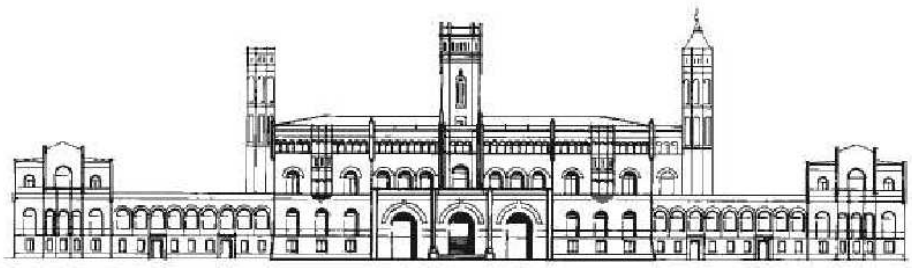

Leibniz Universität Hannover

Fakultät für Elektrotechnik und Informatik
Institut für Praktische Informatik
Fachgebiet Software Engineering

Kontinuierliche Qualitätssicherung von implementierten Architekturmodellen

Masterarbeit

im Studiengang Informatik

von

Jan Hinzmann

Erstprüfer	:	Prof. Dr. Kurt Schneider
Zweitprüferin	:	Prof. Dr. Nicola Henze
Betreuer	:	Dipl.-Wirt.-Inform. Daniel Lübke
Betreuer	:	Dipl.-Ing. Axel Berres

Hannover, den 05.06.2007

Zusammenfassung

Im Verlauf von Software-Projekten entfernt sich die Implementierung vom Design. Dieser Drift liegen Modelldifferenzen zu Grunde, welche projektbegleitend durch Änderungen im Design oder der Implementierung entstehen. In der Regel stellt dies einen Fortschritt dar, weil Probleme oft erst bei der Implementierung erkannt und behoben werden können. Der Nachteil ist aber, dass die Implementierung das Design nicht mehr vollständig abbildet. Damit das Design nicht veraltet, gibt es verschiedene Ansätze, die beiden Modelle miteinander zu synchronisieren (z. B. Roundtrip Engineering). Bei der Synchronisierung wird der Aspekt der Kommunikation in der Regel nicht berücksichtigt, da mit einem Reverse-Engineering-Werkzeug lediglich einen Blick auf das momentane System aus einer anderen Perspektive gewährt wird.

Das Erkennen der Drift deckt in der Regel Kommunikationsbedarf auf, da nicht klar ist, ob das Design oder die Implementierung verbessert werden muss. In dieser Arbeit wird deshalb ein System zur Erkennung von Modelldifferenzen zwischen Design und Implementierung erarbeitet, welche projektbegleitend kommuniziert werden sollen. Hierzu wird die Technik der statischen Codeanalyse benutzt und der übliche Ansatz erweitert, sodass nicht nur eine Datei analysiert wird, sondern zwei Modelle in ihrer Gesamtheit (jeweils bestehend aus mehreren Dateien). Das System ist so entwickelt, dass sich Differenzen zwischen einem Architektur- und einem Entwicklermodell während eines Softwareprojektes kontinuierlich und automatisiert aufdecken lassen, sodass etwaige Abweichungen frühzeitig erkannt, die nötige Kommunikation initiiert und die Modelle entsprechend angepasst werden können. Durch dieses Vorgehen wird nicht nur das Design als Dokumentation aktuell gehalten, es dient gleichermaßen als überprüfbare Vorgabe für die Implementierung.

für Malin, Nick und Kathy

Inhaltsverzeichnis

Glossar

In dieser Arbeit werden folgende Begriffe und ihre Abkürzungen verwendet.

Architekt (A)	Der Architekt erstellt das Architektenmodell.
Entwickler (E)	Der Entwickler erstellt das Entwicklermodell.
Architektenmodell (AM)	Das Architektenmodell ist eine Menge von Quelltextdateien, die aus UML-Diagrammen generiert worden sind.
Entwicklermodell (EM)	Das Entwicklermodell ist eine Menge von Quelltextdateien, die die Architektur implementieren sollen. In einigen Grafiken ist das EM auch als DM (Developer Model) ausgezeichnet.
Modelldifferenzen	Modelldifferenzen sind Abweichungen des EMs vom AM.
MoDi	Ein System zur Erkennung von Modelldifferenzen.
Metarepräsentation (MR)	Die Metarepräsentation ist eine vom AM bzw. EM abstrahierte Darstellung der Architektur bzw. der Implementierung.
Checks (C)	Checks führen Untersuchungen auf der MR durch und finden so Modelldifferenzen.
Regeln (R)	Regeln bewerten Modelldifferenzen bzgl. ihres Schweregrades.
Modellvergleich (MV)	Ein Modellvergleich evaluiert konfigurierte Regeln, die Modelldifferenzen zwischen dem EM und dem AM durch Ausführung von Checks erkennen und bewerten.
Bericht (B)	Der Bericht präsentiert die Menge von Ergebnissen des MVs.
Code-Basis (CB)	Eine Code-Basis ist eine Menge von kompilierbaren Quelltextdateien.
Implementierter Code (IC)	Eine CB, die vom Entwickler erstellt worden ist.
Architekten Code (AC)	Eine CB, die vom Architekten (per UML-Werkzeug) erstellt worden ist und als AM-Äquivalent gilt.

Kapitel 1

Einleitung

Zunächst soll die Arbeit in ihren Kontext eingeordnet und das Problemfeld beschrieben werden. Anschließend werden die Motivation und die Zielsetzung erläutert und schließlich die Gliederung dieses Berichtes vorgestellt.

1.1 Kontext der Arbeit

In aktuellen Vorgehensmodellen der objektorientierten Analyse und Design (OOAD) gibt es im Wesentlichen die drei Rollen des Software-Kundens, des -Architekten und des -Entwicklers.

Nachdem die Anforderungen erarbeitet und festgeschrieben sind, kann ein Architekt in der objektorientierten Design (OOD)-Phase ein Modell der Software erstellen, welches die Anforderungen und somit die Wünsche des Kunden erfüllt. Dieses Architektenmodell (AM) liegt idealerweise in Form von UML-Diagrammen vor, die automatisch in Quellcode überführt werden können. Anschließend wird das AM an das Entwicklerteam übergeben, welches die fehlende Implementierung bewerkstelligt und so das Entwicklermodell (EM) erstellt. Schließlich kann die entstandene Software an den Kunden ausgeliefert werden und das Projekt geht in die Wartungsphase über. Der Entwicklungsprozess ist dann abgeschlossen.

Diese Sicht der Software-Entwicklung ist natürlich stark idealisiert. In Projekten kann es in jeder Phase zu Problemen und Abweichungen von dem oben geschilderten Ablauf kommen. Insbesondere weicht oft das Entwicklermodell vom Architektenmodell ab, sodass das fertige Produkt nicht mehr die ursprünglichen Anforderungen erfüllt. Diese Abweichungen werden in dieser Arbeit *Modelldifferenzen* genannt. Im Verlauf eines Projektes entfernt sich das Entwicklermodell normalerweise immer weiter vom anfänglich spezifizierten Architektenmodell, welches im schlimmsten Fall erst am Ende des Projektes als Dokumentation erneut erstellt wird. So werden im Moment UML-Modelle oft nur am Anfang und am Ende eines Software-Projektes genutzt. Am Anfang, um das initiale Design festzulegen und am Ende, um die Dokumentation zu erstellen und die entstandene Architektur festzuhalten. Viel zu selten kommt es vor, dass auch während des Projektes UML-Modelle aus dem Entwicklermodell erstellt und diskutiert bzw. mit einem Architektenmodell verglichen werden. Dabei ist gerade

ein frühzeitiges Erkennen von Modelldifferenzen wünschenswert.

Hier soll das zu implementierende System MoDi helfen, die entstehenden Modelldifferenzen projektbegleitend zu identifizieren, Kommunikation anzuregen, damit das Design und die Implementierung immer wieder synchronisiert werden kann.

Der Vollständigkeit halber sei noch erwähnt, dass der Entwicklungsprozess zwar mit der Auslieferung abgeschlossen ist, es aber auch in der Wartungsphase zu Änderungen an der Software und somit zu Modelldifferenzen kommen kann, welche dann auch aufgedeckt und kommuniziert werden müssen.

1.2 Motivation und Zielsetzung

Diese Arbeit konzentriert sich auf die Entwicklungsphase, also die Phase, in welcher der Architekt dem Entwickler seinen Entwurf kommuniziert und dieser die Vorgabe durch Auswahl von geeigneten Technologien und Implementierungsarbeit umsetzt. Dies geschieht nicht einmalig, sondern in einem iterativen Prozess, der in Abbildung 1.1 dargestellt ist.

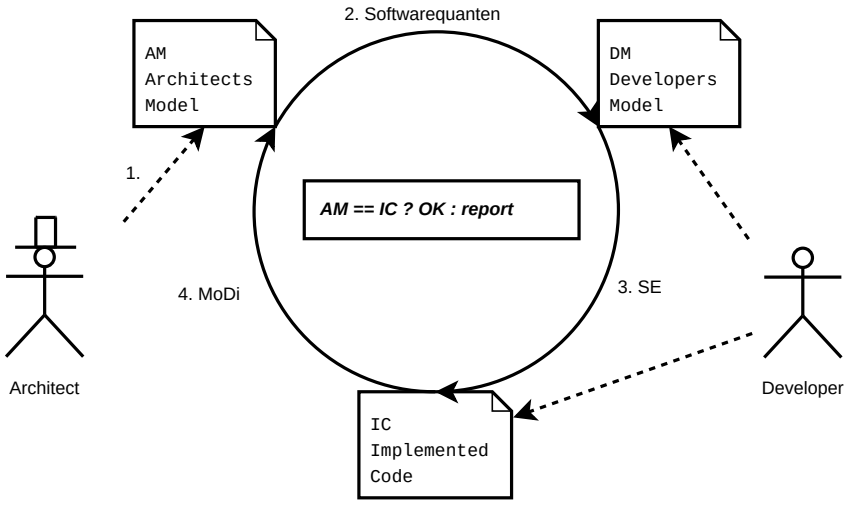

Abbildung 1.1: Darstellung des iterativen Softwareentwicklungsprozesses

Die einzelnen Schritte seien hier noch einmal erklärt. Es wird

1. ein Design erstellt oder geändert und als Architektenmodell (AM) für diese Iteration gekennzeichnet,
2. das AM kommuniziert, sodass das EM entsteht, es aber auch zu Missverständnissen (Softwarequanten [1])kommen kann,

[1] Softwarequanten stellen das Prinzip des Spiels „Stille Post“ dar und verdeutlichen die Stellen im Informationsfluss, an denen es zu Missverständnissen kommt.

3. das EM umgesetzt und so der IC erstellt (SE) und
4. in einem Modellvergleich überprüft, ob der IC mit dem aus dem AM generierten Architekten-Code (AC) übereinstimmt, es also keine Modelldifferenzen gibt. Stimmen AC und IC nicht überein, obwohl alle Missverständnisse ausgeräumt sind, so gibt es zwei Möglichkeiten:
   (a) Das Entwicklermodell muss überarbeitet werden (weiter bei 2.).
   (b) Das Architektenmodell muss überarbeitet werden (weiter bei 1.).
5. In jedem Fall müssen Modelldifferenzen kommuniziert werden, um die Drift der Implementierung vom Design zu vermeiden.

Anmerkung: Im Fall (a) in Schritt 4, soll bei 2. fortgefahren werden. Dies kann im ersten Moment merkwürdig erscheinen und der kritische Leser fragt sich sicherlich, wieso es nicht bei 3. weitergeht. Der Rücksprung in die Kommunikationsphase (2.) ist aber durchaus beabsichtigt und dient dazu etwaige Missverständnisse auszuräumen. Setzt man eine gewisse technische Erfahrung bei den Beteiligten voraus, so ist es sogar viel wahrscheinlicher, dass ein Missverständnis vorliegt, als der Fall, in dem das kommunizierte Architektenmodell falsch umgesetzt worden ist.

Die entstehenden Modelldifferenzen dokumentieren den Fortschritt der Entwicklung. Sie können aber auch Anzeichen für Probleme im Projekt sein. Der zweite Fall ist für diese Arbeit interessant.

Modelldifferenzen beruhen nicht nur auf Missverständnissen zwischen dem Architekten und den Entwicklern, sondern sind auch das Resultat von unerwarteten Schwierigkeiten in der Implementierung. Gründe für Modelldifferenzen können auch ein eingeschränkter Planungshorizont des Architekten sein. Bei sich ergebenen technologischen Problemen zwingt das Entwicklermodell das Architektenmodell zu einer Anpassung und so muss, nach erneuter Kommunikation zwischen den Verantwortlichen, wenigstens eines der beiden angepasst werden. Wichtig ist hier, dass die Entwickler das Problem rechtzeitig eskalieren.

Damit ein Projekt zum Erfolg führen kann, muss das Entwicklermodell das Architektenmodell abbilden, damit der Kunde – der in der Regel höchstens das Architektenmodell kennt – zufrieden gestellt wird und es nicht zu Vertragsverletzungen kommt.

Da auf der einen Seite des Vergleichs das Architektenmodell in Form von UML-Diagrammen und auf der anderen Seite das Entwicklermodell in Form von Code-Basen steht, muss zunächst eine Vergleichsbasis hergestellt werden.

Um das Problemfeld beherrschen zu können, wird sich auf die Sprache Java beschränkt und hier werden zunächst nur die Interfaces untersucht. Diese werden während der OOAD-Phase spezifiziert und später umgesetzt. Die Interfaces müssen auf ihre Bestandteile untersucht und die Anteile identifiziert werden, die variabel sind (z.B. Namen, Methoden, Konstanten). Anschließend müssen Regeln abgeleitet werden, die im Modellvergleich überprüft werden.

Die Ergebnisse des Modellvergleichs sollen anschließend in einem Bericht zusammengefasst

und in passender Repräsentation an entsprechende Teilnehmer des Entwicklungsprozesses kommuniziert werden.

Das oben angedeutete Werkzeug zur Erkennung von Modelldifferenzen soll in einem Prototypen verwirklicht und in einer Fallstudie erprobt werden.

1.3 Gliederung

Die Arbeit gliedert sich folgendermaßen:

- Kapitel 2 beschreibt allgemeine Grundlagen, die zum Verständnis des Werkzeugs notwendig sind, sowie die verwendete Technologie, die bei seiner Realisierung zum Einsatz kommt. Außerdem wird das Deutsche Zentrum für Luft- und Raumfahrt DLR vorgestellt und insbesondere auf die Einrichtung Simulations- und Softwaretechnik SISTEC eingegangen, in deren Abteilung in Braunschweig diese Arbeit entstanden ist.
- Kapitel 3 beschreibt die erarbeiteten Konzepte, welche die Basis für ein Werkzeug zur Erkennung von Modelldifferenzen bilden.
- Kapitel 4 beschreibt die Konsequenzen für den Softwareprozess, die sich durch den Einsatz von Werkzeugen zum Erkennen von Modelldifferenzen (insbesondere durch das in dieser Arbeit realisierte Werkzeug MoDi) ergeben.
- Kapitel 5 beschreibt die Realisierung von MoDi, mit dessen Hilfe sich Modelldifferenzen aufdecken lassen.
- Kapitel 6 beschreibt wie das MoDi-System an zwei Projekten, an denen das DLR beteiligt war, erprobt worden ist. Die Untersuchung des Projektes [SiLEST] fand im Nachhinein statt und es wurden Modelldifferenzen zwischen einzelnen Revisionen des Projektes aufgedeckt. Außerdem wird der Einsatz von MoDi im laufenden Projekt [SESIS] erprobt.
- Kapitel 7 nennt einige verwandte Arbeiten und grenzt die eigene dagegen ab.
- Kapitel 8 fasst die Ergebnisse der Arbeit zusammen und gewährt einen Ausblick auf das weitere Vorgehen. Dies ist im Wesentlichen eine kritische Betrachtung der eigenen Arbeit und ein Aufzeigen der Möglichkeiten für eine weitere Entwicklung bzw. Erweiterung des Systems zur Erkennung von Modelldifferenzen MoDi.

Kapitel 2

Allgemeine Grundlagen

In diesem Kapitel werden die allgemeinen Grundlagen und die verwendete Technologie beschrieben, die für die Erstellung des Werkzeuges zur Erkennung von Modelldifferenzen benötigt wurden.

Diese Masterarbeit ist als externe Arbeit in der Einrichtung Simulations- und Softwaretechnik (SISTEC) des Deutschen Zentrums für Luft- und Raumfahrt (DLR) entstanden. Im Folgenden wird deshalb zunächst das DLR und anschließend die Einrichtung SISTEC kurz vorgestellt.

2.1 Deutsches Zentrum für Luft- und Raumfahrt (DLR)

Das Deutsche Zentrum für Luft- und Raumfahrt stellt sich folgendermaßen dar:

> Das DLR ist das nationale Forschungszentrum für Luft- und Raumfahrt der Bundesrepublik Deutschland. Seine umfangreichen Forschungs- und Entwicklungsarbeiten sind in nationale und internationale Kooperationen eingebunden. Über die eigene Forschung hinaus ist das DLR als Raumfahrt-Agentur im Auftrag der Bundesregierung für die Planung und Umsetzung der deutschen Raumfahrtaktivitäten sowie für die internationale Interessenswahrnehmung zuständig. Zudem fungiert das DLR als Dachorganisation für den national größten Projektträger.
>
> In 27 Instituten und Einrichtungen an den acht Standorten Köln-Porz, Berlin-Adlershof, Bonn-Oberkassel, Braunschweig, Göttingen, Lampoldshausen, Oberpfaffenhofen und Stuttgart beschäftigt das DLR ca. 5.300 Mitarbeiterinnen und Mitarbeiter. Das DLR unterhält Außenbüros in Brüssel, Paris und Washington, D.C.
>
> Die Mission des DLR umfasst die Erforschung von Erde und Sonnensystem, Forschung für den Erhalt der Umwelt und umweltverträgliche Technologien, zur Steigerung der Mobilität sowie für Kommunikation und Sicherheit. Das Forschungsportfolio des DLR reicht in seinen Geschäftsfeldern Luftfahrt, Raumfahrt, Verkehr und Energie von der Grundlagenforschung zu innovativen Anwendungen und Produkten von morgen. So trägt das im DLR gewonnene wissenschaftliche

> und technische Know-How zur Stärkung des Industrie- und Technologiestandortes Deutschland bei. Das DLR betreibt Großforschungsanlagen für eigene Projekte sowie als Dienstleistung für Kunden und Partner. Darüber hinaus fördert das DLR den wissenschaftlichen Nachwuchs, betreibt kompetente Politikberatung und ist eine treibende Kraft in den Regionen seiner Standorte.
>
> Quelle: `http://www.dlr.de` Webseite

Innerhalb des DLR ist die Einrichtung Simulations- und Softwaretechnik (SISTEC) auf die Standorte Braunschweig und Köln verteilt. Diese Arbeit fand größtenteils in Braunschweig statt. Im Folgenden wird die Einrichtung SISTEC beschrieben.

2.1.1 Einrichtung Simulations- und Softwaretechnik

Software ist zu einem wesentlichen Erfolgsfaktor in Forschungs- und Entwicklungs-Projekten geworden. Ihre Entwicklung erfordert aufgrund wachsender Komplexität in zunehmendem Maße IT-Spezialkenntnisse. Die DLR-Einrichtung Simulations- und Softwaretechnik (SISTEC) beteiligt sich an der internationalen Softwareforschung und übernimmt anspruchsvolle Softwareentwicklungsaufgaben in Projekten mit DLR-Instituten. Dies schließt die Entwicklung eigener Softwareprodukte für den Fall ein, dass keine geeigneten kommerziell etablierten Lösungen verfügbar sind. SISTEC ist an den DLR-Standorten Köln und Braunschweig vertreten.

Objektorientierte und komponentenbasierte Softwaretechnologie ist Grundlage moderner Softwareentwicklung in den Bereichen verteilte Systeme, Software-Integration und Workflow-Management für Simulationsanwendungen, Management wissenschaftlicher Daten und graphische Benutzeroberflächen. Mit Produkten und Projektbeiträgen zu diesen Themen ergänzt SISTEC das anwendungsspezifische Know-How seiner Projektpartner. Einen Schwerpunkt bildet die Einführung der Grid-Technologie in DLR-Anwendungen.

SISTEC unterstützt Softwareentwickler im DLR bei der Einführung moderner Software-Engineering-Methoden und -Werkzeuge. Ausgehend von den Erfahrungen eigener Softwareprojekte entwickelt SISTEC effektive Qualitätssicherungs- und Testverfahren für sicherheitskritische Echtzeitsoftware (z.B. zur Satellitenlageregelung). In Zusammenarbeit mit der Industrie und der ESA ist SISTEC an der Erstellung simulationsbasierter Entwurfssysteme für den Schiff- und Satellitenbau beteiligt.

2.2 Backus Naur Form (BNF)

Eine *Backus Naur Form* (BNF) stellt eine Notation dar, mit der die Syntax einer formalen Sprache bzw. ihre Grammatik beschrieben werden kann. In der Notation wird eine Menge von Regeln verwandt, die zusammengenommen die Menge der Wörter beschreiben, die Element der Sprache sind. Auf Programmiersprachen angewandt lässt sich mit Hilfe einer in BNF beschriebenen Grammatik zu einem gegebenen Quelltext die Aussage treffen, ob es sich um

ein gültiges Programmfragment handelt, oder nicht. Die BNF spielt beim Compilerbau eine besondere Rolle in der Phase der Syntaktischen-Analyse.

2.2.1 Allgemeines

Das folgende Zitat [BNF] gibt einen Überblick:

> BNF is an acronym for "Backus Naur Form". John Backus and Peter Naur introduced for the first time a formal notation to describe the syntax of a given language (This was for the description of the ALGOL 60 programming language, see [Naur 60]). To be precise, most of BNF was introduced by Backus in a report presented at an earlier UNESCO conference on ALGOL 58. Few read the report, but when Peter Naur read it he was surprised at some of the differences he found between his and Backus's interpretation of ALGOL 58. He decided that for the successor to ALGOL, all participants of the first design had come to recognize some weaknesses, should be given in a similar form so that all participants should be aware of what they were agreeing to. He made a few modificiations that are almost universally used and drew up on his own the BNF for ALGOL 60 at the meeting where it was designed. Depending on how you attribute presenting it to the world, it was either by Backus in 59 or Naur in 60. (For more details on this period of programming languages history, see the introduction to Backus's Turing award article in Communications of the ACM, Vol. 21, No. 8, august 1978. This note was suggested by William B. Clodius from Los Alamos Natl. Lab).

2.2.2 BNF erklärt in BNF

Die Syntax der [BNF] selbst lässt sich in BNF beschreiben:

```
1      syntax ::= { rule }
2      rule ::= identifier "::=" expression
3      expression ::= term { "|" term }
4      term ::= factor { factor }
5      factor ::= identifier
6          | quoted_symbol
7          | "(" expression ")"
8          | "[" expression "]"
9          | "{" expression "}"
10     identifier ::= letter { letter | digit }
11     quoted_symbol ::= """ { any_character } """
```

Listing 2.1: Die BNF erklärt in BNF

In Listing 2.1 wird die BNF in BNF beschrieben. Dabei gilt:

- Das in Zeile 2 definierte Symbol `::=` heißt „definiert“.

- Das in Zeile 3 definierte Symbol | heißt „oder“.
- Die in Zeile 7 definierten Klammern gruppieren Ausdrücke.
- Die in Zeile 8 definierten Klammern deklarieren optionale Ausdrücke (0 oder 1)
- Die in Zeile 9 definierten Klammern deklarieren wiederholbare Ausdrücke ($0..n$)

Im Wesentlichen geht es darum, zu einem gegebenen Wort, welches auch einen Programmtextellen kann, aus einer Startregel eine Ableitung herzustellen, sodass das Wort akzeptiert wird. Hierzu dient das Konzept der *Terminale* und *Nonterminale*, wobei ein Nonterminal eine noch abzuleitende Einheit darstellt. Können alle Nonterminale mit Hilfe einer gegebenen Regeln zu einem Terminal abgeleitet werden, so sagt man: Das Wort ist Element der Sprache. Die Regeln, welche die Sprache beschreiben, stellen die Grammatik dar und werden mit Hilfe der BNF-Notation beschrieben. Zu einer Programmiersprache existiert in der Regel eine Beschreibung der Syntax in BNF.

Die Aussage „Ein Wort ist Element der Sprache“ ist deshalb so wichtig, da sie ein Entscheidungskriterium darstellt, ob ein Programmtext kompilierbar ist oder nicht. Diese Entscheidung ist für das Werkzeug zur Erkennung von Modelldifferenzen essentiell, da die Kompilierbarkeit der zu prüfenden Code-Basen eine Voraussetzung für die Verarbeitung darstellt.

Im Folgenden wird das Werkzeug ANTLR beschrieben, mit dessen Hilfe sich zu einer kontextfreien Grammatik in einem leicht abgewandelten BNF-Format ein Parsersystem generieren lässt.

2.3 Another Tool for Language Recognition (ANTLR)

Das ANTLR System generiert auf Grundlage einer in BNF (siehe Abschnitt 2.2, S. 14) gegebenen Grammatik einen Lexer und einen Parser, mit deren Hilfe sich ein Abstrakter Syntaxbaum (*Abstract Syntax Tree*, AST) aus einer Quelltextdatei erstellen lässt. Der AST stellt eine Ableitung des Wortes dar und wird zur weiteren Verarbeitung in Übersetzern, Interpretern und Analysewerkzeugen benötigt. Beim Parsen wird die Datei zeichenweise eingelesen, wobei der Lexer die einzelnen Zeichen zu sogenannten *Tokens* zusammenfasst und dem Parser zum Aufbau eines Abstrakten Syntaxbaums bereitstellt.

Auf der Internetseite von ANTLR wird das Parsersystem folgendermaßen beschrieben:

> What is ANTLR? ANTLR, ANother Tool for Language Recognition, (formerly PCCTS) is a language tool that provides a framework for constructing recognizers, compilers, and translators from grammatical descriptions containing Java, C#, C++, or Python actions. ANTLR provides excellent support for tree construction, tree walking, and translation. There are currently about 5,000 ANTLR source downloads a month. - `http://www.antlr.org` (23.04.2007)

Das ANTLR-System wird seit 1989 von Terence Parr, einem Professor für Informatik an der Universität von San Francisco, entwickelt. Er beschreibt seine Arbeit zum Beispiel in [TParr2006].

Beim Einsatz von ANTLR lässt sich aus Quelltexten ein AST erstellen. Hierzu wird eine zur Sprache passende Grammatik benötigt, mit deren Hilfe das Parser-System generiert wird. Es sind verschiedene Grammatiken für Sprachen wie beispielsweise Java, C, C++, Python vorhanden. In dieser Arbeit findet die Java-1.5-Grammatik Anwendung, wie sie auch im Checkstyle-Projekt [Checkstyle] eingesetzt wird. Zum Zeitpunkt der Arbeit hat die Checkstyle-Grammatik eine bessere Qualität, als die auf der ANTLR-Seite angebotene Java-1.5-Grammatik.

Nachdem zu einer Grammatik ein Parsersystem generiert ist, können Dateien der zur Grammatik korrespondierenden Sprache geparst werden. Dies übernehmen die von ANTLR generierten Komponenten `Lexer` und `Recognizer`. Der durch das Parsen erstellte AST muss anschließend ausgewertet werden, was auf verschiedene Arten geschehen kann. ANTLR selbst bietet hierzu das Konzept der *Tree Grammars* an, bei dem in einer Kopie der Grammatik semantische Regeln hinterlegt und anschließend ein `TreeParser` generiert werden kann, der die semantischen Regeln beim Traversieren des Baumes ausführt. Dieses Konzept wird vom Terence Parr in [TParr2004] propagiert.

