Diplomarbeit

UMLDiff

Erkennung und Analyse von Unterschieden in Klassendiagrammen und Sequenzdiagrammen

Martin Girschick

Fachgebiet Praktische Informatik
Professor W. Henhapl
Betreuer: Dipl.-Wirtsch.-Inform. Falk Fraikin

Technische Universität Darmstadt · Fachbereich Informatik · April 2002

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Diplomarbeit ohne Hilfe Dritter und nur mit den angegebenen Quellen und Hilfsmitteln angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Pfungstadt, April 2002 _____ *(Martin Girschick)*

Danksagung

Ich möchte mich an dieser Stelle bei meinen Eltern bedanken, die mir das Studium an der TU-Darmstadt ermöglicht haben. Vielen Dank auch an meine Kommilitonen, mit denen das Informatik-Studium in Darmstadt immer Spaß gemacht hat. Besonderer Dank geht an Philipp Matthias Hahn, David Schäfer und Christoph Müller, die mir viele Anregungen für diese Diplomarbeit gegeben haben.

Durch das Software-Engineering-Praktikum am Fachgebiet *Praktische Informatik* habe ich erstmals einen Einblick in das projektorientierte Entwickeln von Software erhalten. Dieses Praktikum kann ich jedem Informatiker empfehlen, da nur durch die richtige Projektarbeit ein Eindruck über die Schwierigkeiten gewonnen werden kann. Ich erinnere mich gerne an diese Zeit zurück, obwohl es sicherlich einige, stressige Monate waren. Danke an die Mitglieder des Teams *Mind and Magic*, an Professor Henhapl und seine Mitarbeiter.

Dank gilt auch meinen Freunden. Allen voran sicherlich Joyce Wittur, die am meisten unter dieser Diplomarbeit zu leiden hatte. Danke auch an Christian Henning für seine wertvollen Tipps.

Besonderer Dank gilt auch meinem Betreuer Falk Fraikin. Ihm war keine Frage zu stumpfsinnig und er hatte viele Anregungen, die Eingang in diese Arbeit gefunden haben.

Martin Girschick

Inhaltsverzeichnis

Zusammenfassung

In dieser Diplomarbeit werden Verfahren zur Analyse von UML-Klassendiagrammen und UML-Sequenzdiagrammen entwickelt. Ziel ist es, die Unterschiede zwischen verschiedenen Versionen dieser Diagramme zu erkennen und zu analysieren.

Sowohl in der Spezifikations- als auch in der Entwurfsphase und der eigentlichen Entwicklung werden Klassendiagramme eingesetzt und unterliegen dabei vielen Veränderungen. Im Rahmen dieser Arbeit wurde das Softwarepaket UMLDiff$_{cld}$ entwickelt, welches Unterschiede zwischen verschiedenen Versionen eines Klassendiagramms erkennt und mit einem eingefärbten Klassendiagramm visualisiert. Das dabei entwickelte Verfahren kann auch für andere Datenstrukturen angepaßt werden.

Sequenzdiagramme werden ebenfalls zur Spezifikation eingesetzt. An der Technischen Universität Darmstadt wurde ein Werkzeug entwickelt, welches Sequenzdiagramme dazu verwendet, um Tests zu spezifizieren. Das Sequenzdiagramm wird hierzu mit den Aufrufparametern und Rückgabewerten kombiniert. Damit ist es nun möglich, Testläufe durchzuführen. Während des Testlaufs wird aus den tatsächlich auftretenden Aufrufen und Parametern ein Sequenzdiagramm generiert. Durch Vergleich des spezifierenden mit dem generierten Diagramm kann nun der Testlauf untersucht werden. Hierzu wurde das Werkzeug UMLDiff$_{sqd}$ entwickelt. Es untersucht zwei Sequenzdiagramme und stellt die Unterschiede in Form eines eingefärbten Sequenzdiagramms dar. Dabei werden sowohl Unterschiede von Werten als auch strukturelle Unterschiede der Diagramme berücksichtigt.

Abstract

In this diploma thesis a method for the analysis of UML class diagrams and UML sequence diagrams is developed. The aim was to find and analyse differences between two versions of these diagrams.

During the specification and design of software products UML class diagrams are subject to frequent change. In this thesis a software package namend UMLDiff$_{cld}$ was designed which finds differences between two versions of a class diagram. These differences are visualized by means of colored class diagrams. The methods developed for this purpose can be adapted for the use with other data structures as well.

Sequence diagrams are used during specification, too. At the Technical University of Darmstadt a tool has been developed, which uses sequence diagrams to specify tests. The diagrams are combined with the values of the calls. With that it is possible to run automatic tests. During a test run a sequence diagram of the actual calls and values is recorded. By comparing the defining and the observed sequence diagram it is possible to check the test run for inconsistencies. This comparison is carried out by the tool UMLDiff$_{sqd}$. It compares the two diagrams and visualizes the differences using a colored sequence diagram. Both, differences of values and structural differences, are recognized.

Kapitel 1

Einführung

Die Unified Modelling Language (UML) hat sich in den letzten Jahren als standardisierte Modellierungssprache für Softwareprodukte durchgesetzt. Die Darstellung und Modellierung erfolgt mit verschiedenen Diagrammen die unterschiedliche Aspekte des Systems beleuchten. Diese sind beispielsweise die gewünschten Anwendungsfälle (sogenannte use cases), Klassendiagramme (class diagrams) oder Sequenzdiagramme (sequence diagrams). Letztere beschreiben die Kommunikation verschiedenener Klassen untereinander. Ein UML-Modell besteht aus mehreren dieser Diagramme, die in ihrer Gesamtheit, ein Softwareprodukt möglichst vollständig modellieren sollen.

Die Erstellung solcher Dokumente geschieht meist mit Softwareentwicklungswerkzeugen (engl. CASE-Tools). Bekannte Vertreter sind hier Rational Rose®[1] und Together ControlCenter™[2]. Diese Werkzeuge dienen meist nicht nur zur Erstellung der Diagramme, sondern helfen dem Softwareentwickler auch bei der Erstellung des Sourcecodes und der Dokumentation.

Viele CASE-Tools bieten inzwischen das sogenannte Roundtrip-Engineering. Darunter versteht man die Möglichkeit, sowohl mit Diagrammen als auch mit dem eigentlichen Sourcecode zu arbeiten. Änderungen in einem der beiden Teile werden automatisch im anderen Teil nachgeführt. Dabei ist die Generierung von Diagrammen aus bereits bestehendem Sourcecode ebenfalls möglich. Diese Werkzeuge entwickeln sich zunehmend zu Werkzeugen für den gesamten Softwareentwicklungszyklus. So gibt es Programme, die automatisch Tests durchführen können und aus Testläufen Sequenzdiagramme erzeugen.

Selten wird ein Softwareprodukt nur von einer Person entworfen und programmiert. Die Arbeit in Entwicklungsteams stellt somit den Regelfall dar. Softwareentwicklungswerkzeuge bieten auch hier einige Hilfswerkzeuge zur Teamarbeit an, die das gemeinsame Arbeiten an Dokumenten ermöglichen. Dazu gehört neben der Verwaltung von Revisionen und Varianten auch das Softwarekonfigurationsmanagement.

Eine Vielzahl der Softwareentwicklungsprozesse arbeitet iterativ. Diese Iterationen können dabei verschiedene Formen annehmen, auch eine Überlappung ist möglich. Es läßt sich jedoch allgemein feststellen, daß die ständige Überarbeitung und Korrektur der Entwurfsdokumente und Sourcecodes eine Vielzahl von verschiedenen Versionen produziert. Betrachtet man zusätzlich die Möglichkeit, aus vorhandenem Programmen Diagramme generieren zu können (durch Reverse-Engineering oder Testläufe), so nimmt die Anzahl der Diagramme weiter zu. Diese Diagramme

[1] `http://www.rational.com`
[2] `http://www.togethersoft.com`

beinhalten viele wertvolle Informationen über den Entwicklungsprozeß. Durch den Vergleich dieser Diagramme lassen sich verschiedene Informationen gewinnen. Folgende Szenarien sind denkbar:

- Klassendiagramme aus der Entwurfsphase werden verglichen, um den Fortschritt zu kontrollieren. Es läßt sich beobachten, wie neue Klassen hinzugefügt werden und diese mit Attributen und Operationen gefüllt werden. Durch Verwendung der von UML bereitgestellten, frei definierbaren Eigenschaftswerte (engl. properties) lassen sich beliebige Informationen zu diesen Klassen speichern, deren Veränderung ebenfalls überprüft werden kann. Beispielsweise kann in einer Klasse vermerkt werden, ob sie bereits vollständig implementiert wurde.

- Programmierer können überprüfen, ob sich in von ihnen verwendeten Programmteilen relevante Änderungen in einer neuen Version ergeben haben. Dies betrifft sowohl Änderungen an bereits vorhandenen Schnittstellen als auch das Hinzukommen neuer Funktionalität.

- Viele Softwareentwicklungswerkzeuge bieten die Möglichkeit, Klassenmodelle und Sourcecode ständig zu synchronisieren. Dies ist unter Umständen nicht wünschenswert, weil das Klassenmodell als feststehende Spezifikation nicht geändert werden sollte. Es besteht nun die Möglichkeit, automatisch generierte Klassendiagramme (die durch Reverse-Engineering aus dem Sourcecode entstanden sind) mit den Klassendiagrammen der Spezifikation zu vergleichen.

- Klassendiagramme lassen sich vor und nach einem Refactoring gegenüberstellen. Um einen Überblick über die erfolgten Veränderungen zu erhalten, bietet es sich an, alte und neue Version eines Programms anhand eines Klassendiagramms zu vergleichen. So erhält man schneller einen Überblick über die erfolgten Änderungen.

- Verschiedene Varianten eines Klassendiagramms, die unter Umständen von unterschiedlichen Entwicklern stammen, lassen sich vergleichen. Bei der Entwicklung für unterschiedliche Plattformen ist es oft notwendig, einige Komponenten für diese verschiedenen Systeme anzupassen. Durch der Analyse der Unterschiede zwischen diesen Implementierungen ist es möglich, gleiche Teile zu entdecken und entsprechend zusammenzufassen.

- Letztlich ist auch der Vergleich zwischen generierten Klassendiagrammen denkbar. Nicht immer existieren zu jedem verwendeten Paket Klassendiagramme. Diese können durch automatische Generierung erzeugt werden und ebenfalls untersucht werden.

- Änderungen an Sequenzdiagrammen können untersucht werden, um nötige Anpassungen des Sourcecodes zu finden. Bei Sequenzdiagrammen besteht bisher nur für einfache Fälle die Möglichkeit, eine Synchronisation zwischen Diagramm und Sourcecode herzustellen. Werden nun Änderungen an einem Sequenzdiagramm vorgenommen, müssen diese in bereits erfolgten Implementationen nachvollzogen werden. Durch eine graphische Darstellung der Differenzen kann dieser Vorgang beschleunigt werden.

- Sequenzdiagramme, die durch Testläufe generiert wurden, können mit spezifizierenden Sequenzdiagrammen verglichen werden. Die meisten Testverfahren basieren auf den Vergleich von Eingabe- und Ausgabewerten. Eine erweiterte Untersuchung, der auch die strukturellen Unterschiede erkennt, kann die Fehlersuche vereinfachen.

- Mehrere Testläufe können mit Hilfe der generierten Sequenzdiagramme verglichen werden. Hiermit läßt sich beispielsweise untersuchen, ob ein Refactoring Auswirkungen auf den Programmablauf hatte.

Ein kurzes Beispiel findet sich in Abbildung 1.1. Hier werden zwei Klassendiagramme gegenübergestellt. Folgende Unterschiede lassen sich hier erkennen.

- Es wurde eine Klasse `Discussion` mit dem Attribut `topic` und der Operation `setTopic` hinzugefügt. Sie erbt von der Klasse `Event`.
- Das Attribut `location` und die Operation `setLocation` wurden von der Klasse `Concert` in die Klasse `Event` verschoben.
- Das Attribut `reader` der Klasse `Reading` wurde umbenannt und der Typ in ein Array geändert.
- Die Operation `setReader` wurde entfernt, die Operationen `addReader` und `removeReader` hinzugefügt.

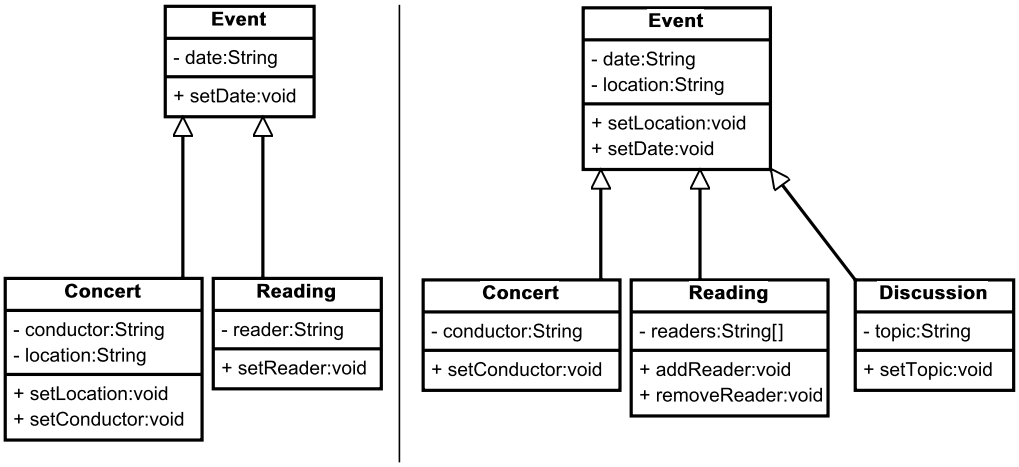


Abbildung 1.1: Ein Klassendiagramm in der Ausgangsform (links) und einer Variante (rechts)

Ziel dieser Arbeit ist es, diese Vergleiche zu automatisieren. Neben der Entwicklung eines geeigneten Algorithmuses werden folgende Ziele verfolgt:

- Analyse der Unterschiede und die Bestimmung ihrer Herkunft.
- Es soll ein geeignetes Format zur Repräsentation der Modelldaten gefunden werden.
- Die erkannten Differenzen sollen in einem Format beschrieben werden, welches die Transformation des einen Diagramms in das andere ermöglicht.
- Die Darstellung der Unterschiede soll durch abgewandelte UML-Diagramme und Reports im Textformat erfolgen.

- Die verwendeten Verfahren sollen auf ihre Anwendbarkeit für andere UML-Diagrammtypen untersucht werden.
- Prototypische Entwicklung eines Werkzeugs zur Differenzanalyse.

Kapitel 2

Status Quo

Dieses Kapitel soll einen Überblick über das Gebiet der Analyse und automatischen Generierung von UML-Modellen geben. Da UML primär in Verbindung mit objektorientierten Programmiersprachen verwendet wird, werden hier nur dafür relevante Ansätze vorgestellt. Hierbei werden sowohl bereits erhältliche Werkzeuge als auch Forschungsarbeiten berücksichtigt.

2.1 Softwareentwicklungswerkzeuge

Durch die Standardisierung der Unified Modelling Language (siehe [17]) wurde der Markt für Softwareentwicklungswerkzeuge neu belebt. Inzwischen existieren eine Vielzahl von Werkzeugen, die nahezu alle Anwendungsbereiche abdecken: Angefangen bei einfachen Werkzeugen zum Zeichnen von UML-Diagrammen über Integrierte Entwicklungsumgebungen mit automatischer Code-Generierung, Benutzungsoberflächenprototyping und Reverse-Engineering zu kompletten Paketen mit Testwerkzeugen, Teamunterstützung und Konfigurationsmanagement. Speziallösungen decken Bereiche wie Real-Time-Modelling und Prozeßssysteme ab.

Ein Großteil der Werkzeuge unterstützt das sogenannte Roundtrip-Engineering. Hierunter versteht man die Möglichkeit, sowohl an dem Objekt- und Klassendiagramm als auch am Sourcecode zu arbeiten. Änderungen in dem einen Teil werden transparent in den anderen übernommen. Fügt man beispielsweise im Sourcecode einer Klasse ein Attribut hinzu, so wird dieses auch im Klassendiagramm angezeigt. In diesem Zusammenhang ist auch das Reverse-Engineering sehr hilfreich, welches aus Sourcecode oder auch aus kompiliertem Code (beispielsweise Java-Klassen) Klassendiagramme erzeugen kann. Diese Klassendiagramme sind jedoch meist nur sehr einfach gehalten. Probleme gibt es beispielsweise in Java bei der Erkennung von Assoziationen, da Java nicht über das Template-Konzept von C++ verfügt. Um die Probleme bei dem Roundtrip-Engineering einzudämmen, setzen manche Programme spezielle Markierungen im Sourcecode. Diese dienen zum Wiederfinden von bestimmten Elementen und zur Speicherung zusätzlicher Informationen. Ein Nachteil des Roundtripping ist die ständige Synchronisation von Sourcecode und Modell, die unter Umständen gar nicht wünschenswert ist. Schließlich dient das Klassendiagramm als Spezifikation, die nicht leichtfertig geändert werden darf. Solche Änderungen müssen im Allgemeinen erst das übliche Change-Management durchlaufen.

Der Entwurf und die Entwicklung von Softwareprodukten wird heutzutage nicht mehr von einzel-

nen Personen durchgeführt, sondern Entwicklungsteams in verschiedenen Größen arbeiten gleichzeitig daran. Dies erfordert natürlich auch die Unterstützung durch das Modellierungswerkzeug. Das gleichzeitige Arbeiten an einem Dokument wird sehr selten praktiziert, da es zu viele Probleme aufwirft. Die meisten Programme verwenden eine zentrale Versionsbibiliothek, welches die Dokumente verwaltet. Durch die Versionierung der Dokumente wird gleichzeitig auch das Konfigurationsmanagement beziehungsweise die Revisions- und Variantenkontrolle ermöglicht.

Um verschiedene Versionen miteinander abzugleichen, verfolgen die Werkzeuge verschiedene Ansätze. Zumeist wird versucht, durch einen Diff/Merge-Prozess die Unterschiede zwischen den Versionen zu erkennen und zu vereinen. Werkzeuge, die anstatt des Vergleichs zweier Versionen den Änderungsprozeß selbst überwachen und die dabei ausgeführten Änderungen protokollieren, existieren nur im Forschungsstadium. Dies würde eine wesentlich genauere Analyse der verschiedenen Versionen ermöglichen. Die Auflösung von Konflikten, die üblicherweise bei dem Zusammenführen zweier Versionen auftreten, stellen das Hauptproblem in der Versionsverwaltung dar. Das automatische Auflösen dieser Konflikte ist nur selten möglich. Durch Kenntnis der Semantik einer Programmiersprache und deren Konstrukte lassen sich hier wesentlich bessere Verfahren entwickeln.

Die aktuellen Versionen vieler Entwicklungswerkzeuge unterstützen inzwischen auch Refactoring. Allerdings ist diese Unterstützung noch sehr einfach gehalten und beschränkt sich oft auf einfache Rename-Operationen. Refactoring-Browser, die es beispielsweise für Smalltalk gibt, sind für andere Programmiersprachen noch nicht verfügbar oder befinden sich noch im Beta-Stadium.

Es gibt sicherlich nicht *das* Werkzeug zur Modellierung von Software, jedes Werkzeug hat seine Vor- und Nachteile. Im folgenden werden einige ausgewählte Produkte vorgestellt[1].

ObjectiF®[2] von Microtool arbeitet mit einem zentralen Repository, welches die Synchronisation der Dokumente auf XML-Basis durchführt. Dies stellt eine Erweiterung gegenüber dem normalen, meist textbasierten Merge/Diff-Prozess dar, da hier mit einer feineren Granularität gearbeitet werden kann.

Rational *Rose®* ermöglicht es, die Änderungen an dem Modell und dem Sourcecode manuell abzugleichen. Hierzu werden im Sourcecode spezielle Tags eingefügt, was in manchen Situationen zu Problemen führt. Eine spezielle Realtime-Version bietet eine erweiterte Unterstützung für State-Diagramme und deren Simulation. Etwas ärgerlich ist die Tatsache, daß Rose nur ein Undo-Schritt erlaubt, hier sind andere Werkzeuge wesentlich komfortabler.

ArgoUML[3] ist eines der wenigen frei verfügbaren und damit kostenlosen UML-Werkzeuge, welches dennoch eine Vielzahl von Features bietet. Besonders hervorzuheben ist der sogenannte „Cognitive Support". Dieser hilft bei vielen Dingen, die nicht direkt mit der Modellierung zu tun haben, aber dennoch benötigt werden. Dazu zählen To-Do-Listen, Designkritiken, alternative Entwurfsrepräsentation und andere Dinge.

Together ist – wie viele andere UML-Werkzeuge auch – in Java geschrieben und somit auf mehreren Plattformen verfügbar. Dies führt allerdings zu dem Nachteil, daß die geringere Ausführungsgeschwindigkeit nicht immer ein flüssiges Arbeiten ermöglicht. Das simultane Roundtripping sorgt für die sofortige Synchronisation bei Änderungen an Modell oder Sourcecode. Dies geschieht auch ohne die Verwendung von speziellen Marken im Sourcecode. Durch eine offene

[1] Eine recht umfangreiche Übersicht findet sich unter `http://www.jeckle.de/umltools.htm`.
[2] `http://www.microtool.de/objectif/en/`
[3] `http://argouml.tigris.org/`

Architektur ist es sehr einfach, eigene Erweiterungen für Together zu entwickeln.

2.2 Testverfahren und Testwerkzeuge

Neben dem eigentlichen Entwurf der Software macht auch das Testen des Produkts einen nicht unerheblichen Teil des Softwareentwicklungszykluses aus. Softwareentwicklungsprozesse wie beispielsweise Extreme Programming (XP)[3] fordern das regelmäßige, automatisierte Testen nach jeder Änderung des Programms. Gerade bei diesem fortwährenden Ändern werden eine Unmenge von Versionen eines Programms generiert. Die Unterschiede zwischen diesen Versionen sind meist nur minimal, doch in ihrer Summe bergen sie viele interessante Informationen über den Entwicklungsprozess eines Softwareprodukts. Doch nicht nur der Sourcecode und damit verbunden das Klassendiagramm ändern sich, auch Abläufe in Sequenzdiagrammen können sich ändern und müssen angepasst werden.

Für Java existiert ein sehr nützliches Werkzeug - JUnit[4]. Die jeweiligen Tests müssen hier jedoch noch selbst programmiert werden. Sinnvoller wäre es, wenn bereits in der Spezifikations- und Entwurfsphase geeignete Tests erstellt werden würden. In XP ist dies Voraussetzung, hier wird nicht mit dem Programmieren begonnen, bevor nicht die dazugehörigen Tests entwickelt wurden.

In [18] wird ein Verfahren vorgestellt, welches automatisch aus UML-Statechart-Diagrammen Testfälle generiert. Dieser Ansatz ist jedoch nur für Programme geeignet, die eine automatenähnliche Struktur aufweisen.

Auch *Sequenzdiagramme* bieten sich zur Spezifikation von Tests an. Das an der TU-Darmstadt entwickelte System SeDiTeC[5] verfolgt diesen Ansatz. Hier lassen sich Programmteile durch automatisch generierte Stubs ersetzen, die ihr Verhalten aus spezifizierenden Sequenzdiagrammen beziehen. Bei Testläufen können nun die zu testenden Komponenten von den Stubs mit Testdaten versorgt werden, dabei lassen sich Rückgabewerte und Aufrufparameter überprüfen.

Die *Testdaten* müssen in diesem Fall noch selbst generiert werden, jedoch gibt es auch hier Werkzeuge, die diese automatisch generieren. Stellvertretend sei hier jtest[6] erwähnt.

Bei *Systemtests* muss im allgemeinen die Bedienoberfläche mit einbezogen werden, hier kommen dann sogenannte *Capture and Replay* Werkzeuge in Frage, die Benutzereingaben simulieren. WinRunner®[7] von Mercury Interactive ist ein solches Werkzeug.

Oft werden auch *Abdeckungstests* durchgeführt, die als implementationsbasierte Tests zu verstehen sind. Hier wird bei Testläufen untersucht, welche Codeteile bereits ausgeführt wurden. Dabei werden verschiedene Ziele betrachtet. Übliche sind hier die Pfadabdeckung, Methodenabdeckung oder Klassenabdeckung. Als Beispiel sei hier JProbe™ Coverage genannt[8].

Nicht unerwähnt sollten die *statischen Analyseverfahren* bleiben, wie beispielsweise *Metriken*, die eine qualitative Aussage über den Sourcecode treffen. Sie sind zwar nicht direkt den Testverfahren zuzuordnen, werden jedoch auch zur Qualitätskontrolle herangezogen. Weitere Betrachtungen finden sich in 2.4.

[4] `http://www.junit.org`
[5] `http://www.pi.informatik.tu-darmstadt.de/seditec/`
[6] `http://www.parasoft.com/products/jtest/index.htm`
[7] `http://www-heva.mercuryinteractive.com/products/winrunner/`
[8] `http://www.sitraka.com/software/jprobe/jprobecoverage.html`

Nicht näher angesprochen wurden Aspekte wie *Performanzanalyse*. Hier werden Untersuchungen über den Speicherverbrauch oder die Rechenzeit bzw. allgemein die Nutzung von Ressourcen gemacht.

Auf einer komplett anderen Ebene befinden sich *Testverfahren für die Analysephase*. Wie allgemein bekannt ist, sind die Kosten zur Beseitigung eines Fehlers um so höher, je später der Fehler erkannt wird. Somit ist es natürlich sinnvoll, bereits in einer möglichst frühen Phase der Softwareentwicklung mit dem Testen zu beginnen.

Durch die verstärkte Verwendung von Modellierungssprachen sind verifizierende Testverfahren wieder mehr ins Forschungsinteresse gerückt. Beispielsweise wird in [1] ein Verfahren vorgestellt, welches Klassendiagramme in eine formale Spezifikation überführt, die von einem Larch-Prover[12] auf Inkonsistenzen überprüft werden kann. Ein sehr ähnlicher Ansatz wird in [7] verfolgt.

Da die Informationen aus der Spezifikations- und Entwurfsphase sehr umfangreich sein können, sind eine Unmenge von Testverfahren denkbar. Auf der einen Seite sind hier die bereits erwähnten formalen Überprüfungen denkbar, andererseits aber auch informelle, die sich in Fragestellungen manifestieren, wie vollständig eine Spezifikation ist und ob alle Anwendungsfälle abgedeckt wurden. Hier finden sich auch viele Lösungsansätze in den verschiedenen Softwareentwicklungsprozesse, wie beispielsweise dem Rational Unified Process (RUP)[9].

Einen anderen Ansatz verfolgen visuelle Programmierwerkzeuge, die aus der Spezifikation ein lauffähiges Programm generieren. Als Beispiel sei hier Fujaba [23] genannt.

2.3 Analyse von UML-Diagrammen

Durch die Verwendung von UML bieten die meisten Entwicklungswerkzeuge bereits die Möglichkeit, einfache Überprüfungen des Modells vorzunehmen. Jedoch lassen sich aus den Modelldaten noch wesentlich mehr Informationen beziehen. In [22] wird die Information aus Klassendiagrammen mit Sequenzdiagrammen verglichen, um hier Konflikte aufzuzeigen. In [21] wird die Information aus einem UML-Diagramm verwendet, um andere UML-Diagrammtypen zu konstruieren.

UML-Klassendiagramme können mitunter sehr umfangreich werden. Unter Berücksichtigung der Modellinformationen ließen sich beispielsweise nicht relevante Teile dynamisch ein- und ausblenden. Diese Idee wird in [19] näher beschrieben.

2.4 Analyse von Sourcecode

Neben den Modelldaten läßt sich natürlich auch der Quellcode selbst verschiedenen Untersuchungen unterziehen. Softwaremetriken geben eine Aussage über die Qualität des Sourcecodes. Sehr einfache Metriken, wie Anzahl der Codezeilen und Verhältnis zwischen Kommentaren und Code sind auch bei nicht objektorientierter Software denkbar, doch gerade die Objektorientierung erlaubt es, viele Eigenschaften eines Programms zu ermitteln und daraus Schlüsse zu ziehen. Hierzu kann beispielsweise die Vererbungshierarchie betrachtet werden. Die Anzahl der Klassen oder Anzahl der Konstruktoren in diesen Klassen sind andere bekannte Metriken. Viele Test-

[9] `http://www.rational.com/products/rup/index.jsp`

werkzeuge und CASE-Tools unterstützen die Berechnung von Metriken. Stellvertretend sei hier Cantata++®[10] von IPL und Together[11] genannt.

