiert werden. Über die Methode `streamOutChangedFiles` werden dann alle geänderten Quelldateien mit den aktuellen Inhalten überschrieben.

Damit erfüllt das Repository alle an das Sourcecode-Management gestellten Anforderungen:

- Verwaltung und Analyse von Quelltexten über die `addFile`- und `resolve`-Methoden
- Zugriff auf Klassen- und Methodendefinitionen über die `getDefinition`- und `getCurrentPackageDefs`-Methoden, die einen direkten Zugriff auf Definitionen und Packages ermöglichen
- Rückblick und Vorausschau auf Rahmenwerksversionen wird über die Methoden `getPredecessors` und `getFutureDefinition` ermöglicht.
- Die in den Meta-Tags dokumentierten Modifikationsinformationen können über das `EntityComment`-Objekt einer Definition erfragt werden.
- Änderungen an der im Syntaxbaum abgelegten Struktur eines Quelltextes können über die Operationen `markAsChanged` und `streamOutChangedFiles` persistent gemacht werden.

5.3. Die Migrationsanalyse-Schicht

Die im letzten Kapitel beschriebene Sourcecodemanagement-Schicht bietet den Zugriff auf Quellcodestrukturen und Symbolinformationen der Rahmenwerks- und Anwendungsklassen und ermöglicht die Abfrage von Vorgänger- und Nachfolgerversionen von Klassen- und Methodendefinitionen. Das Repository ist die Grundlage für die Analysen der Migrationsanalyse-Schicht.
Aufgabe der Migrationsanalyse-Schicht ist, die Stellen in den Anwendungsklassen zu identifizieren, an denen ein Migrationsschritt durchzuführen ist. Die Migrationsanalyse-Schicht generiert für diese Quellcodestellen Migrationshinweise und Migrationsanleitungen.

> **Begriff: Migrationshinweis**
> Ein Migrationshinweis weist auf eine Stelle im Quellcode, an der im Rahmen des Migrationsprozesses Modifikationen durchzuführen sind. Wird beispielsweise in einer Anwendungsklasse eine umbenannte oder in der nächsten Version gelöschte Rahmenwerksklasse referenziert, so muss für diese Stelle ein Migrationshinweis generiert werden.

> **Begriff: Migrationsanleitung**
> Eine Migrationsanleitung kann ergänzend zu einem Migrationshinweis beschreiben, wie die zu modifizierende Quellcodestelle auf die neue Rahmenwerksversion (automatisch) migriert werden kann. Für eine umbenannte Rahmenwerksklasse wird beispielsweise beschrieben, wie der neue Klassenname in den Anwendungs-Quellcode eingefügt werden muss. Für einige Rahmenwerksmodifikationen wie für das Löschen

von Rahmenwerksklassen können keine allgemeingültigen Migrationsanleitungen formuliert werden.

Zur Erstellung von Migrationshinweisen und -anleitungen müssen unter Verwendung des Repositorys die Modifikationen erkannt werden, die am Rahmenwerk durchgeführt worden sind. Für jede dieser Modifikationen muss dann analysiert werden, ob sie zu Strukturkonflikten in den Anwendungsklassen führen.
Es handelt sich dabei um eine nachträgliche Analyse des im Kontext der Change-Impact-Analysis beschriebenen Ripple-Effektes (siehe Kapitel 2.3.1.1).

5.3.1. Identifikation von Rahmenwerksmodifikationen

Um alle Stellen des Rahmenwerkes zu identifizieren, an denen zwischen alter und neuer Version Modifikationen vorgenommen wurden, müssen alle Definitionen des neuen Rahmenwerkes untersucht werden. Das Repository ermöglicht den Zugriff auf den Symbolbaum der Symboltabelle. Alle dort verwalteten Definitionen der neuen Rahmenwerksversion müssen auf Modifikations-Tags geprüft und mit ihren Vorgängern in der alten Rahmenwerksversion verglichen werden.
Für die Traversierung baumartiger Strukturen und die Analyse der traversierten Knoten hat sich das Besuchermuster bewährt (siehe [Gamma et al. 96], S. 301 ff.). Der Traversierungsmechanismus wird dabei getrennt von den dynamisch erweiterbaren Knoten-Analyseverfahren beschrieben. Das Muster wurde wie in Abbildung 5-9 dargestellt für die Symboltabellenanalyse umgesetzt.

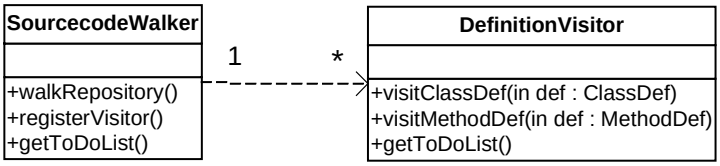


Abbildung 5-9: Umsetzung des Visitor-Patterns

In der Klasse `SourcecodeWalker` ist die Strategie zur Traversierung der im Repository verwalteten Symboltabelle implementiert. An einem `SourcecodeWalker`-Exemplar können beliebig viele `DefinitionVisitor`-Objekte registriert werden, in denen die verschiedenen Analyseverfahren gekapselt sind.
Der `SourcecodeWalker` iteriert über die Definitionen des Repositorys und ruft für jede Definition an jedem registrierten Visitor-Objekt die entsprechende `visit`-Operation. Die Verwendung des Besuchermusters bietet sich an, da die Anzahl der Definitionsklassen der Symboltabelle unveränderlich ist und über die Definition von neuen Visitor-Klassen beliebige Analysefunktionalität dynamisch hinzugefügt werden kann.
Wird an einem DefinitionVisitor die Methode `visitClassDef` oder `visitMethodDef` gerufen, so „besucht“ der Visitor die übergebene Definition und kann über das Repository die Modifikations-Tags der Definition untersuchen oder die Definition mit einer früheren oder zukünftigen Definitionsversion vergleichen. Erkennt der Visitor dabei ein Modifikationsmuster, so muss für diese Modifikation untersucht werden, ob sie zu einem Strukturkonflikt in Anwendungsklassen geführt hat.

5.3.2. Generierung von Migrationshinweisen

Für die vom Visitor ermittelte Modifikation muss eine Ripple-Effekt-Analyse durchgeführt werden. Dazu müssen alle direkt betroffenen Verwendungsreferenzen der in der neuen Rah-

menwerksversion modifizierten PastSpace-Definition untersucht werden. Rekursiv müssen wiederum davon abhängige Verwendungsbezüge berücksichtigt werden. Für jeden auf diesem Wege in Anwendungsklassen aufgefundenen Strukturkonflikt muss ein Migrationshinweis generiert werden. Verantwortlich für die Ermittlung und Generierung von Migrationshinweisen ist der Visitor, der die Modifikation ermittelt hat.
Die Modellierung von Migrationshinweisen (ToDoEntries) und Migrationsanleitungen (ModificationCommands) ist dargestellt in Abbildung 5-10.

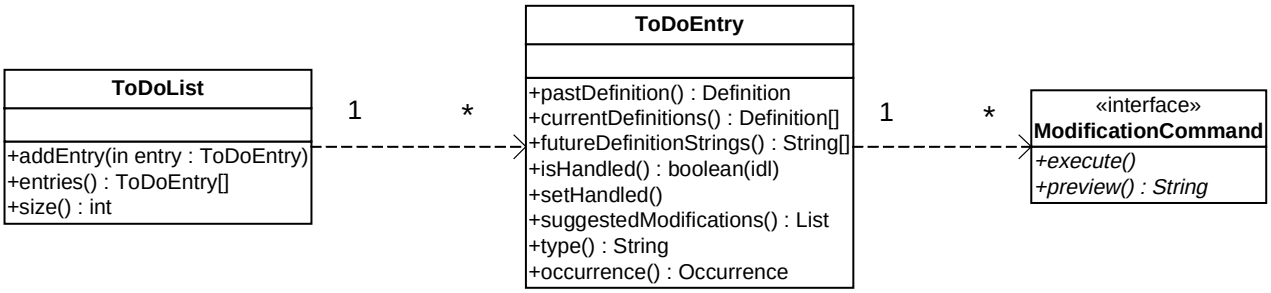


Abbildung 5-10: Modellierung von Migrationshinweisen und Migrationsanleitungen

Ein ToDoEntry verwaltet alle Informationen eines Migrationshinweises. Darin enthalten ist eine Referenz auf die zu migrierende Quelltextstelle, Informationen über die ursprünglich referenzierte Rahmenwerksdefinition, über die aktuellen Nachfolger der ursprünglichen Definition und gegebenenfalls ein Verweis auf eine zukünftige Definitionsgestalt.
Über die `occurrence`-Operation kann die Datei und die genaue Zeilen- und Spaltenposition der Definitionsverwendung innerhalb der Datei ermittelt werden. Jeder `ToDoEntry` hat einen Typ, der die Art der identifizierten Modifikation zwischen der verwendeten alten und aktuellen Definition beschreibt und der im Klartext über die Methode `type` ausgelesen werden kann.
ToDoEntries führen einen Status, über den ermittelt werden kann, ob dieser Migrationshinweis als erledigt betrachtet werden kann. Der Status ist über `setHandled` setzbar und kann über das Prädikat `isHandled` abgefragt werden.
Migrationshinweise werden in einer Liste gesammelt, die von dem jeweiligen Visitor-Objekt verwaltet wird.

Während des gesamten Analysevorganges werden nur Informationen über den nötigen Migrationsprozess zusammengetragen. Das Werkzeug präsentiert später dem mit der Migration betrauten Entwickler die gesammelten Analyseergebnisse. Erst dann werden vom Anwendungsentwickler gesteuert Migrationsschritte durchgeführt. Wie im Kapitel 4 beschrieben, sind für einige Modifikationen bei ausreichender Dokumentation die nötigen Migrationsschritte automatisiert durchführbar. Auch die Quellcodemanipulationen eines automatisierbaren Migrationsschrittes erfolgen erst auf Anweisung des Entwicklers. Die während der Analyse identifizierten notwendigen Quellcodemanipulationen müssen bis zu diesem Auslösungszeitpunkt aufgeschoben werden.

Diese Problemstellung wird von dem Befehlsmuster (engl. Command-Pattern) adressiert (siehe [Gamma et al. 96], S. 273 ff.). In einem Command-Objekt kann ein parametrisierbarer Algorithmus gekapselt werden, der über Aufruf der Methode `execute` zu einem beliebigen Zeitpunkt ausgeführt werden kann. Im Modell von Abbildung 5-10 ist dargestellt, dass einem ToDoEntry `ModificationCommand`-Objekte zugeordnet werden können, die jeweils die Migrationsanleitung für einen automatisiert durchführbaren Migrationsschritt kapseln. Die

`ModificationCommand`-Objekte können von einem Visitor erzeugt und konfiguriert werden, wenn die für die Migration notwendigen Informationen verfügbar sind. Ist die Modifikation semi-automatisierbar, so werden mehrere alternative `ModificationCommand`-Objekte zugeordnet.

5.3.3. Implementation

5.3.3.1. Visitor-Klassen

Nach dem im letzten Abschnitt beschriebenen Muster wurde eine Anzahl Visitorklassen implementiert (siehe Abbildung 5-11). Jeder konkrete Visitor ist so konstruiert, dass er Teilaspekte von Basismodifikationen erkennen kann und gegebenenfalls für automatisiert durchführbare Migrationsschritte eine Migrationsanleitung in Form eines `ModificationCommand`-Objektes generiert.

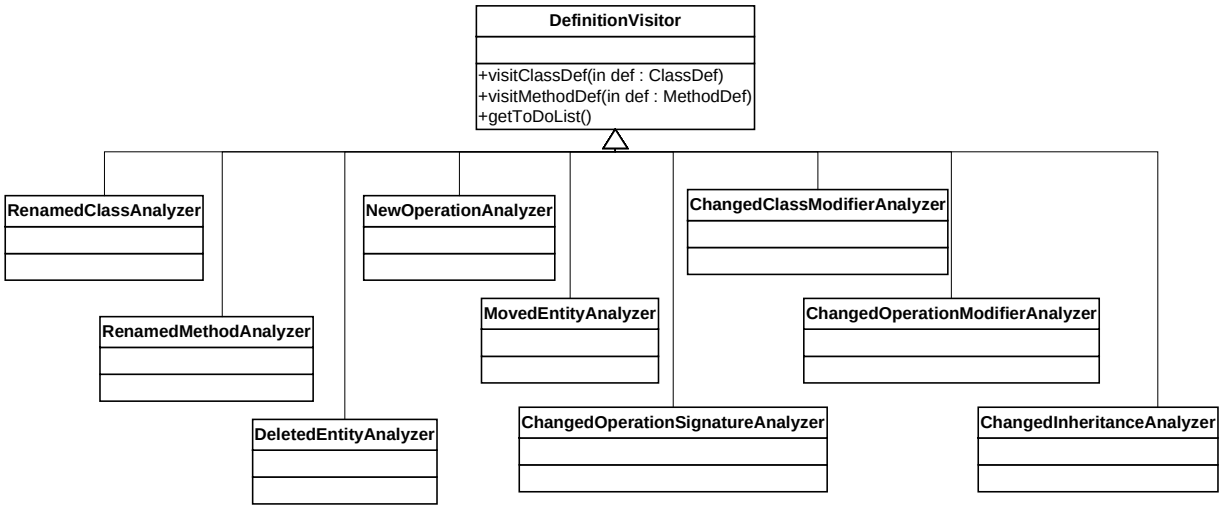


Abbildung 5-11: Klassendiagramm: DefinitionVisitor-Implementationen

Am Beispiel der `RenamedMethodAnalyzer`-Klasse soll verdeutlicht werden, wie die Modifikations- und Auswirkungsanalyse umgesetzt worden ist. Gegeben sei die in Abbildung 5-12 dargestellte Symbolstruktur, bestehend aus den Definitionen des neuen Rahmenwerkes, der vorhergehenden Rahmenwerksversion und einigen Anwendungsklassen.