Alternativ kann der AST aber auch manuell durchwandert werden, um die erforderlichen Informationen auszulesen. In [ATripp2007] wird deutlich, warum dies die bessere Möglichkeit ist, weshalb in dieser Arbeit auch ein manuelles Durchlaufen das Baumes mit einem eigenen `Treewalker` implementiert wird. Bei diesem Ansatz wird keine Kopie der Grammatik benötigt und somit keine Redundanz erzeugt. Außerdem wird die Wartbarkeit des Systems erhöht und es werden keine Sprachkonzepte vermischt, da ein Treewalker in einer einzigen Sprache geschrieben werden kann. So wird die kognitive Last verringert und das System ist leichter verständlich.

Der folgende Abschnitt beschreibt kurz die Unified Modelling Language und führt in Anlehnung an [Pugh2006] das Beispiel eines Pizzabringdienstes ein, welches im Verlauf der Arbeit immer wieder aufgegriffen wird.

2.4 Unified Modelling Language (UML)

Die *Unified Modelling Language* (UML) ist eine erweiterbare Sprache zur Modellierung von Software und wurde von der Object Management Group (OMG) spezifiziert. Mit ihr lässt sich das statische und dynamische Verhalten sowie Schnittstellen von Software beschreiben. Die UML unterteilt sich in Struktur- und Verhaltensdiagramme. In dieser Arbeit wird es in erster Linie um Strukturdiagramme und insbesondere um Klassendiagramme gehen.

In Abbildung 2.1 ist in einem solchen Klassendiagramm beispielhaft ein Pizza-Bringdienst in Anlehnung an [Pugh2006] modelliert. In Klassendiagrammen werden Klassen als Rechteck gezeichnet. Durch waagerechte Linien ist das Rechteck in drei Bereiche unterteilt, welche von oben nach unten den Namen der Klasse, ihre Attribute und schließlich ihre Operationen ent-

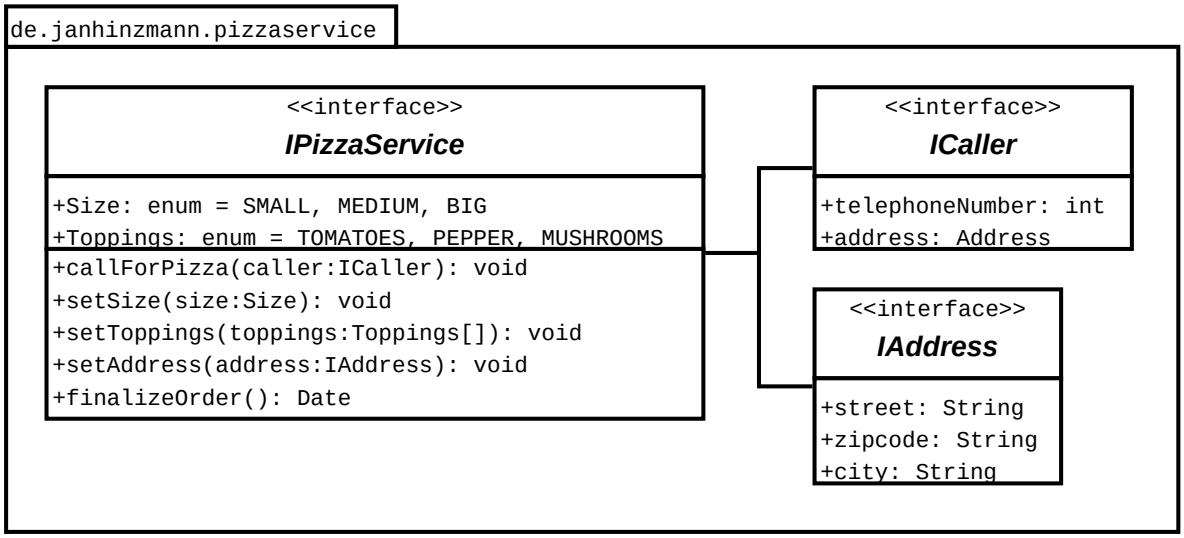

Abbildung 2.1: Ein Pizza-Bringdienst in UML modelliert

halten. Durch die kursive Schreibweise des Klassennamens (*IPizzaService*) wird verdeutlicht, dass es sich um eine abstrakte Klasse handelt und das Schlüsselwort `interface` (in spitzen Klammern) kennzeichnet die Klasse als Interface.

Beziehungen zwischen Klassen werden durch Linien bzw. Pfeile beschrieben, die verschiedene Enden haben und an denen auch Kardinalitäten notiert werden können. Die möglichen Beziehungen und insbesondere die Pfeilarten sind vielfältig und an dieser Stelle sei auf die Spezifikation der UML [UML] verwiesen.

Das Klassendiagramm in Abbildung 2.1 (Seite 18) zeigt die Interfaces *IPizzaService*, *ICaller* und *IAddress*, welche für das Zusammenspiel bei einer Pizza-Bestellung benötigt werden. Das zentrale Objekt ist das Interface *IPizzaService*, welches Methoden zum Bestellen einer Pizza bereithält. Die Interfaces *ICaller* und *IAddress* stellen den Kunden bzw. die Adresse eines Kundens bei einer Pizza-Bestellung dar.

Das Protokoll, also die Reihenfolge, in der die Operationen des *IPizzaService*-Interfaces aufgerufen werden müssen, lässt sich in einem Sequenzdiagrammen darstellen [UML]. Die im Folgenden beschriebene Reihenfolge ist aus dem Klassendiagramm zwar ersichtlich, jedoch nicht vorgeschrieben. Ein Kunde wird zunächst die Methode `callForPizza` aufrufen und sich selbst als Parameter übergeben (damit der Pizza-Bringdienst in seinen Daten nachschauen kann, ob der Kunde schon einmal etwas bestellt hat). Anschließend äußert der Kunde mit Hilfe der Methoden `setSize` und `setToppings` seine Wünsche bzgl. der Größe und des Belages der Pizza und teilt dem Bringdienst seine Adresse durch den Aufruf der Methode `setAddress` mit. Schließlich beendet er den Bestellvorgang durch den Aufruf von `finalizeOrder` und bekommt den Zeitpunkt mitgeteilt, wann die Pizza geliefert wird.

Da in dieser Arbeit UML nur indirekt benutzt wird, sei für weitere Diagrammtypen und Erläuterungen entweder auf die Spezifikation der OMG [UML] oder eines der zahlreichen UML-Bücher, wie beispielsweise [PiloneP2005], verwiesen.

2.5 Zusammenfassung

In diesem Kapitel wurden die Grundlagen besprochen, die im Folgenden für diese Arbeit wichtig sind. Im Wesentlichen ist dies das ANTLR-Framework, mit dem der Java-Parser zur Checkstyle-Grammatik generiert wurde. Das Parsersystem ist die Ausgangsbasis für die weitere Verarbeitung durch das Werkzeug zur Erkennung von Modelldifferenzen. Weiterhin wurde ein kleiner Einblick in die Strukturdiagramme, speziell das Klassendiagramm gegeben. Das Kapitel schließt mit dem Beispiel eines Pizzabringdienstes, welches im Verlauf dieses Berichtes noch öfter aufgegriffen wird.

Kapitel 3

Konzeption

Nachdem in Kapitel 2 die allgemeinen Grundlagen erläutert worden sind, werden in diesem Kapitel Konzepte für ein Werkzeugs zur Erkennung von Modelldifferenzen erarbeitet.

3.1 Modelle

In diesem Abschnitt wird erläutert wie ein Architekten- und ein Entwicklermodell im Allgemeinen aussehen.

3.1.1 Architektenmodell

Eine Software-Architektur gibt einen Rahmen für die Entwicklung vor, indem sie die Software auf einer abstrakten Ebene beschreibt. Ein Architektenmodell besteht aus den Elementen, die sich mit der UML abbilden lassen. Ein Architekt wird zunächst aus den Anwendungsfällen Aktivitätsdiagramme erstellen und anschließend Pakete definieren. Dann werden Schnittstellen zwischen den Paketen definiert. Weiter müssen Klassen, Interfaces und Datenstrukturen definiert werden bevor das Verhalten der Software modelliert werden kann. Die UML sieht für alle diese Aktivitäten verschiedene Diagramme (siehe Abschnitt 2.4, S. 17) vor. Ein Architektenmodell ist naturgemäß eine Untermenge der zugehörigen Implementierung, da es initial erstellt wird und keine konkrete Implementierung enthält. Anschließend wird das Architektenmodell vom Entwicklerteam erweitert.

Je nach Detaillierungsgrad enthält ein Architektenmodell nach der Modellierung also Interfaces und Klassen, welche die Software dann durch Patterns [GHJV1995] und öffentliche Methoden beschreiben.

In dieser Arbeit liegt der Fokus auf den Interfaces, wie sie in der Programmiersprache Java umgesetzt sind. Diese können z.B. mit Hilfe der Klassendiagramme (siehe Abb. 2.1, S. 18) modelliert und bei geeigneten Werkzeugen auch in Quelltext überführt werden.

```
package de.janhinzmann.pizzaservice;

public interface IPizzaService {
    //protocoll
    /*
     * first call your pizzaservice.
     * then set the size of your pizza,
     * continue with setting the desired toppings,
     * set the address the pizza should be delivered to.
     * finally ask for the length of time until the pizza will be delivered.
     */

    //constants
    public static final enum Size {SMALL, MEDIUM, BIG};
    public static final enum Toppings {TOMATOES, PEPPER, MUSHROOMS};

    //methods
    public abstract void callForPizza(ICaller caller);
    public abstract void setSize(Size size);
    public abstract void setToppings(Toppings[] toppings);
    public abstract void setAddress(IAddress address);
    public abstract Date finalizeOrder();
}
```

Listing 3.1: Ein Pizzabringdienst von UML in Code transformiert

Das Interface in Listing 3.1 realisiert den Pizza-Bringdienst aus Kapitel 2.4 und stellt das passende Code-Fragment zu dem Klassendiagramm (siehe Abb. 2.1, S. 18) dar, wie er aus einem UML-Werkzeug generiert werden sollte. Zusätzlich zu den Konstanten und Methoden ist das Protokoll angegeben, welches spezifiziert, wie das Interface zu benutzen ist. Insbesondere ist bei diesem Interface die Reihenfolge wichtig, in der die Methoden benutzt werden sollen. Das Protokoll lässt sich durch Sequenzdiagramme spezifizieren, unklar ist allerdings an dieser Stelle, wie Sequenzdiagramme in Quelltext umgesetzt werden können.

Wenn das Protokoll verletzt wird, beispielsweise wenn `finalizeOrder` aufgerufen wird, bevor die Lieferadresse gesetzt ist, kann die Pizza nicht geliefert werden. Hieraus wird ersichtlich, dass die Reihenfolge der Methodenaufrufe wichtig ist, da die einzelnen Methoden zu einem inkonsistenten Zustand einer Implementierung führen können. Das beschriebene Problem entsteht dadurch, dass das Design des Interfaces `stateful` [Pugh2006] (S.45-48) ist. Hier wäre im Verlauf eines Software-Projektes denkbar, dass es in ein `stateless` Interface umgewandelt wird. Ein solches Refactoring [Fowler1999] (S.64) würde dann unweigerlich zu schwerwiegenden Modelldifferenzen führen, da Methoden des Interfaces gelöscht oder modifiziert werden. Die so entstehenden Modelldifferenzen sollten dringend kommuniziert werden. Ein gültiges Refactoring von Methoden in diesem Kontext könnte sein, dass die Methode dupliziert und anschließend modifiziert wird. Die ursprüngliche Methode sollte dann allerdings nicht gelöscht [Fowler1999] (S. 276), sondern als *deprecated* markiert werden.

Die einzelnen Methoden können natürlich noch um eine Fehlerbehandlung erweitert werden, damit berücksichtigt werden kann, dass z.B. keine Paprika (`PEPPER`) mehr vorrätig ist oder an

bestimmte Adressen nicht geliefert wird. Diese Änderungen würden auch Modelldifferenzen erzeugen und Kommunikation erfordern.

3.1.2 Entwicklermodell

Das Entwicklermodell erweitert die Architektur durch die implementierenden Anteile. Dies ist in der Regel die passende Implementierung der Interfaces. Das Entwicklerteam erzeugt für jedes Interface eine implementierende Klasse und fügt die vorgegebenen Methoden mit den implementierenden Rümpfen ein. Durch Dekomposition werden aber auch nach außen nicht sichtbare Methoden erstellt. Der Umfang des Codes, der hinzugefügt werden muss, hängt von der Menge und vom Detaillierungsgrad der Vorgabe des Architekten ab.

Während der Entwicklung kommt es vor, dass ein Entwickler das vorgegebene Modell verändert, um das von ihm verstandene Verhalten der Software zu erzielen. Jede dieser Änderungen bewirkt eine Modelldifferenz, da nun das AM nicht mehr mit dem EM übereinstimmt.

3.2 Vergleichsbasis für den Modellvergleich

Um Modelldifferenzen erkennen zu können, muss das Architektenmodell mit dem Entwicklermodell verglichen werden. Das AM ist normalerweise als UML-Modell und das EM als Code-Basis gegeben. Um die beiden Modelle vergleichen zu können, muss eine Vergleichsbasis hergestellt werden. Dazu müssen die beiden Modelle von ihren unterschiedlichen Repräsentationen durch Transformation auf eine gemeinsame Ebene gebracht werden.

Um den Aufwand minimal zu halten lohnt es sich nun, die verschiedenen möglichen Repräsentationen zu betrachten und diejenige für einen Vergleich zu identifizieren, auf die sich die beiden Modelle mit dem geringsten Aufwand transformieren lassen. Diese Repräsentation stellt dann die Ebene der Vergleichsbasis dar.

Anders als das aus Quelltexten bestehende Entwicklermodell handelt es sich beim Architektenmodell zunächst um UML-Diagramme. Diese müssen nicht per se maschinenlesbar sein – sie können auch auf einem Blatt Papier oder einer Serviette im Zug gezeichnet worden sein. Solche Diagramme müssen importiert werden, was beispielsweise durch einscannen geschehen kann. Die UML-Diagramme würden anschließend als Bilddatei im Rechner zur weiteren Verarbeitung vorliegen. Besser ist es allerdings, das Modell mit einem UML-Werkzeug direkt am Rechner zu entwickeln oder von vorhandenen, nicht digitalisierten Quellen mit einem UML-Werkzeug abzuzeichnen, sodass es weiterverarbeitet werden kann. Jedoch nicht nur der Unterschied von analogen und digitalen Modellen kann zu Problemen führen, auch die digitale Serialisierung kennt vielfältige Formate. Jedes Werkzeug wird mindestens eine proprietäre Serialisierung implementiert haben. Über dieses Mindestmaß hinaus bieten viele Werkzeuge eine Serialisierung als *XML Metadata Interchange* [XMI] an. XMI ist das standardisierte Austauschformat für Software-Entwicklungswerkzeuge der *Object Management Group* [OMG]. Den XMI-Serialisierungen ist gemein, dass sie auf XML basieren. Leider wird der Standard

von den verschiedenen UML-Werkzeugen unterschiedlich ausgelegt, sodass ein Austausch der Modelle zwischen den verschiedenen Werkzeugen nicht unbedingt gewährleistet ist. Schließlich bieten viele Werkzeuge auch die Umsetzung des UML-Modells in eine Code-Basis (CB) an, welche dann direkt mit dem Entwicklermodell verglichen werden kann.

Anhand der obigen Überlegungen lassen sich also wenigstens vier Repräsentationen identifizieren, auf die mindestens eins der beiden Modelle durch Transformation für den Modellvergleich gebracht werden muss. Die folgende Tabelle zeigt noch einmal die verschiedenen Repräsentationen und gibt einen Überblick, wie ein Modell auf die entsprechende Repräsentation gebracht werden kann.

Repräsentation	AM (UML)	EM (Code-Basis)
proprietäres Format	werkzeugunterstützt	Transformation
Bilddatei	werkzeugunterstützt	Transformation
XMI	werkzeugunterstützt	Transformation
Quelltext	teils werkzeugunterstützt	keine Aktion

Tabelle 3.1: Mögliche Repräsentationen für die Vergleichsbasis

Aus der Tabelle ist bereits jetzt ersichtlich, dass die Quelltextrepräsentation am ehesten für den Modellvergleich geeignet ist, wenn eine Umsetzung des Architektenmodells in eine Code-Basis möglich ist.

Diese vier möglichen Repräsentationen der Vergleichsbasis sollen im Folgenden diskutiert werden. Hierzu werden die Eignung, die nötigen Schritte, Vor- und Nachteile der einzelnen Repräsentation gegeneinander abgewogen und schließlich eine Empfehlung abgegeben.

3.2.1 Proprietäres Format

Auf der Ebene der proprietären Formate ist nicht abzusehen, ob es möglich ist, eine Vergleichsbasis herzustellen. Diese Repräsentation ist sicherlich mit hohem Risiko verbunden und soll von vornherein ausgeschlossen werden.

3.2.2 Bilddatei

Auf der Ebene der Bilddatei kann die Topologie [LiedtkeE1989] der UML-Diagramme untersucht werden, um Beziehungen zwischen den Klassen und Interfaces zu erkennen. Sollen die Modelle des Architekten und des Entwicklers auf dieser Ebene miteinander verglichen werden, muss der Quelltext des Entwicklers in ein UML-Diagramm und anschließend in ein geeignetes Bildformat transformiert werden. Das Architektenmodell liegt bereits als UML-Diagramm vor und muss als Bilddatei im gleichen Format gespeichert werden. Die Bildanalyse ist mit erheblichem Aufwand verbunden, da unklar ist, wie beispielsweise Text erkannt werden kann, welcher in Rastergrafiken nicht gespeichert ist. Deshalb wird diese Repräsentation für den Modellvergleich ebenfalls ausgeschlossen und nicht näher betrachtet.

3.2.3 XMI

Auf der Ebene von XMI können Modelldifferenzen untersucht werden, wenn die XMI-Beschreibungen von Entwickler- und Architektenmodell der gleichen Interpretation des XMI-Formats vorliegen. Ist dies nicht der Fall, kann durch Transformationen, vorzugsweise mit Hilfe der *Extensible Stylesheet Language Family* [XSL], eine gemeinsame Interpretation des XMI-Formats hergestellt werden. Im Fall des Architektenmodells wird diese Repräsentation durch die meisten UML-Werkzeuge unterstützt. Wird ein UML-Werkzeug eingesetzt, welches *Re-Engineering* und XMI-Export unterstützt, kann das Entwicklermodell auf diese Weise transformiert und so dem Modellvergleich zugänglich gemacht werden. Anderenfalls muss der Quelltext geparst und in XMI serialisiert werden.

Diese Repräsentation ist werkzeugabhängig, da die verschiedenen Werkzeuge eigene Interpretationen des XMI-Standards verwenden. Prinzipiell ist diese Repräsentation für den Modellvergleich geeignet; insbesondere die Probleme der verschiedenen XMI-Formate können durch XSL-Transformationen gelöst werden. Es muss allerdings für jede XMI-Interpretation ein Transformations-Stylesheet entwickelt werden. Diese Entwicklung birgt Risiken. Einfache Mappings zwischen einer Interpretation und einer anderen stellen dabei nicht das Problem dar, komplexe Logik allerdings sollte der Übersichtlichkeit halber in einer Programmiersprache abgebildet werden.

3.2.4 Quelltext

Auf der Ebene des Quelltextes können die Modelldifferenzen untersucht werden und lediglich das Architektenmodell muss dafür vorverarbeitet werden. Viele UML-Werkzeuge leisten bereits die Transformation vom UML-Diagramm zum Quelltext. Ist dies nicht der Fall, liegt aber eine Serialisierung im XML-Format (z.B. XMI) vor, so kann durch eine XSL-Transformation von dem XML-Format in die jeweilige Zielsprache Quelltext erzeugt und somit die Vergleichsbasis hergestellt werden.

Die Quelltextebene scheint die vielversprechendste Repräsentation zu sein, da eine Programmiersprache den wenigsten Raum für Interpretationen lässt. Die Anforderungen an eingesetzte Werkzeuge beschränken sich auf die Code-Generierung, Re-Engineering wird nicht benötigt.

3.2.5 Fazit

Das Architektenmodell wird in UML erstellt und gespeichert. Je nach dem welches UML-Werkzeug dabei verwendet worden ist, liegt es dann in einer der unterschiedlichen Serialisierungen vor. Die bleibende Gemeinsamkeit der verschiedenen XMI-Formate ist XML, sodass werkzeugabhängige XSL-Transformationen entwickelt werden können, um die Modelle auf ein einheitliches Format für den Modellvergleich auf dieser Ebene zu bringen. Viele Werkzeuge können die erstellten UML-Diagramme durch eine interne Transformation in Quelltext einer Zielsprache bringen. So steht das Architektenmodell, wie auch das Entwicklermodell für den Modellvergleich auf der gleichen Ebene zur Verfügung. Ist eine Serialisierung im Quelltext-Format nicht gegeben, so kann wenigstens das XML-Format in Quelltext transformiert wer-

den. Ist dies auch nicht der Fall, so muss ein Adapter entwickelt werden, der die proprietäre Serialisierung dieses Werkzeugs in Quelltext übersetzt.

Das Architektenmodell muss also in allen betrachteten Fällen verarbeitet werden. Die Verantwortung dafür kann aber an das eingesetzte Werkzeug delegiert werden. Für ein sinnvolles Arbeiten entsteht hier also eine Mindestanforderung an ein eingesetztes UML-Werkzeug:

> Das eingesetzte UML-Werkzeug muss aus den Diagrammen Quelltext oder XMI generieren können. (3.1)

Das Entwicklermodell muss verarbeitet werden, wenn auf der Ebene von XMI (Tabelle 3.1) verglichen werden soll. In diesem Fall muss der Quelltext geparst und in das entsprechende Format serialisiert werden. Bei einem Vergleich auf der Quelltextebene, entfällt eine Vorverarbeitung des Entwicklermodells.

Die solideste Basis ist also der Quelltext, weil darin die Gesamtheit der Informationen enthalten ist, wie sie später auch zur Laufzeit benötigt werden. Die Syntax von Quelltexten lässt sich durch Grammatiken eindeutig beschreiben und es existieren Parser-Systeme zur Verarbeitung. Für Java beispielsweise existiert eine BNF (siehe Abschnitt 2.2, S. 14), welche die kontextfreie Grammatik der Sprache beschreibt und eine gute Ausgangsbasis für die Entwicklung eines Parsers bildet. Da es zu viele unterschiedliche XMI-Repräsentationen von UML gibt und es letztendlich doch nur um den Quelltext geht, scheint damit die geeignete Vergleichsbasis gefunden zu sein. Da Quelltext eindeutig ist, lassen sich auf dieser Basis die nötigen Vergleiche anstellen und es folgt die Feststellung:

> Die Basis für die Verarbeitung der Modelle zum Zweck der Erkennung von Modelldifferenzen ist Quelltext. (3.2)

Ein Werkzeug zur Erkennung von Modelldifferenzen muss also Quelltext-Basen verarbeiten, um Aussagen über Differenzen zwischen einem vorgegebenen Architektenmodell und einem zu überprüfenden Entwicklermodell machen zu können. Ein erster Ansatz zum Vergleichen zweier Quelltexte könnte nach dem in [Myers1986] beschriebenen Algorithmus durchgeführt werden. Dieser ist auf Unix-Betriebsprogrammen als `diff`-Werkzeug implementiert. Ebenso stellen auch Eclipse [eclipse], CVS [CVS] oder Subversion [Subversion] Vergleiche zweier Quelltext-Dateien zur Verfügung. Diese Werkzeuge vergleichen alle zeilenweise, was ein differenziertes Erkennen von Modelldifferenzen erschwert. Sind beispielsweise zwei Methoden einer Klasse nur verschoben worden sein, ändert dies die Semantik des Programms nicht. Es tritt also keine relevante Modelldifferenz auf. Von den oben genannten Werkzeugen wird ein Verschieben allerdings als Differenz erkannt.

Ein weiteres Problem dieser Herangehensweise ist es, dass immer zwei Dateien miteinander verglichen werden. Heutige Software besteht in der Regel aus mehreren Dateien mit semantischen Beziehungen untereinander. Diese Beziehungen können durch den Vergleich zweier einzelner Dateien nicht erfasst werden.

Wenn also nicht zeilenweise und nicht dateiweise verglichen werden kann, muss die Software im Ganzen erkannt werden. Dies kann durch Parsen und Erstellen eines Abstrakten Syntax-

baumes (AST) erfolgen. Hieraus ergibt sich die Feststellung:

Die Code-Basis muss geparst werden und deshalb kompilierbar sein. (3.3)

Der durch das Parsen erstellte AST hängt ohne weitere Abstraktion von der Sprache ab, in der das Architekten- bzw. das Entwicklermodell vorliegen.

Um diese Sprachabhängigkeiten aufzulösen ist es sinnvoll, eine Abstraktionsebene in Form einer Metarepräsentation (siehe auch Abschnitt 3.3) zu wählen, welche dann als Schnittstelle zwischen der Form der Modelle und den Vergleichen dient. Es können dann allgemeingültige Regeln Anwendung finden, die nicht geändert werden müssen, wenn beispielsweise ein Modell nicht in Java sondern in C++ gegeben ist. Ferner können das AM und das EM auch in verschiedenen Sprachen vorliegen. Ist beispielsweise ein Web-Service mit der Technologie [AXIS] realisiert, liegt ein Architektenmodell als WSDL-Beschreibungen [WSDL] vor. Das Entwicklermodell ist eine Java-Implementierung und so kann trotz verschiedener Sprachen ein Modellvergleich durchgeführt werden. Es folgt also:

Die Vergleichsbasis für die Modellvergleiche ist die Metarepräsentation. (3.4)

Nachdem in den obigen Abschnitten erläutert worden ist, warum es sinnvoll ist einer Metarepräsentation als Vergleichsbasis zum Erkennen von Modelldifferenzen zu wählen, wird diese nun in dem folgenden Abschnitt vorgestellt.

3.3 Metarepräsentation

In diesem Abschnitt soll die Metarepräsentation beschrieben werden. Es handelt sich hierbei um eine interne Repräsentation, die im System eingesetzt wird, um eine Vergleichsbasis zur Erkennung von Modelldifferenzen bereitzustellen.

Die Metarepräsentation muss zum einen die Modelle des Entwicklers und des Architekten abbilden und zum anderen alle Abfragen bzgl. der Modelldifferenzen ermöglichen. Außerdem stellt sie eine Abstraktion von der Programmiersprache dar und dient als Schnittstelle zwischen der Vorverarbeitung der Modelle und der anschließenden Evaluation. Ferner wird es so möglich, Modelle miteinander zu vergleichen, die in verschiedenen Sprachen vorliegen. Das momentan vorherrschende Paradigma moderner Programmierung ist die Objektorientiertheit und deshalb ist die Metarepräsentation so aufgebaut, dass sie die Konzepte objektorientierter Sprachen abbilden kann. Somit ist es möglich, die Modelle der verschiedenen objektorientierten Sprachen in der Metarepräsentation abzubilden und miteinander zu vergleichen.

Hier wäre zum Beispiel denkbar, dass eine Schnittstellenspezifikation, also ein Architektenmodell, in einer Schnittstellenbeschreibungssprache (WSDL, CORBA-IDL o.ä.) vorliegt und das Entwicklermodell eine Java-Implementierung ist. Durch die Wahl einer Metarepräsentation ist es möglich, die beiden Modelle auf dieser abstrakteren Ebene miteinander zu vergleichen, ohne die Auswertungskomponente modifizieren zu müssen.

