Institut für Softwaretechnologie

Abteilung Software Engineering

Universität Stuttgart
Universitätsstraße 38
70569 Stuttgart

Diplomarbeit Nr. 2521

codation – Verbindung von Code mit Zusatzinformation

Jochen Wertenauer

Studiengang:	Softwaretechnik
Prüfer:	Prof. Dr. rer. nat. Jochen Ludewig
Betreuer:	Dipl.-Inf. Markus Knauß
begonnen am:	01. August 2006
beendet am:	31. Januar 2007
CR-Klassifikation:	D.2.3, D.2.8, E.2, H.3, H5.4, I.7

Zusammenfassung

Häufig werden auf den Quelltext verweisende zusätzliche Informationen erhoben, z. B. die Ergebnisse einer Programm-Analyse. Da die Erhebung der Zusatzinformationen aufwändig sein kann, wird eine Möglichkeit benötigt, die Zusatzinformationen zusammen mit dem Quelltext speichern und somit wiederverwenden zu können. Zu diesem Zweck wurde in der vorliegenden Diplomarbeit *Codation* entwickelt. Codation ist ein Informationsmodell, in dem Zusatzinformationen zusammen mit Quelltext gespeichert und nach Änderungen am Quelltext automatisch nachgezogen werden können. Wesentliche Vorteile von Codation gegenüber anderen Werkzeugen sind die automatische Synchronisation nach Änderungen und die erweiterbare Architektur, die es ermöglicht, Zusatzinformationen zu beliebigen Dateitypen zu speichern.

Die prototypische Implementierung des Informationsmodells erlaubt es, von Zusatzinformationen auf syntaktische Elemente von Java-Dateien zu verweisen. Das Synchronisationsverfahren liefert Änderungsinformationen auf syntaktischer Ebene. Dadurch ergeben sich Möglichkeiten, die über die von rein lexikalisch arbeitenden Programmen wie *diff* hinausgehen.

Abstract

Often one collects information that is linked with source code, e. g. the results of source code analysis. As the collection of this additional information might be time-consuming, one needs a tool to store the information together with the source code to be able to reuse the information later. Therefore *Codation* was developed in this thesis. Codation is an information model to store additional information together with source code and to automatically synchronise it after the source code has been changed. Main advantages of Codation compared to other tools are the automatic synchronisation and its extensible architecture allowing to annotate arbitrary files.

The prototype implementing the information model allows to link additional information with syntactic elements of Java files. The synchronisation algorithm provides delta information on syntactic level, too. This results in possibilities exceeding those of tools like *diff*, which are working solely on the lexical level.

Inhaltsverzeichnis

1 Einleitung **17**

1.1 Hintergrund . . . 17

1.2 Aufgabenstellung . . . 19

1.3 Methodisches Vorgehen . . . 19

1.3.1 Anforderungsanalyse und Beschreibungsschema . . . 19

1.3.2 Literaturrecherche . . . 20

1.3.3 Entwicklung eines Informationsmodells . . . 20

1.3.4 Realisierung des Annotationswerkzeugs . . . 21

1.3.5 Fallstudie . . . 21

1.4 Aufbau der Arbeit . . . 21

2 Eclipse **23**

3 Anforderungen an Informationsmodelle und Annotationswerkzeuge **25**

3.1 Speicherung von Code und Zusatzinformationen . . . 25

3.2 Verweise auf Quelltext . . . 25

3.3 Synchronisation . . . 26

3.4 Editierbarkeit . . . 26

4 Beschreibungsschema für Informationsmodelle **27**

4.1 Datenmodell . . . 28

4.1.1 Verweise . . . 28

4.1.2 Datenstruktur . . . 30

4.2 Speicherung . . . 31

4.2.1 Getrennte Speicherung . . . 31

4.2.2 Speicherung in derselben Datei . . . 32

4.3 Synchronisation . . . 32

4.3.1 Synchronisation in der Werkzeugdarstellung . . . 33

4.3.2 Rücksynchronisation aus exportierter Darstellung in die Werkzeugdarstellung . . . 34

4.4 Editierbarkeit und Werkzeugunterstützung . . . 35

4.4.1 Ursprüngliche Funktionalität bleibt nutzbar . . . 35

4.4.2 Editierbare, synchronisierte „Nur-Code“-Sicht vorhanden . . . 36

4.4.3 Verwendbarkeit von Übersetzern . . . 36

5 Überblick über die Literatur **37**
5.1 Informationsmodelle und Annotationswerkzeuge 37
5.1.1 Integrated Software Maintenance Environment 37
5.1.2 Literate Programming 38
5.1.3 Elucidative Programming 39
5.1.4 XSDoc 40
5.1.5 Model-To-Model Traceability Links 40
5.1.6 Gegenüberstellung der Informationsmodelle 41
5.1.7 Weitere Informationsmodelle 44
5.1.8 XML-Markup von Quelltext 45
5.2 Beurteilung der Aufgabenangemessenheit 46
5.2.1 Anwendungsfälle 47
5.2.2 Bewertungsergebnisse 48
5.3 Fazit der Literaturrecherche 50

6 Das Informationsmodell von Codation **51**
6.1 Speicherung der Zusatzinformationen 51
6.1.1 Kombinierte oder getrennte Speicherung? 51
6.1.2 Methoden der getrennten Speicherung 55
6.2 Datenmodell einer Zusatzinformation 58
6.2.1 Verweise in Java-Dateien 59
6.2.2 Bestimmung der markierten AST-Knoten 60
6.3 Synchronisation 64
6.3.1 Methoden zur Bestimmung von Änderungsinformationen . . 65
6.3.2 Ansatzpunkte zur Änderungserkennung bei Eclipse 67
6.3.3 Synchronisation in Codation 71
6.3.4 Syntaktischer Vergleich von Java-Dateien 71
6.4 Realisierung des Informationsmodells 77
6.5 Zusammenfassung der Eigenschaften von Codation 78

7 Fallstudie **81**
7.1 Manuelle Erhebung und Verwaltung der Daten 81
7.2 Verwendung von Codation 82
7.3 Ergebnisse der Fallstudie 84
7.3.1 Vorteile bei der Verwendung des Use-Case-Managers 84
7.3.2 Nachteile und Einschränkungen 84
7.3.3 Fazit 85

8 Fazit **87**

8.1 Rückblick . . . 87
8.1.1 Projektverlauf . . . 87
8.1.2 Probleme während der Arbeit . . . 88
8.2 Fazit . . . 90
8.3 Ausblick . . . 91

A Ergebnisse von Metrikerhebungen 93
A.1 Erhobene Metriken . . . 93
A.1.1 Metriken zur Bestimmung des Programm-Umfangs . . . 93
A.1.2 Metriken zur Beurteilung der Codequalität . . . 94
A.1.3 Metriken zur Beurteilung der Entwurfsqualität . . . 95
A.2 Gemessene Werte . . . 97
A.2.1 Core-Plugin . . . 97
A.2.2 Java-Link-Provider . . . 98
A.2.3 Simple-Link-Provider . . . 99
A.2.4 XML-Storage-Service . . . 100
A.2.5 UI-Plugin . . . 101
A.2.6 Use-Case-Manager . . . 102
A.2.7 JUnit-Tests . . . 103
A.3 Fazit . . . 104
A.3.1 Entwurfsqualität . . . 104
A.3.2 Codequalität . . . 104

B Laufzeitmessungen 105
B.1 Messumgebung . . . 105
B.2 Testdaten . . . 105
B.3 Messergebnisse . . . 107
B.4 Fazit . . . 108

C Entwurf 111
C.1 Plugin-Architektur von Codation . . . 111
C.2 Aufbau einer Zusatzinformation . . . 113
C.3 Verwalten der Zusatzinformationen . . . 114
C.4 Verweise auf Ressourcen . . . 117
C.5 Synchronisation . . . 119
C.5.1 Methoden zur Änderungserkennung . . . 119
C.5.2 Berechnung von Änderungsinformationen . . . 121
C.6 File-Link-Provider . . . 124
C.6.1 Simple-Link-Provider . . . 124

C.6.2 Java-Link-Provider 124
C.7 XML-Storage-Service 128
C.8 Beschreibungen der Extension-Points 132
C.8.1 Annotation-Provider 132
C.8.2 File-Link-Provider 135
C.8.3 Storage-Service-Provider 139

D Testplan **143**
D.1 Einleitung 143
D.1.1 Zweck des Dokuments 143
D.1.2 Testdurchführung 143
D.1.3 Nachführung des Testplans 143
D.2 Testfälle 144
D.2.1 Core- und UI-Plugin 144
D.2.2 Use-Case-Manager 146

E Inhalt der beiliegenden CD **151**

Literaturverzeichnis **153**

Abbildungsverzeichnis

1.1 Ablauf der Visualisierung 18

4.1 Eigenschaften von Informationsmodellen: oberste Ebenen 27
4.2 Eigenschaften von Informationsmodellen: Datenmodell 28
4.3 Eigenschaften von Informationsmodellen: Speicherung 31
4.4 Eigenschaften von Informationsmodellen: Synchronisation 32
4.5 Eigenschaften von Informationsmodellen: Synchronisation in der Werkzeugdarstellung 33
4.6 Eigenschaften von Informationsmodellen: Rücksynchronisation aus der exportierten Darstellung 34
4.7 Eigenschaften von Informationsmodellen: Editierbarkeit und Werkzeugunterstützung 35

6.1 Klassendiagramm einer Zusatzinformation 59
6.2 Beispiel für eine Java-Klasse und deren AST 61
6.3 Klassendiagramm `IJavaElement` 69
6.4 Transformation zwischen zwei geordneten Bäumen. 73
6.5 Editiergraph für die Bäume aus Abbildung 6.4 75
6.6 Änderung der Knotenreihenfolge 76

7.1 Plug-In zur Verwaltung von Anwendungsfällen 83

8.1 Termindrift-Diagramm 89

C.1 Architektur von Codation 112
C.2 Klassendiagramm „Zusatzinformation“ 114
C.3 Klassendiagramm „Verwaltung von Zusatzinformationen“ 115
C.4 Sequenzdiagramm „Änderung von Zusatzinformationen“ 116
C.5 Klassendiagramm „Verweise auf Ressourcen“ 118
C.6 Klassendiagramm „Project-Builder“ 122
C.7 Klassendiagramm „Änderungsinformationen“ 123
C.8 Klassendiagramm „Simple-Link-Provider“ 124
C.9 Klassendiagramm „Java-Link-Provider“ 125
C.10 Klassendiagramm „Editiergraph“ 126
C.11 XML-Schema zur Speicherung (Teil 1) 129

C.12 XML-Schema zur Speicherung (Teil 2) 130
C.13 Klassendiagramm „XML-Storage-Service“ 131

Tabellenverzeichnis

5.1 Gegenüberstellung der Informationsmodelle, Teil 1 42
5.2 Gegenüberstellung der Informationsmodelle, Teil 2 43
5.3 Ergebnisse der Bewertung 49

6.1 Eigenschaften von Codation 80

B.1 Für die Laufzeitmessungen ausgewählte Dateien 106

Abkürzungsverzeichnis

AST	Abstract Syntax Tree, siehe Definition 4.1
DTD	Document Type Definition
EP	Elucidative Programming, siehe Abschnitt 5.1.3
ISME	Integrated Software Maintenance Environment, siehe Abschnitt 5.1.1
JDT	Java Development Toolkit – Java Entwicklungsumgebung in Eclipse
LOC	Lines Of Code – Metrik, die in einer Codeeinheit, z. B. einer Methode, die Anzahl der Codezeilen bestimmt
LP	Literate Programming, siehe Abschnitt 5.1.2
M2MTL	Model-to-Model Traceability Links, siehe Abschnitt 5.1.5
WMC	Weighted Methods per Class – Metrik, definiert als die Summe der zyklomatischen Komplexität aller Methoden einer Klasse

Definitionen und Begriffserklärungen

Definitionen

1.1 Zusatzinformationen . . . 17

1.2 Informationsmodell . . . 18

1.3 Darstellungsmodell . . . 18

1.4 Annotationswerkzeug . . . 18

4.1 Abstrakter Syntaxbaum . . . 29

4.2 Werkzeugdarstellung . . . 31

4.3 Exportierte Darstellung . . . 31

5.1 Aufgabenangemessenheit . . . 46

6.1 Geordneter Baum, ungeordneter Baum . . . 72

6.2 Änderungs-Operation . . . 72

6.3 Geordnete Relation . . . 72

6.4 Transformation . . . 73

6.5 Kostenfunktion γ . . . 74

6.6 Editierabstand . . . 74

6.7 Editiergraph . . . 74

C.1 Annotation-Provider . . . 111

C.2 Storage-Service-Provider, Storage-Service . . . 113

C.3 File-Link-Provider . . . 113

Begriffserklärungen

2.1 Extension-Point 23
2.2 Extension 23
2.3 Plug-In 23
2.4 Feature 24
6.1 Ressource 55
6.2 Listener 67

1 Einleitung

1.1 Hintergrund

Häufig werden bei der Visualisierung von Quelltext neben dem Quelltext selbst weitere Informationen dargestellt. Beispiele für diese Informationen, die entweder bereits getrennt vom Quelltext vorliegen oder noch ermittelt werden müssen, sind:

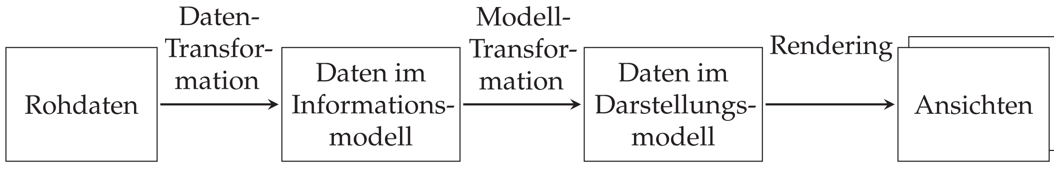
- mit Metriken erhobene Daten
- Informationen zum Programmablauf, wie sie für Glass-Box-Tests verwendet werden
- Ergebnisse einer manuellen Programmanalyse, beispielsweise die Zuordnung von Entwurfsmustern zu Strukturen im Quelltext
- Dokumentation von Anforderungen oder Testfällen

Zusammenfassend werden diese Informationen als *Zusatzinformationen* bezeichnet und wie folgt definiert:

Definition 1.1 (Zusatzinformationen) *Als Zusatzinformationen werden Daten bezeichnet, die den Quelltext ergänzen und für dessen Ausführung nicht notwendig sind. Sie beziehen sich auf einen oder mehrere Teile des Quelltextes, z. B. eine Klasse.*

Nachdem alle Zusatzinformationen ermittelt wurden, ermöglicht eine geeignet gewählte und am Betrachter und dessen Informationsbedarf orientierte Visualisierung eine effektive Vermittlung der Zusatzinformationen durch eine graphische Darstellung.

Die Visualisierung der Daten folgt nach Card et al. (1999) dem in Abbildung 1.1 auf der nächsten Seite dargestellten Ablauf. Aus den Rohdaten, in diesem Fall dem Quelltext, werden im ersten Schritt die benötigten Daten in einem *Informationsmodell* gesammelt. Dieser Schritt erfolgt manuell und teilweise auch automatisch. Der zweite Schritt, die Modell-Transformation, überführt die Daten aus dem Informationsmodell in ein darstellbares Modell. Das darstellbare Modell (*Darstellungsmodell*) ist ein um Darstellungsinformationen erweitertes Informationsmodell. Darstellungsinformationen sind zum Beispiel Koordinaten und Farbe. Im dritten Schritt werden



Legende

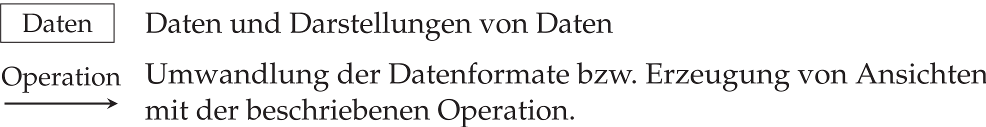


Abbildung 1.1: Ablauf der Visualisierung (nach Card et al. 1999)

aus dem Darstellungsmodell die dem Anwender angezeigten Darstellungen erzeugt (Rendering, siehe auch Foley et al. (1995)).

Auf Basis dieses Ablaufs werden die Begriffe Informationsmodell und Darstellungsmodell in dieser Arbeit wie folgt definiert:

Definition 1.2 (Informationsmodell) *Das Informationsmodell ist ein Modell, in dem sowohl der Quelltext einer Software als auch Zusatzinformationen gespeichert werden.*

Definition 1.3 (Darstellungsmodell) *Im Darstellungsmodell werden die für die Visualisierung aufbereiteten und um Darstellungsinformationen ergänzten Daten des Informationsmodells gespeichert.*

Definition 1.4 (Annotationswerkzeug) *Ein Annotationswerkzeug ist eine Software, in der ein konkretes Informationsmodell umgesetzt ist. Es ist somit in der Lage, Quelltext und Zusatzinformationen zu speichern und zu verwalten.*

Der erste Schritt, die manuelle und teilweise automatische Datensammlung durch Analysen des Quelltextes und die Verknüpfung dieser Daten mit dem Quelltext, ist oft zeitaufwändig und ressourcenintensiv. Außerdem muss darauf geachtet werden, die Zusatzinformationen stets aktuell und konsistent zu halten. Deshalb soll in dieser Diplomarbeit ein Informationsmodell entwickelt werden, das es erlaubt, die gesammelten Zusatzinformationen dauerhaft mit dem Quelltext in einem Informationsmodell zu speichern und aktuell zu halten. Das Annotationswerkzeug, das dieses Informationsmodell implementiert und somit die Wiederverwendung der gesammelten Zusatzinformationen ermöglicht, wird *Codation* [kɒˌdeɪʃən] genannt. Codation ist ein Kunstwort, das sich aus den Worten Code und Annotation zusammensetzt.

1.2 Aufgabenstellung

Zitat aus der Aufgabenstellung:

> In dieser Diplomarbeit soll ein Informationsmodell entwickelt werden, das es ermöglicht, einmal gesammelte Daten aus Analysen eines Programmcodes gemeinsam mit diesem zu speichern. Ziel ist die Minimierung des Aufwandes für die Datenerhebung in einer Visualisierung und eine Maximierung zusätzlicher Information zum Programmcode. Bei der Arbeit gilt es, alle Aspekte des zum Programmcode gehörenden Informationsmodells abzuwägen, zum Beispiel die interne oder externe Speicherung, Editierbarkeit, Aktualität, etc.
>
> Bestandteil der Arbeit ist auch eine prototypische Realisierung des Informationsmodells für die Programmiersprache Java, um die Realisierbarkeit und Praktikabilität nachzuweisen. Vorteilhaft ist eine Realisierung in Java als Eclipse-Plugin.

1.3 Methodisches Vorgehen

Dieser Abschnitt beschreibt die während der Diplomarbeit durchgeführten Tätigkeiten. Sie wurden zu Beginn in einem Projektplan zu Arbeitspaketen zusammengefasst und zeitlich sowie inhaltlich geplant. Die Planung wurde im Verlauf der Diplomarbeit nachgeführt.

Bei der Bearbeitung eines Arbeitspakets wurden zunächst die zu klärenden Forschungsfragen festgehalten. Die Problemstellungen wurden anschließend untersucht und die Forschungsfragen beantwortet.

1.3.1 Anforderungsanalyse und Beschreibungsschema

In diesem Arbeitspaket werden die Anforderungen an Informationsmodelle untersucht. Darauf aufbauend werden die für das Thema der Diplomarbeit wichtigen Eigenschaften von Informationsmodellen ermittelt und in einer Hierarchie angeordnet, die das Schema für die Beschreibung von Informationsmodellen bildet. Die durch die Hierarchie entstehende Strukturierung dient dazu, die Eigenschaften der untersuchten Informationsmodelle besser vergleichen zu können. Eine Bewertung

von Informationsmodellen findet durch die Darstellung im Beschreibungsschema nicht statt.

Für die Bewertung werden Anwendungsfälle bestimmt, die mit einem Annotationswerkzeug ausführbar sein müssen, beispielsweise die Bearbeitung von Quelltext.

1.3.2 Literaturrecherche

Bei der Literaturrecherche werden in der Literatur beschriebene Informationsmodelle geprüft und die verwendeten Ansätze auf ihre Vor- und Nachteile hin untersucht. Die während der Literaturrecherche gewonnenen Erkenntnisse bilden eine wichtige Grundlage für die Entwicklung eines eigenen Informationsmodells. Die Literaturrecherche umfasst folgende Arbeitsschritte:

1. Suche nach Literatur über Informationsmodelle.
2. Untersuchung der gefundenen Informationsmodelle und deren anschließende Dokumentation in den folgenden Schritten:
   a) Vorstellung des beim Informationsmodell verwendeten Ansatzes.
   b) Beschreibung der Eigenschaften des Informationsmodells mit dem Beschreibungsschema.
   c) Sofern ein Annotationswerkzeug vorhanden ist, das das beschriebene Informationsmodell umsetzt, wird seine Aufgabenangemessenheit bewertet. Dazu wird geprüft, ob die in der Analyse bestimmten Anwendungsfälle im Annotationswerkzeug realisiert sind und wie komfortabel die entsprechenden Funktionen zu bedienen sind.

Da die Auswahl an untersuchten Informationsmodellen Einfluss auf die im Beschreibungsschema enthaltenen Eigenschaften und deren hierarchische Anordnung hat, überlappt sich die Literaturrecherche zeitlich mit der Entwicklung des Beschreibungsschemas für Informationsmodelle.

1.3.3 Entwicklung eines Informationsmodells

Bei der Entwicklung eines Informationsmodells wird zunächst ermittelt, welche Entwurfsalternativen zur Auswahl stehen. Die Grundlage dafür bilden das Beschreibungsschema und die während der Literaturrecherche gewonnenen Erkenntnisse. Bei jeder Entwurfsalternative werden daraufhin Vor- und Nachteile untersucht.

Diese werden gegeneinander abgewogen, um daraus eine begründete Entscheidung für eine Entwurfsalternative abzuleiten. Bei der Entscheidung wird auch die Umsetzbarkeit als Eclipse-Plugin beachtet.

1.3.4 Realisierung des Annotationswerkzeugs

Dieses Arbeitspaket umfasst die Realisierung eines Annotationswerkzeugs, welches das entwickelte Informationsmodell umsetzt. Die Realisierung erfolgt als Eclipse-Plugin und besteht aus den folgenden Arbeitsschritten:

1. Anfertigung eines Entwurfs
2. Umsetzung des Entwurfs
3. Erstellung von JUnit-Testfällen und eines Testplans
4. Nachführung von eventuell am Entwurf notwendig gewordenen Änderungen
5. Erstellung einer Dokumentation für Entwickler und Anwender

Als Plattform für die Realisierung wurde Eclipse ausgewählt, weil Eclipse gut erweitert werden kann und zahlreiche Funktionen bereitstellt, auf denen aufgebaut werden kann. Außerdem besitzt Eclipse einen hohen Verbreitungsgrad und große Akzeptanz unter Entwicklern.

1.3.5 Fallstudie

In der Fallstudie wird das entwickelte Annotationswerkzeug evaluiert. Dazu wird unter Zuhilfenahme des Annotationswerkzeugs ein Testprojekt umgesetzt und dokumentiert, wie die Verwendung des Annotationswerkzeugs die Arbeitsabläufe beeinflussen kann.

1.4 Aufbau der Arbeit

Die Einleitung erläutert die Motivation für die Diplomarbeit, deren Aufgabenstellung und das Vorgehen bei der Durchführung.

In Kapitel 2 wird Eclipse vorgestellt. Außerdem werden bei Eclipse verwendete Begriffe, die für die Diplomarbeit relevant sind, erklärt.

Kapitel 3 beschreibt die an ein Informationsmodell gestellten Anforderungen.

Das Schema, mit dem die Eigenschaften von Informationsmodellen und Annotationswerkzeugen beschrieben werden, wird in Kapitel 4 definiert.

Kapitel 5 enthält die Ergebnisse der Literaturrecherche. In diesem Kapitel werden die untersuchten Informationsmodelle vorgestellt und dokumentiert, inwieweit sie die an sie gestellten Anforderungen erfüllen.

Das im Laufe der Diplomarbeit entwickelte Informationsmodell wird in Kapitel 6 beschrieben. Außerdem werden Algorithmen erläutert, die beispielsweise bei der Behandlung von Änderungen an annotiertem Quelltext verwendet werden.

Kapitel 7 beschreibt die Fallstudie und ihre Ergebnisse.

In Kapitel 8 erfolgt ein Rückblick auf die Diplomarbeit. Außerdem werden die Ergebnisse zusammengefasst und mögliche Anwendungsbereiche für das entwickelte Annotationswerkzeug diskutiert.

Die Anhänge enthalten weitere bei der Durchführung der Diplomarbeit entstandene Dokumente.

2 Eclipse

Da das in dieser Arbeit entwickelte Informationsmodell als Eclipse-Plugin realisiert wird, also eine enge Beziehung zu den von Eclipse angebotenen Funktionen besteht, werden in diesem Kapitel einige bei Eclipse verwendete Begriffe erläutert.

Der Begriff Eclipse selbst bezeichnet in dieser Arbeit die von der Eclipse Foundation entwickelte Plattform, die in den offiziellen FAQs (Arthorne u. Laffra, 2004) wie folgt beschrieben wird:

> Eclipse is an open (IDE) platform for anything, and for nothing in particular.

Das zentrale Konzept bei Eclipse ist Offenheit. Das Design von Eclipse erlaubt die einfache Erweiterung durch Dritte. Dazu enthält Eclipse einen Mechanismus, um Module zu integrieren und auszuführen. Diese Module werden als *Plug-Ins* bezeichnet. Laut Gamma u. Beck (2003) kann Eclipse als eine Ansammlung von „Orten, um Dinge einzustöpseln" (*Extension-Points*) und „Dingen, die eingestöpselt werden" (*Extensions*) angesehen werden.

Begriffserklärung 2.1 (Extension-Point) *Ein Extension-Point stellt Dritten Funktionen zur Verfügung oder bietet ihnen die Möglichkeit, bestehende Funktionalität zu erweitern.*

Ein Beispiel für einen Extension-Point ist `org.eclipse.ui.editors`. Mit diesem Extension Point können Editoren für bestimmte Dateitypen registriert werden.

Begriffserklärung 2.2 (Extension) *Eine Extension erweitert einen durch einen Extension-Point festgelegten Teil von Eclipse um zusätzliche Funktionen oder registriert sich dort für die Benutzung eines angebotenen Dienstes. Die Definition der Extension muss ein vom Extension-Point festgelegtes Schema erfüllen. Dadurch wird sichergestellt, dass die Extension alle vom Extension-Point benötigten Informationen zur Verfügung stellt.*

Die Extension `org.eclipse.jdt.ui.CompilationUnitEditor` des Extension-Points `org.eclipse.ui.editors` definiert beispielsweise den Editor für Java-Klassen.

Begriffserklärung 2.3 (Plug-In) *Ein Plug-In stellt ein Modul dar. In seinem Manifest gibt ein Plug-In an, welche Extension-Points es zur Verfügung stellt und welche es über Extensions selbst verwendet.*

Ein bekanntes Plug-In aus dem Lieferumfang von Eclipse ist `org.junit`, mit dem in Eclipse Unit-Tests durchgeführt werden können.

Begriffserklärung 2.4 (Feature) *Plug-Ins können zu Features zusammengefasst werden. Dies erleichtert die Installation von Plug-Ins und die Verwaltung von Abhängigkeiten.*

Eines der wichtigsten Features von Eclipse (neben den Features der Plattform selbst) stellt das Java Development Toolkit (JDT) dar. Es erweitert Eclipse zu einer Entwicklungsumgebung für Java-Programme.

3 Anforderungen an Informationsmodelle und Annotationswerkzeuge

Dieses Kapitel beschreibt die Anforderungen an ein Informationsmodell und die Anforderungen an ein Annotationswerkzeug. Die Anforderungen wurden aus der Aufgabenstellung der Diplomarbeit ermittelt.

Als Beispiel zur Erläuterung der Anforderungen wird, soweit möglich, die Metrik Weighted Methods per Class (WMC) verwendet. Die Metrik berechnet für jede Methode einer Klasse deren zyklomatische Komplexität nach McCabe (1976) und bildet daraus die Summe, die als Maß für die Komplexität einer Klasse angesehen werden kann.

3.1 Speicherung von Code und Zusatzinformationen

Wie aus der Definition des Begriffs Informationsmodell (Definition 1.2 auf Seite 18) hervorgeht, ist die primäre Anforderung an ein Informationsmodell, Zusatzinformationen zusammen mit Quelltext speichern zu können. Die Speicherung muss aber nicht notwendigerweise in derselben Datei erfolgen. Bei der Metrik WMC bestehen die Zusatzinformationen aus der jeweiligen zyklomatischen Komplexität der Methoden und der daraus gebildeten Summe.

3.2 Verweise auf Quelltext

Neben der Speicherung der Zusatzinformationen ist es erforderlich, angeben zu können, auf welchen Teil des Quelltextes sich die Zusatzinformationen beziehen. Im Fall der Programmiersprache Java erfolgen Verweise auf syntaktischer Ebene, wobei die kleinste zu adressierende Einheit das Token ist. Somit ist es nicht notwendig, einzelne Zeichen adressieren zu können.

Die Verweise erfolgen auf syntaktischer und nicht auf lexikalischer Ebene, da in den meisten Fällen auf Sprachkonstrukte und nicht auf Zeichenfolgen verwiesen werden soll. Bei der Metrik WMC sind die Sprachkonstrukte beispielsweise Methoden und Klassen.

3.3 Synchronisation

Ein Informationsmodell muss eine Synchronisation unterstützen, bei der auf Änderungen am Quelltext reagiert wird, um Zusatzinformationen nachzuführen. Beispielsweise muss die oben beschriebene Metrik WMC neu berechnet werden, nachdem eine Methode geändert wurde.

3.4 Editierbarkeit

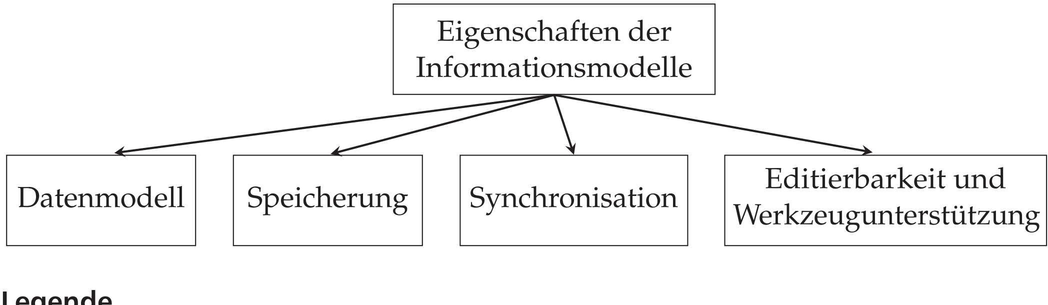
Das Informationsmodell und die Umsetzung im Annotationswerkzeug dürfen die Möglichkeiten zur Bearbeitung des Quelltextes möglichst wenig einschränken. Bei einer Umsetzung als Eclipse-Plugin ist zum Beispiel wichtig, die vom Java Development Toolkit (JDT) angebotenen Werkzeuge weiter benutzen zu können.

Nach Festlegung der Anforderungen wird untersucht, wie diese prinzipiell umgesetzt werden können. Auf diesen Ergebnissen beruht das im nächsten Kapitel definierte Schema für die Beschreibung von Informationsmodellen.

4 Beschreibungsschema für Informationsmodelle

Das in diesem Kapitel vorgestellte Beschreibungsschema dient dazu, bei der Literaturrecherche (nächstes Kapitel) untersuchte Informationsmodelle einordnen und vergleichen zu können. Dazu werden auf Grundlage der im vorigen Kapitel vorgestellten Anforderungen die für das Thema der Diplomarbeit relevanten Eigenschaften von Informationsmodellen bestimmt und in einer hierarchischen Struktur organisiert. Für jede Eigenschaft werden ihre möglichen Ausprägungen angegeben. Die Ausprägungen beschreiben die verschiedenen Möglichkeiten, wie die Eigenschaft in einem Informationsmodell umgesetzt sein kann. Zusammen mit der Strukturierung ermöglicht dies eine einheitliche tabellarische Gegenüberstellung von Informationsmodellen, die es erleichtert, Gemeinsamkeiten und Unterschiede zwischen Informationsmodellen zu erkennen.

In Abbildung 4.1 sind die beiden obersten Ebenen der Hierarchie dargestellt. Die abgebildeten Unterkategorien werden in den folgenden Abschnitten beschrieben.



Legende

Eigenschaft	Eigenschaft eines Informationsmodells oder eines Annotationswerkzeugs
⟶	„Oberkategorie-von“-Beziehung; der Pfeil zeigt von der enthaltenden Eigenschaft zur enthaltenen.

Abbildung 4.1: Eigenschaften von Informationsmodellen: oberste Ebenen

4.1 Datenmodell

In der Kategorie „Datenmodell“ sind Eigenschaften zusammengefasst, die beschreiben, wie die gespeicherten Daten aufgebaut und miteinander verknüpft sind. Sie enthält die Unterkategorien „Verweise“ und „Datenstruktur“. Abbildung 4.2 zeigt die Kategorie und ihre Unterkategorien.

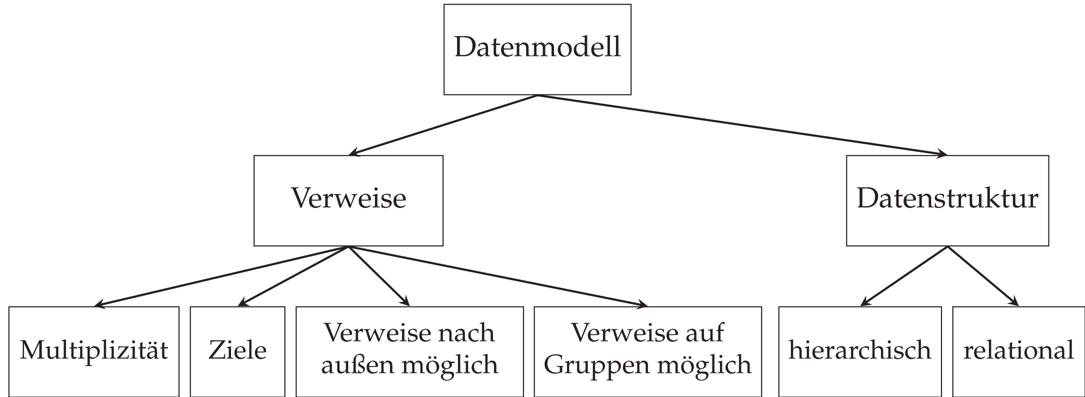


Abbildung 4.2: Eigenschaften von Informationsmodellen: Datenmodell

4.1.1 Verweise

Die Kategorie „Verweise“ enthält Eigenschaften des Informationsmodells, welche die Verweise von Zusatzinformationen auf Teile des Quelltextes betreffen.

Multiplizität

Diese Eigenschaft beschreibt, wie viele Verweise von einer Zusatzinformation ausgehen können und wie viele Zusatzinformationen auf den gleichen Teil des Quelltextes verweisen können.

Mögliche Ausprägungen der Eigenschaft sind ($n, m \in \mathbb{N}_0$):

a) $0 \ldots 1 : 0 \ldots 1$ – Auf einen Teil des Quelltextes kann höchstens eine Zusatzinformation verweisen. Eine Zusatzinformation kann auf höchstens einen Teil des Quelltextes verweisen.

b) $n : 0 \dots 1$ – Auf einen Teil des Quelltextes können beliebig viele Zusatzinformation verweisen. Eine Zusatzinformation kann auf höchstens einen Teil des Quelltextes verweisen.

c) $0 \dots 1 : m$ – Auf einen Teil des Quelltextes kann höchstens eine Zusatzinformation verweisen. Eine Zusatzinformation kann auf beliebig viele Teile des Quelltextes verweisen.

d) $n : m$ – Keine Einschränkungen bezüglich der Multiplizität der Verweise.

Ziele

Definition 4.1 (Abstrakter Syntaxbaum) *Der abstrakte Syntaxbaum (Abstract Syntax Tree, AST) ist eine Baumstruktur, welche die syntaktischen Elemente eines Dokuments (z. B. Anweisungen oder Methoden) in ihrem inhaltlichen Zusammenhang wiedergibt.*

Die Eigenschaft „Ziele“ beschreibt, wie auf einen Teil des Quelltextes verwiesen wird. Mögliche Ausprägungen sind:

a) AST-Knoten – Es wird *ein* Knoten eines abstrakten Syntaxbaums oder einer ähnlichen Datenstruktur als Ziel des Verweises angegeben.

b) AST-Teilstruktur – Als Ziel eines Verweises wird der Teil eines AST oder einer ähnlichen Datenstruktur angegeben. Im Gegensatz zu „AST-Knoten“ kann eine mehrere Knoten umfassende Teilstruktur des AST als Ziel angegeben werden, beispielsweise eine Folge von Anweisungen.

c) Bereich – Es wird auf einen Teil der Datei verwiesen, in dem sich das Ziel des Verweises befindet. Eine Angabe des Ziels könnte also beispielsweise „Datei xy.txt, Startposition 1024, Länge 17“ lauten.

Verweise nach außen möglich

Diese Eigenschaft gibt an, ob zusätzlich zu den Verweisen auf die annotierte Datei, zu der die Zusatzinformation gehört, Verweise in andere vom Annotationswerkzeug verwaltete Dateien möglich sind. Die möglichen Ausprägungen sind „ja“ und „nein“.

Verweise auf Gruppen möglich

Diese Eigenschaft gibt an, ob Verweise auf eine Gruppe von Dateien, z. B. ein Paket, wie es in der Programmiersprache Java verwendet wird, möglich sind. Die möglichen Ausprägungen sind „ja“ und „nein“.

4.1.2 Datenstruktur

Diese Kategorie fasst Eigenschaften zusammen, die angeben, wie die im Informationsmodell gespeicherten Daten strukturiert sind. Sie werden besonders bei kombinierter Speicherung in einer Datei (siehe Abschnitt 4.2 auf Seite 31 f.) relevant.

Die Eigenschaften „hierarchisch“ und „relational“ werden getrennt aufgeführt, da sie sich wechselseitig nicht ausschließen. Ein XML-Dokument ist z. B. hierarchisch aufgebaut, mit Hilfe der Attribute `id` und `idref` sind aber auch Verweise und damit relationale Strukturen möglich.