Als weiteres Gebiet der Sourcecodeanalyse kann die Verifizierbarkeit angesehen werden, die heutzutage vor allem wegen den damit verbundenen immensen Kosten zusehends ans Bedeutung verliert. Die bereits erwähnte Codeüberdeckungsanalyse kann ebenfalls als Codeanalyse angesehen werden.

Die Generierung von Modelldaten aus bereits vorhandenem Sourcecode ist bereits in vielen Entwicklungswerkzeugen integriert. Eine automatische Erkennung von Entwurfsmustern in C++ wird in [2] näher beschrieben.

Für Java existiert bereits ein Werkzeug[12], welches verschiedene Versionen eines Java-Pakets analysiert und die Unterschiede durch modifizerte Javadoc-Dokumente darstellt.

2.5 Versionskontrolle

Die bekannten Versionskontrollsysteme wie RCS, CVS und SCCS bilden noch immer die Basis der meisten Versionsverwaltungen, jedoch sind die dahinterliegenden Algorithmen und Ideen nicht mehr zeitgemäß. Gerade bei dem Konfigurationsmanagement wird eine flexiblere Architektur benötigt. Ein Ansatz ist hier, Software nach ihren Features zu versionieren [26]. Hier werden die einzelnen Versionen mit Attributen belegt, diese lassen sich über Regeln kombinieren und somit die gewünschte Gesamtversion erzeugen. Dieses Konzept läßt sich ähnlich auch auf Programmiersprachenebene verfolgen. Dieses „Aspekt Orientierte Programmieren" wird in [16] vorgestellt.

Der Großteil der Systeme arbeitet versionsorientiert. Hier werden die Versionen eines Programms als Graph betrachtet. Abhängigkeiten, welche Version des einen Teils mit welcher Version eines anderen zusammenarbeitet, lassen sich hier nur schwer definieren. Demgegenüberstehend sind die änderungsorientierten Verfahren, die mehrere Versionen in einer Datei halten und durch bedingte Kompilierung die benötigte Version erstellen. Diese Ansätze und deren Vertreter werden in [8] gegenübergestellt.

Einige wenige Arbeiten beschäftigten sich mit der Versionskontrolle über strukturierte Daten. Hier wäre besonders interessant, wenn nicht nur die syntaktischen, sondern auch die semantischen Unterschiede zwischen Versionen erkannt werden würden. Für eine einfache Sprache wurde das in [14] durchgeführt. Dieser Ansatz arbeitet mit einem Partitionsprinzip, welcher semantisch gleiche Fragmente in Äquivalenzklassen verwaltet. Dieser Ansatz bezieht jedoch keine Objektorientierung ein. Ein Vergleich der Syntaxbäume zweier Programme wird in [24] durchgeführt.

Die Versionsverwaltung von XML-Dokumenten wird in 4.3 näher betrachtet.

[10] `http://www.iplbath.com/tools`
[11] `http://www.togethersoft.com`
[12] `http://sourceforge.net/projects/javadiff/`

Kapitel 3

Differenzanalyse von Klassendiagrammen

Klassendiagramme stellen in den meisten Softwareentwicklungsprozessen ein zentrales Element zur Softwaremodellierung dar. Sie gehören zu den Diagrammtypen, die am meisten im Laufe des Entwicklungszykluses bearbeitet werden müssen. Diese Änderungen sind nicht willkürlicher Natur, sondern verfolgen gewisse Ziele. Deshalb wird im folgenden die Erkennung und Analyse dieser Veränderungen in UML-Modelldaten näher untersucht.

3.1 Der Evolutionsprozess von Klassendiagrammen

Typischerweise steht am Anfang eines Softwareentwicklungsprozesses die Anforderungsanalyse. Ist diese abgeschlossen, beginnt unter anderem der Entwurf einer Klassenhierarchie.

- In der *Aufbauphase* werden Schnittstellen hinzugefügt und eine Aufteilung in Pakete getroffen. Diese Aufteilung und auch die Schnittstellen sind am Anfang sehr labil und ändern sich oft.
- Nachdem die Schnittstellenspezifikation in erster Näherung abgeschlossen erscheint, werden die Klassen mit Leben gefüllt. In dieser *Implementationsphase* sollten die Schnittstellen stabil bleiben.
- Es schließt sich die *Testphase* an. Kleinere Korrekturen werden noch ausgeführt.

Diese drei Phasen können sich mehrfach wiederholen und auch ineinander verzahnt ablaufen. Somit werden im allgemeinen mehrere Iteration benötigt, bis die Spezifikation vollständig erfüllt ist.

3.2 Elemente in Klassendiagrammen

In einem Klassendiagramm werden zusammengehörige Klassen zu Paketen zusammengefasst. Diese Klassen drücken ihre Eigenschaften durch Attribute und Operationen aus. Hinzu kommt

die Vererbungshierarchie der Klassen untereinander, welche durch Generalisierungen dargestellt werden. Als weiteres nützliches Konstrukt erweisen sich Assoziationen, die Abhängigkeiten von Klassen darstellen. Nähere Informationen hierzu finden sich in [17]. Diese Modellierungselemente sind mit Eigenschaften belegt, die beispielsweise ihre Sichtbarkeit gegenüber anderen Elementen beschreiben oder auch ihren Typ definieren. Eine Übersicht findet sich in Tabelle 3.1.

Zur Tabelle 3.1 sind einige Anmerkungen notwendig. Der *Stereotyp* ist ein UML-spezifisches Zusatzmerkmal von vielen Modellierungselementen. Er beschreibt die Funktion des Elements und wird im allgemeinen aus einem festen Wortschatz gewählt. Ein Beispiel für einen Stereotyp einer Assoziation wäre `implicit`. Dieser gibt an, daß die Assoziation nur konzeptioneller Natur ist und nicht wirklich implementiert ist. Die *Sichtbarkeit* definiert die Zugriffsmöglichkeiten auf ein Element. Hier sind folgende Werte üblich: `public`, `private`, `protected`, `package`. Der *final*-Modifikator verhindert eine weitere Änderung des Elements (Vererbung, Änderung von Werten etc.). Der *Typ* bezeichnet den Datentyp. Hier sind sowohl Basistypen (`char`, `int` etc.) als auch komplexe Typen, die durch andere Klassen repräsentiert werden, erlaubt. Klassen können in drei Formen verwendet werden: `Class`, `Interface`, `DataType`. Assoziation werden in drei Gruppen `Association`, `Aggregation` und `Composition` unterteilt.

Diese Elemente können nicht frei in Diagrammen auftauchen, sondern sie sind an andere Elemente gebunden. Diese Abhängigkeit wird in Tabelle 3.2 dargestellt. Sie ist als eine „enthalten in”-Relation zu verstehen. Pakete enthalten Klassen, Klassen enthalten Attribute und Operationen. Assoziationen können sowohl in Attributen als auch in Klassen enthalten sein. Dies Bedarf einer näheren Erklärung: Eine Assoziation wird oft durch ein Attribut repräsentiert. Dieses speichert die Referenz auf das assoziierte Element, somit ist die Assoziation Teil dieses Attributs. Assoziationen können jedoch auch andere Zusammenhänge darstellen. Da sie in den meisten fällen gerichtet sind, kann eine eindeutige Quellklasse bestimmt werden, die diese Assoziation enthält. Generalisierungen gehen ebenfalls von einer Klasse aus, was auch hier die „enthalten in”-Relation rechtfertigt.

Im folgenden werden Pakete, Klassen und Attribute als *Containerelemente* (engl. containers) bezeichnet. Die darin enthaltenen Elemente heißen *Teilelemente* (engl. parts).

Element	Name	Stereotyp	Sichtbarkeit	final	static	abstract	Typ	weitere
package	+	+	–	–	–	–	–	
class	+	+	–	+	–	+	–	extends/implements
attribute	+	+	+	+	+	–	+	multiplicity
operation	+	+	+	–	+	+	+	
parameter	+	–	–	–	–	–	+	
generalization	+	–	–	–	–	–	–	supplier
association	+	+	–	–	–	–	–	client, client multiplicity, supplier, supplier multiplicity, directed

Tabelle 3.1: Elemente in UML-Klassendiagrammen und ihre Eigenschaften

class...	attribute...	operation...	parameter...	generalization...	association...	...gebunden an...
∞	–	–	–	–	–	...package
–	∞	∞	–	2	2	...class
–	–	–	–	–	1 (+ 1 class)	...attribute
–	–	–	∞	–	1 (+ 1 class)	...operation

Tabelle 3.2: Abhängigkeiten zwischen UML-Diagrammelementen

Aus den in der Tabelle gegebenen Abhängigkeiten lässt sich ein gerichteter Graph entwickeln. Dieser ist in Abbildung 3.1 dargestellt. Nun stellt sich die Frage, wie sich Veränderungen an einem solchen Klassendiagramm in dieser Darstellung manifestieren.

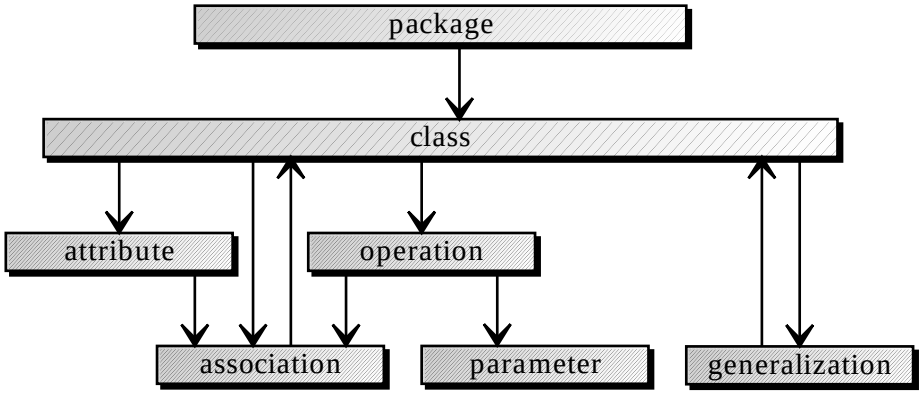


Abbildung 3.1: Die Abhängigkeiten der UML-Elemente als gerichteter Graph

Man erkennt, daß bei der Arbeit mit Klassendiagrammen neben den offensichtlichen Operationen Einfügen und Löschen auch noch das Ändern eines Elements möglich ist. Dies schlägt sich in der Änderung der Eigenschaften eines Elements nieder. Was noch viel interessanter erscheint, sind die Verschiebungen von Elementen, die in einem solchen Graphen auftreten können. Das Verschieben von Elementen ist gerade bei dem Refactoring von Software eine oft wiederkehrende Tätigkeit. Würde man diese Verschiebungen nicht erkennen können, würde man stattdessen das Löschen eines Objekts und das Einfügen eines anderen Objekts an einer anderen Stelle bemerken. Dadurch geht ein großer Teil der Semantik der Änderung verloren.

Verschiebungen sind nur innerhalb der selben Art von Containerlementen möglich. D.h. Attribute können nur von einer Klasse zu einer anderen verschoben werden.

Änderungen eines Elements können beispielsweise durch Umbenennen des Elements oder durch Verändern seiner Sichtbarkeit erreicht werden.

3.3 Ursachenanalyse der Änderungen

Die Motivation hinter diesen Änderungen kann viele Ausprägungen haben. Vor allem in der Entwurfsphase, wenn das Klassendiagramm erstellt wird, sind viele Veränderungen erkennbar. In der eigentlichen Implementationsphase können Änderungen an Schnittstellen durch externe Systeme hervorgerufen werden oder auf Grund falscher Designentscheidungen nötig werden. Auch das Hinzufügen von Funktionalität schlägt sich in geänderten UML-Modelldaten nieder.

Änderungen an den UML-Modelldaten sollten nach Möglichkeit geregelt ablaufen. Durch das sogenannte Refactoring kann dies gewährleistet werden. Hier werden genau festgelegte Regeln bei der Änderung der Klassenstruktur und des Sourcecodes befolgt, so daß eine semantische Äquivalenz beibehalten wird. Diese Refactoring-Schritte sind genau benannt und können zu großen Teilen durch Differenzanalyse erkannt und dokumentiert werden.

Betrachtet man neben den Modelldaten noch andere Parameter eines Softwareprodukts, so lassen sich eine Unmenge von Analysen durchführen. In Kapitel 7 werden einige dieser Möglichkeiten

vorgestellt. Stellvertretend sei hier die Bildung der Differenz von Softwaremetriken genannt. Dies wäre nicht nur im produktiven Umfeld von Interesse, sondern könnte auch Grundlage für Forschungen darstellen. So könnte man untersuchen, ob Refactoring Softwaremetriken beeinflusst.

3.4 Erkennung der Differenzen

Es sind prinzipiell zwei Ansätze denkbar, um die Unterschiede zweier UML-Klassendiagramme zu ermitteln. Diese werden im folgenden vorgestellt und ihre Vor- und Nachteile analysiert.

Mit einer entsprechende Unterstützung des Softwareentwicklungswerkzeugs könnte man die Änderungsoperationen direkt protokollieren. Dies hätte zwar den Vorteil, daß der gesamte Ablauf analysiert werden kann, führt jedoch im allgemeinen zu einer Unmenge von Daten, die teilweise Redundanzen aufweisen. Dies können beispielsweise dadurch entstehen, daß man neue Elemente hinzufügt, diese jedoch vor der nächsten Version wieder entfernt. Durch einen weiteren Analyseschritt ließen sich diese Änderungsoperationen wieder entfernen. Ein weiterer Nachteil ist die eingeschränkte Anwendung. Mit diesem Verfahren können nur ein Dokument und seine Entwicklungshistorie analysiert werden. Eine Analyse zweier paralleler Entwicklungsstränge müsste mit dem letzten gemeinsamen Entwicklungsstand durchgeführt werden. Auch der Vergleich mit automatisch generierten UML-Klassendiagrammen[1] ist nicht möglich.

Der andere Ansatz folgt der Vorgehensweise von Versionsverwaltungsprogrammen. Der Vergleich wird hier über zwei „Schnappschüsse" durchgeführt. Diese stammen aus einer Versionsbibiliothek oder aus einem Reverse-Engineering-Prozess. Durch ein Analyseverfahren werden die Unterschiede extrahiert und können daraufhin weiter analysiert oder visualisiert werden. Ein solches Analysewerkzeug wird im folgenden Kapitel entwickelt.

Um eine Unabhängigkeit von dem gewählten Analyseverfahren zu erreichen, erscheint es sinnvoll, ein für beide Ansätze geeignetes Format zur Beschreibung der Änderungen zu verwenden. Die Verwendung von Transformationsoperationen erfüllt diese Eigenschaft. Diese überführen das Klassendiagramm sukzessive in die daraus entstandene Variante.

Die Komplexität dieser Transformationsoperationen wird durch die Qualität des Erkennungsalgorithmuses bestimmt. Die Erkennung der bereits erwähnten Operationen zum *Verschieben* von Modellierungselementen gestaltet sich als sehr schwierig. Einfacher hingegen sind Operationen wie *hinzufügen*, *löschen* und *modifizieren*.

Eine erste Nomenklatur für Operationen und die dadurch betroffenen Elementtypen wird in Tabelle 3.3 vorgestellt.

Es existieren mehrere Möglichkeiten, einen Erkennungsalgorithmus für Differenzen in Klassendiagrammen zu implementieren. Im folgenden Kapitel werden mehrere Verfahren untersucht und schließlich ein eigenes Verfahren vorgestellt.

[1] entstanden aus Reverse-Engineering bereits vorhanden Sourcecodes

Transformationsoperation	Anwendbarkeit	Bemerkung
add	alle	nur einzelne Elemente
delete	alle	entfernt Element inkl. Inhalt
rename	alle außer generalization	verändert `name`-XML-Attribut
change visibility	class, attribute, operation	verändert `visibility`-XML-Attr.
move	class, attribute, operation	bewegt Element in anderen Container
clone	class, operation	z.B. Erzeugung Klasse aus Interface
change type	alle außer generalization und package	siehe 3.1
modify	alle außer package	`abstract, multiplicity` etc.
add/delete stereotype	alle außer generalization und package	`stereotype`-XML-Attribute
add/del./mod. property	alle	frei definierbare Eigenschaftswerte

Tabelle 3.3: Operationen zur Veränderung von Klassendiagrammen

3.5 Darstellung der Ergebnisse

Neben einer einfachen Ausgabe der Transformationsoperationen als Tabelle sind noch andere Visualisierungen denkbar. Durch farbliche Markierungen ließen sich die hinzugefügten, gelöschten und geänderten Objekte in dem Klassendiagramm darstellen. Verschobene Elemente können zusätzlich mit einer Linie verbunden werden.

Die Platzierung der Klassen innerhalb des Klassendiagramms stellt ein Problem dar. Sofern nicht bereits in den Modelldaten diese Information enthalten ist, muss diese Platzierung automatisch geschehen. In [9] wurde ein System entwickelt, was diese automatische Platzierung leistet. Hierbei werden auch Assoziationen und Generalisierungen berücksichtigt.

Zusätzlich zu statischen Diagrammen ist die Animation der Diagramme möglich. Dies würde auch das Beobachten mehrerer Versionen oder einer Versionshistorie erlauben. So ließe sich die Entstehung einer Software im Zeitraffer anhand eines UML-Diagramms beobachten.

Weitere Verwendungsmöglichkeiten werden in Kapitel 7 vorgestellt.

Kapitel 4

Prototyp zur Differenzanalyse von Klassendiagrammen

Die im vorhergehenden Kapitel gewonnenen Erkenntnisse sollen nun in ein Werkzeug umgesetzt werden. Dieses Werkzeug wird im folgenden mit UMLDiff$_{cld}$ bezeichnet. Der Index *cld* steht für *classdiagram*.

Zuerst wird die Frage nach einem geeigneten Eingabeformat näher beleuchtet. Im folgenden Abschnitt wird dann ein Verfahren zur Erkennung der Unterschiede zweier Klassendiagramme entwickelt. Das Ergebnis dieser Analyse ist eine Transformationsbeschreibung, die aus dem einen Klassendiagramm dessen Variante rekonstruiert. Schließlich wird noch eine Möglichkeit der Visualisierung dieser Informationen anhand von farbigen Klassendiagrammen beschrieben.

Sowohl für UML-Klassendiagramme als auch für XML-Dateien existieren die Bezeichnungen *Attribut* und *Element*. Um Verwechslungen zu vermeiden, wird im folgenden immer von XML-Elementen und XML-Attributen gesprochen.

4.1 Datenformat für UML-Modelldaten

Der Austausch von UML-Modelldaten gestaltet sich etwas schwierig, da die derzeit verfügbaren Austauschformate einige Nachteile aufweisen. Im folgenden werden diese Formate kurz vorgestellt und ihre Vor- und Nachteile beleuchtet.

Rational MDL

Rational Rose war eines der ersten Softwareentwicklungswerkzeuge, das UML als Modellierungssprache verwendete. Dies erklärt die weite Verbreitung und Unterstützung des Rose-Formats. Viele andere Werkzeuge unterstützen den Import oder Export in diesem MDL-Format. Rational selbst stellt keine Informationen über dieses Format zur Verfügung, deshalb müßte man die Datenstruktur durch Reverse-Engineering entschlüsseln. Da dieses weitverbreitete Format demnächst von dem neuen Standard XMI abgelöst wird, erscheint dieser Aufwand nicht sinnvoll.

UMLtext

Im Rahmen eines Praktikums entstand am Northeastern University's College of Computer Science das Format UMLtext. Hier wurde eine Präprozessor verwendet, der die UML-Diagramminformation vom Java-Sourcecode trennt. Gearbeitet wird an einer gemeinsamen Datei – ein Ansatz der aus dem Literate-Programming bekannt ist.

UMLscript

An der Universität Würzburg wurde das Format UMLscript[1] entwickelt. Dieses textbasierte Format wurde anhand einer attributierten Grammatik definiert. Durch automatische Generierung entstand ein Parser für Java. Eine Erweiterung des Formats und Anpassung an die hier benötigten Merkmale ist aufgrund der verwendeten Grammatik zu aufwendig.

```
DIAGRAM
   CLASS CallQueue
   ATTRIBUTES queue:List, source, capacity
END_DIAGRAM
```

Abbildung 4.1: Beispiel: UMLscript

UML eXchange Format (UXF)

Datenformate, die auf XML basieren, bieten wesentlich mehr Möglichkeiten zur Anpassung. Eine DTD[2] für UXF, die mehrere UML-Diagrammtypen beschreibt, wurde an der Keio Universität (Tokyo) entwickelt[3]. In Abbildung 4.2 ist eine einfache UXF-Datei dargestellt. Auf die Verwendung von XML-Attributen wurde verzichtet, was eine etwas umständlichere Darstellung zur Folge hat.

```
<?xml version="1.0"?>
<!DOCTYPE UXF SYSTEM "uml.dtd">
<UXF Version="2.0" xmlns:UXF="http://www.yy.cs.keio.ac.jp/~suzuki/project/uxf/">
  <ClassDiagram>
    <Name>Polygon</Name>
     <Class>
      <Name>Polygon</Name>
      <IsAbstract>true</IsAbstract>
      <Attribute>
        <Name>points</Name>
      </Attribute>
    </Class>
  </ClassDiagram>
</UXF>
```

Abbildung 4.2: Beispiel: UXF

[1] `http://www2.informatik.uni-wuerzburg.de/SugiBib/`
[2] Document Type Definition. Beschreibungssprache für XML- und SGML-Dokumente.
[3] `http://www.yy.cs.keio.ac.jp/~suzuki/project/uxf/`

XML Metadata Interchange (XMI)

Der Wunsch nach einem standardisierten Austauschformat führte zur Initiative mehrerer Forschungseinrichtungen und Firmen, aus der zuerst das Format MOF[4] zum Austausch von Metadaten entstand. Ziel der Entwicklung war ein Austauschformat für beliebige Datenstrukturen. Hier werden neben den Datenstrukturen auch deren Zugriffsmethoden und Verknüpfungen beschrieben. Eine mögliche Repräsentation dieses Metadatenformats ist XMI, welches diese Metadaten in einer XML-Struktur ablegt.

Auf der Basis von MOF wurde ein Metadatenformat für UML-Modelle entwickelt. Dieses Format eignet sich zwar zur Beschreibung aller möglichen Konstrukte innerhalb der Unified Modelling Language, es ist allerdings so komplex, daß eine Implementierung innerhalb dieser Arbeit den Rahmen sprengen würde. Ein weiteres Problem sind die Inkompatibilitäten zwischen den verschiedenen Implementationen von XMI.

UML-Xchange

Eine DTD für SGML[5] namens UML-Xchange[6] stammt von Normand Rivard. Sie wird allerdings nicht mehr weiter entwickelt und unterstützt nicht alle von UML 1.3 geforderten Konstrukte.

Javadox

Speziell auf den Sprachumfang von Java zugeschnitten ist das Java-Doclet Javadox[7]. Doclets sind Erweiterungen des Java-Dokumentationswerkzeugs Javadoc, welches normalerweise eine HTML-Dokumentation der Schnittstellen generiert. Javadox erweitert diese Generierung um einen XML-Export, der teilweise auch den Inhalt eines Klassendiagramms widerspiegelt.

Anuclad UML Diagramm (AUD)

Im Rahmen einer Studienarbeit[8] am Fachgebiet Graphisch Interaktive Systeme[9] der Technischen Universität Darmstadt wurde bereits ein XML-Schema[10] entworfen, welches zur Beschreibung von Klassendiagrammen dient. Hier wurde besonders viel Wert auf eine graphische Darstellungsmöglichkeit der Diagramme gelegt, wie durch das `geometry`-Element in Beispiel 4.3 deutlich wird.

Dieses Datenformat ist bereits recht gut zur Analyse geeignet. Die in 3.2 dargestellten Abhängigkeiten sind durch die Elementschachtelung bereits angedeutet. Im folgenden Abschnitt wird dieses Format etwas angepasst, um eine einfachere Differenzanalyse zu ermöglichen.

---

[4] Metadata Object Facility
[5] Standard Generalized Markup Language
[6] `http://www3.sympatico.ca/rivardn/uml/umlxchng.html`
[7] `http://www.componentregistry.com/jdox/javadox.jar`
[8] `http://www.girschick.net/martin/study/anuclad/`
[9] `http://www.gris.informatik.tu-darmstadt.de`

```
<?xml version="1.0" encoding="UTF-8"?>
<umldiagrams>
  <classdiagram>
    <package name="default" id="default">
      <classes>
        <class name="console" id="default.console" visibility="public">
          <geometry x="300" y="200" />
          <operations>
            <operation name="write" id="default.console_write_String_str_"
                       signature="(String str)" visibility="public" />
          </operations>
        </class>
      </classes>
    </package>
  </classdiagram>
</umldiagrams>
```

Abbildung 4.3: Beispiel: AUD

UML Diagram Data (UDD)

Auf der Basis von AUD wird nun das im folgenden verwendete Format UDD entwickelt. Das XML-Schema ist unter `http://www.girschick.net/martin/study/umldiff/` verfügbar.

Die Sammelelemente (`classes, operations, attributes`) wurden entfernt. Die Package-Struktur, die durch verschachtelte XML-Elemente nachgebildet wurde, ist durch eine einstufige Hierarchie abgelöst worden. Die Signatur von Operationen wurde durch das XML-Unterelement `parameter` konkretisiert.

Ein Diagramm zur Beschreibung des XML-Schemas findet sich auf Seite 25 in Abbildung 4.4. Als Basistyp für alle XML-Elemente wurde ein XML-Schema Complex Type namens `objectType` entworfen, der die XML-Attribute `name, id, stereotype, hidden?` und `url` zusammenfasst.

Ebenfalls zu dem Complex Type gehört das optionale XML-Element `property`. Dieses XML-Element kann für zusätzliche Eigenschaften des Modellierungselements verwendet werden. Beispielsweise wird hier die graphische Positionierung von Klassen vermerkt.

Des weiteren ist anzumerken, daß XML-Attribute, die mit einem Fragezeichen enden, vom Typ `boolean` sind.

Aus Tabelle 3.2 geht hervor, daß Assoziationen auch an Attribute gebunden sein können. Auf diese Darstellung wurde hier verzichtet, da sie die homogene Struktur des Dokuments zerstören würde. Durch das optionale XML-Attribut `client` kann jedoch auf das repräsentierende Attribut verwiesen werden.

In Abbildung 4.5 wird ein Ausschnitt einer UDD-Datei wiedergegeben, welches das Ausgangsdiagramm aus dem Beispiel in Kapitel 1 repräsentiert.