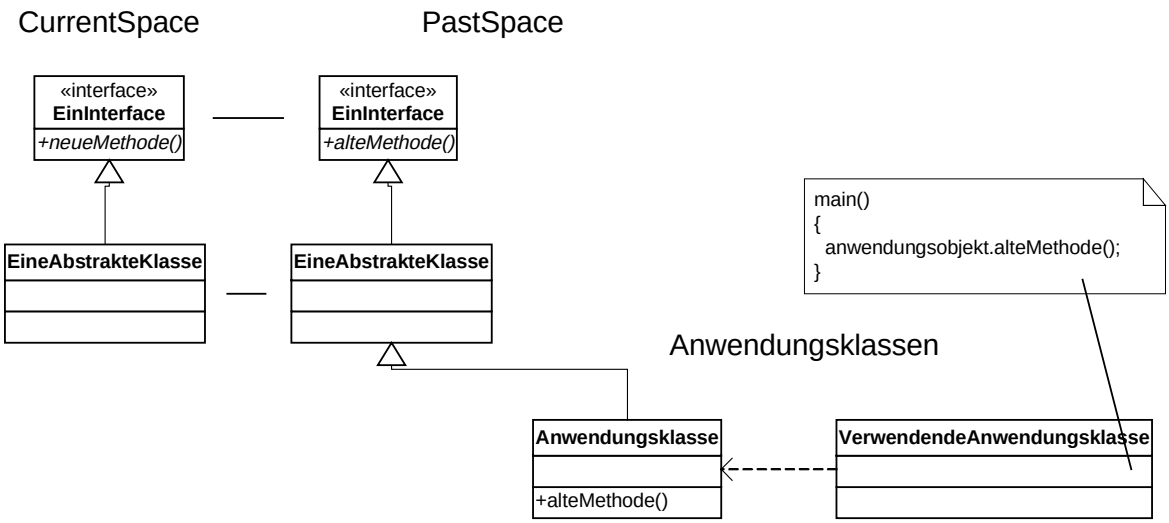


Abbildung 5-12: Klassen für das RenamedMethodAnalyzer-Beispiel

Der SourcecodeWalker traversiert den Symbolbaum des CurrentSpace und gelangt zur Definition der Operation `neueMethode`.
Die Methodendefinition der Operation `neueMethode` wird dem `RenamedMethodAnalyzer` zur Untersuchung übergeben. Der Analyzer bekommt über das Repository die zugehörige PastSpace-Definition und kann erkennen, dass die ursprüngliche Methode im PastSpace einen anderen Namen trägt. Es reicht jetzt nicht, nur die Verwendungsreferenzen der Methodendefinition von `alteMethode` zu berücksichtigen. Zusätzlich müssen auch alle Redefinitionen dieser Methode und ihre Referenzen im PastSpace berücksichtigt werden. Dazu wird im PastSpace ausgehend von der Klasse, die die ursprüngliche Methodendefinition enthält, rekursiv in alle Subinterfaces, deren implementierende Klassen und Subklassen abgestiegen. Wird irgendwo in diesem Vererbungsbaum die Methode `alteMethode` redefiniert, so werden auch deren Verwendungsreferenzen berücksichtigt. Handelt es sich um eine Redefinition oder Verwendung in einer Anwendungsklasse, so wird ein `ToDoEntry` vom Typ „Rename-Method“ erzeugt. Da die Umbenennung einer Methodenverwendung in der Anwendungsklasse automatisiert vollzogen werden kann, wird dem erzeugten `ToDoEntry` ein `ReplaceNameModification`-Command angefügt, in dem alle nötigen Informationen zum automatischen Vollzug der Umbenennung enthalten sind.
In obigem Beispiel würde also bei der Untersuchung der Operation `neueMethode` sowohl die Redefinition von `alteMethode` in der Klasse `Anwendungsklasse`, als auch der Aufruf in der Klasse `VerwendendeAnwendungsklasse`, in einem „Rename-Method“-`ToDoEntry` erfasst werden.

Der `RenamedMethodAnalyzer` erkennt einen Teil der durch Basismodifikationen vom Typ „Ersetze Elementname“ hervorgerufenen Strukturkonflikte. Die im Folgenden in knapper Form vorgestellten Visitor-Klassen decken die Auswirkungen weiterer Modifikationen ab. Die Analyse des Ripple-Effektes wird dabei jeweils nach dem im Beispiel erläuterten Vorgehen durchgeführt.

- **RenamedClassAnalyzer**
  Umbenannte Klassen werden über Vergleich der Namen von alter und neuer Definitionsversion identifiziert.

- **DeletedEntityAnalyzer**
  Der `DeletedEntityAnalyzer` untersucht, ob die jeweils gegebene Klassen- oder Methodendefinition in der zukünftigen Rahmenwerksversion entfernt sein wird. Dafür wird geprüft, ob im Kommentar der Definitionen ein Future-Tag mit dem „#UNDEFINED“-Wert vorhanden ist.
- **NewOperationAnalyzer**
  Der `NewOperationAnalyzer` prüft, ob im Kommentar der gegebenen Methodendefinition ein Since-Tag vorhanden ist. Der Wert des Tags wird verglichen mit der neuen Rahmenwerks-Versionsnummer, die vorher am Repository registriert worden sein muss. Sind die Werte identisch, so handelt es sich um eine in der neuen Rahmenwerksversion hinzugefügte Operation. Handelt es sich um eine in einem Interface definierte oder um eine abstrakte Methodendefinition, so müssen die Anwendungsklassen gesucht werden, in denen die hinzugefügte Methode implementiert werden muss.
- **ChangedInheritanceAnalyzer**
  Der `ChangedInheritanceAnalyzer` vergleicht für Interface- und Klassendefinitionen die Vererbungs- und Implementationsbeziehungen im CurrentSpace und PastSpace.
- **ChangedOperationSignatureAnalyzer**
  Der `ChangedOperationSignatureAnalyzer` vergleicht die Signaturen der aktuellen Methodendefinition mit der im PastSpace befindlichen alten Version.
- **ChangedClassModifierAnalyzer**
  Der `ChangedClassModifierAnalyzer` vergleicht die Modifier der gegebenen aktuellen Klassendefinition mit den Modifiern der im PastSpace befindlichen alten Klassendefinition.
- **ChangedOperationModifierAnalyzer**
  Der `ChangedOperationModifierAnalyzer` vergleicht die Modifier der gegebenen aktuellen Methodendefinition mit der im PastSpace befindlichen alten Version.
- **MovedEntityAnalyzer**
  Der `MovedEntityAnalyzer` vergleicht den Kontext von Klassen- und Methodendefinitionen in Current- und PastSpace. Bei Klassendefinitionen wird geprüft, ob sich alte und neue Definition im gleichen Package befinden.
  Bei Methodendefinitionen werden die Klassen der gegebenen CurrentSpace- und der zugehörigen PastSpace-Vorgänger-Methodendefinition verglichen. Stehen die zugehörigen Klassen nicht in einer Vorgänger-Nachfolger-Beziehung, so hat die Methodendefinition den Kontext gewechselt.

Abschließend wird der Bezug von Visitor-Klassen zu den von ihnen erkannten Basismodifikationen dargestellt:

Basismodifikation	**wird erkannt von**
Erzeuge Element	CreatedOperationAnalyzer
Lösche Element	DeletedEntityAnalyzer
Ersetze Elementname	RenamedClassAnalyzer, RenamedOperationAnalyzer
Ersetze Referenzliste	ChangedOperationSignatureAnalyzer, ChangedInheritanceAnalyzer
Ersetze Elementmodifier	ChangedClassModifierAnalyzer, ChangedOperationModifierAnalyzer
Ersetze Elementwert	semantische Änderung, wird im Rahmen dieser Arbeit nicht berücksichtigt
Ersetze Elementvertrag	semantische Änderung, wird im Rahmen dieser Arbeit nicht berücksichtigt
Ersetze Kontext	MovedEntityAnalyzer

Als Ergebnis kann festgehalten werden:

Die implementierten Visitorklassen erkennen jeweils einen Teilaspekt der Basismodifikationen. Die Gesamtheit aller Visitorklassen erkennt sämtliche Basismodifikationen, die zu einem Strukturkonflikt führen können. Die Visitorklassen können die davon betroffenen Stellen in den Anwendungsklassen identifizieren und durch Migrationshinweise in ToDoEntry-Objekten vergegenständlichen. Ausgeklammert wird die Analyse der rein semantischen Basismodifikationen *Ersetze Elementwert* und *Ersetze Elementvertrag*, die nicht zu einem Strukturkonflikt führen.

5.3.3.2. ModificationCommand-Klassen

Im letzten Abschnitt wurden die `ModificationAnalyzer` vorgestellt, die für das Identifizieren von nötigen Migrationsschritten und für die Generierung von Migrationshinweisen zuständig sind. Einige der Migrationsschritte sind automatisiert durchführbar. Für diese Klasse von Migrationsschritten können Migrationsanleitungen erstellt werden, die als `ModificationCommand`-Klassen modelliert werden. `ModificationCommand-Objekte` werden von `ModificationAnalyzern` erzeugt und an einen `ToDoEntry` gebunden (siehe Abbildung 5-13). Die einzelnen `ModificationCommand`-Klassen werden im Folgenden genauer vorgestellt.

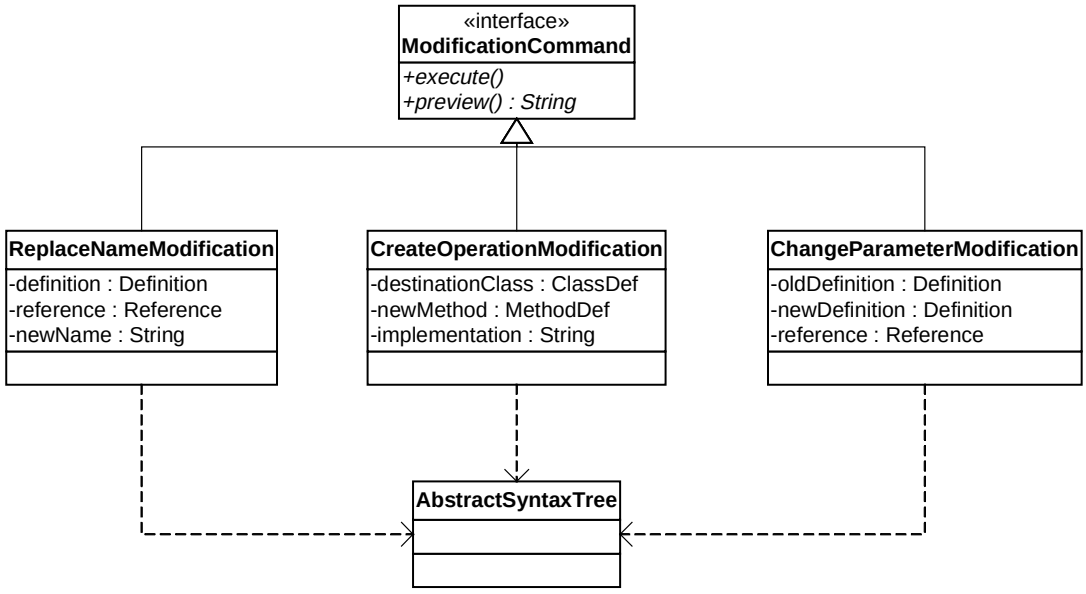


Abbildung 5-13: Klassendiagramm ModificationCommands

- **ReplaceNameModification**
  Das `ReplaceNameModification`-Command repräsentiert eine Migrationsanleitung zur Umbenennung von Definitionen oder Definitionsverwendungen und muss mit der umzubenennenden Definition oder Referenz und dem neuen Namen konfiguriert werden. Die Definition oder Referenz kennt ihren definierenden Knoten im abstrakten Syntaxbaum. Durch Aufruf von `execute` wird dann direkt dieser Knoten des abstrakten Syntaxbaumes modifiziert, der die umzubenennende Quelltextstelle repräsentiert.

- **CreateOperationModification**
  Ein `CreateOperationModification`-Command kapselt einen Migrationsschritt, in dem einer Klasse eine neue Methode hinzugefügt wird. Dazu wird das Command-Objekt mit einer Zielklassen-Definition, der Definition der zu implementierenden neuen Methode und einem Implementationskörper in Form einer Zeichenkette konfiguriert. Bei Ausführung des Commands wird im abstrakten Syntaxbaum der Zielklasse ein neuer Methodendefinitionsknoten erzeugt, der mit der Signatur der neuen Methode und dem konfigurierten Quelltext-Körper versehen wird.

- **ChangeParameterModification**
  Ein `ChangeParameterModification`-Command kapselt einen Migrationsschritt, in dem die Anpassung der Parametrisierung eines Methodenaufrufes oder einer Methodendefinition automatisch vollzogen wird. Das Command-Objekt wird dazu konfiguriert mit der alten Methodendefinition, der neuen Definition und gegebenenfalls mit einer Referenz. Beim Aufruf von `execute` werden zwei Fälle unterschieden:
  Muss die Signatur einer Methodendefinition angepasst werden, so wird im abstrakten Syntaxbaum der Signaturknoten der neuen Methodendefinition kopiert und an die Stelle des alten Signaturknotens der anzupassenden Definition gesetzt. Fehlen in der neuen Signatur Parameter, die in der alten vorhanden waren, so müssen die fehlenden Parameter in der Methode als Variablendeklarationen erzeugt werden.
  Muss die Parametrisierung eines Aufrufs angepasst werden, so werden die alten Aufruf-Parameter der neuen Signatur entsprechend umgestellt. In der neuen Signatur fehlende Parameter werden weggelassen. Neu hinzugekommene Parameter werden mit den Default-Werten, die im Kommentar der neuen Definition beschrieben sind, gefüllt.