Hierarchisch

Die Daten werden in einer hierarchischen Struktur, z. B. einer Baumstruktur, verwaltet. Daten auf derselben Hierarchieebene sind miteinander verknüpft. Wenn beispielsweise Zusatzinformationen zu einer Methode gespeichert werden sollen und der Quelltext als AST abgespeichert wird, werden die Zusatzinformationen an den AST-Knoten angehängt, der die Methode repräsentiert.

Für diese Eigenschaft sind die Ausprägungen „ja“ und „nein“ möglich.

Relational

Die Daten werden in einer relationalen Struktur, wie sie von Datenbanken bekannt ist, verwaltet. Die Verknüpfung kann über Schlüssel und Fremdschlüssel erfolgen. Die möglichen Ausprägungen sind „ja“ und „nein“.

4.2 Speicherung

Definition 4.2 (Werkzeugdarstellung) *Das Format, in dem die im Informationsmodell hinterlegten Daten durch das Annotationswerkzeug gespeichert werden, wird als Werkzeugdarstellung bezeichnet. Diese Darstellung steht im Gegensatz zur exportierten Darstellung (siehe Definition 4.3).*

Definition 4.3 (Exportierte Darstellung) *Bei der exportierten Darstellung wird im Gegensatz zur Werkzeugdarstellung ein anderes Format oder ein anderer Ort zur Speicherung der im Informationsmodell enthaltenen Daten verwendet. Die exportierte Darstellung wird durch eine Funktion des Annotationswerkzeugs erzeugt.*

Die exportierte Darstellung umfasst zum Beispiel aus dem Informationsmodell generierte Dateien, etwa eine für einen Übersetzer geeignete Darstellung des Quelltextes. Änderungen, die ein Anwender an diesen generierten Dateien vornimmt, müssen vom Annotationswerkzeug nicht automatisch in die Werkzeugdarstellung übernommen werden.

In der Kategorie „Speicherung“ sind Eigenschaften zusammengefasst, die Ort und Art der Speicherung in der Werkzeugdarstellung betreffen. In Abbildung 4.3 ist der Aufbau der Kategorie dargestellt. Die Aufteilung in die Eigenschaften „getrennte Speicherung“ und „Speicherung in derselben Datei“ wurde von Aguiar u. David (2005) übernommen.

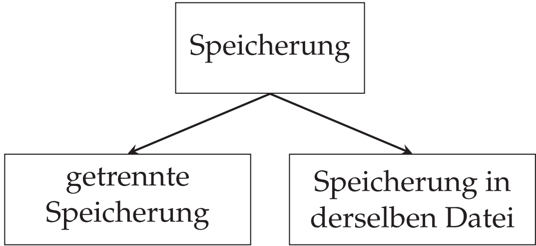


Abbildung 4.3: Eigenschaften von Informationsmodellen: Speicherung

4.2.1 Getrennte Speicherung

Diese Eigenschaft gibt an, dass Zusatzinformationen getrennt vom Quelltext gespeichert werden. Mögliche Ausprägungen dieser Eigenschaft sind:

a) binär – Das Annotationswerkzeug verwendet ein binäres Format zur Speicherung der Daten.
b) XML – Zur Speicherung wird ein XML-Format verwendet.
c) Text – Es wird ein text-basiertes Dateiformat, jedoch nicht XML, verwendet.

Es sind auch Kombinationen möglich, wenn beispielsweise Zusatzinformationen in XML-Dateien gespeichert werden und Quelltext in Textdateien.

XML und andere Textformate werden voneinander unterschieden, da für die Verarbeitung von XML-Dateien zahlreiche Werkzeuge verfügbar sind, was den Austausch von Daten erleichtert.

4.2.2 Speicherung in derselben Datei

Diese Eigenschaft gibt an, dass Zusatzinformationen und Quelltext zusammen in einer Datei gespeichert werden. Die möglichen Ausprägungen dieser Eigenschaft entsprechen denen der getrennten Speicherung.

4.3 Synchronisation

Die Kategorie „Synchronisation“ fasst Kategorien zusammen, die die Synchronisation von Zusatzinformationen und Quelltext sowie die Synchronisation von Werkzeugdarstellung und exportierter Darstellung betreffen. Abbildung 4.4 zeigt den Aufbau der Kategorie.

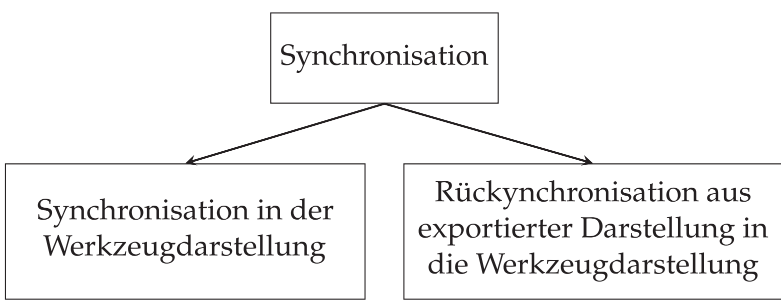


Abbildung 4.4: Eigenschaften von Informationsmodellen: Synchronisation

4.3.1 Synchronisation in der Werkzeugdarstellung

Die Eigenschaften in dieser Kategorie beschreiben, wie die Synchronisation zwischen Zusatzinformationen und Quelltext ausgelöst wird, nachdem der in der Werkzeugdarstellung gespeicherte Quelltext verändert wurde (siehe Abschnitt 3.3 auf Seite 26). Abbildung 4.5 zeigt den Aufbau der Kategorie.

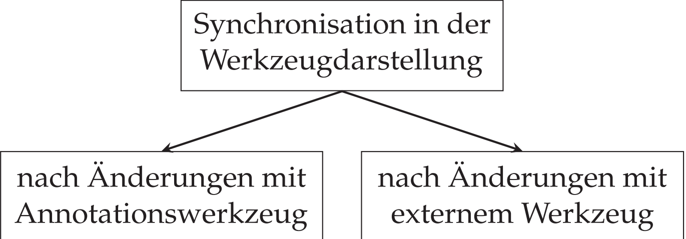


Abbildung 4.5: Eigenschaften von Informationsmodellen: Synchronisation in der Werkzeugdarstellung

Nach Änderungen mit Annotationswerkzeug

Mit dem Annotationswerkzeug wurde der im Informationsmodell gespeicherte Quelltext geändert. Mögliche Arten für den Start der Synchronisation sind:

a) automatisch – Das Annotationswerkzeug erkennt die Änderung und aktualisiert die Zusatzinformationen. Alternativ weist es den Benutzer auf eventuelle Inkonsistenzen hin und dieser kann die Synchronisation durch das Annotationswerkzeug veranlassen.

b) manuell – Der Anwender muss die Synchronisation von Hand starten. Das Annotationswerkzeug macht ihn nicht auf mögliche Inkonsistenzen aufmerksam.

c) nicht möglich – Eine Synchronisation wird vom Werkzeug nicht unterstützt. Damit erfüllt das Annotationswerkzeug allerdings *nicht* die in Kapitel 3 beschriebenen Anforderungen.

Nach Änderungen mit externem Werkzeug

Der Benutzer hat mit einem externen Werkzeug, z. B. einem Texteditor, den im Informationsmodell gespeicherten Quelltext geändert. Die möglichen Ausprägungen für die Eigenschaft entsprechen denen des vorigen Abschnitts.

4.3.2 Rücksynchronisation aus exportierter Darstellung in die Werkzeugdarstellung

Der im Informationsmodell gespeicherte Quelltext wurde in eine exportierte Darstellung überführt. Die Eigenschaften in dieser Kategorie beschreiben, ob das Annotationswerkzeug Änderungen an Quelltext, der in einer exportierten Darstellung gespeichert ist, erkennt und in die Werkzeugdarstellung übernimmt. Die Änderungen können sowohl mit dem Annotationswerkzeug selbst, als auch mit einem externen Werkzeug durchgeführt worden sein. Der Aufbau der Kategorie ist in Abbildung 4.6 dargestellt.

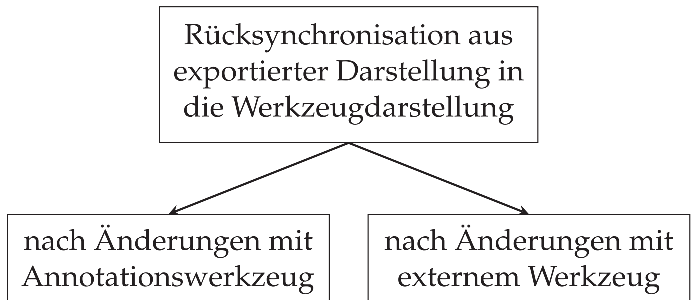


Abbildung 4.6: Eigenschaften von Informationsmodellen: Rücksynchronisation aus der exportierten Darstellung

Nach Änderungen mit Annotationswerkzeug

Im Annotationswerkzeug wurden Änderungen am Quelltext, der in einer exportierten Darstellung gespeichert ist, vorgenommen. Die möglichen Arten der Synchronisation sind:

a) automatisch – Das Annotationswerkzeug erkennt die Änderungen und übernimmt diese automatisch in die Werkzeugdarstellung.

b) manuell – Der Benutzer kann manuell eine Synchronisation anstoßen.

c) nicht möglich – Das Annotationswerkzeug unterstützt eine Synchronisation mit exportierten Daten nicht.

Nach Änderungen mit externem Werkzeug

Der Benutzer hat mit einem externen Werkzeug, z. B. einem Texteditor, den in einer exportierten Darstellung vorliegenden Quelltext geändert. Die möglichen Ausprägungen der Eigenschaft entsprechen denen des vorigen Abschnitts.

4.4 Editierbarkeit und Werkzeugunterstützung

Die in Abbildung 4.7 dargestellte Kategorie Editierbarkeit fasst Eigenschaften zusammen, welche die Bearbeitung und Weiterverarbeitung des im Informationsmodell gespeicherten Quelltextes betreffen.

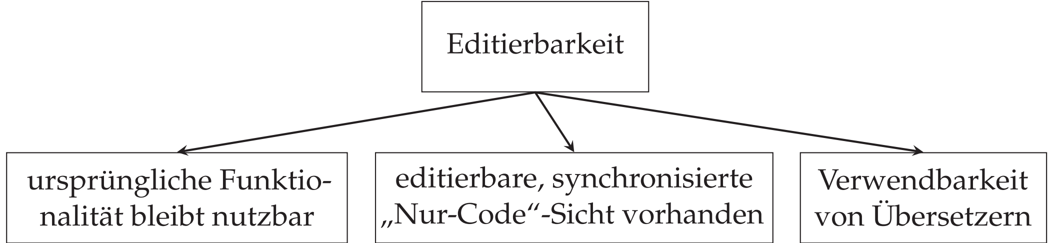


Abbildung 4.7: Eigenschaften von Informationsmodellen: Editierbarkeit und Werkzeugunterstützung

4.4.1 Ursprüngliche Funktionalität bleibt nutzbar

Diese Eigenschaft beschreibt, ob die in einem Werkzeug vorhandene Funktionalität zur Erstellung von Programmen in der vom Informationsmodell unterstützten Programmiersprache durch die Erweiterung zu einem Annotationswerkzeug weiterhin nutzbar bleibt. Im Fall von Eclipse sind dies beispielsweise die vom JDT angebotenen Refactoring-Funktionen. Die möglichen Ausprägungen der Eigenschaft sind „ja“ und „nein“.

Handelt es sich beim Annotationswerkzeug um ein eigenständiges Werkzeug, wird die Ausprägung „nein“ verwendet, da keine schon zuvor nutzbaren Funktionen vorhanden sind.

4.4.2 Editierbare, synchronisierte „Nur-Code“-Sicht vorhanden

Diese Eigenschaft gibt an, ob es eine Sicht gibt, in der Quelltext ohne Zusatzinformationen dargestellt wird und wie von einem Editor gewohnt bearbeitet werden kann. Die möglichen Ausprägungen sind „ja“ und „nein“.

4.4.3 Verwendbarkeit von Übersetzern

Diese Eigenschaft gibt an, ob Übersetzer und Werkzeuge wie JavaDoc für die im Informationsmodell verwendete Programmiersprache genutzt werden können. Mögliche Ausprägungen der Eigenschaft sind:

a) direkt nutzbar – Die Dateien, die den Quelltext enthalten, können direkt von einem normalen Übersetzer für diese Programmiersprache eingelesen werden. Das verwendete Informationsmodell erfordert keine Anpassungen an Übersetzern oder anderen Werkzeugen.

b) Präcompiler nötig – Um Übersetzer oder andere Werkzeuge, die den Quelltext parsen, verwenden zu können, muss zuerst mit einem Präcompiler oder einem ähnlichen Werkzeug eine geeignete Darstellung erzeugt werden.

   Durch den Präcompiler kann sich die Formatierung des Quelltextes von der im Annotationswerkzeug dargestellten Formatierung unterscheiden. Deshalb wird zusätzlich vermerkt, ob die Zeilennummern beibehalten werden. Ohne die Beibehaltung sind Fehlermeldungen eines Übersetzers nur schwer den entsprechenden Stellen im Quelltext zuzuordnen.

Nachdem in diesem Kapitel das Beschreibungsschema für Informationsmodelle definiert wurde, werden im nächsten Kapitel die Ergebnisse der Literaturrecherche vorgestellt. Die Gegenüberstellung der untersuchten Informationsmodelle erfolgt dabei mit Hilfe des Beschreibungsschemas.

5 Überblick über die Literatur

In der Literaturrecherche wird untersucht, ob in der Literatur bereits Informationsmodelle beschrieben sind und welche Ansätze in diesen Informationsmodellen verwendet werden. Die bei der Literaturrecherche gewonnenen Erkenntnisse stellen eine wichtige Hilfe bei der Entwicklung eines eigenen Informationsmodells dar, weil so bei der Entwicklung von Codation die in der Literatur beschriebenen Probleme zumindest teilweise umgangen und gute Ideen übernommen werden können.

Dieses Kapitel beschreibt in Abschnitt 5.1 die im Rahmen der Literaturrecherche untersuchten Informationsmodelle. Um die Informationsmodelle und die sie realisierenden Annotationswerkzeuge beurteilen und miteinander vergleichen zu können, werden sie mit dem in Kapitel 4 auf Seite 27 bis 36 definierten Schema beschrieben. Außerdem wird ihre Aufgabenangemessenheit anhand einiger Anwendungsfälle geprüft.

5.1 Informationsmodelle und Annotationswerkzeuge

5.1.1 Integrated Software Maintenance Environment

Das Integrated Software Maintenance Environment (ISME, beschrieben in Mamas u. Kontogiannis (2000)) ist ein System zur Verwaltung von Dokumenten, die bei der Entwicklung einer Software entstanden sind. Das Ziel von ISME ist, durch die zentrale Verwaltung der im XML-Format gespeicherten Daten die Integration von Werkzeugen zur Software-Wartung zu erleichtern.

In ISME wird der Quelltext geparst und sein AST gespeichert, wobei Document Type Definitions für die Programmiersprachen Java und C++ vorhanden sind. Außerdem ist eine Document Type Definition (DTD) für die allgemeinen Konzepte objektorientierter Programmiersprachen vorhanden.

Werkzeuge wie CASE-Tools können über eine Plugin-Schnittstelle auf die abgelegten Daten zugreifen und eigene Daten speichern. Außerdem können sich Werkzeuge über Änderungen benachrichtigen lassen. Da das ISME kein Format für Verweise vorgibt, sind die Werkzeuge für die Aktualisierung der Verweise und Zusatzinformationen selbst verantwortlich.

5.1.2 Literate Programming

Literate Programming (LP) ist ein von Donald Knuth entwickelter Programmieransatz (Knuth, 1992). Ziel des Ansatzes ist, Lesbarkeit und Verständlichkeit des Quelltextes zu verbessern. Dazu wird der Quelltext bei LP in Form eines Artikels beschrieben. Die Beschreibung und der Quelltext werden zusammen in einer Datei gespeichert, deren Inhalt in drei Sprachen beschrieben wird:

- der Programmiersprache, in der der Quelltext erstellt wird
- einer Sprache zur Gestaltung der Dokumentation, z. B. HTML oder TEX
- einer Sprache zur Verknüpfung der Inhalte

Der Inhalt der Datei ist in Module aufgeteilt. Jedes Modul repräsentiert einen Gedanken oder ein Konzept und besteht aus drei jeweils optionalen Teilen: der Beschreibung des Moduls, Definitionen und dem Quelltext. Der Quelltext kann Verweise auf den Quelltext anderer Module enthalten.

Zur Weiterverarbeitung der Daten werden die Präprozessoren `tangle` und `weave` verwendet. Mit dem Präprozessor `tangle` wird aus der Datei der Quelltext extrahiert. Die Verarbeitung beginnt bei einem als Hauptmodul markierten Modul, Verweise auf andere Module werden durch den dort angegebenen Quelltext ersetzt. Aus einer Datei wird immer genau eine Datei mit Quelltext generiert.

Die Dokumentation wird mit dem Präprozessor `weave` erzeugt. Das ausgegebene Dokument enthält für jedes Modul einen Abschnitt, der aus der Beschreibung und einer Darstellung des Quelltextes des Moduls besteht. Die Reihenfolge der Abschnitte entspricht der Reihenfolge der Module in der Originaldatei. Die Reihenfolge der Module ist somit an der erzeugten Dokumentation und deren Verständlichkeit und nicht am erzeugten Programm ausgerichtet.

Es existieren zahlreiche Umsetzungen des LP-Ansatzes. Einige der wichtigsten Umsetzungen sind:

- WEB (Knuth, 1984): Erstes System für LP zur Erstellung und Dokumentation von Pascal-Programmen. Für die Dokumentation wird TEX verwendet.
- CWEB (Knuth u. Levy, 2001): Mit CWEB können Programme in C, C++ und Java erstellt werden. Als Dokumentationssprache wird TEX verwendet.
- FunnelWeb (Williams, 2000): Kann mit jeder (in Textform dargestellten) Programmiersprache verwendet werden. Dokumentation kann als HTML- und als TEX-Dokument erstellt werden.

- Nuweb (Briggs, 1993): Nuweb ist wie FunnelWeb unabhängig von der Programmiersprache, in der der Quelltext geschrieben ist. Verwendet LaTeX zur Formatierung der Dokumentation.
- Noweb (Ramsey, 1994): Noweb ist ein sehr einfach gehaltenes Werkzeug für LP, das aber über zusätzliche Filter erweitert werden kann. Zur Erzeugung der Dokumentation können HTML, TeX und LaTeX verwendet werden.

Die genannten Umsetzungen bestehen nur aus den beiden Präprozessoren `weave` und `tangle` und enthalten keine Entwicklungsumgebungen. Für CWEB und Noweb existieren Erweiterungen für EMACS (Stallman, 1981). Diese Erweiterungen bieten jedoch kaum mehr als Syntaxhervorhebung.

Als Alternative zu den EMACS-Modi kann der Editor Leo (Leo) verwendet werden. Leo unterstützt den LP-Ansatz zur Erstellung von Dokumenten und enthält einen Präprozessor `tangle`. Außerdem kann Leo die von CWEB und Noweb verwendeten Formate importieren und exportieren. Ein Präprozessor `weave` zur Erzeugung der Dokumentation fehlt jedoch.

5.1.3 Elucidative Programming

Elucidative Programming (EP) ist ein im Wesentlichen von Kurt Nørmark entwickelter Ansatz zur Dokumentation von Quelltext, der in Nørmark (2000a) als Variante von Literate Programming (LP) beschrieben wird. Im Gegensatz zu LP werden bei EP Dokumentation und Quelltext getrennt voneinander gespeichert. Da der Quelltext bei EP nicht verändert werden muss, können bestehende Entwicklungsumgebungen weiter verwendet werden. Dadurch erhoffen sich die Entwickler von EP eine höhere Akzeptanz bei den Programmierern.

Bei den zuerst entwickelten Implementierungen von EP für Java (Nørmark et al., 2000) und Scheme (Nørmark, 2000b) handelt es sich um Webanwendungen, bei denen Quelltext und Dokumentation nebeneinander im Browser angezeigt werden. Die Erstellung der Dokumentation erfolgt bei diesen Werkzeugen mit EMACS. Dabei können Verweise angelegt werden, die von der Dokumentation aus auf Teile des Quelltextes zeigen. Beim Betrachten der in HTML dargestellten Dokumentation kann über diese Verweise zur entsprechenden Stelle im Quelltext gesprungen werden.

Vestdam (2003) beschreibt eine Integration von EP für Java in die Entwicklungsumgebung TogetherJ. Dabei wird TogetherJ um eine Perspektive erweitert, welche die

Anzeige und Bearbeitung der Dokumentation parallel zur Editierung des Quelltextes ermöglicht. Dadurch entfällt die bei Nørmark et al. (2000) vorhandene Trennung in eine Entwicklungsumgebung, einen Editor zur Erstellung der Dokumentation und einen Browser zur Darstellung.

5.1.4 XSDoc

XSDoc (Aguiar u. David, 2005) ist ebenfalls eine Software zur Dokumentation von Quelltext. Die mit XSDoc (Extensible Software Documentation) erstellte Dokumentation wird in einem Wiki verfasst. Über Plug-Ins können Entwicklungsumgebungen (im Prototyp nur Eclipse) Daten mit dem Wiki austauschen.

Alle Daten werden bei XSDoc in XML konvertiert und in einem zentralen Archiv gespeichert. Zur Speicherung von Java-Quelltext verwendet XSDoc das Format JavaML (Badros, 2000; Aguiar et al., 2004). In C++ erstellter Quelltext wird zur Speicherung dem Werkzeug Doxygen (van Heesch, 2006) in eine XML-Darstellung konvertiert.

In der Dokumentation kann der Benutzer von XSDoc mit speziellen Schlüsselwörtern Querverweise auf andere Teile der Dokumentation, auf UML-Diagramme und auf Abschnitte des Quelltextes einfügen. Außerdem können UML-Diagramme und Codeabschnitte in Form von Listings eingebunden werden. Die Dokumentation wird bei XSDoc erst zum Zeitpunkt der Anzeige aus ihren Bestandteilen zusammengestellt, um die Aktualität der eingebundenen Daten sicherzustellen. Sie kann im HTML- und im PDF-Format erzeugt werden.

XSDoc wird zur Zeit im Projekt doc-it! (Aguiar, 2006) in das am Fraunhofer-Institut für Rechnerarchitektur und Softwaretechnik entwickelte SnipSnap-Wiki (SnipSnap) integriert.

5.1.5 Model-To-Model Traceability Links

In Maletic et al. (2005) wird ein Ansatz zur Verwaltung der Verknüpfungen zwischen Modellen, den sogenannten „traceability links“, diskutiert. Bei den im Paper beschriebenen Modellen handelt es sich um Quelltext und Klassendiagramme. Die Verknüpfungen stellen Abhängigkeiten dar und verbinden Teile eines Modells mit Teilen eines anderen Modells. Ändert sich ein Teil eines Modells, müssen die Änderungen an den damit verknüpften Teilen des jeweils anderen Modells nachgezogen werden.

Die Modelle werden in einer in Collard et al. (2002) beschriebenen XML-Darstellung gespeichert. Da die Verknüpfungen in einer separaten XML-Datei gespeichert werden, bestehen keine Einschränkung bezüglich der verwendeten Modelle. Die Verknüpfungen werden mit XPath (Clark u. DeRose, 1999) und XLink (Maler et al., 2001) realisiert.

Zur Erkennung von Änderungen wird „Meta-Differencing“ verwendet (Maletic u. Collard, 2004). Damit kann ermittelt werden, welche syntaktischen Elemente einer XML-Datei sich geändert haben. Meta-Differencing geht somit über eine rein lexikalische Änderungserkennung, wie sie z. B. `diff` bietet, hinaus.

5.1.6 Gegenüberstellung der Informationsmodelle

Die Eigenschaften der vorgestellten Informationsmodelle werden in diesem Abschnitt anhand des in Kapitel 4 auf Seite 27 bis 36 definierten Beschreibungsschemas untersucht. Die Gegenüberstellung der Eigenschaften erfolgt in den Tabellen 5.1 und 5.2 auf den folgenden Seiten. Literate Programming bezieht sich dabei auf CWEB, Elucidative Programming auf das in Vestdam (2003) beschriebene Werkzeug.

	Elucidative Programming	ISME	Literate Programming	M2MTL[1]	XSDoc
Datenmodell					
Verweise					
Multiplizität	$n : m$	$n : m$	$0 \ldots 1 : 0 \ldots 1$	$n : m$	$n : m$
Ziele	AST-Knoten	AST-Knoten, AST-Teilstruktur	Bereich	AST-Knoten, AST-Teilstruktur	AST-Knoten, Bereich
Verweise nach außen möglich	ja	ja	nein	ja	ja
Verweise auf Gruppen möglich	nein	nein	nein	nein	nein
Datenstruktur					
hierarchisch	nein	nein	ja	nein	nein
relational	ja	ja	nein	ja	ja
Speicherung					
getrennt	Code: Text Zusinf.: XML	XML	–	XML	XML
in derselben Datei	–	–	Text	–	–
Synchronisation in der Werkzeugdarstellung nach Änderungen …					
… mit dem Annotationswerkzeug	manuell	automatisch[2]	manuell	automatisch	manuell
… mit einem externen Werkzeug	manuell	manuell	manuell	manuell	manuell
Rücksynchronisation aus exportierter in die Werkzeugdarstellung nach Änderungen …					
… mit dem Annotationswerkzeug	nicht möglich	automatisch[2,3]	nicht möglich	manuell	nicht möglich
… mit einem externen Werkzeug	nicht möglich	manuell[3]	nicht möglich	manuell	nicht möglich

Anmerkungen

[1] Model-to-Model Traceability Links
[2] Der Begriff Annotationswerkzeug umfasst hier auch über Plug-Ins integrierte Werkzeuge.
[3] Abhängig vom über Plug-Ins integrierten Werkzeug

Tabelle 5.1: Gegenüberstellung der Informationsmodelle, Teil 1

	Elucidative Programming	ISME	Literate Programming	M2MTL	XSDoc
Editierbarkeit und Werkzeugunterstützung					
ursprüngliche Funktionalität bleibt nutzbar	ja	nein	nein	–[2]	ja
editierbare, synchronisierte „Nur-Code“-Sicht vorhanden	ja	nein	nein	–[2]	ja
Verwendbarkeit von Übersetzern	direkt nutzbar	Präcompiler nötig, Zeilen sind nicht zuzuordnen	Präcompiler nötig, Zeilen sind nicht zuzuordnen[1]	Präcompiler nötig, Zeilen sind zuzuordnen	direkt nutzbar

Anmerkungen

[1] Bei CWEB können für die Programmiersprachen C und C++ Zeilen über die Direktive `#line` zugeordnet werden.
[2] Aus der Beschreibung in Maletic et al. (2005) nicht ersichtlich.

Tabelle 5.2: Gegenüberstellung der Informationsmodelle, Teil 2

5.1.7 Weitere Informationsmodelle

Außer den zuvor angeführten Informationsmodellen wurden bei der Literaturrecherche weitere Ansätze gefunden, bei denen Zusatzinformationen zu Quelltext gespeichert werden sollen. Diese Ansätze sind aber im Hinblick auf das Thema dieser Arbeit ungeeignet oder es fehlt eine Beschreibung der Eigenschaften, die für den Vergleich mit den anderen Informationsmodellen untersucht werden. Zwei dieser Ansätze werden in den folgenden Unterabschnitten vorgestellt.

Open architecture for visual reverse engineering

Telea (2004) beschreibt eine Architektur für Software-Visualisierungs-Werkzeuge. Der Fokus der Architektur liegt auf der Verwendung beim Reverse-Engineering. Um beim Reverse-Engineering ein Verständnis für die Struktur und Semantik eines Programms zu entwickeln, wird das Programm analysiert. Danach werden die Software-Artefakte und ihre Beziehungen identifiziert und auf verschiedenen Abstraktionsstufen dargestellt.

Das Datenmodell besteht aus zwei Elementen: Einem geschichteten, attributierten Graphen und Selektionen. Die Knoten des Graphen werden durch Programmanalyse, z. B. durch Parsen des Quelltextes erzeugt. Die Schichten des Graphen entstehen beispielsweise durch Zusammenfassen von Code in Komponenten. Die Kanten des Graphen drücken Beziehungen zwischen Knoten aus, wobei keine Einschränkungen bezüglich der Topologie des Graphen bestehen.

Zu den Knoten und Kanten des Graphen können Attribute gespeichert werden. Attribute sind Paare (Schlüssel, Wert), wobei der Schlüssel ein String und ein Wert ein primitiver Datentyp (String, Boolean, Integer, ...) sein muss. Attribute repräsentieren Daten, die bei der Programmanalyse gewonnen bzw. abgeleitet wurden, beispielsweise die Ergebnisse einer Metrik.

Eine Selektion ist eine Teilmenge der Knoten und Kanten des Graphen. Auf ihr können Operationen durchgeführt werden. Außerdem können Selektionen visualisiert werden.

Die Beschreibung der Architektur enthält keine Details zur Art und Weise, in der Quelltext gespeichert wird. Außerdem war auch auf Nachfrage beim Autor keine Implementierung der Architektur zu beschaffen. Die fehlenden Informationen zur Art der Speicherung konnten somit nicht ermittelt werden. Deshalb ist eine Bewertung des Informationsmodells nicht möglich.

Annotationen

Java bietet seit Version 1.5 die Möglichkeit, zusätzliche Informationen zu Klassen, Attributen und Methoden als Annotationen zu speichern (Gosling et al., 2005, Abschnitt 9.7). Die Annotationen können zur Laufzeit und mit dem Annotation Processing Toolkit ausgewertet werden. Teilweise werden sie auch vom Übersetzer ausgewertet.

Die Unterstützung für Annotationen wurde zuvor bereits von Microsoft in der Common Language Runtime eingeführt. Annotierbar sind dort alle über Reflexion erreichbaren Elemente. Die MSDN Library (2006) enthält eine Beschreibung der Verwendung von Annotationen in der Programmiersprache C#.

Wegen der geringen Granularität – es sind nur Methoden, aber nicht Teile von Methoden annotierbar – werden diese Ansätze hier nicht näher untersucht.

Cazzola et al. (2005) beschreiben eine Erweiterung von C#, [a]C#, mit der es zusätzlich möglich ist, Code-Blöcke zu annotieren. Ein Code-Block ist eine durch geschweifte Klammern umschlossene Folge von Anweisungen. Damit ist es jedoch weiterhin nicht möglich, beispielsweise die Bedingung in einer if-Anweisung mit Zusatzinformationen zu versehen.

5.1.8 XML-Markup von Quelltext

Neben den bereits genannten Ansätzen JavaML und srcML gibt es zahlreiche weitere Ansätze, den Quelltext einer Programmiersprache als XML-Dokument darzustellen. Dabei wird der Quelltext mit einem Parser eingelesen und der daraus resultierende AST als XML-Dokument gespeichert. Diese Darstellung führt jedoch zu Problemen, die in diesem Abschnitt beschrieben werden sollen.

Der Vorteil der XML-Darstellung besteht darin, dass Code-Analysen leichter durchgeführt werden können. Zusatzinformationen können direkt an die XML-Elemente angehängt werden. Außerdem können mit Standardtechniken wie XPath oder über IDs Verweise erstellt werden.

Ohne besondere Maßnahmen kann jedoch aus der XML-Darstellung die Originaldarstellung des Quelltextes nicht rekonstruiert werden, da ein Parser Formatierung und Kommentare entfernt. Ohne die Originaldarstellung wird der Einsatz von Werkzeugen, speziell von Debuggern, erschwert. Bei der Entwicklung von JavaML (Badros, 2000; Aguiar et al., 2004) und srcML (Collard et al., 2002) wurde deshalb darauf geachtet, die Wiederherstellung der Originaldarstellung zu ermöglichen.

Ein weiteres Problem ergibt sich aus der Tatsache, dass die XML-Schemata bzw. DTDs für gültigen Quelltext entworfen wurden: fehlerhafter Quelltext lässt sich unter Umständen nicht als AST darstellen. Der AST ist somit möglicherweise unvollständig, was Datenverluste bei der Speicherung hervorrufen kann. Durch Reduktion der Genauigkeit kann das Problem teilweise verringert werden. So werden bei srcML Expressions (Expressions sind u. a. Methodenaufrufe und Zuweisungen) wie im Quelltext angegeben als Text gespeichert und durch den Parser nicht weiterverarbeitet. Dadurch werden aber auch die Möglichkeiten zur Code-Analyse reduziert und Verweise ungenauer.

Da abgesehen von Literate Programming und Elucidative Programming alle oben genannten Informationsmodelle XML-Darstellungen von ASTs verwenden, gelten die genannten Einschränkungen auch für diese.

5.2 Beurteilung der Aufgabenangemessenheit

In Abschnitt 5.1.6 auf Seite 41 bis 44 werden die untersuchten Informationsmodelle und die sie umsetzenden Annotationswerkzeuge gegenübergestellt. Aus den angegebenen Daten kann ermittelt werden, ob sie die in dieser Arbeit gestellten Anforderungen (Speicherung von Zusatzinformationen, Verweise, Synchronisation, Editierbarkeit) erfüllen. Es ist aber keine Aussage darüber möglich, wie gut mit den Annotationswerkzeugen gearbeitet werden kann. Deshalb wird in diesem Abschnitt ihre Aufgabenangemessenheit geprüft.

Definition 5.1 (Aufgabenangemessenheit) *Ein Dialog ist aufgabenangemessen, wenn er den Benutzer unterstützt, seine Arbeit effektiv und effizient zu erledigen. Der Begriff Dialog bezeichnet die Interaktion zwischen einem Benutzer und einem Dialogsystem, um ein bestimmtes Ziel zu erreichen.* aus EN ISO 9241-10

Die Beurteilung der Aufgabenangemessenheit erfolgt über Anwendungsfälle, die beim Einsatz eines Annotationswerkzeugs auftretende Tätigkeiten beschreiben. Es wird geprüft, ob die Annotationswerkzeuge die zur Durchführung der Tätigkeiten notwendigen Funktionen bereitstellen und wie komfortabel diese Funktionen zu bedienen sind.

5.2.1 Anwendungsfälle

Bearbeiten von Quelltext

Bearbeiten von Quelltext im Annotationswerkzeug: Der Benutzer bearbeitet Quelltext im Editor des Annotationswerkzeugs.

Bearbeiten von Quelltext in externem Editor: Zur Bearbeitung von Quelltext wird ein Texteditor eingesetzt, der nicht Teil des Annotationswerkzeugs ist.

Copy & Paste mit externem Werkzeug: Der Benutzer fügt aus einem nicht zum Annotationswerkzeug gehörenden Werkzeug über die Zwischenablage Text in eine im Annotationswerkzeug geöffnete Datei ein.

Dateioperationen

Anlegen einer Datei: Der Benutzer legt mit Hilfe des Annotationswerkzeugs eine neue Datei an.

Hinzufügen einer existierenden Datei: Der Benutzer importiert eine bereits vorhandene, noch nicht im Annotationswerkzeug verwaltete Datei. Zu dieser Datei erfasst er Zusatzinformationen.

Umbenennen einer Datei: Mit Hilfe des Annotationswerkzeugs benennt der Benutzer eine Datei um. Danach müssen bei Bedarf die Verweise aus Zusatzinformationen auf diese Datei aktualisiert werden.

Löschen einer Datei: Der Benutzer löscht eine im Annotationswerkzeug verwaltete Datei mit dem Annotationswerkzeug.

Suche

Suche in einer Datei: Der Benutzer sucht in einer Datei mit Quelltext nach einer Zeichenfolge.

Suche über mehrere Dateien: Der Benutzer sucht in mehreren Dateien nach einer Zeichenfolge.

Zusatzinformationen

Zusatzinformationen anzeigen: Die im Annotationswerkzeug erfassten Zusatzinformationen sollen angezeigt werden.

Werkzeugunterstützung

Parsen des Quelltextes mit externem Werkzeug: Ein externes Werkzeug für die im Annotationswerkzeug annotierte Programmiersprache soll eingesetzt werden (siehe Abschnitt 4.4.3 des Beschreibungsschemas). Das Werkzeug wurde nicht für die Verwendung mit dem Informationsmodell angepasst. Sein Parser benötigt somit die für Quelltext der verwendeten Programmiersprache übliche Darstellung. In Abhängigkeit vom Informationsmodell muss deshalb eventuell zuvor mit einem Präcompiler eine geeignete exportierte Darstellung erzeugt werden.

5.2.2 Bewertungsergebnisse

Um die Bewertung durchführen zu können, wurden die Autoren gebeten, eine Version ihres Annotationswerkzeugs zur Verfügung zu stellen. Leider war bis auf Donald Knuth kein Autor gewillt oder in der Lage, sein Annotationswerkzeug zur Verfügung zu stellen. Zur Bewertung mussten also die Beschreibungen aus den jeweiligen Veröffentlichungen verwendet werden. Die Ergebnisse der Beurteilung sind in Tabelle 5.3 auf der nächsten Seite graphisch dargestellt. Elucidative Programming bezieht sich auf das in Vestdam (2003) beschriebene Werkzeug, Literate Programming auf CWEB.

Zu CWEB ist anzumerken, dass Donald Knuth auf der Website von CWEB die Unterstützung für Java explizit erwähnt. Die Dokumentation erklärt jedoch nicht, wie Java-Quelltext erzeugt werden kann. Stattdessen werden stets Dateien mit der Endung „c“ erzeugt. Diese enthalten dann den Java-Code, aber auch `#line`-Direktiven an einen C-Übersetzer, die vor der Übersetzung mit einem Java-Übersetzer entfernt werden müssen. Außerdem werden Kommentare nicht in die Datei übernommen; JavaDoc kann somit nicht verwendet werden. Der Anwendungsfall „Parsen des Quelltextes mit externem Werkzeug“ ist daher nur mit erheblichen Einschränkungen durchführbar.