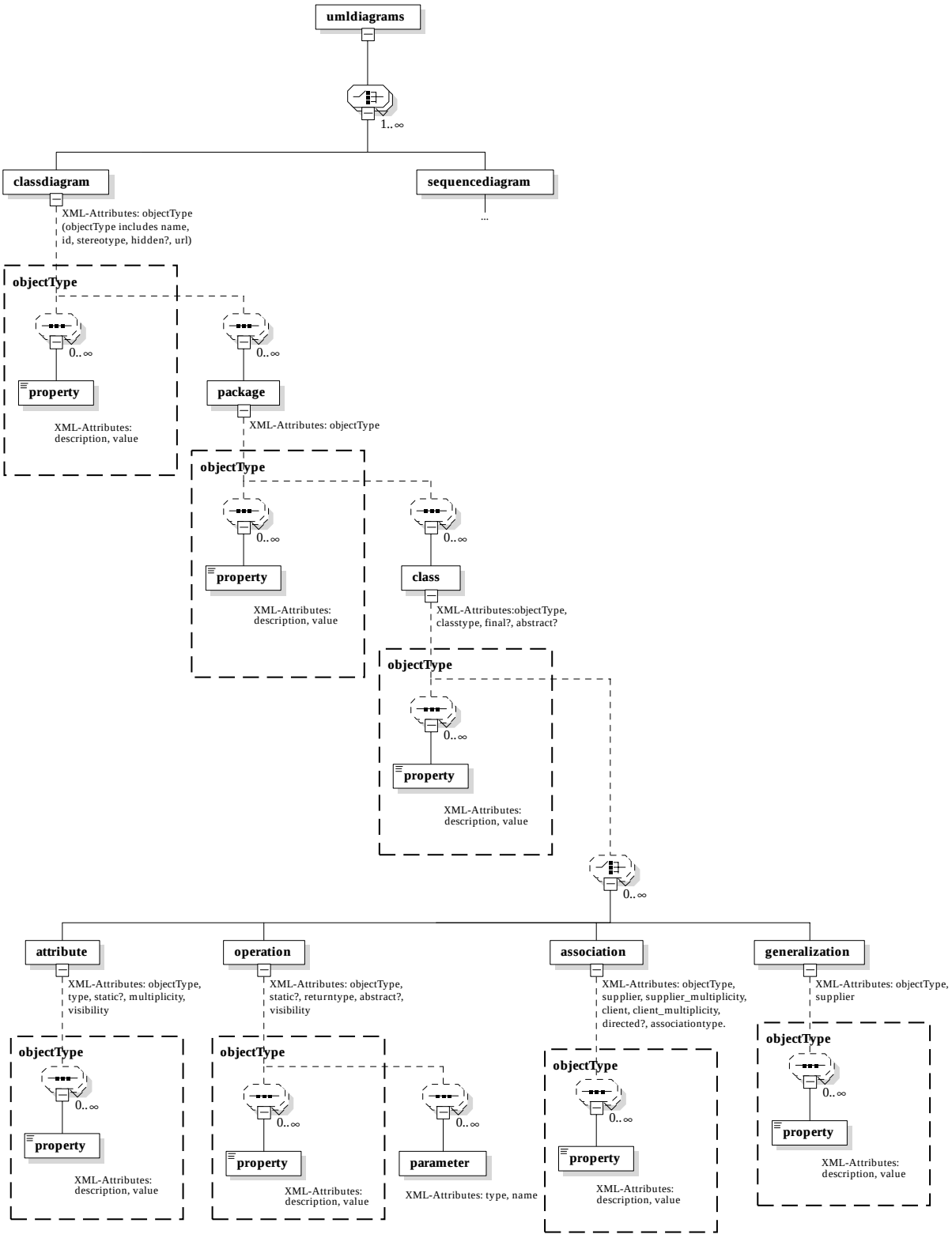


Abbildung 4.4: XML-Schema für UML-Klassendiagramme

```
<?xml version="1.0" encoding="UTF-8"?>
<umldiagrams>
 <classdiagram name="base">
  <package name="base">
   <class name="Event" classtype="Class">
    <operation name="setDate" returntype="void" visibility="public">
     <parameter name="date" type="String"/>
    </operation>
    <attribute name="date" type="String" visibility="private"/>
   </class>
   <class name="Concert" classtype="Class">
    <operation name="setLocation" visibility="public">
     <parameter name="location" type="String"/>
    </operation>
    <operation name="setConductor" visibility="public">
     <parameter name="conductor" type="String"/>
    </operation>
    <attribute name="conductor" type="String" visibility="private"/>
    <attribute name="location" type="String" visibility="private"/>
    <generalization supplier="base/Event"/>
   </class>
   <class name="Reading" classtype="Class">
    <operation name="setReader" returntype="void" visibility="public">
     <parameter name="reader" type="String"/>
    </operation>
    <attribute name="reader" type="String" visibility="private"/>
    <generalization supplier="base/Event"/>
   </class>
  </package>
 </classdiagram>
</umldiagrams>
```

Abbildung 4.5: Beispiel: UDD

4.2 Generierung der Eingabedaten

Um eine Differenzanalyse durchführen zu können, müssen nun UML-Modelldaten in das UDD-Format gebracht werden. Hier wurde ein Module für das CASE-Tool Together entwickelt, welches ein selektiertes Klassendiagramm in eine UDD-Datei exportiert. Together bietet sehr umfangreiche Möglichkeiten, auf die internen Datenstrukturen zurückzugreifen. Selbst der Sourcecode, der sich hinter den Modellelementen befinden kann, ist zugänglich und kann analysiert und modifiziert werden.

Eine weitere Möglichkeit zur Generierung einer UDD-Datei besteht in der Konvertierung der Ausgabe von Javadox. Wie bereits auf Seite 23 erwähnt ist Javadox ein Doclet für das Dokumentationsgenerierungswerkzeug `javadoc`. Es erzeugt eine XML-Datei, welche die Klassenstruktur und die darin enthaltenen Attribute und Operationen sehr umfangreich beschreibt. Leider ist hier kein Rückgriff auf den Sourcecode möglich, da `javadoc` hierzu noch keine Möglichkeit bietet.

4.3 Vergleichsalgorithmen für Dateien und Datenstrukturen

Sowohl der Vergleich zweier Dateien als auch der Vergleich von Datenstrukturen war bereits Forschungsgegenstand mehrerer Arbeiten. Da durch das in 4.1 beschriebene Datenformat die Klassendiagramme als XML-Datei vorliegen, bieten sich verschiedene Algorithmen zur Bestimmung der Unterschiede dieser zwei Dateien an. Im folgenden werden einige dieser Algorithmen auf ihre Anwendbarkeit zum Vergleich strukturierter Dokumente untersucht.

Die rein textbasierten Verfahren, wie man sie beispielsweise aus dem Unix-Tool `diff`[10] kennt, sind zur Analyse ungeeignet, da sie beispielsweise verschobene Passagen nicht wiederfinden können und keine Kenntnis darüber besitzen, welche Änderungen einer Zeile als signifikant einzustufen sind.

Da XML-Dokumente in einer Baumstruktur dargestellt werden können, sind auch Graphenalgorithmen in Betracht zu ziehen. Hier trifft man auf das *Tree-to-Tree-Problem* [20]. Dieses beschreibt das Problem, zwei Bäume auf ihre Unterschiede zu untersuchen. Als Knotenmarkierung wird ein beschränktes Alphabet verwendet. In [13] werden fünf solche Algorithmen vorgestellt und mit vorherigen Ansätzen verglichen. Durch die Einschränkung des beschränkten Alphabets sind diese Algorithmen leider nicht in dem hier benötigten Kontext verwendbar. Einerseits ist das verwendete „Alphabet" sehr umfangreich, was die Laufzeit der Algorithmen stark erhöht. Andererseits sollen auch Elemente gefunden werden, die nicht vollständig identisch sondern nur ähnlich sind.

Algorithmen, die sich mit der Versionierung und Differenzenerkennung in XML-Dokumenten beschäftigen, scheinen am besten geeignet. Auf den Webseiten von IBM/Alphaworks findet sich das Programm XMLTreeDiff[11]. Dieses überträgt die Funktionsweise von `diff` auf XML-Dateien und zeigt die hinzugefügten, entfernten und veränderten XML-Attribute und -Elemente farbig markiert an. Leider werden mit diesem Werkzeug Verschiebungen von Elementen nicht erkannt.

In [5] wird ein Verfahren vorgestellt, welches nicht mit den üblichen Unterschieden zwischen Dateien arbeitet. Hier werden alle Versionen in einer Datei verwaltet. Durch zusätzliche XML-

[10] `http://www.gnu.org/software/diffutils/diffutils.html`
[11] `http://www.alphaworks.ibm.com/tech/xmltreediff`

Attribute wird für jedes XML-Element ein „Lebenszyklus" durch sogenannte Zeitstempel definiert. Will man nun eine bestimmte Version extrahieren, überprüft man anhand der Zeitstempel, ob das dazugehörige XML-Element in dieser Version vorkommt.

Der in [25] entwickelte Algorithmus X-Diff kann mit einer weiteren Besonderheit in XML-Dokumenten umgehen. Die Reihenfolge der XML-Elemente und XML-Attribute kann sich zwischen verschiedenen Versionen ändern, ohne daß dies eine semantische Änderung für das Dokument bedeutet. X-Diff betrachtet deshalb entsprechend der XML-Spezifikation diese XML-Elemente und -Attribute als ungeordnete Sammlung von Objekten. Dadurch wird allerdings eine wesentlich höhere Laufzeit in Kauf genommen. Eine Erkennung von verschobenen Teilen bietet dieser Algorithmus ebenfalls nicht.

Zwei weitere Algorithmen erlauben auch die Erkennung von Verschiebungen und Vertauschungen. Das Verfahren aus [6] versucht durch Referenzen diese Verschiebungen darzustellen. Eine Veränderung von Elementen wird hier allem Anschein nach nicht unterstützt.

Am besten geeignet scheint der Algorithmus von Sudarshan S. Chawathe und Hector Garcia-Molina [4]. Er führt die Zuordnung der Elemente auf ein Graphenproblem zurück, welches die minimalen Kosten für das Matching in einem bipartiten Graphen[12] berechnet. Durch Bewertungsfunktionen (siehe 4.6) wird die Wahrscheinlichkeit eines korrekten Matchings bestimmt, dessen Wert in die Kostenberechnung eingeht.

4.4 Ein Vergleichsalgorithmus für Klassendiagramme

Die in Abschnitt 4.3 vorgestellten Algorithmen sind leider nur bedingt zur Analyse von UML-Klassendiagrammen und deren Unterschieden geeignet. Einer der großen Vorteile, nämlich ihre exakt definierte Struktur, wird von diesen Algorithmen nicht ausgenutzt.

In diesem Abschnitt wird ein Verfahren entwickelt, das diese Informationen berücksichtigt. Dabei werden verschiedene Ideen der im vorherigen Abschnitt erwähnten Algorithmen aufgegriffen und angepasst.

XML dient dazu, Daten in einer strukturierten, textbasierten Form abzulegen. Die Spezifikation dieser Struktur kann mit mehreren Mitteln erfolgen. Eine Möglichkeit sind DTDs, die bereits bei SGML verwendet wurden. Eine neuere, auf XML selbst basierende Beschreibungssprache ist XML-Schema[10]. Diese wurde bereits in Abschnitt 4.1 dazu verwendet, UML-Diagramme mit XML-Dateien zu beschreiben. Aus diesem Schema lassen sich Informationen über den Aufbau einer solchen UDD-Datei gewinnen, die sich für den Vergleich zweier UDD-Dateien verwenden lassen. Neben den Informationen aus dem Schema lassen sich weitere Aussagen über Klassendiagramme und ihre Darstellung als UDD-Datei machen:

1. Der Typ eines XML-Elements kann sich nicht ändern (Beispiel: Aus einer Klasse wird kein Attribut).
2. Die Reihenfolge der Klassen, Attribute und Operationen ist nicht relevant.
3. Die Reihenfolge der Parameter ist relevant.

[12] Ein bipartiter Graph besteht aus zwei Knotenmengen. Kanten verlaufen nur von der einen zur anderen Menge.

4. Klassen, Attribute, Operationen und Parameter besitzen immer das XML-Attribut `name`.
5. Das XML-Attribute `name` ist innerhalb eines Containers eindeutig (Ausnahme sind Operationen, hier muß die Signatur mit einbezogen werden).
6. Das Ändern einer `id` ist nicht erlaubt, da diese zum Wiederfinden eines Elements gedacht ist.
7. Stereotypen können nicht geändert werden (stattdessen wird ein `delete` und `add` ausgeführt).
8. Das `Supplier`-XML-Attribut einer Generalisierung kann nur entlang der eigenen Vererbungshierarchie verändert werden (ansonsten ist es eine andere Generalisierung).
9. Assoziationen können nur entlang der Vererbungshierarchie bewegt werden.

Mit diesem Wissen ist nun nicht nur eine strukturierte, sondern sogar eine semantische Analyse der XML-Datei möglich. Die meisten der im vorherigen Abschnitt vorgestellten Algorithmen konnten nur das Einfügen und Löschen von Informationen erkennen. Einige andere Algorithmen waren auch in der Lage, Vertauschungen zu erkennen, konnten jedoch keine Aussagen darüber machen, ob wirklich eine Vertauschung stattgefunden hatte oder nur zufälligerweise eine Zeile an einer anderen Stelle wieder auftaucht.

Besonders wichtig ist die Erkennung *ähnlicher* Strukturen. Hierunter versteht man Teile, die sich nicht vollständig gleichen, jedoch aus demselben Fragment entstanden sind. Mit den oben beschriebenen „Regeln" lassen sich ähnliche Elemente wiederfinden. Neben den einfachen Transformationsoperationen zum Hinzufügen und Löschen von Elementen besteht nun die Möglichkeit, auch komplexe Transformationen zu erkennen und als Befehle zur Transformation nachzubilden. In Tabelle 3.3 auf Seite 20 wurde bereits eine Liste von Befehlen zusammengestellt. Diese wird nun aufgegriffen und in modifizierter Form hier wiederverwendet.

Im folgenden bezeichnet der Index *b* Elemente des Basisdiagramms und der Index *v* die Elemente der Variante. Das Basisdiagramm selbst wird folglich mit d_b und das Variantendiagramm mit d_v bezeichnet . Bei der Berechnung der Differenzen wird eine Liste von Befehlen generiert, die d_b in d_v transformieren. Als „Befehlssatz" wird `add, delete, modify, move` und `clone` verwendet.

Alle Modellierungselemente können zusätzliche, frei definierbare Eigenschaftswerte (siehe 3.4) besitzen. Da deren Veränderung auch von Relevanz sein kann, werden hierfür ebenfalls Transformationsoperationen generiert. Dasselbe gilt für das XML-Attribut `stereotype`.

- Eigenschaftswerte werden durch `modify`-Befehle bearbeitet. Eine Löschung erreicht man durch das Setzen eines leeren Strings.
- Das XML-Attribute `stereotype` besteht aus einer durch Leerzeichen separierten Liste von Stereotypen. Änderungen an dieser Liste werden durch ein `modify` repräsentiert, welches als `value` die neue Liste der Stereotypen enthält.

Der Vergleich der beiden XML-Datenstrukturen wird anhand einer Baumstruktur durchgeführt. Dabei wird immer nur eine Ebene in diesem Baum betrachtet, d.h. es wird mit den Paketen begonnen, danach werden Klassen analysiert und schließlich die verschiedenen Inhalte dieser Klassen. Der gesamte Ablauf des Verfahrens wird in Abbildung 4.7 auf Seite 36 überblickartig dargestellt.

1. Man betrachte die beiden **Paketmengen** p_b und p_v und versuche, ausgehend von p_v eine Entsprechung für jedes Paket in p_b zu finden. Der dazu notwendige Vergleichsalgorithmus und dessen Bewertungsfunktionen werden in den folgenden Abschnitten (4.5 und 4.6) vorgestellt. Nachdem alle Pakete aus p_v betrachtet wurden, können die notwendigen Transformationsoperationen generiert und d_b aktualisiert werden.

   - Umbenannte Pakete erzeugen einen `modify` des XML-Attributs `name`.
   - Pakete, die keine Entsprechung in p_b haben, sind neu und werden durch ein `add` hinzugefügt.
   - Pakete in p_b ohne Partner in p_v werden durch `delete`-Befehle gelöscht.
   - Das Verschieben von Paketen wird durch ein `modify` des Namens ausgedrückt.

   Bei der Anpassung der Diagramme sind einige Besonderheiten zu beachten. Die zu entfernenden Pakete aus p_b werden vorerst in der Datenstruktur nur als entfernt markiert. Die darin enthaltenen Teilelemente werden dennoch auf Matchings untersucht, da sie vor dem Löschen des Paketes in andere Pakete verschoben worden sein können. Aus demselben Grund werden die Teilelemente der neuen Pakete noch nicht p_v hinzugefügt, da sie ebenfalls durch Verschieben in dieses neue Paket gelangt sein können.

2. Im nächsten Schritt werden nun die beiden Mengen von **Klassen** c_b und c_v auf sich entsprechende Elemente untersucht. Da hierbei die Möglichkeit in Betracht gezogen werden muss, daß Klassen in andere Pakete verschoben wurden, muß bei der Suche nach einer „Partnerklasse" das gesamte Diagramm durchsucht werden.

   Wurde ein Match gefunden, so wird wieder eine entsprechende Befehlsliste generiert. Im allgemeinen ist es nicht möglich, daß ein bereits gematchtes Element nochmals für ein Matching in Frage kommt. Bei Klassen ist dies unter Umständen möglich, wenn beispielsweise eine Klasse dupliziert wurde. Hier finden sich zwei Klassen in c_v, die einer Klasse in c_b sehr ähneln.

   - Verschobene Klassen werden durch ein `move` repräsentiert.
   - Ein modifiziertes `name`-XML-Attribut wird mit einem `modify`-Befehl umgesetzt.
   - Klassen, die nicht in c_b wiederzufinden sind, müssen durch ein `add` hinzugefügt werden.
   - die übrigen Klassen aus c_b werden per `delete` gelöscht.
   - Klassen aus c_b, die mehrfach als Partner in Frage kommen, werden durch `clone` dupliziert.
   - Änderung des Klassentyps (beispielsweise von Interface zu Klasse) wird durch `modify` nachgebildet.
   - `modify` wird auch für die boolean-XML-Attribute `abstract` und `final` verwendet.

   Die bereits bei den Paketen gemachten Anmerkungen bezüglich der Anpassung von d_b sind hier ebenfalls zu beachten. Verschobene Klassen werden sofort inklusive ihrem Inhalt verschoben.

3. Im nächsten Schritt werden die **Generalisierungen** betrachtet. Wie bereits erwähnt, erscheinen nur Verschiebungen innerhalb der eigenen Vererbungshierarchie sinnvoll. Dies wird durch die im nächsten Abschnitt entwickelte Bewertungsfunktion für Generalisierungen nachgebildet. Folgende Befehle können generiert werden.

   - `modify`-Befehl für ein geändertes `supplier`-XML-Attribute
   - `add` für neue Generalisierungen
   - `delete` zur Entfernung von Generalisierungen

   Die Auswirkungen der Befehle können nun direkt am Ausgangsdiagramm nachvollzogen werden, da Generalisierungen keine Teilelemente mehr enthalten.

4. **Attribute** werden als nächstes verglichen. Diese sollten vor den Assoziationen bearbeitet werden, da ein verschobenes Attribut unter Umständen eine verschobene Assoziation nach sich zieht. Durch das Reorganisieren der Datenstruktur nach der Abarbeitung der Attribute kann dies einfacher erkannt werden. Die Erkennung von Attributen gestaltet sich etwas schwierig, da sie nur wenige Eigenschaften aufweisen, an denen man sie wiedererkennen kann (Name und Typ). Folgende Transformationsoperationen sind möglich.

   - `move` verschiebt Attribute.
   - `modify` ändert die XML-Attribute `name`, `static`, `multiplicity` oder `visibility`.
   - `add` wird zum Hinzufügen eines Attributs verwendet.
   - `delete` dient zum Entfernen von Attributen.

   Diese Befehle werden ebenfalls auf dem Ausgangsdiagramm d_b ausgeführt.

5. Nachdem nun die Attribute verarbeitet wurden können die **Assoziationen** betrachtet werden. Auch hier werden die Auswirkungen der Befehle sofort im Ausgangsdiagramm nachvollzogen.

   - `move`-Befehle verschieben Assoziationen in andere Klassen.
   - Verschiebungen müssen ggf. das `client`-XML-Attribut durch ein `modify` anpassen.
   - Weitere `modify`-Befehle ändern `supplier`, `supplier_multiplicity` oder `client_multiplicity`.
   - Auch die Änderungen des Assoziationstyps und des XML-Attributs `directed` werden durch `modify` notiert.
   - `add` fügt Assoziationen hinzu.
   - `delete` entfernt Assoziationen.

6. Im letzten Schritt werden die **Operationen** und ihre Parameter betrachtet. Hier besteht neben dem einfachen Bewegen einer Operation zusätzlich die Möglichkeit, diese zu klonen. Dies tritt beispielsweise auf, wenn eine Methode dupliziert wird und dann andere Parameter oder andere Rückgabetypen erhält. Auch das Erzeugen von nicht-abstrakten Operationen durch Klonen aus der abstrakten Superklasse ist denkbar. Angenommen man erzeugt eine

neue Klasse, die von einer abstrakten Klasse oder einem Interface erbt. Diese Klasse übernimmt viele Eigenschaften der übergeordneten Klasse und implementiert einige der darin definierten Operationen. Dies läßt sich durch einen `clone`-Befehl nachbilden, da die Deklaration der Operationen mit denen der Superklasse übereinstimmen und von dort *kopiert* werden kann.

- `move` verschiebt eine Operation in eine andere Klasse.
- `clone` erzeugt eine Kopie der Operationen.
- `modify` ändert `name`, `returntype`, `static`, `abstract` oder `visibility`.
- `add` fügt eine neue Operation hinzu.
- `delete` löscht eine Operation.

Bei der Untersuchung der **Parameter** ist die Reihenfolge zu beachten. Man hat somit ein recht gutes Erkennungsmerkmal für Parameter. Eine Änderung des Namens oder Typs ist prinzipiell erkennbar, kann jedoch bei vielen ähnlichen Operationen zu Problemen führen. Folgende Befehle werden durch Änderungen an der Signatur generiert.

- `modify` ändert `name` oder `type`.
- `add` fügt einen Parameter hinzu. Da die Reihenfolge der Parameter relevant ist, kann hier optional eine Position bestimmt werden, an der der Parameter eingefügt werden soll.
- `delete` entfernt einen Parameter.[13]

Wurden diese Schritte durchgeführt, so ist nun zu erwarten, daß d_b vollständig in d_v transformiert wurde. Einziger Unterschied sind die als löschbar markierten Pakete und Klassen. Diese enthalten keine weiteren Teilelemente mehr und können nun entfernt werden.

4.5 Die Vergleichsfunktion

Die Vergleichsfunktion FINDMATCH$(\ldots)$ versucht für ein Element e_v aus d_v das passende Element e_b in d_b wiederzufinden. Hierzu bedient sie sich einer Menge von Bewertungsfunktionen. Diese entscheiden anhand bestimmter Merkmale, ob zwei Elemente gleich sind oder zumindest voneinander abstammen. Durch Anpassungen dieser Bewertungsfunktionen oder Hinzufügen weiterer Funktionen kann unter Umständen die Erkennungsgenauigkeit verbessert werden.

FINDMATCH$(\ldots)$ testet jedes Element der Variantenmenge (p_v, c_v etc.) mit der Ausgangsmenge (p_b, c_v etc.). Dieser Test besteht aus der Berechnung aller für diesen Elementtyp geeigneten Bewertungsfunktionen. Eine Bewertungsfunktionen erhält als Parameter die beiden zu vergleichenden Elemente e_v und e_b. Über diese hat sie auch Zugriff auf die entsprechenden Containerelemente. Der Rückgabewert ist vom Typ Integer und liegt zwischen -16 und $+16$. Positive Werte drücken ein hohe Übereinstimmung aus, negative Werte bezeichnen dieses Matching als eher unwahrscheinlich. Diese Einteilung wurde gewählt, da Bewertungsfunktionen im allgemeinen keine kontinuierliche Aussage machen können und somit Integer-Werte eine einfachere Berechnung ermöglichen. Die Verwendung von Wahrscheinlichkeiten erschien nicht sinnvoll, da

[13] Das Ändern der Reihenfolge von Parametern durch Verschieben oder Vertauschen wird nicht erkannt und durch die Befehle add und delete umgesetzt.

Bewertungsfunktionen sowohl gute als auch schlechte Matchings erkennen, was sich durch Wahrscheinlichkeiten nur schwer ausdrücken läßt. Die Summe dieser Bewertungsfunktionen bildet das Gesamtergebnis für die Gleichheit zweier Elemente. Diese zieht auch in Betracht, daß eine höhere Anzahl von Bewertungsfunktionen eine genauere Aussage über ein Matching machen können. Bewertungsfunktionen, die 0 zurückgeben, zeigen so, daß sie keine Aussage zu einem Matching machen können.

Zusätzlich zu den Bewertungsfunktionen gibt es noch ein eindeutiges Identifikationsmerkmal. IDMATCH(...) kann ein „garantiert richtiges" Matching aufgrund der ID erkennen. Wenn also jedes Element eines Klassendiagramms eine unveränderliche und eindeutige Bezeichnung erhält, die sich auch in Folgeversionen nicht verändert, so kann damit ein garantiert richtiges Matching gefunden werden. Liefert diese Funktion also den Wert `true` zurück, so wird der Vergleich abgebrochen, da das passende Element gefunden wurde. Das in Kapitel 1 erwähnte CASE-Tool Together verwendet zur Referenzierung aller Modellelemente ebenfalls eindeutige Bezeichner. Diese `oiref`-Elemente sind allerdings nicht unveränderlich, sondern werden bei Verschiebungen und Umbenennungen angepasst und erfüllen somit nicht die hier geforderten Eigenschaften.

Der Vergleich wird zuerst nur mit einer Teilmenge aus d_b durchgeführt. Hierzu betrachtet man das Containerelement e_v. Dieses Element wurde in einem vorherigen Vergleichslauf (in einer darüber liegenden Ebene) bereits mit einem Containerelement in d_b gematched. Es ist davon auszugehen, daß das gesuchte Element e_b in diesem Containerelement liegt. Deshalb wird der Vergleich zuerst mit dessen Elementen (X_b') durchgeführt.

Wurde der Vergleich mit allen Elementen aus X_b' durchgeführt, so wird der Maximalwert betrachtet, der hierbei aufgetreten ist. Liegt er über eine gewissen Schwelle LIMIT, so wird das dazugehörige Element als passend betrachtet. Ist dies nicht der Fall wird ein erweiterter Vergleich mit allen Elementen desselben Typs aus d_b durchgeführt. Findet sich hier ebenfalls kein Wert über der gesetzten Schwelle, so wurde ein neues Element in d_v gefunden, welches per `add` hinzugefügt wird. Dieser Ablauf des Vergleichs ist in Abbildung 4.6 nochmals kurz dargestellt.

4.6 Die Bewertungsfunktionen

Im folgenden sind einige Bewertungsfunktionen aufgeführt. Zuerst werden Funktionen aufgeführt, die relativ gut eine Ähnlichkeit zweier Elemente erkennen können. Die weiter unten liegenden Funktionen sind ungenauer und liefern im allgemeinen nur einen geringen Beitrag. Zusätzlich zu den aufgeührten Merkmalen gibt es noch die Möglichkeit, daß die beiden zu vergleichenden Elemente dieselbe ID besitzen. Die hier geforderte ID ist über mehrere Versionen hinweg eindeutig und unveränderlich, deshalb ist sie ein sichere Methode, Elemente in verschiedenen Versionen wiederzufinden. Liegt also ein sogenannter IDMATCH(...) vor, müssen die Bewertungsfunktionen nicht weiter evaluiert werden. Trifft der *Name* zu, so ist das ebenfalls ein guter Treffer, was zu einer entsprechend hohen Bewertungswert führt. Da *Typen* weniger eindeutig sind, wird hier ein geringerer Wert für einen Match angesetzt.

location Zuerst wird versucht, ein Element an der Stelle (Paket oder Klasse) in d_b wiederzufinden, an der es sich auch in d_v befindet. Dieser Ort ist wahrscheinlicher als andere. Hat sich das Element verschoben, so ist es wahrscheinlicher, daß es nur entlang der Vererbungshierarchie verschoben wurde. Die Refactoring-Schritte *pull up* oder *push down* verschieben beispielsweise Attribute oder Operationen in eine Ober- oder Unterklasse.