Bei jedem `execute` an einem ModificationCommand werden Modifikationen direkt am abstrakten Syntaxbaum vollzogen. Danach werden am Repository die Quelltexte, an deren Syntaxbäumen die Anpassungen vorgenommen wurden, über die Methode `markAsChanged` als verändert markiert, so dass die Änderungen beim Zurückschreiben in Quelldateien vom Repository berücksichtigt werden.

5.4. Migrationswerkzeug

Nachdem das Sourcecodemanagement und die Klassen der Migrationsanalyse-Schicht dargestellt wurden, wird nun das darauf basierende Migrationswerkzeug vorgestellt.
Das Migrationswerkzeug dient zur Konfiguration, Steuerung und Visualisierung des Migrationsprozesses. Dabei bedient es sich der Funktionalität des Sourcecodemanagements und der Migrationsanalyseschicht.

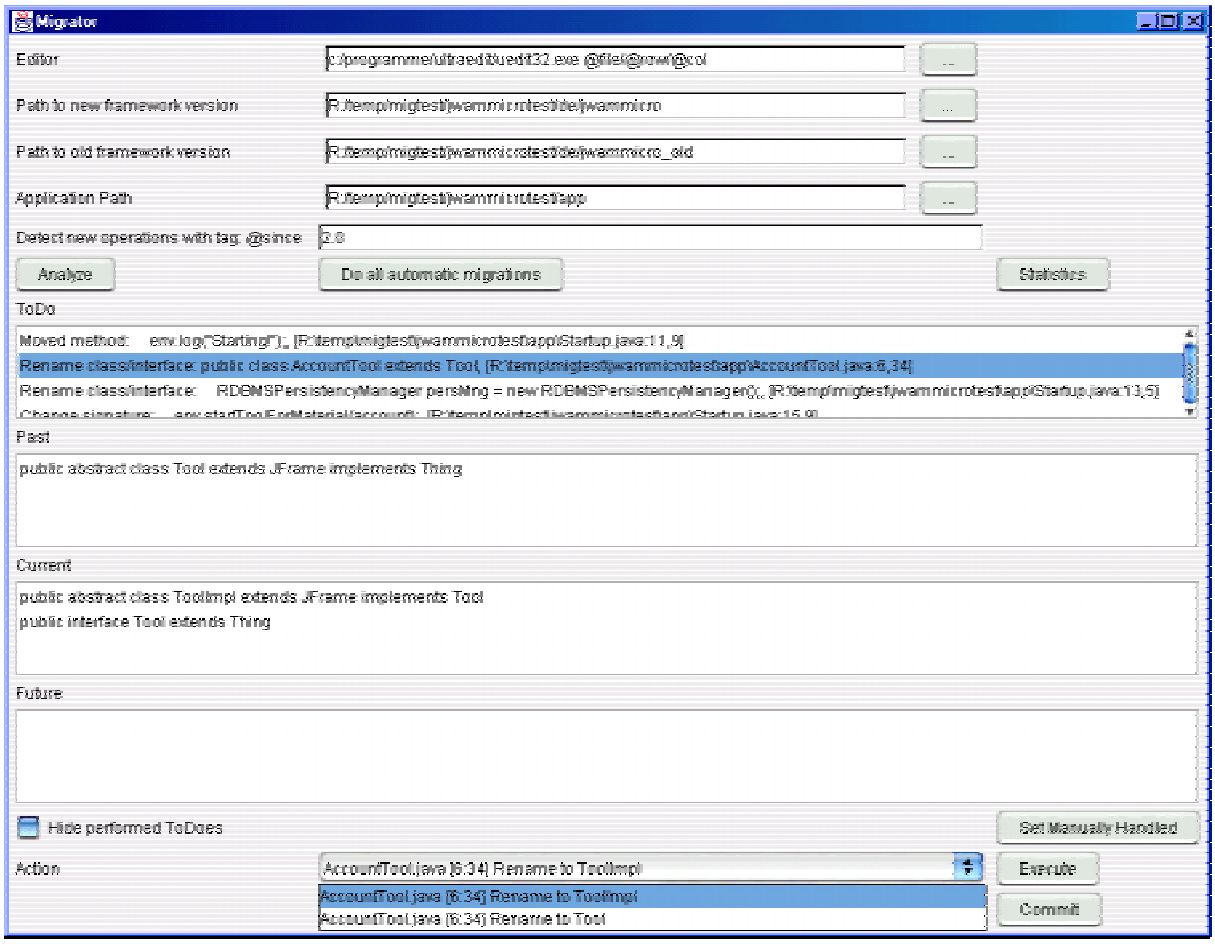


Abbildung 5-14: Migrationswerkzeug

Im oberen Bereich des Werkzeuges (siehe Abbildung 5-14) sind Eingabefelder, in denen der Pfad zur neuen Rahmenwerksversion, zur alten Rahmenwerksversion und der Pfad zu der zu migrierenden Anwendung konfiguriert werden kann.

Definitionen, die in der neuen Rahmenwerksversion hinzugefügt wurden, können über ein Since-Tag mit dem Wert der neuen Versionsnummer erkannt werden. Die neue Versionsnummer kann im oberen Bereich des Werkzeuges definiert werden.
Zusätzlich kann der Pfad zu einem Texteditor gesetzt werden, der für die manuelle Migration verwendet wird.
Über den „Analyze“ –Button wird die Analyse zur Ermittlung der notwendigen Migrationsschritte eingeleitet. Im folgenden Sequenzdiagramm wird der Analyseprozess und das Zusammenspiel der Schichten veranschaulicht:

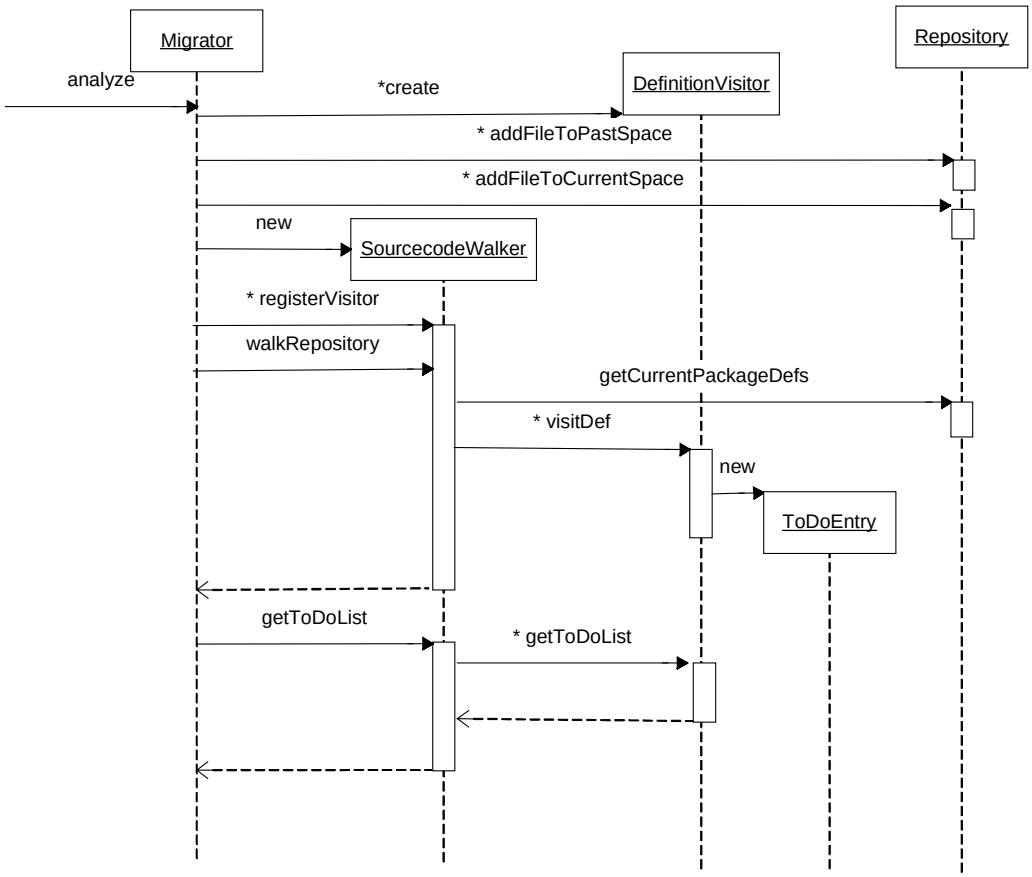


Abbildung 5-15: Sequenzdiagramm des Analyseprozesses

Zuerst werden die vom Werkzeug verwendeten `DefinitionVisitor`-Objekte erzeugt. Dann werden dem Repository alle Dateien zum Past- und CurrentSpace hinzugefügt, die unter den konfigurierten Rahmenwerks- und Anwendungspfaden gefunden werden. Ein `SourcecodeWalker`-Objekt, bei dem die `DefinitionVisitor`-Objekte registriert werden, iteriert dann über alle Definitionen des CurrentSpace und lässt die Definitionen von den registrierten Visitor-Objekten untersuchen. Nach der Iteration kann eine Liste aller Migrationshinweise am SourcecodeWalker erfragt werden.

Der mittlere Bereich des Werkzeuges dient zur Anzeige der Migrationshinweise in der generierten `ToDoList`. Im Klartext wird für jeden `ToDoEntry` der Typ der identifizierten Modifikation und ein Quellcodeausschnitt der betroffenen Zeile in der Anwendungsklasse angezeigt. Durch Auswahl eines ToDoEntries werden in den Feldern darunter nähere Informationen über die am Rahmenwerk durchgeführte oder angekündigte Modifikation angezeigt. Die

von der Anwendungsklasse referenzierte PastSpace-Definition wird zuoberst angezeigt, darunter die modifizierten Nachfolger der Definition im CurrentSpace und gegebenenfalls die Gestalt der Definition in der zukünftigen Rahmenwerksversion.
Im unteren Bereich des Werkzeuges wird der Migrationsprozess gesteuert.

- Für automatisiert durchführbare Migrationsschritte wurden während des Analyseprozesses Migrationsanleitungen in Form von `ModificationCommand`-Objekten an die `ToDoEntry`-Objekte gebunden. Die `ModificationCommand`-Objekte für den selektierten `ToDoEntry` erscheinen unten in der „Action“-Drop-Down-Liste. Für semi-automatisierbare Modifikationen erscheinen mehrere mögliche Alternativen. Durch Drücken der „Execute“-Schaltfläche wird das ausgewählte ModificationCommand in die Liste der durchzuführenden ModificationCommands des Migrationswerkzeuges übernommen und der ausgewählte `ToDoEntry` als bearbeitet markiert.
  Der Benutzer des Migrationswerkzeuges kann auf diese Weise die Liste der ToDoEntries abarbeiten, bis alle automatisierbaren ToDoEntries als bearbeitet markiert sind. Durch Betätigung der „Commit“-Schaltfläche wird dann über die intern verwaltete Liste der durchzuführenden ModificationCommands iteriert. An jedem dieser `ModificationCommand`-Objekte wird die Operation `execute` aufgerufen, die dann auf die im Kapitel 5.3.3.2 beschriebene Weise einen Migrationsschritt automatisiert direkt im Syntaxbaum vollzieht (siehe Sequenzdiagramm 5-16). Am Ende der Iteration werden alle Klassen, deren Syntaxbaum durch ein `ModificationCommand` modifiziert wurde, über das Repository zurück in die Quelldateien geschrieben.

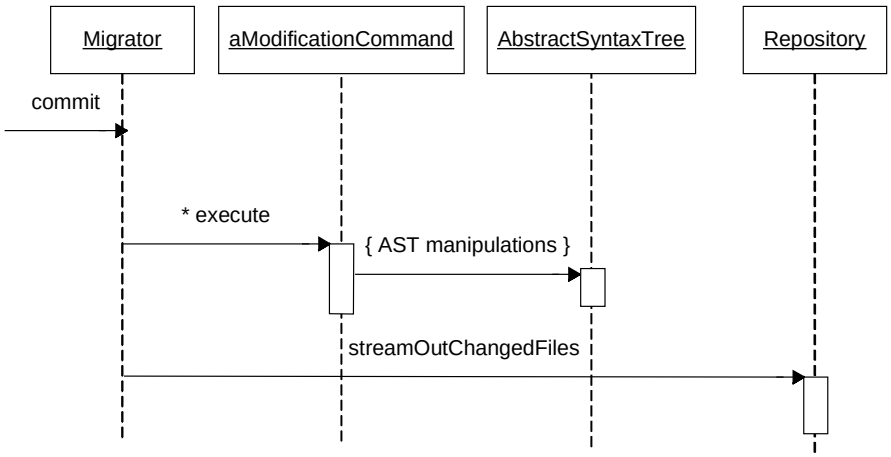


Abbildung 5-16: Sequenzdiagramm des Modifikationsprozesses

- Für nicht automatisiert durchführbare Migrationsschritte wird durch Doppelklick auf einen `ToDoEntry` der im oberen Werkzeugbereich spezifizierte Editor gestartet und der Cursor an der zu modifizierenden Stelle platziert. Hat der Benutzer die Stelle manuell gemäß des durch den `ToDoEntry` beschriebenen Migrationshinweises angepasst, kann der selektierte `ToDoEntry` über den Schaltknopf „Set manually handled“ als bearbeitet markiert werden.

Um die genaueren Bedingungen einer Rahmenwerksmodifikation analysieren zu können, kann durch Doppelklick auf die Past- und Current-Definitionen im mittleren Werkzeugbereich der Quelltext der selektierten Past- oder Current-Rahmenwerksdefinition angezeigt werden.