Bei Literate Programming müssen bestehende Dateien verändert werden, um Zusatzinformationen zu ihnen zu speichern. Da Literate Programming das Verhältnis

	Elucidative Programming	ISME	Literate Programming	XSDoc
Bearbeiten von Quelltext				
im Annotationswerkzeug	●	◑	◑	●
in externem Editor	◕	◑	◑	●
Copy & Paste mit externem Werkzeug	●	◑	●	●
Datei				
anlegen	●	◑	●	●
importieren	●	◕	◔	●
umbenennen	◕	◕	●	◕
löschen	●	◕	●	◕
Suche				
in einer Datei	●	○	●	●
in mehreren Dateien	●	○	◑	●
Zusatzinformationen anzeigen	●	◕	◑	◕
Parsen des Quelltextes mit externem Werkzeug	●	◑	◑	●

Legende

- ○ Funktionalität nicht vorhanden
- ◔ Funktion nur durch sehr umständlichen Workaround erreichbar
- ◑ Funktion nur durch Workaround erreichbar oder vorhanden, aber nur mit erheblichen Einschränkungen brauchbar
- ◕ Funktion vorhanden, aber mit Schwächen bei der Bedienbarkeit
- ● Funktion vorhanden und komfortabel benutzbar

Tabelle 5.3: Ergebnisse der Bewertung

von Quelltext und Kommentaren umkehrt (der Quelltext wird an die Dokumentation angehängt und nicht Kommentare an den Quelltext), müssen die Dateien zudem umstrukturiert werden. Da die Umstrukturierung manuell durchgeführt werden muss, ist dies ein fehleranfälliger Prozess. In der Praxis besteht somit eine hohe Hemmschwelle für den Einsatz von Literate Programming bei existierender Software. Daraus ergibt sich die schlechte Bewertung beim Anwendungsfall „Datei importieren“.

Die Beschreibung des in Abschnitt 5.1.5 auf Seite 40 f. vorgestellten Informationsmodells Model-to-Model Traceability Links enthält nicht genügend Informationen, um eine Bewertung durchzuführen. Eine Beurteilung ist daher an dieser Stelle nicht möglich.

5.3 Fazit der Literaturrecherche

Literate Programming, Elucidative Programming und XSDoc dienen nur zur manuellen Dokumentation von Quelltext, eine Synchronisation ist nicht vorhanden. Verweise und Zusatzinformationen müssen bei allen Werkzeugen von Hand nachgezogen werden.

Das Integrated Software Maintenance Environment (ISME) und das Konzept der Model-To-Model Traceability Links bieten zumindest ansatzweise eine Möglichkeit zur Synchronisation an. Bei ISME fehlen aber beispielsweise detaillierte Änderungsinformationen. Beide Informationsmodelle speichern Quelltext in einer XML-Darstellung, was zu Problemen bei fehlerhaftem Quelltext führen kann (siehe Abschnitt 5.1.8 auf Seite 45 f.). Die Informationsmodelle sind auf den Einsatz in der Wartung oder bei der Programmanalyse ausgerichtet, Bereiche also, in denen der Quelltext stabil ist und geringen Änderungen unterworfen wird. Änderungen am Quelltext sind mühsam, da zwischen der XML- und der „normalen“ Darstellung konvertiert werden muss.

Somit erfüllt keiner der untersuchten Ansätze die Anforderungen, die an ein Informationsmodell und ein Annotationswerkzeug gestellt werden (siehe Kapitel 3 auf Seite 25 f.). Für Codation wurde deshalb ein neues Informationsmodell entwickelt, das die Anforderungen erfüllt. Es wird im nächsten Kapitel beschrieben.

6 Das Informationsmodell von Codation

Dieses Kapitel beschreibt das von Codation verwendete Informationsmodell und die getroffenen Entwurfsentscheidungen. Die Entwurfsentscheidungen wurden vor dem Hintergrund einer Implementierbarkeit des Informationsmodells in Eclipse getroffen und beruhen auf den bei der Literaturrecherche gewonnenen Erkenntnissen.

Bei der Entwicklung eines Informationsmodells muss definiert werden,

- wie eine Zusatzinformation aufgebaut ist,
- wie sie gespeichert wird und
- wie die Synchronisation erfolgt, nachdem der annotierte Quelltext geändert wurde.

Die Art der Speicherung besitzt, wie später beschrieben, starken Einfluss auf die Art der möglichen Verweise und somit auf den Aufbau einer Zusatzinformation. Daher wird zunächst die Frage untersucht, wie die Speicherung der Zusatzinformationen erfolgt. Anschließend wird der Aufbau einer Zusatzinformation erläutert und danach die Synchronisation. Den Abschluss des Kapitels bildet eine Übersicht über das Informationsmodell unter Verwendung des in Kapitel 4 auf Seite 27 bis 36 definierten Beschreibungsschemas.

6.1 Speicherung der Zusatzinformationen

6.1.1 Kombinierte oder getrennte Speicherung?

Bei der Frage, wie die Zusatzinformationen gespeichert werden sollen, ist zunächst zu klären, ob die Speicherung von Zusatzinformationen und Quelltext kombiniert oder getrennt erfolgt. Bei kombinierter Speicherung werden der Quelltext und die dazugehörenden Zusatzinformationen zusammen in derselben Datei gespeichert. Bei getrennter Speicherung werden die Zusatzinformationen dagegen separat vom Quelltext gespeichert, beispielsweise in einer Datenbank.

Kombinierte Speicherung

Die Vorteile der kombinierten Speicherung in einer Datei sind:

- Die Weitergabe einer Datei mit Quelltext und den dazugehörenden Zusatzinformationen ist problemlos möglich, da nur eine Datei weitergegeben werden muss.
- Die Verwendung eines Versionsverwaltungssystems wird nicht erschwert, da beim Einchecken der Datei auch alle Zusatzinformationen im Versionsverwaltungssystem gespeichert werden. Diese Aussage gilt nur eingeschränkt, wenn beim Einchecken automatisch Prüfungen des Quelltextes, z. B. eine Prüfung auf Einhaltung der Programmierrichtlinien, durchgeführt werden sollen (siehe Werkzeugeinsatz unten).
- Da ein eigenes Format verwendet wird, ist ein eigener Editor notwendig, um Quelltext in der Werkzeugdarstellung (siehe Definition 4.2 auf Seite 31) zu ändern. Der Editor besitzt Informationen darüber, an welchen Stellen der Benutzer Änderungen vorgenommen hat. Dadurch ist es möglich, effizient auf im Annotationswerkzeug vorgenommene Änderungen zu reagieren, indem nur die Zusatzinformationen aktualisiert werden, die sich auf geänderte Teile der Datei beziehen.

Die Nachteile der kombinierten Speicherung in einer Datei sind:

- Die Zusatzinformationen sind eng an eine Datei gekoppelt. Somit ist es schwierig, eine Zusatzinformation zu speichern, die sich auf mehrere Dateien bezieht, da sie einer Datei zugeordnet werden muss.

  Als Beispiel für eine Zusatzinformation mit dateiübergreifenden Verweisen stelle man sich folgenden Fall vor: In objektorientiertem Code wird eine Methode einer Klasse `Oberklasse` wie im folgenden Ausschnitt dargestellt verwendet.

```
1  private Oberklasse attribut;
2
3  public String toString() {
4    return attribut.toString();
5  }
```

  Die Klasse `Oberklasse` hat mehrere Kindklassen, welche die in Zeile vier aufgerufene Methode `toString()` überschreiben. Es soll eine Zusatzinformation

gespeichert werden, die Verweise auf den Aufruf selbst und auf alle Implementierungen der Methode, die in Zeile vier ausgeführt werden können, enthält. Die Zusatzinformation muss aktualisiert werden, wenn der Aufruf geändert wurde oder wenn beispielsweise in einer Kindklasse, die bisher die Methode `toString()` überschrieben hat, die Implementierung der Methode entfernt wurde und somit wieder die Implementierung der Klasse `Oberklasse` verwendet wird. Die Zusatzinformation muss deshalb von jedem Verweisziel aus erreichbar sein. Bei einer kombinierten Speicherung ist dies wegen der Zuordnung der Zusatzinformation zu einer Datei nur schwer möglich.

Außerdem sind durch die Kopplung an eine Datei Verweise auf Verzeichnisse kaum sinnvoll darzustellen. Das bedeutet beispielsweise, dass ein Programm, das die Metrik „Lines of Code" (LOC) berechnet, seine Ergebnisse nicht auf Pakete aggregiert speichern kann. Um die LOC für ein Paket ausgeben zu können, muss deshalb über alle im Paket enthaltenen Klassen und Unterpakete iteriert werden. Dazu müssen alle Dateien gelesen werden.

- Die Speicherung verändert die Darstellung des Quelltextes. Deshalb muss der Quelltext mit einem Präcompiler in eine Darstellung konvertiert werden, die von einem Übersetzer weiterverarbeitet werden kann.
- Die vom Präcompiler erzeugte Darstellung des Quelltextes sollte wie in der Darstellung im Annotationswerkzeug formatiert sein. Eine abweichende Formatierung erschwert die Zuordnung der von einem Übersetzer ausgegebenen Meldungen zu den entsprechenden Code-Abschnitten.
- Vorhandene Werkzeuge müssen eventuell aufwändig angepasst werden. Beispielsweise ist bei Eclipse ohne Anpassungen der Java-Editor nicht mehr nutzbar. Bei vielen Werkzeugen, speziell bei kommerziell vertriebenen, sind solche Anpassungen nicht möglich, weil entsprechende Schnittstellen fehlen oder der Quelltext nicht verfügbar ist. Um diese Werkzeuge weiterverwenden zu können, müsste der Quelltext aus der Werkzeugdarstellung in die für den Quelltext übliche Darstellung exportiert werden. Nach Durchführung der Änderungen müssen diese in die Werkzeugdarstellung übernommen werden.

Getrennte Speicherung

Die Vorteile der getrennten Speicherung sind:

- Das Originalformat des Quelltextes kann beibehalten werden. Damit ist kein Präcompiler notwendig und vorhandene Werkzeuge können ohne Einschränkungen weiterverwendet werden.

- Eine Zusatzinformation kann sich auf mehrere Dateien beziehen. Auch Verweise auf Verzeichnisse sind möglich.
- Das Format, in dem Zusatzinformationen gespeichert werden, ist unabhängig vom Quelltext, wodurch eine Erweiterung um zusätzliche Programmiersprachen oder andere Formate wie XML oder PDF möglich ist. Ein Verweis vom Quelltext in die Dokumentation ist somit denkbar.

Die Nachteile der getrennten Speicherung sind:

- Versionsverwaltung und Weitergabe von Daten sind schwieriger als bei einer kombinierten Speicherung, da mehrere Dateien betroffen sind.

  Sollen zum Beispiel nur eine Klasse und die dazu gespeicherten Zusatzinformationen an einen Dritten weitergegeben werden, werden zwei Dateien benötigt: der Quelltext der Klasse und die Datei mit den Zusatzinformationen. Werden alle Zusatzinformationen in einer zentralen Datei gespeichert, erhält der Dritte unter Umständen nicht für ihn bestimmte Informationen. Damit er die Daten in ein bestehendes Archiv aus annotiertem Quelltext einfügen kann, ist zudem eine Import-Funktion notwendig.
- Die Speicherung aller Zusatzinformationen in einer zentralen Datei kann zu Speicherplatzproblemen führen, wenn viele Zusatzinformationen verwaltet werden sollen.

Zusammenfassend lässt sich sagen, dass bei der kombinierten Speicherung die Weitergabe der Daten leichter möglich ist als bei getrennter Speicherung. Allerdings ist die Freiheit bei Verweisen auf den Quelltext eingeschränkt und der Einsatz von bestehenden Werkzeugen wird erschwert. Die getrennte Speicherung hat den Nachteil, dass bei großen Datenmengen und Speicherung aller Zusatzinformationen in einer zentralen Datei Speicherplatzprobleme auftreten können. Diese können aber wie auf den folgenden Seiten erläutert umgangen werden.

Die getrennte Speicherung hat den Vorteil, unabhängig vom Format des Quelltextes zu sein und dieses nicht ändern zu müssen. Wie bereits bei den Anforderungen an ein Informationsmodell (Abschnitt 3.4 auf Seite 26) beschrieben, ist es wichtig, zur Bearbeitung des im Annotationswerkzeug verwalteten Quelltextes die vorhandenen Werkzeuge weiter benutzen zu können. Bei einer Werkzeugdarstellung, bei der das Format des Quelltextes verändert wird, müssen die Werkzeuge jedoch angepasst werden. Dies ist immer nur für einen Teil der Werkzeuge möglich und verursacht einen kaum abzuschätzenden Aufwand. Nicht anpassbare Werkzeuge können nur über Import/Export-Funktionen des Annotationswerkzeugs verwendet werden, was bei häufigem Einsatz dieser Werkzeuge schnell unpraktikabel wird.

Die bessere Werkzeugunterstützung und die Möglichkeit, Zusatzinformationen anzulegen, die sich auf mehr als eine Datei beziehen, stufe ich als wichtiger ein als die potenziell einfachere Versionsverwaltung und Weitergabe von Daten. Deshalb werden im hier vorgestellten Informationsmodell Zusatzinformationen und Quelltext getrennt gespeichert. Ein weiterer wichtiger Vorteil der getrennten Speicherung ist, dass durch die Unabhängigkeit vom Quelltext mehrere Formate unterstützt werden können.

6.1.2 Methoden der getrennten Speicherung

Es bestehen mehrere Möglichkeiten, wie die Zusatzinformationen bei einer getrennten Speicherung abgespeichert werden können. Diese Möglichkeiten, sowie deren Vor- und Nachteile, werden nachfolgend beschrieben.

Ansatz 1: Speicherung in zentralem Archiv

Alle Zusatzinformationen werden bei diesem Ansatz an einer zentralen Stelle gespeichert, eine Aufteilung, z. B. in mehrere Dateien, findet nicht statt. Das Abbildungsverhältnis von annotierten Dateien zu Speicherstellen für Zusatzinformationen ist somit $n : 1$ ($n \in \mathbb{N}$).

a) Speicherung in einer Datei

Dieser Ansatz schränkt die Art der Verweise nicht ein. Wenn viele Zusatzinformationen gespeichert werden, kann diese Datei jedoch sehr groß werden. Dadurch können Probleme wegen Speichermangel entstehen, wenn die Daten beim Einlesen der Datei in den Arbeitsspeicher geladen werden. Außerdem ist es schwierig, nur die von einer Änderung betroffenen Bereiche einer Datei neu zu schreiben.

Dieser Ansatz wird von Eclipse verwendet, um die Informationen zu speichern, die Plug-Ins über die Methode `setPersistentProperty` der Schnittstelle `org.eclipse.core.resources.IResource` zu Ressourcen hinzufügen können.

Begriffserklärung 6.1 (Ressource) *Unter dem Begriff Ressource werden bei Eclipse Projekte, die darin enthaltenen Verzeichnisse und Dateien sowie der „Workspace root“, der die Projekte enthält und die Wurzel der Baumstruktur darstellt, zusammengefasst.*

b) Verwendung einer Datenbank

Bei einer schnellen Datenbankverbindung ergeben sich keine Probleme beim Umgang mit vielen Zusatzinformationen, da nur wenige Daten vom Annotationswerkzeug im Arbeitsspeicher gehalten werden müssen und hochwertige Datenbanksysteme in der Lage sind, auch große Datenmengen schnell zu verwalten.

Für die Inhalte der Datenbank ist jedoch eine getrennte Versionsverwaltung notwendig, während bei der Speicherung in einer Datei die für den Quelltext genutzte Versionsverwaltung wiederverwendet werden kann.

Ansatz 2: Für jede annotierte Datei eine Datei mit Zusatzinformationen

Dieser Ansatz stellt das andere Extrem dar. Für jede annotierte Datei wird eine Datei mit Zusatzinformationen angelegt; das Abbildungsverhältnis ist somit $n:n$ ($n \in \mathbb{N}$).

Wie bei der kombinierten Speicherung sind dateiübergreifende Verweise und Verweise auf Verzeichnisse wegen der Zuordnung einer Zusatzinformation zu einer Datei nur schlecht darstellbar. Außerdem sind viele (langsame) Dateizugriffe notwendig, um die Zusatzinformationen einzulesen. Dafür können Änderungen schneller gespeichert werden, da die zu schreibenden Dateien kleiner sind. Die Anzahl der Schreibvorgänge ist jedoch am höchsten.

Wird auf dateiübergreifende Verweise verzichtet, kann der Bedarf an Arbeitsspeicher auf Kosten längerer Zugriffszeiten auf die Zusatzinformationen reduziert werden, indem Dateien nur bei Bedarf eingelesen werden.

Ansatz 3: Mehrere Dateien zur Speicherung der Zusatzinformationen

Dieser Ansatz stellt einen Kompromiss zwischen den beiden ersten Ansätzen dar. Das Abbildungsverhältnis ist $n:m$ ($n, m \in \mathbb{N}$), d. h. die Zusatzinformationen werden in mehreren Dateien gespeichert, es existiert aber nicht für jede annotierte Datei eine Datei mit Zusatzinformationen. Dadurch kann sich zwar die Dauer des Einlesens gegenüber Ansatz 1 a erhöhen, das Speichern von Änderungen kann jedoch beschleunigt werden, da die zu ändernden Dateien kleiner sind. Im Vergleich zu Ansatz 2 wird der Verwaltungsaufwand reduziert, da die Anzahl der Dateien geringer ist und die Dateien größere logische Einheiten bilden.

a) Zentrale Speicherung in mehreren Dateien

Wie beim ersten Ansatz werden die Zusatzinformationen an einer zentralen Stelle gespeichert. Es wird jedoch nicht nur eine Datei zur Speicherung verwendet, sondern mehrere. Die Aufteilung kann zum Beispiel auf Basis der Komponenten erfolgen, die Zusatzinformationen erzeugen. Für jede Komponente wird eine Datei verwendet, in der die von dieser Komponente erzeugten Zusatzinformationen gespeichert werden.

b) Aufteilung gemäß der Verzeichnishierarchie

Für jedes Verzeichnis wird eine Datei angelegt, in der alle Zusatzinformationen gespeichert werden, die sich auf im Verzeichnis enthaltene Dateien beziehen. Dateiübergreifende Verweise und Verweise auf Verzeichnisse sind möglich. Fraglich ist, wo Zusatzinformationen gespeichert werden, die Verweise auf Dateien aus verschiedenen Verzeichnissen enthalten.

Fazit

Jeder Ansatz bietet Vor- und Nachteile, die je nach Anwendungsprofil ausschlaggebend für die Wahl der Speichermethode sein können. Bei einem Anwendungsprofil sind folgende Punkte zu beachten:

- Werden datei- oder verzeichnisübergreifende Verweise benötigt?
- Werden Verweise auf Verzeichnisse benötigt?
- Wie viele Zusatzinformationen sollen gespeichert werden?
- Wie oft werden Zusatzinformationen geändert? In welchem Häufigkeitsverhältnis stehen Schreib- und Lesezugriffe?
- Welches Versionsverwaltungssystem soll eingesetzt werden?

Da das Anwendungsprofil offen ist, kann hier keine Festlegung für einen der genannten Ansätze erfolgen. Deswegen wird die Persistenzschicht als Schnittstelle abstrahiert, welche die verschiedenen Ansätze der getrennten Speicherung ermöglicht. Dienste implementieren diese Schnittstelle und der Anwender kann entscheiden, welcher Dienst seinen Anforderungen am besten gerecht wird.

Bei ihrer Registrierung beim Anwendungskern müssen die Dienste angeben, welche Arten von Verweisen sie unterstützen (dateiübergreifend, verzeichnisübergreifend, Verweise auf Verzeichnisse). Eine Komponente, die Zusatzinformationen speichern

will, muss bei ihrer Registrierung angeben, welche Arten von Verweisen sie benötigt. So kann sichergestellt werden, dass eine Komponente nur ausgeführt wird, wenn die von ihr erzeugten Zusatzinformationen auch gespeichert werden können.

Der entwickelte Prototyp des Informationsmodells verwendet den Ansatz 3 a zur zentralen Speicherung der Zusatzinformationen in mehreren XML-Dateien: Für jede Komponente, die Zusatzinformationen erzeugt, wird eine eigene Datei angelegt (siehe auch die Beschreibung des Entwurfs in Anhang C auf Seite 111 bis 141).

Nachdem in diesem Abschnitt festgelegt wurde, wie die Speicherung der Zusatzinformationen erfolgt, wird im nächsten Abschnitt beschrieben, wie eine Zusatzinformation aufgebaut ist.

6.2 Datenmodell einer Zusatzinformation

In Abbildung 6.1 auf der nächsten Seite ist der Aufbau einer Zusatzinformation dargestellt. Die Namensgebung der Schnittstellen entspricht den Konventionen des Eclipse-Projekts (Eclipsepedia), nach denen der Name einer Schnittstelle mit dem Buchstaben I beginnt.

Daten zu einer Zusatzinformation, beispielsweise das Ergebnis der Berechnung einer Metrik, können in Form von Paaren (Schlüssel, Wert) gespeichert werden. Der Schlüssel ist eine innerhalb der Zusatzinformation eindeutig zu wählende Zeichenkette. Der Wert muss einen der primitiven Datentypen `boolean`, `double`, `float`, `int`, `long` oder `String` besitzen.

Neben den Daten enthält eine Zusatzinformation (Schnittstelle `IAnnotation`) immer einen oder mehrere Verweise auf Ressourcen. Auf einen Verweis wird über einen innerhalb der Zusatzinformation eindeutigen Namen zugegriffen. Die Namen können gleichzeitig dazu verwendet werden, die Bedeutung der Verweise deutlich zu machen, z. B. „Aufrufer“ und „Aufgerufener“.

Ein Verweis wird durch die Schnittstelle `ILink` dargestellt. Er bezieht sich immer auf eine Datei, ein Verzeichnis oder ein Projekt. Die Schnittstelle `IContainerLink` ermöglicht Verweise auf die bei Eclipse in der Schnittstelle `IContainer` zusammengefassten Verzeichnisse und Projekte. Verweise auf Dateien erfolgen über die Schnittstelle `IFileLink`.

Für jeden Dateityp, zu dem Zusatzinformationen gespeichert werden sollen, stellt ein Modul eine Schnittstelle zur Verfügung, die die Schnittstelle `IFileLink` erweitert.

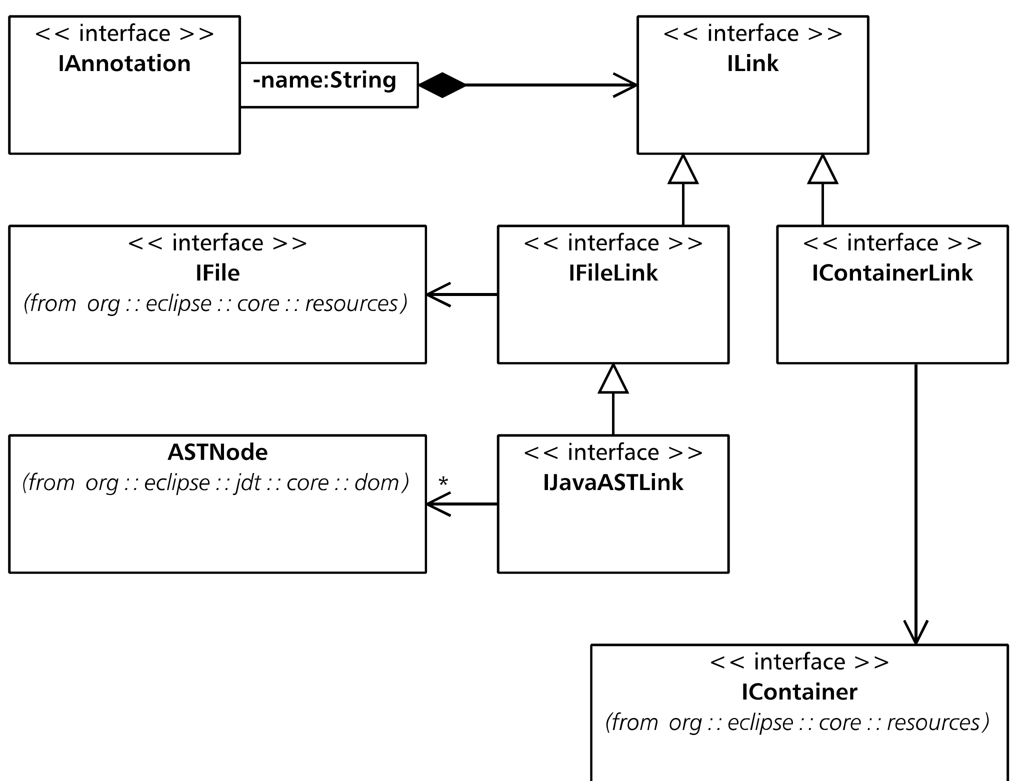


Abbildung 6.1: Klassendiagramm einer Zusatzinformation.
Anmerkung: Von Schnittstellen gehen keine Assoziationen aus. Diese Darstellung wurde gewählt, da sie mir leichter verständlich erscheint als die korrekte Angabe einer Methode (zum Beispiel `getLink(name:String):ILink`).

Die erweiternde Schnittstelle ermöglicht es, auf eine Stelle innerhalb der Datei zu verweisen. Je nach Dateityp kann die Angabe dieser Stelle unterschiedlich erfolgen, bei einem Dokument zum Beispiel über die Seitenzahl. Codation ist so entworfen, dass durch Hinzufügen weiterer Module zusätzliche Dateiformate unterstützt werden können.

6.2.1 Verweise in Java-Dateien

Eines dieser Module ist der „Java-Link-Provider“. Er ermöglicht über die Schnittstelle `IJavaASTLink` Verweise auf den Inhalt von Java-Dateien. Als Verweisziele dienen die Knoten des beim Parsen der Java-Datei erzeugten ASTs (Klasse `ASTNode`).

Außerdem können Sequenzen von AST-Knoten durch Angabe des ersten und des letzten Knotens der Sequenz als Verweisziele verwendet werden. Abschnitt 6.2.2 beschreibt, wie eine Auswahl in einem Texteditor auf eine solche Sequenz abgebildet werden kann.

Zur Speicherung eines Verweises bestimmt der Java-Link-Provider den Bereich, den das Verweisziel in der Java-Datei umfasst. Der Bereich ist definiert durch seine Startposition innerhalb der Datei und seine Länge. Die beiden Werte werden gespeichert und ermöglichen es, beim Laden des Verweises den gewählten AST-Knoten eindeutig zu bestimmen. Der von einem AST-Knoten umfasste Bereich wird vom Parser im AST hinterlegt und muss daher nicht von Codation bestimmt werden.

6.2.2 Bestimmung der markierten AST-Knoten

Da Verweise auf syntaktischer Ebene erfolgen, Java-Dateien aber weiterhin in der Textdarstellung bearbeitet werden, kann es notwendig sein, die mit einer Auswahl im Editor markierten AST-Knoten zu bestimmen. Dieser Abschnitt definiert deshalb einen Algorithmus, mit dem dies möglich ist. Der Algorithmus nutzt aus, dass die Knoten des AST den Bereich gespeichert haben, den sie in der Datei umfassen. Da eine Auswahl im Editor auf die gleiche Weise (Startposition und Länge) definiert ist, kann die Auswahl wie unten beschrieben auf AST-Knoten abgebildet werden.

Berechnungsergebnis

Das vom Algorithmus zu liefernde Ergebnis soll mit Hilfe des in Abbildung 6.2 auf der nächsten Seite abgebildeten Beispiels erläutert werden. Der Anwender hat zunächst einen Bereich ausgewählt, der in Zeile 8 im Wort „String“ beginnt und in Zeile 10 nach der geschweiften Klammer endet.

Das Ergebnis besteht aus den beiden AST-Knoten v_a (Anfang der Auswahl) und v_e (Ende der Auswahl) und soll so genau wie möglich sein. Daher gilt, dass der Bereich vom Beginn des Knotens v_a bis zum Ende des Knotens v_e die Auswahl vollständig enthalten muss, zugleich aber möglichst klein sein muss. Außerdem müssen die Knoten in der Hierarchie soweit oben stehen wie möglich, d. h. der Pfad zur Wurzel muss möglichst kurz sein.

Der Knoten „Simple Name$_2$“ und dessen (indirekte) Vaterknoten enthalten den Beginn der Auswahl. Die Knoten „Simple Name$_2$“ und „Simple Type“ erfüllen die Anforderung, dass der Beginn des Knotens v_a so dicht wie möglich vor dem Beginn

```java
public class Person {
  private String name;

  public Person(String name) {
    this.name = name;
  }

  public String toString() {
    return name;
  }
}
```

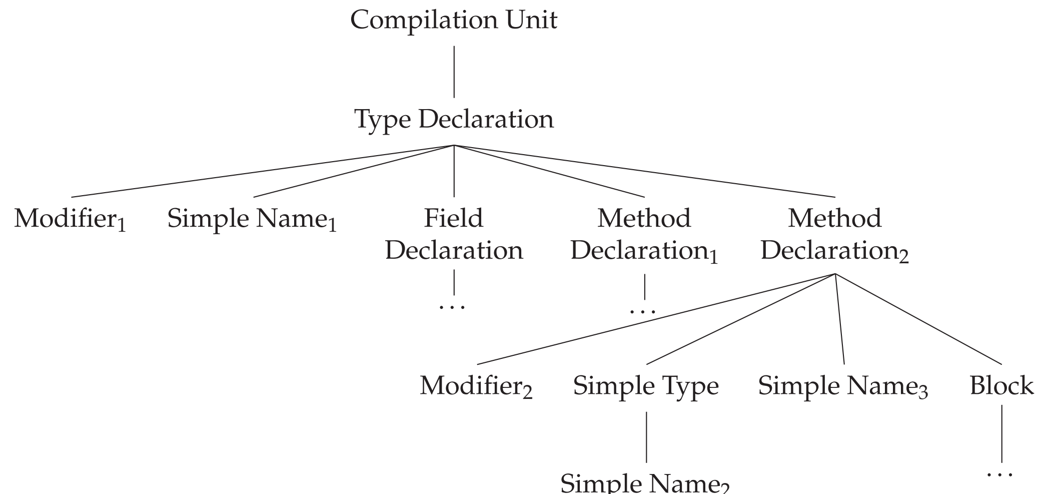


Abbildung 6.2: Beispiel für eine Java-Klasse und deren AST. Die Bezeichner der Knoten entsprechen denen der Java-Sprachdefinition (Gosling et al., 2005). Zur Unterscheidung wurden Knoten gleichen Typs entsprechend einer Preorder-Traversierung durchnummeriert.

der Auswahl liegen muss. Da der Knoten „Simple Type" in der Hierarchie jedoch weiter oben steht als der Knoten „Simple Name$_2$", wird er als Knoten v_a gewählt.

Analog dazu wird der Knoten „Block" als v_e gewählt. Sein Ende ist gleichzeitig auch das Ende der Auswahl. Der Knoten „Method Declaration$_2$" hat die gleiche Endposition wie der Knoten „Block". Der von den Knoten v_a und „Method Declaration$_2$" umfasste Bereich ist jedoch größer, weil der Beginn der „Method Declaration$_2$" vor dem Beginn des Knotens v_a liegt.

Der Anwender möchte nun die ganze Methode `toString` auswählen. Dazu hat er die Zeilen 8 bis 10 vollständig ausgewählt. Der Knoten „Method Declaration$_2$“ wird als Knoten v_a gewählt, da er möglichst kurz vor der Auswahl beginnt und in der Hierarchie am weitesten oben steht, ohne dadurch den umfassten Bereich unnötig zu vergrößern. Als Endknoten v_e wird ebenfalls der Knoten „Method Declaration$_2$“ gewählt. Da v_a und v_e identisch sind, wird nur ein Knoten zurückgegeben.

Ohne die Bedingung, dass die Knoten möglichst weit oben in der Hierarchie stehen müssen, würden die Knoten „Modifier$_2$“ und „Block“ als Resultat zurückgegeben werden. Dies würde aber der Intention des Anwenders widersprechen.

Algorithmus als Pseudocode

Die Methode `findeKleinstenKnoten` bestimmt rekursiv den Knoten auf tiefster Ebene des AST, der die übergebene Position enthält. Da ASTs zyklenfrei sind, terminiert die Rekursion spätestens, wenn sie auf ein Blatt trifft.

```
1  function findeKleinstenKnoten(n : Knoten, position : ℕ0)
2      return Knoten is
3  begin
4    foreach (k ∈ n.KindKnoten) loop
5      if ((position ≥ k.StartPos) ∧ (position ≤ k.StartPos + k.Länge))
6      then
7        return findeKleinstenKnoten(k, position);
8      end if;
9    end loop;
10
11   return n;
12 end;
```

Die Methode `findeStartKnoten` bestimmt den Knoten v_a, der die oben beschriebenen Anforderungen erfüllt. Die Konstante w bezeichnet die Wurzel des ASTs.

```
1  function findeStartKnoten return Knoten is begin
2    Knoten k := findeKleinstenKnoten(w, Auswahl.StartPos);
3    Knoten v := k;
4    while ((v ≠ w)
5        ∧ (v.StartPos = k.StartPos)
6        ∧ (v.StartPos + v.Länge ≤ Auswahl.StartPos + Auswahl.Länge))
7    loop
```

```
8      // Die Bewegung zur Wurzel darf die Genauigkeit nicht reduzieren.
9      k := v;
10     v := v.Vater;
11   end loop;
12   return k;
13 end;
```

Die Schleife in den Zeilen 4 bis 11 verfolgt den Pfad zur Wurzel zurück. Da die Schleife spätestens beim Erreichen der Wurzel terminiert, terminiert auch die Methode.

Die Methode `findeEndKnoten` zur Bestimmung des Knotens v_e ist ähnlich definiert wie die Methode `findeStartKnoten`. Auch sie terminiert spätestens beim Erreichen der Wurzel.

```
1  function findeEndKnoten return Knoten is begin
2    Knoten k := findeKleinstenKnoten(w,
3        Auswahl.StartPos + Auswahl.Länge);
4    Knoten v := k;
5    while ((v ≠ w)
6        ∧ (v.StartPos + v.Länge = Auswahl.StartPos + Auswahl.Länge)
7        ∧ (v.StartPos ≥ Auswahl.StartPos))
8    loop
9      // Die Bewegung zur Wurzel darf die Genauigkeit nicht reduzieren.
10     k := v;
11     v := v.Vater;
12   end loop;
13   return k;
14 end;
```

Die Hauptroutine des Algorithmus ist im folgenden Listing dargestellt.

```
1  Knoten v_a := findeStartKnoten;
2  Knoten v_e := findeEndKnoten;
3
4  if (v_a = v_e) then
5    return {v_a};
6  else
7    return {v_a, v_e};
8  end if;
```

Aufwandsabschätzung

Die Anzahl der Knoten im AST sei $n \in \mathbb{N}$. Der schlechteste Fall für die Methode `findeKleinstenKnoten` tritt ein, wenn

a) der AST zu einer Liste entartet ist, d. h. jeder Knoten (außer dem einzigen Blatt) genau einen Kindknoten hat, und
b) Code ausgewählt wurde, der in diesem Blatt enthalten ist.

In diesem Fall muss somit jeder Knoten einmal besucht werden. Der Aufwand ist somit $O(n)$.

Der Aufwand für die Schleifen in den Methoden `findeStartKnoten` und `findeEndKnoten` ist maximal, wenn jeweils bis zur Wurzel zurückiteriert werden muss. Da dabei jeder Knoten maximal einmal besucht wird, ist der Aufwand $O(n)$. Für die Methoden `findeStartKnoten` und `findeEndKnoten` ergibt sich damit jeweils ein Aufwand von $O(2\,n)$.

Der Aufwand für den gesamten Algorithmus beträgt somit $O(4\,n)$.

6.3 Synchronisation

Nachdem in den vorigen Abschnitten beschrieben wurde, wie eine Zusatzinformation aufgebaut ist und wie sie gespeichert wird, wird in diesem Abschnitt betrachtet, wie die gespeicherten Zusatzinformationen synchronisiert werden.

Die Synchronisation erfolgt nach Änderungen an annotiertem Quelltext und dient dazu, die Zusatzinformationen nachzuführen. Um die Synchronisation durchführen zu können, ist es also wichtig, Änderungen am Quelltext erkennen zu können. Dazu gehören neben inhaltlichen Änderungen an einer Datei auch Ereignisse wie das Anlegen, Löschen, Verschieben oder Umbenennen von Dateien und Verzeichnissen.

Um bei inhaltlichen Änderungen an einer Datei möglichst wenig Zusatzinformationen aktualisieren zu müssen, ist ein Mechanismus zur Bestimmung der geänderten Bereiche notwendig. Je detaillierter die Änderungsinformationen sind, desto mehr Aufwand kann bei der Synchronisation eingespart werden. Zu beachten ist jedoch, dass die Berechnung detaillierter Änderungsinformationen, insbesondere auf syntaktischer Ebene, nicht trivial ist und einen erheblichen Rechenaufwand verursachen kann (siehe Abschnitt 6.3.4 auf Seite 71 bis 77). Der Detaillierungsgrad der Änderungsinformationen sollte deshalb auf das benötigte Maß reduziert sein. Maletic

et al. (2005) sprechen in diesem Zusammenhang davon, dass die Granularitäten von Verweisen und Änderungsinformationen zusammenpassen müssen.

Ist zum Beispiel bekannt, dass nur eine Methode einer Klasse geändert wurde, können alle Zusatzinformationen, die nicht auf die Methode (oder ein die Methode umgebendes Konstrukt) verweisen, herausgefiltert werden, da sie nicht aktualisiert werden müssen. Enthalten die Änderungsinformationen jedoch nur die Information, dass die Klasse geändert wurde, muss auch bei Zusatzinformationen, die auf andere Methoden verweisen, die Aktualisierung angestoßen werden. In diesem Fall ist die Granularität der Änderungsinformationen zu gering.

Ist die Granularität der Änderungsinformationen zu hoch, werden Informationen berechnet, obwohl dadurch keine weiteren Zusatzinformationen herausgefiltert werden können. Beziehen sich zum Beispiel alle Zusatzinformationen auf Klassen, verringert die Information, dass in einer Klasse die Reihenfolge zweier Methoden vertauscht wurde, die Anzahl der zu aktualisierenden Zusatzinformationen nicht.

Ein in Maletic et al. (2005) nicht beachteter Aspekt ist jedoch, dass die Aktualisierung der Zusatzinformationen die zuvor nicht benötigten Details verwerten kann. Wenn nur die Reihenfolge zweier Methoden vertauscht wurde, müssen die meisten Metriken nicht neu berechnet werden. Damit kann die Berechnungsdauer gesenkt werden, obwohl sich die geänderte Zusatzinformation auf die Klasse bezieht.