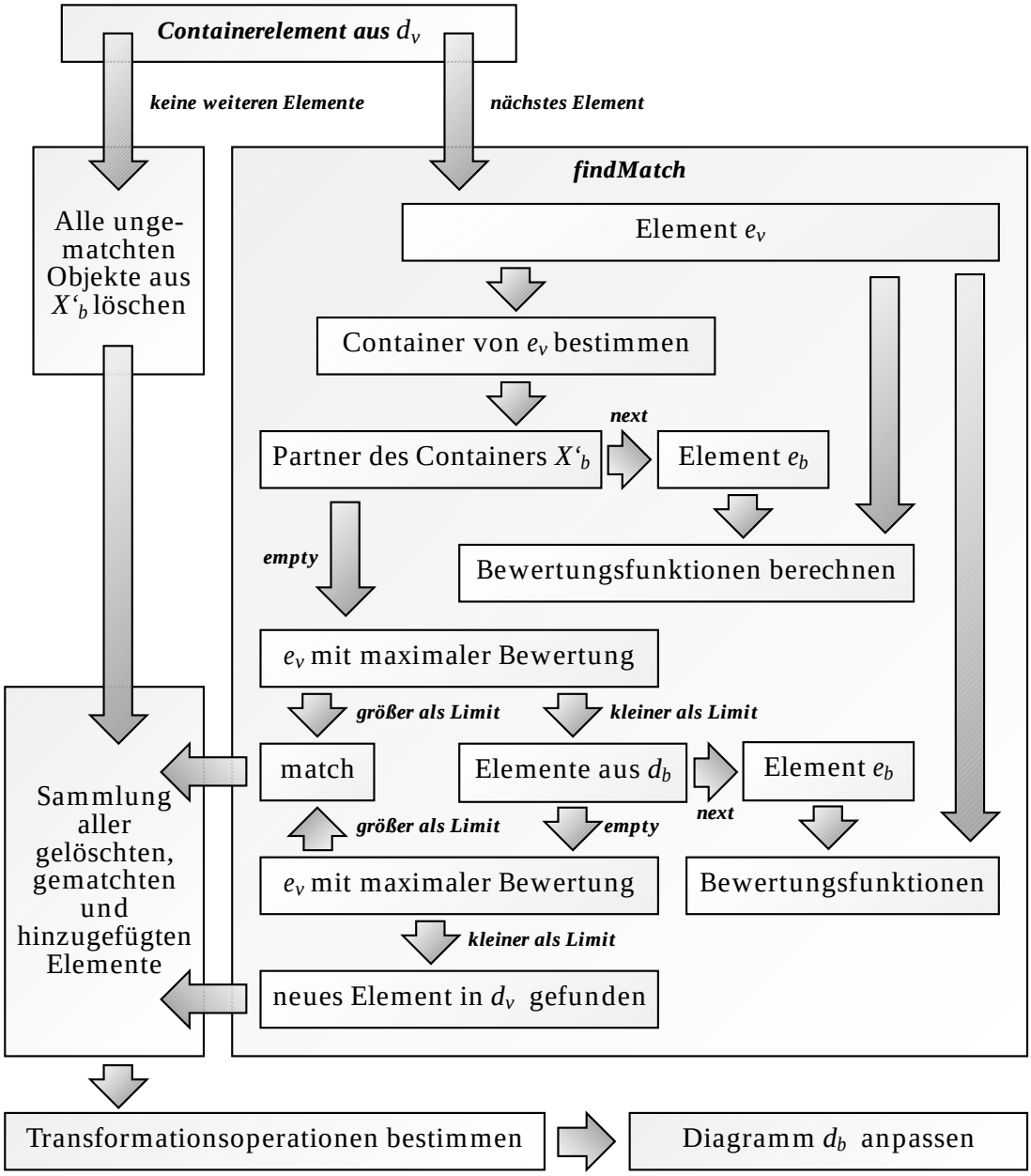


Abbildung 4.6: Ablauf der *findmatch*-Funktion

name Nahezu alle Modellierungselemente besitzen einen Namen. Mit fallendem Bewertungswert wird von folgendem ausgegangen: Name gleich, Name mit anderer Groß/Kleinschreibung, Name mit hinzugefügtem oder entferntem Ende oder Anfang, anderer Name.

stereotype Viele Stereotypen bezeichnen nicht nur lokale Eigenschaften sondern auch globale Verhaltensweisen. Sie sind somit unter Umständen nur einmalig in einem Container vorhanden, und können dadurch zur Bewertung herangezogen werden.

type Der Typ eines Elements ist ebenfalls ein gutes Erkennungsmerkmal. Bei Operationen kann der Typ des Rückgabewertes betrachtet werden, bei Assoziationen ist der supplier interessant. Treten hier Veränderungen auf, so sollte man wieder entlang der Vererbungshierarchie suchen.

signature Die Signatur ist ein eindeutiges Merkmal einer Operation. Ist sie gleich oder unterliegt nur leichten Veränderungen, so kann man hier eine hohe Trefferwahrscheinlichkeit ansetzen.

parts Sowohl bei Paketen als auch Klassen lassen sich die Inhalte auf Gleichheit überprüfen. Dies sollte nur eine ungefähre Abschätzung beinhalten, da der eigentliche Vergleich erst in einem späteren Schritt erfolgt.

predecessor Im allgemeinen wird die Reihenfolge der Attribute und Operationen nicht betrachtet. Wenn jedoch die anderen Erkennungsmerkmale nicht ausreichen, kann überprüft werden, ob der Vorgänger in beiden Containern derselbe ist.

4.7 Postanalyse

Als Ergebnis der Analysephase erhält man eine Liste von Befehlen, die d_b in d_v transformieren. Außerdem enthält die Datenstruktur von d_v nun Verweise zu den gematchten Partnern aus d_b. In einem weiteren Analyseschritt lassen sich daraus noch weitere Informationen gewinnen. Hier eine Auswahl:

- Eine Umsortierung der Liste gruppiert die Änderungen einer Klasse.
- Eine Liste der geänderten Elemente kann generiert werden.
- Die Befehlsliste kann nach typischen Refactoring-Schritten durchsucht werden.
- Metriken und Differenzen zwischen Metriken können berechnet werden. Beispielsweise ließe sich berechnen, wie viele Klassen hinzugefügt oder gelöscht wurden.

4.8 Datenformat für Transformationsoperationen

Die gewonnene Befehlsliste soll nun zur weiteren Verarbeitung in eine Datei geschrieben werden. Hierzu wurde wieder XML als Beschreibungssprache gewählt. Eine solche Datei trägt die Endung `.cdt`, ihr Aufbau wird im folgenden beschrieben.

Das Wurzelelement wird mit `classdiagramtransformer` bezeichnet. Die Unterelemente sind die einzelnen Transformationsoperationen.

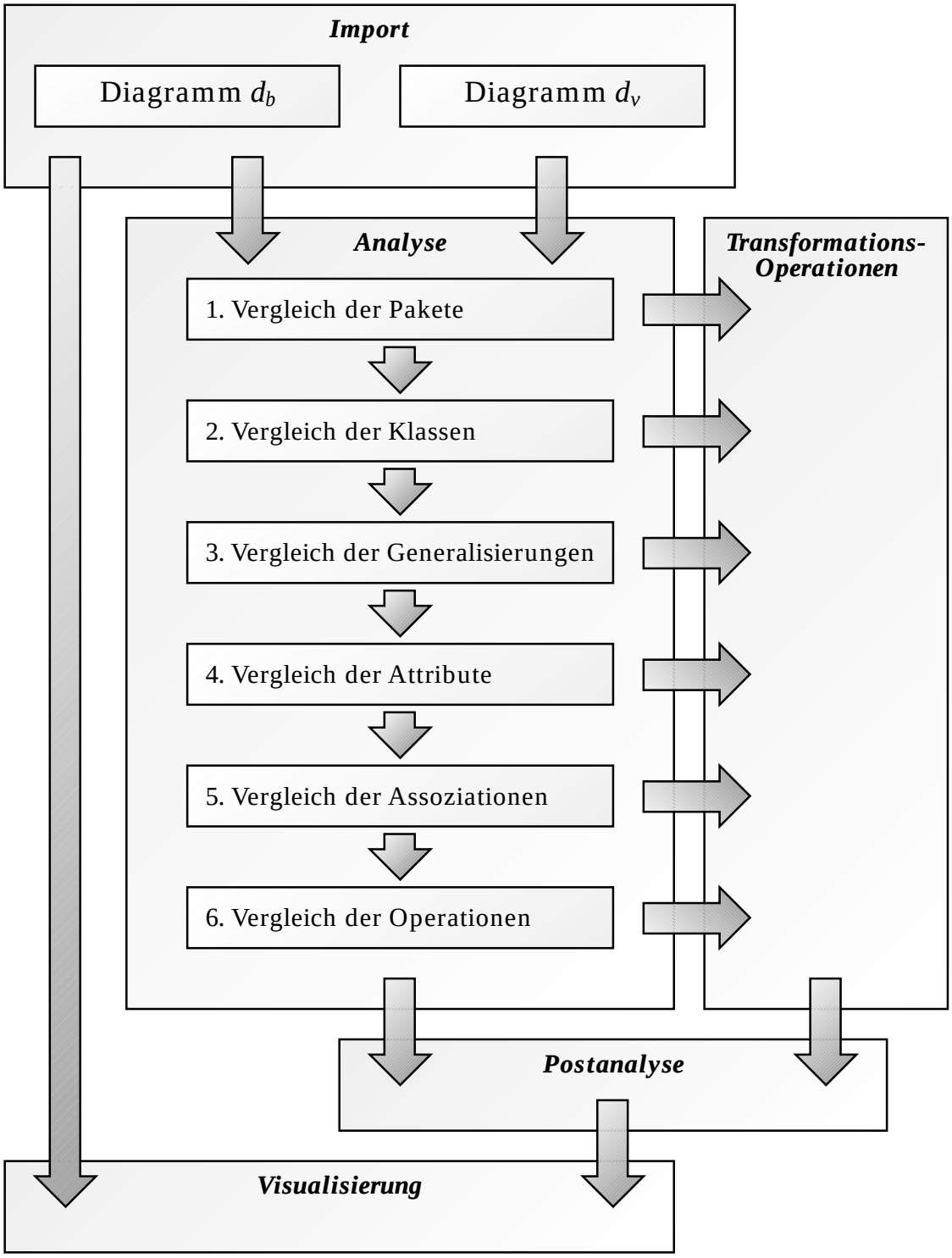


Abbildung 4.7: Überblick über den Ablauf von UMLDiff für Klassendiagramme

Es existieren sechs verschiedene Befehlstypen. Alle besitzen das XML-Attribut `path`. Dieses bezeichnet eindeutig ein Element in dem Klassendiagramm, auf welches der Befehl angewendet wird. Die weiteren XML-Attribute sind in Tabelle 4.1 aufgeführt. Der Aufbau von `path` soll hier näher erläutert werden.

Pakete Name des Paketes abgeschlossen mit einem `/`

Klassen Paketname und Klassenname, die durch einen `/` getrennt sind.

Alle Element einer Klasse erhalten als Prefix den Klassenpfad. Es folgt der Teilpfad des Elements.

Attribute Als Teilfpad wird ein Punkt und der Name des Attributs verwendet.

Operationen Ein Punkt, gefolgt von dem Namen und einer Parametertypliste (`<parametertype>`, ...)

Assoziationen werden durch `-<name>.<supplier>` beschrieben.

Der Teilpfad von Generalisierungen besteht aus einem `-` und dem Namen des Pakets und der Klasse, von der generalisiert wird.

Parameter von Operationen werden durch einen angehängten Punkt und einem Index spezifiziert.

Das XML-Attribut `tag` wird bei `modify` dazu verwendet, das zu ändernde Objekt zu beschreiben. Dies können sowohl XML-Attribute eines Modellierungselements als auch Schlüsselworte eines Eigenschaftswerts sein. Mögliche Werte für `tag` sind in folgender Aufzählung beschrieben:

Allgemein name, type, id[14],

Wahrheitswerte abstract, final, static, directed

Assoziationen und Generalisierungen associationtype, client, client_multiplicity, supplier, supplier_multiplicity

Sonstige visibility, stereotype, returntype, classtype, multiplicity

Eigenschaftswerte Jeder beliebige String

XML-Element	XML-Attribut	Beschreibung
add	Udd-Element	Das hinzuzufügende Element in der XML-Darstellung
delete	virtual	Ist `virtual true`, so wird das Element nur als entfernt markiert.
modify	tag	XML-Attribut oder Eigenschaftswert, welcher geändert werden soll.
	value	Der neue Wert des XML-Attributs oder des Eigenschaftswerts.
move/clone	destination	Der Pfad zu dem Containerelement in das verschoben/kopiert werden soll.

Tabelle 4.1: Transformationsoperation in ihrer XML-Darstellung

In Abbildung 4.8 ist ein kurzes Beispiel aufgeführt. Die dort angegebenen Transformationsoperationen beziehen sich auf die beiden Klassendiagramme, die im ersten Kapitel auf Seite 9 als Beispiel verwendet wurden.

[14] Änderungen einer ID werden beispielsweise für `clone` benötigt

```
<?xml version="1.0" encoding="UTF-8"?>
<classdiagramtransformations name="base2variant">
  <nop path="base/" reason="-" />
  <modify path="base/" tag="name">variant</modify>
  <add path="variant/">
    <class name="Discussion" classtype="Class">
    </class>
  </add>
  <move path="variant/Concert.setLocation(String)" destinati-
on="variant/Event" shadow="true" />
  <move path="variant/Concert.location" destination="variant/Event" sha-
dow="true" />
  <modify path="variant/Concert-base/Event" tag="supplier">variant/Event</modify>
  <delete path="variant/Reading.setReader(String)" virtual="true" />
  <add path="variant/Reading">
    <operation name="addReader" returntype="void" visibility="public">
      <parameter name="reader" type="String" />
    </operation>
  </add>
  <add path="variant/Reading">
    <operation name="removeReader" returntype="void" visibility="public">
      <parameter name="reader" type="String" />
    </operation>
  </add>
  <modify path="variant/Reading.reader" tag="type">String[]</modify>
  <modify path="variant/Reading.reader" tag="name">readers</modify>
  <modify path="variant/Reading-base/Event" tag="supplier">variant/Event</modify>
  <add path="variant/Discussion">
    <operation name="setTopic" returntype="void" visibility="public">
      <parameter name="topic" type="String"/>
    </operation>
  </add>
  <add path="variant/Discussion">
    <attribute name="topic" type="String" visibility="private"/>
  </add>
  <add path="variant/Discussion">
    <generalization supplier="variant/Event"/>
  </add>
</classdiagramtransformations>
```

Abbildung 4.8: Beispiel: CDF (ClassDiagramTransformer)

4.9 Visualisierung

Mit Hilfe des Klassendiagramms d_b und der Transformationsdatei kann nun eine Visualisierung der Daten vorgenommen werden. Im Prototyp wurden mehrere Visualisierungen entwickelt, die hier kurz vorgestellt werden sollen.

Zu Dokumentationszwecken und für Reports scheint eine Darstellung in Form einer HTML-Datei sinnvoll. Hier werden die Transformationen tabellarisch dargestellt. Der Report für das in Kapitel 1 vorgestellte Beispiel ist in Abbildung 4.9 auf Seite 40 zu sehen.

Als weitere Darstellungsmöglichkeit bieten sich abgewandelte Klassendiagramme an. Hier wird als Ausgangsbasis eine Darstellung des Klassendiagramms d_b gewählt. Durch farbige Hinterlegungen werden nun die Unterschiede dargestellt. In Abbildung 4.9 ist eine Legende der verwendeten Farben zu sehen. Sie werden sowohl für die Tabelle des Reports als auch das Diagramm selbst verwendet. Die Bezeichnungen der verwendeten Farben entsprechen der Netscape-Farbtabelle[15]. Blautöne wurden für neu hinzugefügte oder geklonte Elemente verwendet. Grüntöne zeigen verschobene Elemente an. Ist der Hintergrund orange, so wurde das Element gelöscht. Sowohl bei verschobenen als auch bei neuen oder geklonten Elementen ist eine Besonderheit zu beachten. Diese Elemente kommen in beiden Diagrammen vor und können unterschiedliche Eigenschaften aufweisen, die durch `modify`-Operationen nachgebildet wurden. Um darzustellen, daß hier Unterschiede vorhanden sind, wird der Text des Elements violett eingefärbt.

Die Darstellung der Diagramme sollte nach Möglichkeit in einem Webbrowser und damit plattformübergreifend erfolgen. Hier existieren mehrere Möglichkeiten, dies zu realisieren.

DHTML Durch das Layer-Konzept ist es möglich, Elemente in HTML-Seiten frei zu plazieren. Da jedoch viele Inkompatibilitäten zwischen den verschiedenen Browsern und ihren Implementationen auf verschiedenen Plattformen existieren, ist die Verwendung dieses Konstrukts sehr fehleranfällig.

JPEG/GIF/PNG Die gängigen Bildformate für Webseiten sind rein pixelbasiert. Dadurch wird Text nicht als Text, sondern als Bitmapgraphik repräsentiert und kann nicht weiter verwendet werden. Das Vergrößern und Verkleinern ist nur schwierig möglich und bringt starke Qualitätsverluste mit sich. Animation ist nur in eingeschränktem Rahmen bei GIF-Dateien möglich.

VRML Also Standard für dreidimensionale Darstellung im Webbrowser ist VRML relativ weit verbreitet, allerdings ist bisher kein Browser auf dem Markt, der von Haus aus VRML unterstützt. Eine dreidimensionale Darstellung wird hier nicht benötigt, deshalb erscheint ein Einsatz von VRML nicht sinnvoll.

PDF Zur Darstellung von PDF-Dokumenten wird in den meisten Fällen der Acrobat® Reader[16] von Adobe eingesetzt. Er hat eine ähnlich starke Verbreitung wie Webbrowser. Das PDF-Format weist jedoch einige Nachteile auf. So ist die Verarbeitung solcher Dateien aufgrund des komplexen, an Postscript angelehnten Befehlssatzes sehr schwierig. Außerdem bietet PDF keine direkte Möglichkeit zur Animation.

[15] `http://selfhtml.teamone.de/diverses/anzeige/farbnamen_netscape.htm`
[16] `http://www.adobe.com/products/acrobat/readstep2.html`

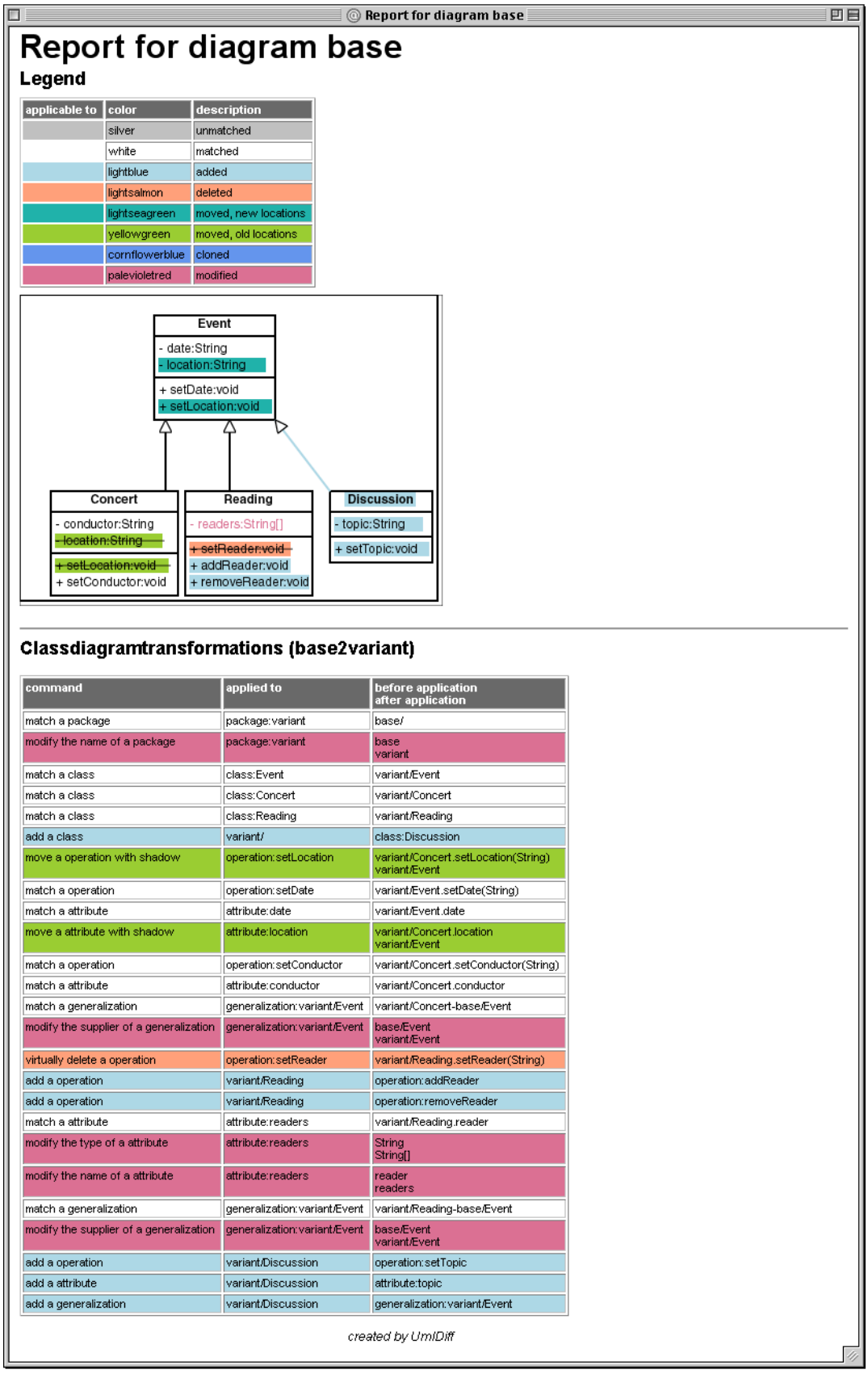


Abbildung 4.9: Beispiel eines HTML-Reports

SVG Das Scalable Vector Graphics[11] Format wurde erst vor kurzem von der W3C-Organisation verabschiedet. Es basiert auf XML und ermöglicht die vektororientierte und animierte Darstellung von Graphiken innerhalb des Webbrowsers. Zur Zeit ist die Einbindung in den Webbrowser noch über Plugins gelöst, jedoch wird bereits an der Integration in den Browser Mozilla[17] gearbeitet. Das Plugin von Adobe[18] zur Darstellung von SVG ist für die Plattformen Microsoft Windows®, Apple Mac® OS sowie Linux® verfügbar. Die Vorteile dieses Formats liegen in dem allgemein anerkannten Standard und der Verwendung von XML. Die Darstellung erfolgt vektororientiert und in hoher Qualität. Verkleinern und Vergrößern ist mit dem Plugin ohne Qualitätsverluste möglich. Animation wird unterstützt.

Das SVG-Format scheint somit alle gewünschten Eigenschaften zu erfüllen. Um jedoch für verschiedene Diagrammtypen[19] eine möglichst einfache Implementierung zu ermöglichen wurde ein Zwischenformat entwickelt, welches speziell für Diagramme konstruiert wurde. Dieses Format, welches als Generic Diagram Language (GDL)[20] bezeichnet wird, findet auch zur Darstellung von Sequenzdiagrammen in Kapitel 6 Verwendung. Es basiert ebenfalls auf XML und beschränkt sich auf die zur Darstellung von UML-Diagrammen benötigten Graphikprimitive.

Anhand eines kurzen Beispiels sollen die Unterschiede zwischen GDL und SVG erläutert werden. In Abbildung 4.10 wird ein einfaches Klassendiagramm durch eine GDL-Datei dargestellt. Wie man sieht, werden Rechtecke (engl. rectangles) als Container verwendet, in denen andere Objekte untereinander dargestellt werden können. Dies ermöglich eine kompakte Darstellung von Mehrzeiligen Textkästen. Verbindungen von Rechtecken werden durch Linienzüge (engl. polylines) dargestellt. Diese sind nicht an feste Punkte gebunden, sondern verweisen einfach auf das Rechteck, welches sie berühren sollen.

SVG ist ein wesentlich komplexeres Beschreibungsformat für Vektorgraphiken, welches auch die Möglichkeit zur Animation bietet. Konvertiert man die in Abbildung 4.10 dargestellte GDL-Datei in das SVG-Format, so werden für alle graphischen Elemente die notwendigen Koordinaten berechnet. Die daraus entstehende Datei ist um ein vielfaches größer, kann jedoch nun mit jedem SVG-fähigen Darstellungswerkzeug (Batik bzw. FOP[21] oder Adobe SVG Viewer[22]) angezeigt werden. Ein Ausschnitt des Konvertierergebnisses ist in 4.11 dargestellt. Eine Darstellung dieser SVG-Datei mit dem Adobe SVG Viewer ist in Abbildung 4.12 zu finden. Da SVG ein Vektorformat ist, erlaubt der SVG Viewer eine verlustfreie Vergrößerung und Verkleinerung der Graphik. Eine Eigenschaft, die vor allem für umfangreiche Diagramme sehr nützlich ist. Außerdem ist es möglich, Textpassagen aus den Graphiken herauszukopieren, was bei einer Darstellung durch ein pixelorientiertes Bild nicht möglich wäre.

Die Konvertierung von GDL nach SVG wird von einem weiteren Java-Paket durchgeführt. Hierzu ist es nicht unbedingt notwendig, die XML-Datei abzuspeichern. Zur Bearbeitung von XML-Datenstrukturen gibt es eine standardisierte API namens DOM[23], die auch den direkten Austausch

[17] `http://www.mozilla.org/projects/svg/`
[18] `http://www.adobe.com/svg/viewer/install/main.html`
[19] Dies betrifft sowohl UML-Diagrammtypen als auch andere Diagramme oder graphenähnliche Konstrukte.
[20] `http://www.girschick.net/martin/study/gdl/`
[21] `http://xml.apache.org`
[22] `http://www.adobe.com/svg/viewer/install/`
[23] Document Object Model

```
<?xml version="1.0" encoding="UTF-8"?>
<diagram>
  <data>
    <rectangle id="Klasse" x="5" y="5">
      <text bold="true">Klasse</text>
      <divider/>
      <text>+attribut:type</text>
    <divider/>
    <text>+implementierteMethode:void</text>
    </rectangle>
    <rectangle id="Interface" x="50" y="150">
      <text bold="true" italic="true">Interface</text>
      <divider/>
      <divider/>
      <text italic="true"> abstrakteMethode:void</text>
    </rectangle>
    <polyline markerEnd="hollow">
      <point ref="Interface"/>
      <text>Generalisierung</text>
      <point ref="Klasse"/>
    </polyline>
  </data>
</diagram>
```

Abbildung 4.10: Beispiel: Generic Diagram Language (GDL)