Sind alle ToDoEntries entweder automatisiert oder manuell als bearbeitet markiert, so ist der Migrationsprozess der Anwendung abgeschlossen. Die Anwendungsklassen sind jetzt auf die Verwendung der neuen Rahmenwerksversion umgestellt.

Als Ergebnis wird festgehalten:

Das Sourcecodemanagement, die Migrationsanalyse-Schicht und das Migrationswerkzeug erfüllen alle Punkte der in Abschnitt 5.1 spezifizierten Anforderungen.

5.4.1. Anwendung auf das JWAMMicro-Beispiel

Um die Praktikabilität des Migrationswerkzeuges zu untersuchen, wurde das JWAMMicro-Beispiel aufgegriffen. Es wurde geprüft, ob die zehn am Beispielrahmenwerk durchgeführten Modifikationen vom Werkzeug erkannt werden, wie die Migration der Anwendungsklassen unterstützt wird und welche Kompatibilitätsklassen die Beispielmodifikationen mit der Werkzeugunterstützung erreichen. Eine detaillierte Darstellung des Migrationsvorganges für die zehn Modifikationen findet sich im Anhang A.2.
Das Ergebnis ist zusammengefasst in der folgenden Tabelle:

Modifikation	**im Beispiel erreichte Kompatibilitätsklasse**
Klasse erzeugen	sicher-kompatibel
Interface oder Klasse umbenennen	automatisierbar
Interface oder Klasse löschen	aufgeschoben potenziell-inkompatibel
Interface oder Klasse auftrennen	automatisierbar
Interface oder Klasse zusammenfassen	semi-automatisierbar
Operation erzeugen	automatisierbar
Operation löschen	aufgeschoben potenziell-inkompatibel
Operation verschieben	potenziell-inkompatibel
Parameterliste einer Operation ändern	automatisierbar
Modifier einer Klasse oder einer Operation ändern	automatisierbar durch Abhängigkeit von Modifikation „Operation erzeugen“

Das Migrationswerkzeug hat alle Modifikationen, die an den Rahmenwerksklassen vorgenommen wurden, erkannt. Alle Stellen in Anwendungsklassen, die von den Modifikationen betroffen sind, wurden von den ModificationAnalyzern des Werkzeuges identifiziert und durch die Migrationshinweise in den ToDo-Einträgen beschrieben.
Sechs der Modifikationen in diesem Beispiel sind entweder sicher-kompatibel oder automatisierbar, der Aufwand für die Anpassung der Anwendungsklassen ist also vernachlässigbar. Eine weitere Modifikation führt zu einem semi-automatisierbaren Migrationsschritt, der auch mit wenig Aufwand durchführbar ist.
Für die verbliebenen drei (aufgeschoben-) potenziell-inkompatiblen Modifikationen leitet das Migrationswerkzeug den mit der Migration betrauten Anwendungsentwickler, indem es auf alle anzupassenden Stellen in den Anwendungsklassen mit einer Erklärung der an der referenzierten Definition vorgenommenen Modifikation deutet. Durch einen Doppelklick kann aus dem Werkzeug an die anzupassende Stelle im Editor navigiert werden. Auch die alte oder modifizierte Definition kann bequem durch Doppelklick eingesehen werden.

Ergebnis:

Der Migrationsprozess konnte für dieses Beispiel erfolgreich durch das Migrationswerkzeug unterstützt und beschleunigt werden.

5.5. Zusammenfassung

In diesem Kapitel wurde die Konstruktion eines Migrationswerkzeuges beschrieben. Das Werkzeug ermittelt aus einem mit Historisierungs-Tags annotierten Rahmenwerk die am Rahmenwerk vorgenommenen Modifikationen. Für jede dieser Modifikationen analysiert das Migrationswerkzeug die Auswirkungen auf Anwendungsklassen und erstellt Migrationshinweise und Migrationsanleitungen für daraus resultierende Migrationsschritte. Die Funktionalität des Werkzeuges ist in drei Schichten organisiert.
Die Sourcecodemanagement-Schicht stellt Basisdienste zur Quellcodeanalyse bereit. Das Repository als zentrale Verwaltungsinstanz ermöglicht den Zugriff auf Symbolinformationen und erlaubt die Abfrage von Vorgänger- und Nachfolgerversionen von Symbolen.
Die Migrationsanalyse-Schicht identifiziert Rahmenwerks-Modifikationen und führt darauf basierend eine Auswirkungsanalyse durch. Die Analysemechanismen sind in Visitor-Klassen gekapselt. Die Visitor-Klassen generieren Migrationshinweise für die Stellen im Quellcode, die im Modifikationsprozess angepasst werden müssen. Für automatisiert durchführbare Migrationsschritte werden Migrationsanleitungen in Command-Objekten formuliert.
Über die Oberfläche des Migrationswerkzeuges kann der Migrationsprozess konfiguriert und gesteuert werden. Die ermittelten Migrationshinweise werden visualisiert, und automatisiert durchführbare Schritte werden zur Auswahl bereitgestellt.
Anhand des JWAMMicro-Beispieles wurde demonstriert, dass viele der eigentlich inkompatiblen Modifikationen mit dem Werkzeug automatisierbar sind und der Migrationsprozess dadurch erheblich beschleunigt werden kann.
Bezogen auf die Werkzeuge verwandter Forschungsfelder kann festgehalten werden, dass das Migrationswerkzeug Aspekte verschiedener Werkzeuge aufgreift und erweitert.
Das Migrationswerkzeug kann durch Auswertung der Historisierungs-Tags zu einem beliebigen Zeitpunkt eine nachträgliche Analyse des Ripple-Effektes durchführen. Die durchgeführten Modifikationen können über die Historisierungs-Tags ermittelt werden, die Analyse der Modifikationsauswirkungen wird über die Visitorklassen realisiert.
Während bei den Software-Transformations-Werkzeugen die Transformationsregeln manuell programmiert werden müssen, werden die Regeln für die Migration vom Migrationswerkzeug aus den ermittelten Modifikationen dynamisch generiert.
Der gesamte Migrationsprozess kann interpretiert werden als ein Mergen der alten Anwendung mit der neuen Rahmenwerksversion. Durch syntaktisches Mergen wird die Verwendung der kompletten Rahmenwerksschnittstelle in die Anwendungsklassen eingepflegt, was einen großen Fortschritt gegenüber dem morphologischen Mergen einzelner Dateien bei den Werkzeugen der SCM-Systeme darstellt.

6. Zusammenfassung und Ausblick

Ich habe in dieser Arbeit untersucht, wie sich der Aufwand für die Migration einer Anwendung auf eine neue Rahmenwerksversion reduzieren lässt und wie ein Werkzeug für die Migrationsunterstützung konstruiert werden kann.
Ich habe dafür den Rahmenwerks-Grundbegriff definiert und den konkreten Kontext geschildert, in dem die Problematik der Anwendungsmigration entstanden ist: Das JWAM-Rahmenwerk ist die Basis verschiedener, im praktischen Einsatz befindlicher Software-Systeme, die nach den Metaphern des WAM-Ansatzes entwickelt wurden. Veränderungen am JWAM-Rahmenwerk führen zu Strukturkonflikten mit den darauf basierenden Anwendungen. Ich habe einen kurzen Einblick in den WAM-Ansatz gegeben und die grobe Struktur des JWAM-Rahmenwerkes skizziert. Um die folgenden Untersuchungen der Arbeit übersichtlicher darstellen zu können, habe ich einen Rahmenwerkskern, das JWAMMicro-Rahmenwerk, definiert. Ich habe dargestellt, welche Modifikationen an einem Rahmenwerk durchführbar sind. Anhand des JWAMMicro-Rahmenwerkes und einer darauf basierenden Beispielanwendung habe ich analysiert, wie sich Rahmenwerksänderungen auf davon abhängige Anwendungen auswirken: Nahezu alle Modifikationen führen zu Strukturkonflikten mit den rahmenwerksbasierten Anwendungen und erzeugen Migrationsaufwand. Ziel der folgenden Untersuchungen war, den Strukturkonflikt zwischen Anwendung und neuer Rahmenwerksversion aufzulösen.

Ich habe betrachtet, mit welchen Werkzeugen und Strategien in verwandten Bereichen die aus der Veränderung von Strukuren resultierenden Probleme angegangen werden.
Im Bereich der Change-Impact-Analysis kommen Werkzeuge zur Analyse des Ripple-Effektes zum Einsatz, mit denen die Auswirkungen von Veränderungen vor der Durchführung analysiert werden können. Die beschriebenen Werkzeuge und Techniken benötigen im Moment der Veränderung Zugriff auf den gesamten Quellcode. Im Falle eines veröffentlichten Rahmenwerkes sind aber zum Modifikationszeitpunkt die davon abhängigen Anwendungen nicht verfügbar.
Auf dem Gebiet der Software-Configuration-Management–Systeme sind Merge-Strategien beschrieben, mit denen das nachträgliche Einpflegen von geänderten Zweigen möglich ist. Die Unterstützung für Merge-Vorgänge durch Werkzeuge ist bisher aber gering. Merge-Werkzeuge arbeiten meist auf morphologischer und nicht auf syntaktischer Ebene. Strukturelle Abhängigkeiten zwischen einem Rahmenwerk und einer Anwendung werden nicht berücksichtigt.
Software-Transformations-Werkzeuge arbeiten dagegen auf einer syntaktischen Basis. Quelldateien werden nach statisch im Werkzeug kodierten Regeln auf Zielplattformen oder Zielsprachen transformiert. Abhängigkeiten zwischen den Dateien werden dabei nicht berücksichtigt. Die auf diesem Gebiet beschriebenen Lösungen lassen sich deshalb nicht auf den Kontext eines sich stetig weiterentwickelnden Rahmenwerkes mit davon abhängenden Anwendungsklassen übertragen.
Für die Migration von Legacy- und Datenbanksystemen gibt es verschiedene spezialisierte Transformationswerkzeuge. Für den aufwändigen Migrationsprozess haben sich inkrementelle Verfahren in der Praxis bewährt. Über den Einsatz von Adaptern können die Systeme in kleinen Schritten migriert werden.
Keines der untersuchten Werkzeuge bietet eine ausreichende Unterstützung für die Migration von Anwendungen auf neue Rahmenwerksversionen. Zwei Strategien haben sich aber in den verwandten Gebieten bewährt: die Anpassung von Quelldateien über eine syntaktische Transformation und der inkrementelle Migrationsprozess über den Einsatz einer Vermittlerschicht.

Im Anschluss habe ich geprüft, ob der bewährte Ansatz der inkrementellen Migrationsunterstützung auf die Problematik der rahmenwerksbasierten Anwendungsmigration übertragbar ist.
Dazu müsste eine Adapterschicht ausgeliefert werden, die als Vermittler zwischen Anwendung und neuer Rahmenwerksversion dient und eine sanfte, schrittweise Migration der Anwendung ermöglicht. Die Adapterschicht präsentiert sich gegenüber der Anwendung mit der Schnittstelle der alten Rahmenwerksversion, verwendet intern aber die neue Rahmenwerksversion. Über die Adapterschicht kann die Anwendung sofort die neue Rahmenwerksversion verwenden. In kleinen Migrationsschritten können Anwendungsentwickler die Verwendung der Adapterschicht durch die direkte Verwendung der neuen Rahmenwerksversion ersetzen.
Das von Gamma et al. definierte Adaptermuster ist für die Kapselung einer einzelnen Klasse beschrieben. Ich habe untersucht, ob das Muster ausgedehnt werden kann auf die Kapselung einer ganzen Rahmenwerksschnittstelle:
Eine Adapterschicht unter Verwendung von Objektadaptern kann grundsätzlich für Whitebox-Rahmenwerke nicht konstruiert werden. Objektadapter verwalten Exemplare von Rahmenwerksklassen. Sie leiten Aufrufe von der Anwendung an die gekapselten Rahmenwerksobjekte weiter. Ein umgekehrter Kontrollfluss aus dem gekapselten Rahmenwerk an überschriebene Methoden in den Anwendungsklassen ist mit einem Objektadapter nicht realisierbar, da die gekapselten Rahmenwerksobjekte die Anwendungs- und Adapterklassenobjekte nicht kennen.
Wird die Adapterschicht über Klassenadapter gebildet, so müssen die Vererbungsbeziehungen der ursprünglichen Rahmenwerksversion in den Klassenadaptern erhalten bleiben. Zusätzlich müssen die Klassenadapter von der zu adaptierenden neuen Rahmenwerksversion erben. Das daraus resultierende Problem der Mehrfachvererbung lässt sich über die Bereitstellung einer parallelen Interface-Hierarchie umgehen. Zusammengefasst sind aber die bei der Rahmenwerks- und Anwendungskonstruktion einzuhaltenden Regeln so einschränkend, dass auch die Klassenadapter-Lösung für den Kontext von JWAM- und JWAMMicro-Anwendungen nicht sinnvoll verwendbar ist.