Bei der Auswahl der Algorithmen und den Überlegungen zum benötigten Rechenaufwand muss beachtet werden, wann und vor allem wie oft die Synchronisation durchgeführt wird. Alle durch die Synchronisation ausgelösten Berechnungen müssen abgeschlossen sein, bis die Datei das nächste Mal synchronisiert wird.

6.3.1 Methoden zur Bestimmung von Änderungsinformationen

Um Änderungsinformationen zu bestimmen, gibt es zwei Methoden:

1. Der Editor zur Bearbeitung des Quelltextes speichert, an welchen Stellen der Benutzer welche Änderungen vorgenommen hat. Zu bestimmten Zeitpunkten, beispielsweise beim Speichern der Datei, wird eine Nachricht mit den protokollierten Änderungsinformationen verschickt.
2. Der Stand des Quelltextes vor der Änderung wird gespeichert. Nach der Änderung wird dieser Stand mit dem aktuellen Stand verglichen.

Die erste Methode hat den Vorteil, dass keine Kopien der annotierten Dateien verwaltet werden müssen. Da bekannt ist, an welchen Stellen Änderungen vorgenommen wurden, ist zudem der Aufwand zur Bestimmung der Änderungsinformationen geringer, weil nicht die ganze Datei nach Änderungen durchsucht werden muss.

Die Methode hat jedoch den schwerwiegenden Nachteil, dass nur im Editor vorgenommene Änderungen erfasst werden. Bietet das Annotationswerkzeug beispielsweise einen Mechanismus zum Suchen und Ersetzen in mehreren Dateien an, der nicht den Editor verwendet, bleiben diese Änderungen unerkannt. Die erste Methode muss also für extern durchgeführte Änderungen die zweite Methode benutzen. Andernfalls müsste sie sich auf die Angabe beschränken, dass die Datei geändert wurde, ohne Details über den Inhalt der Änderung zu liefern.

Der Vorteil der zweiten Methode liegt darin, dass nach jeder Veränderung der Datei eine Berechnung der Änderungsinformationen möglich ist. Voraussetzung für den Einsatz der Methode ist, dass erkannt wird, dass sich eine Datei geändert hat. Der Nachteil der zweiten Methode ist, dass Datei und Kopie immer vollständig verglichen werden müssen.

Weil der erste Ansatz nur bei im Editor vorgenommenen Änderungen Änderungsinformationen bestimmen kann, Eclipse jedoch zahlreiche weitere Funktionen anbietet, die eine Datei verändern können (Suchen und Ersetzen, Refactoring, . . .), wird bei Codation die zweite Methode verwendet. Bei der Berechnung der Änderungsinformationen wird also eine Datei mit ihrer vorigen Version verglichen. Um die vorige Version der Datei zu erhalten, bieten sich zunächst drei Möglichkeiten an:

1. Verwendung des Versionsverwaltungssystems
2. Verwendung der local history von Eclipse, in der Eclipse alte Versionen von Dateien festhält.
3. Eigene Verwaltung von Kopien.

Die erste Möglichkeit scheidet aus, da über die Versionsverwaltung nur der zuletzt eingecheckte Stand ermittelt werden kann, nicht aber der Stand vor der letzten Änderung. Außerdem funktioniert die Möglichkeit nicht, wenn ein Projekt kein Versionsverwaltungssystem einsetzt oder dieses vom Annotationswerkzeug nicht unterstützt wird.

Die local history kann ebenfalls nicht verwendet werden. Die letzte Version einer Datei ist nur verfügbar, wenn sie bereits zuvor in Eclipse bearbeitet wurde. Bei neu angelegten oder gerade importierten Dateien liegt also noch keine alte Version vor, was nach der ersten Änderung eine Berechnung von Änderungsinformationen

unmöglich macht. Da die local history nicht unter der Versionsverwaltung steht, kann nach der Aktualisierung der Dateien aus der Versionsverwaltung nur mit dem zuletzt lokal gespeicherten Stand verglichen werden und nicht mit dem Stand, zu dem die Zusatzinformationen erhoben wurden. Außerdem hat der Benutzer von Eclipse die Möglichkeit, die local history zu deaktivieren.

Somit bleibt nur die bei Codation verwendete Möglichkeit, selbst Kopien anzufertigen und zu verwalten. Dadurch erhöht sich jedoch der benötigte Speicherplatz.

Im nächsten Abschnitt wird beschrieben, wie in Eclipse erkannt werden kann, dass eine Datei geändert wurde und deshalb die Synchronisation der Zusatzinformationen und die Aktualisierung der Kopien nötig sind.

6.3.2 Ansatzpunkte zur Änderungserkennung bei Eclipse

Eclipse bietet verschiedene Funktionen an, mit denen auf Änderungen an Dateien reagiert werden kann. Da Codation als Eclipse-Plugin realisiert werden soll, wird in diesem Abschnitt ihre Verwendbarkeit geprüft.

Begriffserklärung 6.2 (Listener) *Mit dem Begriff Listener werden Objekte bezeichnet, die über den Eintritt bestimmter Ereignisse benachrichtigt werden (vergleiche Publisher-Subscriber-Pattern).*

Resource-Change-Listener

Die Schnittstelle `IResourceChangeListener` wird vom Resources-Plugin (`org.eclipse.core.resources`) zur Verfügung gestellt. Der Listener muss beim Resources-Plugin für bestimmte Ereignisse registriert werden und wird dann von diesem aufgerufen, sobald bei einer Ressource (siehe Begriffserklärung 6.1 auf Seite 55) eines dieser Ereignisse eingetreten ist.

Eclipse lädt Plug-Ins erst bei Bedarf (lazy loading). Weil ein Plug-In den Listener erst registrieren kann, nachdem es geladen wurde, bleiben alle zuvor eingetretenen Ereignisse zunächst unbemerkt. Daher wird ein Mechanismus benötigt, um diese Ereignisse abzurufen. Dazu kann sich ein Plug-In als „Workspace-Save-Participant“ registrieren. Beim Laden des Plug-Ins können dann alle seit der letzten Beendigung des Plug-Ins eingetretenen Ereignisse abgerufen werden.

Allerdings entsteht so eine zeitliche Differenz zwischen dem Eintritt eines Ereignisses und seiner Verarbeitung. Durch Erweitern des Extension-Points `org.eclipse`

`.ui.startup` muss das Plug-In also erzwingen, beim Start von Eclipse geladen zu werden. Da der Benutzer in der Konfiguration in der Lage ist, den automatischen Start zu deaktivieren, kann sich ein Plug-In nicht darauf verlassen, tatsächlich beim Start von Eclipse geladen zu werden.

Der Listener empfängt Ereignisse als Resource-Change-Event. Die Angabe der Änderungen ist auf Ressourcen-Ebene beschränkt. Es kann also nur erkannt werden, dass eine Datei geändert wurde. Die am Inhalt vorgenommenen Änderungen müssen vom Plug-In selbst ermittelt werden. Dazu ist die zweite der zuvor beschriebenen Methoden notwendig.

Eine weitere wesentliche Einschränkung ist, dass während der Verarbeitung der Ereignisse keine Änderungen am Workspace vorgenommen werden dürfen. Änderungen an den Zusatzinformationen können zu diesem Zeitpunkt also nicht gespeichert werden. Außerdem müssen für die Berechnung der Änderungsinformationen Kopien erstellt werden. Weil der Workspace gesperrt ist, ist dies nur asynchron möglich.

Da nicht garantiert ist, die Resource-Change-Events zeitnah zu ihrem Eintritt verarbeiten zu können und der gesperrte Workspace Operationen auf Dateien wesentlich erschwert, eignet sich der Resource-Change-Listener nicht für die Anwendung in einem Annotationswerkzeug.

Element-Changed-Listener

Das JDT bietet die Möglichkeit, einen `IElementChangedListener` zu registrieren. Der Listener wird benachrichtigt, wenn sich am vom JDT verwalteten Modell der Java-Elemente etwas ändert. Das in Abbildung 6.3 auf der nächsten Seite dargestellte Modell ist jedoch weit weniger detailliert als ein vollständiger AST. So wird beispielsweise eine Methode nur als Ganzes im Modell repräsentiert und ihr Inhalt nicht weiter strukturiert.

Das JDT verwendet zur Änderungserkennung die erste der in Abschnitt 6.3.1 erwähnten Methoden: Es merkt sich, welche Änderungen mit dem Java-Editor oder anderen Werkzeugen des JDT durchgeführt wurden. Da keine Kopien erstellt werden, um Vergleiche durchzuführen, erhält der Listener bei nicht mit dem JDT durchgeführten Änderungen nur die Information, dass die Datei geändert wurde.

Wie bei Resource-Change-Listenern gilt auch hier, dass Ereignisse erst empfangen werden, nachdem das Plug-In geladen wurde und den Listener registriert hat. Die

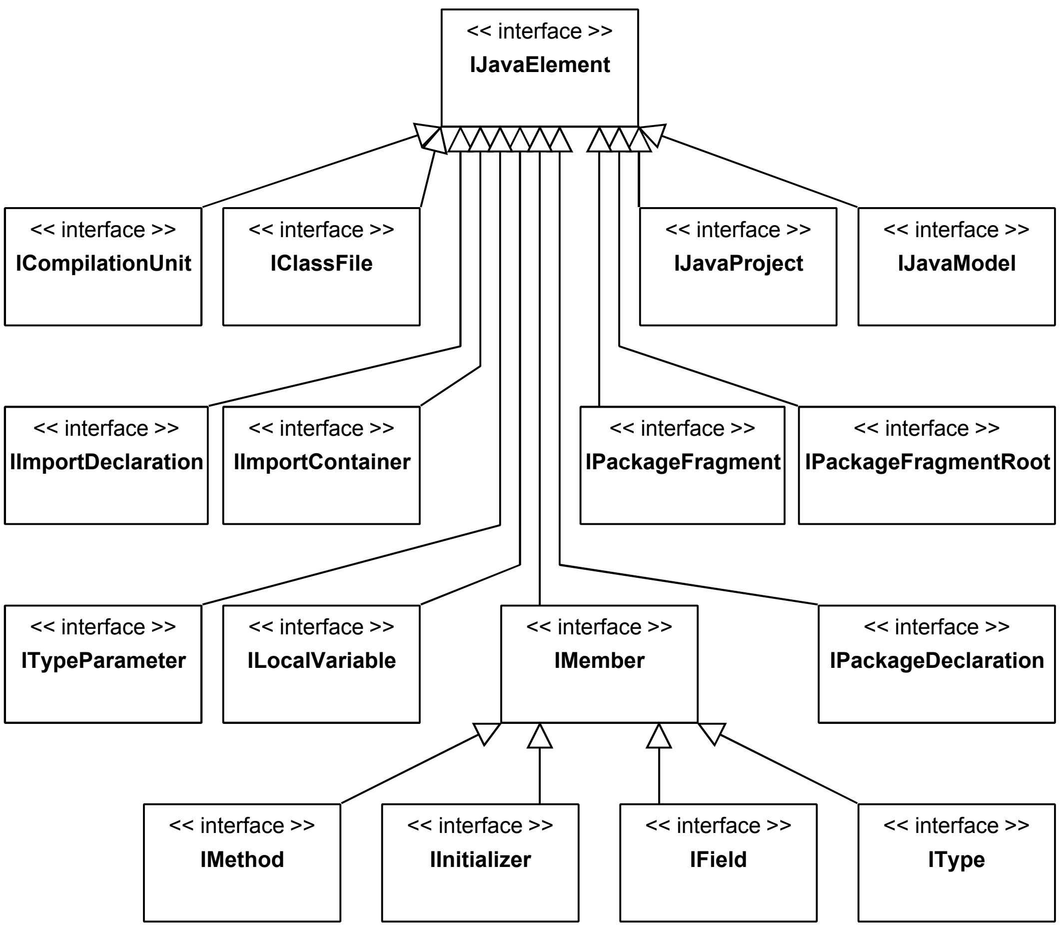


Abbildung 6.3: Klassendiagramm `IJavaElement` (aus Gründen der Übersichtlichkeit werden nur Vererbungsbeziehungen dargestellt)

Möglichkeit, sich zusätzlich als Workspace-Save-Participant zu registrieren, besteht an dieser Stelle jedoch nicht.

Der Listener wird bei zwei Ereignissen über die Änderungen informiert:

1. Nach der Aktualisierung des Editors (reconciling): Dieses Ereignis tritt unter anderem nach jeder Cursorbewegung ein. Damit eignet es sich wegen der Häufigkeit des Eintritts nicht als Auslöser für aufwändige Berechnungen.
2. Nach der Speicherung von Änderungen: Wie bei Resource-Change-Listenern, die zu diesem Zeitpunkt ebenfalls benachrichtigt werden, ist der Workspace gesperrt. Dateien können also nicht geändert werden.

Da Operationen auf Dateien wegen des gesperrten Workspace wesentlich erschwert sind, Änderungsinformationen nur wenig detailliert vorliegen und nur Java-Dateien betrachtet werden, eignet sich der Element-Change-Listener nicht für die Anwendung in einem Annotationswerkzeug.

Compilation-Participant

Über den Extension-Point `org.eclipse.jdt.core.compilationParticipant` des JDT kann sich ein Plug-In in den Übersetzungsvorgang einhängen. Der Compilation-Participant wird aufgerufen, wenn eine Java-Datei übersetzt werden soll. Es stehen jedoch keine Änderungsinformationen zur Verfügung.

Der Compilation-Participant hat den Vorteil, Dateien verändern zu können, da der Workspace nicht gesperrt ist. Sein Einsatz ist aber auf das JDT beschränkt. Außerdem erhält er keine Nachricht, wenn beispielsweise eine Datei gelöscht oder ein Verzeichnis angelegt wurde.

Wegen der Beschränkung auf die Programmiersprache Java und der Tatsache, dass viele Operationen auf Dateien und Verzeichnissen unbemerkt bleiben, kann auch der Compilation-Participant nicht sinnvoll in einem Annotationswerkzeug verwendet werden.

Project-Builder

Der Extension-Point `org.eclipse.core.resources.builders` bietet die Möglichkeit, eigene Übersetzer als Project-Builder bei einem Projekt zu registrieren. Nach der Registrierung ist der Einsatz eines Project-Builders unabhängig vom Ladezustand des Plug-Ins, das ihn registriert hat.

Project-Builder arbeiten inkrementell, das bedeutet, dass nur geänderte Dateien neu übersetzt werden. Dazu erhält ein Project-Builder bei seinem Aufruf durch Eclipse die gleichen Änderungsinformationen wie ein Resource-Change-Listener. Da der Workspace beim Aufruf nicht gesperrt ist, können Ressourcen verändert werden. Weitere Vorteile eines Project-Builders sind, dass er unabhängig vom JDT und somit auch für andere Programmiersprachen verwendbar ist, sowie die Tatsache, dass die Registrierung für jedes Projekt einzeln vorgenommen werden kann.

Bedingt durch die Unabhängigkeit vom Dateiformat besteht wie auch bei den Resource-Change-Listenern der Nachteil, dass Informationen über Änderungen

am Inhalt einer Datei selbst bestimmt werden müssen. Project-Builder erfordern deswegen die Verwendung der zweiten Methode zur Ermittlung detaillierter Änderungsinformationen.

Project-Builder sind für die Verwendung in einem Annotationswerkzeug geeignet, da sie es ermöglichen, auf Änderungen an Ressourcen zu reagieren und dabei Ressourcen zu verändern.

6.3.3 Synchronisation in Codation

Zur Synchronisation verwendet Codation einen Project-Builder, da die anderen Möglichkeiten wegen der beschriebenen Einschränkungen ausscheiden. Änderungsinformationen werden mit der zweiten Methode bestimmt, d. h. der neue Zustand einer Datei wird mit dem zuvor gesicherten verglichen. Die Verwaltung der Kopien wird vom Project-Builder übernommen.

Die Synchronisation bei Änderungen am Inhalt einer Datei läuft wie folgt ab:

1. Eclipse startet nach einer Änderung automatisch den Project-Builder. Der Project-Builder arbeitet in einem eigenen Thread im Hintergrund, wodurch der Benutzer seine Arbeit während der Synchronisation fortsetzen kann.
2. Das Modul, das Verweise auf den Inhalt der geänderten Datei erlaubt, wird vom Project-Builder aufgerufen, um Änderungsinformationen zu bestimmen. Dazu vergleicht das Modul den aktuellen Inhalt mit dem in der Kopie der Datei gesicherten Inhalt.
3. Der Project-Builder ermittelt die von der Änderung betroffenen Zusatzinformationen.
4. Die Komponenten, die diese Zusatzinformationen erzeugt haben, werden vom Project-Builder benachrichtigt. Sie erhalten die Änderungsinformationen, um die Zusatzinformationen zu aktualisieren.
5. Nach der Aktualisierung der Zusatzinformationen aktualisiert der Project-Builder die Kopie der geänderten Datei.

6.3.4 Syntaktischer Vergleich von Java-Dateien

Da Verweise in Java-Dateien bei Codation auf AST-Knoten erfolgen, müssen die Änderungsinformationen ebenfalls auf syntaktischer Ebene, d. h. auf AST-Ebene, berechnet werden. Bei der Berechnung der Änderungsinformationen auf AST-Ebene

muss ermittelt werden, mit welchen Änderungen der alte AST in den neuen AST überführt werden kann. Die Problemstellung entspricht dem *Tree Edit Distance Problem* (auch bekannt als *Tree-to-Tree Correction Problem*).

Definition 6.1 (Geordneter Baum, ungeordneter Baum) *Ein geordneter Baum ist ein Baum, bei dem die Kinder jedes Knoten untereinander eine feste Reihenfolge besitzen. Andernfalls ist ein Baum ungeordnet.*

Bei AST-Knoten entspricht die Reihenfolge von Geschwister-Knoten der Reihenfolge, in der sie im Quelltext definiert sind.

Die Berechnung einer Lösung für ein Tree Edit Distance Problem ist bei ungeordneten Bäumen nach Zhang et al. (1992) MAX-SNP-hart. Für die Verwendung in Codation eignen sich ungeordnete Bäume also nicht. Für geordnete Bäume existieren mehrere Algorithmen (siehe Bille (2005)); Valiente (2002) beschreibt einen Algorithmus, der das Problem in $O(n_1 n_2)$ Zeit und $O(n_1 n_2)$ zusätzlichem Platz löst (n_1 und n_2 sind die Anzahl der Knoten in zu vergleichenden Bäume). Dieser Algorithmus wird bei Codation verwendet und im Folgenden beschrieben.

Definition 6.2 (Änderungs-Operation) $T_1 = (V_1, E_1)$ *und* $T_2 = (V_2, E_2)$ *seien geordnete Bäume. Eine Änderungs-Operation auf* T_1 *oder* T_2 *ist entweder*

- *das Entfernen eines Blattes* $v \in V_1$ *aus* T_1*, dargestellt durch* (v, λ)*,*
- *das Ersetzen eines Knotens* $v \in V_1$ *durch einen Knoten* $w \in V_2$*, dargestellt durch* (v, w)*, oder*
- *das Einfügen eines Blattes* $w \notin V_2$ *in* T_2*, dargestellt durch* (λ, w)*.*

Das Entfernen eines Knotens v *aus* T_1 *impliziert das Entfernen der Kante* $(\mathrm{parent}[v], v) \in E_1$ *(sofern* v *nicht Wurzel ist). Das Einfügen eines Knotens* $w \notin V_2$ *als Kind eines Blattes* $\mathrm{parent}[w] \in V_2$ *impliziert das Einfügen einer Kante* $(\mathrm{parent}[w], w) \notin E_2$ *(sofern* w *nicht Wurzel ist).*

Lösch- und Einfüge-Operationen werden gemäß Definition 6.2 nur auf Blättern durchgeführt. Das Entfernen eines Knotens v, der kein Blatt ist, erfordert deshalb das vorherige Entfernen aller Kindknoten von v. Analog dazu kann ein Knoten w erst eingefügt werden, wenn sein Vaterknoten existiert.

Über eine Sequenz von Änderungs-Operationen kann ein Baum in einen anderen überführt werden (siehe Beispiel in Abbildung 6.4 auf der nächsten Seite).

Definition 6.3 (Geordnete Relation) *Eine geordnete Relation* $R \subseteq A \times B$ *ist eine geordnete Menge (Sequenz) von geordneten Paaren* (a, b) *mit* $a \in A$ *und* $b \in B$*.*

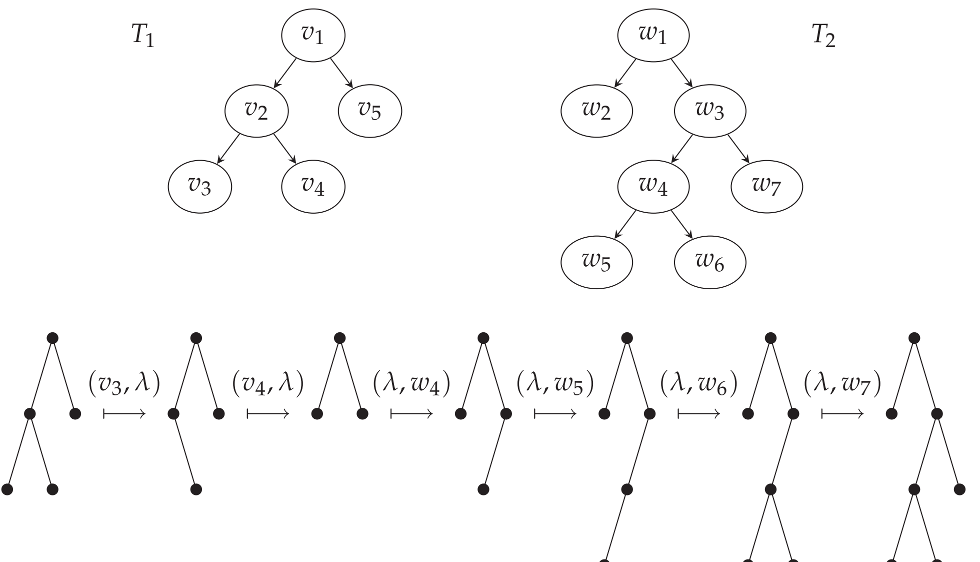


Abbildung 6.4: Transformation zwischen zwei geordneten Bäumen. Übernommen aus Valiente (2002).

Definition 6.4 (Transformation) *Es seien $T_1 = (V_1, E_1)$ und $T_2 = (V_2, E_2)$ geordnete Bäume. Eine Transformation von T_1 in T_2 ist eine geordnete Relation E mit $E \subseteq (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\})$, für die gilt:*

- $\{v \in V_1 \mid (v, w) \in E, w \in V_2 \cup \{\lambda\}\} = V_1$
  (alle Knoten aus V_1 sind in der Relation enthalten)
- $\{w \in V_2 \mid (v, w) \in E, v \in V_1 \cup \{\lambda\}\} = V_2$
  (alle Knoten aus V_2 sind in der Relation enthalten)
- *Für alle v_1, v_2, w mit $v_1, v_2 \in V_1 \cup \{\lambda\}$ und $w \in V_2$ gilt:*
  $(v_1, w) \in E \wedge (v_2, w) \in E \rightarrow v_1 = v_2$
  (zwei Knoten dürfen nicht auf denselben Knoten aus T_2 abgebildet werden)
- *Für alle v, w_1, w_2 mit $v \in V_1$ und $w_1, w_2 \in V_2 \cup \{\lambda\}$ gilt:*
  $(v, w_1) \in E \wedge (v, w_2) \in E \rightarrow w_1 = w_2$
  (ein Knoten aus T_1 darf nicht auf zwei Knoten abgebildet werden)

Damit eine Transformation E gültig ist, müssen die Vater- und die Geschwister-Reihenfolge beibehalten werden. Dies stellt sicher, dass das Ergebnis der Transformation auch ein geordneter Baum ist. Beibehaltung der Vater-Reihenfolge bedeutet, dass der Vaterknoten eines Nicht-Wurzel-Knoten v aus T_1, der durch einen Nicht-Wurzel-Knoten w aus T_2 ersetzt wird ($(v, w) \in E$), durch den Vaterknoten von w

ersetzt werden muss. Im Beispiel in Abbildung 6.4 auf der vorherigen Seite wird der Knoten v_5 durch den Knoten w_3 ersetzt. Deshalb muss der Vaterknoten von v_5, also v_1, durch w_1 ersetzt werden. Außerdem muss die Ersetzung von Geschwister-Knoten aus T_1 durch Geschwister-Knoten aus T_2 deren jeweilige relative Reihenfolge erhalten. Im Beispiel dürfen also nicht v_2 durch w_3 und v_5 durch w_2 ersetzt werden, auch wenn dies Änderungs-Operationen einsparen würde.

Definition 6.5 (Kostenfunktion γ) *Es seien $T_1 = (V_1, E_1)$ und $T_2 = (V_2, E_2)$ geordnete Bäume. Die Kosten einer Änderungs-Operation auf T_1 und T_2 werden durch eine Funktion $\gamma : (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\}) \rightarrow \mathbb{R}_0^+$ angegeben.*

Die Kosten einer Transformation $E \subseteq (V_1 \cup \{\lambda\}) \times (V_2 \cup \{\lambda\})$ von T_1 nach T_2 sind definiert als $\gamma(E) = \sum_{(v,w)\in E} \gamma(v, w)$.

Die Kosten der Änderungs-Operationen sind meist so definiert, dass $\gamma(v, w) = 1$ ist, falls entweder $v = \lambda$ oder $w = \lambda$ ist. Sonst gilt $\gamma(v, w) = 0$.

Definition 6.6 (Editierabstand) *Der Editierabstand (edit distance) zwischen zwei geordneten Bäumen T_1 und T_2 ist definiert als $\delta(T_1, T_2) = \min\{\gamma(E)|E$ ist eine gültige Transformation von T_1 in $T_2\}$.*

Das Tree Edit Distance Problem kann als gitterförmiger Editiergraph dargestellt werden. Dies reduziert das Problem, eine gültige Transformation zu finden, darauf, einen Weg von der linken oberen Ecke in die rechte untere Ecke zu finden. Mit Hilfe der Kostenfunktion γ und des Dijkstra-Algorithmus (Dijkstra, 1959) kann so der Editierabstand berechnet werden. Die Korrektheit des Ansatzes wird in Valiente (2002, S. 63 ff.) bewiesen.

Definition 6.7 (Editiergraph) *Es seien $T_1 = (V_1, E_1)$ und $T_2 = (V_2, E_2)$ geordnete Bäume. Der Editiergraph von T_1 und T_2 (Kantenmenge sei E_G) enthält einen Knoten der Form vw für jedes Paar von Knoten $v \in \{v_0\} \cup V_1$ und $w \in \{w_0\} \cup V_2$, wobei $v_0 \notin V_1$ und $w_0 \notin V_2$ Dummy-Knoten sind. Weiter gilt für $0 \leq i \leq n_1$ und $0 \leq j \leq n_2$ (wobei die Knoten in der Reihenfolge nummeriert sind, in der sie bei einer Preorder-Traversierung der Bäume besucht würden):*

- $(v_i w_j, v_{i+1} w_j) \in E_G$ *genau dann, wenn* $\mathrm{depth}[v_{i+1}] \geq \mathrm{depth}[w_{j+1}]$
- $(v_i w_j, v_{i+1} w_{j+1}) \in E_G$ *genau dann, wenn* $\mathrm{depth}[v_{i+1}] = \mathrm{depth}[w_{j+1}]$
- $(v_i w_j, v_i w_{j+1}) \in E_G$ *genau dann, wenn* $\mathrm{depth}[v_{i+1}] \leq \mathrm{depth}[w_{j+1}]$

Außerdem gilt $(v_i w_{n_2}, v_{i+1} w_{n_2}) \in E_G$ für $0 \leq i < n_1$, und $(v_{n_1} w_j, v_{n_1} w_{j+1}) \in E_G$ für $0 \leq j < n_2$.

Im Editiergraph repräsentiert eine senkrechte Kante der Form $(v_i w_j, v_{i+1} w_j)$ das Entfernen von v_{i+1} aus T_1. Eine diagonale Kante $(v_i w_j, v_{i+1} w_{j+1})$ stellt das Ersetzen des Knoten v_{i+1} aus T_1 durch den Knoten w_{j+1} aus T_2 dar. Eine horizontale Kante $(v_i w_j, v_i w_{j+1})$ repräsentiert das Einfügen eines Knotens w_{j+1} in T_2. In Abbildung 6.5 ist ein Beispiel für einen Editiergraphen dargestellt.

Der Weg durch den Editiergraphen, der die geringsten Kosten verursacht, repräsentiert die Transformation, aus der letztlich die Änderungsinformationen abgeleitet werden.

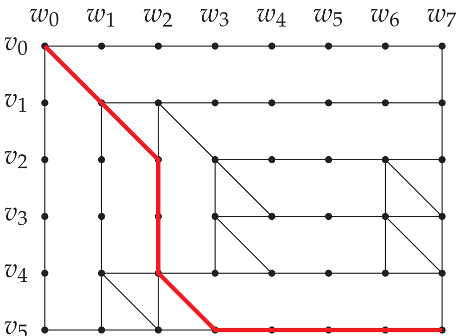


Abbildung 6.5: Editiergraph für die Bäume aus Abbildung 6.4. Der hervorgehobene Pfad entspricht der dort dargestellten Transformation. Übernommen aus Valiente (2002).

Verweise auf Knoten-Sequenzen und Änderungen

Nach der Berechnung der Änderungsinformationen müssen Verweise auf Sequenzen von AST-Knoten, wie sie in Abschnitt 6.2.1 auf Seite 59 f. beschrieben werden, auf ihre Konsistenz hin überprüft werden. Eine Sequenz ist definiert durch ihren Namen, den Start- und den Endknoten. Von Änderungen am Quelltext kann die Sequenz auf die folgenden Arten betroffen sein:

1. Der Start- oder der Endknoten wurde gelöscht.
2. Die Reihenfolge von Code wurde vertauscht. Dadurch liegt der Endknoten jetzt vor dem Startknoten. In Abbildung 6.6 auf der nächsten Seite ist ein Beispiel dargestellt.
3. Zwischen Start- und Endknoten wurden Knoten hinzugefügt, gelöscht oder ersetzt.

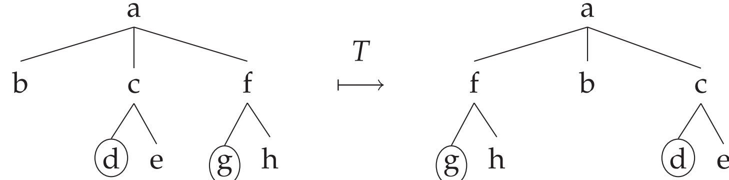


Abbildung 6.6: Die Knoten d und g bilden eine Sequenz, die die Knoten d, e und g umfasst. Bei einer Änderung T wurde Knoten f an den Anfang verschoben. Die Sequenz zwischen d und g umfasst somit die Knoten g, h, b und d, aber nicht mehr den Knoten e.

Über diese Ereignisse wird in den Änderungsinformationen benachrichtigt. Im ersten und im zweiten Fall wird die Sequenz gelöscht, da sie bei einem fehlenden Knoten oder bei einer Vertauschung der Reihenfolge kaum sinnvoll weiterverwendet werden kann. Die Komponente, die den Verweis auf die Sequenz von Knoten angelegt hat, muss also eine neue Sequenz anlegen. Im dritten Fall wird die Sequenz nicht entfernt. Die Komponente kann die Sequenz bei Bedarf anpassen.

Anpassungen am Algorithmus

Der zuvor beschriebene Algorithmus kann eine Änderung der Reihenfolge zweier AST-Knoten (z. B. die Vertauschung der Reihenfolge zweier Methoden) nicht erkennen. Bei der Implementierung des Prototyps stellte sich außerdem heraus, dass der Speicherbedarf zu groß ist. Der AST von großen Klassen kann leicht eine fünfstellige Anzahl an AST-Knoten enthalten. Da der Editiergraph quadratisch viele Knoten enthält, umfasst der Editiergraph für zwei Klassen mit jeweils 10 000 AST-Knoten 100 Millionen Knoten. Der Editiergraph enthält im schlimmsten Fall, in dem alle Knoten der verglichenen ASTs (bis auf die jeweiligen Wurzeln) die gleiche Entfernung zur Wurzel besitzen, die dreifache Menge an Kanten.

Mit dieser Knoten- und Kantenanzahl lässt sich der benötigte Speicherplatz abschätzen. Ein Knoten des Editiergraphen muss speichern, auf welches Paar von AST-Knoten er sich bezieht. Eine Kante muss speichern, welche Knoten des Editiergraphen sie verbindet und welches Kantengewicht sie besitzt. Auf einem 32 Bit-System benötigt der oben beschriebene Editiergraph also

$$10^8 \text{ Knoten} \cdot 8 \,\frac{\text{Byte}}{\text{Knoten}} + 3 \cdot 10^8 \text{ Kanten} \cdot 12 \,\frac{\text{Byte}}{\text{Kante}} = 4{,}4 \cdot 10^9 \text{ Byte.}$$

Da ein Speicherbedarf von 4,4 GByte zuzüglich des Speicherbedarfs für Verwaltungsinformationen nicht praktikabel ist, musste der Algorithmus angepasst werden. In der angepassten Version arbeitet der Algorithmus in zwei Modi, in denen jeweils nur Teile des ASTs miteinander verglichen werden. Im ersten Modus werden die Kindknoten von Typdeklarationen nicht in den Editiergraphen aufgenommen. Stösst der Algorithmus in diesem Modus auf Typdeklarationen, wird der Vergleichsalgorithmus im zweiten Modus aufgerufen.

Im zweiten Modus wird zu jeder in der Typdeklaration des ersten ASTs enthaltenen Deklaration (Deklarationen sind Methoden, Attribute und lokale Typen) die dazugehörende Deklaration des zweiten ASTs gesucht. Die Suche erfolgt über die Position im AST und über Signaturen. Bei Methoden besteht die Signatur aus dem Namen der Methode und ihren Parametern, bei Attributen und lokalen Typen aus ihrem Namen. Die beiden Deklarationen werden danach auf Gleichheit geprüft. Dieser Vergleich auf Isomorphie wird vom JDT angeboten, benötigt keinen zusätzlichen Speicherplatz und hat einen maximalen Zeitaufwand von $O(n)$. Bei unveränderten Deklarationen kann somit Laufzeit eingespart werden. Unterscheiden sich die beiden Deklarationen, werden durch einen rekursiven Aufruf des Vergleichsalgorithmus im ersten Modus Änderungsinformationen berechnet.

Durch diese Anpassungen werden die Dauer für die Berechnung der Änderungsinformationen und der benötigte Speicherplatz verringert, da nur Teile der Datei verglichen werden. Außerdem wurde der Algorithmus so angepasst, dass eine Änderung der Reihenfolge der Deklarationen erkannt wird. In Anhang B auf Seite 105 bis 109 sind die Ergebnisse von Messungen beschrieben, bei denen die für den Vergleich benötigte Zeit gemessen wurde.

6.4 Realisierung des Informationsmodells

Nach der Entwicklung des in den Abschnitten 6.1 bis 6.3 beschriebenen Informationsmodells erfolgt die Realisierung als Eclipse-Plugin. Dazu wird zunächst ein Entwurf erstellt, der in Anhang C beschrieben ist. Anschließend erfolgt die Implementierung, die mit dem in Anhang D enthaltenen Testplan geprüft wird.

Nach der Implementierung werden die Eigenschaften von Codation mit dem in Kapitel 4 definierten Beschreibungsschema zusammengefasst dargestellt und die Vorteile gegenüber den in der Literaturrecherche untersuchten Informationsmodellen beschrieben (nächster Abschnitt). Im nächsten Kapitel wird der Nutzen von Codation untersucht. Außerdem werden Metriken erhoben, um Aussagen über

den Umfang der Implementierung treffen zu können und um Schwachstellen im Entwurf oder der Umsetzung zu suchen. Die Erhebung der Metriken wird in Anhang A beschrieben. Um den Ressourcenbedarf des in Abschnitt 6.3.4 erläuterten Algorithmus zur Bestimmung von Änderungsinformationen für Java-Dateien zu ermitteln, wurde eine Laufzeitmessung durchgeführt, die in Anhang B beschrieben ist.

6.5 Zusammenfassung der Eigenschaften von Codation

Um einen Vergleich von Codation mit den in der Literaturrecherche untersuchten Werkzeugen zu ermöglichen und die Eigenschaften von Codation zusammenzufassen, werden die Eigenschaften in diesem Abschnitt mit dem in Kapitel 4 auf Seite 27 bis 36 definierten Schema beschrieben. Das Ergebnis ist in Tabelle 6.1 auf Seite 80 dargestellt. Die Eigenschaften der in Kapitel 5 vorgestellten Werkzeuge werden in den Tabellen 5.1 und 5.2 auf Seite 42 f. gegenübergestellt.

Die Besonderheiten von Codation im Vergleich zu den in der Literaturrecherche untersuchten Werkzeugen sind:

1. Es ist möglich Zusatzinformationen zu speichern, die sich auf Verzeichnisse beziehen. Dadurch ist es beispielsweise möglich, Berechnungsergebnisse zu aggregieren. Diese Möglichkeit fehlt den in der Literaturrecherche untersuchten Werkzeugen.
2. Das Format der annotierten Dateien ist nicht festgelegt. Über zusätzliche Module können weitere Formate unterstützt werden.
3. Die Art der Verweise kann abhängig vom Format der annotierten Datei gewählt werden. Beispielsweise können Verweise bei Java-Dateien auf einen AST-Knoten zeigen, während sie bei PDF-Dateien auf eine Seite des Dokuments zeigen.
4. Wie bei Elucidative Programming und XSDoc bleiben die annotierten Dateien unverändert. Bereits vorhandene Werkzeuge können ohne Anpassung weiterverwendet werden.
5. Im Gegensatz zu den vorgestellten Werkzeugen unterstützt Codation eine automatische Synchronisation von Zusatzinformationen und annotierten Dateien, die auch Änderungen mit externen Werkzeugen erlaubt.

6. Mit Codation können alle Tätigkeiten, die in den Anwendungsfällen in Abschnitt 5.2.1 auf Seite 47 f. beschrieben sind, ohne Einschränkung durchgeführt werden.