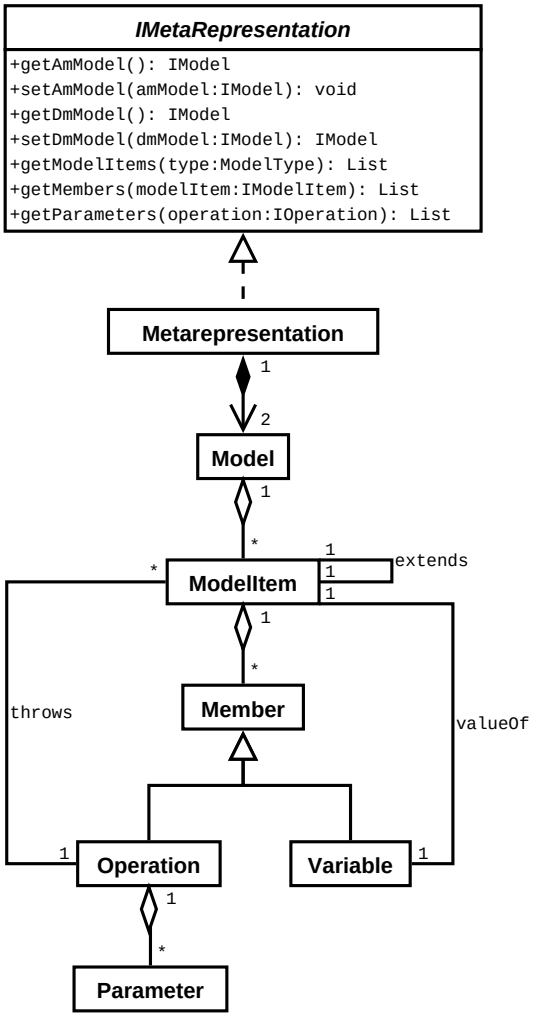

Abbildung 3.1: Metarepräsentation als UML-Diagramm

Die `Metarepresentation` (siehe Abb. 3.1, S. 27) implementiert das Interface *IMetarepresentation*, welches die Zugriffe auf die einzelnen Elemente der Modelle sicherstellt. Ein `Metarepresentation`-Objekt ist mit genau zwei Modellen assoziiert, dem AM und dem EM. Ein Modell besitzt dann wiederum eine Liste von Elementen des Typs `ModelItem`, die ihrerseits eine Liste mit `Member`-Objekten haben. Ein `Member`-Objekt kann durch eine `Operation` oder eine `Variable` repräsentiert werden. Dies ist durch die Vererbungshierarchie modelliert. Eine Operation hat schließlich noch eine Liste mit `Parameter`-Objekten.

Weitere Attribute der Metarepräsentation sind der Übersichtlichkeit halber nicht in Abbildung 3.1 (S. 27) dargestellt. Jedes Element der Metarepräsentation hat einen Namen und einen Namensraum. Diese beiden Attribute zusammen ergeben seinen Identifizierer. In Java gibt es das Konzept des *fully qualified classnames* (§6.7 [JSL]). Durch die Identifizierung der Modellelemente mit einem Namensraum und einem Namen ergeben sich in der Metarepräsentation auch *fully qualified operationname*, *fully qualified parametername* und *fully qualified variablename*. So besitzt zum Beispiel die Methode `setSize` aus dem

Beispiel des Pizzabringdienstes (siehe Abschnitt 2.4, S. 17) den vollqualifizierenden Namen `de.janhinzmann.pizzaservice.IPizzaService.setSize` und ihr Parameter `size` wird durch `de.janhinzmann.pizzaservice.IPizzaService.setSize.size` identifiziert.

Die Metarepräsentation hält ferner verschiedene Methoden bereit, mit denen sich Fragen an die Modelle stellen lassen (siehe Abb. 3.1, S. 27). So erhält man zum Beispiel mit der Methode `getModelItems` alle Modellelemente zu einem Modelltyp. Die Metarepräsentation kennt die beiden Modelltypen `AM` für Architektenmodell und `DM` für Entwickler(Developer)-modell.

Nachdem die in dieser Arbeit entwickelte Metarepräsentation vorgestellt worden ist, sollen nun die Elemente von Interfaces am Beispiel der Sprache Java untersucht werden, um variable Anteile zu identifizieren, welche potentielle Ausgangspunkte für Modelldifferenzen darstellen.

3.4 Interfaces in Java

In diesem Abschnitt soll untersucht werden, wie sich Interfaces in der Sprache Java zusammensetzen. Dazu sollen alle Bestandteile der Interface-Definition identifiziert werden. Die variablen Elemente werden anschließend weiter untersucht, da sie potentielle Quellen für Modelldifferenzen darstellen.

3.4.1 Aufbau von Java-Interfaces

In Java-Interfaces werden Konstanten und Methoden deklariert, welche die Struktur von Klassen beschreiben. Hierbei unterscheidet sich die Interface-Deklaration von einer Klassendeklaration im Wesentlichen dadurch, dass Methodendeklarationen in Interfaces keine Implementierung besitzen, sondern nur aus abstrakten Methodensignaturen bestehen.

```
[visibility] interface InterfaceName [extends other interfaces] {
        constant declarations
        abstract method declarations
}
```

Listing 3.2: Elemente einer Interface Deklaration

Listing 3.2 zeigt eine verkürzte Interfacedefinition. Ein Interface ist nach §6.6.1 der Java-Sprachspezifikation [JSL] implizit *package scoped*, wenn das Schlüsselwort `public` nicht explizit angegeben ist. Konstanten innerhalb eines Interfaces sind dann implizit `public static final` und Methoden `public abstract` deklariert.

In Kapitel 18 der Java-Sprachspezifikation [JSL] wird die Grammatik für die Sprache dargestellt. Für die Untersuchungen in dieser Arbeit reicht der Teil der Grammatik, der die Interfaces beschreibt. Die irrelevanten Regeln wurden deshalb in Listing 3.3 weggelassen. Bei der Notation der Regeln wird vom Stil der Java-Sprachspezifikation abgewichen und der in Kapitel 2.2 vorgestellte BNF-Stil verwendet, da dieser übersichtlicher erscheint.

```
1  ...
2
3  ClassOrInterfaceDeclaration ::=
4      { Modifier }( ClassDeclaration | InterfaceDeclaration )
5
6  Modifier ::=
7      Annotation | "public" | "protected" | "private" | "static"
8      | "abstract" | "final" | "native" | "synchronized" | "transient"
9      | "volatile" | "strictfp"
10
11
12 InterfaceDeclaration ::= NormalInterfaceDeclaration
13     | AnnotationTypeDeclaration
14
15 NormalInterfaceDeclaration ::=
16     "interface" Identifier [ TypeParameters ][ "extends" TypeList ]
17      InterfaceBody
18
19 InterfaceBody ::= "{" { InterfaceBodyDeclaration } "}"
20
21 InterfaceBodyDeclaration ::= { Modifier } InterfaceMemberDecl
22
23 InterfaceMemberDecl ::= InterfaceMethodOrFieldDecl
24     | InterfaceGenericMethodDecl
25     | "void" Identifier VoidInterfaceMethodDeclaratorRest
26     | InterfaceDeclaration
27     | ClassDeclaration
28
29 ...
```

Listing 3.3: Interface-bezogene Regeln aus der JLS in BNF

Die angegebene Grammatik ist stark gekürzt und es sei an dieser Stelle auf die vollständige Fassung in Kapitel 18 der *Java Language Specification Third Edition* [JSL] verwiesen.

Prinzipiell lässt sich die gesamte Interface-Definition in eine Signatur (Zeile 15) und einen Rumpf (Zeile 19) aufteilen. Diese beiden Hauptbestandteile sollen nun im Weiteren erläutert werden.

3.4.2 Interface-Signatur

Eine Interface-Signatur besteht aus einer Liste von Modifiern, mit der Kardinalität $0..n$. Es folgt das Schlüsselwort `interface` gefolgt von genau einem Identifier. Danach kann durch das Schlüsselwort `extends` eine `TypeList`, als eine Liste von Interface-Namen mit der Kardinalität $1..n$ eingeleitet werden. Dabei werden die einzelnen Interface-Namen durch Kommata getrennt.

Modifier

Die Grammatik aus der Sprachspezifikation schränkt die Benutzung der Modifier nicht ein. Nach [JavaTutorial] wird allerdings klar, dass nur die Modifier `public` und `abstract` Verwendung finden können. Implizit ist ein Interface immer `abstract`, sodass die Angabe dieses Schlüsselwortes nur eine Frage des Stils ist. Der Modifier `public` erweitert die Sichtbarkeit des Interfaces über das Paket hinaus, da implizit `protected` angewandt wird.

Identifier

Identifier stellen den Namen eines Interfaces dar. Sie beginnen mit einem Buchstaben, dem Zeichen `$` oder _. Es folgen dann $0..n$ weitere Zeichen aus der Menge $\{a..z, 0..9, \$, \_, \texttt{unicode character over } 00C0\}$. Die Identifier stellen einen eindeutigen Namen für das Interface innerhelb seines Paketes dar. Zur Identifizierung des Interfaces über das Paket hinaus werden die *fully qualified names* §6.7 [JSL] verwendet, die sich aus dem Paketnamen und dem Identifizierer des Interfaces zusammensetzen.

TypeList

Ein Interface kann andere Interfaces erweitern bzw. beerben. Hierzu werden ähnlich wie bei den Klassendeklarationen durch das Schlüsselwort `extends` die erweiterten Interfaces angegeben. Anders als bei Klassen kann ein Interface mehrere Interfaces erweitern. Es kann also eine Liste von *Superinterfaces* nach dem Schlüsselwort `extends` angegeben werden, wobei die einzelnen Interface-Namen durch ein Komma getrennt werden. Ein Superinterface ist hierbei das Interface, welches erweitert wird, also in der Vererbungshierarchie über dem erweiternden Interface steht.

Nachdem im obigen Abschnitt die Interface-Signatur dargestellt worden ist, soll nun der Rumpf eines Interfaces erläutert werden.

3.4.3 Interface-Rumpf

Der Interface-Rumpf wird durch die Regel `InterfaceBody` beschrieben. Er wird durch eine geschweifte Klammer eingeleitet und es folgt eine Aufzählung von 0 bis n sogenannten `InterfaceBodyDeclarations`. Diese Aufzählung endet mit der korrespondierenden schließenden geschweiften Klammer.

Eine `InterfaceBodyDeclaration` besteht dann aus optionalen Modifiern gefolgt von einer `InterfaceMemberDecl`-Deklaration.

Durch die Regel `InterfaceMemberDecl` werden unter anderem Methodensignaturen und Konstanten deklariert. Hierbei sind alle Methoden implizit `public abstract` und alle Konstanten `public static final` deklariert. Da die Methoden `abstract` deklariert sind, werden bei den

Methoden in Interfaces keine Rümpfe angegeben. Dies bleibt den implementierenden Klassen vorbehalten.

3.5 Genese von Modelldifferenzen

Im vorangegangenen Abschnitt wurden die Bestandteile von Interfaces vorgestellt. Ein Interface besteht aus einer Signatur und einem Rumpf, wobei sich der Rumpf weiter in Konstanten und Methoden aufteilen lässt. Natürlich sind die Schlüsselwörter invariant, die restlichen Bestandteile bilden jedoch die Grundlage für Modelldifferenzen.

3.5.1 Was kann geändert werden?

Zunächst soll die Signatur eines Interfaces betrachtet werden:

- Die `modifier`-Liste des Interfaces kann modifiziert werden. Ein Interface kann *package scoped* sein, was der voreingestellte Wert ist. Die Sichtbarkeit des Interfaces kann durch den Modifier `public` erhöht werden, sodass es in allen Paketen, aber auch außerhalb der Software benutzt werden kann. Wenn dies der Fall ist, dann gilt das Interface als veröffentlicht [Fowler1999] (S. 64).
- Der Identifier des Interfaces kann sich ändern, d.h. ein Interface wurde umbenannt.
- Die `extends`-Liste des Interfaces kann sich ändern. Es können weitere Superinterfaces hinzugefügt oder gelöscht werden.

Im Rumpf eines Interfaces können sich Methoden und Konstanten ändern:

- Eine Methode kann gelöscht werden.
- Eine Methode kann hinzugefügt werden.
- Eine Methodendeklaration kann geändert werden:
  - Die `modifier`-Liste kann vernachlässigt werden, da alle Methoden implizit `public abstract` deklariert sind.
  - Der Rückgabewert einer Methode kann sich ändern (`void | <ReturnType>`).
  - Der Identifizierer einer Methode kann sich ändern. Dies kommt dem Löschen der alten und Hinzufügen einer neuen Methode gleich.
  - Die Parameterliste einer Methode kann sich ändern. Es können Parameter hinzukommen oder gelöscht werden. Wenn sich der Typ oder der Name eines Parameters ändert, so kommt dies dem Löschen eines alten und Hinzufügen eines neuen Parameters gleich.
- Eine Konstante kann gelöscht werden.

- Eine Konstante kann hinzugefügt werden.
- Eine Konstante kann geändert werden:
  - Die `modifier`-Liste kann vernachlässigt werden, da alle Konstanten implizit `public static final` deklariert sind.
  - Der Typ einer Konstanten kann sich ändern.
  - Der Identifizierer einer Konstanten kann sich ändern. Dies kommt dem Löschen der alten und Hinzufügen einer neuen Konstanten gleich.
  - Der zugewiesene Wert einer Konstanten kann sich ändern.

Nachdem nun untersucht worden ist, was geändert werden kann, soll im Folgenden das Wie untersucht werden.

3.5.2 Wie entstehen Modelldifferenzen?

In diesem Abschnitt soll erläutert werden, wie es zu Abweichungen des Entwicklermodells vom Architektenmodell kommt.

Modelldifferenzen entstehen immer dann, wenn entweder das Architekten- oder das Entwicklermodell verändert wird, ohne die Änderungen ebenfalls in dem jeweils anderen Modell durchzuführen. Hierbei können Änderungen entweder das Hinzufügen, Löschen oder Modifizieren von Modellelementen bedeuten. Das Modifizieren der Identifizierer von Modellelementen wird in dieser Arbeit als zusammengesetzte Operation aus Löschen und Hinzufügen betrachtet, ebenso das Verschieben.

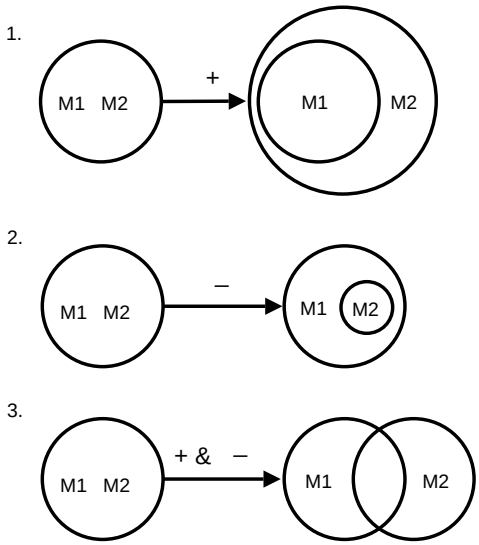

Abbildung 3.2: Schnittmengen bei Veränderungen am Entwicklermodell

In Abbildung 3.2 sind die drei Fälle dargestellt. Auf der linken Seite ist jeweils die Ausgangssituation zu sehen, in der ein Architekt ein AM (z.B. M1) erstellt und an die Entwickler

übergeben hat. Diese haben es übernommen und das AM stimmt mit dem EM (z.B. M2) überein. Auf der rechten Seite ist die Situation nach Veränderungen durch die Entwickler bzw. durch den Architekten dargestellt und der Pfeil deutet die jeweilige Operation (Hinzufügen (+)), Löschen (–), Modifizieren (+ & –) an, die ausgeführt worden ist. Im Folgenden sind die drei Fälle beschrieben:

1. Werden Modellelemente zu M_2 hinzugefügt, so wächst es und M_1 bildet M_2 nicht mehr vollständig ab. Dies ist der Fall, wenn die Entwickler Code hinzufügen, sodass das EM wächst und das AM zu einer Teilmenge des Entwicklermodells wird. Dies ist die gewünschte Vorgehensweise. Im Bild wird dies klar, wenn M_1 das Architektenmodell und M_2 das Entwicklermodell verkörpert. Andersherum kann es auch sein, dass der Architekt Modellelemente hinzugefügt, wenn das Design erweitert wird. Dann verkörpert M_1 das EM und M_2 das AM.

2. Werden Modellelemente aus M_2 gelöscht, wird M_2 zu einer Teilmenge von M_1. Im Fall M_2=EM wurden also Modellelemente aus dem Entwicklermodell gelöscht und das EM bildet das AM nicht mehr vollständig ab. Dies ist unerwünscht, da Teile der Architektur im Entwicklermodell verloren gehen. Im Fall M_2=AM wurden Modellelemente aus dem Architektenmodell gelöscht und das Entwicklermodell beinhaltet nicht benötigte Anteile, die ebenfalls gelöscht werden müssen, um beide Modelle konsistent zu halten.

3. Werden sowohl Modellelemente hinzugefügt, als auch gelöscht, so entsteht eine Schnittmenge zwischen M_1 und M_2. Je öfter diese Operationen ausgeführt werden, desto weiter entfernen sich die beiden Modelle voneinander, bis die beiden Mengen disjunkt sind. Dann wurde die vorgegebene Architektur durch das EM ersetzt, da es schließlich der Quelltext ist, der ausgeführt wird. Das SW-Projekt ist in Gefahr seinen Architekten zu verlieren.

Dies sind theoretische Überlegungen, die nicht die Beweggründe von Architekten und Entwicklern wiedergeben können. Wie sieht es denn nun in der Praxis aus? Klar ist, dass sich Modelldifferenzen während eines Check-Ins in das Repository manifestieren.

3.5.3 Warum wurde etwas modifiziert?

Eine E-Mail-Umfrage, in der kurz die Bestandteile, die sich ändern können (Vergl. 3.5.1) geschildert wurden und dann zu Kommentaren und Anekdoten aufforderte, ergab unter den Entwicklern im DLR (siehe Abschnitt 2.1, S. 13) folgende Reaktionen:

Entwickler 1: Ich ändere die Interfaces in dem Moment, wo die Änderungen weniger Arbeit verursachen als das Drumherum-Programmieren eines Workarounds mit den vorhandenen Interfaces. Das ist natürlich keine offizielle Aussage, sondern kommt aus der Praxis :-D Es ist doch so: wenn ich schnell was implementieren muss, ist es oft einfacher, einen Workaround zu schreiben, als eine Interface-Änderung zu „beantragen“. Das führt dann auf Dauer zur vielzitierten „Aufweichung“ des anfänglichen Designs. Das wurde z.B. in XP berücksichtigt und anders gemacht: konstantes Re-Engineering und Design-Änderungen sind da Alltag!!!

Entwickler 2: ca. 90% (gefühlter Wert) der wesentlichen Interface-Änderungen entstanden bei SiLEST aus 3 Gründen:

1. während der Entwicklung merkt man, dass eine Information fehlt, also wurde das Interface um eine Methode erweitert.
2. oder man merkt, dass eine Funktionalität doch nicht benötigt wurde oder schon durch eine andere Methode abgedeckt ist und die entsprechende Methode wurde entfernt.
3. Umbenennung einer Methode oder des Interfaces, weil sich im Laufe der Zeit einfach ein besserer Name fand oder die Funktionalität sich leicht geändert hat.

Entwickler 3: Interfacesignatur: modifier, name, extends-Liste

Modifier und Name in der Interfacesignatur werden höchstens geändert, wenn ich die Architektur noch in UML modelliere. Während der Implementierung sind Änderungen daran böse.

Die extends-Liste ist normalerweise nach Festlegung der Architektur fest. Wenn dann doch Änderungen auftreten, dann weil die zu Grunde liegende Basisklasse noch nicht angegangen wurde (Inkementelles Abarbeiten der einzelnen Aspekte). Alles andere sind nachträgliche Architekturänderungen die nach ECSS nicht so einfach möglich sind, da man ansonsten den gesamten Architekturprozess nocheinmal durchlaufen muss.

Interfacerumpf: Methoden: Modifier, Rückgabewert, Name, Parameter

Die Sachen sind wiedermal ziemlich böse und sollten bereits nach der UML-Modellierung klar sein. Wenn nicht, dann sollte man das nächste Mal vorher etwas nachdenken. Was durchaus passiert, dass noch Methoden mit private hinzukommen, aber das gehört ja nicht zum öffentlichen Interface... Wenn Methoden hinzukommen, dann höchsten durch das inkrementelle Abarbeiten der einzelnen Aspekte.

Änderungen an den Parametern kommen leider öfters vor, wenn man sich nicht richtig Gedanken über den Prozess zur Nutzung des Interfaces gemacht hat. Irgendwelche Informationen werden dann benötigt, die noch nicht vorhanden sind. Bei Bird (ein weiteres DLR-Projekt A.d.A) war das z.B. sehr stark in den Setup-Methoden notwendig, da ich mit jeder Implementierung eines neuen Systemteils dort die Parameter um die entsprechende Klasse ergänzen musste. Das lag aber an dem vorgegebenen Setup-Prozess. Ich würde an der Stelle eigentlich einen Call-Back vorsehen, so dass nur noch eine Instanz zu einem zentralen Kommunikationsknoten als Parameter vorgesehen wird, über die die Pointer zu den einzelnen Threads und Systemkomponenten ausgetauscht werden.

Ansonsten ist das natürlich immer diese leidige Geschichte, dass Informationen einfach nur kleckerweise hereinkommen. Nach der Festlegung der Architektur steht man dann bei solchen Änderungen schnell im Wald (Kann ich diesen Pointer fällen?).

Konstanten: Modifier, Typ, Name, Wert

Kommt auf den Verwendungszweck der Konstanten an. Handelt es sich um Konstanten, welche Zustände der Software beschreiben (Besonders bei Konstanten einer Aufzählung), dann werden die bereits in den Anforderungsdokumenten definiert. Diese Konstanten werden schließlich in mehr als nur einem Softwaresystem verwendet und sind meist Teil der Housekeeping-Daten.

> Sind es jedoch Konstanten, die für die Berechnung irgendwelcher Werte intern verwendet werden, so werden die erst nach und nach vergeben. Eine Änderung von Namen gibt es nie, da lauern einfach zu viele Bäume und im Wald habe ich Angst. :lol:
>
> Wobei ich jetzt aber zu Konstanten sagen muss, da haben wir bei Bird zwei Arten von Konstanten verwendet. Zum einen wirkliche Konstanten, an denen nicht gedreht wird, zum anderen Konstanten, welche programmiertechnisch Variablen sind, aber wie Konstanten benutzt werden. Die Dinger dürfen nur in genau einem Kommandohandler schreibend angefasst werden, nämlich der zum Kommando SBC_ACS_SET_CONS. Das Kommando ist hochkritisch und darf nur mit Bestätigung versandt werden. Es hat aber auch den Satelliten bei einigen Notfällen gerettet, da mit ihm Berechnungen an veränderte Bedingungen angepasst werden können ohne an der Software zu drehen...

3.5.4 Fazit

Änderungen entstehen also aus Zeitdruck und Sperrigkeit des Software-Entwicklungsprozesses. Weiterhin entstehen Modelldifferenzen bei zu geringem Planungshorizont bzw. durch den evolutionären Charakter der Software-Entwicklung. Dies gilt besonders bei Forschungsprojekten.

Weiterhin ergibt sich aus der Umfrage und der weiter oben erarbeiteten Theorie die Motivation für die folgenden Regeln bezüglich Modelldifferenzen.

3.6 Regeln für den Umgang mit Veränderungen

In dieser Arbeit lag der Schwerpunkt zunächst nur auf den Interfaces in der Programmiersprache Java, konnte später zu einem gewissen Teil sogar auf `ModelItems` (s. Abschnitt 3.3) und somit auch auf die Klassen ausgeweitet werden. Die im Folgenden erarbeiteten Regeln beziehen sich zunächst auf die Interfaces und im Abschnitt „Weiterführende Regeln“ werden ein paar Ideen zu Regeln bzgl. Klassen und Interfaces beschrieben.

Eine Regel ist hierbei eine Zuordnung einer überprüfbaren Aussage über Elemente aus einem Modell zu einem Schweregrad.

3.6.1 Regeln bzgl. der Menge der Interfaces

1. *Ein Interface darf nicht gelöscht werden.* Das Löschen eines Interfaces ist ein schwerwiegender Eingriff in die Software und es kann hierdurch zu ungeahnten Seiteneffekten kommen. Das Problem liegt hierbei nicht unbedingt innerhalb der Software selbst, sondern in Software Dritter, die auf der nun gelöschten Schnittstelle aufbauen und nicht mehr funktionieren können. Aber auch innerhalb einer Software ist das Löschen eines Interfaces ein schwerwiegender Verstoß gegen die Architektur. Es kann also differenziert werden, ob es sich um eine öffentliche oder um eine interne Schnittstelle handelt.

2. *Ein Interface darf hinzugefügt werden.* Das Hinzufügen von Interfaces ist nicht so schwerwiegend wie das Löschen, jedoch kann es hierdurch zu einem Aufweichen der Architektur (vergl. 3.5.3 Entwickler 1) kommen. Gerade, wenn das neue Interface beispielsweise Features von vorhandenen Interfaces dupliziert.

3.6.2 Regeln für die Interfacesignatur

1. *Die Sichtbarkeit eines Interfaces darf nicht reduziert werden.* Wenn ein Interface öffentlich ist, darf diese Sichtbarkeit aus den gleichen Gründen wie in 3.6.1 Regel 1. nicht reduziert werden, da dies aus Sicht der Klienten dem Löschen gleich käme.

2. *Die Sichtbarkeit eines Interfaces darf erweitert werden.* Wird die Sichtbarkeit eines Interfaces erweitert (`protected` → `public`), so ist dies eine Veröffentlichung dieses Interfaces. Dies darf dann nicht mehr rückgängig gemacht werden (3.6.1 Regel 1.).

3. *Ein Interface darf nicht umbenannt werden.* Das Umbenennen eines Interfaces kommt in dieser Arbeit dem Löschen des alten und Hinzufügen eines neuen Interfaces gleich, da Interfaces über den vollqualifizierten Klassennamen identifiziert werden (siehe Abschnitt 3.3, S. 26). Zwar gibt es Bestrebungen [XingS2005], durch Ähnlichkeiten Umbenennungen zu erkennen, aber wieder ist dies kritisch bei bereits veröffentlichten Interfaces (3.6.1 Regel 1.).

4. *Die Menge der Interfaces in der `extends`-Liste darf nicht verkleinert werden.* Wird ein Interface aus der `extends`-Liste eines Interfaces gelöscht, so verliert das Interface an Funktionalität, was wiederum zu dem in 3.6.1 Regel 1. beschriebenen Problem führen kann.

5. *Die Menge der Interfaces in der `extends`-Liste darf erweitert werden.* Auch hier ist allerdings die eventuelle Aufweichung der Architektur zu bedenken und es bedarf der Kommunikation zwischen dem Architekten und den Entwicklern (3.6.1 Regel 2.).

3.6.3 Regeln für den Interfacerumpf

1. *Es dürfen keine Member aus einem Interface gelöscht werden.* Das Löschen von Methoden oder Konstanten (Member) verletzt die Architektur und führt dazu, dass die Software nicht wie gewünscht funktionieren kann.

2. *Es dürfen Member zu einem Interface hinzugefügt werden.* Das Hinzufügen von Methoden oder Konstanten erweitert die Architektur und bedarf der Kommunikation der Entwickler mit dem Architekten.