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="gdl.css" type="text/css"?>
<svg class="diagramBody" xmlns="http://www.w3.org/2000/svg"
 xmlns:xlink="http://www.w3.org/1999/xlink" id="svgroot"
 width="220" height="218">
 <g class="default" id="uid0_children">
  ...
  <g>
   <g id="Klasse" style="clip-path:url(#Klasse_clip)"
    transform="translate(5 5)">
    <clipPath id="Klasse_clip">
     <rect width="167" height="69" />
    </clipPath>
    <rect x="0" y="0" width="167" height="69" class="container"
     id="Klasse_rect" style="stroke-width:2px;stroke:black;fill:white" />
    <g class="default" id="Klasse_children">
     <g>
      <g transform="translate(0 0)">
       <text x="0" y="12" class="textBasic" id="uid1"
        style="font-size:12px;font-weight:bold;fill:black;text-anchor:start"
        xml:space="preserve">  Klasse
       </text>
      </g>
      <line x1="0" y1="4" x2="600" y2="4" class="divider"
       id="uid2_line1" style="stroke-width:2px;stroke:black"
       transform="translate(0 16)" />
      ...
```

Abbildung 4.11: Beispiel: Scalable Vector Graphics (SVG)

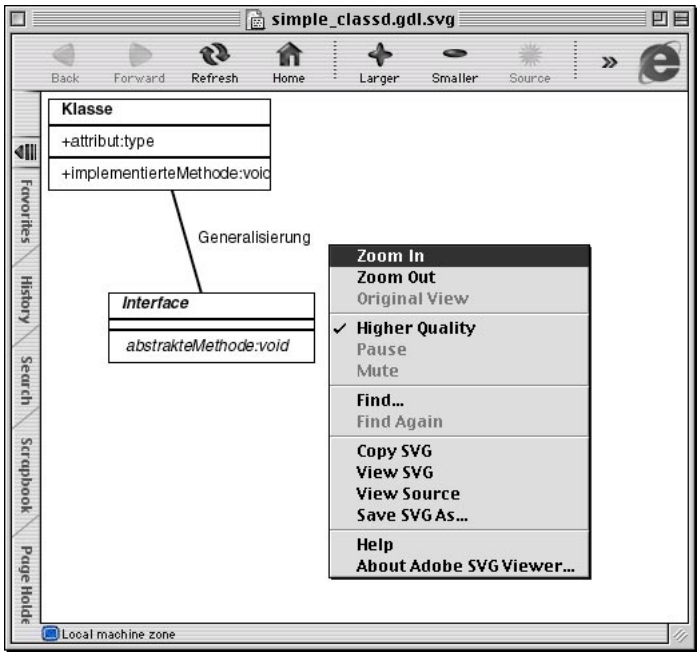


Abbildung 4.12: Beispiel: SVG-Grafik, dargestellt mit dem Adobe SVG Viewer

von XML-Dokumenten zwischen Programmen ermöglicht. Damit ist es also möglich, die Konvertierung von GDL nach SVG direkt in das Differenzanalysewerkzeug einzubinden.

Der gesamte Analysevorgang und die Visualisierung läßt sich also in folgenden Schritten zusammenfassen:

1. Erzeugen der Klassendiagramme. Dies kann mit Hilfe eines Softwareentwicklungswerkzeugs geschehen oder aus einem Reverse-Engineering-Prozess.
2. Übergabe der Klassendiagramme an das Analysewerkzeug.
   (a) Einlesen der Diagramme
   (b) Analyse der Diagramme
   (c) Generierung der Transformationsoperationen
   (d) Erzeugung eines kombinierten, eingefärbten Klassendiagrammes
3. Darstellung der Ergebnisse (HTML oder SVG) mit einem geeignetem Darstellungswerkzeug.

4.10 Bemerkungen zur Implementierung

Als Implementationssprache wurde Java[24] gewählt. Zur Verarbeitung der XML-Dateien wurde JDOM[25] verwendet. Diese API ist wesentlich besser an Java angepaßt als die sehr allgemein gehaltenen APIs DOM und SAX.

[24] `http://java.sun.com`
[25] `http://www.jdom.org`

UMLDiff wurde in drei Java-Packages realisiert, die als Unterpakete von SeDiTeC angelegt wurden.

pi.seditec.umldiff Enthält Klassen zur Generierung von HTML-Reports, eine Klasse zum Logging der Programmausgaben sowie verschiedene Basisinterfaces der UDD-Datenstruktur und Klassen zur Verarbeitung von XML-Dateien. Ein Klassendiagramm dieses Paketes befindet sich auf Seite 44.

pi.seditec.umldiff.cld Neben den Klassen, die die Datenstruktur von UML-Klassendiagrammen nachbilden befindet, sich hier die Klasse `CldDifferenceFinder`, die den eigentlichen Vergleich von Klassendiagrammen durchführt.

pi.seditec.umldiff.sqd Befaßt sich mit der Analyse von Sequenzdiagrammen und wird in Kapitel 6 näher erläutert.

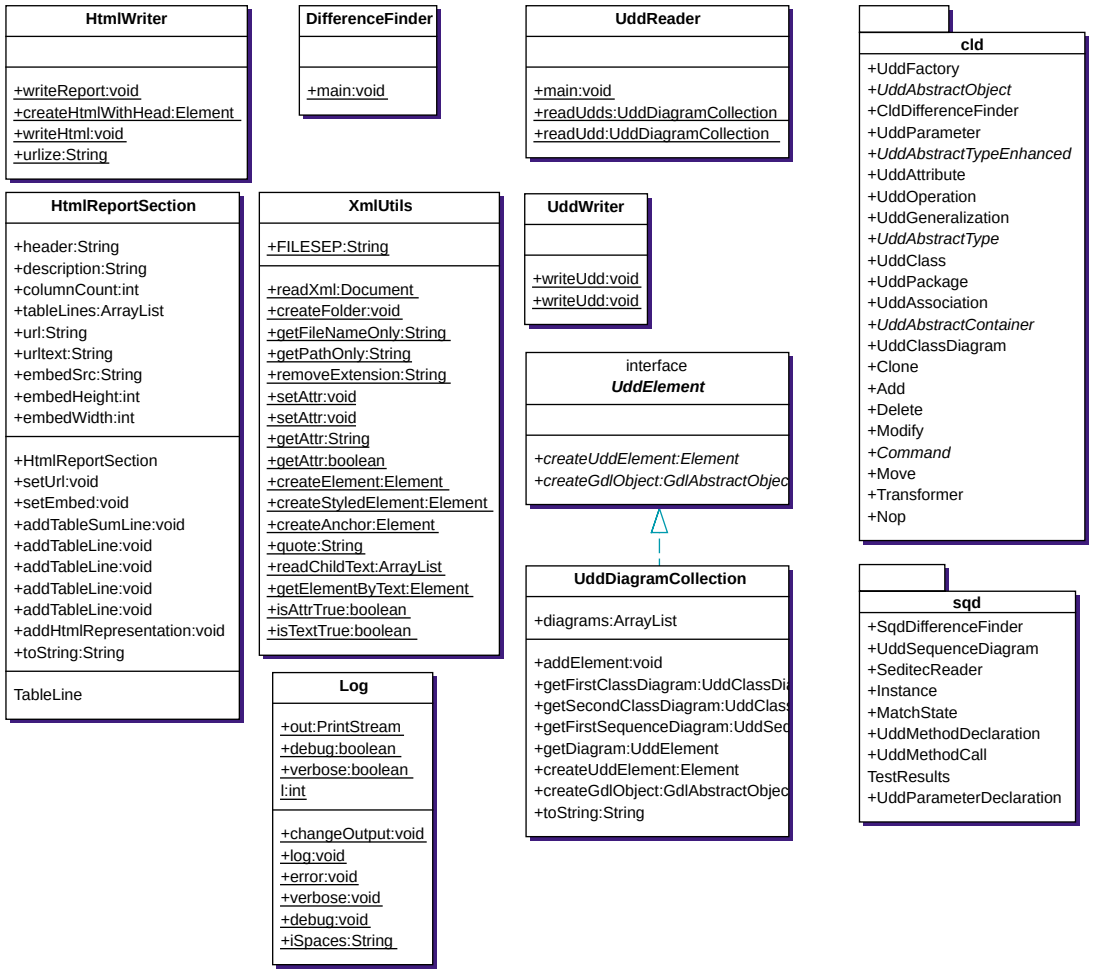


Abbildung 4.13: Das Klassendiagramm des Paketes `pi.seditec.umldiff`

Auf der Grundlage der in Abbildung 4.4 dargestellten XML-Datenstruktur wurde eine Klassenstruktur entworfen, die in Abbildung 4.14 zu sehen ist. Die verschiedenen Eigenschaften der

Klassendiagrammelemente wurde auf vier Grundtypen zurückgeführt, die die Basis der Vererbungshierarchie bilden.

UddAbstractObject Basisklasse aller Klassendiagrammelemente. Enthält neben dem Namen des Elements seine eindeutige ID, falls diese vorhanden ist. Desweiteren sind hier die Stereotypen und Eigenschaftswerte abgelegt.

UddAbstractType Fügt der Klasse UddAbstractObject die Attribute `type` und `mult` (multiplicity) hinzu.

UddAbstractTypeEnhanced Erweitert UddAbstractType um die Sichbarkeit (`vis`) sowie den Wahrheitswerten `isFinal` und `isStatic`.

UddAbstractContainer Erbt von UddAbstractObject und enthält außerdem Referenzen auf weitere UddAbstractObject-Elemente.

Die einzelnen Modellierungselemente von Klassendiagrammen werden nun durch Java-Klassen nachgebildet. Sie erweitern dazu eine der abstrakten Klassen. Diese fügen weitere Attribute und benötigte Operationen hinzu.

UddAssociation erbt von UddAbstractType und verwendet den dortigen Typ und Vielfachheit (multiplicity) zur Darstellung des suppliers der Assoziation. Neue Attribute sind `client_mult` (Vielfachheit auf Client-Seite), `associationtype` (Assoziation, Komposition oder Aggregation) und `isDirected` (Assoziation ist gerichtet).

UddAttribute erbt von UddAbstractTypeEnhanced.

UddClass erbt von UddAbstractContainer und fügt die Attribute `isAbstract` und `classtype` hinzu.

UddGeneralization erbt von UddAbstractType.

UddOperation erbt von UddAbstractTypeEnhanced und enthält außerdem die Referenzen auf seine Parameter sowie das Attribut `isAbstract`.

UddPackage erbt von UddAbstractContainer.

UddParameter erbt von UddAbstractType.

Die übrigen Klassen werden zur Erstellung der Daten sowie zur Durchführung der Differenzanalyse benötigt.

UddFactory bietet eine einfache Möglichkeit, aus XML-Elementen die entsprechenden Udd-Klassen zu instanziieren.

CldDifferenceFinder startet den Analyseprozeß. Zur Verwendung von UMLDiff$_{cld}$ sei auf den folgenden Abschnitt verwiesen.

transform.Transformer Sammelt die Transformationsoperationen und steuert ihre Ausführung.

transform.Command Abstrakte Klasse, die die Basisfunktionalität einer Transformationsoperation zur Verfügung stellt.

transform.x Konkrete Implementationen der Transformationsoperationen Add, Clone, Delete, Modify und Move. Zusätzlich wurde die Klasse Nop hinzugefügt, die keine Transformation ausführt. Sie dient zur Gliederung der Transformationsoperationen und wird beispielsweise dazu verwendet, ein Match in der Übersichtstabelle darzustellen.

Ablauf eines Analysevorgangs

Die Klasse **DifferenceFinder** im Paket **pi.seditec.umldiff** implementiert eine `main`-Methode und kann somit von der Kommandozeile aus gestartet werden. Als Parameter werden die beiden Dateinamen der Klassendiagramme erwartet. Der dritte Parameter bestimmt den Namen der Ausgabedatei. Wird als weiterer Parameter `-debug` angegeben, werden mehr Informationen über den Analyseprozeß ausgegeben.

DifferenceFinder liest die Diagramminformationen ein und übergibt sie an die Klasse **CldDifferenceFinder**. Von hier werden die 6 Stufen der Analyse gestartet. Zuerst werden die Pakete verglichen, dann die Klassen und schließlich die Inhalte der Klassen (Generalisierungen, Attribute, Assoziationen und Operationen).

Der Vergleich findet in der Klasse **UddAbstractContainer** statt, da hier die zu vergleichenden Elemente abgelegt sind. Der Ablauf folgt dem in Abbildung 4.6 auf Seite 34. Die Bewertungsfunktionen wurden den Klassen zugeordnet, in denen das zu bewertende Attribut definiert ist.

Nachdem der Vergleich auf allen Stufen durchgeführt wurde, wird ein HTML-Report und ein kombiniertes, eingefärbtes Differenzklassendiagramm erzeugt.

4.11 Beispiele

Zuerst wird nochmals das Beispiel aus Kapitel 1 aufgegriffen. Die beiden dort abgebildeten Klassendiagramme wurden in das UDD-Format konvertiert und an UMLDiff$_{cld}$ übergeben. Das Ergebnis ist in Abbildung 4.15 zu sehen.

Nach diesem kleinen Beispiel soll nun ein umfangreicheres untersucht werden. Hierzu wurde eine Beispielapplikation namens *CanDispenser* modelliert und in Java implementiert. Dieses Beispiel wird auch im folgenden Teil zur Demonstration der Sequenzdiagrammanalyse verwendet.

CanDispenser beschreibt einen Verbund von Getränkeautomaten, die mit einem Verwaltungssystem verbunden sind. Die Automaten melden sich bei dem Verwaltungssystem an, wenn sie gestartet werden. Dort kann zentral der Füllstand der Automaten abgefragt werden. An den Automaten selbst können Getränkedosen aufgefüllt und entnommen werden.

Bei der Modellierung dieser Applikation wurden alle von UMLDiff$_{cld}$ unterstützten Modellierungselemente verwendet. Das System besteht aus zwei Paketen. Das *server*-Paket kümmert sich um die Verwaltung der Automaten. Die Automaten selbst werden durch das *dispenser*-Paket modelliert.

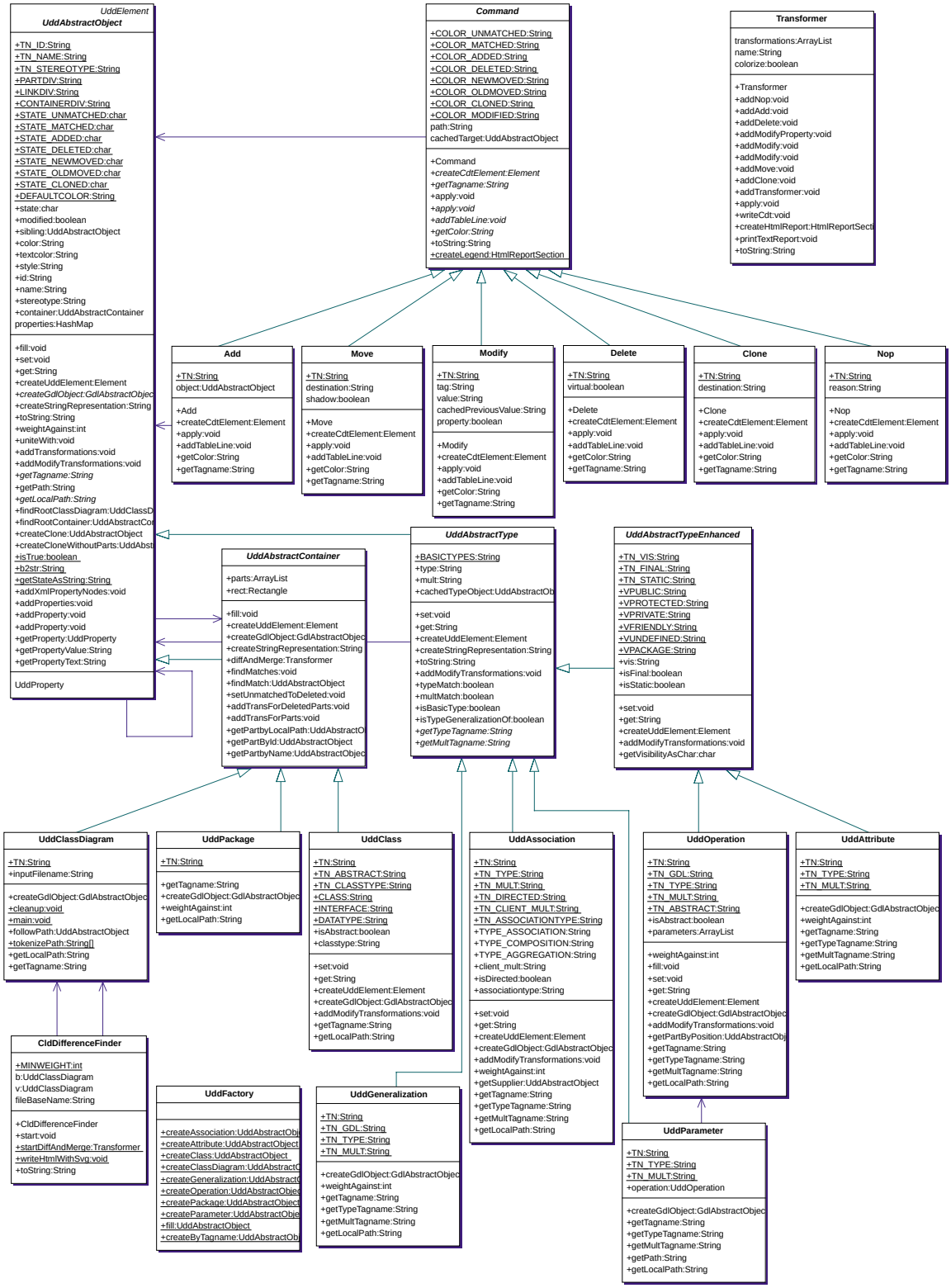


Abbildung 4.14: Das Klassendiagramm des Paketes `pi.seditec.umldiff.cld`

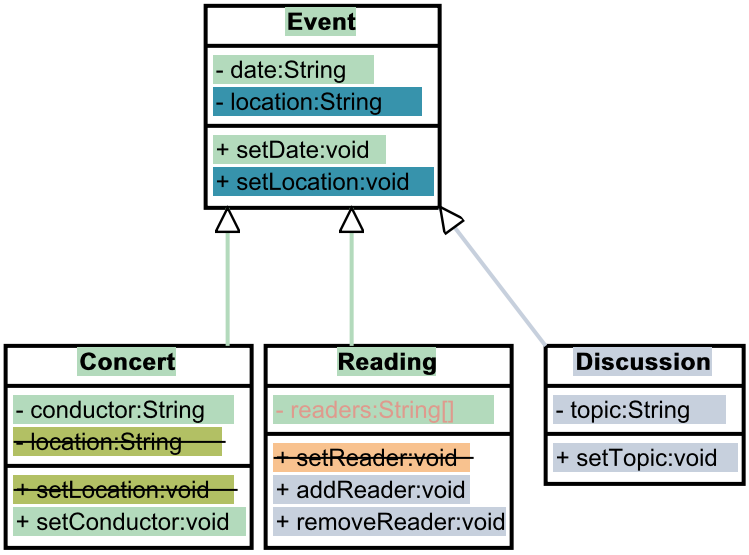


Abbildung 4.15: Ein einfaches Differenzklassendiagramm

Es wurden nun einige Änderungen an dem *dispenser*-Paket vorgenommen. Es wurde dabei versucht, übliche Anpassungen durchzuführen, wie sie auch bei der normalen Softwareentwicklung vorkommen.

In Abbildung 4.16 ist das ursprüngliche Diagramm und in Abbildung 4.17 seine Variante dargestellt.

Beispielsweise wurde die neue Klasse `Beer` hinzugefügt, die das Interface `DispenserItem` implementiert. Einige Attribute und Operationen wurden verändert, ein weiteres Attribut verschoben.

Das kombinierte und eingefärbte Diagramm ist in Abbildung 4.18 dargestellt. Wie zu sehen ist, konnte die Differenzanalyse bei diesem Beispiel nicht alle Änderungen erkennen. Die Klasse `ConsumerInterface` wurde in `DispenserInterface` umbenannt. Da keine weiteren Anhaltspunkte für diese Klasse bestanden, konnte UMLDiff$_{cld}$ diese Änderung jedoch nicht erkennen. Alle anderen Veränderungen konnten jedoch richtig erkannt werden. Beispielsweise die Verschiebung des Attributes `taste` von `DispenserItem` nach `Lemonade`.

Die hier vorgestellten Demonstrationsdateien können auf der Webseite von UMLDiff[26] heruntergeladen und betrachtet werden. Dort finden sich auch weitere Informationen zu UMLDiff.

[26] `http://www.girschick.net/martin/study/umldiff/`

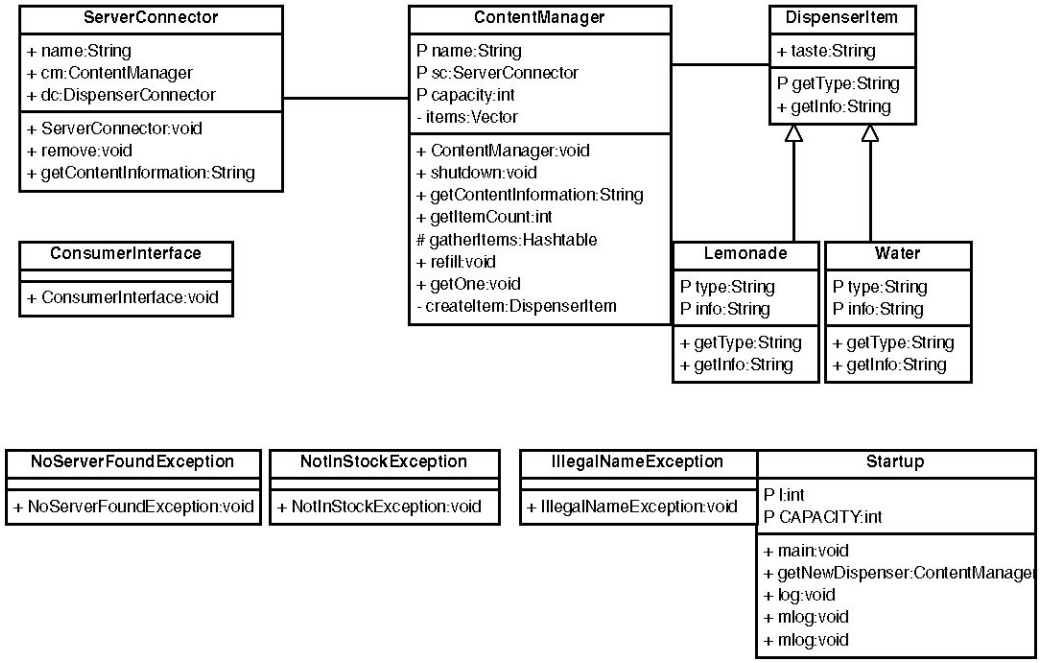


Abbildung 4.16: Originalversion des canDispenser.dispenser-Paketes

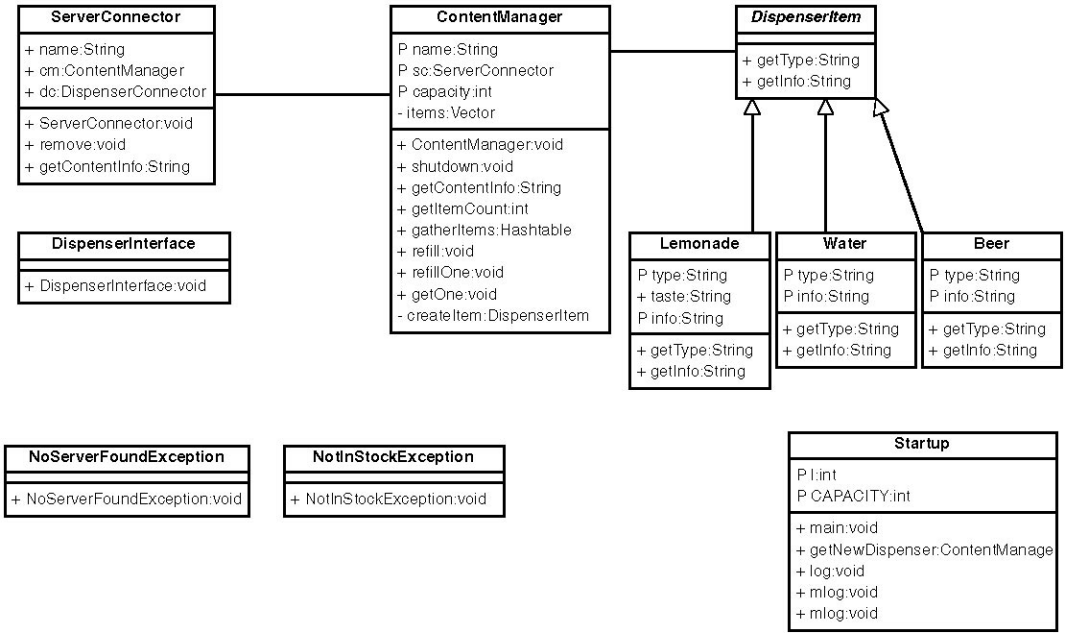


Abbildung 4.17: Variante des canDispenser.dispenser-Paketes

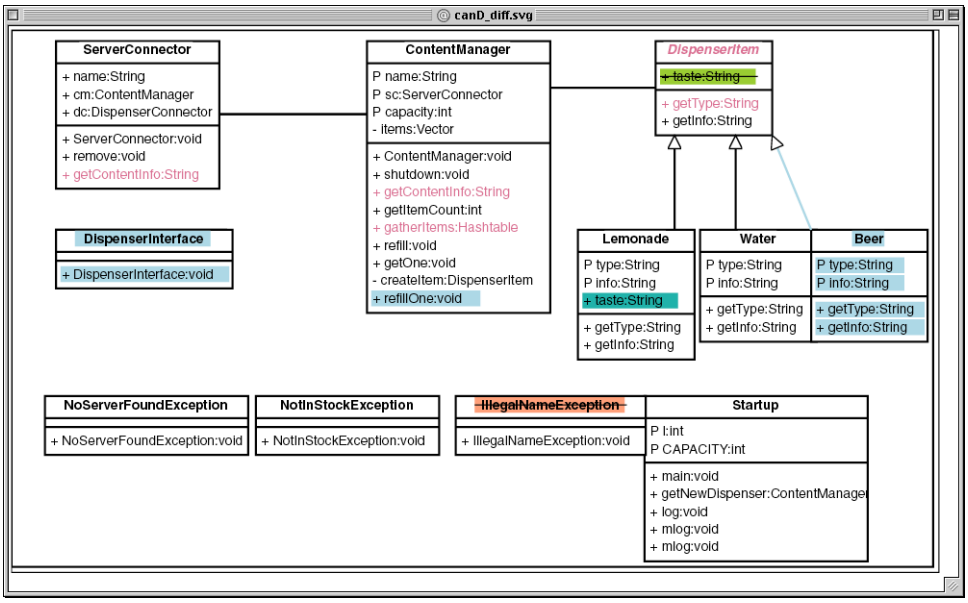


Abbildung 4.18: kombiniertes Klassendiagramm

Kapitel 5

Differenzanalyse von Sequenzdiagrammen

5.1 Motivation

In Kapitel 3 wurde ein Verfahren entwickelt, welches die Unterschiede zwischen zwei Klassendiagrammen erkennt und daraus verschiedene Informationen gewinnt. Ein ähnliches Verfahren erscheint auch für Sequenzdiagramme sinnvoll. Neben der Analyse zweier Versionen eines Sequenzdiagramms ist hier vor allem der Vergleich mit automatisch generierten Sequenzdiagrammen möglich, die beispielsweise durch Testläufe entstanden sind.

In Abschnitt 2.2 wurde bereits das Forschungsprojekt SeDiTeC der TU-Darmstadt vorgestellt. Der dortige Ansatz, Sequenzdiagramme nicht nur zur Spezifikation eines Programms, sondern auch zur Spezifikation der dafür benötigten Tests zu verwenden, vereinfacht das Testen von Software enorm. Sequenzdiagramme müssen ohnehin im Rahmen der Anforderungsanalyse und der Spezifikation erstellt werden. Versieht man diese Diagramme nun mit Testdaten (bestehend aus Testeingaben und Überprüfungswerten), so erhält man bereits eine vollständige Testspezifikation. Durch Kombination mehrerer Sequenzdiagramme lassen sich umfangreiche Testläufe erstellen. SeDiTeC stellt hierfür eine geeignete Umgebung zur Verfügung.

Vereinfacht läßt sich die *bisherige* Funktionsweise von SeDiTeC wie folgt beschreiben:

1. Zuerst werden in Together die benötigten Sequenzdiagramme erstellt. Da SeDiTeC das Zusammensetzen von Sequenzdiagrammen beherrscht, können hier mehrere, kleine Diagramme entworfen werden, die sich dann zu den benötigten Tests zusammensetzen lassen.
2. Im nächsten Schritt werden diese Diagramme mit einem Together-Modul in das SeDiTeC-XML-Format exportiert.
3. Diese XML-Datei läßt sich nun mit SeDiTeC öffnen. Hier fügt man nun den Sequenzdiagrammen einen oder mehrere Testfälle hinzu. Jeder Testfall beschreibt für die Methodenaufrufe des Sequenzdiagramms die benötigten Parameter- und Rückgabewerte.
4. In einem weiteren Dialog lassen sich mehrere Sequenzdiagramme aneinanderhängen. Es lassen sich kombinierte Testfälle zusammenstellen, indem man zu jedem Sequenzdiagramm einen dazugehörigen Testfall auswählt.

5. Man entscheidet sich nun, welche Klassen getestet werden sollen. Alle anderen Klassen werden mit Hilfe eines zweiten Together-Moduls in sogenannte Stubs konvertiert. Diese Stubs verhalten sich entsprechend der Spezifikation der Sequenzdiagramme. Dies kann auch dazu genutzt werden, noch nicht fertig gestellte Teile eines Systems zu simulieren.

6. Mit SeDiTeC lassen sich nun diese Tests ausführen. Die Stubs produzieren entsprechend der Sequenzdiagramme Methodenaufrufe und überprüfen Rückgabewerte und Aufrufparameter. Weichen die zu testenden Klassen hierbei vom spezifizierten Verhalten ab oder liefern falsche Werte zurück, so wird ein Fehlerprotokoll generiert.

Durch die automatische Generierung von Stubs ist ein sehr komfortables Testen möglich. Das System ist bereits im unvollständigen Zustand lauffähig, und es lassen sich stückweise Stubs durch „richtige" Klassen ersetzen.

Sequenzdiagramme enthalten im allgemeinen nicht alle tatsächlich notwendigen Methodenaufrufe sondern sie dienen nur als Spezifikationsmittel. Bei der eigentlichen Implementierung werden üblicherweise wesentlich mehr Methodenaufrufe benötigt, um das gewünschte Verhalten zu erzielen. Durch die Verwendung von Teststubs wird bei SeDiTeC nur die Kommunikation zwischen den Stubs und dem zu testenden Programm beobachtet. Damit läßt sich sehr genau steuern, welche Klassen getestet werden sollen. Treten Abweichungen von dem durch das Sequenzdiagramm vorgegebenen Verlauf auf, so gilt der Testlauf als nicht bestanden.