Eigenschaft	Ausprägung
Datenmodell	
Verweise	
Multiplizität	n : m
Ziele	Art der Verweisziele wird durch Module festgelegt. Bereiche, AST-Knoten und AST-Teilstrukturen sind möglich. Im Prototyp AST-Knoten.
Verweise nach außen möglich	ja
Verweise auf Gruppen möglich	ja (auf Verzeichnisse und Projekte)
Datenstruktur	
hierarchisch	nein
relational	ja
Speicherung	
getrennt	Annotierte Dateien bleiben unverändert, das Speicherformat für Zusatzinformationen wird durch den verwendeten Speicherdienst festgelegt. Im Prototyp XML.
in derselben Datei	–
Synchronisation in der Werkzeugdarstellung nach Änderungen …	
… mit Annotationswerkzeug	automatisch oder manuell, da der Benutzer den Project-Builder deaktivieren kann
… mit externem Werkzeug	automatisch oder manuell, s. o.
Rücksynchronisation aus exportierter Darstellung in die Werkzeugdarstellung nach Änderungen …	
… mit Annotationswerkzeug	automatisch oder manuell (s. o.), falls die exportierten Daten Teil des Projekts sind
… mit externem Werkzeug	automatisch oder manuell (s. o.), falls die exportierten Daten Teil des Projekts sind
Editierbarkeit und Werkzeugunterstützung	
ursprüngliche Funktionalität bleibt nutzbar	ja
editierbare, synchronisierte „Nur-Code“-Sicht vorhanden	ja
Verwendbarkeit von Übersetzern	direkt nutzbar

Tabelle 6.1: Eigenschaften von Codation

7 Fallstudie

Wie in Abschnitt 1.1 auf Seite 17 f. beschrieben, kann die Datenerhebung für die Visualisierung von Quelltext, insbesondere wenn sie manuell erfolgt, aufwändig sein. In dieser Diplomarbeit wurde deshalb die Möglichkeit geschaffen, erhobene Zusatzinformationen zur späteren Wiederverwendung speichern zu können. Die Speicherung erfordert eine effiziente Nachführung der Zusatzinformationen nach Änderungen an dem Quelltext, auf den sie sich beziehen.

In der Fallstudie wird untersucht, inwieweit Codation den Aufwand für die Visualisierung und Datenerhebung reduzieren kann. Dazu wird das Problem betrachtet, dass die während der Spezifikationsphase definierten Anwendungsfälle mit den Stellen im Quelltext verknüpft werden sollen, an denen sie umgesetzt sind. Dabei muss sowohl erkennbar sein, wo ein Anwendungsfall umgesetzt wurde (Navigationsrichtung Anwendungsfall → Quelltext) als auch, welche Anwendungsfälle ein Codeabschnitt realisiert (Navigationsrichtung Quelltext → Anwendungsfall). Nach einer Änderung am Quelltext müssen von der Änderung betroffene Anwendungsfälle bestimmt und dem Benutzer angezeigt werden. Diese Daten unterstützen den Entwickler bei der Nachführung des Testplans. Außerdem kann der Entwickler leicht erkennen, auf welche Anwendungsfälle er beim nächsten Test besonders achten muss. Auch eine Fortschrittskontrolle ist möglich. Beispielsweise ist in Abbildung 7.1 auf Seite 83 an den fehlenden Verweisen erkennbar, dass der Anwendungsfall „Erinnerung anzeigen“ noch nicht implementiert wurde.

Bei der Fallstudie wird zunächst untersucht, wie eine manuelle Verwaltung der Anwendungsfälle erfolgen kann. Daraufhin wird Codation zur Verwaltung der Anwendungsfälle eingesetzt. Abschließend werden die Vor- und Nachteile, die sich aus der Verwendung von Codation ergeben, beschrieben.

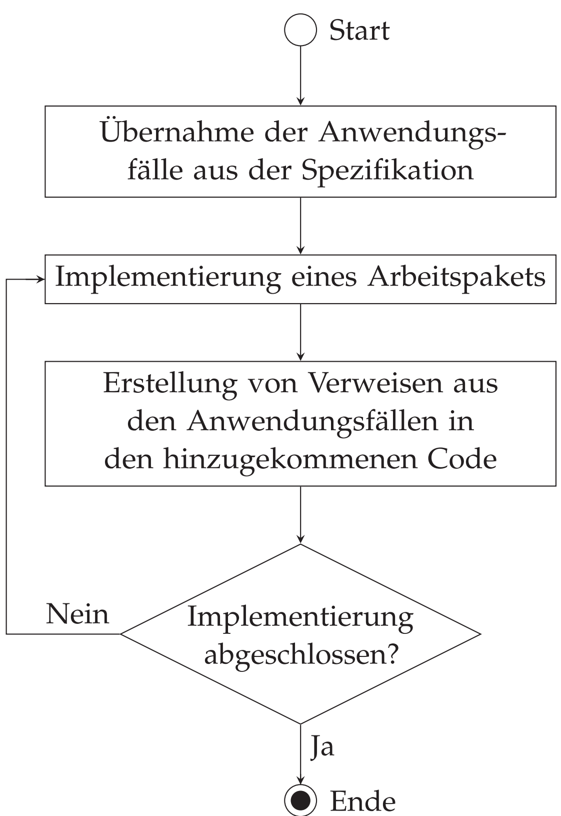
7.1 Manuelle Erhebung und Verwaltung der Daten

Bei manueller Verwaltung der Anwendungsfälle – zum Beispiel in einer Textdatei – obliegt es dem Benutzer, die Verweise zu pflegen. Er benötigt ein Verfahren, um Verweise so zu speichern, dass die Verweisziele auch nach einer Änderung am Quelltext wiedergefunden werden können. Andernfalls müssen nach jeder Änderung

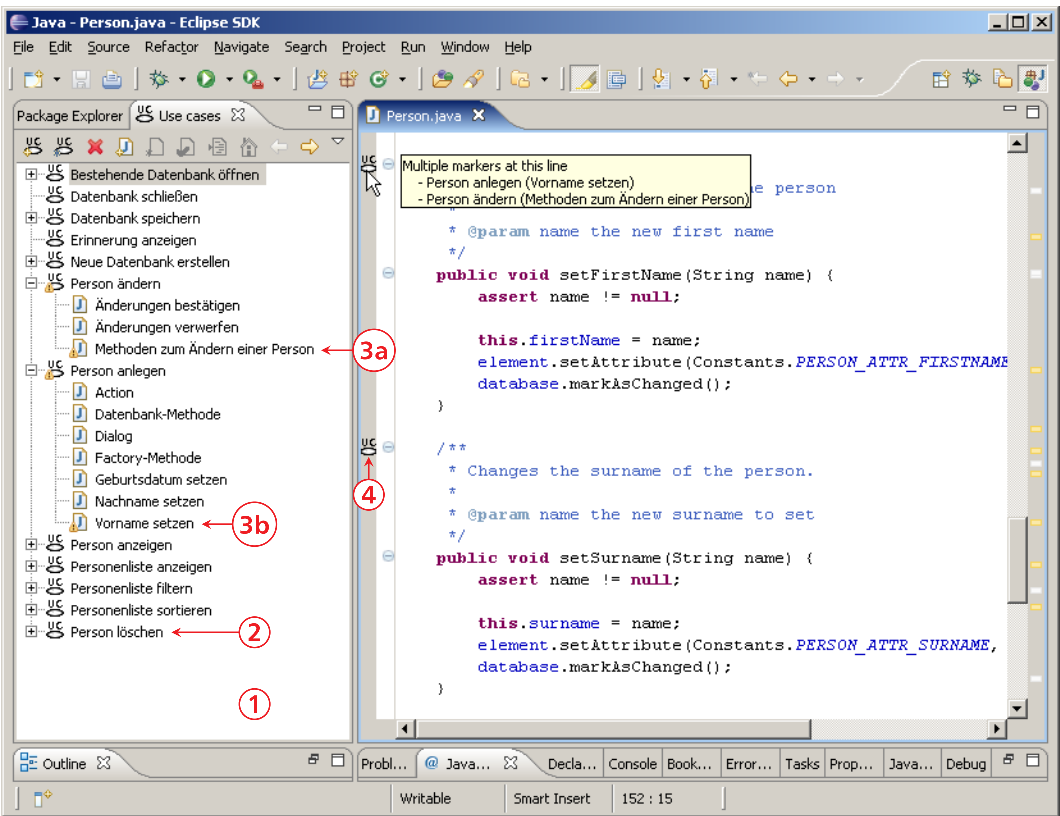
die betroffenen Verweise nachgeführt werden. Beispielsweise können Verweise auf Typen, Attribute und Methoden in der bei JavaDoc verwendeten Syntax angegeben werden. Um zu bestimmen, welche Anwendungsfälle von einer Änderung betroffen sind, muss das gewählte Format eine Suche ermöglichen.

7.2 Verwendung von Codation

Um Anwendungsfälle zu verwalten, wurde im Rahmen dieser Arbeit ein Plug-In mit dem Namen „Use-Case-Manager" entwickelt, dessen Benutzungsoberfläche in Abbildung 7.1 auf der nächsten Seite dargestellt ist. Die Implementierung kann bei Verwendung des Use-Case-Managers nach dem folgenden Prozess erfolgen:



Unter Verwendung des Use-Case-Managers wurde gemäß diesem Prozess ein Programm implementiert, das seinen Benutzer an Geburtstage erinnern soll.



Legende

1	View zur Darstellung und Verwaltung der Anwendungsfälle
2	Ein Anwendungsfall
3a, 3b	Verweise in eine Java-Datei, die nach einer Änderung geprüft werden müssen
4	Marker, der in einer Java-Datei auf einen Anwendungsfall verweist

Abbildung 7.1: Plug-In zur Verwaltung von Anwendungsfällen

7.3 Ergebnisse der Fallstudie

7.3.1 Vorteile bei der Verwendung des Use-Case-Managers

Gegenüber der manuellen Erhebung und Verwaltung der Daten ergeben sich aus der Verwendung des Use-Case-Managers folgende Vorteile:

Die Integration in Eclipse ermöglicht es, von einem Anwendungsfall direkt zu einer verknüpften Stelle im Quelltext zu springen. Durch die Marker im Editor kann der Entwickler außerdem direkt erkennen, welche Anwendungsfälle dieser Teil des Quelltextes realisiert. Codation erhöht also den Komfort bei der Navigation und verringert die dazu benötigte Zeit. Außerdem muss der Entwickler nicht zwischen verschiedenen Anwendungen wechseln.

Da auf AST-Knoten verwiesen wird, ist eine hohe Genauigkeit der Verweise möglich. Bei manueller Pflege der Daten ist die Genauigkeit hingegen aus praktischen Gründen beschränkt. Es besteht zwar die Möglichkeit, die Genauigkeit der Verweisziele bis auf Zeichengenauigkeit zu erhöhen, die Nachführung der Verweise (etwa weil eine Leerzeile entfernt wurde) wird dadurch aber sehr aufwändig. Auch der Aufwand, der nötig ist, um die von einer Änderung betroffenen Anwendungsfälle zu bestimmen, wächst mit zunehmender Genauigkeit der Verweisziele.

Nach der Änderung eines Codeabschnitts werden alle Verweise auf diesen Codeabschnitt als „zu prüfen“ markiert. So kann leicht festgestellt werden, welche Anwendungsfälle durch die Änderung betroffen sind. Bei manueller Pflege der Daten ist dies aufwändiger, da zwei Schritte nötig sind:

1. Bestimmung der geänderten Teile des Quelltextes. Nach Änderungen wie z. B. dem Umbenennen einer Klasse kann sich dies als schwierig erweisen, da das JDT automatisch alle Referenzen auf die Klasse anpasst. Es muss also – beispielsweise über die Suche – festgestellt werden, wo das JDT Änderungen vorgenommen hat.
2. Bestimmung der Anwendungsfälle, die Verweise auf geänderten Quelltext enthalten. Auch hier muss zumindest einmal eine Suche angestoßen werden.

7.3.2 Nachteile und Einschränkungen

Die Übernahme der Anwendungsfälle aus der Spezifikation ist mit zusätzlichem Aufwand verbunden, der aber über eine Import-Funktion verringert werden kann.

Bei Änderungen der Spezifikation müssen die Daten nachgezogen werden. Es besteht also die Gefahr inkonsistenter Daten. Können die Daten hingegen direkt in dem für die Spezifikation verwendeten Werkzeug verwaltet werden, entfällt diese Gefahrenquelle.

Ein weiterer Nachteil ist, dass die im Hintergrund ablaufende Synchronisation CPU-Zeit benötigt. Werden große Dateien verglichen, kann dies die Arbeit des Entwicklers behindern. Um ein Bild darüber zu bekommen, wie viel Zeit der verwendete Vergleichsalgorithmus für die Bestimmung der Änderungen zwischen zwei Java-Dateien benötigt, wurden Laufzeitmessungen durchgeführt, die in Anhang B auf Seite 105 bis 109 beschrieben werden.

Weil Codation von jeder Datei eine Kopie anlegen muss, um die Synchronisation durchführen zu können, verdoppelt sich der benötigte Speicherplatz. Eine manuelle Pflege der Daten benötigt je nach gewähltem Format weniger Speicherplatz.

7.3.3 Fazit

Als Ergebnis der Fallstudie kann festgestellt werden, dass der Aufwand, der nötig ist, um Anwendungsfälle und Verweise in den Quelltext zu verwalten, mit Codation wesentlich reduziert werden kann. Die Angabe konkreter Zahlen über die eingesparte Zeit ist jedoch nicht möglich, da viele Faktoren wie die Art der Änderungen, die Anzahl und die Struktur der geänderten Dateien sowie die Methode, in der die manuelle Datenverwaltung erfolgt, das Ergebnis bestimmen. Eine Messung würde nur für einen sehr kleinen Ausschnitt der Realität zutreffen und daher keine allgemeingültigen Schlüsse zulassen.

Einschränkungen bei der Verwendbarkeit von Codation können auftreten, wenn Codation auf Rechnern mit langsamem Hauptprozessor verwendet werden soll oder nur wenig Festplattenkapazität verfügbar ist (siehe Anhang B).

Im vorigen Kapitel wurden der Entwurf und die Realisierung von Codation beschrieben, dessen Entwicklung mit der hier geschilderten Fallstudie abgeschlossen wurde. Das nächste Kapitel blickt auf den Entwicklungsprozess zurück und fasst die Ergebnisse zusammen.

8 Fazit

Dieses Kapitel bildet den Abschluss des Berichts. Es gibt einen Rückblick auf den Verlauf der Diplomarbeit und fasst die Ergebnisse zusammen. Außerdem enthält es einen Ausblick, in dem Beispiele für die zukünftige Verwendung von Codation und offene Punkte beschrieben werden.

8.1 Rückblick

8.1.1 Projektverlauf

Zu Beginn der Arbeit wurde ein Projektplan erstellt, in dem folgende Arbeitsschritte geplant waren:

1. Projektplanung mit Meilensteinen und Risikoanalyse
2. Anforderungsanalyse und Entwicklung eines Verfahrens für den Vergleich von Informationsmodellen (Beschreibungsschema)
3. Literaturrecherche
4. Entwicklung eines Informationsmodells
5. Realisierung eines Werkzeugs, das das Informationsmodell verwendet
6. Fallstudie
7. Dokumentation der Ergebnisse der Diplomarbeit in diesem Bericht

Abgesehen von den zum Teil parallel durchzuführenden Arbeitsschritten 2. und 3. wurden die Tätigkeiten sequentiell in der angegebenen Reihenfolge durchgeführt.

Die geplanten Zeiträume für die Durchführung der Arbeitsschritte wurden weitgehend eingehalten: Die Projektplanung war drei Tage vor dem geplanten Termin fertig, die Literaturrecherche und die Entwicklung des Informationsmodells haben jeweils drei Tage länger gedauert. Die einzige große Abweichung ergab sich bei der Implementierung, die sechs statt viereinhalb Wochen gedauert hat. Die Zeitplanung war insgesamt also zu optimistisch. Durch einen eingeplanten Zeitpuffer von zwei Wochen konnten die Verzögerungen kompensiert und zusätzlich Laufzeitmessungen

durchgeführt werden, in denen der Ressourcenbedarf des entwickelten Vergleichsalgorithmus untersucht wurde. In Abbildung 8.1 auf der nächsten Seite sind die Zeitplanung und die Abweichungen als Termindrift-Diagramm dargestellt.

Zurückblickend bewerte ich es als richtig und wichtig, einen Projektplan zu erstellen, da man sich dadurch schon zu Beginn überlegen muss, welche Tätigkeiten durchzuführen sind und wie viel Zeit dafür jeweils zur Verfügung steht. Durch die Erstellung und zeitliche Planung der Arbeitspakete können jederzeit Soll- und Ist-Zustand verglichen werden.

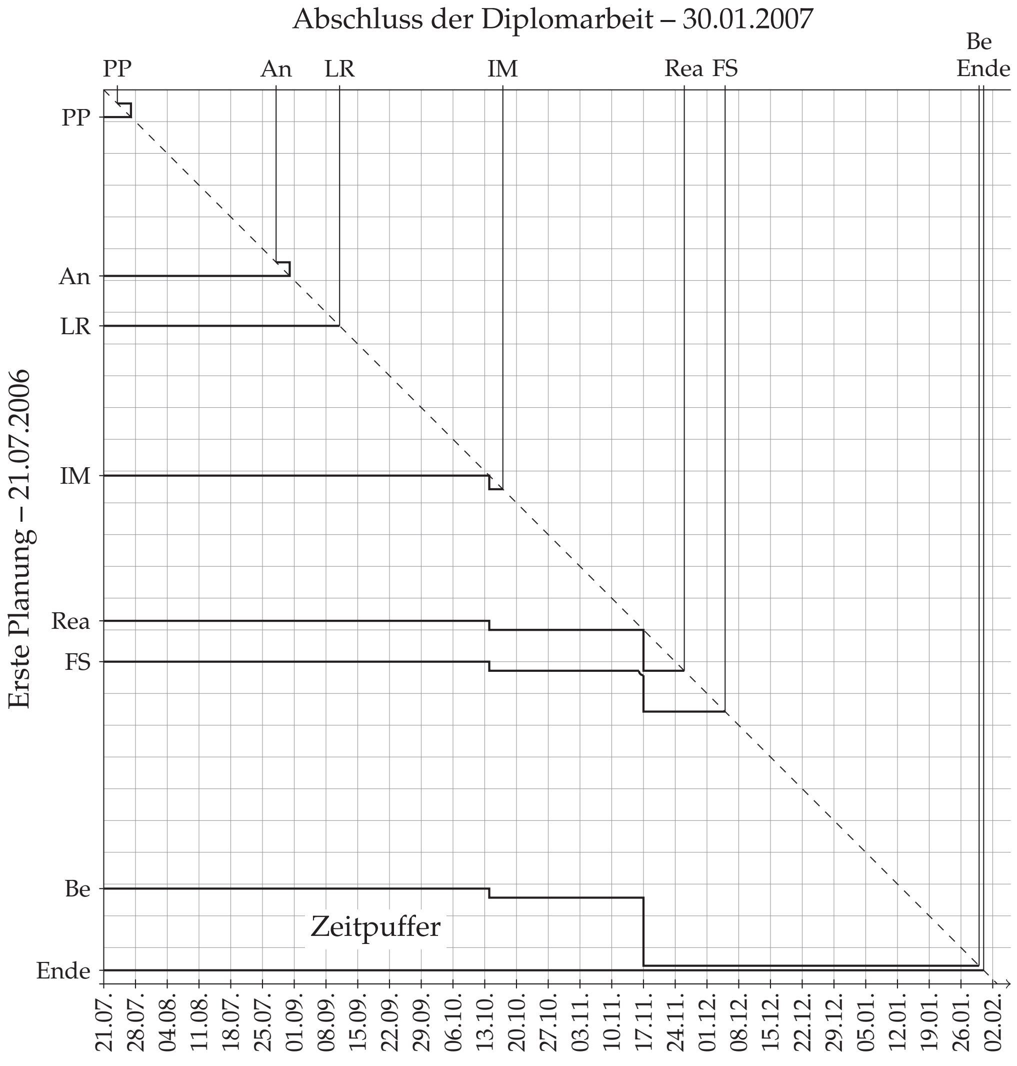
Zur Selbstkontrolle waren auch die wöchentlichen Besprechungen mit dem Betreuer meiner Diplomarbeit hilfreich. Bei den Besprechungen habe ich die Tätigkeiten und aufgetretenen Probleme der vorigen Woche beschrieben und die für die nächste Woche geplanten Tätigkeiten vorgestellt. Dadurch wurde regelmäßig der aktuelle Stand ermittelt und mit der Planung verglichen. In Kombination mit der Rückmeldung vom Betreuer ließen sich so inhaltliche und zeitliche Probleme frühzeitig erkennen.

Als sinnvoll erachte ich außerdem die explizite Einplanung eines Zeitpuffers in Kombination mit einem straffen Zeitplan. Falls die Planung eingehalten wird, kann der Puffer für zusätzliche Arbeiten oder zusätzliche Qualitätssicherung eingesetzt werden. Falls wie bei dieser Arbeit der Aufwand für ein Arbeitspaket unterschätzt wurde, kann der Zeitpuffer die Verzögerung abfangen, ohne dass dadurch an anderer Stelle gespart werden muss. Ohne eine solche Planung besteht besonders gegen Projektende kein zeitlicher Spielraum, da die auf alle Arbeitspakete verteilte zusätzliche Zeit wahrscheinlich zu langsamerem Arbeiten führt.

Hilfreich bei der Literaturrecherche war die Erstellung eines Formulars, in welchem die folgenden Punkte festgehalten wurden: Was ist der Inhalt des Papers? Wie hat der Autor die zuvor notierten Fragen beantwortet? Welche Fragen bleiben offen? Welche Verbindungen bestehen zu anderen Arbeiten?

8.1.2 Probleme während der Arbeit

Als schwierig empfand ich, bei der Literatursuche einen Schlusspunkt zu setzen. Ich habe alle mir als sinnvoll erscheinenden Stichwörter sowie die CR-Klassifikation zur Suche verwendet und alle erfolgversprechenden Literaturverweise geprüft. Dennoch hatte ich das Gefühl, nicht alles Verwertbare gefunden zu haben. Wegen der großen Menge an zu durchsuchender Literatur kann Vollständigkeit jedoch nicht erreicht werden.



Legende

An	Anforderungsanalyse, Entwicklung eines Vergleichsverfahrens
Be	Bericht anfertigen
Ende	Ende der Diplomarbeit
FS	Fallstudie
IM	Entwicklung eines Informationsmodells
LR	Literaturrecherche
PP	Projektplan
Rea	Realisierung eines Werkzeugs

Abbildung 8.1: Termindrift-Diagramm

Bei der Literaturrecherche ergab sich das Problem, dass nur bei CWEB ein Annotationswerkzeug verfügbar war. Deshalb habe ich bei den übrigen Autoren nachgefragt, ob sie ein Annotationswerkzeug zur Verfügung stellen können. Eine Antwort kam jedoch nur von Ademar Aguiar, der aber keine funktionsfähige Version seines Werkzeugs XSDoc bereitstellen konnte. Dadurch musste die Bewertung auf Basis der Dokumentation durchgeführt werden.

Eine weitere Schwierigkeit lag darin, einen Vergleichsalgorithmus für abstrakte Syntaxbäume (ASTs) zu finden. Die in der Literatur beschriebenen Algorithmen können meist nur die Ähnlichkeit zwischen zwei ASTs bestimmen, die Unterschiede ermitteln sie jedoch nicht. Der gewählte Algorithmus, der auf dem Tree Edit Distance Problem beruht, benötigt in der ursprünglichen Implementierung zu viele Ressourcen und kann Änderungen an der Reihenfolge von Methoden nicht erkennen.

Durch eine aufwändige Anpassung des auf beliebige Bäume anwendbaren Algorithmus an die Besonderheiten der Programmiersprache Java kann der Algorithmus nun erkennen, dass die Reihenfolge von Methoden verändert wurde. Außerdem konnten Laufzeit und Speicherverbrauch reduziert werden. Bei Klassen mit großem AST ist der Vergleich jedoch weiterhin zu langsam (siehe Messungen auf Seite 105).

8.2 Fazit

Informationsmodelle dienen dazu, zusätzliche Informationen zu Quelltext zusammen mit diesem zu speichern und über Verweise Verknüpfungen herzustellen. Ziel bei der Verwendung eines Informationsmodells ist, einmal erhobene Zusatzinformationen wiederverwenden zu können und so die unter Umständen aufwändige Erhebung der Zusatzinformationen nur einmal durchführen zu müssen.

In dieser Arbeit wurde ein Beschreibungsschema erstellt, in dem Eigenschaften von Informationsmodellen enthalten und in einer Hierarchie strukturiert sind. In der Literatur beschriebene Informationsmodelle wurden vorgestellt und mit Hilfe des Beschreibungsschemas gegenübergestellt, um die Unterschiede bei den verfolgten Ansätzen zu zeigen. Außerdem wurde untersucht, inwieweit die in dieser Arbeit definierten Anforderungen erfüllt werden. Die Anforderungen sind die Speicherung von Zusatzinformationen und Quelltext, die Möglichkeit, in den Quelltext zu verweisen, die Existenz einer Synchronisation zur Nachführung von Zusatzinformationen und die gute Editierbarkeit des Quelltextes.

Schwerpunkt dieser Arbeit war die Entwicklung von Codation, einem Annotationswerkzeug auf Basis von Eclipse. Codation erlaubt durch seine Erweiterbarkeit, von Zusatzinformationen in den Quelltext beliebiger Programmiersprachen zu verweisen. Der verwendete Synchronisations-Mechanismus ermöglicht es, die von einer Änderung am Quelltext betroffenen Zusatzinformationen zu bestimmen und dadurch den Aufwand zur Aktualisierung der Zusatzinformationen gering zu halten.

Die vorliegende Implementierung von Codation unterstützt unter anderem Verweise auf syntaktische Elemente von Java-Code. Bei der Synchronisation wird zur Berechnung der Änderungsinformationen ein Algorithmus verwendet, der Java-Dateien auf syntaktischer Ebene vergleicht.

8.3 Ausblick

Neben dem in der Fallstudie beschriebenen Beispiel der manuell gepflegten Liste mit Anwendungsfällen können in Codation nahezu beliebige Arten von Zusatzinformationen gespeichert werden, sowohl manuell erhobene, wie Dokumentation, als auch automatisch erhobene, wie Metriken. Das Informationsmodell von Codation ist außerdem nicht an die Programmiersprache Java gebunden. Durch Erstellung weiterer Plug-Ins können Verweise in beliebige Dateitypen ermöglicht werden. Ein solches Plug-In muss lediglich eine Methode anbieten, um Änderungsinformationen zu berechnen. Sowohl die Verwaltung der dazu notwendigen Kopien aller annotierten Dateien als auch die Speicherung der Verweise erfolgt durch Codation.

Durch die beiden Möglichkeiten zur Erweiterung ergeben sich zahlreiche weitere Einsatzgebiete für Codation. Zwei Beispiele für Erweiterungen von Codation werden nachfolgend beschrieben.

An der Universität Stuttgart wird derzeit in der Abteilung Software Engineering des Instituts für Softwaretechnologie ein Studienprojekt durchgeführt. Darin soll ein Testwerkzeug entwickelt werden, das u. a. die Möglichkeit bietet, während eines Glass-Box-Tests Überdeckungsmaße und Metriken zu erheben. Das Testwerkzeug soll möglichst unabhängig von der zu testenden Programmiersprache sein. Codation ist gut für die Verwendung in diesem Testwerkzeug geeignet, da es die Möglichkeit bietet, mit Zusatzinformationen den Teil des Quelltextes zu markieren, der von einem Testfall überdeckt wird. Nach Änderungen am Quelltext kann dem Benutzer eine Liste mit den Testfällen angezeigt werden, die von der Änderung betroffen sind

und neu ausgeführt werden sollten (obwohl natürlich die erneute Durchführung aller Tests die bessere Alternative wäre).

Ein namhaftes deutsches Unternehmen entwickelt für einen großen schwäbischen Automobilhersteller eine Software in einem Cobol-Dialekt. Bei diesem Cobol-Dialekt existiert neben der Datei mit dem Quelltext eine weitere Datei, in welcher der Ablauf jedes Abschnitts – dem Gegenstück zu einer Methode – in natürlicher Sprache beschrieben ist. Bei der Implementierung muss der Entwickler bislang häufig zwischen den Dateien wechseln. Wenn ein Plug-In für den Cobol-Dialekt entwickelt werden würde, könnte man die Beschreibung mit der Implementierung verknüpfen und z. B. in einem View die Beschreibung der Section anzeigen, in der sich der Cursor zur Zeit befindet.

Als offener Punkt bleibt das Problem, dass der Vergleichsalgorithmus für Java-Dateien bei großen Dateien trotz der vorgenommenen Verbesserungen zu langsam ist. Es sollte geprüft werden, inwieweit eine Verringerung der Genauigkeit die Geschwindigkeit erhöhen kann, ohne die Ergebnisse wesentlich zu verschlechtern. Beispielsweise könnte man qualifizierte Namen (z. B. „org.eclipse.jdt“) als Ganzes vergleichen, statt die Kaskade von einfachen Namen („org“, „eclipse“ und „jdt“) zu vergleichen, aus denen sie sich zusammensetzen. Ein qualifizierter Name aus n einfachen Namen entspricht $2n + 1$ AST-Knoten, die durch die Vereinfachung auf einen Knoten reduziert werden könnten. Allerdings ist zu beachten, dass dadurch die von einer Änderung betroffenen Verweise eventuell schwerer zu bestimmen sind oder die Genauigkeit von Verweisen entsprechend reduziert werden muss.

A Ergebnisse von Metrikerhebungen

Nach Abschluss der Implementierung wurden Metriken für den Quelltext von Codation erhoben. Dieses Kapitel beschreibt die erhobenen Metriken und die gemessenen Werte. Detaillierte Daten, bei denen die Messwerte für jede Klasse getrennt aufgeführt sind, sind auf der beiliegenden CD enthalten.

Das Ziel der Metrikerhebungen ist, mit den Ergebnissen den Umfang des Quelltextes zu bestimmen und die Qualität von Quelltext und Entwurf beurteilen zu können. Aussagen über die Qualität sind jedoch nur in beschränktem Ausmaß möglich, weil die Bildung des Mittelwerts Schwankungen und Extremwerte unterdrückt. Daher werden zusätzlich sowohl die Standardabweichung, mit der die Streuung der Einzelwerte abgeschätzt werden kann, als auch das Maximum angegeben, wodurch Ausreißer erkannt werden können.

Zur Erhebung wurden folgende Programme verwendet:

- SourceMonitor in Version 2.2.0.1: Ermittlung des Kommentaranteils
- Metrics in Version 1.3.6: Erhebung der übrigen automatisch erhobenen Metriken

A.1 Erhobene Metriken

Dieser Abschnitt beschreibt die erhobenen Metriken, die entsprechend der zuvor beschriebenen Zielstellung in die drei Gruppen Programm-Umfang, Codequalität und Entwurfsqualität eingeteilt wurden. Die Beschreibung der Metriken basiert teilweise auf Chidamber u. Kemerer (1994).

A.1.1 Metriken zur Bestimmung des Programm-Umfangs

Lines of Code: Gesamtanzahl aller Zeilen. Nicht mitgezählt werden leere Zeilen und Kommentare.

Number of Packages: Anzahl der Pakete im Plug-In.

Number of Classes: Anzahl der Klassen in einem Paket.

Number of Interfaces: Anzahl der Schnittstellen in einem Paket.

Number of Attributes: Anzahl nicht-statischer Attribute in einer Klasse.

Number of Static Attributes: Anzahl statischer Attribute in einer Klasse. Bei Codation sind dies Konstanten und Attribute von Singletons.

A.1.2 Metriken zur Beurteilung der Codequalität

Number of Methods (NOM): Anzahl nicht-statischer Methoden in einer Klasse.

Ein hoher Wert ist ein Anzeichen dafür, dass die Klasse eventuell aufgeteilt werden sollte, da anzunehmen ist, dass große Klassen schlechter zu entwickeln und warten sind und schlechter wiederverwendet werden können.

Number of Static Methods: Anzahl statischer Methoden in einer Klasse.

Für statische Methoden gilt das zuvor für nicht-statische Methoden Gesagte. Da statische Methoden den Zustand eines Objektes nicht verändern können (von globalen Variablen abgesehen), haben statische Methoden jedoch keine Seiteneffekte.

Number of Parameters: Parameterzahl einer Methode.

Je mehr Parameter eine Methode besitzt, desto unübersichtlicher wird ihr Aufruf, was die Gefahr von Fehlern erhöht. Bei Konstruktoren oder Factory-Methoden kann eine höhere Parameterzahl dennoch vorteilhaft sein, wenn bei der Klasse dadurch auf den Einsatz von set-Methoden verzichtet werden kann (Bloch, 2001, Item 13: Favor immutability).

Zyklomatische Komplexität nach McCabe: Berechnung der Komplexität einer Methode auf Basis des Kontrollflusses (McCabe, 1976).

Die Komplexität einer Methode beeinflusst den Entwicklungs- und Wartungsaufwand, da komplexe Methoden schwerer verständlich und deshalb anfälliger für Fehler sind.

Weighted Methods per Class (WMC): Summe der zyklomatischen Komplexität aller Methoden einer Klasse.

Je höher dieser Wert ist, desto komplexer und somit schwerer verständlich ist eine Klasse.

Schachtelungstiefe: Maximale Schachtelungstiefe in einer Methode.

Je tiefer der Code einer Methode geschachtelt ist, desto schwieriger ist sie zu verstehen. Eventuell sollte eine solche Methode mit hoher Schachtelungstiefe aufgeteilt werden.

Anteil Kommentarzeilen: Anzahl der Zeilen, die Kommentare enthalten, im Verhältnis zur gesamten Zeilenanzahl.

Kommentare innerhalb von Methoden erhöhen die Wartbarkeit, da sie Programmabläufe leichter verständlich machen. JavaDoc-Kommentare verbessern die Erweiterbarkeit und erleichtern die Wiederverwendung, da sie die Aufgabe von Methoden und Typen beschreiben.

A.1.3 Metriken zur Beurteilung der Entwurfsqualität

Number of Children: Anzahl *direkter* Kindklassen einer Klasse. Eine Klasse, die eine Schnittstelle implementiert, wird als Kindklasse dieser Schnittstelle gezählt.

Ein hoher Wert bedeutet, dass viele Klassen Methoden und Attribute dieser Klasse wiederverwenden. Zwischen den Kindklassen findet aber keine Wiederverwendung statt. Eine tiefere Vererbungshierarchie könnte eventuell die Wiederverwendung von Code zwischen den Kindklassen erhöhen. Eine Klasse mit vielen Kindklassen hat einen hohen Einfluss, was die Wartbarkeit verringert.

Depth of Inheritance Tree (DIT): Länge des Pfades im Vererbungsbaum von einer Klasse zur Klasse `Object`.

Je größer der Wert ist, desto höher ist die potentielle Wiederverwendung geerbter Methoden und Attribute. Allerdings steigt die Komplexität, da das Verhalten bei Methodenaufrufen schlechter vorhersehbar wird, weil mehr Klassen und Methoden beteiligt sind.

Number of Overridden Methods (NORM): Anzahl der überschriebenen Methoden. Nicht mitgezählt werden:

- überschriebene abstrakte Methoden,
- Methoden, die `super` aufrufen und
- die zum Überschreiben gedachten Methoden toString, hashCode und equals.

Eine hohe Anzahl von überschriebenen Methoden erhöht die Komplexität, da schlechter vorhersehbar ist, welche Methode aufgerufen wird.

Specialization Index: Definiert als $\frac{\text{NORM} \cdot \text{DIT}}{\text{NOM}}$.

Diese Metrik ist ein Maß für die Komplexität des Entwurfs. Ein hoher Wert bedeutet, dass eine tiefe Vererbungshierarchie besteht und viele Methoden überschrieben werden. Dies kann die Wartbarkeit verringern, da Fehler schlechter zu finden sind.

Größe der API: Die oben aufgeführten Metriken erlauben keine Aussage über die Kopplung zwischen den Plug-Ins. Daher wird zusätzlich die Größe der API bestimmt. Zur API zählen alle Typen und Enumerations, die nicht als intern gekennzeichnet sind und somit von anderen Plug-Ins verwendet werden können. Es wird unterschieden zwischen:

- Enumerations und Konstanten-Klassen: Konstanten-Klassen, die ausschließlich finale, statische Attribute besitzen und nicht instanzierbar sind. Enumerations sind diesen Klassen ähnlich, da ihre Elemente wie Konstanten verwendet werden.
- Schnittstellen: Schnittstellen sind öffentlich sichtbaren Klassen vorzuziehen, da sie von anderen Plug-Ins nicht instanziert werden können. Außerdem kann die implementierende Klasse geändert werden, ohne dass die Schnittstelle geändert werden muss, die API ist also stabiler. Umgekehrt haben Entwickler anderer Plug-Ins, die eine Schnittstelle implementieren müssen, größere Freiheiten in ihrem Entwurf, da Schnittstellen die Vererbungshierarchie nicht einschränken.
- Klassen: Alle Klassen, die nicht in eine der oberen Kategorien fallen.

A.2 Gemessene Werte

A.2.1 Core-Plugin

Dieses Plug-In bildet den Anwendungskern von Codation.

Metrik	Summe	Mittelwert	Standard-Abweichung	Maximum
Lines of Code	2490			
Number of Packages	10			
Number of Classes	40	1,481	2,455	8
Number of Interfaces	16	1,6	2,538	7
Number of Attributes	58	2,148	2,189	9
Number of Static Attributes	91	3,37	9,206	36
Number of Methods	122	4,519	3,785	15
Number of Static Methods	40	1,481	2,455	8
Number of Parameters		1,167	1,297	5
McCabe		1,951	1,895	12
Weighted Methods per Class	316	11,704	11,556	42
Schachtelungstiefe		1,519	0,78	5
Anteil Kommentarzeilen	44 %			
Number of Children	5	0,185	0,669	3
Depth of Inheritance Tree		1,519	0,787	4
Number of Overridden Methods	5	0,185	0,611	3
Specialization Index		0,092	0,38	2
Größe der API	26			
davon Enumerations und Konstanten	4			
davon Schnittstellen	16			
davon Klassen	6			

A.2.2 Java-Link-Provider

Mit Hilfe dieses Plug-Ins kann in Zusatzinformationen auf AST-Knoten von Java-Dateien verwiesen werden. Außerdem enthält es den Algorithmus zum Vergleich von Java-Dateien.