3. *Es dürfen keine Member eines Interfaces modifiziert werden.* Da das Modifizieren eine aus Löschen und Hinzufügen zusammengesetzte Operation darstellt und das Löschen verboten ist, ist auch das Modifizieren nicht erlaubt.

4. *Das Protokoll eines Interfaces darf nur nach Absprache und Änderung im Architektenmodell verändert werden.* Das Protokoll eines Interfaces setzt sich aus den definierten

Methoden und insbesondere der Reihenfolge der Aufrufe dieser zusammen. Wird hier etwas verändert **muss** zwischen allen Beteiligten Kommunikation erfolgen und insbesondere den externen Projektpartnern darf das neue Protokoll nicht vorenthalten werden. Ferner muss das AM angepasst werden. Das Protokoll eines Interfaces ist allerdings in der Regel nicht im Interface selbst deklariert, sondern in einem externen Artefakt (z.B. in einem Sequenzdiagramm).

Anmerkung: Diese Regel ist wohl zu generell – außerdem ist das Protokoll in Hinsicht auf die Reihenfolge der Aufrufe der einzelnen Methoden höchstens in Sequenzdiagrammen spezifiziert, welche hier nicht betrachtet werden. An dieser Stelle ist es sinnvoll das folgende HAPE-Prinzip (Accenture TS Custom Application Design School) beim Design eines Protokolls zu beherzigen:

- Hiding Of Information, verbergen von Implementierungsdetails.
- Atomicity, jede Operation überführt von einem validen Zustand in einen anderen validen Zustand.
- Parametrized Constructors, um Atomarität zu gewähleisten.
- Exceptions, Operationen, welche die Atomarität verletzen.

3.6.4 Weiterführende Regeln

In den obigen Regeln wurden immer nur Modelldifferenzen zwischen den Interfaces im AM und EM betrachtet. Nimmt man die implementierenden Klassen bzw. überhaupt Klassen hinzu, so ergeben sich neue Fragestellungen:

1. *Das Hinzufügen von öffentlichen Methoden zu einer implementierenden Klasse weicht die Architektur auf.* Wird beispielsweise zu einer Klasse `Joeys`, die das Interface `IPizzaService` (siehe Abb. 2.1 bzw. Abb. 3.1) implementiert, eine öffentliche Methode hinzugefügt, die nicht auch in `IPizzaService` definiert worden ist, würde dies zu einem Aufweichen der Architektur führen. Allerdings muss bei der Umsetzung beachtet werden, dass Klassen auch öffentliche Methoden von ihrer Superklasse erben. Diese geerbten öffentlichen Methoden werden normalerweise aber nicht in Interfaces deklariert, sodass sich hier fälschlicherweise Modelldifferenzen ergeben würden. In Java beispielsweise erbt jede Klasse implizit von `Object` und hat so automatisch die drei öffentlichen Methoden `equals`, `hashcode` und `toString`. Überschreibt eine Klasse eine der genannten Methoden, so entstehen wie oben erwähnt fälschlicherweise Modelldifferenzen. Die so entstehenden Modelldifferenzen können jedoch durch Eintragen in eine „Schwarzen Liste“ gefiltert werden.

2. *Das Hinzufügen von private Methoden stellt eine gute Form der Dekomposition dar.* Die Entwickler werden bei der Umsetzung der Architektur nicht nur die öffentlichen Methoden, welche der Architekt vorgegeben hat implementieren, sondern auch private Hilfsmethoden hinzufügen. Die Kapselung von Teilproblemen in private Hilfsmethoden stellt einen guten Stil dar und deshalb ist eine solche Erweiterung des EMs im Allgemeinen als wenig Schwerwiegend einzustufen. Solche Modelldifferenzen sollten gerade

deswegen aufgedeckt werden, stellen sie doch eine Möglichkeit für den Architekten dar, diese im täglichen MoDi-Meeting motivierend aufzugreifen.

3. *Einhaltung der Schichten.* Das Überprüfen von Schichtzugriffsverletzungen ist ebenfalls ein interessanter Aspekt für die Betrachtung von Modelldifferenzen. Allerdings ist hier zunächst zu klären, wie Schichten in Software überhaupt definiert bzw. hinterlegt sind. In Java kann dies durch Pakete geschehen und diese Information kann dann für eine Überprüfung genutzt werden. Allerdings ist nicht spezifiziert, welche Schicht auf welcher anderen Schicht aufbaut bzw. darunter oder darüber liegt. Diese Information muss extern vorgegeben und deshalb konfiguriert werden, damit eine Regel die Einhaltung von Schichten überprüfen kann.

3.7 Zusammenfassung

In diesem Kapitel wurden das Architekten- und das Entwicklermodell vorgestellt. Anschließend wurden die einzelnen Repräsentationen der Modelle im Bezug auf ihre Vergleichbarkeit untersucht. Hierbei wurden die aufeinander aufbauenden Folgerungen erarbeitet, die schließlich in der Entscheidung mündeten, als Vergleichsbasis eine Metarepräsentation zu wählen. Diese Metarepräsentation enthält nur die erforderliche Information und abstrahiert vom Quelltext der Modelle und bildet die Elemente einer objektorientierten Sprache ab. So ist es theoretisch möglich, auch verschiedene Sprachen miteinander zu vergleichen (z.B. WSDL mit Java).

Weiterhin wurden die Bestandteile von Interfaces untersucht, um die variablen Anteile zu identifizieren. Hieraus konnten dann die Möglichkeiten für Modelldifferenzen theoretisch erarbeitet werden. Außerdem wurde eine Umfrage unter den Entwicklern des DLR beschrieben, die in einem kleinen Umfang Erfahrungen aus der Praxis darstellt. Schließlich wurden Regeln für Veränderungen erarbeitet, die entstandenen Differenzen bewerten. Hierfür bildete die erarbeitete Theorie und die Ergebnisse der Umfrage die Grundlage.

Kapitel 4

Konsequenzen für den Software-Entwicklungsprozess

In diesem Kapitel sollen die sich ergebenen Konsequenzen für den Software-Entwicklungsprozess im DLR dargestellt werden, wenn ein Werkzeug zur Erkennung von Modelldifferenzen zur Verfügung steht und in den Entwicklungsprozess integriert wird.

4.1 Allgemeine Überlegungen

In Softwareprojekten des DLR wird UML zur Modellierung des Designs oft nur am Anfang und am Ende eines Projektes benutzt. Am Anfang zur Unterstützung des Entwurfs und um sich „ein Bild zu machen“, sowie am Ende für die Dokumentation. Dies sollte aber nicht die Regel sein.

Teilweise wird auch während der Entwicklung zu bestimmten Besprechungen (z.B. Designdiskussionsrunden oder Code-Reviews) UML angefertigt. Allerdings scheint hier oft das UML-Diagramm nicht mit der tatsächlichen Code-Basis synchronisiert zu sein.

Hier kann der Einsatz von UML durch Werkzeuge verbessert werden, die das sogenannte *Roundtrip-Engineering* unterstützen. Roundtrip-Engineering bedeutet, dass das Werkzeug nicht nur Code-Basen aus den gezeichneten Diagrammen generieren kann, sondern auch in der Lage ist, Diagramme aus einer Code-Basis zu erzeugen. So kann ein Architekenmodell direkt aus einem Entwicklermodell erstellt werden. Prinzipiell ist das Re-Engineering eine gute Möglichkeit, das momentane Entwicklermodell aus einer anderen Sicht zu betrachten, Modelldifferenzen werden hierbei allerdings nicht berücksichtigt. Doch gerade diese sind in dieser Arbeit interessant.

Werden während eines Projektes kontinuierlich Modelldifferenzen aufgedeckt und propagiert, so fördert dies die Kommunikation zwischen den Beteiligten und bestehende Missverständnisse werden erkannt und ausgeräumt.

Im Folgenden wird nun der Prozess der DLR-Einrichtung SISTEC betrachtet und nach Integration-Möglichkeiten für eine Werkzeug zur Erkennung von Modelldifferenzen gesucht.

4.1.1 Ist-Prozess

Eine übliche Herangehensweise bei der Softwareentwicklung ist es, zunächst die Anforderungen zu klären und festzuhalten. Im Anschluss werden Anwendungsfälle definiert und man gelangt über das Grobdesign zum Feindesign und schließlich zur Implementierung.

Der Software-Entwicklungsprozess, wie er von der Einrichtung „Simulations- und Softwaretechnik“ des Deutschen Zentrums für Luft- und Raumfahrt (DLR) vorgeschlagen wird, ist nach [ECSS1996] in vier Phasen aufgeteilt:

A Planung

B Primäres Design

C/D Design und Produktion

E Operation

Wie erwartet wird in Phase A zunächst das Projekt geplant und in Phase B ein primäres Design entworfen. Anschließend wird in der Phase C/D (siehe Abb. 4.1, S. 41) der Entwurf verfeinert und die Software produziert. Anschließend wird der Betrieb der Software aufgenommen (Phase E), was allgemein auch als Wartungphase bekannt ist. Im Folgenden wird nun auf die Phase C/D näher eingegangen.

Phase C/D: Moderne Entwicklungsprozesse befürworten die inkrementelle Entwicklung, daher sind die Aktivitäten im SW-Prozess aufgeteilt in:

- daily (täglich oder gar stündlich)
- monthly (monatlich oder gar wöchentlich)
- quarterly (quartalsweise oder gar monatlich)

Die genaue Dauer eines Inkrements ist von dem jeweiligen Projekt abhängig, ebenso versteht sich der gesamte Prozess als Vorschlag und muss bei Bedarf an die Gegebenheiten eines konkreten Projektes angepasst werden (Tailoring). Die Aktivitäten des Prozesses sind in drei zeitliche Ebenen (vierteljährlich, monatlich und täglich) aufgeteilt. Innerhalb einer Ebene werden die Aktivitäten von links nach rechts ausgeführt und ggf. in eine tiefere bzw. höhere Ebene verzweigt. Die einzelnen Aktivitäten haben in ihrer linken oberen Ecke eine Nummer, damit sie referenziert werden können. Außerdem hat jede Aktivität einen Namen und unter ihr ist ein Werkzeug vermerkt, was zur Unterstützung bei SISTEC eingesetzt bzw. vorgeschlagen wird. Ist kein Werkzeug vorhanden oder bekannt, so ist ein Fragezeichen vermerkt. Die

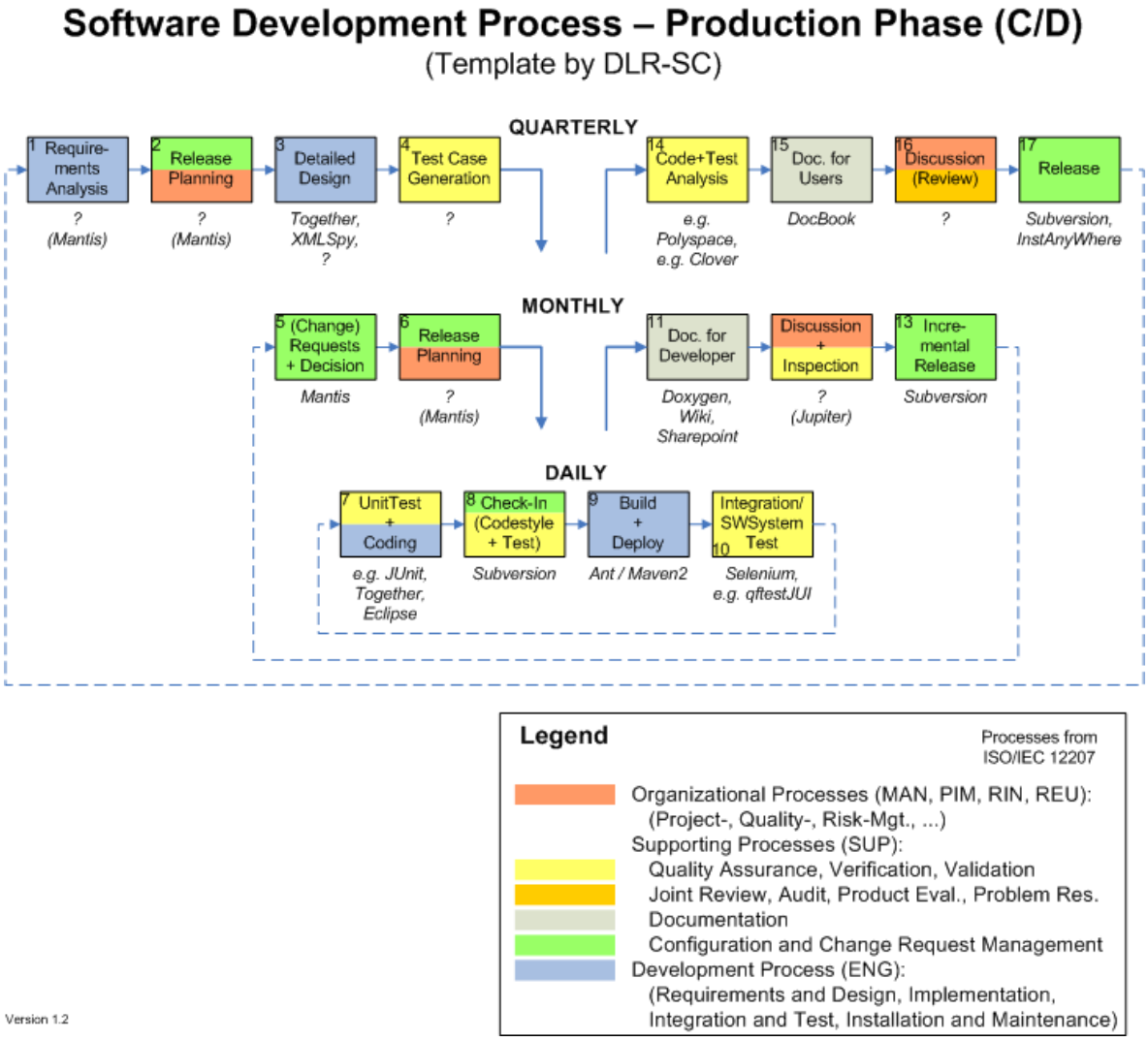

Abbildung 4.1: Softwareentwicklungsprozess des DLR

gestrichelten Pfeile verdeutlichen die wiederkehrenden Aktivitäten auf den unterschiedlichen Ebenen. Zusätzlich sind die Aktivitäten farblich einem Prozess nach ISO 12207 zugeordnet, der in der Legende näher erläutert wird. Hierbei stellen eingerückte Prozesse Subprozesse dar.

Die einzelnen Aktivitäten des SW-Entwicklungsprozesses werden im Weiteren genauer beschrieben:

Vierteljährlich ausgeführte Aktivitäten:

1. *Requirements Analysis*: Ermittlung/Definition von abstrakteren Anforderungen (bzw. Features). Derzeit wird kein kommerzielles Requirements-Management-Tool (wie z.B. Caliber Analyst, DOORS oder RequisitePro) benutzt, stattdessen werden die Anforderungen teilweise im Bugtracking-System [MANTIS] über Vater-Kind-Beziehungen verwaltet.

2. *Release Planning*: Die Release-Planung wird teilweise im MANTIS durchgeführt, indem Anforderungen Release-Nummern zugeordnet bekommen.

3. *Detailed Design*: Der Feinentwurf einer Anforderung muss drei Aspekte berücksichtigen:

   - Komponenten-Modellierung (Klassendiagramm) z.B. durch Together.
   - Daten-Modellierung (XML- bzw. Datenbank-Schemata) z.B. durch XMLSpy.
   - Verhaltens-Modelling (dynamisches Zustandsdiagramm); hier sollte ein Werkzeug zur Zustandsmodellierung genutzt werden (wie z.B. StateFlow oder Statemate), da UML-Use-Cases in der Regel zu aufwendig sind.

4. *Test Case Generation*: Akzeptanz- bzw. Systemtests werden aus den Anforderungen erstellt. Im Falle einer testgetriebenen Entwicklung sollten die Tests möglichst frühzeitig erstellt und automatisiert werden. Es ist derzeit kein praktikables Werkzeug zur Generierung von Akzeptanz- und Systemtests zur Unterstützung dieser Aktivität bekannt.

Monatlich ausgeführte Aktivitäten:

5. *(Change) Requests + Descision*: Für das Change-, Request- und Problem-Management wird ebenfalls MANTIS benutzt. Hier werden die Einträge auch bewertet und Entwicklern zugeteilt.

6. *Release Planning*: Die inkrementelle Release-Planung wird teilweise im MANTIS durchgeführt (z.B. über Prioritäten oder das Feld "Release").

Täglich ausgeführte Aktivitäten:

7. *UnitTest + Coding*: Der Entwicklung von UnitTests (z.B. mit JUnit) folgt die Entwicklung der Software (z.B. mit Eclipse und Together).

8. *Check-In (Codestyle + Test)*: Einchecken der bearbeiteten Dateien in das Subversion-Repository mit automatischer Ausführung aller Unit-Tests und Überprüfungen der Kodierrichtlinien (über Hook-Skripte mit z.B. JUnit und Checkstyle). *An dieser Stelle besteht eine Möglichkeit, MoDi einzusetzen*

9. *Build + Deploy*: Automatisiertes Erzeugen der Binär-Dateien im Nightly-Build und Installation auf dem Zielsystem (Deployment) durch Ant/Maven2. *An dieser Stelle besteht eine weitere Möglichkeit, MoDi einzusetzen*

10. *Integration / System Test*: Automatisierte Software-Integrationstests und -Systemtests werden bislang nur teilweise durchgeführt (z.B. durch Selenium oder QF-Test).

Monatlich ausgeführte Aktivitäten (nach täglicher Entwicklungsphase):

11. *Doc for Developer*: Die Entwickler-Dokumentation wird zum Teil automatisch erzeugt (z.B. durch Doxygen) und teilweise manuell in Content Management Systemen (z.B. [MoinMoin] oder [MS Sharepoint]) gepflegt.

12. *Discussion + Inspection*: Entwickler-Treffen sollten durch eine Microsoft-Sharepoint-Seite unterstützt werden. Code-Inspections werden künftig evtl. durch Jupiter [Jupiter] unterstützt.

13. *Incremental-Release*: Ein inkrementelles Release wird mit Hilfe von Subversion erzeugt.

Vierteljährlich ausgeführte Aktivitäten (nach monatlicher Entwicklungsphase):

14. *Code + Test Analysis*: Aufwendige Code-Analysen (soweit nicht bereits während der Entwicklung geschehen) können mit z.B. Polyspace erstellt werden. Analysen bezüglich der Tests (z.B. Testabdeckung durch Clover [Clover]) erfolgen spätestens jetzt.

15. *Doc for User*: Die Nutzer-Dokumentation wird mit Hilfe von DocBook erzeugt.

16. *Discussion (Review)*: Projekt-Treffen werden bei SISTEC durch eine Microsoft Sharepoint-Seite unterstützt.

17. *Release*: Ein vollständiges Release wird mit Hilfe von Subversion erzeugt.

Die Aktivitäten 9, 10, 11, 14, 15, werden von CruiseControl unterstützt. In Aktivitäten acht und neun zeigen sich zwei Einsatzmöglichkeiten des Werkzeuges zur Erkennung von Modelldifferenzen. Außerdem ist ferstzuhalten, dass sich während Aktivität acht die Modelldifferenzen manifestieren.

4.2 Kommunikation in einem Softwareprojekt

Es ist von Projekt zu Projekt unterschiedlich, wieviel Kommunikation erforderlich ist, um erfolgreich beendet zu werden. Das Maß an Kommunikation hängt sicherlich von verschiedenen Faktoren ab, wie zum Beispiel:

- Mitarbeitererfahrenheit (erfahren $\leftrightarrow$ unerfahren)
- Problemstellung (klar $\leftrightarrow$ unklar)
- Technologie (erprobt $\leftrightarrow$ neu)

Es gibt sicherlich noch weitere Kriterien, die hier aber nicht weiter betrachtet werden sollen. Schon mit diesen drei Kriterien wird klar, unter welchen Umständen wieviel Kommunikation sinnvoll ist. In Projekten mit bekannten Problemstellungen und Mitarbeitern, die ähnliche Projekte schon mehrfach bewältigt haben, hat die Kommunikation einen geringeren Stellenwert als beispielsweise in Projekten mit unklarer Problemstellung und unerfahrenen Mitarbeitern. Immer muss jedoch solange kommuniziert werden, bis sich alle Beteiligten über das Vorgehen bzw. die Software einig sind.

Schließlich muss das Maß an Kommunikation abgewägt werden, da sie sowohl helfen als auch schaden kann. Ziel der Kommunikation muss es immer sein, Missverständnisse auszuräumen, das weitere Vorgehen abzustimmen und, wo möglich, den Ablauf zu optimieren. Es gilt jedoch stets abzuwägen, ob weitere Kommunikation über Optimierung sinnvoll ist, oder ob mit dem gleichen Aufwand die zu erledigende Arbeit bereits getan wäre.

4.3 Werkzeugintegration in den SW-Prozess

Um die Stelle zu identifizieren, an der das Werkzeug zur Erkennung von Modelldifferenzen in den Software-Entwicklungsprozess integriert werden kann, werden im Folgenden die Möglichkeiten erörtert und anschließend eine Empfehlung gegeben.

Für den Einsatz kommen prinzipiell alle Ebenen (siehe Abb. 4.1, S. 41) in Betracht, aber auch in kürzeren Intervallen lässt sich ein solches Werkzeug einsetzen. Zum Beispiel:

- in der Integrierten Entwicklungsumgebung (IDE) - minütlich,
- in einem Source-Code-Management-System (SCM-System) pro Commit - stündlich bis täglich (Aktivitäten 8 in Abb. 4.1),
- im Daily-Build - täglich (Aktivität 9 in Abb. 4.1),
- im inkrementellen Release - monatlich (Aktivität 13 in Abb. 4.1) oder
- im Review - quartalsweise (Aktivität 14 oder 16 in Abb. 4.1).
- In Post-Mortem-Analysen - Hier ist dann Kommunikation über Modelldifferenzen als Erfahrungsgewinn möglich.

Nachdem das AM erstellt oder angepasst worden ist, wird es kommuniziert und von den Entwicklern übernommen. Dies stellt die initiale Situation dar. Danach schreitet die Entwicklung fort und es sind Modelldifferenzen zu erwarten. Idealerweise stellen die entstandenen Modelldifferenzen den Fortschritt im Projekt dar und können nach kurzer Kommunikation in das AM übernommen werden, sodass die Modelldifferenzen minimiert werden und alle Beteiligten auf dem neuesten Stand sind. Je länger es aber dauert, bis geprüft wird, ob es zu Modelldifferenzen gekommen ist, desto weiter kann sich die Implementierung vom Design entfernen, ohne dass dies aufgedeckt und kommuniziert wird. Je größer die so entstehende Drift zwischen Design und Implementierung, desto mehr Modelldifferenzen gibt es und desto mehr Kommunikation muss stattfinden. Die Frequenz der Prüfungen bestimmt also die Menge an Kommunikation. Es ist sinnvoll, die Frequenz der Prüfungen möglichst hoch zu halten, damit genug kommuniziert wird. Ist die Frequenz zu niedrig, wird also seltener geprüft, so müssen mehr Modelldifferenzen auf einmal kommuniziert werden. Da die Organisation der Kommunikation und die Kommunikation selbst u.U. einen erheblichen Aufwand darstellen (vergl. 4.2), ist dies nicht wünschenswert.

Der Einsatz pro Commit ist ebenfalls zu erwägen. SCM-Systeme wie zum Beispiel [Subversion] bieten durch sogenannte Hook-Skripte die Möglichkeit, Commits Prüfungen zu unterziehen. Ein Commit kann dann akzeptiert oder abgelehnt werden, wenn zum Beispiel die Unit-Tests nicht erfolgreich durchlaufen oder in es diesem Fall zu Modelldifferenzen kommt. Es ist jedoch wünschenswert, dass die Entwickler ihre Ideen zunächst ungehindert veröffentlichen können und eine Diskussion erst im Anschluss stattfindet. So werden die in Abschnitt 3.5.3 geschilderten Workarounds vermieden. Ferner besteht die Möglichkeit nach einem erfolgten Commit Prüfungen durchzuführen. So kann nach jedem Commit ein Modellvergleich durchgeführt und der entstehende Bericht an die entsprechenden Rollen versandt werden. Bei größeren Teams

ist eine höhere Frequenz der Commits zu erwarten und es werden mehr Berichte verschickt, die kommuniziert werden müssen. Hier ist es sinnvoll, die angefallenen Modelldifferenzen zu sammeln und gemeinsam zu besprechen.

Wenn die Berichte aber gesammelt besprochen werden, bietet sich eher an, ein Werkzeug zur Erkennung von Modelldifferenzen im Daily-Build einzusetzen und die Kommunikation über die gefundenen Modelldifferenzen beispielsweise in einer tägliche Besprechung, wie es in SCRUM [Schwaber2001] praktiziert wird, zu integrieren. Damit wäre ein Kompromiss zwischen Aufwand und erforderlicher Kommunikation gefunden. Auf der einen Seite kann der Architekt die Umsetzung seines Architekenmodells kontinuierlich überprüfen und seine Ideen kommunizieren. Auf der anderen Seite haben so auch die Entwickler die Möglichkeit, Probleme mit der Architektur rechtzeitig zu eskalieren, ohne das ihr Arbeitsfluss unterbrochen wird. Insgesamt können so Missverständnisse schnell erkannt und ausgeräumt werden. So folgt:

> Ein Werkzeug zur Erkennung von Modelldifferenzen sollte projektbegleitend im Daily-Build eingesetzt werden. (4.1)

> Die aufgedeckten Modelldifferenzen sollen in einer täglichen MoDi-Besprechung kommuniziert werden. (4.2)

Ferner ist es wichtig, dass die Modelldifferenzen nicht nur aufgedeckt und kommuniziert werden, es muss auch Konsequenzen für das Fortschreiten geben. So, dass die Kommunikation ein Bedürfnis wird, damit weitergearbeitet werden kann. Nur so ist sichergestellt, dass die erforderliche Kommunikation auch wirklich stattfindet.

Konsequenzen ergeben sich, wenn ein Werkzeug zur Erkennung von Modelldifferenzen

- in der IDE integriert ist, die ein Einchecken bei Modelldifferenzen nicht erlaubt. Dann bekommt der Entwickler ein schnelles Feedback und kann schon vor einem Commit Modelldifferenzen erkennen und beheben bzw. kommunizieren.
- in einem SCM-Werkzeug wie z.B. [SVNChecker] eingesetzt wird. Dann kann eine Konsequenz aus einer Modelldifferenz sein, dass ein Commit abgelehnt wird. Oder ein Commit wird erlaubt und es werden Nachrichten an verantwortliche Personen verschickt. Wann ein Commit abgelehnt oder trotz Modelldifferenzen akzeptiert werden soll, kann anhand der Bewertung (Severity) durch die Regeln entschieden werden.
- im Daily-Build eingesetzt wird, dann kann ein Buils scheitern, wenn Modelldifferenzen aufgedeckt werden.
- bei einem Release benutzt wird. Dann könnte zum Beispiel das Release erst stattfinden, wenn alle Modelldifferenzen beseitigt worden sind. Damit dies geschehen kann, muss wiederum Kommunikation stattfinden.
- bei einem Review zum Einsatz kommt, um zum Beispiel Vertragsverletzungen aufzudecken. Dies kann insbesondere hilfreich sein, wenn Teile der Entwicklung ausgelagert

werden. Wird ein Architektenmodell Bestandteil des Vertrages zwischen Softwarekunde und -firma, so kann die gelieferte Software gegen ihre Spezifikation geprüft werden. Ein Kunde kann sich ggf. auch einen Modellvergleich vorführen lassen und somit einen Whitebox-Akzeptanztest durchführen. Bei dem Modellvergleich als Akzeptanztest dürfen dann keine Modelldifferenzen auftreten.