SeDiTeC nimmt nur eine Überprüfung der Rückgabewerte und Aufrufparameter vor. Die Methodenaufrufe müssen in genau derselben Reihenfolge erfolgen, wie sie durch das Sequenzdiagramm spezifiziert wurden. Es wäre nun wünschenswert, wenn neben der Überprüfung der Werte auch eine Überprüfung der Struktur durchgeführt werden würde. Diese Strukturüberprüfung kann für viele unterschiedliche Zwecke genutzt werden:

- Auch unbekannte Methodenaufrufe können analysiert werden. Der eigentliche Testlauf kann unter Umständen vollständig zu Ende geführt werden, obwohl bereits ein Fehler entdeckt wurde.
- Die tatsächliche Aufrufreihenfolge kann von der spezifizierten Reihenfolge abweichen und dennoch die richtigen Werte liefern. Hier wären die Unterschiede der beiden Aufrufreihenfolgen interessant.
- Durch Analyse der bei dem Testlauf gewonnenen Aufrufreihenfolge lassen sich unnötige oder umständliche Sequenzen erkennen.

Bisher wurde von SeDiTeC nur die Kommunikation zwischen Stubs und den zu testenden Klassen beobachtet. Weitet man diese Beobachtung auf *alle* erfolgten Methodenaufrufe aus, so ergeben sich weitere Anwendungsmöglichkeiten.

- Aufgrund der beobachteten Ablaufdaten lassen sich Sequenzdiagramme generieren.
- Verschiedene Testläufe mit unterschiedlichen Testdaten könnten verglichen werden.
- Gilt ein Test als nicht bestanden, so kann mit Hilfe der zusätzlichen Informationen eine genauere Fehlerdiagnose durchführt werden.

Es ergibt sich also ein weites Feld von interessanten und nützlichen Anwendungsbereichen, die nicht nur auf das Testen von Software beschränkt sind, sondern auch für den Entwurf und die Dokumentation hilfreich sein können.

Im folgenden wird mit einem **definierenden Sequenzdiagramm** ein spezifizierendes Sequenzdiagramm mit dazugehörigen **Testdaten** (Aufrufparameter und Rückgabewerte) bezeichnet. Das aufgezeichnete Sequenzdiagramm und dessen **Meßdaten** wird **observiertes Sequenzdiagramm** genannt.

5.2 Unterschiede zwischen Sequenzdiagrammen

Die hier betrachteten Sequenzdiagramme beinhalten nicht nur strukturelle Informationen über die Aufrufreihenfolge, sondern auch über die dabei verwendeten Parameter- und Rückgabewerte. Unterschiede zwischen solchen Diagrammen können somit nicht nur struktureller Natur sein, sondern auch rein auf Unterschieden zwischen diesen Werten beruhen.

Im folgenden werden die Unterschiede zwischen Diagrammen näher beleuchtet und in Kategorien eingeteilt. Die Erkennung dieser Unterschiede wird dann in Abschnitt 6.4 näher untersucht.

Es lassen sich vier Kategorien zur Beschreibung der Unterschiede finden. Zuerst werden rein wertbasierte Unterschiede betrachtet, wie sie bereits von SeDiTeC unterstützt wurden. Bei strukturellen Unterschieden lassen sich fehlende Aufrufe und Aufrufe in falscher Reihenfolge unterscheiden. In der letzten Kategorie sind alle sonstigen Fälle aufgeführt, die hier von Interesse sind. In Tabelle 5.1 sind diese Fälle aufgeführt und näher beschrieben. Zur Illustration wird ab Seite 54 für jeden Fall ein Sequenzdiagrammfragment des definierenden und des observierten Diagramms gegenübergestellt.

5.3 Allgemeine Betrachtungen zum Ablauf der Differenzanalyse

Um die im vorherigen Abschnitt vorgestellten Unterschiede zu erkennen, sind zwei Ansätze denkbar. Der erste Ansatz lehnt sich an die Funktionsweise von SeDiTeC an. Hier wird bereits während des Testlaufs ein Vergleich mit den gerade entstehenden Daten durchgeführt. Wie man recht schnell erkennt, ist dies mit den geforderten Merkmalen des vorherigen Abschnitts nicht vereinbar, da zur Analyse auch Informationen späterer Schritte benötigt werden.

Der zweite Ansatz ist die „Offline"-Analyse. Hier wird der Testlauf erst vollständig ausgeführt. Die dabei entstandenen Ablaufdaten können dann nachträglich mit anderen Ablaufdaten oder dem spezifizierenden Sequenzdiagramm und dessen Testdaten verglichen werden.

In Kapitel 6 wird ein Prototyp eines solchen Vergleichswerkzeugs vorgestellt, welches sich in das Programmpaket SeDiTeC integriert und damit die strukturbasierte Analyse eines Testlaufs ermöglicht.

v	**Analyse der Werte**	
*v*1	Falscher Wert	Die Meßdaten weichen von den Testdaten ab.
*v*2	Falsche Instanz als Wert	Ein Aufrufdatum ist vom richtigen Typ, jedoch der falschen Instanz.
*v*3	Falsche Signatur	Oft werden mehrere Methoden mit gleichem Namen aber unterschiedlicher Signatur erstellt. Diese haben meist das selbe Verhalten, nehmen hierzu aber unterschiedliche Parametertypen.
a	**Übersprungene oder fehlende Aufrufe**	
*a*1	Fehlende Aufrufe	In dem aufgezeichneten Diagramm fehlen Aufrufe.
*a*2	Beobachtung zusätzlicher Aufrufe	Eine Instanz kommt in beiden Diagrammen vor, es werden jedoch zusätzliche Aufrufe auf dieser beobachtet, die nicht in S_d vorkommen. Dies kann folgende Gründe haben: • Die Spezifikation ist unzureichend. • Die Implementierung ist zu umständlich.
*a*3	Beobachtung zusätzlicher Instanzen	Meist in Verbindung mit falschen Aufrufdaten (siehe *v*2). Aufrufe (die durchaus in S_d vorkommen können) erfolgen auf falschen Instanzen.
*a*4	Beobachtung unbekannter Klassen	In dem observierten Diagramm sind Instanzen von Klassen (und Aufrufe auf diesen) vorhanden, die im definierenden Diagramm nicht enthalten sind.
s	**Reihenfolge der Aufrufe**	
*s*1	Vertauschung innerhalb einer Instanz	Von einer Instanz gehen mehrere Aufrufe aus. Deren Reihenfolge ist in den beiden Diagrammen unterschiedlich.
*s*2	Übersprungene Instanz	Ein Aufruf fehlt im aufgezeichneten Diagramm. Der folgende Aufruf hat jedoch dieselbe Quelle wie der fehlende Aufruf. Dies ist so zu verstehen, daß eine dazwischenliegende Instanz nicht einbezogen wurde, obwohl die Spezifikation dies vorsieht.
*s*3	unbekannter Mittelsmann	Dies ist das gegenteilige Fall zu *s*2. Er wird jedoch bereits durch *a*2 abgedeckt.
m	**Sonstiges**	
*m*1	Rekursion und Iteration	Erkennung sollte ebenfalls möglich sein, kann in weiterer Analysephase auswertet werden.
*m*2	Exceptions	Der Testlauf wurde mit einem Laufzeitfehler oder einer Exception abgebrochen. Dies sollte ebenfalls erkannt und analysiert werden.

Tabelle 5.1: Kategorisierung und Beschreibung möglicher Differenzen in Sequenzdiagrammen

Abbildung 5.1: Fall *v*1: Falscher Wert

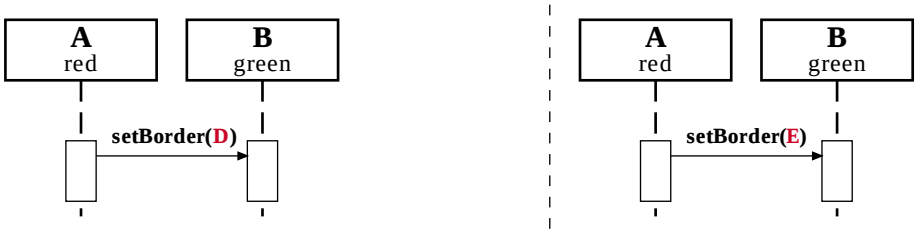


Abbildung 5.2: Fall *v*2: Falsche Instanz als Wert

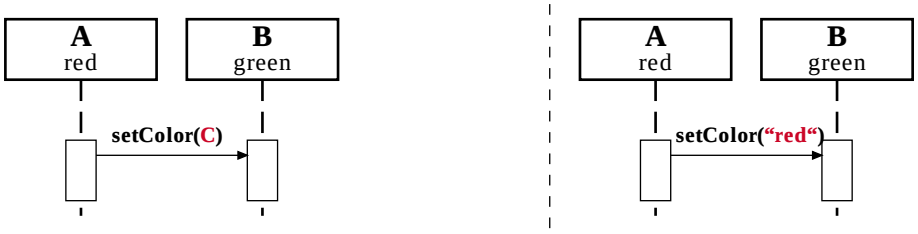


Abbildung 5.3: Fall *v*3: Falsche Signatur

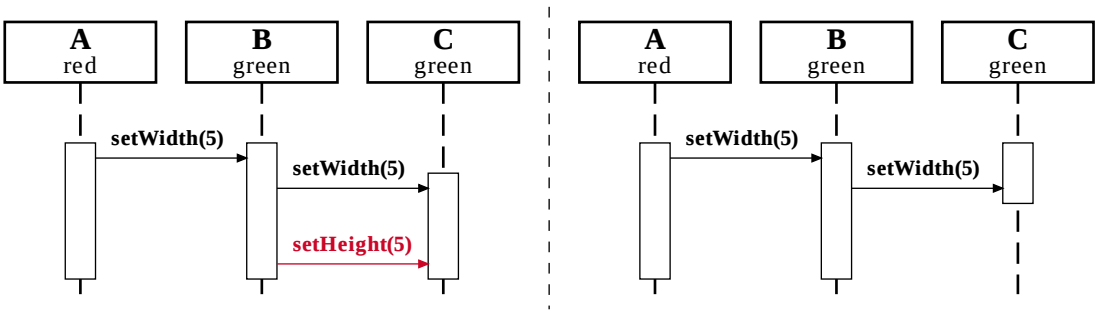


Abbildung 5.4: Fall *a*1: Fehlende Aufrufe

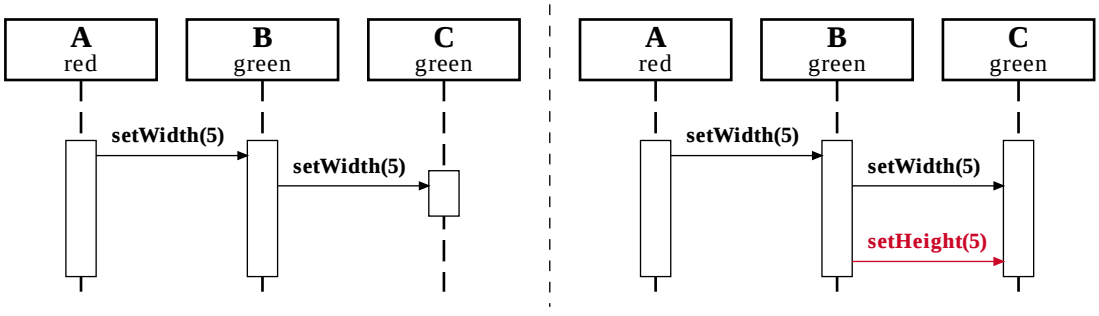


Abbildung 5.5: Fall *a*2: Beobachtung zusätzlicher Aufrufe

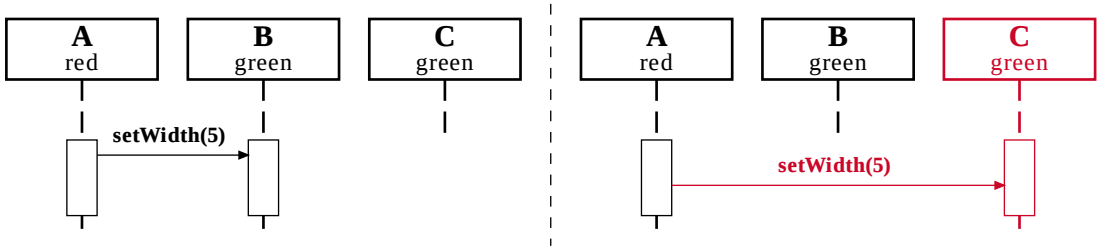


Abbildung 5.6: Fall *a*3: Beobachtung zusätzlicher Instanzen

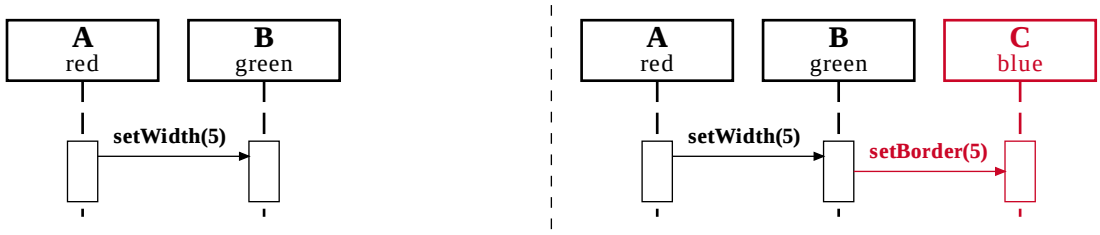


Abbildung 5.7: Fall *a*4: Beobachtung unbekannter Klassen

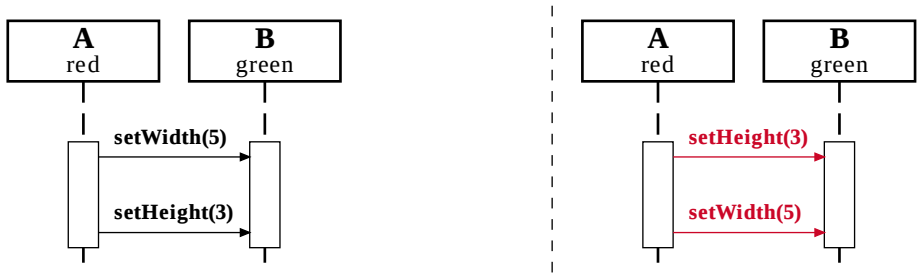


Abbildung 5.8: Fall *s*1: Vertauschung innerhalb einer Instanz

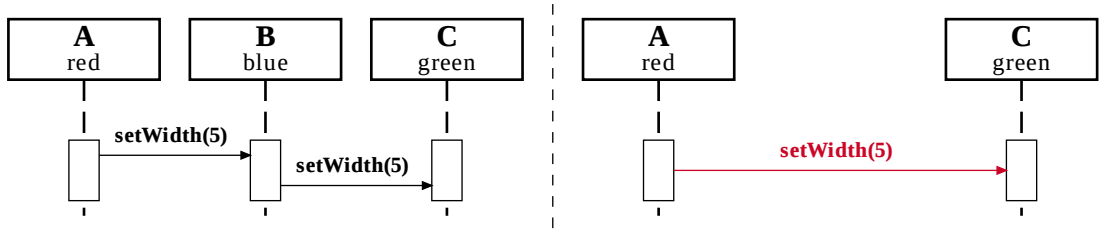


Abbildung 5.9: Fall *s*2: Übersprungene Instanz

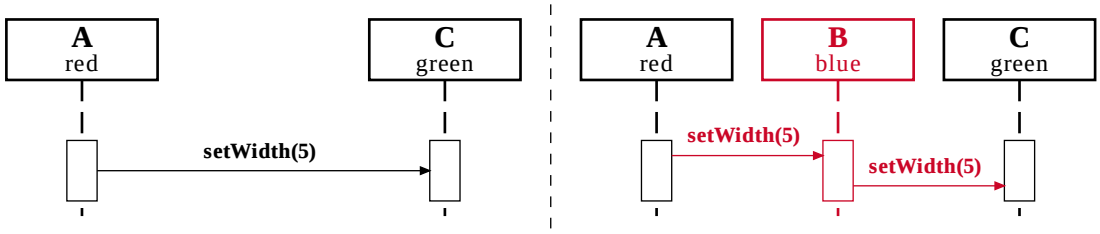


Abbildung 5.10: Fall *s*3: unbekannter Mittelsmann

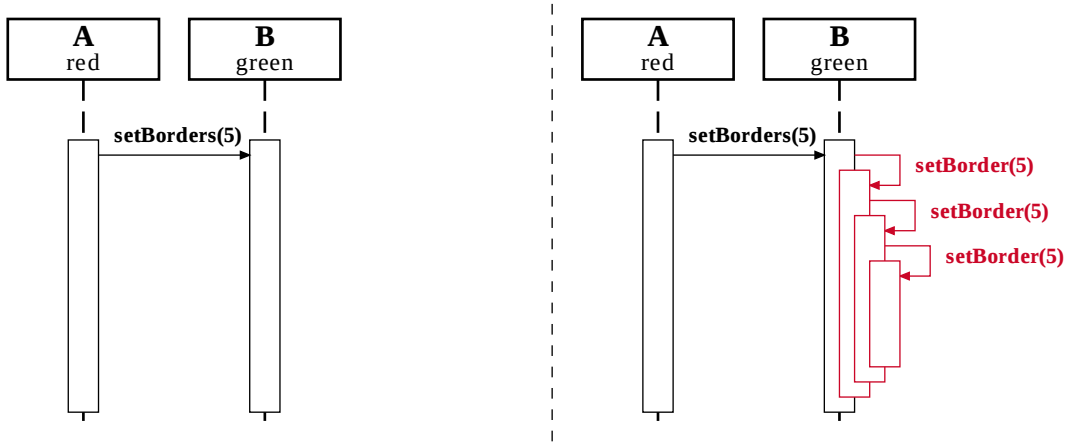


Abbildung 5.11: Fall *m*1: Rekursion und Iteration

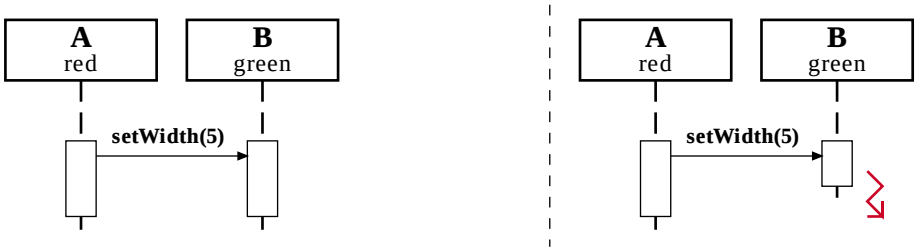


Abbildung 5.12: Fall *m*2: Exceptions

5.4 Darstellung der Ergebnisse

Nach der Analyse der Ablaufdaten sollten die Ergebnisse in übersichtliche Darstellung gebracht werden.

- Tabelle der Instanzen, die in beiden oder nur einem der Diagramme vorkommen.
- Tabelle der Methodenaufrufe des observierten Diagramms, ergänzt um die fehlenden Aufrufe des definierenden Sequenzdiagramms. Farbige Markierungen machen die Unterschiede deutlich.
- Kombiniertes Sequenzdiagramm mit Aufrufen aus beiden Ausgangsdiagrammen.

Kapitel 6

Prototyp zur Differenzanalyse von Sequenzdiagrammen

Das im vorhergehenden Kapitel entworfene Verfahren zum Testen von Software durch den Vergleich von Sequenzdiagrammen soll nun in einer prototypischen Implementierung umgesetzt werden. Analog zu Kapitel 4 soll das hier entwickelte Programm UMLDiff$_{sqd}$ genannt werden.

6.1 Entwurf eines Austauschformats für Sequenzdiagramme

Betrachtet man die definierenden und observierten Sequenzdiagramme, so fällt auf, daß diese prinzipiell dieselbe Datenstruktur verwenden. Sequenzdiagramme können sowohl aus Testläufen als auch aus der Testspezifikation stammen. Da auch der Vergleich zwischen verschiedenen Testläufen ermöglicht werden soll, erscheint es sinnvoll, für die beiden Eingaben dasselbe Datenformat zu verwenden.

Die Repräsentation der Diagramme soll hier wieder in XML erfolgen. Wie bereits bei Klassendiagrammen sind im Rahmen anderer verschiedener Forschungsvorhaben Formate zur Darstellung von Sequenzdiagrammen entwickelt worden. Allerdings beinhalten diese im allgemeinen keine Informationen aus den Testläufen. SeDiTeC stellt eine geeignete Datenstruktur zur Verfügung, die sowohl Informationen über die Methoden und das spezifizierende Sequenzdiagramm als auch Testdaten, Meßdaten und Meßdiagramme beinhaltet. Zur Bestimmung der Differenzen wird allerdings ein Format benötigt, welches unabhängig von Spezifikation und Testlauf ist. Ausgehend von der von SeDiTeC vorgegeben Datenstruktur wird hier ein Format entwickelt, welches diese Eigenschaft besitzt.

Betrachten wir das XML-Format, welches von SeDiTeC nach einem Testlauf erzeugt wird. Dieses Format besteht aus drei Teilen:

1. Am Anfang steht die **MethodDeclarationCollection**. Hier werden alle Operationen definiert. Dies beinhaltet einen eindeutigen Bezeichner, die deklarierende Klasse und eine Liste der Parametertypen sowie der Typ des Rückgabewerts.
2. Es folgt die **SequenceDiagramCollection**. Sie enthält alle Sequenzdiagramme, die für ein

Testprojekt angelegt wurden. Ein Sequenzdiagramm besteht aus einer Liste von Methodenaufrufen und einer Sammlung von Testfällen.

- Die Methodenaufrufe bestehen aus der Sequenznummer, einer Referenz auf die Methodendeklaration, der Sequenznummer der Methode von der sie aufgerufen wurden und auf welcher Instanz sie aufgerufen wurden.
- Ein Testfall spezifiziert für alle Methodenaufrufe die Parameter- und Rückgabewerte. Konstruktoren liefern als Rückgabewerte einen Bezeichner, der für die erzeugte Instanz verwendet wird. Damit läßt sich diese Instanz im späteren Verlauf des Testfalls wiederverwenden. Dies ist insbesondere notwendig, um Aufrufe auf dieser Instanz zu beschreiben.

3. An dritter Stelle steht die **CombinedSequenceDiagramCollection**. Sie beschreibt zuerst, welche Sequenzdiagramme aus der SequenceDiagramCollection hintereinander ausgeführt werden sollen. Nun folgen die Informationen des eigentlichen Testlaufs. Hierzu wird für jedes Sequenzdiagramm ein Testfall ausgewählt. Danach folgen die mitprotokollierten Methodenaufrufe des Testlaufs. Sie haben denselben Aufbau, wie die SequenceDiagramCollection.

Wie man sieht, sind die beiden zu vergleichenden Sequenzdiagramme in zwei unterschiedliche Darstellungen eingebettet. Außerdem wurden die Testdaten von den Methodenaufrufen getrennt.

Folgenden Schritte müssen nun unternommen werden, um dieses Format zu homogenisieren:

1. Die getrennten Sequenzdiagramme aus der Spezifikation werden entsprechend der Informationen aus der CombinedSequenceDiagramCollection zusammengesetzt. Dabei müssen die Sequenznummern angepaßt werden.
2. Die Testdaten der SequenceDiagramCollection können nun in die Liste der Methodenaufrufe integriert werden. Da bei dem Vergleich der Sequenzdiagramme immer nur ein Testfall betrachtet wird, können alle weiteren außer acht gelassen werden.
3. Die Methodenaufrufe des Testlaufs müssen nicht zusammengesetzt werden, da nur ein Aufrufprotokoll pro Testlauf erstellt wird.
4. Auch die Meßdaten werden in das Aufrufprotokoll integriert.

Aus diesen Betrachtungen ergibt sich eine einfachere, homogene Datenstruktur für Test-Sequenzdiagramme.

Das in Abschnitt 4.1 entwickelte Format für UML-Klassendiagramme wurde bereits so entworfen, daß weitere Diagrammtypen hinzugefügt werden können. Das dortige Wurzelelement `umldiagrams` kann nun als Unterelement auch ein `sequencediagram` enthalten.

Die Abbildung 6.1 zeigt das XML-Schema. Einige Besonderheiten sind zu erwähnen.

- `sequence` bezeichnet die Sequenznummer. Wird hier `actor` eingetragen, sind Aufrufe des Benutzers (Actor) gemeint.
- `fromsequence` verweist auf die Sequenznummer, von der dieser Aufruf erfolgt.

- `toinstance` gibt den Bezeichner der aufzurufenden Instanz an. Bei Konstruktoren oder statischen Aufrufen ist diese Angabe nicht notwendig.
- Falls der `name` eines `parameter`s die Zeichenkette `return` enthält, so ist der Rückgabewert gemeint.

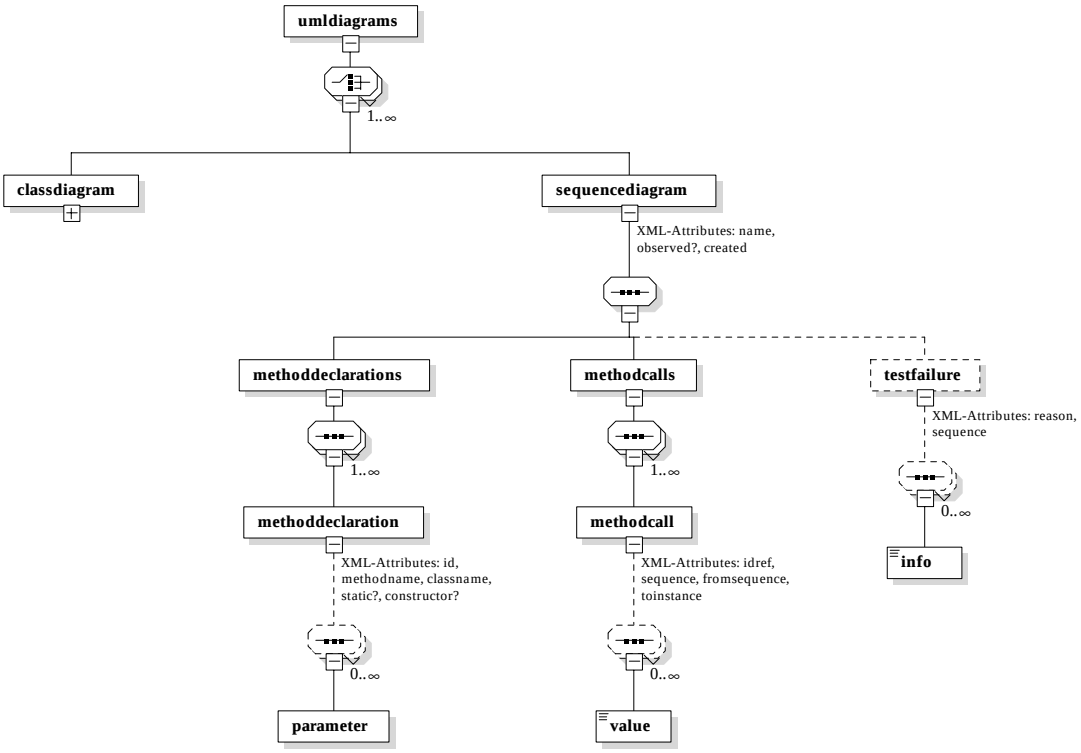


Abbildung 6.1: XML-Schema zur Beschreibung von UML-Sequenzdiagrammen

6.2 Erzeugung der Eingabedaten

Mit Hilfe von Together werden die spezifizierenden Sequenzdiagramme erzeugt. Nach dem XML-Export fügt man mit SeDiTeC die Testdaten hinzu. Bei dem Testlauf wird das Meßdiagramm generiert. Als Ausgabedatei entsteht ein XML-Dokument im SeDiTeC-Format.

Um zu dem Meßdiagramm zu kommen, mußte eine Möglichkeit geschaffen werden, Java-Programme bei ihrer Ausführung zu beobachten. Nach Untersuchungen des Debugger-Interfaces von Java stellte sich heraus, daß dieses nicht alle benötigten Informationen zur Verfügung stellt. Außerdem war mit starken Performanzeinbußen im Testlauf zu rechnen.

Stattdessen wurde ein weiteres Modul für Together entwickelt, was den bereits vorhandenen Sourcecode instrumentiert. In jeder Operation wird am Anfang ein Aufruf an SeDiTeC eingefügt. Dieser enthält alle Aufrufparameter dieser Operation. Außerdem wird eine Anweisung vor jedem `return` eingefügt, um den Rückgabewert an SeDiTeC zu melden. Bei einem Testlauf werden nun alle Methodenaufrufe von SeDiTeC mitprotokolliert. In Abbildung 6.2 wird eine solche instrumentierte Methode dargestellt.