Metrik	Summe	Mittelwert	Standard-Abweichung	Maximum
Lines of Code	3436			
Number of Packages	6			
Number of Classes	28	4,667	6,872	20
Number of Interfaces	5	0,833	1,863	5
Number of Attributes	52	1,857	2,31	7
Number of Static Attributes	40	1,429	3,396	18
Number of Methods	158	5,643	6,799	28
Number of Static Methods	139	4,964	17,047	89
Number of Parameters		1,205	1,006	5
McCabe		1,882	5,324	85
Weighted Methods per Class	559	19,964	36,205	186
Schachtelungstiefe		1,263	0,655	6
Anteil Kommentarzeilen	40,9 %			
Number of Children	3	0,107	0,557	3
Depth of Inheritance Tree		1,536	0,823	4
Number of Overridden Methods	23	0,821	1,197	4
Specialization Index		0,587	1,034	3
Größe der API	11			
davon Enumerations und Konstanten	3			
davon Schnittstellen	5			
davon Klassen	3			

A.2.3 Simple-Link-Provider

Dieses Plug-In definiert einen File-Link-Provider, mit dem Verweise auf beliebige Dateien erzeugt werden können, ohne jedoch auf eine Position innerhalb der Dateien zu verweisen.

Metrik	Summe	Mittelwert	Standard-Abweichung	Maximum
Lines of Code	115			
Number of Packages	2			
Number of Classes	4	2	0	2
Number of Interfaces	1	0,5	0,5	1
Number of Attributes	1	0,25	0,433	1
Number of Static Attributes	3	0,75	0,829	2
Number of Methods	1	0,25	0,433	1
Number of Static Methods	1	0,25	0,433	1
Number of Parameters		0,833	1,213	5
McCabe		1,389	1,38	7
Weighted Methods per Class	25	6,25	4,815	14
Schachtelungstiefe		1,111	0,314	2
Anteil Kommentarzeilen	47,5 %			
Number of Children	0	0	0	0
Depth of Inheritance Tree		1,25	0,433	2
Number of Overridden Methods	0	0	0	0
Specialization Index		0	0	0
Größe der API	3			
davon Enumerations und Konstanten	1			
davon Schnittstellen	1			
davon Klassen	1			

A.2.4 XML-Storage-Service

Mit diesem Plug-In werden die Zusatzinformationen in XML-Dateien gespeichert.

Metrik	Summe	Mittelwert	Standard-Abweichung	Maximum
Lines of Code	910			
Number of Packages	3			
Number of Classes	9	3	1,633	5
Number of Interfaces	0	0	0	0
Number of Attributes	24	2,667	3,3	9
Number of Static Attributes	34	3,778	9,647	31
Number of Methods	68	7,556	9,476	33
Number of Static Methods	9	1	2,211	7
Number of Parameters		0,844	0,774	3
McCabe		1,571	0,918	6
Weighted Methods per Class	121	13,444	13,557	47
Schachtelungstiefe		1,403	0,541	3
Anteil Kommentarzeilen	35,5 %			
Number of Children	1	0,111	0,314	1
Depth of Inheritance Tree		1,222	0,416	2
Number of Overridden Methods	1	0,111	0,314	1
Specialization Index		0,003	0,01	0,03
Größe der API	0			

A.2.5 UI-Plugin

Dieses Plug-In enthält die Benutzungsoberfläche von Codation.

Metrik	Summe	Mittelwert	Standard-Abweichung	Maximum
Lines of Code	156			
Number of Packages	3			
Number of Classes	3	1	0	1
Number of Interfaces	0	0	0	0
Number of Attributes	2	0,667	0,943	2
Number of Static Attributes	9	1	2,211	7
Number of Methods	9	3	1,633	5
Number of Static Methods	9	3	4,243	9
Number of Parameters		0,733	0,65	2
McCabe		1,444	1,165	6
Weighted Methods per Class	26	8,667	5,437	13
Schachtelungstiefe		1,167	0,373	2
Anteil Kommentarzeilen	37,7 %			
Number of Children	0	0	0	0
Depth of Inheritance Tree		3	0,816	4
Number of Overridden Methods	2	0,667	0,943	2
Specialization Index		0,533	0,754	1,6
Größe der API	0			

A.2.6 Use-Case-Manager

Das im Rahmen der Fallstudie entstandene Plug-In zur Verwaltung von Anwendungsfällen.

Metrik	Summe	Mittelwert	Standard-Abweichung	Maximum
Lines of Code	2159			
Number of Packages	6			
Number of Classes	26	4,333	2,427	9
Number of Interfaces	2	0,333	0,471	1
Number of Attributes	44	1,692	2,839	13
Number of Static Attributes	96	3,692	9,941	42
Number of Methods	143	5,5	4,971	21
Number of Static Methods	19	0,731	1,851	9
Number of Parameters		0,778	1,149	6
McCabe		1,679	1,408	9
Weighted Methods per Class	272	10,462	7,525	32
Schachtelungstiefe		1,414	0,829	5
Anteil Kommentarzeilen	31,7 %			
Number of Children	15	0,577	1,864	7
Depth of Inheritance Tree		3,115	1,761	5
Number of Overridden Methods	11	0,423	0,567	2
Specialization Index		0,517	0,831	3
Größe der API	0			

A.2.7 JUnit-Tests

Um die zuvor aufgeführten Plug-Ins zu testen, wurden JUnit-Tests erstellt.

Metrik	Summe	Mittelwert	Standard-Abweichung	Maximum
Anzahl Testfälle[1)]	162			
Anzahl asserts[2)]	602			
Lines of Code	2996			
Number of Packages	6			
Number of Classes	45	7,5	7,274	18
Number of Interfaces	0	0	0	0
Number of Attributes	73	1,622	2,163	8
Number of Static Attributes	56	1,244	4,682	30
Number of Methods	387	8,6	10,072	60
Number of Static Methods	5	0,111	0,314	1
Number of Parameters		0,638	1,026	4
McCabe		1,107	0,427	3
Weighted Methods per Class	434	9,644	11,768	60
Schachtelungstiefe		1,064	0,244	2
Anteil Kommentarzeilen	30,2 %			
Number of Children	22	0,489	2,4	16
Depth of Inheritance Tree		1,556	0,617	3
Number of Overridden Methods	14	0,311	0,463	1
Specialization Index		0,146	0,264	1
Größe der API	0			

1) Anzahl der Methoden, die durch die Annotation `@Test` als JUnit-Test markiert sind.

2) Anzahl der Aufrufe der Methode assert, mit der das Ergebnis einer Methode oder der Wert einer Variable geprüft wird.

A.3 Fazit

A.3.1 Entwurfsqualität

Durch die Metriken konnten keine Mängel im Entwurf festgestellt werden.

Abgesehen von den sieben Actions für die Werkzeugleiste im Use-Case-Manager, die alle dieselbe Oberklasse besitzen, und JUnit-Testfällen hat eine Klasse in Codation maximal drei Kindklassen. Auch die Tiefe des Vererbungsbaums ist gering. Berücksichtigt man nur die zu Codation gehörenden Klassen, ist die Tiefe maximal zwei. Berücksichtigt man auch die Klassen von Eclipse, ist die Tiefe maximal fünf. Da zudem die Zahl überschriebener Methoden gering ist, kann die Komplexität des Entwurfs ebenfalls als gering angesehen werden.

Die API der Plugins wurde so klein wie möglich gehalten. Zudem besteht sie größtenteils aus Konstanten und Schnittstellen. Dies erleichtert wie zuvor beschrieben die Wartbarkeit, da andere Plug-Ins durch Änderungen weniger stark betroffen sind.

A.3.2 Codequalität

Die Ergebnisse der Metriken, die zur Beurteilung der Codequalität verwendet wurden, zeigen in ihren Mittelwerten keine Auffälligkeiten. Es gibt jedoch einige Klassen, die neben einer Methode mit hoher zyklomatischer Komplexität zahlreiche weitere Methoden enthalten. Bei den komplexen Methoden handelt es sich um Methoden, die in einer Case-Anweisung Aufgaben delegieren.

Beispielsweise existiert eine Klasse mit einer Methode, mit der die Kindknoten eines AST-Knotens bestimmt werden können. Da an dieser Stelle die Überladung von Methoden nicht verwendet werden kann, enthält die Klasse eine Methode, welche die Bestimmung der Kindknoten abhängig vom Typ des übergebenen AST-Knotens an die passende Methode weiterleitet. Dadurch besitzt die Methode, die nur aus einer Case-Anweisung besteht, eine zyklomatische Komplexität von 85. Durch die Delegation ergibt sich auch die hohe Methodenanzahl von 89.

B Laufzeitmessungen

Der Algorithmus, mit dem die abstrakten Syntaxbäume zweier Java-Dateien verglichen werden, verursacht einen Zeitaufwand in Höhe von $O(n^2)$ (siehe Abschnitt 6.3.4 auf Seite 71 bis 77). Um zu prüfen, wie lange der Vergleich in der Praxis dauert, wurden Laufzeitmessungen durchgeführt, deren Ergebnisse in diesem Kapitel beschrieben werden. Die zu klärenden Fragen sind:

1. Welchen Einfluss hat die an einer Klasse vorgenommene Änderung auf die Laufzeit?
2. Wie wirkt sich die Größe der Methoden auf die Laufzeit aus?
3. Welchen Einfluss hat die Anzahl der in einer Klasse enthaltenen Methoden auf die Laufzeit?

B.1 Messumgebung

Die Messergebnisse beziehen sich auf die im folgenden beschriebene Hard- und Softwareumgebung.

Hardware:
- Intel Core Duo @T2300 ($2 \times 1{,}66$ GHz)
- 2 GB Arbeitsspeicher (DDR2, 533 MHz)

Software:
- Windows XP Professional SP2
- Eclipse SDK 3.2.1-200609210945;
Codation ist das einzige zusätzlich installierte Feature;
Aufrufparameter: „-vmargs -Xms512M -Xmx1024M“
- Sun JDK 1.5 Update 6

B.2 Testdaten

Für die Messungen wurden die in Tabelle B.1 auf der nächsten Seite aufgeführten Klassen des JDT (Version 3.2.0-671) ausgewählt. Um den Einfluss der Methoden-

größe zu prüfen, wurden drei Klassen gewählt, die ungefähr die gleiche Anzahl an Methoden und Attributen besitzen, aber unterschiedlich groß sind. Um den Einfluss der Methodenanzahl zu prüfen, wurden drei weitere Klassen mit unterschiedlich vielen Methoden gewählt, die durchschnittlich rund 50 AST-Knoten pro Methode besitzen.

	Dateigröße[1)] [Byte]	AST-Knoten	Methoden und Attribute
SourceTypeElementInfo[2)]	9 399	1 379	32
FieldReference[3)]	20 842	2 522	33
SearchPattern[4)]	74 458	6 439	34
CreateMethodOperation[2)]	4 543	623	9
Openable[2)]	15 485	1 653	33
ASTRewriteFlattener[5)]	41 943	5 732	99

1) zum Vergleich: die durchschnittliche Dateigröße bei Codation beträgt 4618 Byte
2) Paket `org.eclipse.jdt.internal.core`
3) Paket `org.eclipse.jdt.internal.compiler.ast`
4) Paket `org.eclipse.jdt.core.search`
5) Paket `org.eclipse.jdt.internal.core.dom.rewrite`

Tabelle B.1: Für die Laufzeitmessungen ausgewählte Dateien

Alle Klassen wurden in der Praxis relevanten Änderungen unterworfen, beispielsweise dem Umbenennen einer Methode. Die folgenden Änderungen wurden vorgenommen (jeweils vom Originalzustand ausgehend):

a) Hinzufügen einer Methode am Ende der Klasse

b) Verschieben der ersten Methode an das Dateiende

c) Umbenennen der letzten Methode und Anpassung aller Aufrufe dieser Methode

d) Hinzufügen einer Anweisung zur ersten Methode

e) Hinzufügen eines Attributs am Anfang der Klasse

f) Umbenennen des ersten Attributs und Anpassung aller Stellen, an denen dieses Attribut verwendet wird

B.3 Messergebnisse

Auf der nächsten Seite sind die Ergebnisse der Messungen dargestellt. Für jede Kombination wurden 50 Messungen durchgeführt. In den Tabellen ist jeweils der Median der Messwerte in Millisekunden angeben.

Klasse SourceTypeElementInfo:

	a	b	c	d	e	f
Original	281	266	266	281	281	281
a	—	266	281	281	281	266
b		—	266	266	266	266
c			—	266	266	266
d				—	281	281
e					—	266

Klasse FieldReference:

	a	b	c	d	e	f
Original	2 484	2 453	2 485	2 438	2 469	2 453
a	—	2 406	2 484	2 547	2 500	2 453
b		—	2 437	2 484	2 406	2 438
c			—	2 437	2 454	2 500
d				—	2 484	2 454
e					—	2 453

Klasse SearchPattern:

	a	b	c	d	e	f
Original	45 625	45 781	45 594	45 609	44 484	46 328
a	—	46 312	46 296	46 109	45 860	45 469
b		—	44 985	45 656	45 265	45 547
c			—	45 593	45 610	45 766
d				—	45 719	45 796
e					—	45 719

Klasse CreateMethodOperation:

	a	b	c	d	e	f
Original	109	109	109	109	109	109
a	—	109	94	110	109	110
b		—	93	109	109	109
c			—	109	109	110
d				—	109	110
e					—	110

Klasse Openable:

	a	b	c	d	e	f
Original	547	547	532	532	532	547
a	—	547	547	547	531	547
b		—	532	531	547	531
c			—	547	547	547
d				—	547	547
e					—	547

Klasse ASTRewriteFlattener:

	a	b	c	d	e	f
Original	1 750	1 750	1 750	1 750	1 735	1 766
a	—	1 734	1 750	1 750	1 735	1 781
b		—	1 734	1 750	1 735	1 750
c			—	1 750	1 750	1 750
d				—	1 735	1 750
e					—	1 765

B.4 Fazit

Einfluss der Änderungen

Aus der Homogenität der Werte innerhalb einer Tabelle kann geschlossen werden, dass die Art der durchgeführten Änderung keinen messbaren Einfluss auf die Berechnungsdauer besitzt.

Einfluss der Größe von Methoden

Erkennbar ist der große Einfluss, den die Größe der Methoden besitzt (Zeitaufwand etwa $O(3n^2)$). Die Klasse `FieldReference` ist etwa doppelt so groß wie die Klasse `SourceTypeElementInfo`, die Laufzeit steigt aber auf mehr als das Neunfache. Zwischen den Klassen `SearchPattern` und `FieldReference` steigt die Zahl der AST-Knoten um den Faktor 2,5, die Laufzeit jedoch um den Faktor 18.

Einfluss der Methodenanzahl

Durch die in Abschnitt 6.3.4 beschriebenen Änderungen am Vergleichsalgorithmus wirkt sich die Methodenanzahl nicht wie in der Originalversion quadratisch, sondern annähernd linear auf die Laufzeit aus. So steigt die Methodenanzahl zwischen den Klassen `CreateMethodOperation` und `Openable` um den Faktor 3,6 während sich die Laufzeit verfünffacht. Bei der Klasse `ASTRewriteFlattener`, die die dreifache Anzahl an Methoden enthält wie die Klasse `Openable`, steigt die Laufzeit um den Faktor drei.

C Entwurf

Dieses Kapitel beschreibt den Entwurf von Codation und dient somit als Grundlage für Erweiterungen von Codation. Der Entwurf verfolgt einen erweiterbaren, komponentenbasierten Ansatz, d. h. es existieren verschiedene Komponenten mit klar abgegrenztem Aufgabenbereich. Abschnitt C.1 erläutert die einzelnen Komponenten.

Abschnitt C.2 beschreibt den Aufbau einer Zusatzinformation, Abschnitt C.3, wie Zusatzinformationen abgerufen, erzeugt und verändert werden können. Abschnitt C.4 beschreibt, wie von Zusatzinformationen auf Ressourcen (siehe Begriffserklärung 6.1) verwiesen wird. Die nach Änderungen an annotierten Dateien durchgeführte Synchronisation wird in Abschnitt C.5 erläutert. In den Abschnitten C.6 und C.7 sind die Plug-Ins zur Erstellung von Verweisen und zur Speicherung beschrieben. Abschließend enthält Abschnitt C.8 die Dokumentation der Extension-Points.

C.1 Plugin-Architektur von Codation

In Abbildung C.1 auf der nächsten Seite ist die Architektur von Codation dargestellt. Jede Komponente wird über ein Eclipse-Plugin realisiert. Mittelpunkt der Architektur ist der Anwendungskern *Core*, der als Vermittler zwischen den anderen Plug-Ins fungiert. Der Anwendungskern enthält außerdem den zur Synchronisation verwendeten Project-Builder.

Am Anwendungskern können sich *Annotation-Provider* registrieren. Vom Anwendungskern werden sie über Änderungen an Ressourcen benachrichtigt und können dementsprechend reagieren. Zeitgleich können mehrere Annotation-Provider registriert sein. Der entwickelte Prototyp enthält das Plug-In zur Verwaltung von Anwendungsfällen (siehe Fallstudie, Kapitel 7 auf Seite 81 bis 85).

Definition C.1 (Annotation-Provider) *Ein Annotation-Provider erzeugt Zusatzinformationen. Er ist auch für deren Darstellung verantwortlich und wird vom Anwendungskern über Änderungen an Dateien benachrichtigt (Synchronisation). Ein Annotation-Provider erweitert den Extension-Point de.jwertenauer.codation.core.annotationProvider.*

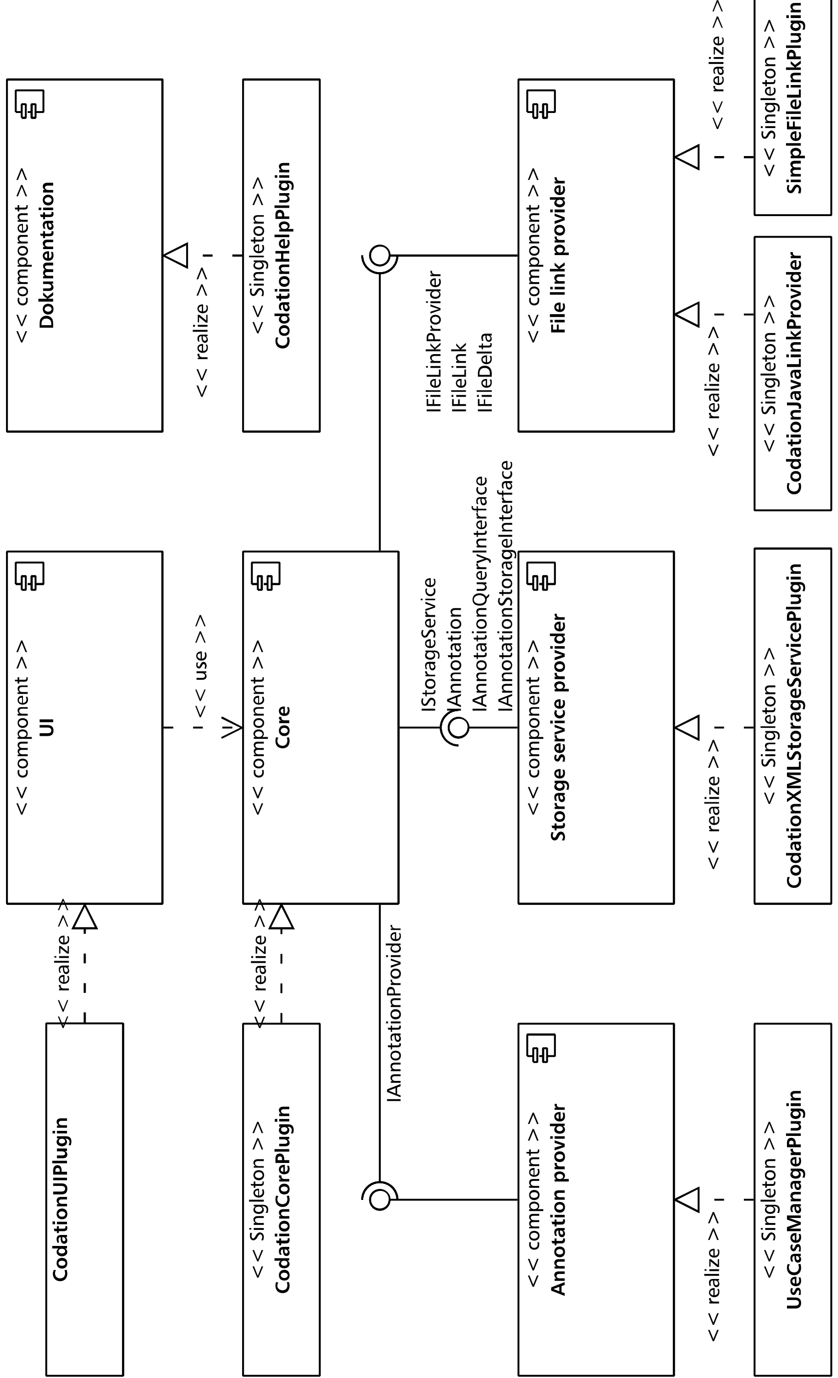


Abbildung C.1: Architektur von Codation

Zur Speicherung der Zusatzinformationen verwendet der Anwendungskern den *Storage-Service* eines *Storage-Service-Providers*. Dieser übernimmt die Speicherung und ermöglicht die Suche nach Zusatzinformationen. Alle Anfragen von Annotation-Providern werden vom Anwendungskern an den aktiven Storage-Service weitergeleitet. Der Prototyp enthält ein Plug-In, das Zusatzinformationen in XML-Dateien speichert (siehe Abschnitt C.7 auf Seite 128 bis 132).

Definition C.2 (Storage-Service-Provider, Storage-Service) *Ein Storage-Service-Provider stellt einen Storage-Service zur Verfügung und erweitert den Extension-Point de .jwertenauer.codation.core.storageServiceProvider. Ein Storage-Service ist für die Speicherung der Zusatzinformationen und Verweise zuständig.*

Verweise auf bestimmte Punkte innerhalb von Dateien ermöglichen die *File-Link-Provider* (siehe Abschnitt C.4 auf Seite 117 bis 119). Sie werden vom Anwendungskern über Änderungen an den von ihnen unterstützten Dateien benachrichtigt und können daraufhin detaillierte Änderungsinformationen berechnen, die der Anwendungskern an die Annotation-Provider weiterleitet. Der Prototyp enthält zwei File-Link-Provider, die in Abschnitt C.6 auf Seite 124 bis 127 beschrieben werden.

Definition C.3 (File-Link-Provider) *File-Link-Provider ermöglichen Verweise auf Bereiche innerhalb von Dateien. Sie erweitern den Extension-Point de.jwertenauer.codation.core .fileLinkProvider.*

Die Komponente *UI* enthält die Benutzeroberfläche, über die Codation für ein Projekt aktiviert werden kann. In der Komponente *Dokumentation* ist die Dokumentation enthalten, die in der Hilfe von Eclipse angezeigt wird.

C.2 Aufbau einer Zusatzinformation

In Abbildung C.2 auf der nächsten Seite ist die Schnittstelle `IAnnotation` dargestellt, die eine Zusatzinformation repräsentiert. Die Schnittstelle muss von einem Storage-Service-Provider implementiert werden.

Eine Zusatzinformation kann Daten in Form von Eigenschaften mit den Datentypen `boolean`, `double`, `float`, `int`, `long` und `String` speichern. Eine Eigenschaft besitzt einen innerhalb der Zusatzinformation eindeutigen Schlüssel und kann mit Hilfe der get-Methoden und ihres Schlüssels ausgelesen werden. Zusatzinformationen sind normalerweise schreibgeschützt. Um eine Eigenschaft einer Zusatzinformation

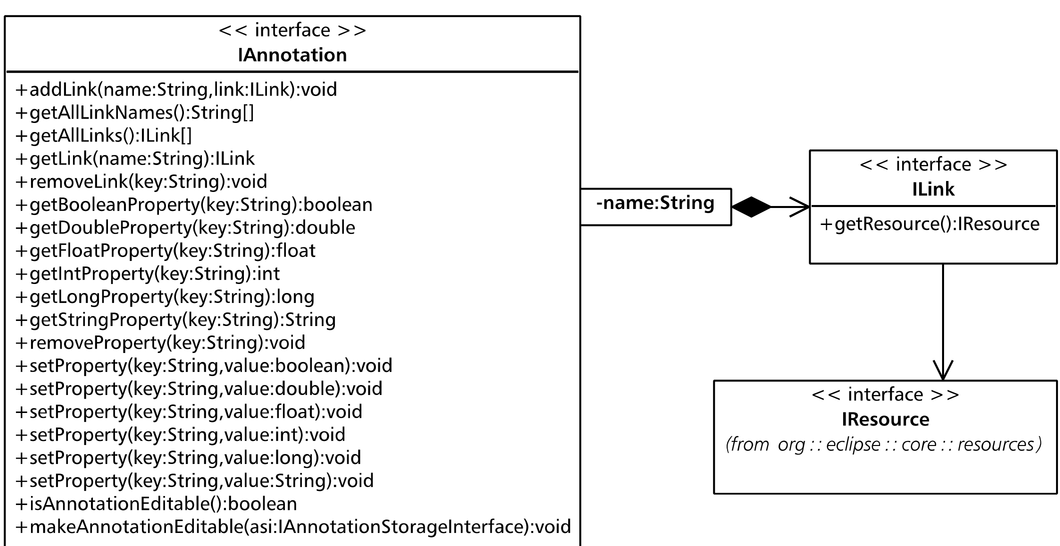


Abbildung C.2: Klassendiagramm „Zusatzinformation“
(Paket de.jwertenauer.codation.core.model)

setzen zu können, muss die Zusatzinformation zuvor editierbar gemacht werden (Methode `makeAnnotationEditable`, siehe Abschnitt C.3 auf Seite 114 bis 117).

Zusätzlich zu den Daten enthält eine Zusatzinformationen mindestens einen Verweis auf eine Ressource. Verweise implementieren die Schnittstelle `ILink` (siehe Abschnitt C.4 auf Seite 117 bis 119) und besitzen einen Namen, der innerhalb einer Zusatzinformation eindeutig sein muss. Wie auch Properties können Verweise nur geändert werden, wenn die Zusatzinformation zuvor editierbar gemacht wurde.

Wird versucht, eine Zusatzinformation zu speichern, die keinen Verweis auf eine Ressource enthält, erzeugt der Storage-Service automatisch einen Verweis mit dem Namen „default“, der auf das Projekt zeigt, zu dem die Zusatzinformation gehört (siehe Abschnitt C.3 auf Seite 114 bis 117).

C.3 Verwalten der Zusatzinformationen

In Abbildung C.3 auf der nächsten Seite sind die für die Verwaltung von Zusatzinformationen relevanten Schnittstellen und Klassen dargestellt.

Über die Schnittstelle `IAnnotationQueryInterface` kann ein Annotation-Provider nach Zusatzinformationen suchen. Instanzen der vom Storage-Service-Provider

<< interface >>
IAnnotationChangeAction

+run(storage:IAnnotationStorageInterface,monitor:IProgressMonitor):void
+getAnnotationProviderID():String

<< interface >>
IAnnotationQueryInterface

+getAnnotationsForResource(resource:IResource):IAnnotation[]
+getAllAnnotations():IAnnotation[]

<< interface >>
IAnnotationStorageInterface

+createAnnotation():IAnnotation
+removeAnnotation(annotation:IAnnotation):void

<< interface >>
IStorageService

+getAllAnnotationsForResource(resource:IResource,annotationProviderID:String[]):Map<String, IAnnotation[]>
+getAnnotationQueryInterface(providerID:String,project:IProject):IAnnotationQueryInterface
+getAnnotationStorageInterface(providerID:String,project:IProject):IAnnotationStorageInterface
+revertChanges(changes:IAnnotationStorageInterface):void
+saveChanges(changes:IAnnotationStorageInterface):void
+saveChangesOfAPs(changes:List<IAnnotationStorageInterface>):void

<< Singleton >>
CodationCorePlugin
(from de :: jwertenauer :: codation :: core)

+getDefault():CodationCorePlugin
+createContainerLink(container:IProject):IContainerLink
+createContainerLink(container:IFolder):IContainerLink
+getAnnotationQueryInterface(providerID:String,project:IProject):IAnnotationQueryInterface
+run(changeAction:IAnnotationChangeAction,project:IProject):void
+getFileLinkProvider(linkProviderID:String):IFileLinkProvider
+isAnnotationProviderActive(annotationProviderID:String):boolean
+isCodationActivatedForProject(project:IProject):boolean
+isCodationActive():boolean
+isCodationFile(file:IFile):boolean

Abbildung C.3: Klassendiagramm „Verwaltung von Zusatzinformationen“ (Paket de.jwertenauer.codation.core.storage)

implementierten Schnittstelle erhält der Annotation-Provider über die Methode `CodationCorePlugin#getAnnotationQueryInterface`. Es findet *keine* direkte Kommunikation zwischen Annotation-Providern (oder File-Link-Providern) und einem Storage-Service statt.

Um Änderungen an Zusatzinformationen durchführen zu können, wird eine Instanz der Klasse benötigt, die die Schnittstelle `IAnnotationStorageInterface` implementiert. Diese Klasse wird vom Storage-Service-Provider zur Verfügung gestellt und *nicht* von Annotation-Providern instanziert. Instanzen erhält der Annotation-Provider auf zwei Arten:

1. Bei der Synchronisation, siehe Abschnitt C.5 auf Seite 119 bis 121
2. Durch Implementieren der Schnittstelle `IAnnotationChangeAction`. Der Annotation-Provider implementiert diese Schnittstelle und lässt in der Methode `run` die benötigten Operationen durchführen. Eine Instanz der erstellten Klasse wird als Parameter der Methode `CodationCorePlugin#run` übergeben. Das Vorgehen ist vergleichbar mit einer `IWorkbenchChangeAction` des Resources-Plugins und in Abbildung C.4 dargestellt.

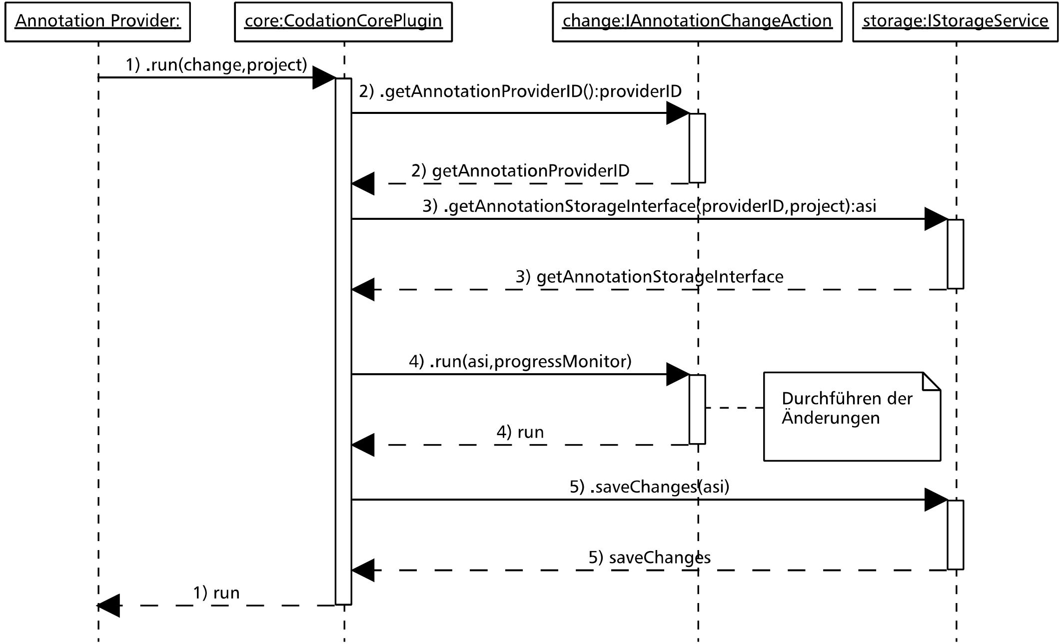


Abbildung C.4: Sequenzdiagramm „Änderung von Zusatzinformationen“

Möchte ein Annotation-Provider eine Zusatzinformation ändern, muss er sie zuvor mit der Methode `makeAnnotationEditable` editierbar machen. Andernfalls bricht jeder Versuch, die Zusatzinformation zu ändern, mit einer `IllegalStateException` ab. Wird nur ein Verweis geändert, muss die Zusatzinformation trotzdem zuvor editierbar gemacht werden, damit der Storage-Service die Änderungen speichert.

C.4 Verweise auf Ressourcen

In Abbildung C.5 auf der nächsten Seite sind die Schnittstellen dargestellt, die für die Erstellung von Verweisen auf Ressourcen (siehe Begriffserklärung 6.1 auf Seite 55) wichtig sind. Alle Verweise leiten von der Schnittstelle `ILink` ab und verweisen entweder auf Projekte und Ordner (`IContainerLink`) oder Dateien (`IFileLink`). Kann ein Verweis nicht wiederhergestellt werden, weil die Datei gelöscht wurde, wird vom Storage-Service eine Instanz von `IBrokenLink` zurückgegeben. Beim nächsten Speichern der betroffenen Zusatzinformation wird dieser Verweis entfernt.

Eine wichtige Grundregel, deren Einhaltung vom Storage-Service sichergestellt werden muss, ist, dass alle Ressourcen, auf die eine Zusatzinformation verweist, zum selben Projekt gehören müssen. Diese Einschränkung ist notwendig, weil Codation für jedes Projekt im Workspace einzeln (de)aktiviert werden kann.

Ein Storage-Service-Provider definiert, welche Arten von Verweisen sein Storage-Service abspeichern kann (Verweise auf Verzeichnisse, dateiübergreifende Verweise, verzeichnisübergreifende Verweise). Ein Annotation-Provider muss angeben, welche Arten von Verweisen er unter Umständen benötigt und welche File-Link-Provider er verwendet. Beim Start von Codation wird für jeden Annotation-Provider geprüft, ob alle von ihm angelegten Verweise vom Storage-Service gespeichert werden können und ob alle benötigten File-Link-Provider korrekt konfiguriert sind. Annotation-Provider, deren Anforderungen nicht erfüllt sind, werden nicht aktiviert. Ob ein Annotation-Provider aktiviert wurde, kann über die Methode `CodationCorePlugin#isAnnotationProviderActive` bestimmt werden.

Sofern der Storage-Service Verweise auf Verzeichnisse unterstützt, können diese über die Methode `CodationCorePlugin#createContainerLink` angelegt werden.

Die Schnittstelle `IFileLink` wird von File-Link-Providern implementiert, die zusätzlich die Schnittstelle `IFileLinkProvider` implementieren. File-Link-Provider fügen zusätzliche Methoden hinzu, die spezifisch für den Dateityp sind, in den über

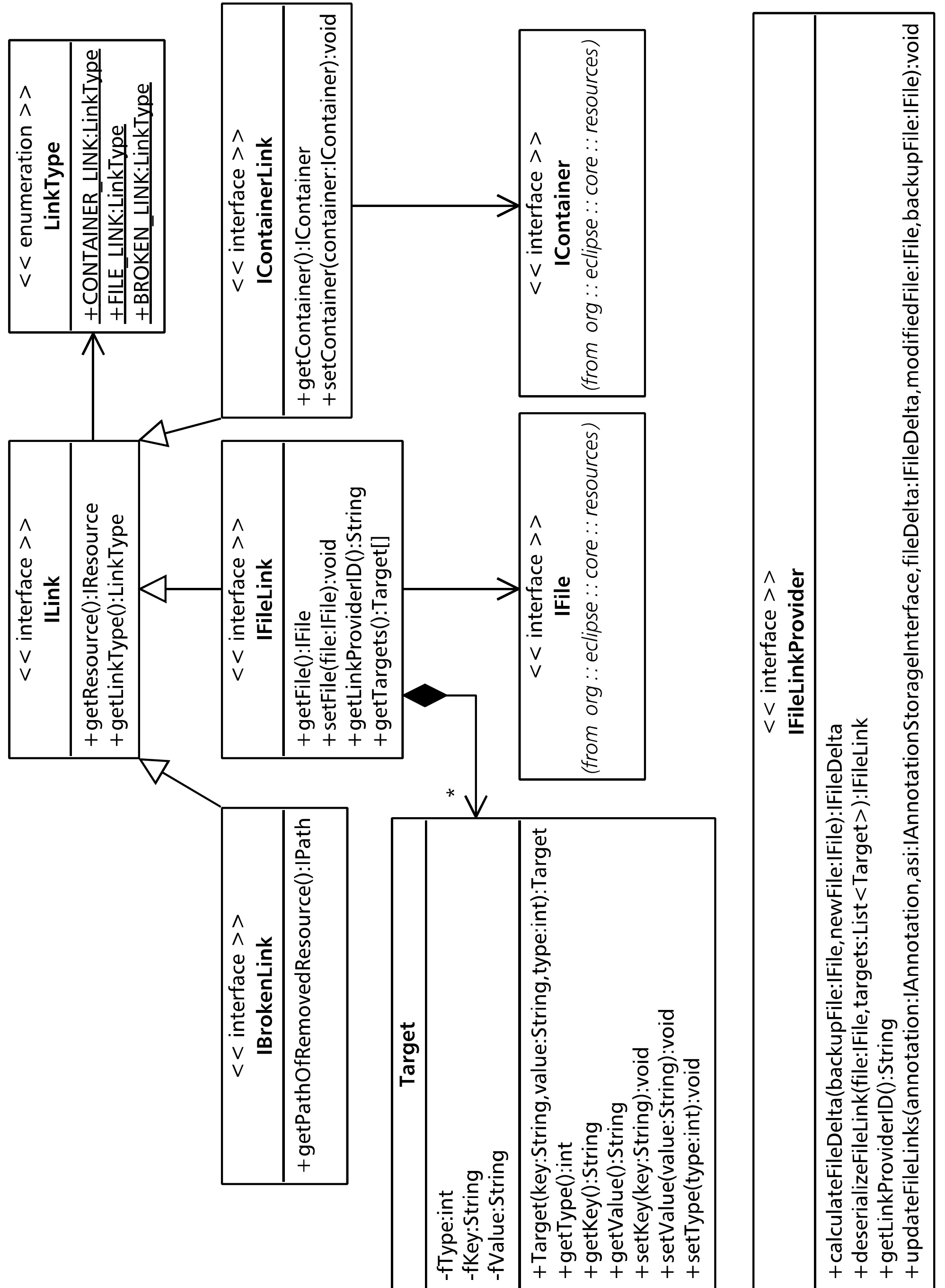


Abbildung C.5: Klassendiagramm „Verweise auf Ressourcen“
(Paket de.jwertenauer.codation.core.model)

sie verwiesen wird. Bei Verweisen in Dokumente können dies beispielsweise Methoden zur Festlegung der Seitenzahl sein. Annotation-Provider erhalten Instanzen von `IFileLinkProvider` über die Methode `CodationCorePlugin#getFileLinkProvider`, sie instanzieren die jeweiligen Klassen *nicht* selbst.