Ich habe anschließend die werkzeuggestützte Anpassung der rahmenwerksabhängigen Anwendungsklassen untersucht. Damit ein Werkzeug nötige Migrationsschritte erkennen und durchführen kann, müssen die am Rahmenwerk vollzogenen Modifikationen für das Werkzeug lesbar dokumentiert sein.
Ich habe eine Möglichkeit der Modifikationsdokumentation direkt in den Rahmenwerksklassen über Tags dargestellt und gezeigt, dass sich mittels der Past-, Future-, Since-, Default- und Paramdef-Tags alle Basismodifikationen dokumentieren lassen. Ich habe das JWAMMicro-Beispiel aufgegriffen und anhand der daran durchgeführten Modifikationen demonstriert, wie die an Anforderungen aus der Praxis angelehnten, zusammengesetzten Modifikationen dokumentiert werden können.
Nachfolgend habe ich dargelegt, wie ein Migrationswerkzeug konstruiert werden kann. Das Werkzeug muss Rahmenwerks- und Anwendungscode einlesen können. Es muss die Modifikationen am Rahmenwerk erkennen können und für die Modifikationen eine nachträgliche Analyse des Ripple-Effektes durchführen. Darüber können die Stellen in den Anwendungsklassen identifiziert werden, die an die neue Rahmenwerksstruktur angepasst werden müssen. Das Werkzeug muss den Anwendungsentwickler mit Migrationshinweisen auf diese Stellen aufmerksam machen und gegebenenfalls durch automatisiert durchführbare Migrationsschritte unterstützen.
Diese Aufgaben werden von den Schichten des Sourcecodemanagements, der Migrationsanalyse-Schicht und des Migrationswerkzeuges erfüllt. Ich habe erläutert, wie die Basismodifikationen durch die `ModificationAnalyzer`-Klassen erkannt werden und wie Migrationsanleitungen für automatisiert durchführbare Migrationsschritte in `ModificationCommand`-

Klassen gekapselt werden. Ich habe die Struktur des Migrationswerkzeuges beschrieben, das den Anwender durch den Migrationsprozess führt.
Abschließend habe ich das Beispiel des JWAMMicro-Rahmenwerkes aufgegriffen und die mit dem Migrationswerkzeug erzielten Ergebnisse für die Migration der Beispielanwendung dargestellt. Das Werkzeug hat alle Rahmenwerksmodifikationen und deren Auswirkungen erkannt. Viele der notwendigen Migrationsschritte sind automatisiert durchführbar, bei den verbleibenden manuell durchzuführenden Modifikationsschritten wird der Anwendungsentwickler vom Werkzeug geleitet. Am Ende des Migrationsprozesses ist die Beispielanwendung auf die Verwendung des JWAMMicro-Rahmenwerkes in der Version 2 migriert.
Verglichen mit den Werkzeugen aus den verwandten Forschungsbereichen sind folgende Erweiterungen bemerkenswert:

- Das Migrationswerkzeug ermöglicht das syntaktische Mergen ganzer getrennt weiterentwickelter Rahmenwerkszweige.
- Für ein mit Historisierungs-Tags ausgestattetes Rahmenwerk kann eine nachträgliche Analyse des Ripple-Effektes durchgeführt werden.
- Die Regeln zur Transformation von Anwendungsklassen müssen nicht statisch kodiert werden, sondern können dynamisch durch Analyse der Tags gewonnen werden.

Das in dieser Arbeit entwickelte Migrationswerkzeug bietet eine gute Grundlage für die Untersuchung weitergehender Migrationsautomatisierungen. Bisher werden alle Modifikationen erkannt und die Verwendungsstellen in Anwendungsklassen identifiziert. Manche der bisher potenziell inkompatiblen Modifikationen ließen sich aber über weitere Analysen automatisiert in den Anwendungsklassen anpassen. Als Beispiel seien hier nur die Verschiebung von Klassenmethoden oder das Entfernen von „final“-Modifiern genannt, die bei der Analyse als potenziell inkompatible Methodenverschiebungs- und Modifieränderungs-Modifikationen erkannt werden, bei weiterer Untersuchung aber durchaus zu automatisiert durchführbaren Migrationsschritten führen könnten.
Ein weiterer Bereich, der in diesem Kontext besser unterstützt werden könnte, ist die Pflege der Historisierungs-Tags. Die Past-Tags werden, wie beschrieben, automatisiert nach einem Release erzeugt. Die weitere Pflege anderer Historisierungs-Tags muss aber vom Rahmenwerksentwickler manuell durchgeführt werden. Diese bisher fehleranfällige Tätigkeit könnte durch entsprechend erweiterte Entwicklungsumgebungen unterstützt werden.

7. Literaturverzeichnis

[ANTLR 02] *The ANTLR Translator Generator*: www.antlr.org. 2002.

[Beck 00] K. Beck: *eXtreme Programming explained: Embrace Change*. Reading, Massachusetts: Addison-Wesley, 2000.

[Bersoff et al. 80] E. Bersoff, V. Henderson, S. Siegel: *Software Configuration Management, An Investment in Product Integrity.* Englewood Cliffs, New Jersey: Prentice-Hall, 1980.

[Booch et al. 98] G. Booch, I. Jacobson, J. Rumbaugh: *The Unified Modeling Language User Guide*. Reading, Massachusetts: Addison Wesley, 1998.

[Brodie & Stonebraker 93] M. Brodie, M. Stonebraker: *DARWIN: On the Incremental Migration of Legacy Information Systems*. Technical Report TR-022-10-92-165 GTE Labs Inc., März 1993.

[Brodie & Stonebraker 95] M. Brodie, M. Stonebraker: *Migrating Legacy Systems: Gateways, Interfaces and the Incremental Approach*. Morgan-Kaufman Publishers, 1995.

[Chyfo 02] *Chyfo*. Ispirer Systems: http://www.ispirer.com/products. 2002.

[Conradi & Westfechtel 99] R. Conradi, B. Westfechtel: *SCM: Status and Future Challenges.* Proceedings 9th International Workshop on System Configuration Management, Toulouse, LNCS 1675, Springer-Verlag, 1999, pp. 228-231.

[Deutsch 89] L. P. Deutsch: *Design reuse and frameworks in the Smalltalk-80 system*. In: Ted J. Biggerstaff und Alan J. Perlis, Hrsg., *Software Reusability, Volume II: Applications and Experience*. Reading, Massachusetts: Addison Wesley, 1989, pp. 57-71.

[Earles 00] J. Earles: *Framework Evolution! One Box, Two Box, White Box, Black Box*. Online-Article: http://www.cbd-hq.com/PDFs/cbdhq_000401je_frameworks2.pdf. 2000.

[Feldman et al. 93] S. Feldman, D. Gay, M. Maimone, N. Schryer: *A Fortran to C Converter.* AT&T Technical Report No. 149, 1993.

[Foote 92] B. Foote: *A fractal model of the lifecycle of reusable object*s. OOPSLA'92 Workshop on Reuse, Vancouver, British Columbia, Canada. Oktober 1992.

[Fowler 99] M. Fowler: *Refactoring: Improving the Design of Existing Code*. Reading, Massachusetts: Addison-Wesley, 1999.

[Frühauf & Zeller 99] K. Frühauf, A. Zeller: *Software Configuration Management: State of the Art, State of the Practice.* Proc. 9th International Symposium on System Configuration Management (SCM-9), Toulouse, France, Vol. 1675 of LNCS, September 1999, pp. 217-227.

[Gamma et al. 95] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Design Patterns. Elements of Reusable Object-Oriented Softwar*e. Addison-Wesley, 1995.

[Gamma et al. 96] E. Gamma, R. Helm, R. Johnson, J. Vlissides: *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Softwar*e. Übersetzung von D. Riehle, Bonn: Addison Wesley, 1996. (Übersetzung von [Gamma et al. 95].)

[Jacobson et al. 99] Ivar Jacobson, Grady Booch, James Rumbaugh: *The Unified Software Development Process*. Reading, Massachusetts: Addison Wesley Publishing, Januar 1999.

[JavaCC 02] *Java Compiler Compiler:* www.webgain.com/products/java_cc, 2002.

[Johnson & Foote 88] R. Johnson, B. Foote: *Designing Reusable Classes.* The Journal of Object-Oriented Programming, 1 (2), June/July, 1988, pp. 22-35.

[JWAM 02] *The JWAM-Framework*: www.jwam.org. 2002.

[King et al. 97] R. King, M. Novak, and C. Och: *Sybil: Supporting heterogeneous database interoperability with lightweight alliances.* In: The Third International Workshop on Next Generation Information Technologies and Systems, 1997.

[Kontogiannis et al. 98] K. Kontogiannis, J. Martin, K. Wong, R. Gregory, H. Muller, and J. Mylopoulos: *Code Migration Through Transformations: An Experience Report*. In: Proceedings of CASCON '98, Toronto, ON, 1998, pp. 1-13.

[Lippert et al. 02] M. Lippert, S. Roock, H. Wolf: *Software entwickeln mit eXtreme Programming.* Heidelberg: dpunkt.verlag, 2002.

[Meyer 91] B. Meyer: *Design by Contract*. In: D. Mandrioli, B. Meyer (Hrsg.): *Advances in Object-Oriented Software Engineerin*g. New York, London: Prentice-Hall, 1991, pp. 1-50.

[Meyer 97] B. Meyer: *Object-Oriented Software Construction*. New York, London: Prentice-Hall, Second Edition, 1997.

[Morgenthaler 97] J.D. Morgenthaler: *Static Analysis for a Software Transformation Tool*, Ph.D. thesis, University of California, San Diego, 1997.

[OMW 02] *Oracle Migration Workbench:*
http://otn.oracle.com/tech/migration/workbench/content.html. 2002.

[Opdyke 92] W.F. Opdyke: *Refactoring Object-Oriented Framework*s. Ph.D. thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1992.

[Rajlich 97] V. Rajlich: *A Model for Change Propagation Based on Graph Rewriting*. Proc of ICSM'97, 1997.

[Roock 02] S. Roock: *Extreme Frameworking: How to Aim Applications at Evolving Frameworks.* In: G. Succi und M. Marchesi, Hrsg., *Extreme Programming Examined*, Addison-Wesley, 2001, pp. 71-82.

[Roock 03] S. Roock: *Werkzeuggestützte Anwendungsmigration auf neue Verwendungseinheitenversionen.* Dissertation. Universität Hamburg, Fachbereich Informatik, Arbeitsbereich Softwaretechnik, 2003 (geplant).

[Stevens et al. 98] P. Stevens and R. Pooley: *Software Reengineering Patterns*. Proceedings of SIGSOFT'98 6th International Symposium on Foundations of Software Engineering, 1998.

[Sun J2SE 02] *Java 2 Platform, Standard Edition*, http://java.sun.com/j2se. 2002.

[Sun J2EE 02] *Java 2 Platform, Enterprise Edition*, http://java.sun.com/j2ee. 2002.

[Terekhov & Verhoef 00] A.A. Terekov, C. Verhoef: *The realities of language conversion*. IEEE Software 2000; 17(6): S.111–124. 2000.

[Transmogrify 01] *Transmogrify*: http://transmogrify.sourceforge.net. 2001.

[Vahdat & Anderson 98] A. Vahdat and T. Anderson: *Transparent Result Caching*. In: *Proceedings of the USENIX 1998 Annual Technical Conference*, 1998, pp. 25–38.

[Van der Hoek et al. 01] André van der Hoek, Marija Mikic-Rakic, Roshanak Roshandel, and Neno Medvidovic: *Taming Architectural Evolution*. In: Proceedings of the Sixth European Software Engineering Conference (ESEC) and the Ninth ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-9), Vienna, Austria, September 2001.

[Yau et al. 88] S.S. Yau, R.A. Nicholl, J.J. Tsai, S. Liu: *An Integrated Life-Cycle Model for Software Maintenance*. IEEE Trans. Software Engineering, 15(7), 1988, pp. 58-95.

[Züllighoven 98] H. Züllighoven: *Das objektorientierte Konstruktionshandbuch nach dem Werkzeug- und Material-Ansatz*. Heidelberg: dpunkt-Verlag 1998.

A Anhang

A.1 Quellcode der Beispielanwendung

Materialklasse `Account`:

```
package app;

import de.jwammicro.handling.*;
import de.jwammicro.lang.*;

public class Account extends ThingImpl
{
  public void setBalance(double amount){
    Contract.require(amount >= 0);
    _amount = amount;
  }

  public double getBalance(){
    return _amount;
  }
}
```

Werkzeugklasse `AccountTool`:

```
package app;

import de.jwammicro.handling.*;
import de.jwammicro.technology.*;

public class AccountTool extends Tool
{
  public void useMaterial(Thing material){
    _account = (Account)material;
  }

  public void changeBalance(double newBalance){
    _account.setBalance(newBalance);
  }

  public void doStop(){
    PersistencyManager persMng = new RDBMSPersistencyManager();
    persMng.store(_account);
    persMng.commit();
  }

  private Account _account;
}
```

Die Methoden `useMaterial` und `doStop` sind in der Oberklasse definiert und werden in der Werkzeugklasse redefiniert. Das `AccountTool`-Werkzeug wird mit diesen spezialisierten Methoden-Redefinitionen in das Rahmenwerksprotokoll eingebunden.

Klasse `Startup` zum Initialisieren der Anwendung:

```
package app;

import de.jwammicro.handling.*;
import de.jwammicro.technology.*;

public class Startup
{
  public static void main(String[] args)
  {
    Environment env = new Environment();
    env.log("Starting!");
    env.registerTool(new AccountTool(), Account.class);
    RDBMSPersistencyManager persMng =
           new RDBMSPersistencyManager();
    Account account = (Account)persMng.retrieve(
                            "Account Andreas Havenstein");
    env.startToolForMaterial(account);
  }
}
```

A.2 Migration der JWAMMicro-Beispielanwendung

In diesem Abschnitt wird detailliert die Anwendung des Migrationswerkzeuges demonstriert. Dazu wird das JWAMMicro-Beispiel aufgegriffen. Es wird untersucht, ob die zehn am Beispielrahmenwerk durchgeführten Modifikationen vom Werkzeug erkannt werden, wie die Migration der Anwendungsklassen unterstützt wird und welche Kompatibilitätsklassen die Beispielmodifikationen mit der Werkzeugunterstützung erreichen.
Dazu werden die Modifikationen benannt und jeweils der Ausschnitt des Werkzeuges abgebildet, der die relevanten Informationen der Modifikation und Migrationsunterstützung enthält. Die an der Erkennung der jeweiligen Modifikation beteiligten ModificationAnalyzer und die erreichte Kompatibilitätsklasse werden für jede Modifikation genannt.