```
public String getInfo()
{
 // initialization
 String zUniqueMethodName =
  "<oiref:java#Member#pi.canDispenser.dispenser.Lemonade#getInfo#(##)#:oiref>";
 TestCenter tc = TestCenter.getTestCenter();
 // check call order
 MethodCallDataWrapper mcdWrapper =
  tc.acceptInstrumentedCall(this, zUniqueMethodName, null);
 Object objResult = info;
 tc.acceptInstrumentedReturn(this, zUniqueMethodName, mcdWrapper, objResult);
 return (java.lang.String) objResult;
}
```

Abbildung 6.2: Beispiel einer instrumentierten Methode

Die XML-Ausgabedatei des Testlaufs kann nun an das Differenzanalysewerkzeug weitergeleitet werden. Während des Einlesevorgangs wird die Datenstruktur in das oben beschriebene Format konvertiert.

6.3 Datenstrukturen

Um Sequenzdiagramme effizient miteinander vergleichen zu können, wird eine geeignete Datenstruktur benötigt. Diese sollte folgende Abfragen ermöglichen:

- Von welcher Instanz erfolgt der Aufruf?
- Was ist der nächste Aufruf?
- Wann kehrt ein Aufruf zurück?
- Welche Aufrufe erfolgen auf einer Instanz?
- Welche Aufrufe werden von einer Instanz erzeugt?
- Welche Aufrufe sind noch aktiv?

Um, diese Abfragen durchführen zu können wurde die Udd-Datenstruktur erweitert. In Abbildung 6.3 ist diese Datenstruktur durch das Klassendiagramm von UMLDiff$_{sqd}$ dargestellt.

Die Klasse **UddSequenceDiagram** speichert die Methodendeklarationen, die durch die Klassen **UddMethodDeclaration** und **UddParameterDeclaration** modelliert werden. Außerdem sind hier Referenzen auf alle Instanzen und alle Methodenaufrufe abgelegt.

UddInstance repräsentiert eine Instanz einer Klasse. Hier ist ebenfalls eine Liste von Methodenaufrufen enthalten. In dieser Liste sind alle Aufrufe abgelegt, die auf dieser Instanz erfolgen oder von ihr ausgeführt werden. Außerdem wird jeder Aufruf an zwei Stellen in dieser Liste eingetragen. Zuerst wenn der Aufruf erfolgt und ein weiteres Mal, wenn der Aufruf vollständig ausgeführt wurde und der Kontrollfokus zurück an die aufrufende Instanz geht.

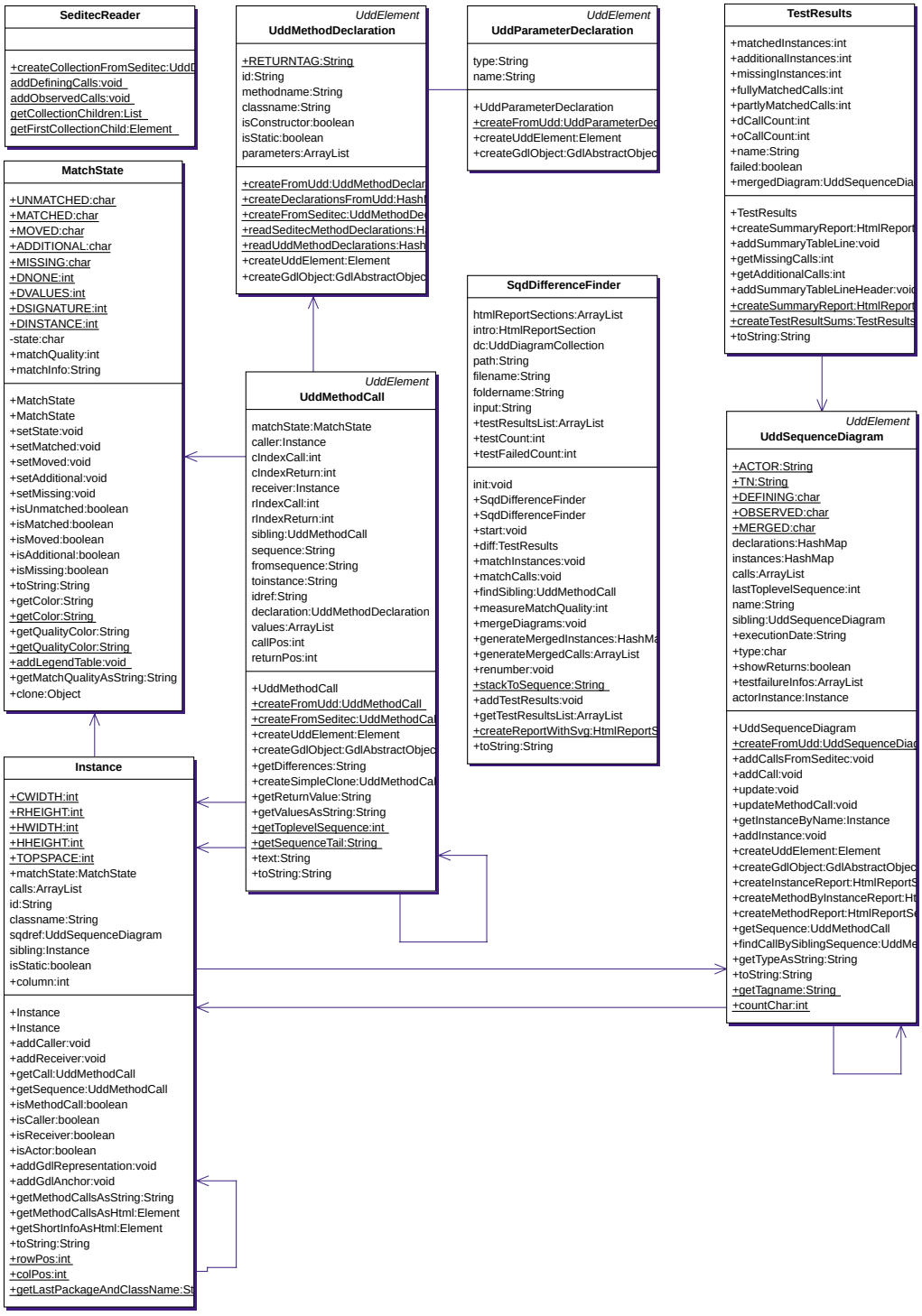


Abbildung 6.3: Klassendiagramm des Paketes `pi.seditec.umldiff.sqd`

Die Kennzeichnung dieser Aufrufe erfolgt in der **UddMethodCall**-Klasse. Diese enthält jeweils einen Verweis auf die Sender- und Empfänger-Instanz. Sowohl für Sender als auch Empfänger werden zwei Indexwerte verwaltet.

cIndexCall Gibt den Index in der Liste der Methodenaufrufe der Senderinstanz an, die den Aufruf dieser Methode darstellt.

cIndexReturn Index der Rückkehr dieses Aufrufs an die Senderinstanz.

rIndexCall Index in der Methodenaufrufliste des Empfängers. Aufrufzeitpunkt.

rIndexReturn Index in der Methodenaufrufliste des Empfängers. Rückkehr.

Diese Struktur soll anhand eines kleinen Beispiels näher erläutert werden. In Abbildung 6.4 ist ein Sequenzdiagrammfragment abgebildet. Die dafür notwendige Datenstruktur ist in Tabelle 6.1 aufgeführt. Wie man sieht, verweist Instanz A zweimal auf Aufruf C. Dieser hat einen Rückverweis an die Instanz A und durch die beiden Indexwerte findet er sich in der Aufrufliste von A wieder.

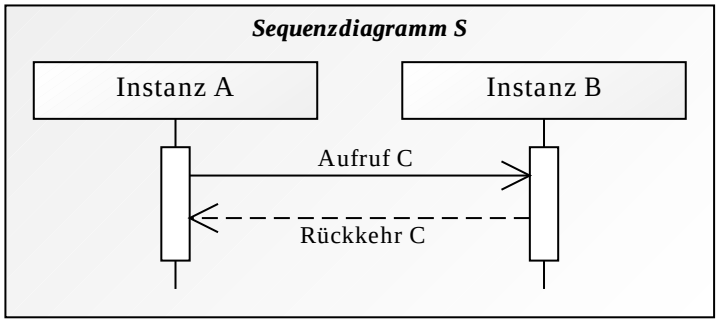


Abbildung 6.4: Ein einfaches Sequenzdiagramm

Sequenzdiagramm S		**Instanz A**		**Instanz B**		**Aufruf C**	
instances(0)	Instanz A	calls(0)	Aufruf C	calls(0)	Aufruf C	caller	Instanz A
instances(1)	Instanz B	calls(1)	Aufruf C	calls(1)	Aufruf C	cIndexCall	0
calls(0)	Aufruf C	sqdref	S	sqdref	S	cIndexReturn	1
						receiver	Instanz B
						rIndexCall	0
						rIndexReturn	1

Tabelle 6.1: Datenstruktur des Sequenzdiagramms aus Abbildung 6.4

Um nun die Aufrufsequenz eines Diagramms zu untersuchen, beginnt man bei der ersten Instanz (Actor), die überlicherweise ein virtuelles Objekt ist, welches den Actor repräsentiert. In dieser Instanz betrachtet man die Liste der Methodenaufrufe. Der erste Eintrag verweist auf ein `UddMethodCall`-Objekt, welches als Empfänger die Instanz angibt, die den Kontrollfokus erhält. In dem `UddMethodCall`-Objekt ist außerdem vermerkt, an welcher Stelle der Aufrufliste dieser Instanz der Aufruf erfolgt ist. Betrachtet man nun den folgenden Eintrag dieser Liste, so trifft man auf den nächsten auszuführenden Aufruf.

6.4 Der Vergleichsalgorithmus

Die in Abschnitt 4.3 angesprochenen Algorithmen zur Differenzanalyse in XML-Dokumenten können auch hier nicht angewendet werden, da wiederum Kenntnisse über die Semantik nötig sind, um die gewünschten Unterschiede erkennen zu können.

Der hier beschriebene Algorithmus läßt sich in fünf Phasen einteilen:

1. Einlesen der Diagrammdaten
2. Analyse und Vergleich der Instanzen
3. Vergleich der Ablaufdaten
4. Postanalyse und Generierung der Ergebnisse
5. Erzeugen der Ausgabedateien

Im folgenden bezeichnet S_d das definierende Sequenzdiagramm und S_o das observierte Diagramm. Ein Überblick über die Funktionsweise von UMLDiff$_{sqd}$ läßt sich aus Abbildung 6.5 gewinnen.

In der Analysephase werden zuerst die Instanzen verglichen, die in den beiden Diagrammen vorkommen. Werden dabei gleiche Instanzen entdeckt, wird die Datenstruktur entsprechend aktualisiert. Man kann davon ausgehen, daß ein Großteil der Methodenaufrufe in beiden Diagrammen übereinstimmen. Daraus ergibt sich, daß auch die verwendeten Instanzen in beiden Diagrammen übereinstimmen.

In SeDiTeC erhält jedes Objekt, welches instanziiert wird, einen eindeutigen Bezeichner. Bei der Durchführung des Testlaufs wird dieses Objekt mit seinem Bezeichner in einer Tabelle abgelegt und kann so während des Testlaufs wiedergefunden werden. Diese Information fließt dann in das observierte Diagramm ein.

Alle ungematchten Objekte in S_d und S_o müssen nun noch näher betrachtet werden. Treten in S_d Klassen beziehungsweise Instanzen dieser Klassen auf, die nicht in S_o enthalten sind, so werden die Aufrufe auf diesen Instanzen ebenfalls in S_o fehlen. Dies trifft nicht notwendigerweise auf Aufrufe zu, die von diesen Instanzen aus erfolgen. In Abbildung 5.9 wird dies durch zwei Sequenzdiagramme illustriert. Dieses Verhalten wurde bereits in Abschnitt 5.1 angesprochen und dort als *übersprungene Instanz (s2)* bezeichnet.

Nun beginnt der eigentliche Vergleich der Ablaufdaten. Da das primäre Ziel die Überprüfung von S_o anhand von S_d darstellt, werden nun die Abläufe des definierenden Sequenzdiagramms betrachtet und ein entsprechendes Matching in dem observierten Diagramm gesucht.

Folgende Definitionen werden benötigt. Treten diese mit den Indizes *d* oder *o* auf, so ist das jeweilige Element in S_d respektive S_o angesprochen:

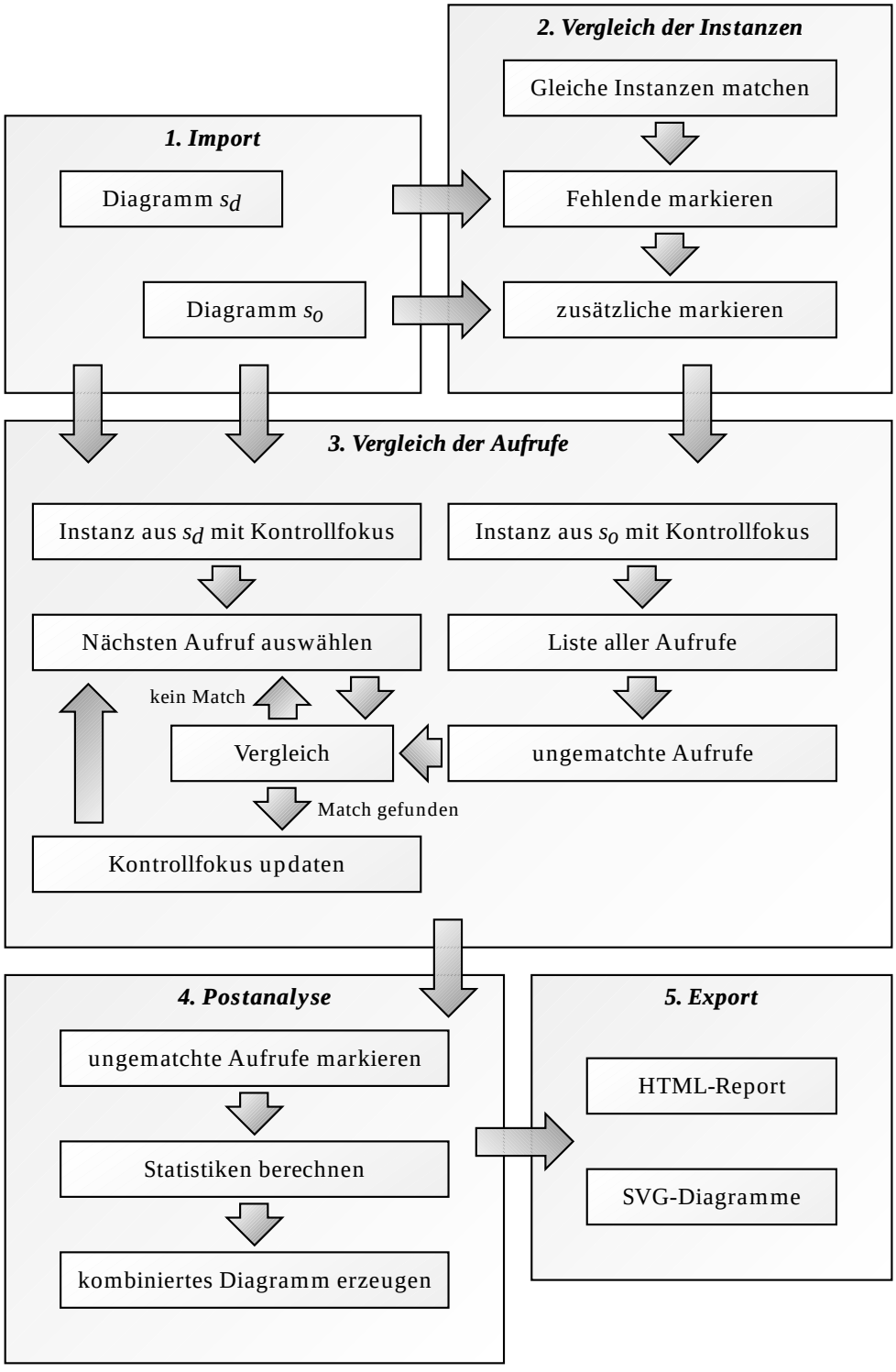


Abbildung 6.5: Überblick über die Funktionsweise der Sequenzdiagrammanalyse

a	Aufruf
$Q(a)$	Senderinstanz des Aufrufs
$Z(a)$	Empfängerinstanz des Aufrufs
$W(a)$	Werte des Aufrufs (beinhaltet Parameterwerte und Rückgabewert)
i	Instanz einer Klasse
$L(i)$	Liste der Methodenaufrufe in Instanz i[1]

Am Anfang wird der Kontrollfokus bei beiden Diagrammen auf den Aktor gelegt, der durch eine „virtuelle" Instanz repräsentiert wird. Es wird nun ausgehend von S_d die Aufrufkette verfolgt.

Der Vergleich eines Aufrufs läuft nach folgendem Schema ab:

1. Betrachte den nächsten Aufruf a_d in der aktuellen Kontrollfokusinstanz aus S_d
2. Betrachte die Methodenaufrufe $L(i_o)$ der Kontrollfokusinstanz i_o aus S_o.
3. Suche in $L(i_o)$ den ähnlichsten Aufruf zu a_d.
4. Existiert dieser, aktualisiere die Kontrollfoki und ermittle die Unterschiede zwischen den beiden gematchten Aufrufen.
5. Wird kein passender Aufruf entdeckt, dann markiere a_d als fehlend.
6. Betrachte nun den nächsten Aufruf in der aktuellen Kontrollfokusinstanz von S_d.
7. Trifft man hierbei auf einen Aufruf, dessen rIndexReturn dem Index des gerade betrachteten Aufrufs entspricht, so wurde ein „Return" gefunden und man kehrt zur vorherigen Kontrollfokusinstanz zurück.

Wurden nun alle Aufrufe in S_d untersucht, schließt sich die **Postanalyse** an. Hier werden nun die Unterschiede zwischen den gematchten Elementen untersucht und nicht gematchte Elemente markiert. Hierfür wurde bereits in Tabelle 5.1 eine Kategorisierung vorgenommen. Die Postanalyse läuft in folgenden Schritten ab.

1. Zuerst werden alle gematchten Aufrufe betrachtet:
   (a) Sind Werte unterschiedlich? ($\rightarrow v1$ und $v2$)
   (b) Sind Signaturen unterschiedlich? ($\rightarrow v3$)
   (c) Sind Zielinstanzen unterschiedlich? ($\rightarrow a3$)
2. In beiden Diagrammen können Klassen und Instanzen enthalten sein, die in dem anderen Diagramm nicht vorkommen. Dies wurde bereits am Anfang der Analysephase untersucht, muß jedoch nun erneut analysiert werden:
   (a) Instanzen aus S_o, die in S_d nicht vorkommen, können dennoch Auslöser von nun gematchten Aufrufen sein. Sie fallen somit in die Kategorie $s2$.
   (b) Instanzen aus S_o können sowohl Quelle als auch Ziel von nun gematchten Aufrufen sein. Dies entspricht $s2$ und $a3$.

[1] beinhaltet alle Aufrufe und „Returns", die Quelle oder Ziel in i haben.

   (c) Alle übrigen Instanzen aus S_o, die keine gematchten Aufrufe besitzen, gelten als unbekannte Instanzen und fallen in Kategorie $a4$.

3. Alle ungematchten Aufrufe aus S_d sind $a1$ zuzuordnen.

4. Nun muß noch die Reihenfolge untersucht werden:

   (a) Hierzu werden alle Instanzen aus S_o betrachtet.

   (b) Die dortigen Aufruflisten $L(i_o)$ werden mit den Listen aus S_d verglichen. Vertauschungen können so erkannt werden.

   (c) Sind in der Liste Aufrufe mehrfach enthalten, kann Fall $m1$ vorliegen.

   (d) Aufrufe, denen keine Nummern zugewiesen wurde, gehören zu Fall $a2$.

5. Fall $m2$ kann in S_o vorkommen, wenn bei dem Testlauf ein Fehler aufgetreten ist. Er muß entsprechend verarbeitet werden.

6.5 Darstellung der Ergebnisse

Wünschenswert wäre ein Format, welches alle oben angesprochenen Fälle vollständig beschreibt. Damit wäre es möglich, beliebige Visualisierungen der gewonnenen Erkenntnisse durchzuführen. Da die Definition eines derart umfangreichen Formats den Rahmen dieser Arbeit sprengen würde, wurde darauf verzichtet. Anstatt dessen werden zwei mögliche Repräsentationen der Ergebnisse angesprochen, die in dieser Arbeit umgesetzt wurden. Beispiele hierzu finden sich im folgenden Abschnitt.

- In Form eines Reports werden die in der Postanalyse erkannten Differenzen als HTML-Datei abgespeichert.

- Das Sequenzdiagramm S_d wird als Basis für eine graphische Darstellung verwendet. Fehlende Klassen, Instanzen und Aufrufe aus S_o werden hinzugefügt. Die Unterschiede werden farblich gekennzeichnet. Als Format wird das in Abschnitt 4.9 vorgestellte GDL verwendet, was durch eine Konvertierung in das standardisierte SVG-Format ebenfalls mit einem Webbrowser dargestellt werden kann.

6.6 Beispiel

Die in 4.11 vorgestellte Beispielapplikation *CanDispenser* soll hier erneut zur Demonstration verwendet werden.

Um einen Eindruck über die einfache Anwendbarkeit von SeDiTeC und UMLDiff$_{sqd}$ zu erhalten, werden nun kurz die einzelnen Schritte zur Durchführung eines Testlaufs illustriert. In Abbildung 6.6 wird die Erstellung der Sequenzdiagramme dargestellt. Für *CanDispenser* wurden vier Sequenzdiagramme erzeugt.

1. Start des Servers.

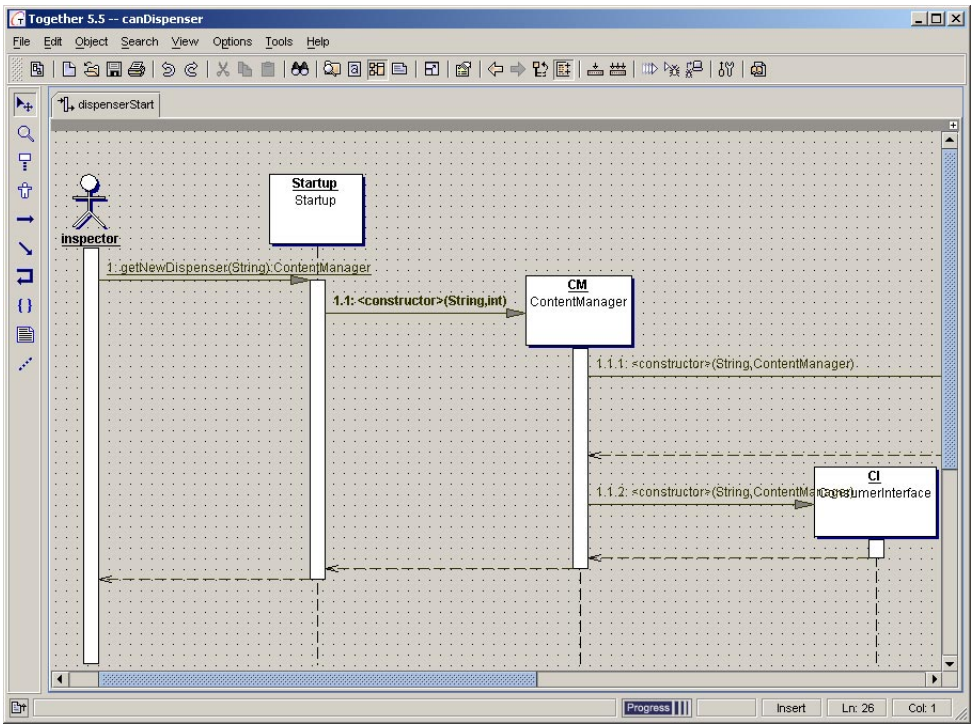


Abbildung 6.6: Erstellung der Sequenzdiagramme in Together

2. Start eines Dispensers.
3. Nachfüllen von Getränken mit Füllstandskontrolle.
4. Entnahme eines Getränkes.

Diese Diagramme können nun mit SeDiTeC beliebig kombiniert und mit Testdaten versehen werden. Dies geschieht in Abbildung 6.7. Zwei Tests sollen durchgeführt werden.

1. Test 1 startet den Server und einen Dispenser.
2. Test 2 startet ebenfalls Server und Dispenser, füllt zwei Dosen Wasser nach und versucht dann eine Dose Limonade zu entnehmen.

Der Testlauf wird von SeDiTeC gestartet, es werden beide Tests durchführt und eine Ergebnisdatei im XML-Format erstellt. Diese wird von UMLDiff$_{sqd}$ eingelesen und analysiert.

Als Ausgabe erhält man einen HTML-Report, in dem das kombinierte Sequenzdiagramm integriert ist. In Abbildung 6.8 bis 6.10 sind Teile des Reports[2] dargestellt.

Am Anfang ist eine Legende der verwendeten Farben aufgeführt. Grün steht für Aufrufe, die in beiden Diagrammen vorkommen. Wenn diese Aufrufe sich nicht vollständig gleichen, wird in einer zweiten Textzeile unter dem Aufruf angegeben, worin die Unterschiede bestanden. Hier

[2]Unter `http://www.girschick.net/martin/study/umldiff/` kann der gesamte Report eingesehen werden.

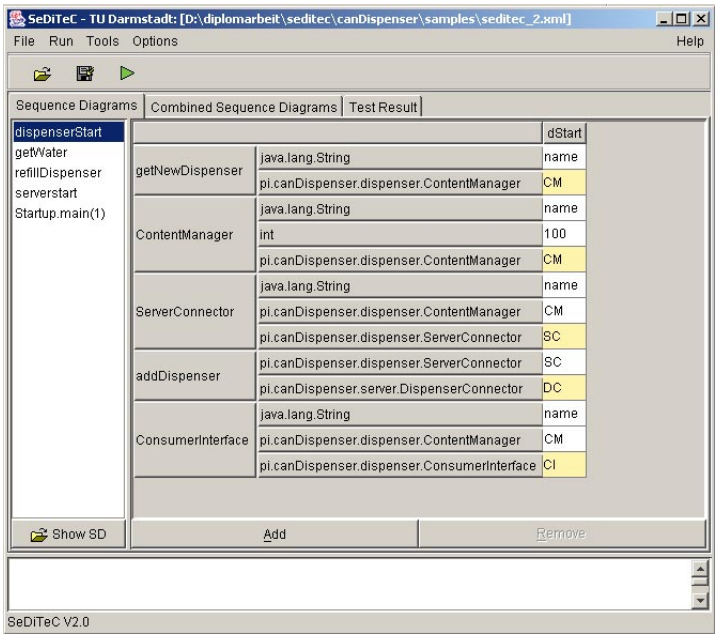


Abbildung 6.7: Hinzufügen der Testdaten mit SeDiTeC

werden die gelben Farbtöne als Markierung verwendet. Blau markiert sind alle Aufrufe, die im definierenden Diagramm nicht vorkamen, rot sind fehlende Aufrufe.

Nach der Legende kommt ein Überblick über die durchgeführten Tests. Dann folgen die Berichte der beiden Tests.

Wie man im Diagramm zu Test 1 sieht wurde die Initialisierung einer Singleton-Klasse vergessen. Deshalb kommt es im späteren Verlauf zu der Fehlersituation *Aufruf auf falscher Instanz*.

Die folgenden Tabellen zeigen nochmals die Instanzen und Methoden beider Diagramme. Die farbige Hinterlegung ermöglicht es hier, einen schnellen Überblick zu gewinnen.

Im zweiten Test hat die Instanziierung der Klasse funktioniert. Es wurden einige zusätzliche Aufrufe erkannt. An einer Stelle weicht die Reihenfolge von der spezifizierenden Reihenfolge ab und am Ende tritt ein Fehler auf, weil versucht wurde, ein Getränk aus dem Automaten zu entnehmen, welches nicht mehr vorhanden ist.

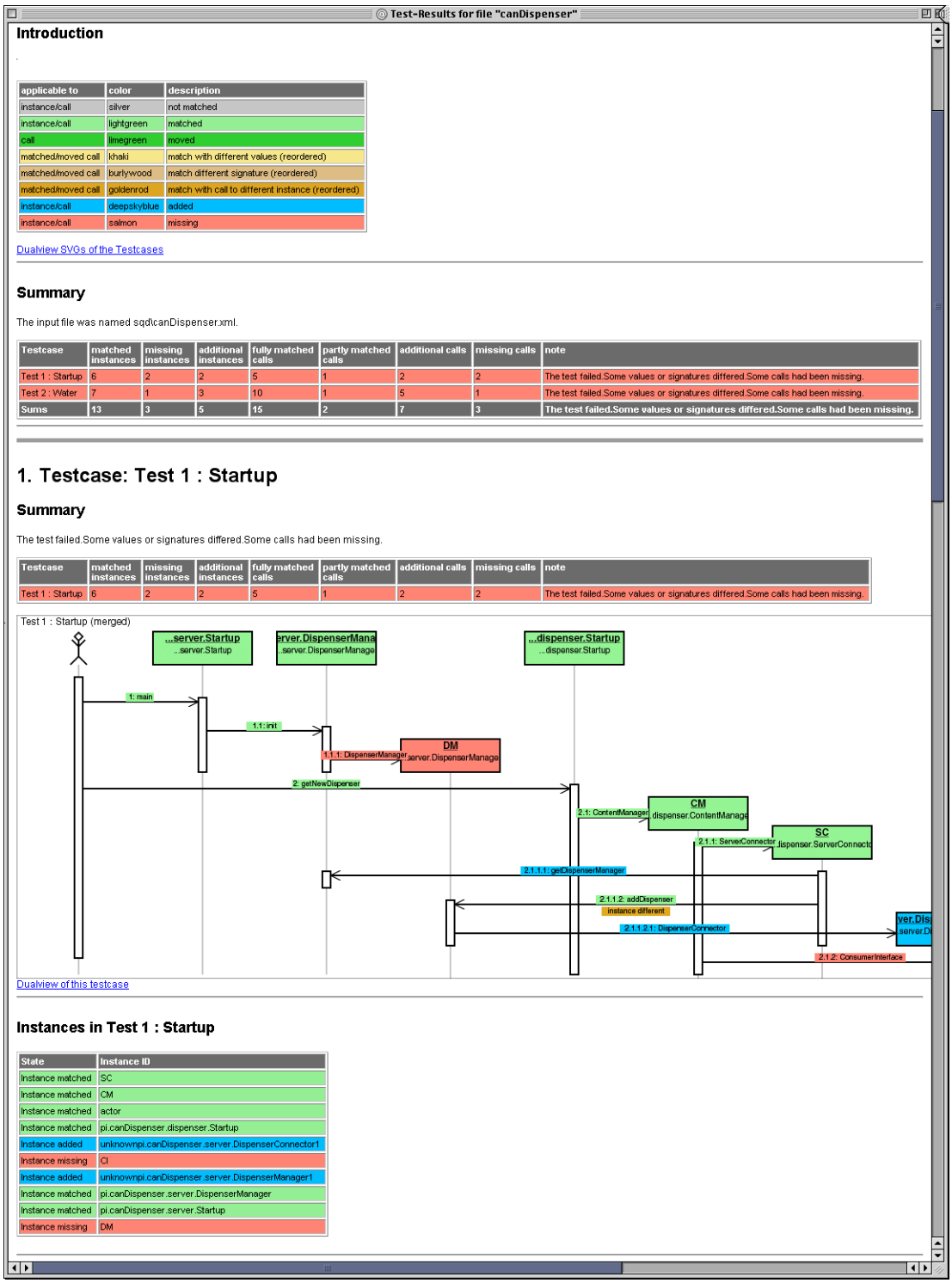


Abbildung 6.8: Erster Teil des HTML-Reports

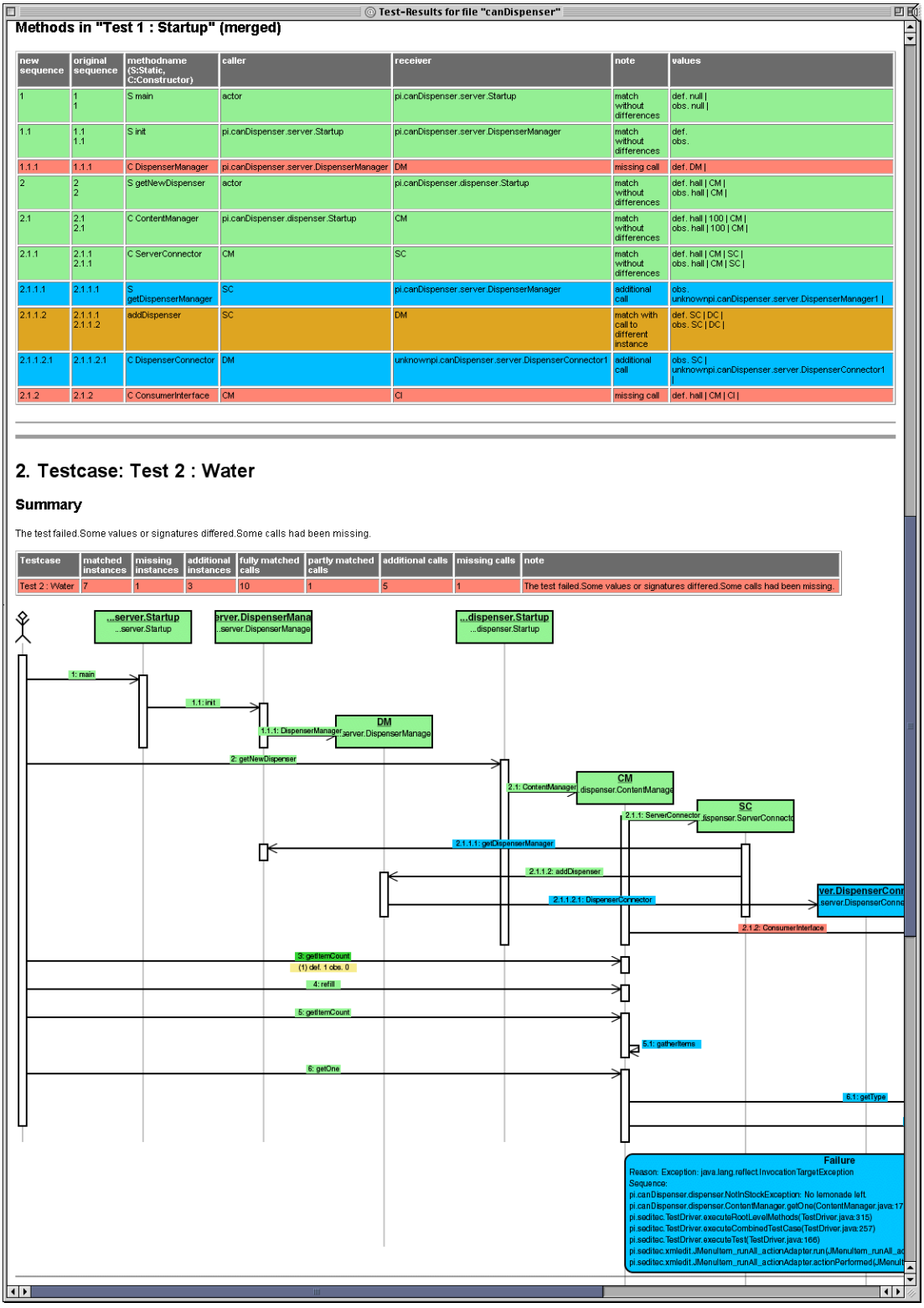
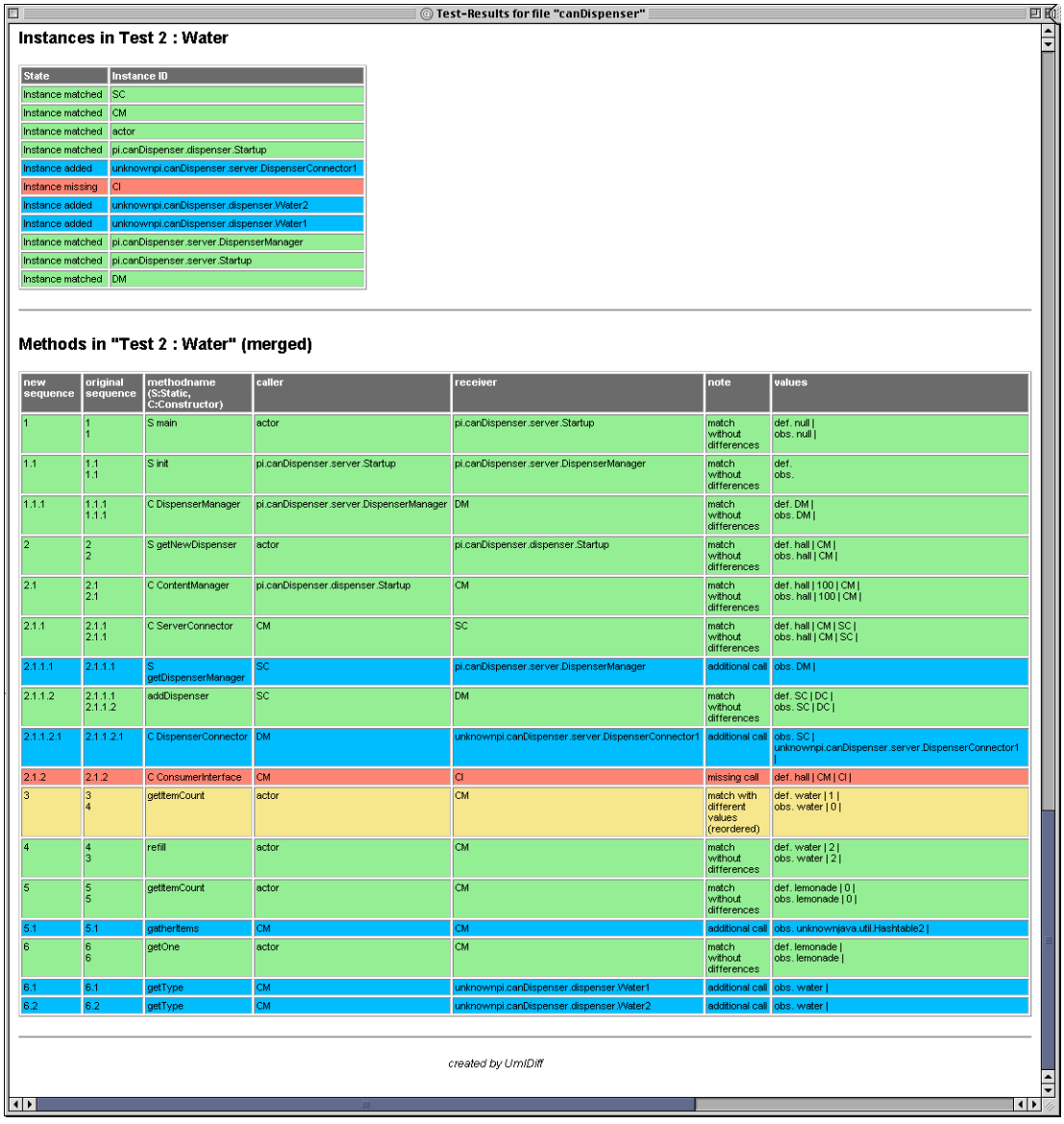


Abbildung 6.9: Zweiter Teil des HTML-Reports



Test-Results for file "canDispenser"

Instances in Test 2 : Water

State	Instance ID
Instance matched	SC
Instance matched	CM
Instance matched	actor
Instance matched	pi.canDispenser.dispenser.Startup
Instance added	unknownpi.canDispenser.server.DispenserConnector1
Instance missing	CI
Instance added	unknownpi.canDispenser.dispenser.Water2
Instance added	unknownpi.canDispenser.dispenser.Water1
Instance matched	pi.canDispenser.server.DispenserManager
Instance matched	pi.canDispenser.server.Startup
Instance matched	DM

Methods in "Test 2 : Water" (merged)

new sequence	original sequence	methodname (S:Static, C:Constructor)	caller	receiver	note	values
1	1 1	S main	actor	pi.canDispenser.server.Startup	match without differences	def. null | obs. null |