Aus der Feststellung 4.1 (S. 45) folgt der Einsatz im Daily-Build und die Besprechung der Ergebnisse in einer täglichen MoDi-Besprechung 4.2 (S. 45). So ist eine kontinuierliche Kommunikation im Projekt gewährleistet. Ferner bestimmt die Menge der Modelldifferenzen das Maß an Kommunikation in den einzelnen Besprechungen. Je weniger Modelldifferenzen es gibt, desto kürzer fällt das Meeting aus und alle kommen schneller wieder an die Arbeit. Die Teilnehmer bekommen ein Gefühl dafür, „was im Projekt los ist“ und wenn kommuniziert wird, dann wird sich ein selbstverstärkender Prozess einstellen.

Für den beschriebenen Software-Entwicklungsprozess (siehe Abb. 4.1, S. 41) heißt das:

Das Werkzeug zur Erkennung von Modelldifferenzen soll in die Aktivität 9 *Build + Deploy* integriert werden und die Ergebnisse sollen vor Aktivität 7 *UnitTest + Coding* besprochen werden. (4.3)

4.4 Zusammenfassung

In diesem Kapitel wurden die Konsequenzen für einen Software-Entwicklungsprozess besprochen. Dies ist am Beispiel des Prozesses, wie er von der Einrichtung Simulations- und Softwaretechnik des Deutschen Zentrums für Luft- und Raumfahrt vorgeschlagen wird geschehen. Es wurden die einzelnen Phasen des Software-Entwicklungsprozess erläutert und anschließend untersucht wieviel Kommunikation über die entstandenen Modelldifferenzen zwischen dem Architekten und den Entwicklern erforderlich ist. Schließlich wurde dargelegt, welche Möglichkeiten es zur Integration eines Werkzeuges für die Erkennung von Modelldifferenzen in den Software-Entwicklungsprozess gibt. Mit Feststellung 4.3 wurde eine Empfehlung ausgesprochen die besagt, dass Modelldifferenzen im Daily-Build erkannt und täglich kommuniziert werden sollten.

Kapitel 5

Realisierung des Werkzeuges zur Erkennung von Modelldifferenzen

In diesem Kapitel werden die Anforderungen, der Entwurf und die Implementation eines Werkzeuges zur Erkennung von Modelldifferenzen, im Folgenden MoDi genannt, beschrieben.

Zunächst werden Anwendungsfälle beschrieben, aus denen dann detailliertere Anforderungen abgeleitet werden sollen.

5.1 Anwendungsfälle

Als erstes wurden die Stakeholder identifiziert. Die direkt betroffenen Personen sind der Architekt und Entwickler. Der Architekt erstellt ein Architektenmodell, welches die Vorgabe für die Entwickler bildet. Die Entwickler erhalten das AM und beginnen mit der Umsetzung. Dabei erweitern sie das EM und so entstehen Modelldifferenzen. Anschließend muss das EM gegen das AM geprüft werden. Bei dem Modellvergleich werden Modelldifferenzen aufgedeckt, die von Architekt und Entwicklern kommuniziert werden. Der Architekt entscheidet dann, ob die Änderungen, die zu den Modelldifferenzen geführt haben in das AM übernommen werden oder nicht.

Der Hauptanwendungsfall für MoDi ist das „Aufdecken von Modelldifferenzen“, welcher sich in vier Unteranwendungsfälle gliedern lässt (siehe Abb. 5.1, S. 48):

1. Modell bereitstellen, zum einen das AM und zum anderen das EM.
2. Regeln definieren, Einstellen, welche Modelldifferenzen erkannt und wie diese bewertet werden sollen.
3. Modelle vergleichen, Modellvergleich bewirken.
4. Bericht auswerten, Modelldifferenzen einsehen und interpretieren.

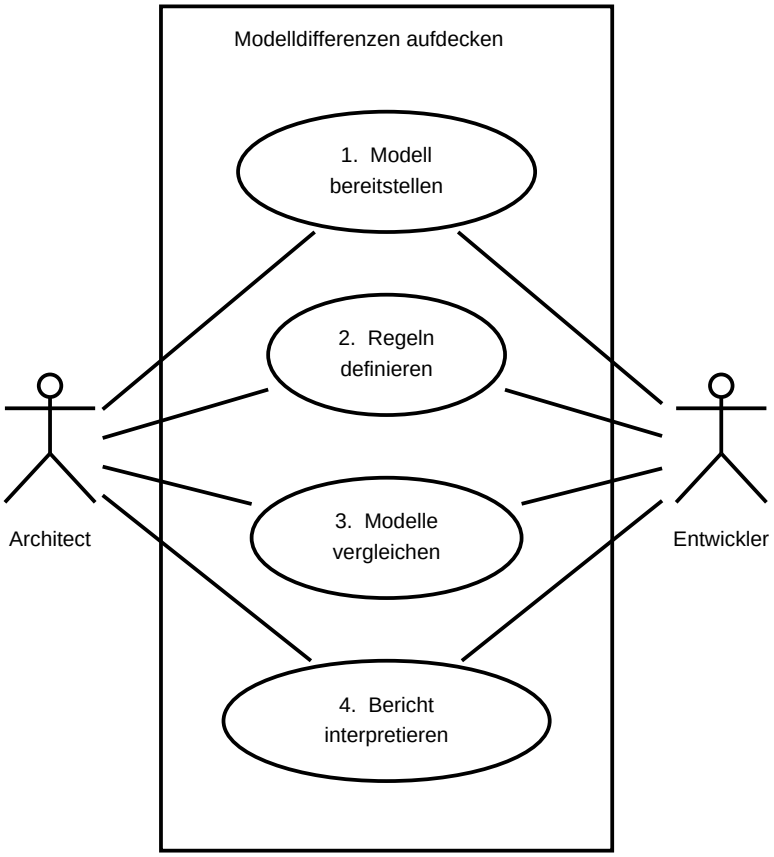

Abbildung 5.1: Anwendungsfall: „Modellvergleich aufdecken“

Zunächst müssen die Modelle bereitgestellt werden (1.). Anschließend wird eine Menge von Regeln definiert (2.), die die Modelldifferenzen bewerten (z.B. „Das Löschen eines Interfaces ist schwerwiegend“). Dann kann ein Modellvergleich durchgeführt werden (3.) und schließlich muss der entstandene Bericht ausgewertet und interpretiert werden (4.). Die Ergebnisse sind dann im Team zu kommunizieren. Das Anwendungsfalldiagramm gibt diese Reihenfolge natürlich nicht vor ([Oestereich2001] S.197)

Im Folgenden werden die vier Anwendungsfälle noch einmal tabellarisch und detaillierter dargestellt.

Use Case Nr. 1	*Modell bereitstellen.*
Umfeld	IT-Infrastruktur
Ebene	Dateisystem
Hauptakteur	System
Stakeholder u. Interesse	*Stakeholder* — *Interesse* Architekt — Einstellen eines Architektenmodells Entwickler — Einstellen eines Entwicklermodells
Voraussetzungen	Internetverbindung, Plattenplatz (Subversion-Zugangsdaten bekannt), Daten fertig editiert
Erfolgsfall	Für nachfolgende Komponenten steht eine *working copy* des AMs und des EMs bereit.
Auslöser	Architekt, Entwickler
Beschreibung	Schritt Aktion 1. Einchecken eines Moduls in das Repository. 2. serverseitiges Auschecken des Moduls aus dem Repository in ein vorkonfiguriertes Architekten- bzw. Entwicklermodellverzeichnis.
Technologie	CruiseControl / Skripte

Tabelle 5.1: UC: Modelle bereitstellen

Use Case Nr. 2	*Regeln definieren*
Umfeld	Konfiguration
Ebene	Konfiguration
Hauptakteur	Architekt
Stakeholder u. Interesse	*Stakeholder* — *Interesse* Architekt — erstellt oder ändert Regeln für den Modellvergleich
Voraussetzungen	System installiert, Schreibrechte
Erfolgsfall	gültige Konfiguration mit Regeln für das System verfügbar.
Auslöser	Regelwerk initial erstellen oder ändern
Beschreibung	Schritt Aktion 1. Menge von Checks definieren 2. Namen für die Regeln festlegen 3. Schweregrad für die Regeln festlegen {`HIGH, MEDIUM, LOW`}
Technologie	Konfigurationsdatei

Tabelle 5.2: UC: Regeln definieren

Use Case Nr. 3	*Modelle vergleichen*
Umfeld	Server
Ebene	Metarepräsentation
Hauptakteur	System
Stakeholder u. Interesse	*Stakeholder* — *Interesse* Architekt, Entwickler — Bericht erhalten
Voraussetzungen	AM und EM sind auf dem Server vorhanden
Erfolgsfall	Es entsteht ein Bericht über den Modellvergleich
Auslöser	Nightly Build, OnDemand, Hook-Skript
Beschreibung	Schritt Aktion 1. Das AM und EM werden in die Metarepräsentation übersetzt 2. Konfigurierte Regeln werden ausgewertet 3. Bericht wird erstellt und gespeichert
Technologie	Java, ANTLR, Subversion, CruiseControl

Tabelle 5.3: UC: Modelle vergleichen

Use Case Nr. 4	*Bericht interpretieren*
Umfeld	MoDi-Besprechung
Ebene	Kommunikation
Hauptakteur	Architekt, Entwickler
Stakeholder u. Interesse	*Stakeholder* — *Interesse* Architekt, Entwickler — Bericht einsehen und auswerten
Voraussetzungen	Modellvergleich würde durchgeführt
Erfolgsfall	Es findet Kommunikation über die gefundenen Modelldifferenzen statt.
Auslöser	Modellvergleich
Beschreibung	Schritt Aktion 1. Ein Bericht über Modelldifferenzen geht per E-Mail vom Nightly Build ein. 2. Der Bericht wird gelesen und interpretiert. 3. Alle Modelldifferenzen werden kommuniziert. 4. Leichte Differenzen werden abgesegnet. 5. Für Differenzen wird eine Lösung erarbeitet und ggf. das AM angepasst.
Technologie	E-Mail, Chat, Telefon, Videokonferenz, Whiteboard

Tabelle 5.4: UC: Bericht interpretieren

Modelle bereitstellen: Bei diesem Anwendungsfall stellt ein Architekt bzw. ein Entwickler sein Modell (AM bzw. EM) dem System bereit. Ein bereitgestelltes Modell wird dann später zu einem Modellvergleich herangezogen. Da für die Vergleichsbasis Quelltext benötigt wird (vergl. Folgerung 3.2 25) und das Entwicklermodell in aller Regel schon in einem Repository gespeichert ist, bietet es sich an das AM ebenfalls dort zu speichern.

Regeln definieren: Ein Architekt definiert Regeln für den Modellvergleich. Bei der Definition einer Regel wird festgelegt, welche Modelldifferenzen erkannt werden sollen und wie diese bewertet werden. Hierzu stehen eine Reihe von Checks zur Verfügung und es kann für den Schweregrad aus den Werten {`HIGH, MEDIUM, LOW`} ausgewählt werden.

Modelle vergleichen: Je nach Einstellung wird ein Modellvergleich beispielsweise durch ein Hook-Skript initiiert. Dies ist der Fall, wenn der Modellvergleich an die Commits gebunden ist und MoDi somit stündlich bis täglich eingesetzt wird. Beim Einsatz im Daily-Build kann der MV beispielsweise durch [CruiseControl] initiiert werden.

Bericht interpretieren: Abschließend muss der entstandene Bericht über die Modelldifferenzen interpretiert werden. Dies muss durch die beteiligten Rollen geschehen und kann nicht automatisiert werden.

Nachdem die Anwendungsfälle besprochen worden sind, werden im nächsten Abschnitt die Detailanforderungen bezüglich der drei zu automatisierenden Schritte dargestellt.

5.2 Detailanforderungen

Im Folgenden werden die Anforderungen an MoDi identifiziert. Prinzipiell sind drei Schritte nötig, um Modelldifferenzen aufzudecken. Zunächst müssen die beiden zu vergleichenden Modelle importiert werden. Anschließend findet der eigentliche Vergleich statt, auf dessen Grundlage abschließend ein Bericht generiert wirden. Das Format der Anforderungen folgen dabei den *Sprachregeln für Anforderungen* aus der Vorlesung „Anforderungen und Entwurf“ des Fachgebietes für Software-Engineerung aus dem Sommersemester 2005.

Transformation: Codebasen/Architektenrmodelle/SCM

1. Das System muss Code-Basen aus dem Dateisystem entnehmen können.
2. Das System muss Architektencode (AC) in die Metarepräsentation umwandeln, wenn ein Modellvergleich durchgeführt wird (Folgerung 3.3, 3.4).
3. Das System muss Entwicklercode (IC) in die Metarepräsentation umwandeln, wenn ein Modellvergleich durchgeführt wird (Folgerung 3.3, 3.4).
4. Das System muss dem Architekten ermöglichen, eine Revision seines AMs für den Modellvergleich festzulegen.

Verarbeitung: Regeln/Vergleich

5. Das System muss dem Architekt ermöglichen, Regeln für Modellvergleiche festzulegen.
   - 5.1 Das System muss einem Architekten ermöglichen, eine Regel zu erstellen.
   - 5.2 Das System muss einem Architekten ermöglichen, eine Regel zu ändern.
   - 5.3 Das System muss einem Architekten ermöglichen, eine Regel zu löschen.
   - 5.4 Das System muss einem Architekten ermöglichen, festzulegen, wie schwerwiegend ein Regel ist (mögliche Werte sind {`HIGH, MEDIUM, LOW`}).
6. Das System muss das Entwicklermodell mit dem Architektenmodell auf Basis von Regeln vergleichen.
7. Das System muss Interfaces aus der Metarepräsentation filtern können.
8. Das System muss Interfaces aus dem Architektenmodell den Interfaces aus dem Entwicklermodell zuordnen.
9. Ein MV muss durch ein Subversion-Hook-Skript initiiert werden können.
10. Ein MV muss durch ein CruiseControl-Skript initiiert werden können.

Ergebnis: Bericht/Benachrichtigung

11. Das System muss einen Bericht über den Vergleich erzeugen können.
12. Das System muss beteiligte Rollen mit dem Bericht benachrichtigen.
    - 12.1 Das System soll den Entwickler mit einem Entwicklerbericht benachrichtigen (z.B. Jupiter-Report [Jupiter]).
    - 12.2 Das System soll den Architekten mit einem Architektenbericht benachrichtigen (z.B. XHTML-Report).

Nachdem die Anforderungen erfasst worden sind. kann im Folgenden der Entwurf dargestellt werden.

5.3 Entwurf

In diesem Abschnitt wird der Entwurf von MoDi beschrieben. Beginnend mit der Grobarchitektur wird das Design verfeinert und die einzelnen Komponenten erklärt.

5.3.1 Grobentwurf – MoDi im Überblick

In diesem Abschnitt wird die prinzipielle Struktur des Werkzeugs MoDi als Grobentwurf dargestellt und es soll ein Überblick über das System gegeben werden (siehe Abb. 5.2, S. 54).

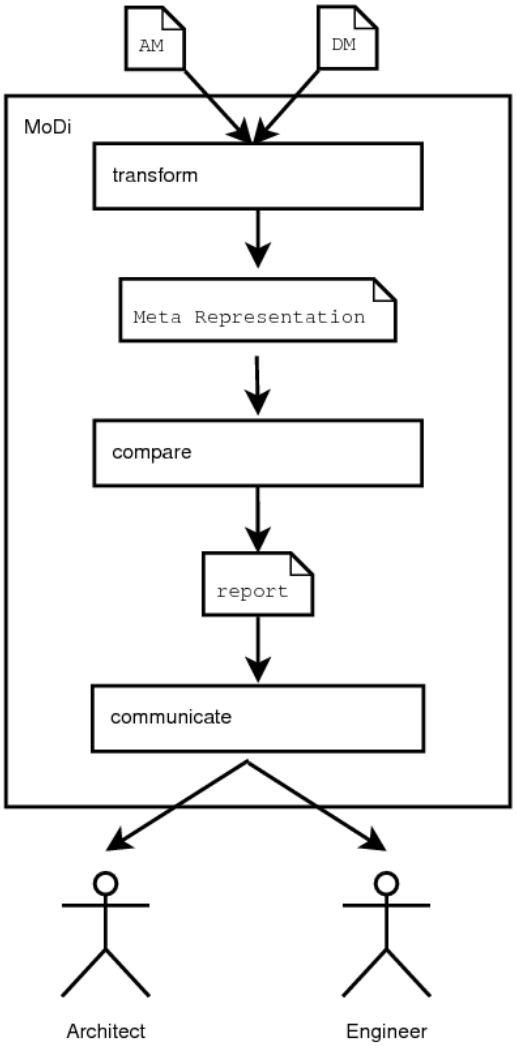

Abbildung 5.2: Grobentwurf des Systems MoDi

Das System muss ein Architekten- (AM) und ein Entwicklermodell (EM), die als kompilierbare Code-Basen vorliegen (vergl. Feststellung 3.2) in die Metarepräsentation (vergl. Feststellung 3.4) überführen. Anschließend werden durch einen Modellvergleich die Modelldifferenzen gesammelt und in einem Bericht zusammengefasst.

Die Transformationskomponente (transform) überführt die zu vergleichenden Modelle in die Metarepräsentation (siehe Abschnitt 3.3, S. 26).

Die Vergleichskomponente (compare) führt nun einen Modellvergleich durch, bei dem die vorkonfigurierten Regeln ausgewertet werden. Hierzu werden die Eigenschaften der zu vergleichenden Modelle von der Metarepräsentation abgefragt und die Ergebnisse des Vergleichs in einem Bericht (report) zusammengefasst.

Die Kommunikationskomponente (communicate) verschickt den entstandenen Bericht an die

beteiligten Rollen und soll so die nötige Kommunikation einleiten.

5.3.2 Feinentwurf – Die einzelnen Komponenten

In Abbildung 5.2 (S. 54) wurde ein Überblick über das System gegeben. Im Folgenden werden nun die einzelnen Komponenten vorgestellt.

Transformation

Die Transformationskomponente (siehe Abb. 5.3, S. 55) überführt das Architekten- und das Entwicklermodell in die Metarepräsentation. Das AM und das EM liegen als Quelltext in entsprechenden Ordnern vor und die Pfade zu diesen Ordnern werden dem System als Parameter übergeben.

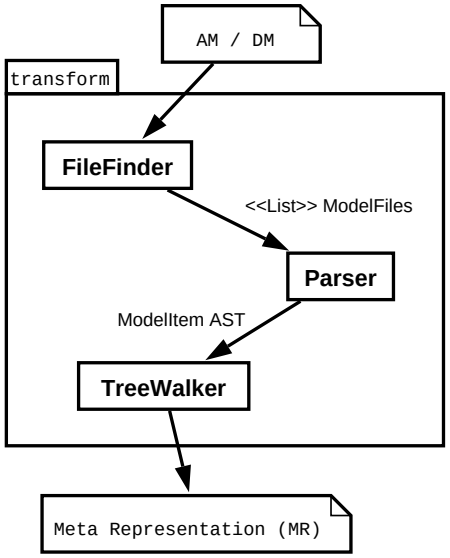

Abbildung 5.3: Transformationskomponente mit den einzelnen Subkomponenten

Ein `FileFinder`-Objekt findet zunächst alle relevanten Dateien. Anschließend wird die so entstehende Datei-Liste an einen `Parser` übergeben, der für jede Datei einen *Abstract Syntax Tree* (AST) erstellt. Die so entstehenden Bäume werden von einem `TreeWalker` traversiert, welcher die relevanten Informationen ausliest und die Metarepräsentation (MR) (siehe Abschnitt 3.3, S. 26) sukkzesive füllt.

Modellvergleich

In der Vergleichskomponente (siehe Abb. 5.4, S. 56) werden nun die in der Metarepräsentation gespeicherten Modelle verglichen und ein Bericht über den Vergleich erstellt.

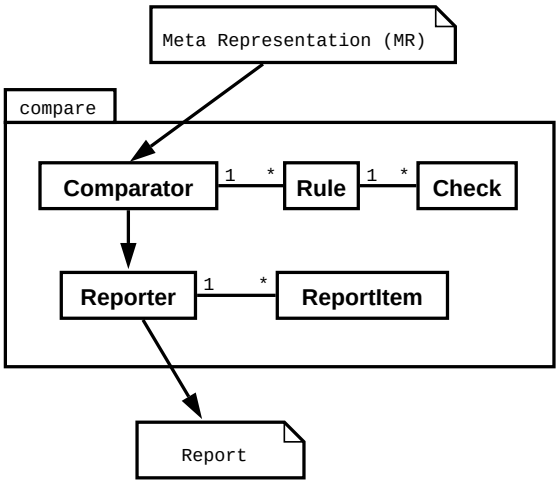

Abbildung 5.4: Vergleichskomponente mit den einzelnen Subkomponenten

Der auf der Metarepräsentation arbeitende `Comparator` wertet die konfigurierten Regeln (`Rule`) aus. Hierzu führt er die den Regeln zugrundeliegenden Checks (`Check`) aus und fügt die hierdurch gefundenen Modelldifferenzen einem Bericht hinzu. Jede Regel bewertet hierbei gleichzeitig die Ergebnisse der Checks bezüglich eines Schweregrades (*severity = HIGH, MEDIUM, LOW*). Die Modelldifferenzen werden in einem Bericht festgehalten, welcher zusätzlich noch Metadaten über das Projekt speichert. Der Bericht kann in verschiedenen Formaten durch sog. Reporter generiert werden, sodass zum Beispiel für den Entwickler ein Review-Report (Jupiter [Jupiter] (XML)) und für den Architekten ein HTML-Dokument erstellt werden kann.

Kommunikation

Die Kommunikationskomponente (siehe Abb. 5.5, S. 56) schließt sich an die Vergleichskomponente an und reicht den Bericht an einstellbare Rollen weiter.

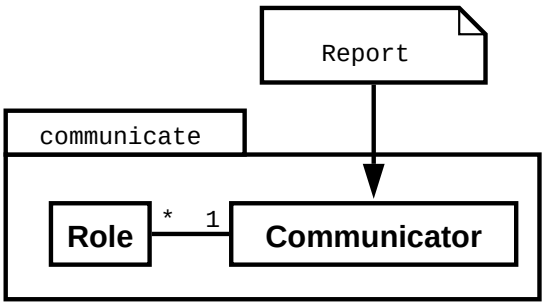

Abbildung 5.5: Kommunikationskomponente mit den einzelnen Subkomponenten

Die Empfänger (Architekt, Entwickler) werden in der Konfigurationsdatei eingetragen. Für die Kommunukationskomponente kann der Bericht in verschiedenen Formaten angefertigt werden, um auf die Bedürfnisse des Empfängers einzugehen. Hier sind Berichte im erwähnten Jupiter [Jupiter]-Format für die Entwickler und im xHTML- oder PDF-Format für den Architekten sinnvoll.

Im Folgenden wird auf einige Implementierungsdetails näher eingegangen und beispielsweise das Konzept der Checks erläutert, welches dazu dient Modelldifferenzen aufzudecken.

5.4 Implementierung

5.4.1 Checks

Checks sind in MoDi das Konzept zur Überprüfung von bestimmten Fragestellungen an die Metarepräsentation. Sie bekommen über die Regeln ihren Schweregrad zugeordnet. In dieser Arbeit sind Checks implementiert worden, die Antworten auf die Fragestellungen aus Kapitel 3 geben.

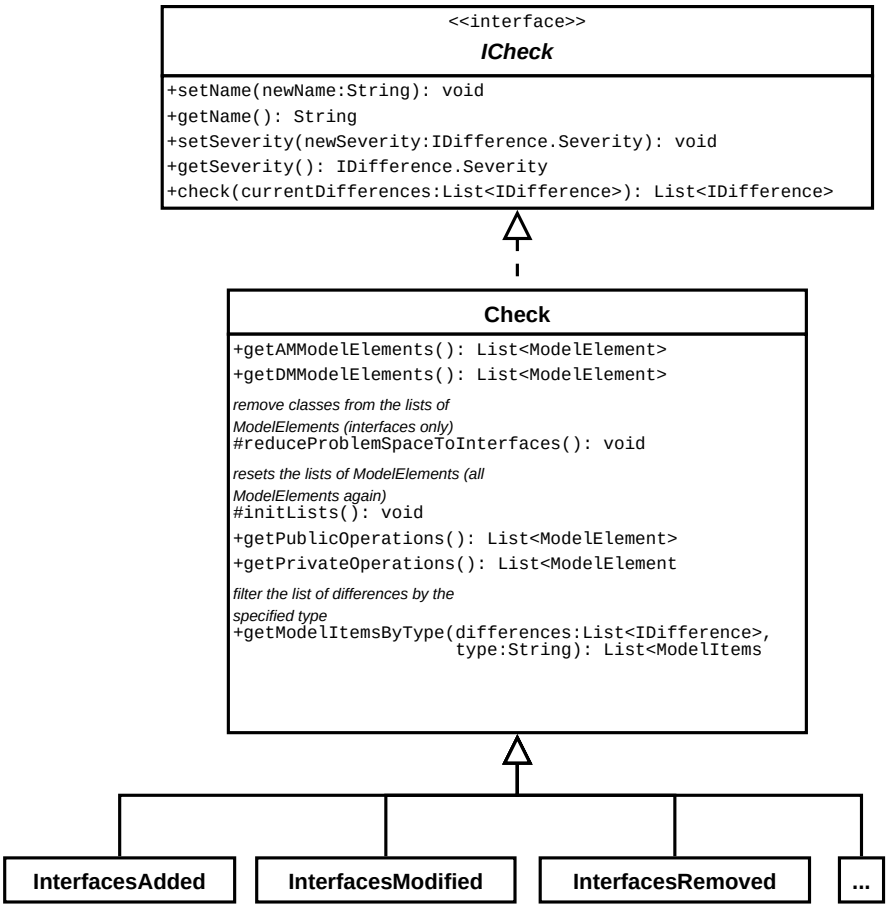

Abbildung 5.6: Vererbungshierarchie der Checks

In Abbildung 5.6 ist zunächst das Interface *ICheck* zu sehen, welches absichert, dass jeder implementierte Check einen Namen und einen Schweregrad besitzt. Weiterhin definiert es die Methode `check`, die eine Liste der bereits gefundenen Modelldifferenzen übergeben bekommt und eine Liste aller gefundenen Modelldifferenzen zurückgeben muss. Ferner ist die Klasse `Check` dargestellt, von der sich die verschiedenen implementierten Checks ableiten. Die Klasse `Check` implementiert verschiedene Hilfs-Methoden, die sie ihren Subklassen zur Verfügung stellt. Diese Hilfs-Methoden geben beispielsweise die Listen der Modellelemente zurück, können aber auch den Suchraum auf Interfaces begrenzen (`reduceProblemSpaceToInterfaces`), was bei den Checks `InterfacesAdded`, `InterfacesRemoved` und `InterfacesModified` sinnvoll ist. Die Methode `initLists` setzt die Listen der Modellelemente zurück. Dies ist sinnvoll, wenn zuvor `reduceProblemSpaceToInterfaces` aufgerufen wurde, anschließend jedoch wieder der gesamte Problemraum betrachtet werden muss. Darüber hinaus können Checks verkettet werden. Wenn zum Beispiel nach gelöschten Methoden in Interfaces gesucht wird, sollten die Klassen vorher ausgeschlossen werden und es reicht die modifizierten Interfaces zu untersuchen. Hier kann dann eine Regel mit den Checks `InterfacesModified, MembersRemoved` konfiguriert werden.