Das Werkzeug wird konfiguriert mit der alten Rahmenwerksversion, der neuen Rahmenwerksversion mit Historisierungstags und dem Pfad zu den zu migrierenden Anwendungsklassen. Der Wert des Since-Tags zur Erkennung neuer Klassen und Methoden wird auf die Versionsnummer „2.0“ gesetzt (siehe Abbildung A-1).

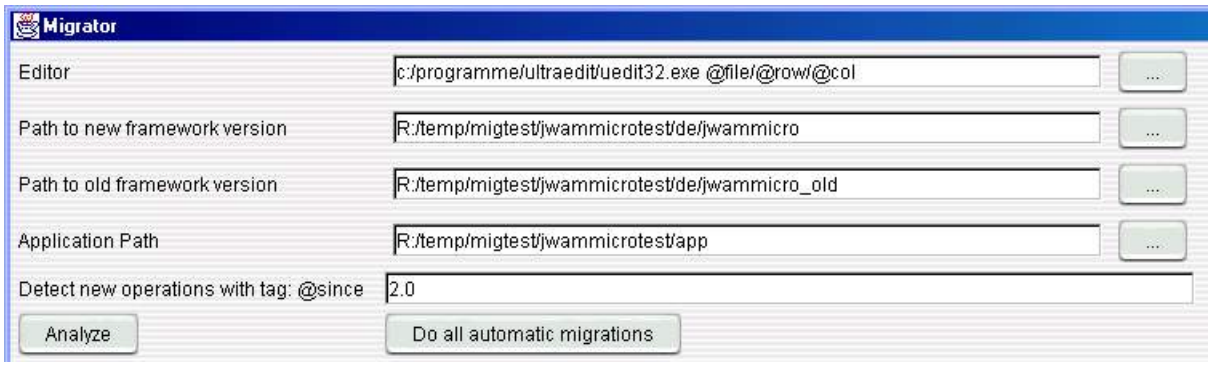


Abbildung A-1: Migrationswerkzeug-Konfiguration

Durch Druck auf den Analyze-Knopf wird das Parsen und die Analyse gestartet. Als Ergebnis erscheint eine Liste mit insgesamt 16 ToDoEntries. Im Folgenden werden die ToDoEntries den Modifikationen zugeordnet. Dazu wird der Ausschnitt des Migrationswerkzeuges abgebildet, der die ToDoEntry-Informationen der jeweiligen Modifikation enthält. ToDoEntries anderer Modifikationen sind auf den Abbildungen jeweils ausgeblendet.

A.2.1 Modifikation: Klasse erzeugen

Beispiel: Die Klasse `Logger` wurde dem Rahmenwerk hinzugefügt.
Die sicher-kompatible Modifikation wird vom Werkzeug über das Since-Tag der Klassendefinition erkannt, aber nicht weiter berücksichtigt. Sicher-kompatible Modifikationen haben keine Auswirkung auf die auf der alten Rahmenwerksversion basierenden Anwendungsklassen.

A.2.2 Modifikation: Interface oder Klasse umbennenen

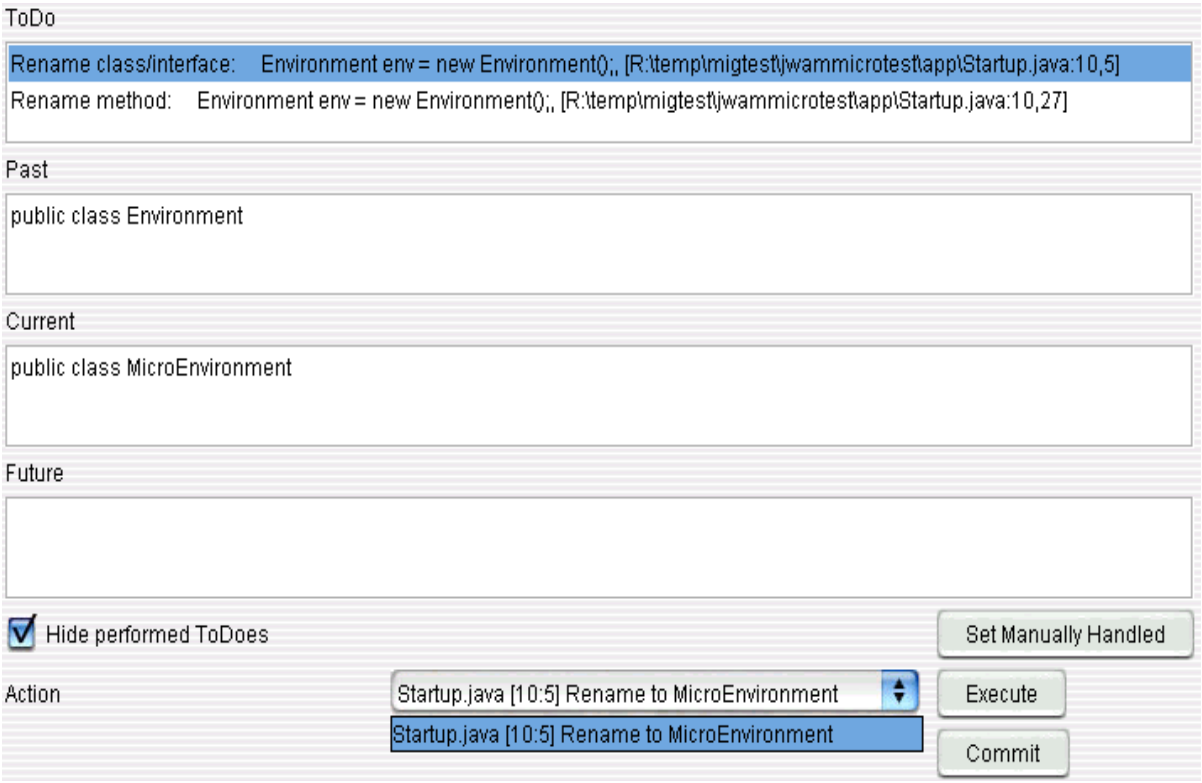


Abbildung A-2: Migrationswerkzeug: Interface oder Klasse umbenennen

Das Werkzeug erkennt zwei Stellen in den Anwendungsklassen, die an den geänderten Klassennamen angepasst werden müssen. An den Quellcode-Ausschnitten der ToDo-Einträge ist erkennbar, dass die Umbenennung der Klasse Environment in MicroEnviroment die Deklaration einer Variablen `env` betrifft und in der gleichen Zeile der Konstruktor-Aufruf auf den neuen Klassennamen angepasst werden muss. Beiden ToDo-Einträgen ist jeweils ein ModificationCommand zur automatisierten Umbenennung der jeweiligen Quellcodestelle zugeordnet. In der Abbildung ist unten unter „Action“ das `ReplaceNameModification`-Command für den selektierten ToDo-Eintrag dargestellt, das automatisiert den alten Klassennamen der Variablendeklaration durch den neuen Namen ersetzen kann. An dem Command ist ablesbar, an welcher Position (Zeile 10, Spalte 5) in welcher Klasse (Startup.java) ein Migrationsschritt (Umbenennung der Quellcodestelle in „MicroEnvironment“) durchzuführen ist.
Durch Betätigung des Execute-Knopfes wird das ModificationCommand in die Liste der durchzuführenden ModificationCommands aufgenommen.

Beteiligte ModificationAnalyzer: `RenameClassAnalyzer`, `RenameMethodAnalyzer`
Erreichte Kompatibilität: Beide Migrationsschritte sind automatisierbar.

A.2.3 Modifikation: Interface oder Klasse löschen

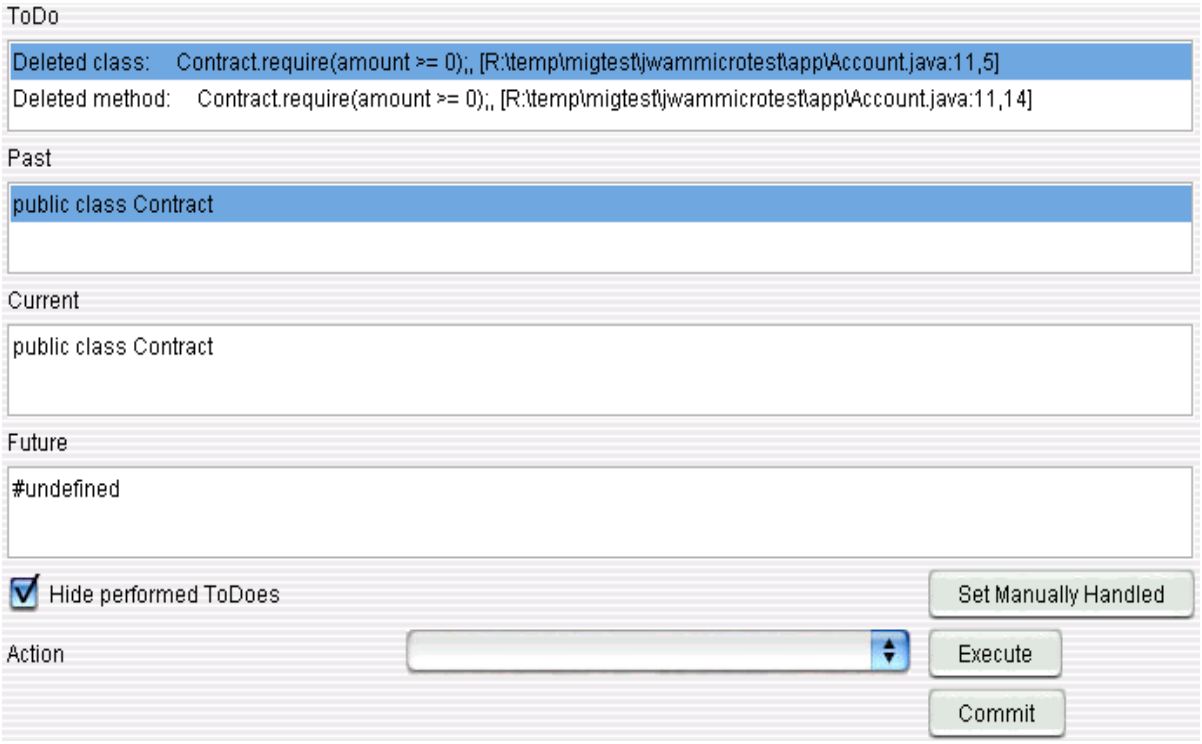


Abbildung A-3: Migrationswerkzeug: Interface oder Klasse löschen

Da in der nächsten Rahmenwerksversion die Klasse Contract nicht mehr vorhanden ist, fehlt dann auch die darin enthaltene Methode `require`. Das Migrationswerkzeug erkennt zwei von der Klassen- und Methodenlöschung betroffene Stellen in den Anwendungsklassen. Beide ToDo-Einträge beziehen sich auf eine Zeile in der Klasse `Account`. Für beide ToDo-Einträge existieren keine ModificationCommands, die Anpassungen in der Anwendungsklasse müssen a e „Current“-Definition ka werden:

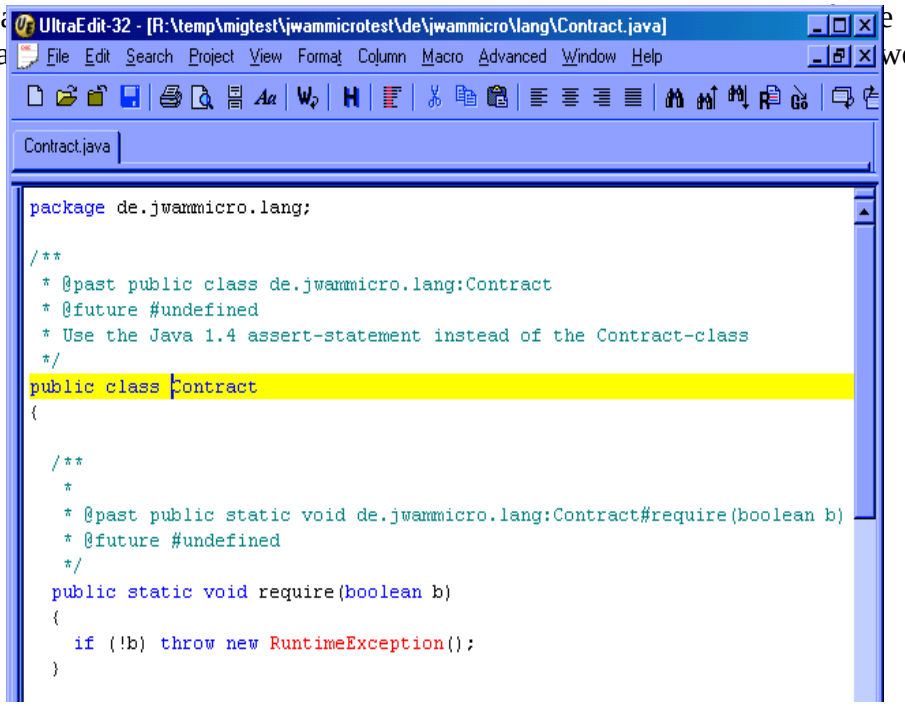


Abbildung A-4: Definitionsanzeige im Editor

Der Cursor wird auf die Definition positioniert. Der Anwender kann im Kommentar der zu löschenden `Contract`-Klasse sehen, dass statt der `Contract`-Klasse die `assert`-Statements der Java-1.4-Sprachspezifikation zu verwenden sind. Durch Doppelklick auf die ToDo-Einträge können dann die entsprechenden Änderungen manuell in der `Account`-Klasse vorgenommen werden.