1.1	1.1 1.1	S init	pi.canDispenser.server.Startup	pi.canDispenser.server.DispenserManager	match without differences	def. obs.
1.1.1	1.1.1 1.1.1	C DispenserManager	pi.canDispenser.server.DispenserManager	DM	match without differences	def. DM | obs. DM |
2	2 2	S getNewDispenser	actor	pi.canDispenser.dispenser.Startup	match without differences	def. hall | CM | obs. hall | CM |
2.1	2.1 2.1	C ContentManager	pi.canDispenser.dispenser.Startup	CM	match without differences	def. hall | 100 | CM | obs. hall | 100 | CM |
2.1.1	2.1.1 2.1.1	C ServerConnector	CM	SC	match without differences	def. hall | CM | SC | obs. hall | CM | SC |
2.1.1.1	2.1.1.1	S getDispenserManager	SC	pi.canDispenser.server.DispenserManager	additional call	obs. DM |
2.1.1.2	2.1.1.1 2.1.1.2	addDispenser	SC	DM	match without differences	def. SC | DC | obs. SC | DC |
2.1.1.2.1	2.1.1.2.1	C DispenserConnector	DM	unknownpi.canDispenser.server.DispenserConnector1	additional call	obs. SC | unknownpi.canDispenser.server.DispenserConnector1 |
2.1.2	2.1.2	C ConsumerInterface	CM	CI	missing call	def. hall | CM | CI |
3	3 4	getItemCount	actor	CM	match with different values (reordered)	def. water | 1 | obs. water | 0 |
4	4 3	refill	actor	CM	match without differences	def. water | 2 | obs. water | 2 |
5	5 5	getItemCount	actor	CM	match without differences	def. lemonade | 0 | obs. lemonade | 0 |
5.1	5.1	gatherItems	CM	CM	additional call	obs. unknownjava.util.Hashtable2 |
6	6 6	getOne	actor	CM	match without differences	def. lemonade | obs. lemonade |
6.1	6.1	getType	CM	unknownpi.canDispenser.dispenser.Water1	additional call	obs. water |
6.2	6.2	getType	CM	unknownpi.canDispenser.dispenser.Water2	additional call	obs. water |

created by UmlDiff

Abbildung 6.10: Dritter Teil des HTML-Reports

Kapitel 7

Resümee und Ausblick

In den vorherigen Kapiteln wurden Verfahren zur Bestimmung der Unterschiede zwischen Klassendiagrammen und Sequenzdiagrammen entwickelt. Betrachtet man die dabei gewonnnenen Erkenntnisse, so fallen einige Besonderheiten auf.

Durch die genaue Definition der Semantik von UML-Modellierungselementen wurde die hier vorgestellte Analysetechnik erst ermöglicht. Da UML als hauptsächliches Darstellungselement Diagramme verwendet, ist eine eingängige Darstellung der Ergebnisse möglich.

Betrachtet man die eingangs definierten Ziele dieser Diplomarbeit (siehe Seite 9), so lassen sich folgende Schlüsse ziehen:

- Obwohl die beiden Diagrammtypen der Unified Modelling Language entstammen, weisen sie eine stark unterschiedliche Struktur auf. Somit sind die angewandten Verfahren nicht auf den anderen Diagrammtyp übertragbar. Auch die Anwendbarkeit auf andere UML-Diagrammtypen ist nicht möglich. In Abschnitt 6.3 wurden Kollaborationsdiagramme angesprochen. Diese kann man als eine alternative Repräsentationsform für Sequenzdiagramme auffassen, folglich können die dortigen Algorithmen auch bei diesen Diagrammen angewendet werden. Die Algorithmen zur Differenzanalyse von Klassendiagrammen wurden auch auf ihre Anwendbarkeit für beliebigen hierarchische Datenstrukturen untersucht. Durch Definition passender Regelwerke ist dies ebenfalls möglich. Das Tracking von Änderungen mit Transformationsoperationen kann so auch in anderen Bereichen sinnvoll angewendet werden.
- Ein Datenformat zur Beschreibung der Transformationen eines Sequenzdiagramms wurde nicht spezifiziert, da der mögliche Anwendungsbereich zu klein ist und ein solches Format vielen Anforderungen genügen muß.
- Sowohl für Klassen- als auch für Sequenzdiagramme konnte eine Nomenklatur zur Beschreibung der Differenzen zwischen Ausprägungen dieser Diagrammtypen erstellt werden.
- Neben der getrennten Betrachtung von Klassen- und Sequenzdiagrammen ist auch eine gemeinsame Analyse möglich, die beispielsweise Inkonsistenzen zwischen diesen beiden Darstellungsarten erkennt. Solche Ansätze wurden bereits in [22] vorgestellt.
- Die Möglichkeit, eine strukturelle Analyse eines Testlaufs durchzuführen, ergänzt die Funktionalität von SeDiTeC recht gut. Leider weist das Erkennungsverfahren noch einige Mängel

auf, die unter Umständen durch komplexere Algorithmen behoben werden können. So wird zur Zeit kein Backtracking durchgeführt. Das bedeutet, daß bei einer Uneindeutigkeit der Interpretation immer die erste Alternative gewählt wird. Die tritt beispielsweise auf, wenn zwei Aufrufe mit gleichen Parametern erfolgen und entschieden werden muß, welcher davon im definierenden Sequenzdiagramm spezifiziert wurde.

Erweiterungsmöglichkeiten für UMLDiff$_{cld}$

Das Werkzeug zur Differenzanalyse in Klassendiagrammen kann noch durch weitere Funktionalität ergänzt werden. Im folgenden sollen einige Ansätze hierzu vorgestellt werden.

Die Differenzanalyse der Struktur war Hauptinteresse dieser Arbeit. Unter Betrachtung von Zusatzeigenschaften, wie sie beispielsweise durch Eigenschaftswerte vorhanden sind, können weitere Vergleiche angestellt werden:

Differenzen von Metriken Durch Untersuchung der Metriken in beiden Klassendiagrammen und Vergleich der dabei entstehenden Werte können Aussagen über die qualitative Verbesserung eines Softwareprodukts gemacht werden.

Entwicklungsstand durch Eigenschaftswerte Das Konzept der Javadoc-Tags läßt sich für verschiedene Dinge nutzen. So wäre es denkbar, im Sourcecode jeder Methode ein Javadoc-Tag anzufügen, der den aktuellen Entwicklungsstand dieser Methode beschreibt. Denkbar sind Werte wie *Entwurf, Implementation, Test*. Diese Informationen lassen sich in dem Differenzklassendiagramm darstellen, Unterschiede zur Vorversion können farblich hervorgehoben werden. Somit läßt sich ein schneller Überblick über den derzeitigen Entwicklungsstand geben. Alternativ wäre eine Angabe über den Entwicklungsstand in Prozent denkbar, wobei hier eine Abschätzung der derzeitigen Situation schwierig ist.

Erkennung von Refactoring-Schritten Viele der Transformationsoperationen weisen Ähnlichkeiten mit Refactoring-Schritten auf. Weitere Refactorings können durch Zusammenfassung von Transformationsoperationen erkannt werden.

Erkennung von Entwurfsmustern Viele Entwurfsmuster lassen sich durch Analyse des Klassendiagramms erkennen und können als solche dargestellt werden. In [15] werden Refactoring-Verfahren vorgestellt, die versuchen, bestimmte Entwurfsmuster zu erzeugen.

Auf technischer Seite sind ebenfalls einige Verbesserungen denkbar:

ID-Matching Durch Verwendung von Eigenschaftswerten wird jedem Modellierungselement eine eindeutige, persistente ID zugewiesen, die den Erkennungsprozeß wesentlich verbessert. Hier muß allerdings auch der Entwickler dazu angehalten sein, keine Inkonsistenzen einzuführen (z.B. Umbennung eines Attributs zur Verwendung für anderen Zweck).

CVS-Support Durch Unterstützung einer Versionsverwaltungssoftware wie CVS könnte eine automatische Generierung von Differenzinformationen realisiert werden. Der aktuelle Projektstand wird regelmäßig aus der Versionsbibiliothek geholt und mit einer vorherigen Version verglichen. Durch nachträgliche Analyse könnten Rückschlüsse auf den Entwicklungsprozess gemacht werden.

vollständige Integration in CASE-Tool Durch eine direkte Anbindung an ein Softwareentwicklungswerkzeug wäre es unter Umständen möglich, die Änderungsoperationen direkt mitzuschreiben.

Das Konzept der farbigen Darstellung von Klassendiagrammen wurde bereits in verschiedenen Softwareentwicklungswerkzeugen umgesetzt. Durch das flexible Konzept des hier entwickelten Protoyps läßt sich die Darstellung für verschiedene Zwecke erweitern und verbessern. Je nach gewünschter Information kann Farbe oder andere visuelle Präsentationsmittel zur Kennzeichnung unterschiedlicher Aspekte verwendet werden:

Animation Die Darstellung der Transformationen kann zusätzlich zur farblichen Kennzeichnung durch Animation erfolgen. Beispielsweise kann dadurch das Verschieben von Elementen oder das Erzeugen von Klassen aus Interfaces dargestellt werden.

selektive Darstellung In [19] wurde ein Verfahren vorgestellt, welches gerade nicht relevante Teile aus einem Klassendiagramm ausblendet. Durch Navigation kann man so zu verschiedenen Klassen springen, woraufhin die zu dieser Klasse relevanten Klassen eingeblendet werden. Dies wäre auch für Differenzklassendiagramme denkbar.

Interaktion Der Benutzer erhält durch Anklicken der Modellierungselemente nährere Informationen, bei verschobenen Elementen beispielsweise die Quelle, aus der dieses Element stammt.

Abdeckungsanalyse Durch Speicherung der Ergebnisse einer Abdeckungsanalyse in den Eigenschaftswerten können diese auch in dem Klassendiagramm dargestellt werden.

Erweiterungsmöglichkeiten für UMLDiff$_{sqd}$

Bisher geht das Verfahren davon aus, daß die Eingabe ein spezifizierendes und ein aufgezeichnetes Sequenzdiagramm ist. Um sinnvoll Unterschiede zwischen zwei aus Testläufen generierten Diagrammen erkennen zu können, sind einige Modifikationen notwendig.

An der TU-Darmstadt wird im Rahmen einer Diplomarbeit am Fachgebiet Praktische Informatik ein Roundtrip-Werkzeug für das Testwerkzeug JUnit entwickelt, welches Sequenzdiagramme und Testklassen synchronisiert. Hier ist eine Strukturanalyse zur Erkennung von Unterschieden in Sequenzdiagrammen ebenfalls sehr hilfreich.

Bisher unterstützt SeDiTeC nur einfache, sequentielle Methodenaufrufe. Eine Unterstützung weiterer Programmkonstrukte wäre wünschenswert. Hier eine Auswahl:

- Unterstützung von Multithreading
- Möglichkeit der Spezifikation von zeitlichen Einschränkungen, wie sie beispielsweise in Echtzeitanwendungen benötigt werden.
- Spezifikationsmöglichkeit von „freien” Aufrufsequenzen. Hiermit ist ein Ausdrucksmittel gemeint, welches die Reihenfolge einer Menge von Aufrufen offen läßt. Im eigentlichen Testlauf kann diese dann beliebig gewählt werden, was Freiheiten in der Implementierung erlaubt.

- Anstatt reiner Testwerte ließen sich auch Testbereiche definieren. So lange der Meßwert innerhalb des Testbereichs liegt, kann er als richtig angesehen werden.

Literaturverzeichnis

[1] ANDRE, PASCAL, ANNYA ROMANCZUK, JEAN-CLAUDE ROYER und ALINE VASCONCELOS: *Checking the Consistency of UML Class Diagrams Using Larch Prover*. In: *Proceedings of Rigorous Object-Oriented Methods 3*, University of York, UK, January 2000.

[2] BANSIYA, JAGDISH: *Automating Design-Pattern Identification*. Dr. Dobb's Journal, June 1998. `http://www.ddj.com/documents/s=919/ddj9806a/9806a.htm`.

[3] BECK, KENT: *Extreme Programming Explained: Embrace Change*. Addison Wesley, New York, 2000.

[4] CHAWATHE, SUDARSHAN S. und HECTOR GARCIA-MOLINA: *Meaningful Change Detection in Structured Data*. In: *ACM SIGMOD International Conference on Management of Data*, Seiten 26–37, Tucson, Arizona, May 1997.

[5] CHIEN, SHU-YAO, VASSILIS J. TSOTRAS und CARLO ZANIOLO: *Version Management of XML Documents*. WebDB, Seiten 75–80, 2000.

[6] CHIEN, SHU-YAO, VASSILIS J. TSOTRAS und CARLO ZANIOLO: *XML Document Versioning*. SIGMOD Records, 30(3), 2001.

[7] COMPTON, KEVIN, YURI GUREVICH, JAMES HUGGINS und WUWEI SHEN: *An Automatic Verification Tool for UML*. University of Michigan EECS Department Technical Report, 423(00), 2000.

[8] CONRADI, REIDAR und BERNHARD WESTFECHTEL: *Configuring Versioned Software Products*. In: *6th International Workshop on Software Configuration Management*, Seiten 88–109, March 1996.

[9] EICHELBERGER, HOLGER: *Automatisches Zeichnen von UML-Klassendiagrammen durch den Sugiyama-Algorithmus*. In: S.DIEHL und ANDREAS KERREN (Herausgeber): *Tagungsband GI-Workshop Softwarevisualisierung 2000 (A01/00)*, Seiten 1–13, Saarbrücken, 2000. Universität Saarbrücken.

[10] FALLSIDE, DAVID C.: *XML Schema Part 0: Primer - W3C Recommendation, 2 May 2001*, 2000. `http://www.w3.org/TR/2001/REC-xmlschema-0-20010502/`.

[11] FERRAIOLO, JON: *Scalable Vector Graphics (SVG) 1.0 Specification*, 2000. `http://www.w3.org/TR/2000/CR-SVG-20001102/`.

[12] GUTTAG, JOHN V., JAMES J. HORNING, S.J. GARLAND, K.D. JONES, A. MODET und J.M. WING.: *Larch: Languages and Tools for Formal Specification*. Springer, New York, 1993. `http://www.sds.lcs.mit.edu/spd/larch/pub/larchBook.ps`.

[13] HOFFMANN, CHRISTOPH M. und MICHAEL J. O'DONNELL: *Pattern Matching in Trees*. Journal of the ACM, 29(1):68–95, 1982.

[14] HORWITZ, SUSAN: *Identifying the Semantic and Textual Differences Between Two Versions of a Program*. In: *ACM SIGPLAN90 Conference on Programming Language Design and Implementation*, Seiten 234–245, New York, 1990. ACM Press.

[15] KERIEVSKY, JOSHUA: *Refactoring To Patterns*. `http://industriallogic.com/`.

[16] KICZALES, GREGOR, JOHN LAMPING, ANURAG MENHDHEKAR, CHRIS MAEDA, CRISTINA LOPES, JEAN-MARC LOINGTIER und JOHN IRWIN: *Aspect-Oriented Programming*. In: AKŞIT, MEHMET und SATOSHI MATSUOKA (Herausgeber): *ECOOP '97 — Object-Oriented Programming 11th European Conference, Jyväskylä, Finland*, Band 1241, Seiten 220–242. Springer-Verlag, New York, NY, 1997.

[17] OBJECT MANAGMENT GROUP, INC. und OTHERS: *OMG Unified Modeling Language Specification, Version 1.3, June 1999*. OMG, Framingham, MA, 1999.

[18] OFFUTT, JEFF und AYNUR ABDURAZIK: *Generating Tests from UML Specifications*. In: *UML '99*, Band 1723, Seiten 416–429. Springer, 1999. `http://www.ise.gmu.edu/faculty/ofut/rsrch/papers/uml99.pdf`.

[19] RCZ, FERENC DSA und KAI KOSKIMIES: *Tool-Supported Compression of UML Class Diagrams*. In: ROBERT FRANCE, BERNHARD RUMPE (Herausgeber): *Proceedings of UML '99 2nd International Workshop, Fort Collins*, Band 2, Seiten 172–180, Berlin, 1999. Springer. ISBN: 3540667121.

[20] SELKOW, S. M.: *The tree-to-tree editing problem*. Information Processing Letters, 6(6):184–186, December 1977.

[21] SELONEN, P., K. KOSKIMIES und M. SAKKINEN: *How to Make Apples from Oranges in UML*. In: *34th Annual Hawaii International Conference on System Sciences*. IEEE, 2001.

[22] TSIOLAKIS, ALIKI: *Consistency Analysis of UML Class and Sequence Diagrams based on Attributed Typed Graphs and their Transformation*. Technical Report, 2000(3), 2000.

[23] UNIVERSITÄT PADERBORN: *Fujaba*. `http://www.fujaba.de/`.

[24] YANG, WUU: *How to Merge Program Texts*. The Journal of Systems and Software, 27(2):129ff, 1994.

[25] YUAN WANG, DAVID J. DEWITT, JIN-YI CAI: *X-Diff: A Fast Change Detection Algorithm for XMLDocuments*. `http://www.cs.wisc.edu/niagara/papers/xdiff.pdf`.

[26] ZELLER, ANDREAS und GREGOR SNELTING: *Unified Versioning through Feature Logic*. ACM Transactions on Software Engineering and Methodology, 6(4):398–441, 1997.