Im Folgenden werden die implemenierten Checks vorgestellt:

- `InterfacesAdded`: Dieser Check überprüft, ob es im EM Interfaces gibt, die nicht im AM vorkommen.
- `InterfacesRemoved`: Dieser Check überprüft, ob Interfaces gelöscht worden sind. Dies ist der Fall, wenn ein Interface im AM vorkommt, aber nicht im EM.
- `InterfacesModified`: Wenn ein Interface modifiziert worden ist, so wird dies von diesem Check erkannt.
- `MembersAdded`: Dieser Check sucht nach Membern, also Operationen oder Variablen, die zu einem modifizierten Modellelement hinzugefügt worden sind.
- `MembersRemoved`: Dieser Check findet die Member, die aus einem Modellelement gelöscht worden sind.
- `MembersModified`: Dieser Check überprüft, ob ein Modellelement modifiziert worden ist.
- `ParameterAdded`: Bei diesem Check handelt es sich um die Überprüfung von Operationen. Wenn eine Operation aus dem EM einen Parameter enthält, der nicht auch in der gleichen Operation aus dem AM vorhanden ist, so werden diese Modelldifferenzen von diesem Check gefunden.
- `ParameterRemoved`: Dieser Check funktioniert ähnlich wie der `ParameterAdded`-Check, nur wird hier nach gelöschten Parametern gesucht.
- `ModelItemsAdded`: Dieser Check findet alle Modellelemente (Klassen oder Interfaces), die im EM auftauchen, aber nicht im AM.
- `ModelItemsRemoved`: dieser Check findet alle Modellelemente, die gelöscht worden sind. Dabei handelt es sich um diejenigen, die im AM, aber nicht im EM vorhanden sind.

- `ModelItemsModified`: Dieser Check findet alle Modellelemente, die im EM modifiziert vorliegen.
- `AddedPublicMethodsInImplementingClasses`: Dieser Check findet diejenigen Operationen, die im EM spezifiziert sind, nicht jedoch im AM und zusätzlich müssen diese Operationen mit dem Modifier `public` gekennzeichnet sein (siehe Abschnitt 3.6.4, Regel 1).
- `AddedPrivateMembersInClasses`: Bei diesem Check handelt es sich um eine Implementierung zur Fragestellung aus 3.6.4 Regel 2. Es wurde überprüft, ob im EM im Vergleich zum AM private Methoden hinzugefügt worden sind.

Es können sicherlich noch weitere Checks bzgl. der Architektur implementiert werden. So war es nicht weiter schwierig einen Check zu implementieren, der überprüft, ob eine Klasse ein gesamtes Paket importiert (`import java.util.*;`). Ein solcher Check ist nützlich, wenn Beziehungen zwischen Modellelementen untersucht werden sollen. Dann muss aus den Namen und den Import-Anweisungen in den Modellelementen auf den vollqualifizierenden Namen des Modellelementes geschlossen werden können, weshalb *Star-Imports* im Vorfeld ausgeschlossen werden müssen.

Es ist zwischen Checks zu unterscheiden, die eindimensional – also auf nur einem Modell – arbeiten und denen, die auf dem AM **und** dem EM arbeiten. Für die eindimensionalen Checks können auch Werkzeuge wie Checkstyle [Checkstyle] oder PMD [PMD] eingesetzt werden. Wie oben am Beispiel der Star-Imports angedeutet kann es aber auch sinnvoll sein, eindimensionale Check zu implementieren, um Informationen für nachfolgende Checks bereitzustellen. Unter Umständen können sogar die in den anderen Projekten bereits implementierten Checks genutzt werden.

Die in Kapitel 3.6.4 Regel 1 beschriebene Fragestellung ist als `Check` mit dem Namen `AddedPublicMethodsInImplementingClasses` implementiert worden. Durch die Art der Implementierung des Systems MoDi ist das dort beschriebene Problem der geerbten Methoden allerdings ausgeschlossen, da diese nicht im Quelltext stehen und somit nicht geparst und auch nicht in die Metarepräsentation übernommen werden.

5.4.2 Bericht

Der bei einem Modellvergleich entstehende Bericht ist in einem eigenen XML-Format gehalten, dessen Wurzelknoten `report` heißt und einen `meta` Kindknoten mit Metadaten für das zu Grunde liegende Projekt enthält. Diesem Knoten folgt ein `differences`-Knoten, der die gefundenen Modelldifferenzen beschreibt. Für dieses XML-Format wurde ein XSLT-Stylesheet entwickelt, das einen XHTML-Bericht generiert.

5.4.3 Zusammenhang

Abbildung 5.7 zeigt das detaillierte System im großen Zusammenhang. Man erkennt die Fassade, welche die einzelnen Komponenten für einen Modellvergleich steuert und in der die ein-

zelnen Zwischenprodukte angedeutet sind.

Es ist möglich, Transformationskomponenten für verschiedene Sprachen zu implementieren. Hierzu muss eine Grammatik zur Verfügung stehen, mit deren Hilfe ein ANTLR-Parser generiert wird. Zusätzlich muss ein `TreeWalker` implementiert werden, der die Informationen aus dem Modellbaum ausliest und die Metarepräsentation füllt. Die Auswertung bleibt davon unberührt, da sie nur auf der Metarepräsentation arbeitet. Ebenso lassen sich weitere Checks implementieren, die zum Beispiel allgemeine Architekturregeln prüfen, oder Metriken implementieren.

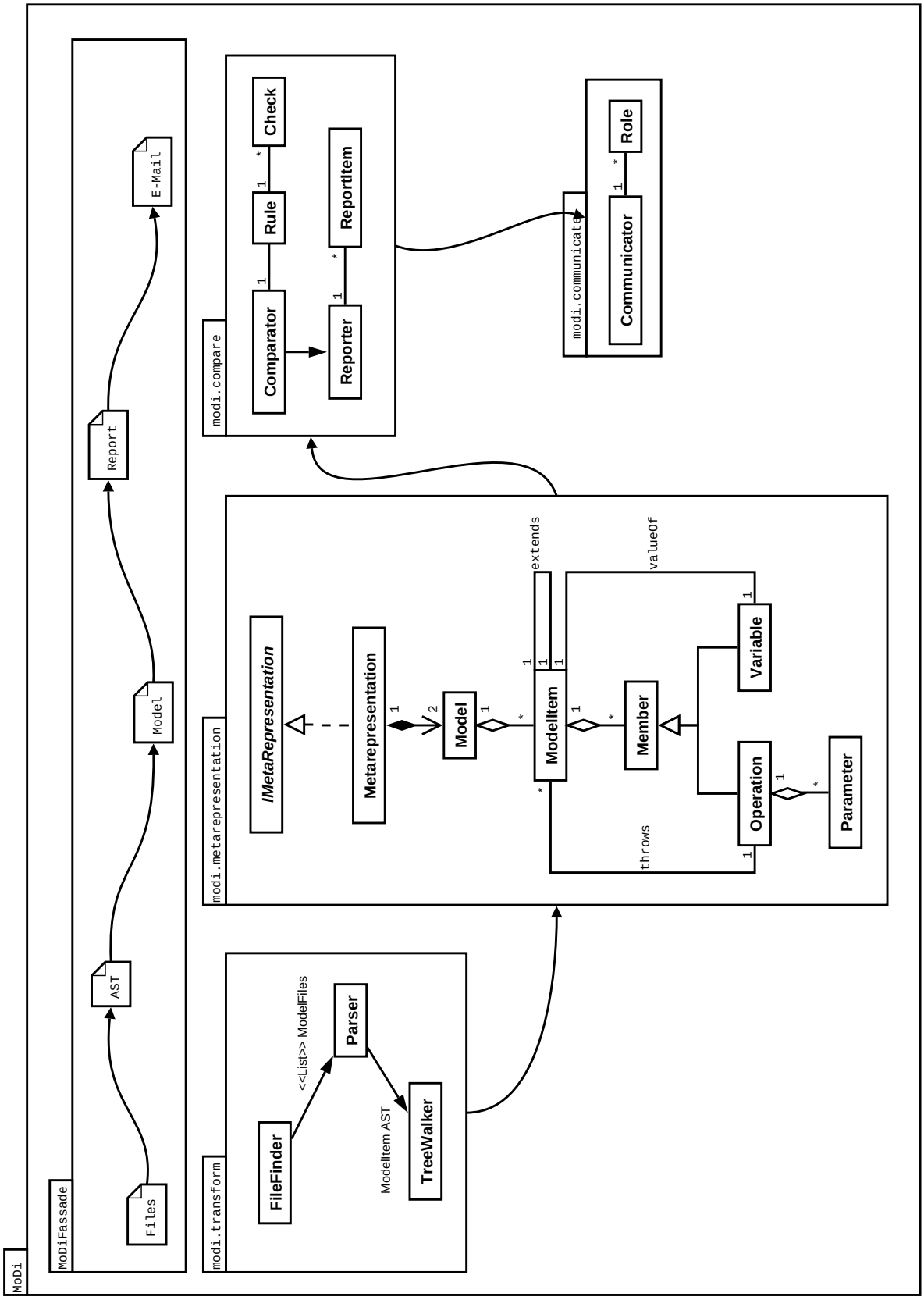

Abbildung 5.7: MoDi in der Übersicht

5.4.4 Erweiterungsmöglichkeiten

Transformationen

Momentan existiert ein Parser für die Sprache Java. Dieser wurde mit Hilfe von [ANTLR] und der Java-Grammatik aus dem Checkstyle-Projekt [Checkstyle] generiert.

Für verschiedene andere Sprachen sind ebenfalls Grammatiken auf [ANTLR] vorhanden, sodass MoDi nicht auf Java beschränkt ist. Um MoDi dahingehend zu erweitern, dass auch Code-Basen in anderen Sprachen verarbeitet werden können, ist es erforderlich, mit einer Grammatik für die Sprache ein ANTLR-Lexer und -Recognizer zu generieren. Diese beiden generierten Komponenten müssen dann durch eine eigene Implmentierung zu einem Parser-System verbunden werden. Anschließend muss ein *Treewalker* entwickelt werden, der die Informationen aus dem AST ausliest und die Metarepräsentation füllt. Hierbei kann sich an den vorhandenen *TreeWalker* `ASTtoMetaRepresentationTransformer` angelehnt werden. Letztlich ist das neu erstellte Parser-System in die `TransformerFactory` und die `language`-Sprachkonstanten einzutragen, damit es im System zur Verfügung steht.

Berichte

Nach dem Modellvergleich steht der Report als Java-Objekt zur Verfügung. Er kann mit entsprechenden Technologien durch entsprechende Reporter in verschiedene Formate serialisiert werden. Momentan ist ein XML-Reporter implementiert, der den entstandenen Bericht im XML-Format serialisiert. Ferner ist ein XHTML-Reporter implementiert worden, der den XML-Bericht mit Hilfe eines XSLT-Stylesheets in XHTML transformiert, sodass sich der Bericht in einem Browser anzeigen lässt. Außerdem existiert noch ein Terminalreporter, der den Bericht auf die Konsole ausgibt.

Kommunikation

Momentan ist die Kommunikation des Berichtes per E-Mail vorgesehen. Hier können aber noch andere Medien genutzt werden. Beispielsweise könnte ein SMS- oder Instant-Messaging-Dienst (z.B. ICQ) die wichtigsten Modelldifferenzen sofort an einen Verantwortlichen senden. Das könnte in Projekten interessant sein, in denen kritische Software entwickelt wird und Modelldifferenzen möglichst frühzeitig erkannt werden sollen. Ferner könnte auch eine Lava-Lampe wie in [Clark2004] den Projektstatus anzeigen.

5.4.5 Konfiguration und Beispiel

In der Konfigurationsdatei werden die auszuwertenden Regeln, Metadaten über das Projekt und die zu benachrichtigenden Rollen gespeichert. Für einen Modellvergleich können beliebig viele Regeln definiert werden. Die Anzahl der sinnvollen Regeln hängt aber sicherlich von der Menge der zur Verfügung stehenden Checks ab. Momentan sind die in Kapitel 5.4.1

beschriebenen Checks implementiert. Diese Checks finden dann in den Regeln Anwendung. Jede Regel hat dabei zusätzlich einen Namen und einen Schweregrad, der aus leicht, mittel und schwer ausgewählt werden kann.

```
MoDi.Rule.1.Name=Are there removed members?
MoDi.Rule.1.Severity=HIGH
MoDi.Rule.1.Checks=MembersRemoved

MoDi.Rule.2.Name=Are there modified interfaces?
MoDi.Rule.2.Severity=MEDIUM
MoDi.Rule.2.Checks=InterfacesModified

MoDi.Rule.3.Name=Are there additional classes?
MoDi.Rule.3.Severity=LOW
MoDi.Rule.3.Checks=ClassesAdded
```

Listing 5.1: Beispielkonfiguration einer Regel

Beispielhaft sind in Listing 5.1 Regeln angegeben, welche die Frage nach gelöschten Membern beantworten, dafür den Schweregrad `HIGH` einstellt und den Check `MembersRemoved` benutzt. Ferner ist die Modifikation von Interfaces auf `MEDIUM` eingestellt und für das Hinzufügen von Klassen ist der Schweregrad `LOW` vergeben worden. Mit dieser Konfiguration wurde ein Modellvergleich durchgeführt. Dabei wurde als AM die generierte Code-Basis (siehe Abb. 3.1, S. 21) aus dem Pizzabringdienst-Beispiel (siehe Abb. 2.1, S. 18) benutzt. Als EM diente eine Kopie der AM Code-Basis mit den folgenden Modifikationen:

1. Aus dem Interface `IPizzaService` wurde die Methode `setAddress` gelöscht. Die Intention dahinter war, dass sich die Adresse auch aus dem übergebenen `ICaller`-Objekt schließen lässt.
2. Dem Modell wurde eine Klasse mit dem Namen `Joeys` hinzugefügt. Diese Klasse implementiert das Interface `IPizzaService`.

Anschließend wurde ein Modellvergleich durchgeführt.

In Abbildung 5.8 (S. 64 ist der entstandene Bericht zu sehen. Es werden zunächst die enthaltenen Metadaten, also das Datum des Berichtes, der Architekt und die Entwickler, sowie die Versionen des AMs bzw. des EMs und die Anzahl der Differenzen zwischen ihnen, dargestellt. Die Anzahl der Differenzen sind zusätzlich noch nach ihrem Schweregrad aufgeschlüsselt, sodass man auf einen Blick erkennen kann, wieviel Differenzen es mit dem Schweregrad `HIGH`, `MEDIUM` und `LOW`, gegeben hat. Zusätzlich sind diese Zahlen mit einem Hyperlink versehen, wodurch es möglich ist, direkt zu den nach ihrem Schweregrad gruppierten Differenzen zu gelangen. Durch einen weiteren Hyperlink ist es möglich, wieder zu der Übersicht zurückzukehren. Innerhalb dieser Gruppen sind schließlich die einzelnen Differenzen nummeriert aufgelistet und es wird eine Beschreibung der Modelldifferenzen angezeigt.

Hier erkennt man nun die Bewertungen der oben beschriebenen Modifikationen (s. 5.4.5) wieder und es ist zu erkennen, dass eine schwere, eine mittlere und eine leichte Regelverletzung

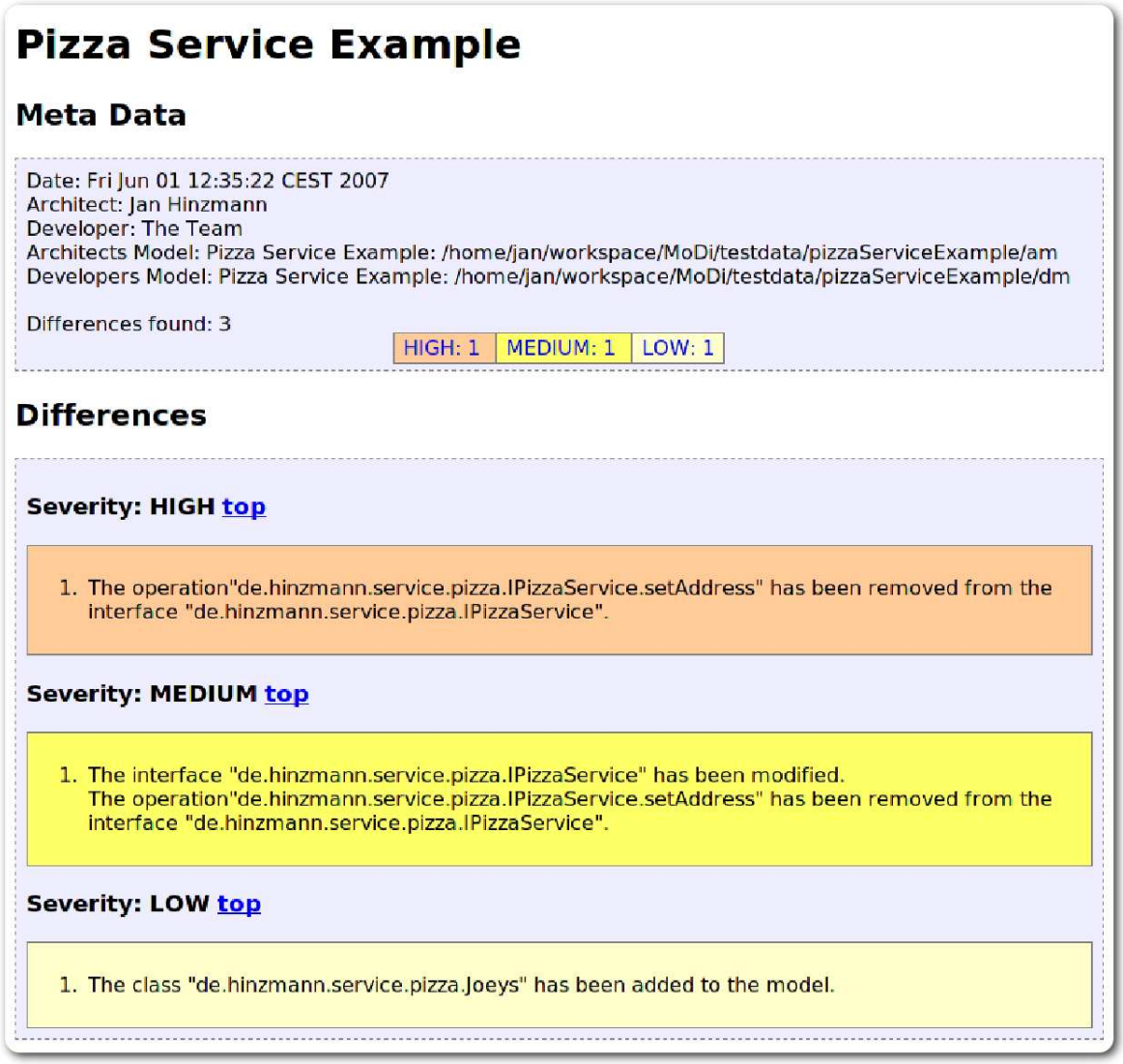

Abbildung 5.8: Ein XHTML-Bericht des Modellvergleichs vom Pizzabringdienst

durch den Modellvergleich erkannt worden ist. Bei der Regelverletzung mit dem Schweregrad `MEDIUM` sieht man beispielsweise, dass das Interface `IPizzaService` modifiziert worden ist und zwar wurde die Methode `setAddress` entfernt. Außerdem wurde die Klasse Joeys hinzugefügt (`LOW`). Ebenfalls sind hier die in Kapitel 3.3 geschilderten vollqualifizierenden Namen zu sehen.

5.5 Zusammenfassung

In diesem Kapitel wurden zunächst die Stakeholder identifiziert und anschließend die Realisierung des Werkzeugs zur Erkennung von Modelldifferenzen MoDi beschrieben. Weiter wurde der Haupt- und vier Unteranwendungsfälle definiert und die Anforderungen an das System beschrieben. Anschließend wurde der Entwurf vorgestellt und die Implementierung der Checks

erläutert. Weiterhin wurde das Gesamtsystem in einem *big picture* gezeigt und auf weitere wichtige Details von MoDi eingegangen. Schließlich wurde anhand des Beispiels über den Pizzabringdienst (siehe Abb. 2.1, S. 18) ein Modellvergleich durchgeführt und die Konfiguration, sowie der aus dem Modellvergleich resultierende Bericht beschrieben. Zuvor wurden einige Änderungen im EM unternommen.

Kapitel 6

Fallstudie

In diesem Kapitel wird beschrieben, wie das MoDi-System an den beiden DLR-SISTEC-Projekten SiLEST und SESIS erprobt worden ist. Das Projekt SiLEST wurde im Jahr 2007 abgeschlossen während das Projekt SESIS bis 2008 geplant ist. Prinzipiell sollte MoDi projektbegleitend eingesetzt werden, jedoch lassen sich auch im Nachhinein Untersuchungen bzgl. Modelldifferenzen anstellen.

6.1 MoDi ausprobiert – das Projekt SiLEST

Die Untersuchungen mit dem SiLEST-Projekt waren schwierig, da in diesem Projekt, aufgrund des dort gelebten Entwicklungsprozesses, keine Architektenmodelle identifiziert werden können. Dennoch konnten Modellvergleiche durchgeführt werden.

6.1.1 Vorgehensweise

Um trotz fehlender Architektenmodelle Aussagen über Modelldifferenzen während des Projektverlaufs treffen zu können, wurden immer zwei aufeinander folgende Revisionen miteinander verglichen.

Als Konfiguration wurden alle Regeln benutzt, die Interfaces betreffen. Zusätzlich wurde der XMLRporter eingeschaltet, um später die Berichte automatisiert auswerten zu können.

Für die Modellvergleiche wurden die Revisionen N_i und N_{i+1} mit $318 < i < 2913$ aus dem Subversion-Repository paarweise ausgecheckt und mit Hilfe von Modi ein Modellvergleich durchgeführt, um einen Bericht über die Differenzen zwischen den Interfaces zu erhalten. Danach wurden beide Revisionen gelöscht. So ergaben sich 2595 Berichte über Modelldifferenzen. Aufgrund von teilweise behebbaren Problemen (s. Abschnitt 6.1.2) war es notwendig, eine Revision zweimal auszuchecken.

Aus den entstandenen Berichten wurde mit Hilfe eines XSLT-Stylesheets die Anzahl der Dif-

ferenzen pro Vergleich aus den Reports ermittelt und zusammen mit der Revisionsnummer des AMs in eine CSV-Datei geschrieben. Mit Hilfe von gezielten Abfragen an das Subversion-Repository wurden zu den Revisionen noch das Datum und der Name des Entwicklers ermittelt, der diese Revision erzeugt hat. Schließlich wurden die entstandenen Modelldifferenzen aggregiert und so die Modelldifferenzen pro Tag ermittelt wurden.

Bevor nun die Ergebnisse dargestellt werden sollen, wird auf die Probleme eingegangen, die sie bei dieser Untersuchung ergaben.

6.1.2 Probleme bei der Durchführung

Beim Durchlauf durch die Revisionen ergab sich das Problem, dass manche Dateien von MoDi nicht verarbeiten werden konnten. Zum einen kann das ANTLR-Parser-System Dateien nicht verarbeiten, die keine Leerzeile am Ende der Datei enthalten. Dies ist durch [JSL] begründet eine Leerzeile am Ende der Datei vorschreibt. Dieses Problem konnte durch einen Workaround umgangen werden: Das ANTLR-System wirft eine Exception, sodass an die betreffende Datei eine Leerzeile angehängt und anschließend die Datei erneut geparst werden kann. Wenn danach allerdings noch ein Fehler auftritt, kann das System keine sinnvollen Aktionen mehr durchführen, sodass die Anwendung beendet wird. Nach Einführung dieses Workarounds konnte die Untersuchung ab Revision 613 (als der Fehler zum ersten Mal aufgetreten ist) fortgesetzt werden.

Im weiteren Verlauf kam es allerdings erneut zu Problemen, da Code eingecheckt worden ist, der nicht kompilierbar ist. Dies verstößt gegen Folgerung 3.3 aus Kapitel 3.2.5. Glücklicherweise kam dieses Problem nur in den Revisionen 1283 und 1284 vor, welche deshalb aus der Untersuchung ausgeschlossen wurden.

In Listing 6.1, Zeile 31 bis 43 ist die betreffende Stelle zu sehen, die scheinbar aufgrund eines nicht entdeckten Merge-Konfliktes entstanden ist. Weiterhin ist zu sehen, dass nach Zeile (50) der nach §3.7 [JSL] erforderliche Zeilenumbruch bei sog. *EndOfLineComments* fehlt.

```
1  //
2  // Copyright (C) 2005 DLR/SISTEC, Germany
3  // All rights reserved
4  //
5  // filename Commitable.java
6  // @author  23.03.2006 **********
7  // @version $Id$
8  //
9  package de.silest.versioncontrol;
10
11 /**
12  * this interface specify the commitable content
13  */
14 public abstract class Commitable {
15
16     /**
17      * represents the commitable type
18      */
19     private int myType;
```

```
20
21      /**
22       * file name
23       */
24      private String myFileName;
25
26      /**
27       * Version
28       */
29      private Version myVersion;
30
31  <<<<<<< .mine
32      /**
33       * default constructor
34       * @param fileName
35       *        filename
36       */
37  =======
38      /**
39       * constructor
40       * @param fileName
41       *        file name
42       */
43  >>>>>>> .r1279
44      protected Commitable(final String fileName) {
45          myFileName = fileName;
46      }
47  }
48
49  // modification list
50  // $Log$
```

Listing 6.1: Ausschnitt aus der nicht kompilierbaren Datei

Während das Problem der nicht kompilierbaren Dateien nicht behoben werden kann, konnte das Problem des fehlenden Zeilenumbruchs dadurch umgangen werden, dass die auftretende `Exception` gefangen, eine Leerzeile an die Datei angehängt und ein erneutes Parsen der betroffenen Datei veranlasst wurde. So konnte die Fallstudie über alle bis auf zwei Revisionen (1283, 1284) durchgeführt werden.

Im Folgenden werden nun die Ergebnisse der Untersuchung beschrieben.

6.1.3 Ergebnisse – Auswertung der Projektereignisse

In Abbildung 6.1 sind die erkannten Modelldifferenzen pro Tag aufsummiert und über die Zeit aufgetragen. Den Modelldifferenzen wurden anschließend Projektereignisse zugeordnet. Der Zeitraum der Untersuchung reicht vom Dezember 2005 bis zum April 2007.