Beteiligte ModificationAnalyzer: `DeletedEntityAnalyzer`
Erreichte Kompatibilität: aufgeschoben potenziell-inkompatibel

A.2.4 Modifikation: Interface oder Klasse auftrennen

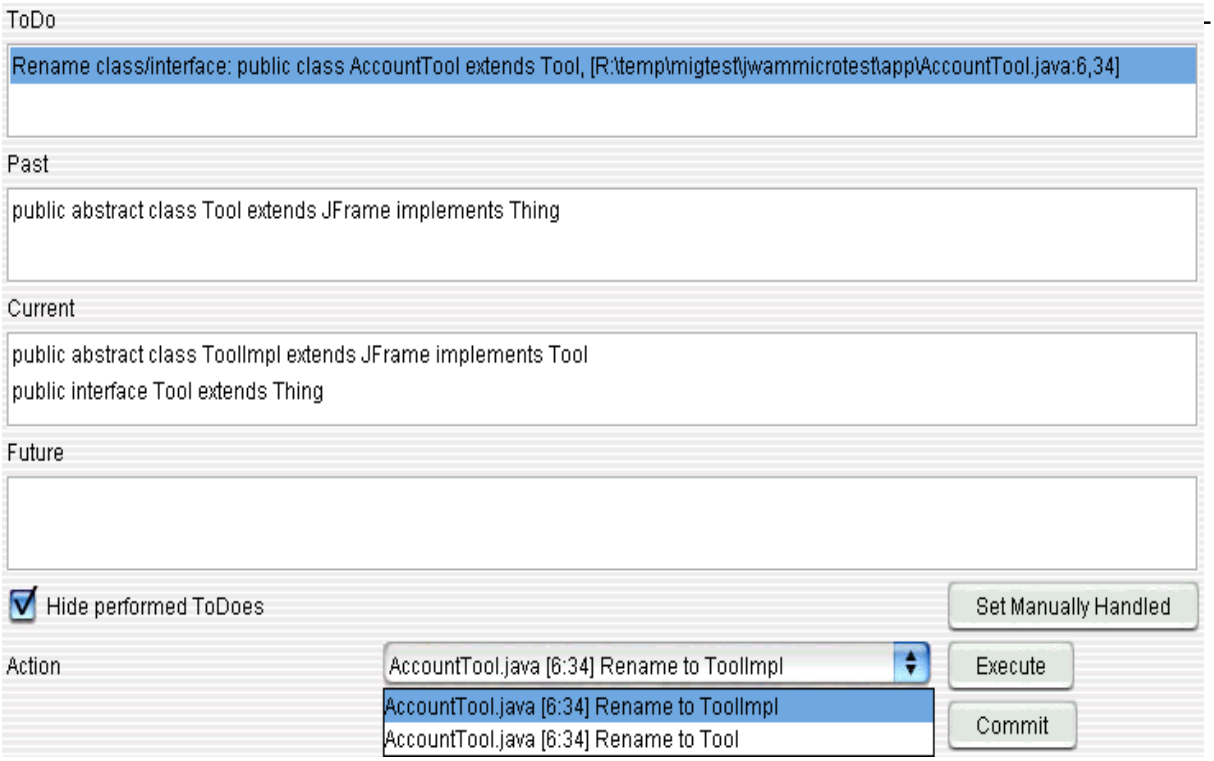


Abbildung A-5: Migrationswerkzeug: Interface oder Klasse auftrennen

Das Werkzeug erkennt die Trennung in `Tool` und `ToolImpl` und identifiziert eine die alte Klasse referenzierende Stelle in den Anwendungsklassen. Bei der Klassendeklaration von `AccountTool` gibt es für die referenzierte Klasse `Tool` zwei Nachfolger im CurrentSpace. Dem ToDo-Eintrag sind deshalb zwei mögliche ModificationCommands zugeordnet. Der Benutzer muss sich manuell für eine der beiden möglichen `ReplaceNameModification`-Commands entscheiden.

Beteiligte ModificationAnalyzer: `RenameClassAnalyzer`
Erreichte Kompatibilität: semi-automatisierbar

A.2.5 Modifikation: Interface oder Klasse zusammenfassen

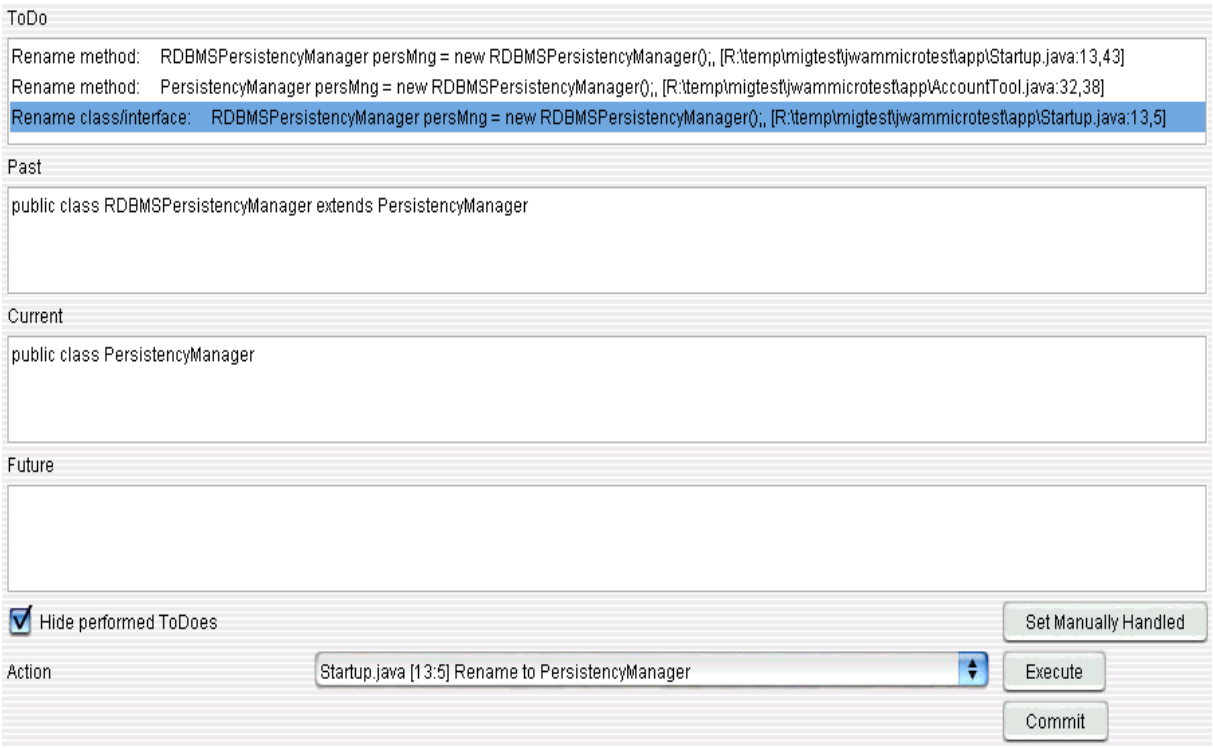


Abbildung A-6: Migrationswerkzeug: Interface oder Klasse zusammenfassen

Das Migrationswerkzeug stellt fest, dass die Klasse `PersistencyManager` eine Verschmelzung der alten `PersistencyManager`- und `RDBMSPersistencyManager`-Klassen ist. Es identifiziert drei anzupassende Stellen in den Anwendungsklassen. Die Klassendeklaration und zwei Konstruktoraufrufe der ursprünglichen `RDBMSPersistencyManager`-Klasse müssen angepasst werden auf die Konstruktor- und Klassennamen der neuen, verschmolzenen Klassenversion.
Jedem ToDoEntry ist genau ein `ReplaceNameModification`-Command zugeordnet. In der Abbildung wird beispielsweise unten das `ModificationCommand` für das RenameClass-ToDo angezeigt.
Alle `ReplaceNameModification`-Commands sind automatisiert ausführbar.

Beteiligte ModificationAnalyzer: `RenameClassAnalyzer`, `RenameMethodAnalyzer`
Erreichte Kompatibilität: automatisierbar

A.2.6 Modifikation: Operation erzeugen

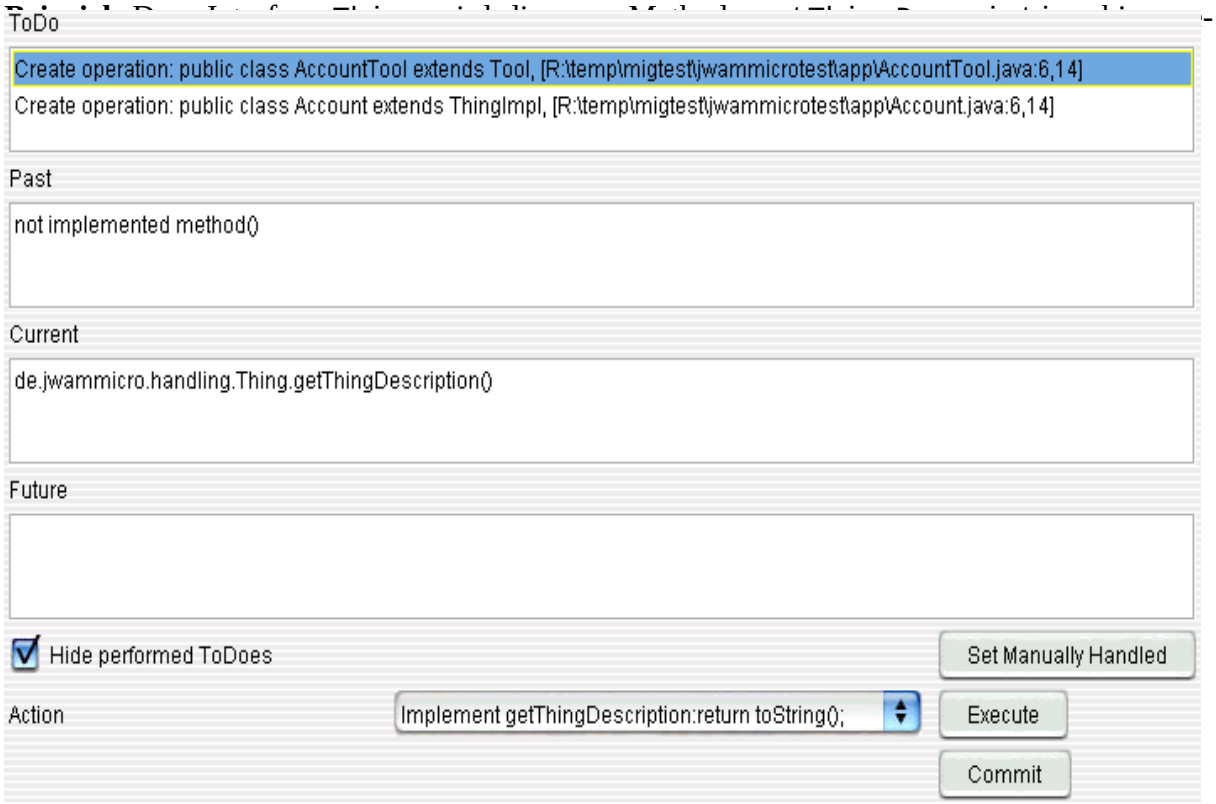


Abbildung A-7: Migrationswerkzeug: Operation erzeugen

Das Werkzeug identifiziert zwei Klassen, in denen die neue Methode implementiert sein muss. Die Anwendungsklasse `Account` erbt von der abstrakten Rahmenwerksklasse `Thing-Impl`, die das modifizierte `Thing`-Interface implementiert. Die Klasse `AccountTool` erbt von der alten `Tool`-Klasse, die ebenfalls das `Thing`-Interface implementiert.
Beiden ToDo-Einträgen ist jeweils ein `CreateOperationModification`-Command zugeordnet, das automatisiert eine neue Methode mit der im Kommentar zur neuen Methodendefinition definierten Standardimplementation erzeugen kann. In der Abbildung ist unten für den ausgewählten ToDo-Eintrag die Vorschau des ModificationCommands mit dem zu implementierenden Methodencode dargestellt.

Beteiligte ModificationAnalyzer: `CreateOperationAnalyzer`
Erreichte Kompatibilität: automatisierbar

A.2.7 Modifikation: Operation löschen

Beispiel: Die `beginTransaction`- und `commit`-Operation in der `PersistencyManager`-Klasse sind entfernt worden.