Zur Speicherung von `IFileLinks` müssen diese auf ein Feld der Klasse `Target` abgebildet werden. Dazu wird vom File-Link-Provider die Methode `IFileLink#getTargets` implementiert. Das zurückgegebene Feld muss alle Daten (außer der Ressource, auf die der Verweis zeigt) enthalten, um den Verweis wiederherstellen zu können. Zur Wiederherstellung wird vom Storage-Service die Methode `deserializeFileLink` des entsprechenden File-Link-Providers aufgerufen.

C.5 Synchronisation

Die Synchronisation ist einer der wichtigsten Teile von Codation und wird vom Anwendungskern ausgelöst. Zur Synchronisation wird ein Project-Builder verwendet, dessen Aufgabe es ist,

- die Backups zu verwalten, die zur Berechnung der Änderungsinformationen verwendet werden,
- die Berechnung der Änderungsinformationen durch die relevanten File-Link-Provider anzustoßen und
- die betroffenen Annotation-Provider über die Änderungen zu informieren.

Über eine Project-Nature wird der Project-Builder einem Projekt zugeordnet. Die Zuordnung nimmt der Benutzer von Codation über die Eigenschaften von Projekten vor, denen das UI-Plugin einen weiteren Eintrag hinzufügt.

Die Backups sind schreibgeschützte Kopien der Original-Dateien, deren Dateinamen das Suffix „.codation_backup“ angehängt wurde.

C.5.1 Methoden zur Änderungserkennung

Incremental build

Wenn der Project-Builder für Projekt aktiviert wurde, löst Eclipse automatisch einen *Incremental build* aus, nachdem sich eine Ressource geändert hat (siehe auch Abschnitt 6.3.2 auf Seite 67 bis 71). In diesem Fall stellt Eclipse die Information zur

Verfügung, welche Ressource geändert wurde. Darauf aufbauend berechnet der Anwendungskern Änderungsinformationen und benachrichtigt die betroffenen Annotation-Provider. Für alle übergeordneten Verzeichnisse wird anschließend ebenfalls eine Information über die Änderung an die Annotation-Provider verschickt.

Wird beispielsweise die Ressource „datei" mit dem Pfad „/projekt/ordner/datei" geändert, werden drei Ereignisse ausgelöst. Beim ersten werden alle Annotation-Provider, die Verweise auf „datei" angelegt haben, benachrichtigt. Das zweite Ereignis informiert alle Annotation-Provider, die Verweise auf „ordner" angelegt haben, das dritte alle, die Verweise auf das Projekt „projekt" angelegt haben.

Full build

Der Project-Builder wird auch aufgerufen, wenn Codation für ein Projekt aktiviert wurde, wenn Eclipse gestartet wurde oder wenn der Benutzer dies angefordert hat (über den Menüeintrag „Project → Clean"). In diesem Fall löst Eclipse einen *Full build* aus, bei dem der Project-Builder über alle Ressourcen des Projekts iteriert und prüft, ob Ressourcen geändert wurden. Um dies zu ermöglichen, speichert der Project-Builder für jede Ressource in einem Flag, ob die Ressource bereits früher bei einer Synchronisation besucht wurde. Das Flag wird über die Methode `IResource#setPersistentProperty` gesetzt.

Eine Datei gilt als verändert, wenn ein Backup existiert, dessen Änderungsdatum sich vom Änderungsdatum der Datei unterscheidet oder wenn das Flag gesetzt ist, aber das Backup fehlt (in diesem Fall ist keine Berechnung der Änderungsinformationen möglich). Ist nur noch das Backup vorhanden, wurde die dazugehörende Datei gelöscht. Ist kein Backup vorhanden und das Flag nicht gesetzt, wurde die Datei seit dem letzten Build hinzugefügt.

Bei Verzeichnissen ist die Erkennung von Änderungen schwieriger. Es kann lediglich das Hinzufügen eines Verzeichnisses festgestellt werden, da in diesem Fall das Flag nicht gesetzt wurde.

Da Eclipse bei Änderungen an Ressourcen immer ein `IResourceDelta` zur Verfügung stellt, das einen Incremental build ermöglicht, wird die oben beschriebene Methode zur Erkennung geänderter Ressourcen nur benötigt, nachdem Codation aktiviert wurde.

In Abbildung C.6 auf Seite 122 sind die Klassen des Project-Builder und der Project-Nature dargestellt. Die übrigen Klassen des Diagramms sind dafür zuständig, In-

formationen über die Änderungen zu sammeln und an die betroffenen Annotation-Provider weiterzuleiten. Die Informationen werden in der Schnittstelle `ISynchronisationEvent` zusammengefasst, die in Abbildung C.7 auf Seite 123 dargestellt ist.

C.5.2 Berechnung von Änderungsinformationen

Die Berechnung der Änderungsinformationen für eine geänderte Datei erfolgt nach folgendem Algorithmus:

1. Alle Zusatzinformationen bestimmen, die einen Verweis auf die geänderte Datei enthalten. Das Ergebnis nach Annotation-Providern sortieren und für jeden enthaltenen Annotation-Provider folgende Schritte durchführen:
   a) Für alle Zusatzinformationen des Annotation-Providers folgende Schritte durchführen:
      i. Alle Verweise auf die geänderte Datei aktualisieren (Methode `IFileLinkProvider#updateFileLinks`). Dies gibt den File-Link-Providern zum Beispiel die Möglichkeit, Offsets nachzuziehen (siehe auch Abschnitt C.6.2 auf Seite 124 bis 127).
      ii. Für alle Verweise auf die geänderte Datei prüfen, ob sie durch die Änderung betroffen sind (Methode `IFileDelta#isFileLinkAffected`). Das `IFileDelta` eines File-Link-Providers enthält detaillierte Änderungsinformationen und wird beim ersten Zugriff über den `IDeltaService` durch den File-Link-Provider berechnet (Methode `IFileLinkProvider#calculateFileDelta`). Können Änderungsinformationen nicht berechnet werden, weil beispielsweise das Backup fehlt, gilt jeder Verweis auf die geänderte Datei als betroffen.
      iii. Enthält die Zusatzinformation betroffene Verweise (`IAffectedLink`), wird eine `IAffectedAnnotation` erstellt.
   b) Den Annotation-Provider benachrichtigen.
2. Änderungen an den Zusatzinformationen speichern
3. Backup der Datei aktualisieren: Backup mit neuer Version des Originals ersetzen und Schreibschutz aktivieren. Änderungsdatum des Backups auf das des Originals setzen und abschließend Workspace aktualisieren.

Der größte Teil der Synchronisation wird somit vom Anwendungskern durchgeführt. File-Link-Provider müssen „nur“ den Vergleichsalgorithmus zur Verfügung stellen und bestimmen, ob ein Verweis durch die festgestellten Änderungen betroffen ist.

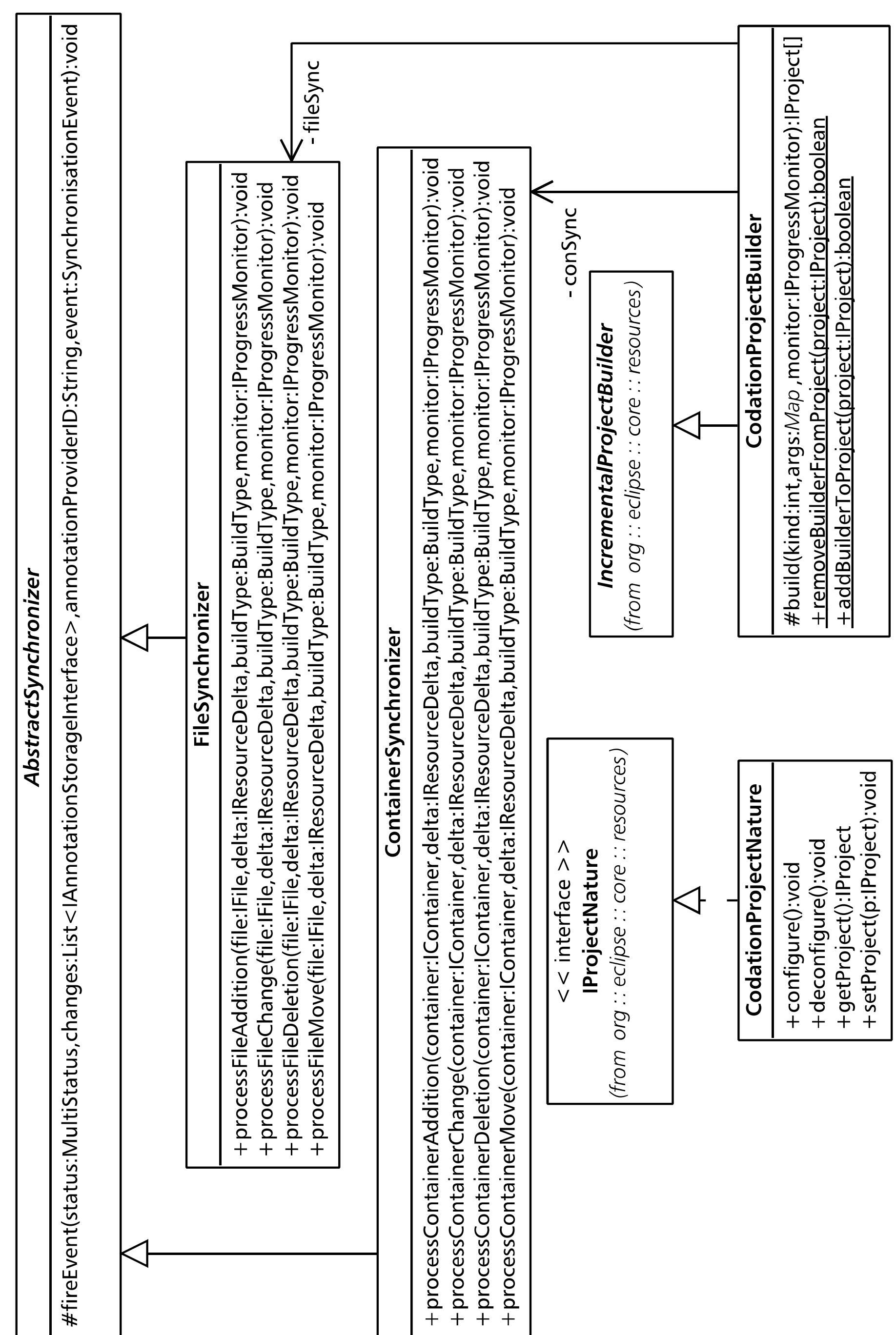


Abbildung C.6: Klassendiagramm „Project-Builder“
(Paket de.jwertenauer.codation.core.internal.synchronisation)

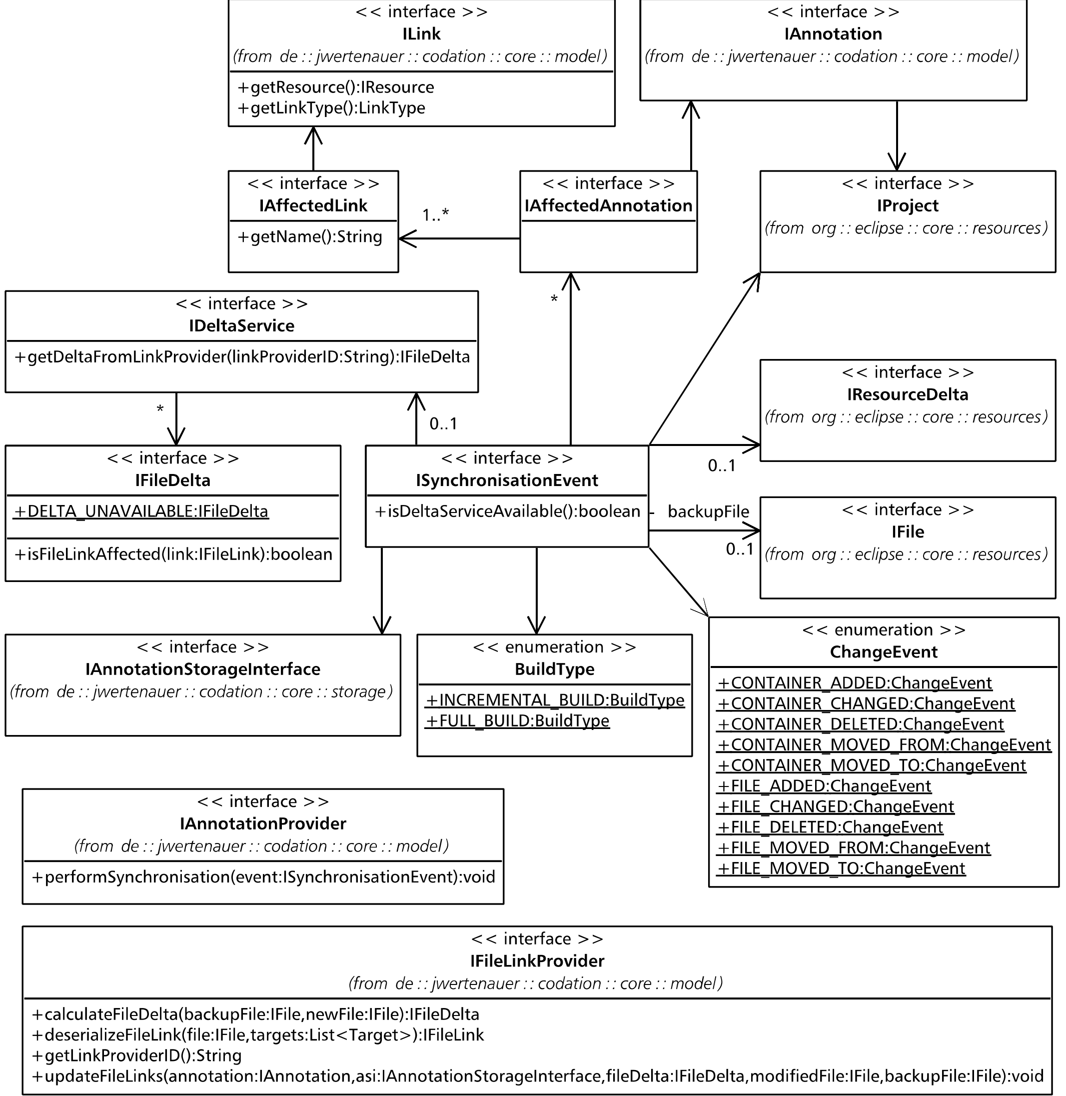


Abbildung C.7: Klassendiagramm „Änderungsinformationen“
(Paket de.jwertenauer.codation.core.synchronisation)

C.6 File-Link-Provider

C.6.1 Simple-Link-Provider

Der Simple-Link-Provider, dessen API in Abbildung C.8 dargestellt ist, erlaubt Verweise auf beliebige Dateien, wobei jedoch kein spezieller Bereich innerhalb einer Datei angegeben werden kann.

Da außer dem Pfad des Verweisziels keine Informationen benötigt werden, wird bei der Speicherung ein leeres Feld zurückgegeben (siehe Abschnitt C.4 auf Seite 117 bis 119).

Bei der Synchronisation berechnet der Simple-Link-Provider keine Änderungsinformationen, die Methode `SimpleLinkProvider#calculateFileDelta` gibt also stets `IFileDelta#DELTA_UNAVAILABLE` zurück. Dementsprechend sieht der Anwendungskern jeden Verweis als von einer Änderung betroffen an.

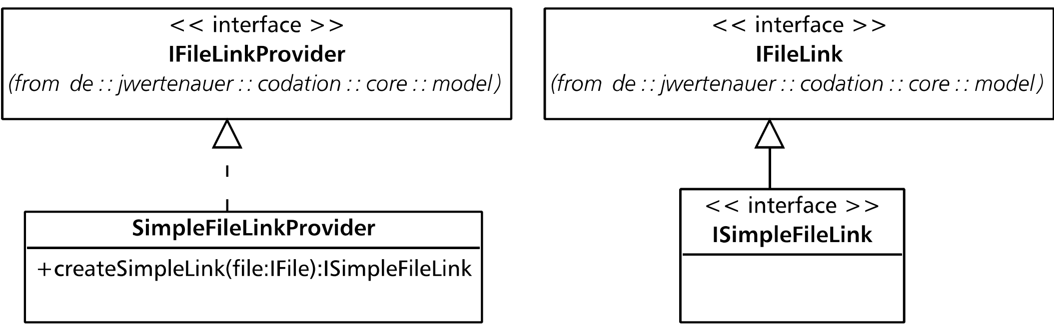


Abbildung C.8: Klassendiagramm „Simple-Link-Provider“
(Paket de.jwertenauer.codation.simplelinkprovider)

C.6.2 Java-Link-Provider

Der Java-Link-Provider erlaubt Annotation-Providern, auf AST-Knoten zu verweisen. Die API ist in Abbildung C.9 auf der nächsten Seite dargestellt. Ein Verweis wird durch die Klasse `IJavaASTLink` repräsentiert und kann auf mehrere AST-Knoten und Sequenzen von AST-Knoten verweisen, wobei Sequenzen durch ihren ersten und letzten AST-Knoten definiert sind.

Für einen Verweis auf einen AST-Knoten werden drei `Targets` erzeugt, die Startposition, Länge und Knotentyp speichern (siehe auch Abschnitt C.4 auf Seite 117

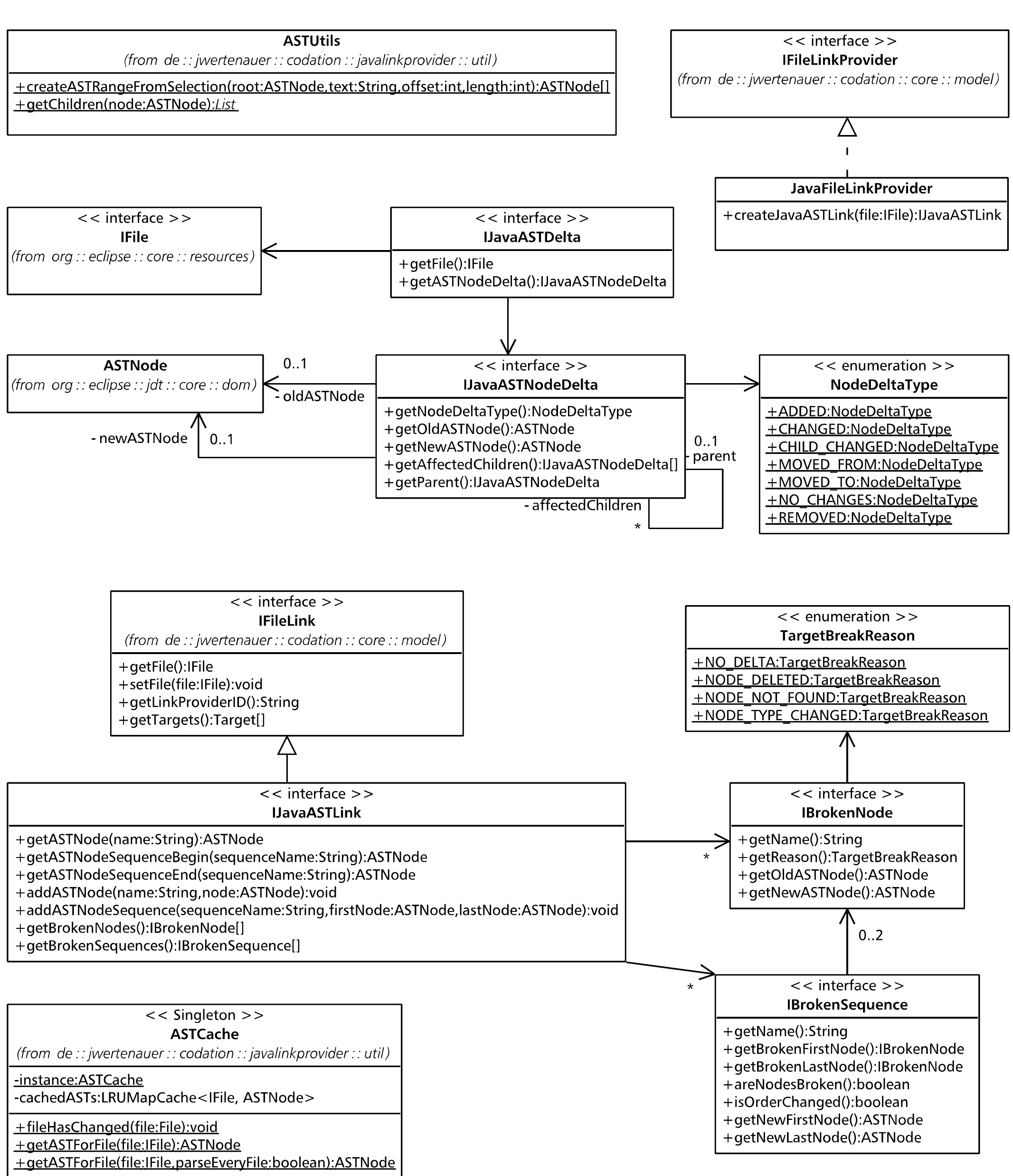


Abbildung C.9: Klassendiagramm „Java-Link-Provider“
(Paket de.jwertenauer.codation.javalinkprovider)

bis 119). Bei Verweisen auf AST-Sequenzen werden sechs `Targets` erzeugt, jeweils drei mit Startposition, Länge und Knotentyp für Start- und Endknoten.

Synchronisation

Zur Berechnung der Änderungsinformationen verwendet der Java-Link-Provider den in Abschnitt 6.3.4 auf Seite 71 bis 77 beschriebenen Algorithmus. Die Elemente des Editiergraphen sind in Abbildung C.10 dargestellt.

Um den Editiergraph zu erzeugen, wird der AST zunächst in Preorder-Reihenfolge traversiert. Dabei wird für jeden AST-Knoten eine Instanz von `PreOrderNode` angelegt, in der Position und Tiefe des AST-Knotens im AST gespeichert sind. Ein Paar dieser `PreOrderNodes` bildet einen Knoten im Editiergraphen (Klasse `EditGraphNode`). Zwei Knoten werden über eine `EditGraphEdge` verbunden. Diese Kante

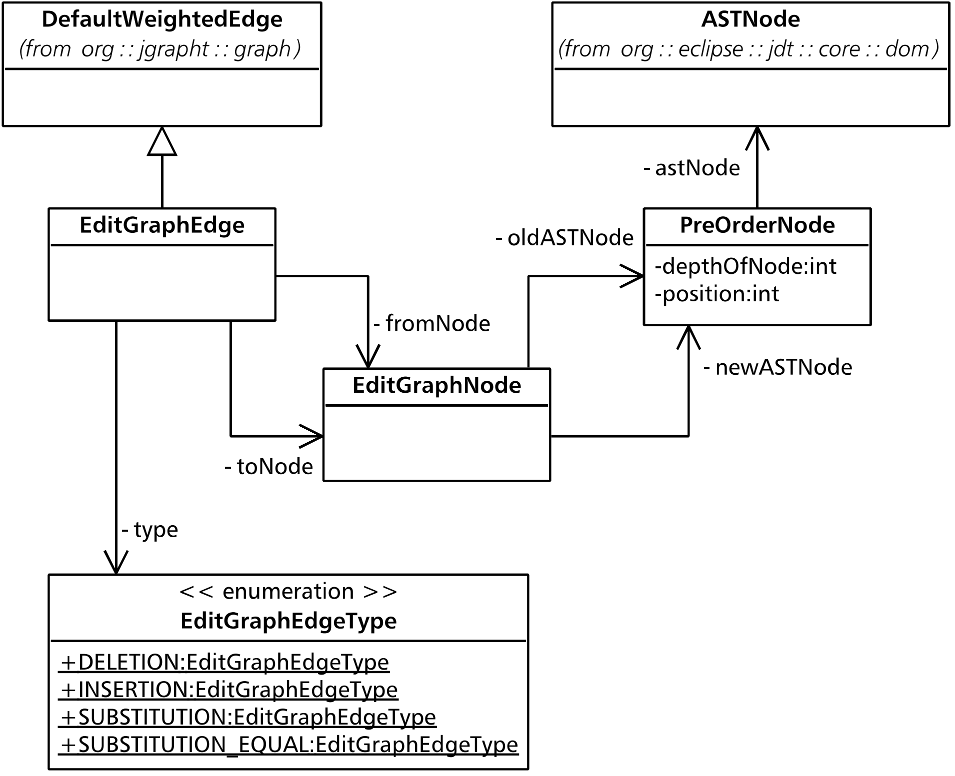


Abbildung C.10: Klassendiagramm „Editiergraph“
(Paket de.jwertenauer.codation.javalinkprovider.internal.model)

repräsentiert eine der in der Enumeration `EditGraphEdgeType` festgelegten Operationen. Um die Größe des Editiergraphen kleiner zu halten, arbeitet der Vergleichsalgorithmus in zwei verschiedenen Modi. Die Modi werden in Abschnitt 6.3.4 auf Seite 71 bis 77 beschrieben.

Aus dem kürzesten Weg durch den Editiergraphen, der mit Hilfe der Bibliothek JGrapht berechnet wird, werden die Änderungsinformationen berechnet, die als Instanz der Klasse `IJavaASTDelta` an den Anwendungskern weitergeleitet werden. Instanzen der Klasse `IJavaASTNodeDelta` repräsentieren die Änderungen an AST-Knoten, der `NodeDeltaType` die Art der Änderung.

Nachführung von Verweisen

Damit der Java-Link-Provider beim Laden eines Verweises die Verweisziele wiederfinden kann, müssen bei der Synchronisation die `Targets` nachgezogen werden. Dies erfolgt in der Methode `JavaFileLinkProvider#updateFileLinks` auf Basis der Änderungsinformationen. Kann ein Verweisziel nicht aktualisiert werden, weil z. B. der Zielknoten gelöscht wurde, wird eine Instanz von `IBrokenNode` erzeugt, die über die Methode `IJavaASTLink#getBrokenNodes` abgefragt werden kann. Analog dazu existiert die Methode `IJavaASTLink#getBrokenSequences` zur Abfrage von Sequenzen, die nicht aktualisiert werden konnten. Nicht aktualisierbare Verweisziele werden beim Speichern entfernt.

Werkzeuge

Das Paket `de.jwertenauer.codation.javalinkprovider.util` enthält zwei Werkzeuge, die auch von anderen Plug-Ins verwendet werden können.

Die Klasse `ASTCache` enthält einen Zwischenspeicher für ASTs von Java-Klassen. Der Zwischenspeicher ist in der Größe beschränkt und enthält immer die zehn zuletzt verwendeten ASTs.

Die Klasse `ASTUtils` bietet die Möglichkeit, die Kindknoten eines AST-Knotens zu bestimmen. Außerdem implementiert sie den in Abschnitt 6.2.2 auf Seite 60 bis 64 beschriebenen Algorithmus zur Bestimmung der von einer Auswahl im Editor umfassten AST-Knoten.

C.7 XML-Storage-Service

Das Plug-In `de.jwertenauer.codation.storage.xml` speichert als Storage-Service Zusatzinformationen in XML-Dateien. Die XML-Dateien werden mit Hilfe der Bibliothek JDOM eingelesen und gespeichert, da diese Bibliothek einen komfortablen Zugriff auf den Datei-Inhalt erlaubt. Die Abbildungen C.11 und C.12 zeigen den Aufbau der erzeugten Dateien als XML-Schema-Definition.

Für jeden Annotation-Provider wird im Wurzelverzeichnis des Projekts eine Datei angelegt, deren Name aus der ID des Annotation-Providers mit dem Suffix „.codation_data“ besteht. Damit verwendet der Storage-Service für die getrennte Speicherung von Zusatzinformationen und Quelltext den in Abschnitt 6.1.2 auf Seite 55 bis 58 beschriebenen Ansatz 3 a zur Aufteilung der Daten auf mehrere Dateien.

In Abbildung C.13 auf Seite 131 sind die Klassen dargestellt, die vom Storage-Service verwendet werden. Der Zugriff auf die Dateien mit den Zusatzinformation eines Projekts erfolgt über den Singleton `DataManager`. Dieser hält die Instanzen der Klasse `DataFile` vor, die jeweils eine XML-Datei repräsentieren. Die Klasse `DataFile` speichert aus der XML-Darstellung (Klasse `Element`) geparste Zusatzinformationen zwischen und stellt die benötigten Suchfunktionen, die intern mit XPath-Anfragen realisiert werden, zur Verfügung.

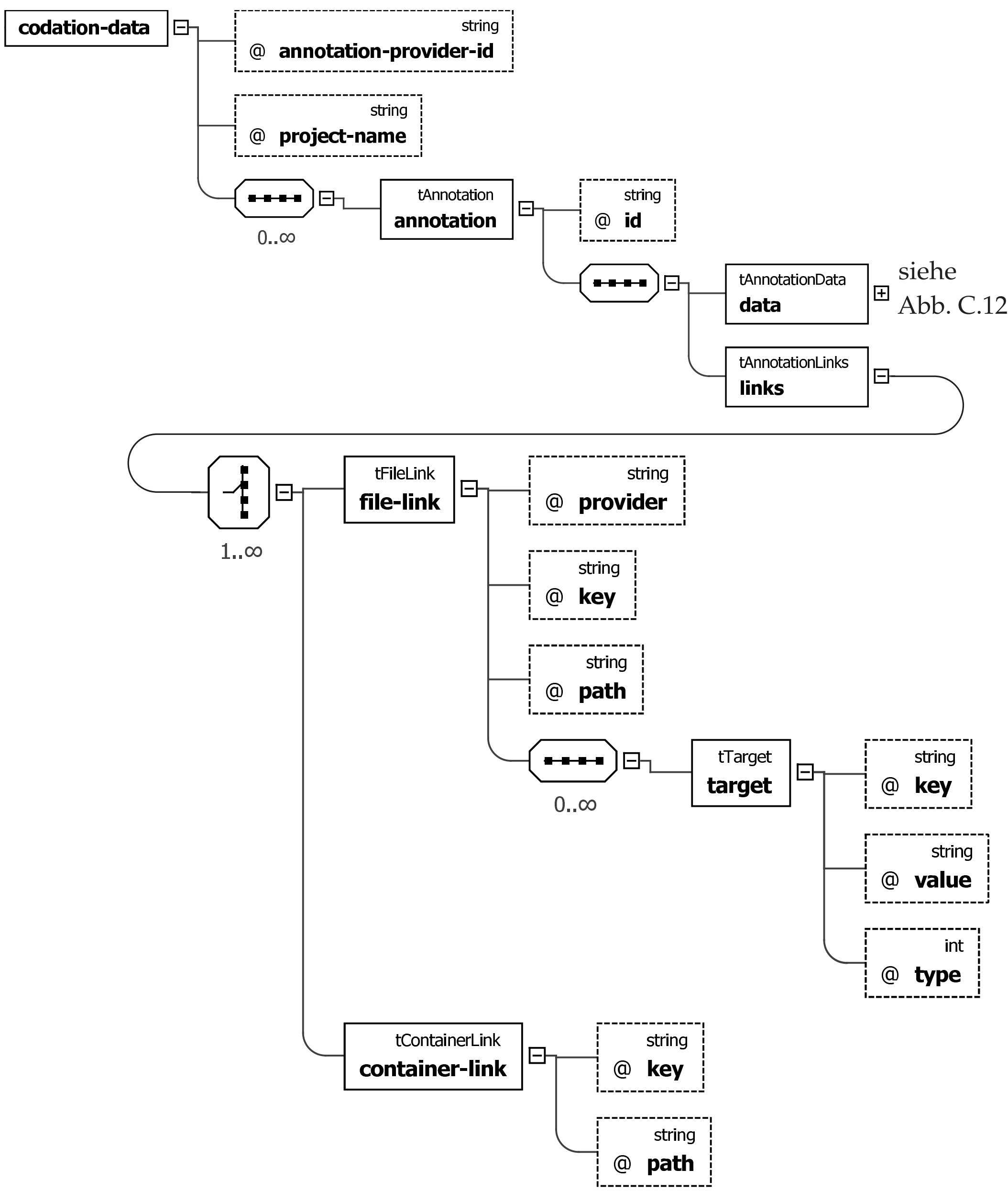


Abbildung C.11: XML-Schema zur Speicherung (Teil 1)

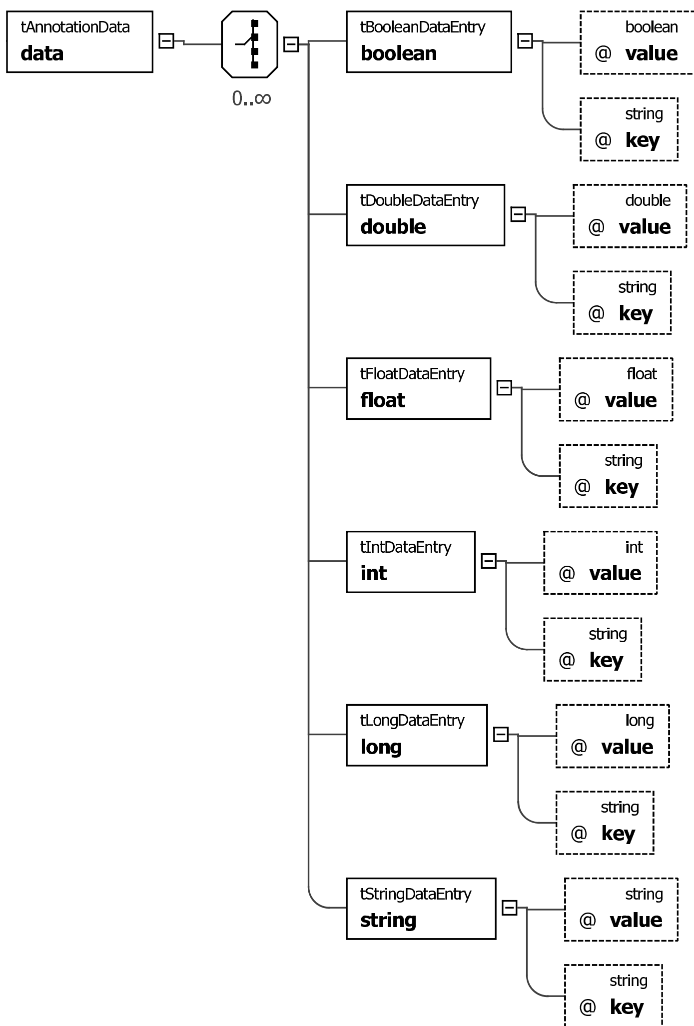


Abbildung C.12: XML-Schema zur Speicherung (Teil 2)

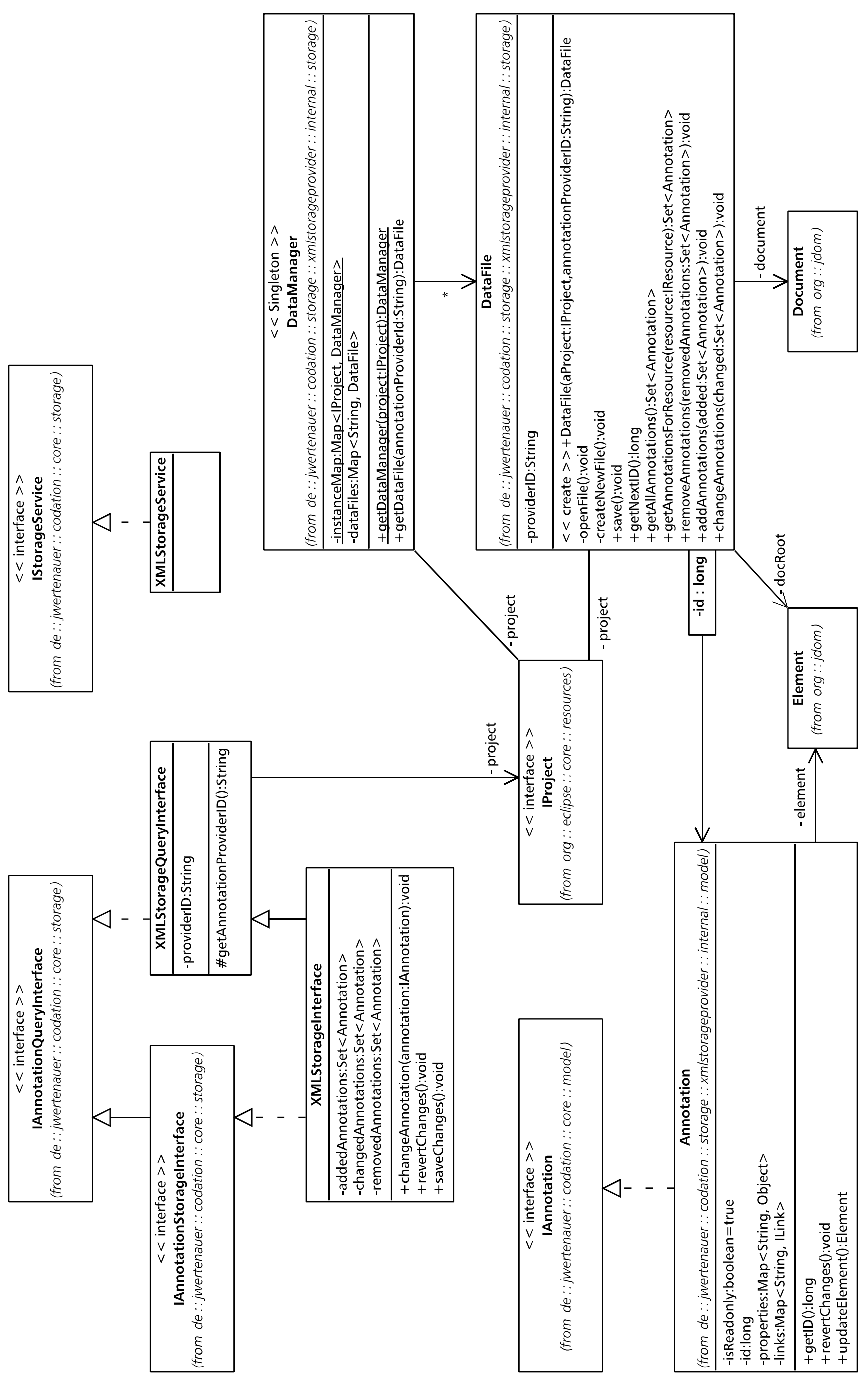


Abbildung C.13: Klassendiagramm „XML-Storage-Service“ (Paket de.jwertenauer.codation.storage.xmlstorageprovider.internal.storage)

C.8 Beschreibungen der Extension-Points

Dieser Abschnitt enthält die Beschreibungen der Extension-Points, wie sie im Dokumentations-Plugin enthalten sind. Wie die Kommentare des Quelltextes und die Hilfe wurden sie in englischer Sprache verfasst.