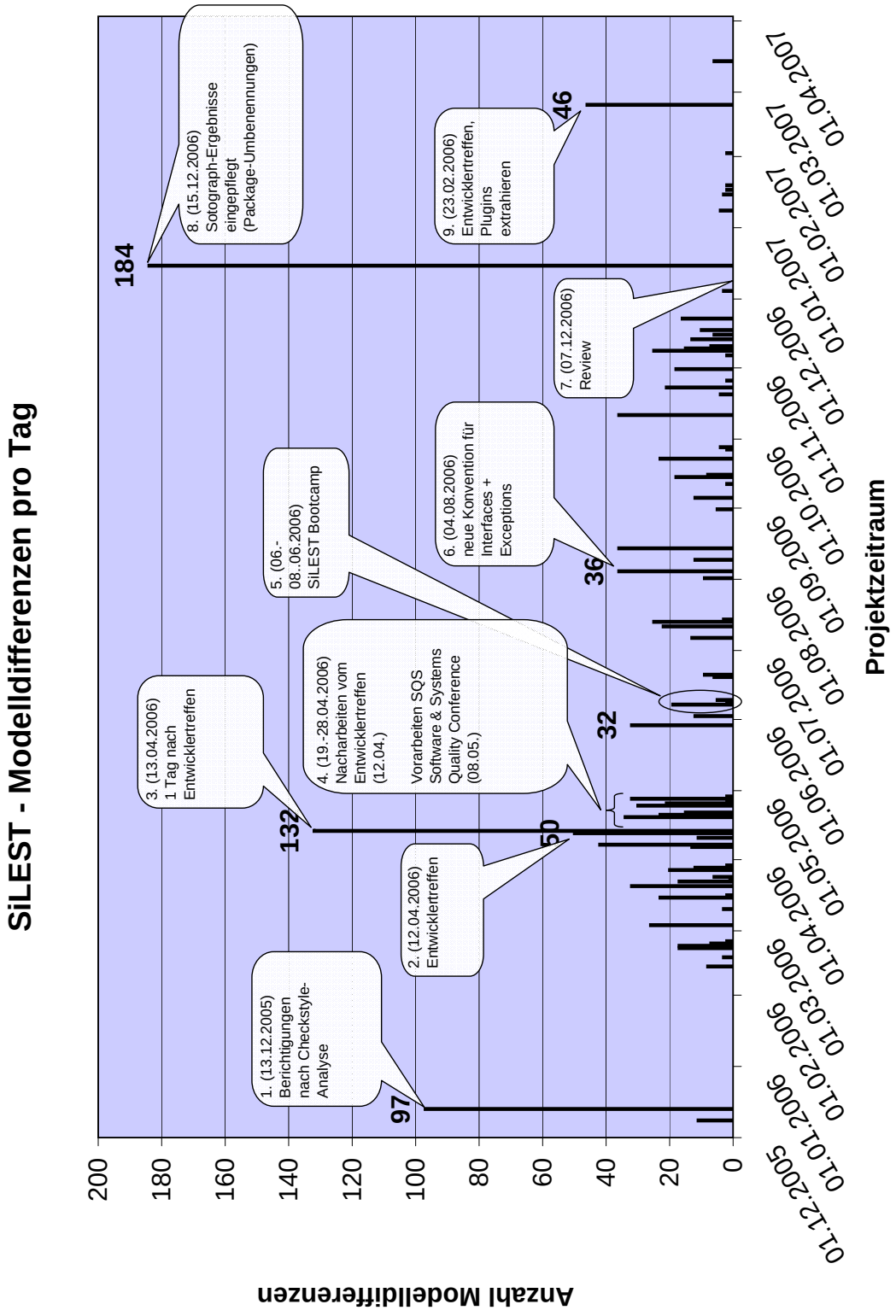

Abbildung 6.1: Projektereignisse und Differenzen im SiLEST-Projekt

Durch Vergleiche der Revisionen mit den Subversion-Logeinträgen der Entwickler und der Dienstreiseabrechnungen der Mitarbeiter konnten den Modelldifferenzen neun Projektereignisse zugeordnet werden, welche hier im Einzelnen beschrieben werden:

1. Am 13.12.2005 wurden nach einer Code-Analyse mit dem Werkzeug [Checkstyle], welches Schwachstellen im Code erkennt, Anpassungen im Code vorgenommen und so sind 97 Modelldifferenzen entstanden. Die Analyse beinhaltet Coding-Standards, aber auch weiterführende Untersuchungen, wie zum Beispiel: „class design problems, duplicate code, or bug patterns like double checked locking“.

2. Am 12.04.2006 fand eines der quartalsweise durchgeführten Entwicklertreffen (vergl. 4.1 Aktivität 16) am Standort Berlin Adlershof statt. Bei einem Entwicklertreffen kommen alle am Projekt beteiligten Entwicklerteams zusammen und diskutieren den momentanen Stand des Projektes. Dabei werden Wünsche und Probleme im Projekt erörtert und Designentscheidungen herbeigeführt. Ferner werden die Buglisten aus dem Bugtracking-System durchgegangen und die einzelnen Aufgaben an Entwickler verteilt. In Diagramm 6.1 ist zu sehen, dass die Anzahl der Modelldifferenzen zum Entwicklertreffen hin zunimmt und während bzw. kurz danach ihr Maximum erreicht (50 bzw. 132 Differenzen).

3. Die Verzögerung von einem Tag ist darin begründet, dass ein Entwickler direkt nach dem Treffen seine Änderungen eingepflegt hat (50 Modelldifferenzen) und ein anderer erst die Rückreise zum Standort Braunschweig angetreten ist (135 Modelldifferenzen).

4. Bei den hier zu sehenden Modelldifferenzen handelt es sich um Nachbearbeitungen des Entwicklertreffens, aber auch um Vorbereitungen für die Konferenz *Software & Systems Quality* (SQS), auf der das Projekt SiLEST vorgestellt wurde. Für die Demonstration auf der Konferenz war es notwendig, die Software auf einen gewissen Stand zu bringen.

5. Vom 06. bis zum 08.06.2006 fand ein SiLEST-Bootcamp, also ein dreitägiges Entwicklertreffen statt, auf dem – wie bei den anderen Entwicklertreffen auch – gezielt Probleme gemeinsam angegangen worden sind. Die wesentlichen Ziele des Bootcamps waren, eine Demonstration zu erstellen und weiterhin die Lauffähigkeit auf verschiedenen Betriebsystemen inklusive „WebFrontend“ zu gewährleisten. Weiterhin sollte sichergestellt werden, dass die eingesetzte Plugin-Technologie von allen beteiligten Projektpartnern verstanden wird und alle in der Lage sind, selber Plugins zu implementieren. Außerdem galt es, Qualitätsmerkmale festzulegen und die Konfigurierbarkeit des Systems zu verbessern. Anschließend wurde der Quelltext durchgegangen und `FIXME`-Tags und Bug-Fixes kontrolliert und soweit wie möglich entfernt. Schließlich wurden Integrationstests festlegt.

6. Am 04.08.2006 wurden neue Konventionen für Interfaces und Exceptions beschlossen und umgesetzt, was 36 Modelldifferenzen nach sich zog.

7. Am 07.12.2006 fand ein Review statt. Bei diesem Entwicklertreffen wurden anscheinend keine Änderungen in der Architektur beschlossen, sodass es zu keinen sichtbaren Modelldifferenzen gekommen ist.

8. Vor dem 15.12.2006 im Rahmen einer Vorführung des Werkzeugs Sotograph [BischofbergerKL2004] wurde SiLEST einer Qualitätsanalyse unterzogen. Die Ergebnisse wurden dann eingepflegt und so kam es zu 184 Modelldifferenzen. Dabei wurden

zwei Pakete umbenannt, wodurch eine große Anzahl von Modelldifferenzen entstanden sind, da Umbenennungen prinzipiell als Löschen und Hinzufügen behandelt werden. Somit wurden alle Interfaces aus den umbenannten Paketen gelöscht und in ein neues Paket hinzugefügt.

9. Am 23.02.2006 fand ein Entwicklertreffen statt, bei dem beschlossen wurde, Plugins zu extrahieren. Dies hatte 46 Modelldifferenzen zur Folge.

Die Anzahl der Modelldifferenzen ist also mit Vorsicht zu bewerten. Beispielsweise ergab die Umbenennung von wenigen Paketen eine große Anzahl von Differenzen. Das zeigt an, dass es eine für die Architektur „große“, also bedeutende Änderung war, jedoch nicht, dass besonders viel geändert worden ist.

6.2 MoDi ausprobiert – das Projekt SESIS

Bei dem Projekt in der zweiten Fallstudie sind Architektenmodelle vorhanden und der Einsatz von MoDi konnte also nachträglich simuliert werden.

Im dem Verbundprojekt wird unter anderem das Framework RCE (Reconfigurable Computing Environment) entwickelt, welches als Basissystem für SESIS, einen Schiffentwurfs- und Simulationssystem, eingesetzt wird. Auf der Webseite des Projektes [SESIS] ist sein Ziel folgendermaßen beschrieben:

> Ziel von SESIS ist es, ein integriertes schiffbauliches Entwurfs- und Simulationssystem mit dem Fokus auf der frühen Entwurfsphase, in der die wesentlichen Parameter eines Schiffneubaus festgelegt werden, zu entwickeln. SESIS unterstützt insbesondere die Zusammenarbeit von Werft und Zulieferunternehmen und ist eine integrierte Arbeitsumgebung für den Schiffbauingenieur, die auf einer modernen Softwarekonzeption basiert und bewährte schiffbauliche Entwurfsmethoden nutzt.
> –

Abbildung 6.2: Ziele des Verbundprojektes SESIS

Die Partner in diesem Projekt sind

1. die Flensburger Schiffbau-Gesellschaft (FSG,Flensburg),
2. die Lindenau GmbH (Schiffswerft und Maschinenfabrik, Kiel),
3. die SAM Electronics GmbH (Hamburg),
4. das Center of Maritime Technologies (CMT,Hamburg),
5. die Technische Universität Hamburg-Harburg,
6. das Deutsche Zentrum für Luft- und Raumfahrt (DLR, Köln) und

7. das Fraunhofer-Institut für Algorithmen und Wissenschaftliches Rechnen (SCAI, Sankt Augustin).

Das DLR stellt den Softwarearchitekten, der freundlicherweise sieben Modelle für eine Untersuchung mit dem MoDi-System zur Verfügung gestellt hat. Die Revisionen 1561, 1879, 1922, 1928, 1942, 1961 und 1967 stellen Modelle einer Komponente des RCE-Frameworks dar. Hierbei kann die Revision 1561 als initiales Architektenmodell und die restlichen Revisionen als Entwicklermodelle angesehen werden. Die entstandenen Entwicklermodelle wurden manuell überprüft und als Architektenmodell übernommen. Aus der Erfahrung im Projekt ist zu erwarten, dass es zwischen den ersten vier Revisionen moderate Differenzen gibt und es bei den beiden letzten Revisionen zu schwerwiegenden Differenzen, da dort Klassen gelöscht worden sind.

6.2.1 Vorgehensweise

In dieser Fallstudie wurden jeweils zwei benachbarte Revisionen miteinander verglichen, um den Entwicklungsprozesses unter Einsatz von MoDi nachzustellen. Hierbei diente die Vorgänger-Revision als AM und die nachfolgende als EM.

Konfiguration

Bei den Modellvergleichen wurden die folgenden Regeln angewendet, deren Bewertung auf den Ergebnissen aus Kapitel 3.6 basieren und für dieses Projekt mit dem Architekten abgestimmt worden ist. In Tabelle 6.1 sind die Regeln für den Modellvergleich dargestellt:

Nach Kapitel 3.6.1, Regel 2 dürfen Interfaces hinzugefügt, aber nicht gelöscht (Kapitel 3.6.1, Regel 1) werden. Das Modifizieren von Interfaces wird als mittelschwer bewertet.

Weiterhin dürfen keine Member – also Operationen oder Konstanten – gelöscht werden (Kapitel 3.6.3, Regel 1). und das Modifizieren wird wiederum als mittelschwer bewertet. Hier wird von der Empfehlung aus Kapitel 3.6.3, Regel 3 abgewichen.

Das Hinzufügen oder Löschen von Parametern wird als schwerwiegend bewertet, da dies die Methoden derart verändern würde, dass sie von den anderen Programmteilen ohne Anpassung nicht mehr verstanden werden können.

Die weiterführende Regel 1 aus Kapitel 3.6.4 detektiert öffentliche Methoden in Klassen, die nicht durch eine zugehörige Interfacedefinition abgedeckt werden. Die wird als schwerwiegend bewertet. Die Regel 2 aus Kapitel 3.6.4 schließlich findet private Methoden in den vorhandenen Klassen und wird, mangels Schweregrad `GOOD`, mit `LOW` bewertet. Schließlich findet die Regel 11 die hinzugefügten Klassen und bewertet dies ebenfalls mit `LOW`.

Interface

#	Name	Checks	Severity
1	Are there additional interfaces?	InterfacesAdded	LOW
2	Do NOT remove interfaces!	InterfacesRemoved	HIGH
3	Check for modified interfaces	InterfacesModified	MEDIUM

Member

#	Name	Checks	Severity
4	Are there additional members?	MembersAdded	MEDIUM
5	Are there removed members?	MembersRemoved	HIGH
6	Are there modified members?	MembersModified	MEDIUM

Parameter

#	Name	Checks	Severity
7	Are there additional parameter?	ParameterAdded	HIGH
8	Are there removed parameter?	ParameterRemoved	HIGH

Extra

#	Name	Checks	Severity
9	Do not add pulic methods to classes, add them to the interface too	AddedPublicMethods-InImplementingClasses	HIGH
10	Are there additional private methods in classes?	AddedPrivateMembersIn-Classes	LOW
11	Are there additional classes?	ClassesAdded	LOW

Tabelle 6.1: Regelsatz für die Untersuchung von SESIS

Vergleich zweier benachbarter Revisionen

Im Verlauf des Projektes wurden die entstandenen Entwicklermodelle geprüft und anschließend als neues AM übernommen. Dies ist dem natürlichen Weg nachempfunden, wenn das MoDi-System projektbegleitend eingesetzt wird. Geht man also von diesem Vorgehen aus, so muss zunächst das initiale AM mit der Revisionsnummer 1561 mit dem Nachfolger (1879) verglichen werden. Anschließend wird dann die Revision 1879 mit der Revision 1922 verglichen und dies setzt sich fort, bis schließlich Revision 1961 mit 1967 verglichen wird.

Mit Hilfe eines Skriptes `report` werden automatisiert die sechs Modellvergleiche erstellt. Hierbei werden die einzelnen Berichte als HTML-Dateien gespeichert. Hierzu werden die Konfigurationsdatei und die beiden Ordner des Architektenmodells bzw. des Entwicklermodells als Parameter an MoDi übergeben. Der Aufruf, wie er für die sechs Modellvergleich nötig ist, sei hier beispielhaft für den ersten Modellvergleich angegeben:

```
java -jar modi.jar modi.conf revision1561 revision1879 && \
mv modireport.html modireport1561-1879.html
```

Anmerkung: Das Projekt SiLEST aus der vorherigen Fallstudie besteht aus wesentlich mehr Dateien. So hat beispielsweise die Revision 2444 alleine 330 Java-Dateien, in denen insgesamt 61.457 Zeilen enthalten sind. In dieser Fallstudie wird nur eine Teilkomponente des RCE-Frameworks untersucht, die im Durchschnitt aus 11 Dateien und 862 Zeilen besteht (siehe Tabelle 6.2). Deshalb ist die Anzahl der Modelldifferenzen insgesamt auch kleiner. Anhand dieser einfachen Metrik lässt sich erkennen, dass von Revision 1942 zu 1961 eine Datei (RCEBundleInstanceTable.java) gelöscht worden ist.

Revision	Zeilen	Dateien
1561	180	4
1879	315	6
1922	621	10
1928	837	12
1942	1309	14
1961	1258	13
1967	1516	15

Tabelle 6.2: Einfachste Metrik der RCE-Komponente

6.2.2 Ergebnisse – Auswertung der Berichte

Nachdem nun alle Berichte erstellt worden sind, können die im HTML-Format vorliegenden Dateien mit einem Browser geöffnet und ausgewertet werden. Hierzu werden die einzelnen Differenzen bezüglich ihres Schweregrades unterschieden und gezählt. Die einzelnen Werte für den Schweregrad `HIGH`, `MEDIUM` und `LOW`, sowie die Summe der drei Schweregrade sind in Tabelle 6.3 zusammengefasst:

Revision AM-EM	ALL (A)	HIGH (H)	MEDIUM (M)	LOW (L)
1561-1879	10	2	4	4
1879-1922	11	2	5	4
1922-1928	9	1	2	6
1928-1942	24	6	8	10
1942-1961	29	12	11	6
1961-1967	14	3	6	5

Tabelle 6.3: Ergebnisse des Vergleichs zweier benachbarter Revisionen

Wie bereits aus der Erfahrung im Projekt zu erwarten war, ist zu sehen dass es bei den ersten drei Modellvergleichen zu moderaten Modelldifferenzen gekommen ist. Die Modellvergleiche zwischen den Revisionen 1928-1942 und 1942-1961 hingegen zeigen deutlich mehr Modelldifferenzen mit der Bewertung `MEDIUM` und einen Anstieg der schwerwiegenden Modelldifferenzen. Dies wurde erwartet, da die Klasse `RCEBundleInstanceTable` gelöscht worden sind.

In Abbildung 6.3 sind die Modellvergleiche noch einmal graphisch dargestellt:

Insgesamt ist zu sehen, dass die Modelldifferenzen mit dem vierten Modellvergleich drastisch

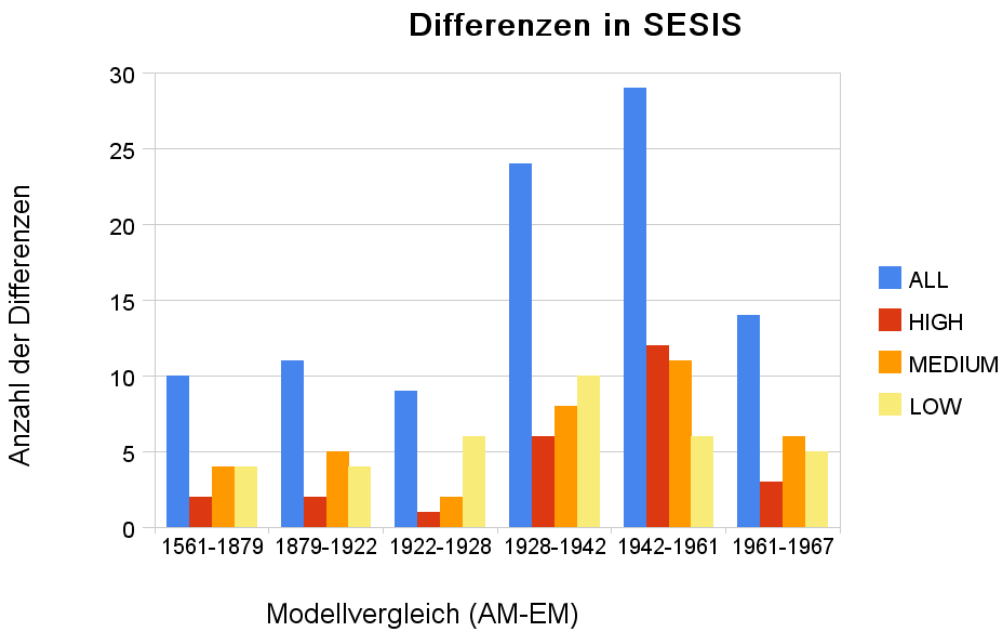

Abbildung 6.3: Ergebnisse der Vergleiche zwischen zwei Modellen

zugenommen haben, im fünften ihr Maximum erfahren und danach wieder abnehmen.

Die Anzahl der Modelldifferenzen insgesamt und aufgeschlüsselt nach dem Schweregrad geben nur einen groben Anhaltspunkt, wie sich das Softwareprojekt entwickelt. Welche Modelldifferenzen zwischen den einzelnen Revisionen aufgetreten sind lässt sich aus den generierten Berichten detailliert entnehmen. Im Folgenden sollen einzelne interessante Modelldifferenzen aus den Berichten besprochen werden.

Modellvergleich 1561-1879: Beim Einchecken des EMs der Revision 1879 manifestierten sich 10 Modelldifferenzen, davon zwei mit dem Schweregrad `HIGH`, vier mit `MEDIUM` und vier mit dem Schweregrad `LOW`.

Dabei wurde die Operationen `sendCommunicationObject` aus dem `Communicator`-Interface und die Operationen `send` aus `Communication`-Interface gelöscht.Aus den Modelldifferenzen, die mit dem Schweregrad `MEDIUM` bewertet wurden, kann entnommen werden, dass die Operation `send` – wie oben bereits gesehen – aus dem `Communication`-Interface gelöscht wurde. Es ist allerdings zusätzlich zu sehen, dass die Operation `send` dem `Communicator`-Interface hinzugefügt worden ist. Die `send`-Methode wurde offenbar von dem einem Interface in das andere verschoben. Schließlich sind vier Modelldifferenzen mit dem Schweregrad `LOW` erkannt worden, da die Klassen `CommunicationBundleActivator` und `RMICommunicator` dem Entwicklermodell hinzugefügt worden sind. Außerdem wurde die Konstante `ERROR_INIT_COMM_BUNDLE` und die Operation `instance` der Klasse `CommunicationImpl` hinzugefügt.

Leider sind der Lognachricht zu der Revision 1879 keine Gründe für die aufgetretenen Modelldifferenzen zu entnehmen.

Modellvergleich 1879-1922: Nach dem Einchecken der Revision 1879 konnten insgesamt 11 Modelldifferenzen (`HIGH`: 2, `MEDIUM`: 5, `LOW`: 4) erkannt werden. Die vier Modelldifferenzen mit dem Schweregrad `LOW` wurden gefunden, da die Klassen `RMIBundleActivator`, `RMIServerImpl` und `RMIClient`, sowie das Interface `RMIServer` dem EM hinzugefügt worden sind.

Aus der Lognachricht *Simple RMI Communication works.* ist zu erkennen, dass jetzt die *Remote Method Invokation* (RMI) funktioniert.

Modellvergleich 1922-1928: Durch die Revision 1928 wurden neun Modelldifferenzen (H: 1, M: 2, L: 6) zur Vorgänger-Revision erzeugt. Die Operation `send` wurde aus dem Interface `Communication` gelöscht. Dies ist darin begründet, dass sich die Signatur der Methode, geändert hat, was einem Löschen und Hinzufügen der Operation gleichkommt. Mit Hilfe des GNU-Diff Werkzeugs [Myers1986] lässt sich zeigen, dass sich ihr Rückgabewert geändert hat.

```
<     CommunicationObject send(CommunicationObject communicationObject);
---
>     Future<CommunicationObject> send(CommunicationObject communicationObject);
```

Diese Modifikation ist auch den Modelldifferenzen mit dem Schweregrad `MEDIUM` zu entnehmen. Weiterhin ist zu erkennen, dass die Klassen `CommunicationHandler` und `RMIConfiguration` dem EM hinzugefügt worden sind. Ferner wurde der Klasse `CommunicationImpl` um die Variable `myThreadPool` erweitert, und der Klasse `CommunicatorFactory` wurden die Member `RMI_BUNDLE`, `RMI_COMMUNICATOR` und `logger` hinzugefügt.

Modellvergleich 1928-1942: Bei diesem Modellvergleich sind insgesamt 24 Modelldifferenzen aufgetreten, davon sechs schwerwiegende, acht mit dem Schweregrad `MEDIUM` und zehn mit `LOW`.Durch diesen Commit wurde der Operationen `call` aus dem Interface `RMIServer` der neue Parameter `returnObjectId` hinzugefügt. Außerdem wurden vier Klassen dem Entwicklermodell hinzugefügt, was durch die folgenden Meldungen in der Kategorie `LOW` ersichtlich ist:

1. The class „de.rcenvironment.rce.communication.ObjectInstanceTable“ has been added to the model.
2. The class „de.rcenvironment.rce.communication.MethodCaller“ has been added to the model.
3. The class „de.rcenvironment.rce.communication.RCEBundleInstanceTable“ has been added to the model.
4. The class „de.rcenvironment.rce.communication.CommunicationCallable“ has been added to the model.

Schließlich wurde das EM noch um sechs private Member erweitert.

Modellvergleich 1942-1961: Bei diesem Modellvergleich ergaben sich die meisten (29) Modelldifferenzen (H: 12, M: 11, L: 6). In diesem Commit gab es acht Parameter- und in vier Operationenmodifikationen, wodurch die 12 schwerwiegenden Modelldifferenzen erklärt sind. Dies zog die als Modifikationen von Interfaces erkannten Modelldifferenzen mit dem Schweregrad `MEDIUM` nach sich. Schließlich wurden fünf Variablen und eine private Methode dem Entwicklermodell hinzugefügt. Das Löschen der Klasse `RCEBundleInstanceTable` wurder allerdings nicht erkannt, da nur das Löschen von Interfaces beobachtet wurde (Tabelle 6.1 Regel 2). Diese Modelldifferenz hätte mit dem Check `ModelItemsRemoved` erkannt werden können.

Modellvergleich 1961-1967: Bei diesem Modellvergleich ergaben sich insgesamt 14 Modelldifferenzen (H: 3, M: 6, L:5). Es wurde die Operation Interface `Communicator` gelöscht. Außerdem wurde der Parameter `communicationRoute` zu der Operation `call` des Interfaces `RMIServer` hinzugefügt. In diesem Commit wurden die Konstante `NOT_YET_INITIALIZED_ERROR` zu der Klasse `ObjectInstanceTable` hinzugefügt. Neben den privatem Variablen `myRoutingTable` in der Klasse `CommunicationBundleActivator` und `myRoute` in der Klasse `RMICommunicator` wurden auch die Klassen `RoutingTable` und `CommunicationServer` EM hinzugefügt.

Fazit

Gerade die schwerwiegenden Regelverletzungen können durch den projektbegleitenden Einsatz von MoDi frühzeitig erkannt werden. Hier kann der Projektleiter die Entwicklermodelle ablehnen und so dafür sorgen, dass die Architektur richtig umgesetzt bzw. an die neuen Gegebenheiten angepasst wird.

6.3 Zusammenfassung

In diesem Kapitel wurde beschrieben, wie das System MoDi anhand zweier Fallstudien erprobt worden ist. Hierbei wurden die beiden Projekte SiLEST und SESIS des Deutschen Zentrums für Luft- und Raumfahrt untersucht. Bei dem abgeschlossenen Projekt SiLEST konnten auf Grund des dort gelebten Prozesses im Nachhinein keine Architektenmodelle identifiziert werden. Deshalb wurden über den gesamten Projektverlauf immer zwei benachbarte Revisionen miteinander verglichen. Mit Hilfe der Logeinträge aus dem Repository, Dienstreiseabrechnungen und Protokollen von Besprechungen konnten den Modelldifferenzen Projektereignisse zugeordnet werden. Außerdem wurde so erkannt, dass die Anzahl der Modelldifferenzen mit der Größe des Eingriffs in die Architektur zusammenhängt. So entstanden durch wenige Paketumbennung eine beträchtliche Anzahl von Differenzen.

Bei dem zweiten Projekt SESIS konnten Architektenmodelle identifiziert werden. Es es konnte der vorgeschlagene Prozess, dass ein EM gegen ein AM verglichen wird und anschließend als neues AM übernommen wird nachempfunden werden, da die EMs manuell gegen das AM geprüft worden sind. Mit Hilfe vom MoDi wäre diese Prüfung sicherlich einfacher gewesen.

Kapitel 7

Verwandte Arbeiten

In diesem Abschnitt sollen verwandte Arbeiten besprochen und gegen die eigene abgegrenzt werden. Die sich ergebenen Synergie-Effekte, können für weitere Arbeiten genutzt werden.

7.1 Wissenschaftliche Untersuchungen

UMLDiff: An Algorithm for Object-Oriented Design Differencing:

In [XingS2005] wird ein Algorithmus vorgestellt, mit dessen Hilfe sich automatisch strukturelle Änderungen zwischen verschiedenen Versionen einer objektorientierten Software entdecken lassen. Das Ziel ist es die Beweggründe für Software-Evolution herauszufinden. Hierbei werden aus zwei Code-Basen durch Reverse-Engineering Klassenmodelle in Form von abstrakten Syntaxbäumen erstellt. Diese werden mit Hilfe des Visitor-Patterns [GHJV1995] traversiert und die gewonnene Information in einer PostgreSQL-Datenbank gespeichert. Basierend auf diesen *ground facts* lassen sich anschließend verschiedene *derived facts* ableiten. Die abgeleiteten Fakten werden in Datenbank-Sichten (Views) abgelegt. Häufig abgefragte Relationen werden allerdings aus Performance-Gründen in eigenen Tabellen gespeichert. Um eine Vererbungshierarchie darstellen zu können muss die transitive Hülle der betreffenden Klassen (respektive Interfaces) berechnet werden. Hierfür wurde der PostgreSQL-Server erweitert, da keine rekursiven Berechnungen möglich waren. Die Zuordnung von korrespondierenden Modellelementen findet in UMLDiff anhand einer *name similarity* und einer *structure similarity* statt. Diese Heuristiken ermöglichen es Umbenennungen zu erkennen, werden aber als sehr Zeit-intensiv geschildert.