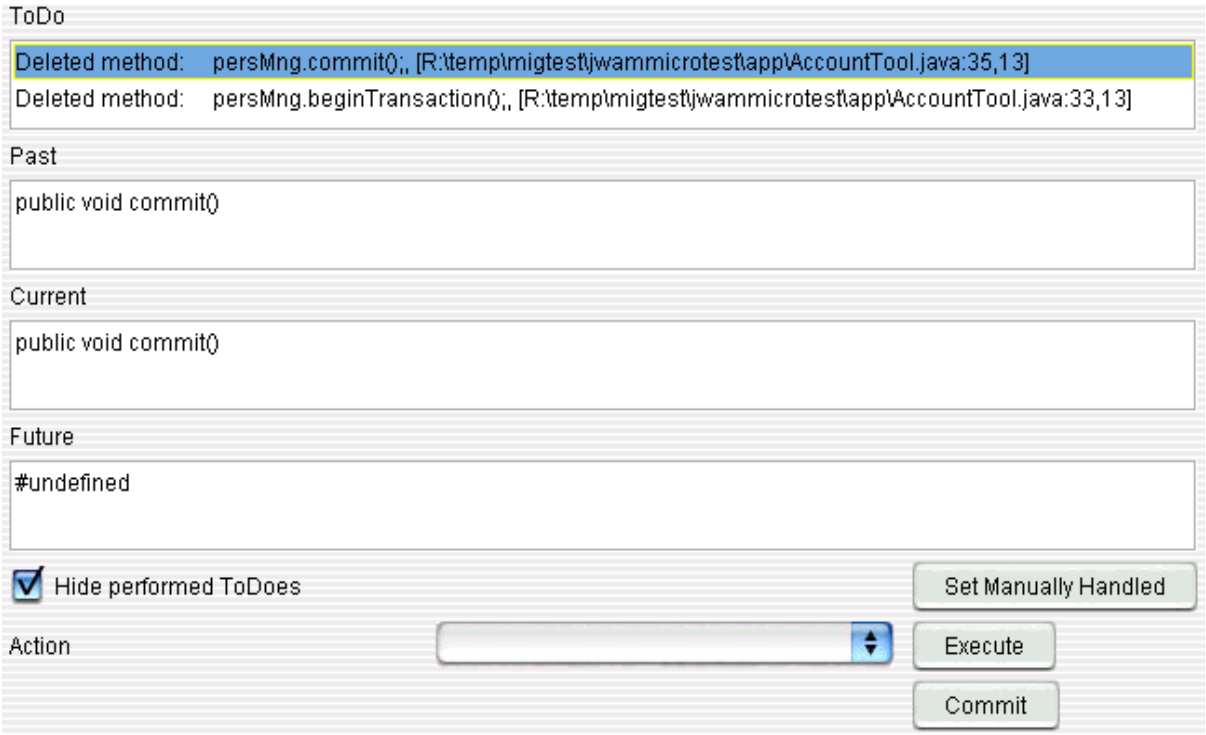


Abbildung A-8: Migrationswerkzeug: Operation löschen

Das Migrationswerkzeug identifiziert zwei Verwendungsstellen der zu löschenden Operationen in den Anwendungsklassen.
Auch hier könnten wieder, wie bei dem Beispiel der zu löschenden `Contract`-Klasse, im Kommentar der zu löschenden Definitionen Erläuterungen der Rahmenwerksentwickler eingesehen werden, aus denen ersichtlich wird, dass in Zukunft ein explizites Transaktionshandling nicht mehr unterstützt wird. Der Anwendungsentwickler muss diese Entscheidung der Rahmenwerksentwickler manuell an den identifizierten Stellen umsetzen.

Beteiligte ModificationAnalyzer: `DeletedEntityAnalyzer`
Erreichte Kompatibilität: aufgeschoben potenziell-inkompatibel

A.2.8 Modifikation: Operation verschieben

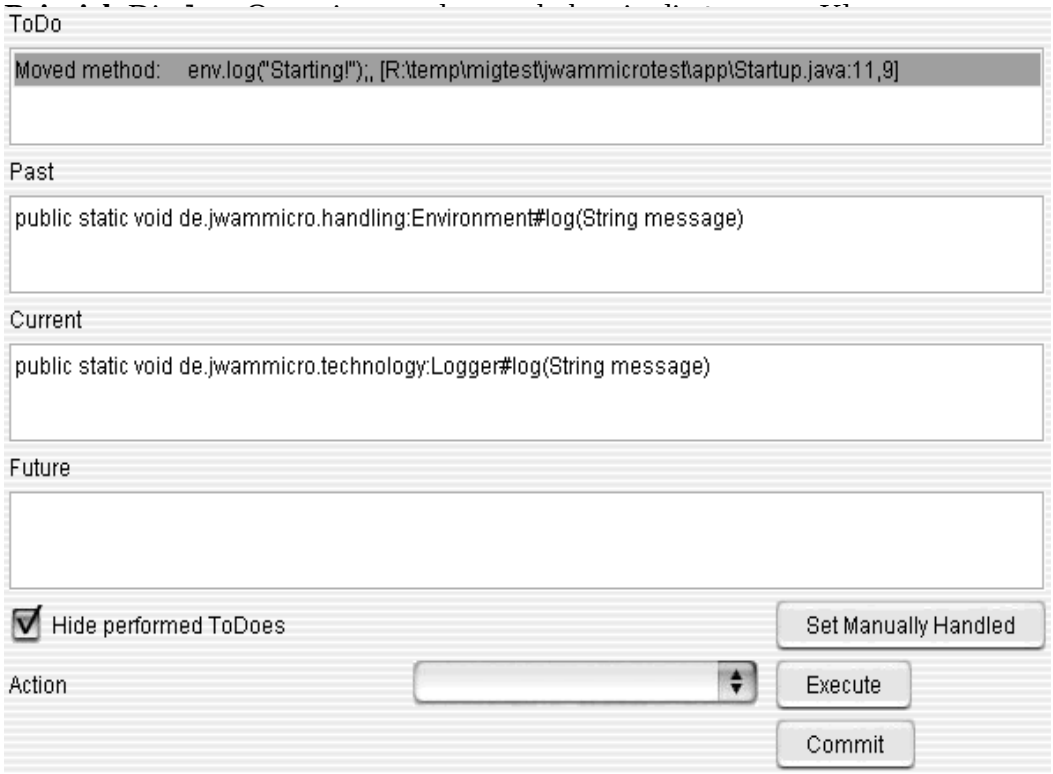


Abbildung A-9: Migrationswerkzeug: Operation verschieben

Das Werkzeug erkennt die verschobene `log`-Operation und identifiziert eine Verwendungsstelle in den Anwendungsklassen. Das Werkzeug zeigt dem Benutzer den voll qualifizierten Kontext der alten und neuen Methodendefinition der in der Anwendungsklasse referenzierten Operation an.
Da es sich um eine Klassenmethode handelt, wäre prinzipiell eine automatisierte Anpassung der Verwendungsstelle möglich. In der derzeitigen Version des Migrationswerkzeuges werden aber noch keine weiteren Kontextanalysen durchgeführt. Der Werkzeuganwender muss die identifizierte Stelle manuell anpassen.

Beteiligte ModificationAnalyzer: `MovedEntityAnalyzer`
Erreichte Kompatibilität: potenziell-inkompatibel

A.2.9 Modifikation: Parameterliste einer Operation ändern

Beispiel: Die Operation `startToolForMaterial` in der `Environment` Klasse wurde um

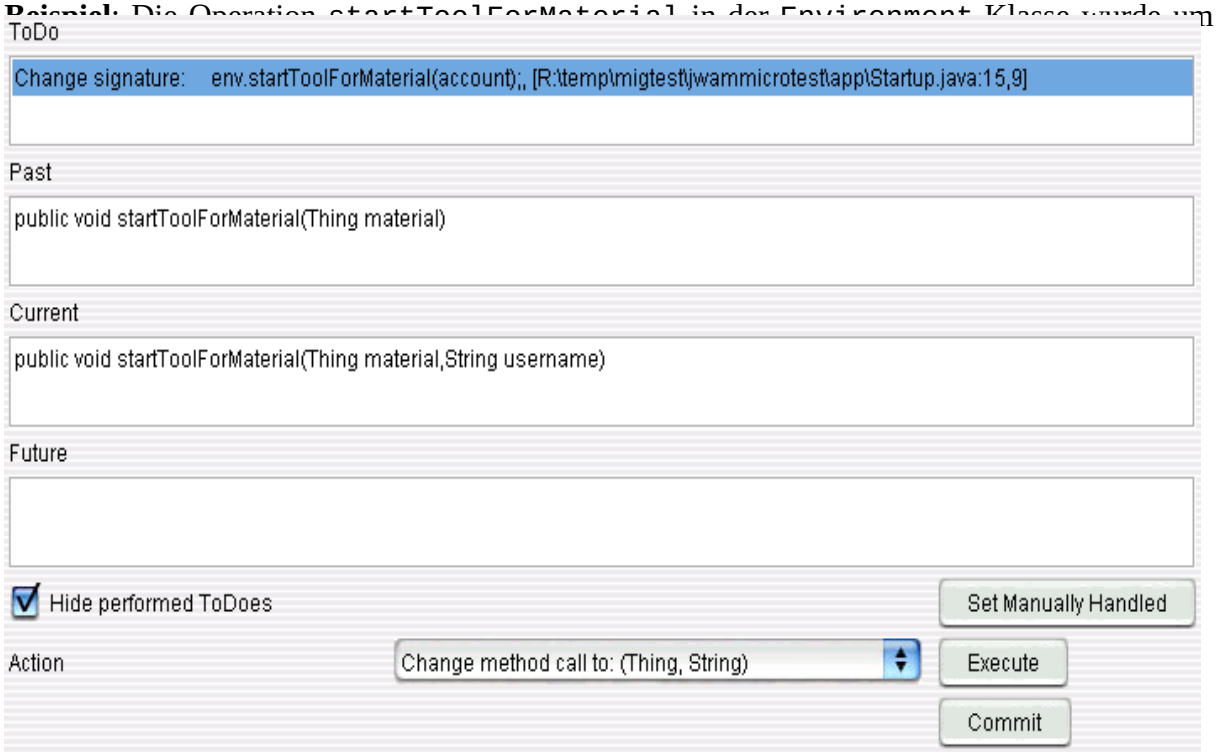


Abbildung A-10: Migratorwerkzeug: Parameterliste ändern

Das Migratorwerkzeug erkennt eine Stelle im Anwendungscode, an der die Operation `startToolForMaterial` mit der alten Signatur referenziert wird. Da für den in der neuen Methodendefinition hinzugekommenen Parameter in dem Kommentar der Definition ein Defaultwert über das Paramdef-Tag angegeben wurde, kann der Aufruf automatisiert an die neue Signatur angepasst werden. Dem ToDo-Eintrag wurde ein `ChangeParameterModification`-Command zugeordnet, das die automatisierte Anpassung des Aufrufs durchführen kann.

Beteiligte ModificationAnalyzer: `ChangeOperationSignatureAnalyzer`
Erreichte Kompatibilität: automatisierbar

A.2.10 Modifikation: Modifier einer Klasse oder einer Operation ändern

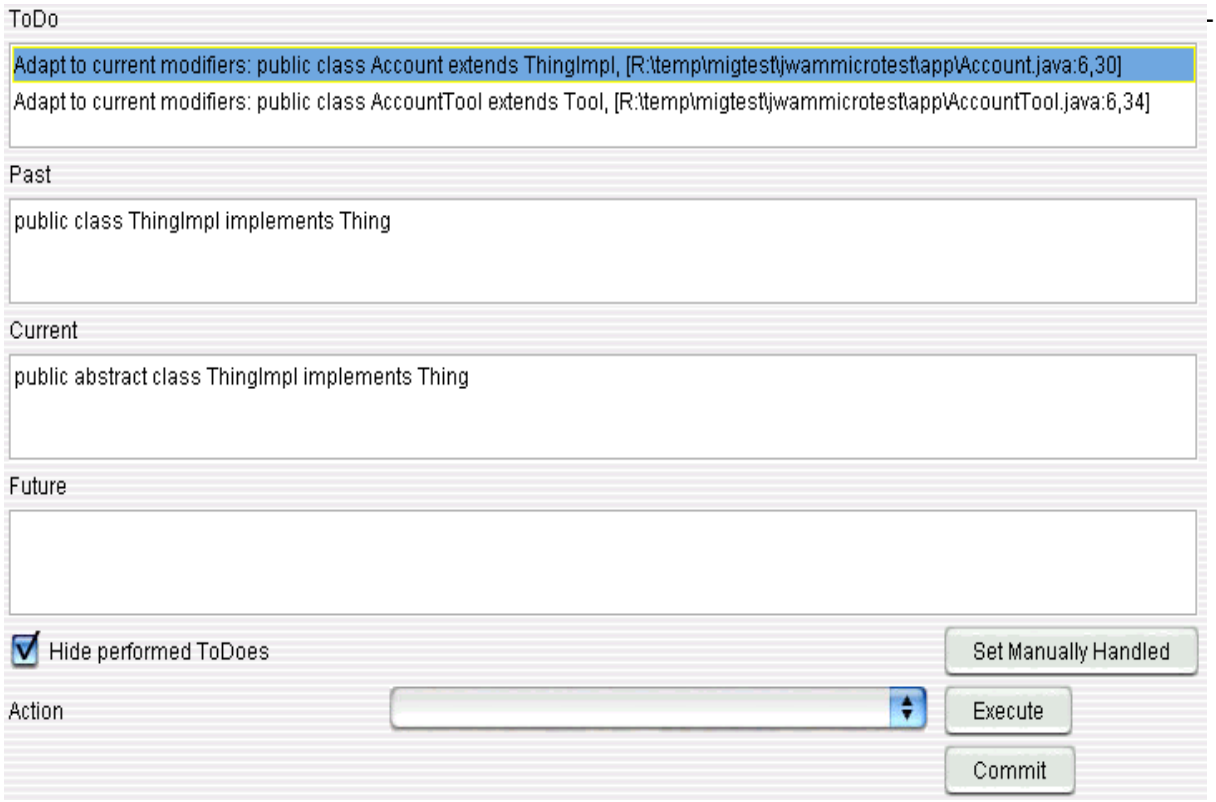


Abbildung A-11: Migratorwerkzeug: Modifier ändern

Das Werkzeug hat die Unterschiede zwischen den Modifiern der alten und neuen Version der Klasse `Thing` festgestellt und zwei Anwendungsklassen identifiziert, die über Vererbung mit der `Thing`-Klasse in Beziehung stehen. Das Werkzeug hat für diese beiden Klassen jeweils einen ToDo-Eintrag mit dem Hinweis auf die geänderten Modifier generiert. Änderungen an Modifiern gehören zur Klasse der potenziell-inkompatiblen Modifikationen und sind nicht generell automatisierbar.
In diesem Beispielsfall sind aber durch die Default-Implementationen der neuen Methoden im Interface `Thing` die notwendigen Anpassungen über die Schritte, die im Abschnitt A.6 beschrieben wurden, automatisiert durchführbar.
Die Abhängigkeiten dieser Modifikation und der Modifikation aus dem Abschnitt A.6 werden von der derzeitigen Werkzeugversion nicht automatisch erkannt.

Beteiligte ModificationAnalyzer: `ChangeClassModifierAnalyzer`
Erreichte Kompatibilität: automatisiert erledigt durch `ModificationCommands` des Abschnittes A.6