C.8.1 Annotation-Provider

Identifier

de.jwertenauer.codation.core.annotationProvider

Description

By extending this extension point one can create a so called annotation provider. In Codation, an annotation provider is a component that creates annotations. Annotations are small pieces of information that are created to store additional information to a resource in the Eclipse workspace (except the workspace root).

To point on resources, instances of `de.jwertenauer.codation.core.model.ILink` are used. Creating an link does not change the annotated resource. To point into files, an annotation provider uses file link providers (see corresponding extension point).

Annotations are saved using a storage service. Depending on the storage service, different kinds of links on resources can be saved. For example, a certain storage service might not be able to save links on folders and projects. Therefore a storage service registers its possibilities. Your extension has to specify what possibilities it requires. If the storage service used by codation does not meet these requirements, your extension will be disabled.

This mechanism ensures, that your extension will not come to a point, where it is not able to store its annotations. So be careful when specifying the needs of your extension.

For the same reason, an annotation provider has to specify, which file link provider it uses.

Configuration Markup

```
<!ELEMENT extension (annotationProvider)>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED>

<!ELEMENT annotationProvider (fileLinkProviderRequirements)>
<!ATTLIST annotationProvider
  id                           CDATA #REQUIRED
  name                         CDATA #IMPLIED
  class                        CDATA #REQUIRED
  usesFileSpanningLinks        (true | false)
  usesDirectorySpanningLinks   (true | false)
  createsLinksOnDirectories    (true | false) >
```

This element represents an annotation provider as described above.

id Unique ID of the annotation provider. It may consist of

- a-z
- A-Z
- 0-9
- "."
- "_"
- "-"

name Name of the annotation provider. This name will be used in UI messages. If you leave it empty, the ID will be taken instead.

class A class implementing `de.jwertenauer.codation.core.model.IAnnotationProvider`. This class must provide a public default constructor (a public constructor without parameters).

usesFileSpanningLinks This flag tells codation, that this annotation provider might create annotations containing links into two or more different files.

usesDirectorySpanningLinks This flag tells codation, that this annotation provider might create annotations that contain links into two or more different files located in different folders (of the same project).

createsLinksOnDirectories This flag tells codation, that this annotation provider might use `de.jwertenauer.codation.core.model.IContainerLinks`.

```
<!ELEMENT fileLinkProviderRequirements (requiredFileLinkProvider)>
```

This sequence allows an annotation provider to specify, which file link providers it uses. The annotations provider will only be activated if all file link providers are available.

```
<!ELEMENT requiredFileLinkProvider EMPTY>
<!ATTLIST requiredFileLinkProvider
  id CDATA #REQUIRED>
```

This element represents a required file link provider.

id Fully qualified ID of a required file link provider.

Examples

```
<extension point=
    "de.jwertenauer.codation.core.annotationProvider">
  <annotationProvider
      id="de.jwertenauer.codation.aps.javaFanOut.annotationProvider"
      name="Java Fan Out"
      class="de.jwertenauer.codation.aps.javaFanOut.internal.
             FanOutAnnotationProvider"
      createsLinksOnDirectories="true"
      usesDirectorySpanningLinks="false"
      usesFileSpanningLinks="false">
    <fileLinkProviderRequirements>
      <requiredFileLinkProvider
          id="de.jwertenauer.codation.javalinkprovider.
              fileLinkProvider"/>
    </fileLinkProviderRequirements>
  </annotationProvider>
</extension>
```

API Information

Annotation providers must implement the interface `de.jwertenauer.codation.core.model.IAnnotationProvider` (and no other interface).

The interface `de.jwertenauer.codation.core.model.IAnnotation` (implemented by the storage service), represents an annotation.

For linking to projects and folders, take a look at the interface `de.jwertenauer.codation.core.model.IContainerLink`. Its documentation explains how to create instances. To create links into a file, you need an instance of `de.jwertenauer codation.core.model.IFileLinkProvider`. It can be retrieved by calling `de.jwertenauer.codation.core.model.CodationCorePlugin#getFileLinkProvider`. To create annotations and to manage the stored annotations, use the appropriate methods from `de.jwertenauer.codation.core.model.CodationCorePlugin` and take a look at the documentation of the following interfaces:

- de.jwertenauer.codation.core.storage.IAnnotationChangeAction
- de.jwertenauer.codation.core.storage.IAnnotationQueryInterface
- de.jwertenauer.codation.core.storage.IAnnotationStorageInterface

Supplied Implementation

See plugin `de.jwertenauer.codation.aps.useCaseManager` for an annotation provider.

C.8.2 File-Link-Provider

Identifier

de.jwertenauer.codation.core.fileLinkProvider

Description

By extending this extension point you can create a so called "file link provider" (FLP). A FLP allows annotation providers to create links into files to attach their data to a certain point within a resource. The created file links are automatically stored by the storage service, so it is not your task to store them.

Beside extending this extension point, you have to do three things:

1. Create a class implementing `IFileLink`: This class represents the link defined by you. Add methods to this class to set the position within the file, the link is pointing on. If you are creating a FLP that can create links into simple text files, one method could be `setOffset(int)`.

To enable the storage service to save the offset, the method `de.jwertenauer.codation.core.model.IFileLink#getTargets` must return an array containing a target with the offset, e. g.

```
public Target[] getTargets() {
    return new Target[] {
      new Target("offset", offset, 0)
    };
}
```

When a stored file link is loaded, you will get this target (but another instance) again and can restore the link (method `deserializeFileLinks` in IFileLinkProvider).

You must also create an publicly visible interface that declares the methods of your file link class, as annotation providers need it. The implementing class can safely be (and therefore should be) private. Your FLP is the only party that creates instances of it.

2. Create a class implementing `IFileLinkProvider`: This class represents the FLP you declare in this extension. Add factory methods to this class to enable annotation providers to use your fancy links.

The core component of Codation creates an instance of this class using the default constructor, but you do not have to make it visible to other plugins than your own. Especially, annotation providers do not create instances. At least they should not. If you hide the class from others, you have to provide a corresponding interface for it. This enables annotation providers to cast the generic instance of `de.jwertenauer.codation.core.model.IFileLinkProvider` they get from the core plugin and afterwards create new links.

3. Create a class implementing `IFileDelta`: This class represents change information and is created by your FLP when a file has been changed. A powerful delta algorithm providing accurate results is crucial!

Things to Remember:

1. Every annotation belongs to exactly one project. Links to other projects are not allowed.
2. In your file link, you have to define a method `getFile` that gives the core plugin and annotation providers the ability to check which file is linked. When this file is renamed, you have to update this information.
3. If you have to change some data of your links, e. g. during synchronisation, the changes are only saved, if you tell Codation that the annotation the link belongs to has been changed. It's sufficient to call the `makeAnnotationEditable` method of the annotation. The synchronisation methods provide all necessary instances.

Configuration Markup

```
<!ELEMENT extension (fileLinkProvider+)>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED>

<!ELEMENT fileLinkProvider (supportedFileExtension*)>
<!ATTLIST fileLinkProvider
  id               CDATA #REQUIRED
  name             CDATA #IMPLIED
  class            CDATA #REQUIRED
  supportsAllFiles (true | false) "false">
```

id ID of this file link provider.

name The name of the file link provider. This value is shown to the user. If it is omitted, the ID is used.

class A class that implements the interface `de.jwertenauer.codation.core.model.IFileLinkProvider`. A default constructor, i. e. a constructor without parameters, must be available and must be public.

supportsAllFiles This file link provider supports all files. Even those without an extension.

```
<!ELEMENT supportedFileExtension EMPTY>
<!ATTLIST supportedFileExtension
  extension CDATA #REQUIRED>
```

extension A file extension supported by this file link provider (without the point), e. g. java.

Examples

```
<extension
    id="javaFileLinkProvider"
    point="de.jwertenauer.codation.core.fileLinkProvider">
  <fileLinkProvider
      class="de.jwertenauer.codation.javalinkprovider.
             JavaFileLinkProvider"
      id="de.jwertenauer.codation.javalinkprovider.fileLinkProvider"
      name="Java File Link Provider">
    <supportedFileExtension extension="java"/>
  </fileLinkProvider>
</extension>
```

API Information

The only interfaces a file link provider has to implement are:

1. `de.jwertenauer.codation.core.model.IFileLinkProvider`,
2. `de.jwertenauer.codation.core.model.IFileLink`, and
3. `de.jwertenauer.codation.core.synchronisation.IFileDelta`.

Useful methods from other classes or interfaces, respectively, are:

- The methods beginning with “isCodation” from `de.jwertenauer.codation.core.CodationCorePlugin` allow you to check, if codation is activated for a project.
- The method `de.jwertenauer.codation.core.model.IAnnotation#makeAnnotationEditable` helps you to get your data stored.
- The class `de.jwertenauer.codation.core.model.Target` describes the data you have to store in order to restore the link at a later time.

When methods defined in `de.jwertenauer.codation.core.model.IFileLinkProvider` are called, you will get instances of several interfaces you may need for synchronisation. Beside the class `Target` mentioned above, there is no need to create instances of any class contained in the core plugin.

Supplied Implementation

Two file link providers are shipped with Codation. One creates links into Java files (plug-in `de.jwertenauer.codation.javalinkprovider`), the other creates dumb links on files without pointing into them (plugin `de.jwertenauer.codation.simplelinkprovider`).

C.8.3 Storage-Service-Provider

Identifier

de.jwertenauer.codation.core.storageServiceProvider

Description

In Codation, a storage service is used to store annotations that are created by annotation providers (see corresponding extension point). With this extension point, you can create your own storage service.

Your storage service might support annotations with different kinds of links (see below). During declaring your extension, you have to provide information about what your storage service supports. This data enables codation to disable annotation providers that won't be able to store their annotations. Therefore it is required that you provide accurate information.

Only one storage service is used by Codation. Codation will use the first one that it finds. So to ensure that your storage service is used, you have to tell the user to disable all other plugins that define storage services.

Configuration Markup

```
<!ELEMENT extension (storageService)>
<!ATTLIST extension
  point CDATA #REQUIRED
  id    CDATA #IMPLIED
  name  CDATA #IMPLIED>
```

A storage service provider provides exactly one storage service.

```
<!ELEMENT storageService EMPTY>
<!ATTLIST storageService
  class                             CDATA #REQUIRED
  allowsLinksOnDirectories          (true | false)
  allowsDirectorySpanningAnnotations (true | false)
  allowsFileSpanningAnnotations     (true | false)
  usesFiles                         (true | false) "true"
  filenamePattern                   CDATA "*.codation_data">
```

This element represents a storage service as described above. There must be exactly one of these elements.

class A class implementing `de.jwertenauer.codation.core.storage.IStorage-Service`. This class must have a public default constructor, i. e. a constructor without parameters and public visibility.

allowsLinksOnDirectories The defined storage service supports `de.jwertenauer.codation.core.model.IContainerLinks`.

allowsDirectorySpanningAnnotations The storage service supports annotations that contain links on resources that are located in different folders.

allowsFileSpanningAnnotations The storage service supports annotations that contain links on two or more different files.

usesFiles The defined storage service uses files to store the annotations. If this flag is set to true, you will also have to provide a value for the attribute filenamePattern.

filenamePattern A pattern that described all names (excluding folders) of the files, that are used by the storage service. The pattern may contain the wildcards “?” and “*”.

Examples

```
<extension
    point="de.jwertenauer.codation.core.storageServiceProvider">
  <storageService
    allowsDirectorySpanningAnnotations="true"
    allowsFileSpanningAnnotations="true"
    allowsLinksOnDirectories="true"
    class="de.jwertenauer.codation.storage.xmlstorageprovider.↩
           internal.storage.XMLStorageService"
    filenamePattern="*.codation.data"
    usesFiles="true"/>
</extension>
```

API Information

A storage service has to implement the following interfaces:

- `de.jwertenauer.codation.core.model.IAnnotation`
- `de.jwertenauer.codation.core.storage.IAnnotationQueryInterface`
- `de.jwertenauer.codation.core.storage.IAnnotationStorageInterface`
- `de.jwertenauer.codation.core.storage.IStorageService`

Supplied Implementation

Codation ships with the plug-in `de.jwertenauer.codation.storage.xml`, a storage service that supports all kinds of links and uses XML to store the data.

D Testplan

D.1 Einleitung

D.1.1 Zweck des Dokuments

Dieses Dokument stellt den Testplan zu Codation dar. Die darin aufgeführten Testfälle sind manuell durchzuführen und ergänzen die JUnit-Testfälle. Sie dienen dazu, die nicht oder nur mit großen Aufwand mit Unit-Tests testbaren Teile des Funktionsumfangs von Codation zu prüfen. Dies sind vornehmlich die Benutzeroberfläche und der Project-Builder, der zur Synchronisation eingesetzt wird.

D.1.2 Testdurchführung

Die Testfälle sind vor Veröffentlichung einer neuen Version von Codation durchzuführen. Ein negatives Testergebnis ist zwingende Voraussetzung dafür, dass die neue Version veröffentlicht werden darf.

Bei einem Testlauf sind stets alle aufgeführten Tests durchzuführen und die Ergebnisse zu protokollieren.

D.1.3 Nachführung des Testplans

Bei Änderungen an der zu implementierenden Funktionalität ist der Testplan nachzuführen. Außerdem muss, wie zuvor beschrieben, ein neuer Testlauf durchgeführt werden.

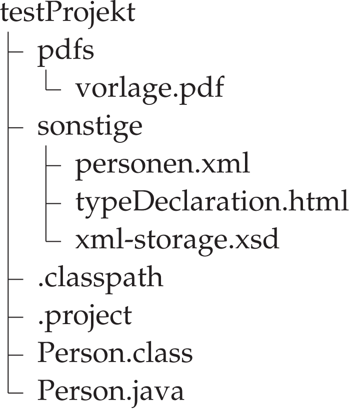
D.2 Testfälle

D.2.1 Core- und UI-Plugin

Vorbereitungen

Für die Testfälle in diesem Abschnitt muss ein Testprojekt angelegt werden:

1. Ein neues Java-Projekt mit dem Namen „testProjekt“ anlegen.
2. Die Archivdatei „testProjekt.zip“ in dieses Projekt importieren. Danach sollten folgende Dateien vorhanden sein (im Navigator überprüfen):



Codation (de)aktivieren

1. Vorbedingungen: Codation ist deaktiviert.
   Beschreibung: Aktivieren von Codation über die Projekteigenschaften.
   Nachbedingungen: Codation ist aktiviert. Das Log enthält einen entsprechenden Eintrag. Von jeder Datei wurde ein Backup angelegt. Außerdem wurde für den Use-Case-Manager eine Datei mit dem Namen „de.jwertenauer.codation.aps.useCaseManager .codation_data“ angelegt.
2. Vorbedingungen: Codation ist aktiviert.
   Beschreibung: Deaktivieren von Codation über die Projekteigenschaften. Danach aus der Datei „Person.java“ die Leerzeile am Anfang entfernen und Datei speichern.

Nachbedingungen: Codation ist deaktiviert. Das Log enthält eine entsprechende Nachricht. Das Backup der Datei „Person.java“ wurde nicht aktualisiert.

3. Vorbedingungen: Codation ist deaktiviert.
   Beschreibung: Erneutes Aktivieren von Codation über die Projekteigenschaften.
   Nachbedingungen: Codation ist aktiviert. Das Log enthält einen entsprechenden Eintrag. Das Backup der Datei „Person.java“ wurde aktualisiert.

Aktionen auf Dateien

4. Vorbedingungen: Codation ist aktiviert.
   Beschreibung: In die Datei „Person.java“ die Leerzeile wieder einfügen und Änderung speichern.
   Nachbedingungen: Das Backup der Datei „Person.java“ wurde aktualisiert.

5. Vorbedingungen: Codation ist aktiviert.
   Beschreibung: Im Verzeichnis „sonstige“ eine neue Datei mit dem Namen „test.txt“ anlegen (über Popup-Menü → *New* → *File*).
   Nachbedingungen: Die Datei wurde angelegt. Außerdem wurde ein Backup mit dem Namen „test.txt.codation_backup“ angelegt.

6. Vorbedingungen: Codation ist aktiviert.
   Beschreibung: Die angelegte Datei in „ein Test.txt“ umbenennen.
   Nachbedingungen: Der Dateiname des Backups wurde ebenfalls angepasst. Er lautet nun „ein Test.txt.codation_backup“.

7. Vorbedingungen: Codation ist aktiviert.
   Beschreibung: Die Datei „ein Test.txt“ löschen.
   Nachbedingungen: Das Backup der Datei wurde ebenfalls gelöscht.

8. Vorbedingungen: Codation ist aktiviert.
   Beschreibung: Ins Wurzel-Verzeichnis des Projekts eine beliebige Datei, z. B. dieses Dokument, importieren.
   Nachbedingungen: Die Datei wurde importiert und ein Backup angelegt.

9. Vorbedingungen: Codation ist aktiviert.
   Beschreibung: Die importierte Datei ins Verzeichnis „pdfs“ verschieben.
   Nachbedingungen: Das Backup der Datei wurde ebenfalls verschoben.

10. Vorbedingungen: Codation ist aktiviert.
    Beschreibung: Die verschobene Datei löschen.
    Nachbedingungen: Das Backup der Datei wurde ebenfalls gelöscht.

Aktionen auf Verzeichnissen

Da die meisten Aktionen auf Verzeichnisse keine sichtbaren Veränderungen auslösen, sind nicht alle Aktionen testbar.

11. Vorbedingungen: Codation ist aktiviert.
    Beschreibung: Ein Verzeichnis, das Dateien enthält, importieren.
    Nachbedingungen: Das Verzeichnis wurde importiert. Für alle enthaltenen Dateien wurde ein Backup angelegt.

12. Vorbedingungen: Codation ist aktiviert.
    Beschreibung: Das soeben importierte Verzeichnis löschen. Dabei wird eine Frage angezeigt, ob schreibgeschützte Dateien gelöscht werden sollen. Diese ist zu bejahen.
    Nachbedingungen: Das Verzeichnis wurde mit allen darin enthaltenen Dateien gelöscht.

D.2.2 Use-Case-Manager

Die Testfälle in diesem Abschnitt testen die Funktionalität des Use-Case-Managers. Dabei werden auch Teile des Core-Plugins und des Storage Service getestet, die von den vorigen Testfällen nicht abgedeckt sind.

Vorbereitungen

Für die Testfälle in diesem Abschnitt muss ein Testprojekt angelegt werden:

1. Ein neues Java-Projekt mit dem Namen „ucm“ anlegen.
2. Die Archivdatei „ucm.zip“ in dieses Projekt importieren. Danach sollte folgende Struktur vorhanden sein:

```
ucm (enthält 2 Dateien)
├ de
│ └ jwertenauer
│   └ geburtstagsverwaltung
│     ├ core (enthält 13 Dateien)
│     └ ui (enthält 33 Dateien)
└ lib (enthält 4 Dateien)
```

3. Codation für das Projekt aktivieren.
4. Den View „Use-Case-View“ öffnen.

Testfälle

13. Vorbedingungen: Codation ist aktiviert und Projekt geöffnet.
    Beschreibung: Im Navigator das Projekt „ucm“ auswählen. Daraufhin im View über das Kontextmenü einen neuen Anwendungsfall anlegen. Keine Eingaben vornehmen und nicht bestätigen.
    Nachbedingungen: Dialog zur Erstellung eines Anwendungsfalls wird angezeigt. Die Schaltfläche OK ist deaktiviert, da kein Name eingegeben wurde.
14. Vorbedingungen: Dialog zur Erstellung eines Anwendungsfalls wird angezeigt.
    Beschreibung: Im Dialog auf Cancel klicken.
    Nachbedingungen: Dialog ist geschlossen. Es wurde kein Anwendungsfall angelegt.
15. Vorbedingungen: Codation ist aktiviert und Projekt geöffnet.
    Beschreibung: Über das Pull-down-Menü des Views den Dialog zur Erstellung eines Anwendungsfalls aufrufen. Als Name „UC 1“ eingeben. ID und Beschreibung leer lassen. Dialog bestätigen.
    Nachbedingungen: Es wurde ein Anwendungsfall mit dem Namen „UC 1“ angelegt. Dieser wird im View angezeigt. Er enthält keine Verweise.
16. Vorbedingungen: Codation ist aktiviert und Projekt geöffnet.
    Beschreibung: Wie im vorigen Testfall einen weiteren Anwendungsfall „UC 2“ anlegen. Den Dialog jedoch über die Menüleiste öffnen.

Nachbedingungen: Es wurde ein Anwendungsfall mit dem Namen „UC 2“ angelegt. Er enthält keine Verweise. Im View werden zwei Anwendungsfälle angezeigt.

17. Vorbedingungen: Der Anwendungsfall „UC 1“ wurde angelegt.
    Beschreibung: Auf „UC 1“ doppelklicken.
    Nachbedingungen: Es öffnet sich ein Dialog zur Bearbeitung des Anwendungsfalls. Die zuvor eingegebenen Daten werden korrekt angezeigt.

18. Vorbedingungen: Der Anwendungsfall „UC 1“ wurde angelegt.
    Beschreibung: Als ID „1“ angeben, als Beschreibung „Dies ist der erste Use Case.“. Den Namen in „UC eins“ ändern. Änderungen bestätigen.
    Nachbedingungen: Der Dialog wird geschlossen und der View aktualisiert.

19. Vorbedingungen: Die Anwendungsfälle „UC eins“ und „UC 2“ wurden angelegt.
    Beschreibung: Unter Verwendung der Schaltflächen und Menüeinträge den Dialog zur Bearbeitung der beiden Anwendungsfälle öffnen und die angezeigten Daten überprüfen. Dialoge schließen.
    Nachbedingungen: Die Daten werden korrekt angezeigt:
    UC eins: ID=1, Beschreibung=Dies ist der erste Use Case.
    UC 2: ID und Beschreibung sind leer.

20. Vorbedingungen: Die Anwendungsfälle „UC eins“ und „UC 2“ wurden angelegt.
    Beschreibung: Die Datei „Person.java“ öffnen und die Methode „getFirstName“ (inkl. JavaDoc) markieren. Danach „UC eins“ markieren und im Kontextmenü den Eintrag „Add link in Java file“ anklicken. Als Name „Vorname abrufen“ eingeben. Dialog bestätigen.
    Nachbedingungen: Der View wurde aktualisiert und der Anwendungsfall enthält einen Verweis. Im Editor wird ein Marker angezeigt.

21. Vorbedingungen: Der Verweis wurde angelegt.
    Beschreibung: Auswahl im Editor ändern und im Use-Case-View auf den Verweis doppelklicken.
    Nachbedingungen: Die Methode „getFirstName“ ist markiert.

22. Vorbedingungen: Die Anwendungsfälle „UC eins“ und „UC 2“ wurden angelegt. „UC eins“ enthält einen Verweis.
    Beschreibung: Zu Anwendungsfall „UC eins“ einen Verweis auf die Methode „getSurname“ mit dem Namen „Nachname auslesen“ hinzufügen.
    Nachbedingungen: Der neue Verweis wurde hinzugefügt und der View aktualisiert.

23. Vorbedingungen: „UC eins“ enthält einen Verweis „Nachname auslesen“.
    Beschreibung: Den Verweis „Nachname auslesen“ in „Nachname abrufen“ umbenennen.
    Nachbedingungen: Der Verweis wurde umbenannt und der View aktualisiert. Der neue Name wird angezeigt.

24. Vorbedingungen: „UC 2“ enthält keinen Verweis.
    Beschreibung: Zu Anwendungsfall „UC 2“ einen Verweis auf die Methode „getSurname“ mit dem Namen „Nachname auslesen“ hinzufügen.
    Nachbedingungen: Der neue Verweis wurde hinzugefügt und der View aktualisiert.

25. Vorbedingungen: Anwendungsfälle und Verweise wurden angelegt.
    Beschreibung: Die Reihenfolge der Methoden „getSurname“ und „getBirthDay“ vertauschen. Geänderte Datei speichern.
    Nachbedingungen: Die Marker werden entsprechend verschoben. Die beiden Verweise auf die Methode „getSurname“ sind als zu prüfen markiert.

26. Vorbedingungen: Anwendungsfälle und Verweise wurden angelegt.
    Beschreibung: Eclipse neu starten und prüfen, ob die Anwendungsfälle korrekt wiederhergestellt wurden.
    Nachbedingungen: Die Anwendungsfälle und die Verweise wurden korrekt wiederhergestellt. Doppelklicks auf die Verweise markieren die annotierten Methoden.

27. Vorbedingungen: Der Verweis „Nachname auslesen“ wird als zu prüfen dargestellt.
    Beschreibung: Den Verweis „Nachname auslesen“ als geprüft markieren.
    Nachbedingungen: Die Markierung wurde von Verweis und Anwendungsfall entfernt.

28. Vorbedingungen: Anwendungsfall „UC eins“ enthält einen Verweis auf die zu löschende Methode.
    Beschreibung: Die Methode „getFirstName“ löschen und Datei speichern.
    Nachbedingungen: Der Verweis und der ihn enthaltene Anwendungsfall wird mit einem roten X markiert.

29. Vorbedingungen: Der Verweis „Vorname abrufen“ und der ihn enthaltende Anwendungsfall sind mit einem roten X markiert.
    Beschreibung: Den Verweis „Vorname abrufen“ markieren und auf die Schaltfläche zur Löschung klicken. Bestätigungsdialog bestätigen.
    Nachbedingungen: Der Verweis wurde gelöscht und der View aktualisiert. Der Anwendungsfall wird mit einem gelben Ausrufezeichen markiert.

30. Vorbedingungen: Anwendungsfall „UC 2“ existiert.
    Beschreibung: Anwendungsfall „UC 2“ markieren und im Kontextmenü auf „Remove item“ klicken.
    Nachbedingungen: Anwendungsfall „UC 2“ wurde gelöscht. Der View wurde aktualisiert.

E Inhalt der beiliegenden CD

Diesem Dokument ist eine CD mit folgendem Inhalt beigefügt:

- dieses Dokument als PDF-Datei
- der Quelltext von Codation
- ausführbare Version von Codation
- Installationsanleitung
- Projektplan als PDF-Datei
- mit Metriken erhobene Daten
- für die Laufzeitmessungen verwendete Dateien
- die in Anhang D beschriebenen Testdaten
- Version 3.2.1 des Eclipse SDKs

Weitere Informationen über Inhalt und Aufbau der CD enthält die Datei `index.html`, die sich im Wurzelverzeichnis der CD befindet.

Literaturverzeichnis

AGUIAR, Ademar: *doc-it! Supporting Agile Software Documentation with Wikis*. Projekt-Wiki. http://doc-it.fe.up.pt/. Version: August 2006, Abruf: 30.08.2006

AGUIAR, Ademar ; DAVID, Gabriel: WikiWiki weaving heterogeneous software artifacts. In: *WikiSym '05: Proceedings of the 2005 international symposium on Wikis*. New York, NY, USA : ACM Press, 2005. – ISBN 1–59593–111–2, 67–74

AGUIAR, Ademar ; DAVID, Gabriel ; BADROS, Greg J.: JavaML 2.0: enriching the markup language for Java source code. In: *Proceedings of XATA 2003, XML: Aplicações e Tecnologias Associadas*, 2004

ARTHORNE, John ; LAFFRA, Chris ; GAMMA, Erich (Hrsg.) ; NACKMAN, Lee (Hrsg.) ; WIEGAND, John (Hrsg.): *Official Eclipse 3.0 FAQs*. Addison-Wesley Professional, 2004 (the eclipse series). http://wiki.eclipse.org/index.php/Eclipse_FAQs. – ISBN 0321268385

BADROS, Greg J.: JavaML: a markup language for Java source code. In: *Proceedings of the 9th international World Wide Web conference on Computer networks : the international journal of computer and telecommunications networking*. Amsterdam, The Netherlands : North-Holland Publishing Co., 2000, 159–177

BILLE, Philip: A survey on tree edit distance and related problems. In: *Theoretical Computer Science* 337 (2005), Nr. 1-3, 217–239. DOI 10.1016/j.tcs.2004.12.030. – ISSN 0304–3975

BLOCH, Joshua: *Effective Java Programming Language Guide*. Mountain View, CA, USA : Sun Microsystems, Inc., 2001. – ISBN 0–201–31005–8

BRIGGS, Preston: Nuweb, A Simple Literate Programming Tool / Rice University. Version: 1993. http://sourceforge.net/projects/nuweb/, Abruf: 30.08.2006. Houston, TX, USA, 1993. – Forschungsbericht. – Aktuelle Version auf der Projektseite bei Sourceforge und auf http://www.literateprogramming.com/nuweb.pdf

CARD, Stuart K. ; MACKINLAY, Jock D. ; SHNEIDERMAN, Ben: *Readings in information visualization : using vision to think*. San Francisco, CA, USA : Morgan Kaufmann Publishers, 1999 (The Morgan Kaufmann series in interactive technologies). – ISBN 1–55860–533–9

CAZZOLA, Walter ; CISTERNINO, Antonio ; COLOMBO, Diego: [a]C#: C# with a customizable code annotation mechanism. In: *SAC '05: Proceedings of the 2005 ACM symposium on Applied computing*. New York, NY, USA : ACM Press, 2005. – ISBN 1–58113–964–0, 1264–1268

CHIDAMBER, Shyam R. ; KEMERER, Chris F.: A metrics suite for object oriented design. In: *IEEE Transactions on Software Engineering* 20 (1994), Juni, Nr. 6, S. 476–493. DOI 10.1109/32.295895. – ISSN 0098–5589

CLARK, James ; DEROSE, Steven: XML Path Language (XPath) / World Wide Web Consortium (W3C). Version: 1.0 vom 16. November 1999. `http://www.w3.org/TR/xpath`, Abruf: 30.08.2006. – W3C Recommendation

COLLARD, Michael L. ; MALETIC, Jonathan I. ; MARCUS, Andrian: Supporting document and data views of source code. In: *DocEng '02: Proceedings of the 2002 ACM symposium on Document engineering*. New York, NY, USA : ACM Press, 2002. – ISBN 1–58113–594–7, 34–41

DIJKSTRA, Edsger W.: A note on two problems in connexion with graphs. In: *Numerische Mathematik* 1 (1959), 269–271. `http://gdzdoc.sub.uni-goettingen.de/sub/digbib/loader?did=D196313`

Eclipse platform. `http://www.eclipse.org`, Abruf: 14.07.2006

Eclipsepedia: Development Conventions and Guidelines. `http://wiki.eclipse.org/index.php/Development_Conventions_and_Guidelines`, Abruf: 25.09.2006

FOLEY, James D. ; DAM, Andries van ; FEINER, Steven K. ; HUGHES, John F.: *Computer Graphics: Principles and Practice in C*. zweite Auflage. Addison-Wesley Professional, 1995. – ISBN 0–20184–840–6

GAMMA, Erich ; BECK, Kent: *Contributing to Eclipse: Principles, Patterns, and Plugins*. Addison-Wesley Professional, 2003. – ISBN 0–321–20575–8

GOSLING, James ; JOY, Bill ; STEELE, Guy ; BRACHA, Gilad: *The Java Language Specification, Third Edition*. Addison-Wesley Professional, 2005 `http://java.sun.com/docs/books/jls/`. – ISBN 0–321–24678–0

HEESCH, Dimitry van: *Doxygen*. Handbuch zu Version 1.4.7, 10. Juni 2006. `http://www.doxygen.org`, Abruf: 20.07.2006

JDOM. `http://www.jdom.org`, Abruf: 28.12.2006

JGrapht – a free Java Graph Library. http://jgrapht.sourceforge.net/, Abruf: 29.12.2006

KNUTH, Donald E.: Literate Programming. In: *The Computer Journal* 27 (1984), Mai, Nr. 2, S. 97–111

KNUTH, Donald E.: *Literate Programming*. Stanford, CA, USA : Center for the Study of Language and Information, 1992. – ISBN 0–9370–7380–6

KNUTH, Donald E. ; LEVY, Silvio: *The CWEB System of Structured Documentation*. 3. Reading, Massachusetts : Addison-Wesley, 2001 http://www-cs-faculty.stanford.edu/~knuth/cweb.html. – ISBN 0–201–57569–8

Leo. http://webpages.charter.net/edreamleo/front.html, Abruf: 28.08.2006

MALER, Eve ; ORCHARD, David ; DEROSE, Steven: XML Linking Language (XLink) / World Wide Web Consortium (W3C). Version: 1.0 vom 27. Juni 2001. http://www.w3.org/TR/xlink/, Abruf: 23.08.2006. – W3C Recommendation

MALETIC, Jonathan I. ; COLLARD, Michael L.: Supporting source code difference analysis. In: *ICSM '04: Proceedings of the 20th IEEE International Conference on Software Maintenance*. Washington, DC, USA : IEEE Computer Society, 11.–14. September 2004. – ISBN 0–7695–2213–0, 210–219

MALETIC, Jonathan I. ; COLLARD, Michael L. ; SIMOES, Bonita: An XML based approach to support the evolution of model-to-model traceability links. In: *TEFSE '05: Proceedings of the 3rd international workshop on Traceability in emerging forms of software engineering*. New York, NY, USA : ACM Press, 2005. – ISBN 1–59593–243–7, 67–72

MAMAS, E. ; KONTOGIANNIS, K.: Towards Portable Source Code Representations Using XML. In: *WCRE '00: Proceedings of the Seventh Working Conference on Reverse Engineering (WCRE'00)*. Washington, DC, USA : IEEE Computer Society, 2000. – ISBN 0–7695–0881–2, 172–182

MCCABE, Thomas J.: A Complexity Measure. In: *IEEE Transactions on Software Engineering* SE-2 (1976), Dezember, Nr. 4, S. 308–320. – ISSN 0098–5589

Metrics. http://metrics.sourceforge.net, Abruf: 28.12.2006

MSDN Library: *Attribute (C#-Programmierhandbuch)*. Dokumentation zu Visual Studio 2005. http://msdn2.microsoft.com/de-de/library/z0w1kczw.aspx. Version: 2006, Abruf: 30.08.2006

NØRMARK, Kurt: Elucidative Programming. In: *Nordic Journal of Computing* 7 (2000), Nr. 2, S. 87–105. – ISSN 1236–6064

NØRMARK, Kurt: An Elucidative Programming Environment for Scheme. In: *Proceedings of NWPER'2000 – Nordic Workshop on Programming Environment Research*, 2000

NØRMARK, Kurt ; ANDERSEN, Max ; CHRISTENSEN, Claus ; KUMAR, Vathanan ; STAUN-PEDERSEN, Søren ; SØRENSEN, Kristian: Elucidative Programming in Java. In: *IPCC/SIGDOC '00: Proceedings of IEEE professional communication society international professional communication conference and Proceedings of the 18th annual ACM international conference on Computer documentation*. Piscataway, NJ, USA : IEEE Educational Activities Department, 2000. – ISBN 0–7803–6431–7, 483–495

RAMSEY, Norman: Literate Programming simplified. In: *IEEE Software* 11 (1994), September, Nr. 5, 97–105. DOI 10.1109/52.311070. – ISSN 0740–7459

SnipSnap, The easy Weblog and Wiki Software. `http://snipsnap.org/`, Abruf: 30.08.2006

SourceMonitor. `http://www.campwoodsw.com`, Abruf: 28.12.2006

STALLMAN, Richard M.: EMACS the extensible, customizable self-documenting display editor. In: *Proceedings of the ACM SIGPLAN SIGOA symposium on Text manipulation*. New York, NY, USA : ACM Press, 1981. – ISBN 0–89791–050–8, 147–156

TELEA, Alexandru C.: An Open Architecture for Visual Reverse Engineering. In: KHAN, Khaled M. (Hrsg.) ; ZHANG, Yan (Hrsg.): *Managing Corporate Information Systems Evolution and Maintenance*. Hershey, PA, USA : Idea Group Pub., 2004. – ISBN 1–59140–366–9, Kapitel 9, S. 211–227

VALIENTE, Gabriel: *Algorithms on trees and graphs*. Berlin : Springer, 2002. – ISBN 3–540–43550–6

VESTDAM, Thomas: Elucidative Programming in Open Integrated Development Environments for Java. In: *PPPJ '03: Proceedings of the 2nd international conference on Principles and practice of programming in Java*. New York, NY, USA : Computer Science Press, Inc., 2003. – ISBN 0–9544145–1–9, 49–54. – siehe auch Vestdam (2004)

VESTDAM, Thomas: *Elucidative Programming – Tools, Patterns and Experiments*, Department of Computer Science, Aalborg University, Diss., 16. Januar 2004. http://www.cs.auc.dk/~odin/thesis/, Abruf: 13.07.2006

WILLIAMS, Ross N.: *FunnelWeb Reference Manual*. http://www.ross.net/funnelweb/reference/. Version: 3.2d vom 9. Januar 2000, Abruf: 25.08.2006

ZHANG, Kaizhong ; STATMAN, Rick ; SHASHA, Dennis: On the editing distance between unordered labeled trees. In: *Information Processing Letters* 42 (1992), Nr. 3, S. 133–139. DOI 10.1016/0020–0190(92)90136–J. – ISSN 0020–0190

Erklärung

Hiermit versichere ich, diese Arbeit selbstständig verfasst und nur die angegebenen Quellen benutzt zu haben.

Stuttgart, den 30. Januar 2007: ______________________

(Jochen Wertenauer)