Das vorgestellte Werkzeug soll alle Differenzen finden. Mit dem MoDi-System können selektiv bestimmte Modelldifferenzen gesucht und bewertet werden. Ein wesentliches Ziel dieser Arbeit ist es, Kommunikation über Modelldifferenzen anzuregen. Das Ziel von [XingS2005] ist es Software-Evolution zu untersuchen.

Während der beschriebene Ansatz die Vergleiche mittels Heuristiken vornimmt, beruhen die Modellvergleiche von MoDi auf vollqualifizierenden Namen, was einen Geschwindigkeitsvorteil

für MoDi mit sich bringt. Umbenennungen werden in MoDi als zusammengesetzte Operation von Löschen und Hinzufügen betrachtet. Betrachtet man Interfaces, so ist eine Umbenennung ein schwerwiegender Eingriff in die Architektur (s. Regel 3 S. 36) bzw. [Fowler1999] (S.64), da Software Dritter dann nicht mehr funktionieren kann, ist der vermeintliche Vorteil zu vernachlässigen. Während sich UMLDiff dazu eignet Post-Mortem-Analysen im Bereich der Software-Evolution durchzuführen, ist MoDi eher für den Einsatz in der Entwicklungsphase des Software-Lebenszyklus gedacht. Dort soll es im Daily-Build eingesetzt werden (Bedingung 4.1). UMLDiff verwendet mehrere Technologien (PostgreSQL, Java). MoDi ist ausschließlich in Java implementiert, sodass die kognitive Last, das System kennenzulernen bei MoDi kleiner ist. Außerdem lassen sich rekursive Berechnungen in Java leicht implementieren, sodass die oben erwähnten Schwierigkeiten bei der Erstellung der Vererbungshierarchie nicht gegeben sind.

Das in der Fallstudie von [XingS2005] erwähnte JFreeChart mit über 800 Klassen und ca 200.000 Zeilen Code konnte mit der Konfiguration aus der SESIS-Fallstudie in einer Minute verarbeitet werden. Beim Modellvergleich zwischen dem Release 1.0.4 und 1.0.5 ergaben sich 178 Modelldifferenzen. Davon wurden 91 mit dem Schweregrad `HIGH`, 48 mit `MEDIUM` und 39 mit `LOW` bewertet. Dies sei nur der Vollständigkeit halber erwähnt, da es nicht mit der obigen Studie verglichen werden kann, wo 31 Releases in 3 Stunden verarbeitet wurden und wesentlich mehr Information extrahiert worden ist. Für den Vergleich zweier Versionen von JFreeChart wird eine mittlere Verarbeitungsdauer von 10-12 Minuten angegeben.

A Differencing Algorithm for Object-Oriented Programs:

In [ApiwattanapongOH2004] wird ein Algorithmus zur Erkennung von Differenzen zwischen Objekt-Orientierten-Programmen vorgestellt, der auch korrespondierende Anteile zwischen zwei Versionen des Programms identifiziert. Der beschriebene Algorithmus wurde in einem Werkzeug JDiff für Java-Programme umgesetzt und seine Fähigkeiten wurden in einer empirischen Studie ähnlich der in Kapitel 6.1 beschriebenen untersucht. Der Algorithmus klassifiziert Programmelemente als *added, deleted, modified, unchanged* und diese Information soll helfen, die Kosten für das wiederholte Ausführen von Performance-Tests zu senken, da nur die modifizierten Anteile getestet werden. Der Algorithmus `CalcDiff` findet Paare korrespondierender Elemente aus dem zu vergleichenden Programmen. Beginnend auf der Klassen- bzw. Interface-Ebene werden Klassen anhand ihres vollqualifizierenden Namens einander zugeordnet. Anschließend werden auf der Methoden-Ebene Paare durch Vergleich der Methoden-Signatur gesucht. Danach wird versucht, nicht zugeordnete Methoden durch ihren Namen zu identifizieren. So können Parameter, die hinzugefügt oder gelöscht worden sind bestimmt werden. Auf der *node*-Ebene schließlich werden Kontrollflussgraphen (CFG) des Methodenrumpfes erstellt und so das Verhalten der Software untersucht.

Das Vorgehen von JDiff weicht von der in dieser Arbeit vorgestellten Methode insofern ab, als das es ab der Methoden-Ebene auf Kontrollflussgraphen arbeitet, während MoDi zunächst ein ganzheitliches Modell (Metarepräsentation) der Software aufbaut. Anschließend werden in MoDi selektiv Untersuchungen auf der Metarepräsentation durchgeführt und die Ergebnisse bewertet. Ziel von JDiff ist es, die Kosten von Regressions- und Performance-Tests zu senken, wohingegen das Ziel von MoDi eine Steigerung der Kommunikation ist.

ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links

In [MurtaHW2006] wird versucht die *Traceability* von Architektur und Implementierung durch sog. *traceability links* zu gewährleisten. Diese Links sollen kontinuierlich pro Commit anhand von konfigurierbaren Richtlinien überprüft und erneuert werden. Das dafür entwickelte Werkzeug ArchTrace basiert auf Java, xADL, einer Beschreibungssprache für Architekturen und Subversion.

Mit dem Werkzeug ArchTrace werden Architekturelemente Implementierungselementen zugeordnet. Dies geschieht in kleinen Schritten pro Commit, sodass die *traceability links* immer aktuell gehalten werden. Dies käme einem Einsatz von MoDi in Aktivität acht aus dem in Abbildung 4.1 (S. 41) gezeigten Software-Entwicklungsprozess gleich. Der in dieser Arbeit betrachtete Aspekt der Kommunikation wird in [MurtaHW2006] nicht berücksichtigt.

Enabling Architectural Refactorings through Source Code Annotations: Der Ansatz in [Krahn2006] arbeitet invasiv, d.h. die zu untersuchende Code-Basis muss mit Annotationen versehen und kann dann verarbeitet werden. Die Projekte, die so untersucht werden sollen müssen also zunächst modifiziert werden. Dies ist bei dem in dieser Arbeit vorgestellten Verfahren nicht der Fall, da hier nicht-invasiv gearbeitet wird.

Ähnlich wie diese Arbeit wird auch in [AldrichCN2002] und darauf aufbauend in [Abi-Antoun2005] die Code-Basis mit bestimmten Schlüsselworten angereichert, um Architekturbeschreibungen mit der Implementierung zu verknüpfen.

Es ist ebenfalls möglich Modelldifferenzen nach Beendigung eines Projektes, zu analysieren. In [Wagner2007] wurde die Evolution einer Software vom *Proof of Concept* zu einem echten Produkt im Nachhinein untersucht. Dabei wurden Metriken für die einzelnen Meilensteine erstellt, sodass allgemeine Aussagen über die Qualität getroffen werden können. Diese Aussagen konnten allerdings nicht auf einfache Weise vertieft werden. Hier kann ein Werkzeug zur Erkennung von Modelldifferenzen helfen genauere Untersuchungen durchzuführen.

7.2 Vorhandene Werkzeuge

Sotograph

Das kommerzielle Werkzeug Sotograph arbeitet ebenfalls mit einem Relationalen-Datenbank-Management-System (RDBMS) und importiert die erforderliche Information aus Code-Basen von verschiedenen Sprachen (C++, Java). MoDi verfolgt einen leichtgewichtigen Ansatz und zielt auf den täglichen Einsatz. In einer Vorstellung des Sotographen wurde beim Deutschen Zentrum für Luft- und Raumfahrt die SiLEST-Software untersucht. So konnten verschiedene Metriken, Zyklen-Analysen und Querverweise zwischen Artefakten auf verschiedenen Ab-

straktionsebenen erstellt bzw. aufgedeckt werden. Dabei mussten die Schichten der SiLEST Software im Sotographen konfiguriert werden.

Dieses Werkzeuges eignet sich für Reviews, während MoDi für den täglichen Einsatz konzipiert ist.

Ähnliche Projekte, die ebenfalls eine statische Code-Analyse durchführen sind beispielsweise [PMD] und [Checkstyle]. Diese Werkzeuge arbeiten jedoch nur „eindimensional“, also auf nur einer Code-Basis. Das in dieser Arbeit vorgestellte Werkzeug arbeitet hingegen „zweidimensional“, also auf der Grundlage von zwei Code-Basen (AM u. EM).

7.3 Zusammenfassung

In diesem Kapitel wurden verwandte Arbeiten vorgestellt und die eigene dagegen abgegrenzt. Das UMLDiff-Werkzeug [XingS2005] ist Modi wohl am ähnlichsten, allerdings wird dort ein RDBMS zur Abbildung seines Metamodells benutzt und eher für Modellvergleiche über mehrere Versionen einer Software gedacht ist.

Kapitel 8

Zusammenfassung und Ausblick

In diesem Kapitel werden die gewonnenen Erkenntnisse zusammengefasst und es wird ein Ausblick gegeben, welche Aspekte nicht im Kontext dieser Arbeit betrachtet wurden.

8.1 Zusammenfassung

Das System `MoDi`, welches in dieser Arbeit entwickelt wurde, stellt ein Werkzeug zur Überprüfung einer Implementierung bzgl. seiner Architektur zur Verfügung. Es kann in den Softwareentwicklungsprozess z.B. im Rahmen der kontinuierlichen Integration eingebunden werden. Durch einstellbare Regeln kann festgelegt werden, welche Modelldifferenzen während eines Modellvergleichs aufgedeckt werden sollen und wie das jeweilige Auftreten bewertet wird. Durch die daraufhin angeregte Kommunikation soll sichergestellt werden, dass Design und Implementierung während des gesamten Projektverlaufs konsistent gehalten werden, indem Missverständnisse ausgeräumt und neu auftretene Probleme rechtzeitig kommuniziert werden. In Kapitel 2 wurden die Grundlagen der Arbeit besprochen. Im Wesentlichen ist dies das ANTLR-Framework, mit dem der Java-Parser zur Checkstyle-Grammatik generiert wurde. Das Parsersystem ist die Ausgangsbasis für die weitere Verarbeitung durch MoDi. Weiterhin wurde ein kleiner Einblick in die Strukturdiagramme, speziell das Klassendiagramm gegeben. Das Kapitel schließt mit dem Beispiel des Pizzabringdienstes, welches im Verlauf dieses Berichtes noch öfter aufgegriffen wurde.

In Kapitel 3 wurde das Architekten- und das Entwicklermodell vorgestellt. Anschließend wurden die einzelnen Repräsentationen der Modelle im Bezug auf ihre Vergleichbarkeit untersucht. Hierbei wurden die aufeinander aufbauenden Schlussfolgerungen erarbeitet, die schließlich in der Entscheidung mündeten, als Vergleichsbasis eine Metarepräsentation zu wählen. Diese Metarepräsentation enthält nur die nötigen Informationen und abstrahiert vom Quelltext der Modelle, sodass es möglich ist, auch verschiedene Sprachen miteinander zu vergleichen (z.B. WSDL $\leftrightarrow$ Java)). Die Metarepräsentation bildet die Elemente einer Objekt-orientierten Sprache als Modell ab. Weiterhin wurden die Bestandteile von Interfaces untersucht, um die variablen Anteile zu identifizieren. Hieraus konnten die Möglichkeiten für Modelldifferenzen theoretisch erarbeitet werden. Außerdem wurde eine Umfrage unter den Entwicklern des DLR

beschrieben, die in einem kleinen Umfang Erfahrungen aus der Praxis darstellt. Aus den so gewonnenen Erkenntnissen wurden Regeln für Veränderungen erarbeitet, die entstandenen Modelldifferenzen bewerten.

In Kapitel 4 wurden die Konsequenzen für einen Software-Entwicklungsprozess besprochen. Dies ist am Beispiel des Prozesses, wie er in der Einrichtung Simulations- und Softwaretechnik des Deutschen Zentrums für Luft- und Raumfahrt gelebt wird geschehen. Es wurden die einzelnen Phasen des Software-Entwicklungsprozess erläutert und anschließend untersucht wieviel Kommunikation über die entstandenen Modelldifferenzen zwischen dem Architekten und den Entwicklern erforderlich ist. Schließlich wurde dargelegt, welche Möglichkeiten es zur Integration eines Werkzeuges für die Erkennung von Modelldifferenzen in den Software-Entwicklungsprozess des DLR gibt. Mit Feststellung 4.3 (S. 46) wurde die Empfehlung für den Einsatz im Daily-Build ausgesprochen, wo Modelldifferenzen erkannt und täglich kommuniziert werden sollen.

In Kapitel 5 wurde die Realisierung von MoDi, einem Werkzeugs zur Erkennung von Modelldifferenzen beschrieben. Zunächst wurden die Stakeholder identifiziert und ein Haupt-, sowie vier Unteranwendungsfälle definiert und die Anforderungen an das System beschrieben. Anschließend wurde der Entwurf vorgestellt und die Implementierung der Checks erläutert. Ferner wurden die Erweiterungsmöglichkeiten besprochen. Schließlich wurde anhand des Pizzabringdienst-Beispiels (siehe Abb. 2.1, S. 18) ein Modellvergleich durchgeführt und die Konfiguration, sowie der aus dem Modellvergleich resultierende Bericht beschrieben. Zuvor wurden einige Änderungen im EM unternommen.

In Kapitel 6 wurde MoDi an den zwei Projekten SiLEST und SESIS des Deutschen Zentrums für Luft- und Raumfahrt erprobt und die Ergebnisse diskutiert. Es es konnte der vorgeschlagene Prozess, dass ein EM gegen ein AM verglichen und nach Abnahme als neues AM übernommen wird nachempfunden werden, da die Entwicklermodelle manuell gegen das AM geprüft worden sind. Mit Hilfe vom MoDi wäre diese Prüfung sicherlich einfacher gewesen.

In Kapitel 7 wurden verwandte Arbeiten vorgestellt und die eigene dagegen abgegrenzt. Am nächsten steht MoDi wohl das UMLDiff-Werkzeug [XingS2005], welches im Untershied zu MoDi ein RDBMS zur Abbildung seines Metamodells benutzt und eher für Modellvergleiche über mehrere Versionen einer Software gedacht ist.

8.2 Ausblick

Oft werden in Projekten UML-Modelle der Architektur im Nachhinein aus dem bestehenden Code generiert. In solchen Fällen kann mit dem MoDi-System eine Auswertung in der Form stattfinden, dass die Revisionen als Architektenmodell festgehalten werden, aus denen die UML-Modelle generiert worden sind. Als zugehörige Entwicklermodelle werden dann alle nachfolgenden Revisionen identifiziert.

Das System `MoDi` kann nicht nur zur Erkennung von Modelldifferenzen genutzt werden. Durch seine Architektur ist es einfach, weitere Analysen (Checks s. 5.4.1, S. 57) zu implementieren, die auf der Metarepräsentation arbeiten. Zum Beispiel könnten Designregeln überprüft wer-

den, wie es auch Werkzeuge wie etwa [PMD] oder [Checkstyle] tun. Allerdings wären diese Überprüfungen dann nicht an die Sprache Java gebunden, da `MoDi` nicht direkt auf dem abstrakten Syntaxbaum arbeitet, sondern eine Metarepräsentation gewählt wurde.

Die Architektur von `MoDi` bietet mehrere Erweiterungspunkte. Momentan können die Modelle (AM, EM) aus dem Dateisystem entnommen werden, hier kann eine Komponente entwickelt werden, welche die Modelle direkt aus einem SCM-System importieren kann. So müsste dies nicht mehr von dem angrenzenden Systemen übernommen werden. Hierzu muss ein solcher `SCMImporter` das *IFileFinder*-Interface implementieren und in der `FileFinderFactory`, sowie der zugehörigen Konstante `ImportHandler` eingetragen werden.

Weiterhin wird im Transformationsschritt von den Modellen zur Metarepräsentation momentan nur die Sprache Java unterstützt. Für das ANTLR-System existieren weitere Grammatiken anderer Sprachen für die ebenfalls Parser generiert werden können. Diese Parser erstellen wiederum Abstrakte Syntaxbäume, welche wieder durch Treewalker ausgelesen werden können. So kann die Metarepräsentation mit Informationen gefüllt werden und das Modell steht für einen Modellvergleich zur Verfügung.

Das MoDi-Systems könnte ferner dahingehend erweitert werden, dass auch das Verhalten von Software, also die dynamischen Aspekte berücksichtigt werden. Hierzu gab es bereits Arbeiten im DLR (z.B. [Bock2005]) und es kann versucht werden, die dort verwendeten Konzepte in MoDi zu übernehmen.

Durch die frühzeitige, feingranulare und kontinuierliche Erkennung von Modelldifferenzen in Softwareprojekten kann die sonst entstehende Drift zwischen der Architektur und ihrer Implementierung vermieden werden. Mit dem in dieser Arbeit entwickelten Werkzeug zur Erkennung von Modelldifferenzen `MoDi` wurde eine Grundlage für die kontinuierliche Qualitätssicherung von implementierten Architektenmodellen gelegt.

Abbildungsverzeichnis

Tabellenverzeichnis

Literaturverzeichnis

[Pugh2006] Ken Pugh, *Interface-Oriented Design (Pragmatic Programmers)*, The Pragmatic Programmers LLC., June, 2006

[Oestereich2001] Bernd Oestereich, *Objektorientierte Softwareentwicklung (Analyse und Design mit UML)*, Oldenbourg, 2001,

[GHJV1995] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addosin Wesley, 1995

[Fowler1999] Martin Fowler, *Refactoring: Improving the Design of Existing Code*, Addosin Wesley, 1999

[PiloneP2005] Dan Pilone, Neil Pitman, *UML 2.0 in a Nutshell*, O'Reilly Media, Inc.; 2 edition (June 1, 2005)

[Schwaber2001] Ken Schwaber, Mike Beedle, *Agile Software Development with SCRUM*, Prentice Hall; 1st edition (October 15, 2001)

[Clark2004] Mike Clark, *Pragmatic Project Automation: How to Build, Deploy, and Monitor Java Applications* The Pragmatic Programmers, August 2004

[ECSS1996] European Cooperation for Space Standardization, *M-30A Project Phasing and Planning*, ESA Publications Division, Noordwijk, 1996

[Bock2005] Michael Bock, *Dynamisches Code-Analyseverfahren für Benutzerschnittstellen am Beispiel einer Web-Anwendung*, Diplomarbeit TU Braunschweig, 2005

[XingS2005] Zhengchang Xing, Eleni Stroulia, *UMLDiff: An Algorithm fpr Object-Oriented Design Differencing*, In Proceedings of the 20th IEEE/ACM International Conference on Automated software engineering, Seiten 54–65, 2005

[ApiwattanapongOH2004] Taweesup Apiwattanapong, Alessandro Orso, Mary Jean Harrold, *A Differencing Algorithm for Object-Oriented Programs*, In Proceedings of the 19th IEEE International Conference on Automated Software Engineering, Seiten 2–13, 2004

[MurtaHW2006] Leonardo G. P. Murta, Andre van der Hoek, Claudia M. L. Werner, *ArchTrace: Policy-Based Support for Managing Evolving Architecture-to-Implementation Traceability Links*, In Proceedings of the 21st IEEE International Conference on Automated Software Engineering, Seiten 135–144, 2006

[Krahn2006] Holger Krahn, Bernhard Rumpe, *Enabling Architectural Refactorings through Source Code Annotations.*, Modellierung 2006, Seiten 203–212, 2006

[Myers1986] Eugene W. Myers, *An $O(ND)$ Difference Algorithm and Its Variations* In Algorithmica, Seiten 251–266 1986

[Wagner2007] Matthias Wagner, *Evolution from a Scientific Application to an Applicable Product*, In Proceedings of 11th European Conference on Software Maintenance and Reengineering, Seiten 223–232, 2007

[AldrichCN2002] Jonathan Aldrich, Craig Chambers, David Notkin, *ArchJava: connecting software architecture to implementation.*, In Proceedings International Conference on Software Engineering (ICSE), Seiten 187–197, 2002

[Abi-Antoun2005] Marwan Abi-Antoun, Jonathan Aldrich, David Garlan, Bradley Schmerl, Nagi Nahas, Tony Tseng, *Improving system dependability by enforcing architectural intent*, In Proceedings of the 2005 workshop on Architecting dependable systems, Seiten 1–7, 2005

[Ludewig2002] Jochen Ludewig *Modelle im Software Engineering - eine Einführung und Kritik*, In Modellierung 2002, Seiten 7–22, 2002

[BischofbergerKL2004] Walter R. Bischofberger, Jan Kühl, Silvio Löffler, *Sotograph - A Pragmatic Approach to Source Code Architecture Conformance Checking.*, In Proceedings EWSA, Seiten 1–9, 2004

[LiedtkeE1989] Claus-E.Liedtke, Manfred Ender, *Wissensbasierte Bildverarbeitung*, Springer-Verlag, 1989

[ATripp2007] Andy Tripp, *Manual Tree Walking Is Better Than Tree Grammars*, `http://www.antlr.org/article/1170602723163/treewalkers.html`, 04.02.2007

[TParr2004] Terence Parr, *Translators Should Use Tree Grammars*, `http://www.antlr.org/article/1100569809276/use.tree.grammars.tml`

[TParr2006] Terence Parr, `http://www.cs.usfca.edu/~parrt/course/652/lectures/antlr.html`, 23.04.2007

[UML] UML-Spezifikationen der OMG, `http://www.omg.org/technology/documents/formal/uml.htm`, 24.04.2007

[XMI] XML Metadata Interchange, `http://www.omg.org/technology/documents/formal/xmi.htm`, 31.05.2007

[XSL] XSL - Extensible Stylesheet Language Family, `http://www.w3.org/Style/XSL/`, 31.05.2007

[JSL] James Gosling, Bill Joy, Guy Steele and Gilad Bracha, *The Java^{TM} Language Specification Third Edition*, ISBN 0-321-24678-0, `http://java.sun.com/docs/books/jls/index.html`

[JavaBNF] A BNF for Java, `http://cui.unige.ch/db-research/Enseignement/analyseinfo/JAVA/BNFindex.html`, 28.11.2006

[BNF] Backus Naur Form, `http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html`, 11.12.2006

[SESIS] SESIS - Schiffsentwurfs- und Simulationssystem `http://www.sesis.de`, 30.05.2007

[SiLEST] SiLEST -Software-in-the-Loop-Testumgebung `http://www.silest.de`, 31.05.2007

[JavaTutorial] Defining an Interface, `http://java.sun.com/docs/books/tutorial/java/IandI/interfaceDef.html`, 08.01.2007

[Java] Java Programming Language, `http://java.sun.com/` 30.04.2007

[ANTLR] ANother Tool for Language Recoqnition, Parser Generator, `http://www.antlr.org/`, 30.04.2007

[eclipse] Eclipse - an open development platform, `http://www.eclipse.org`, 03.06.2007

[Checkstyle] Checkstyle, Coding standards, `http://checkstyle.sourceforge.net/`, 30.04.2007

[Clover] Clover, Code coverage, `http://www.cenqua.com/clover/`, 28.05.2007

[PMD] PMD, Check for possible flaws, `http://pmd.sourceforge.net/`, 28.05.2007

[Jupiter] Jupiter, Code review, `http://csdl.ics.hawaii.edu/Tools/Jupiter/`, 16.05.2007

[Hibernate] Hibernate, a persistence and query service, `http://www.hibernate.org/`, 28.05.2007

[Subversion] Ben Collins-Sussman, *The subversion project: buiding a better CVS*, Linux Journal, Seite 3, Specialized Systems Consultants, Inc., 2002

[CVS] Tom Morse, *CVS*, Linux Journal, Seite 3, Specialized Systems Consultants, Inc., 1996

[CruiseControl] CruiseControl, a framework for a continuous build process, `http://cruisecontrol.sourceforge.net/`, 01.06.2007

[MANTIS] Mantis Bug Tracker, `http://www.mantisbt.org/`, 04.06.2007

[SVNChecker] SVNChecker, `https://wiki.sistec.dlr.de/SVNChecker`, 31.05.2007

[WSDL] Web Services Description Language, `http://www.w3.org/TR/wsdl`, 20.05.2007

[AXIS] Web Services - Axis, `http://ws.apache.org/axis/`, 20.05.2007

[MS Sharepoint] Microsoft Sharepoint, `http://www.microsoft.com/sharepoint/`, 31.05.2007

[MoinMoin] Moin-Moin-Wiki, `http://moinmoin.wikiwikiweb.de/`, 31.05.2007

[OMG] Object Management Group, `http://www.omg.org/`, 04.06.2007

Danksagung

Ich bedanke mich bei den folgenden Personen, die alle dazu beigetragen haben, dass ich diese Masterarbeit und somit den Abschluss meines Masterstudiums erreichen konnte:

Herrn Prof. Dr. Kurt Schneider danke ich, dass er es mir ermöglicht hat, dieses Thema zu bearbeiten. Gerade seine Vorlesungen sind bewusst und unbewusst mit in diese Arbeit eingeflossen.

Ebenso bedanke ich mich bei Frau Prof. Dr. Nicola Henze, die mir durch ihre Vorlesung Semantic Web Techniken gezeigt hat, die ich auch in dieser Arbeit wieder anwenden konnte.

Bei meinem externen Betreuer Axel Berres bedanke ich mich für die anregenden Diskussionen, die das Projekt MoDi sehr vorangetrieben haben.

Bei meinem internen Betreuer Daniel Lübke, der mir immer mit gutem Rat und Ideen zur Seite stand, möchte ich mich besonders bedanken.

Ferner bedanke ich mich bei den Kollegen der Einrichtung Simulations- und Softwaretechnik des Deutschen Zentrums für Luft- und Braunschweig, die immer ein offenes Ohr für für mich hatten und mir bei kniffeligen Problemen zur Seite standen.

Ganz besonders bedanke ich mich bei meinem Sohn Nick Elias und meiner Tochter Malin Anna, einfach weil sie da sind, sowie meiner Lebensgefährtin Katharina Kirsch die mir immer in schwierigen Situationen zur Seite steht. Weiterhin bedanke ich mich bei meinen Eltern Helga und Dieter, sowie bei meiner Schwester Meike Hinzmann, für die tolle Unterstützung während des Studiums.

Schließlich danke ich Herrn Prof. Dr. Rainer Parchmann, der mich in meinem Entschluss, vom Studium der Humanmedizin zur Informatik zu wechseln, bestärkt hat.

Erklärung

Hiermit versichere ich, dass ich die vorliegende Masterarbeit selbständig und ohne fremde Hilfe verfasst und keine anderen als die in der Arbeit angegebenen Quellen und Hilfsmittel verwendet habe. Die Arbeit hat in gleicher oder ähnlicher Form noch keinem anderen Prüfungsamt vorgelegen.

Hannover, den 05.06.2007

_____

Eingegangen am (Datum/Stempel):_____

Inhalt der beiliegenden CD-ROM

Die beiliegende CD-ROM enthält im Einzelnen:

1. Diesen Bericht
   (a) im PDF-Format
   (b) als LaTeX-Quelltext
2. Das Programm MoDi
   (a) Quelltext des Programms
   (b) Binär-Version des Programms (modi.jar)