Konzeption und prototypische Realisierung eines Integrationsframeworks für Entwicklungswerkzeuge

Masterarbeit im Fach Informatik

vorgelegt von

Iuliia Mailova

geb. 14.07.1986 in Kiew

angefertigt am

Department Informatik
Professur für Open-Source-Software
Friedrich-Alexander-Universität Erlangen–Nürnberg
(Prof. Dr. Dirk Riehle)

Betreuer: Prof. Dr. Dirk Riehle, Dr.-Ing. Martin Jung

Beginn der Arbeit: 30.10.2012
Abgabe der Arbeit: 30.04.2013

Ich versichere, dass ich die Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

Der Universität Erlangen-Nürnberg, vertreten durch die Professur für Open-Source-Software, wird für Zwecke der Forschung und Lehre ein einfaches, kostenloses, zeitlich und örtlich unbeschränktes Nutzungsrecht an den Arbeitsergebnissen der Masterarbeit einschließlich etwaiger Schutzrechte und Urheberrechte eingeräumt.

Erlangen, den 30.04.2013

Iuliia Mailova

Masterarbeit

Thema: Konzeption und prototypische Realisierung eines Integrationsframeworks für Entwicklungswerkzeuge

Hintergrund: In industriellen Entwicklungsprojekten kommt typischerweise eine Vielzahl unterschiedlicher Werkzeuge zum Einsatz, z.B. für Anforderungsmanagement, Design, Implementierung und Test. Zumeist wird in den Entwicklungsprojekten, speziell bei der Entwicklung sicherheitskritischer Systeme, eine nachvollziehbare Kette durch alle Werkzeugdaten, ein sogenanntes End-To-End Tracing, verlangt. Häufig sind die eingesetzten Werkzeuge allerdings nicht untereinander kompatibel, so dass Daten aus einem Werkzeug nicht direkt mit Daten aus einem anderen Werkzeug in Beziehung gesetzt werden können. Daher ist an den Schnittstellen zwischen den Werkzeugen ein händischer Datenabgleich notwendig, der langwierig und fehlerträchtig ist. Auch eine automatische Analyse über die Gesamtheit aller Trace-Daten wird auf diese Weise massiv erschwert.

Aufgabenstellung: In dieser Arbeit soll ein allgemeines Framework konzipiert und prototypisch umgesetzt werden, mit dessen Hilfe beliebige Entwicklungswerkzeuge miteinander im Sinne eines Tracings verbunden werden können und für das Tracing relevante Daten ausgetauscht werden können. Dabei muss ein allgemeines Datenschema für Entwicklungsartefakte und deren Beziehungen untereinander entworfen werden, in das die Entwicklungsdaten aus den jeweiligen Datenmodellen der Werkzeuge abgebildet werden können. Die dazu notwendigen Domänenmodelle der einzelnen Werkzeuge (z.B. Datenmodell für Requirements Engineering) können aus der Literatur übernommen werden. Das Framework ist anhand der Werkzeuge für Architektur (UML, Topcased), Implementierung (Java, Eclipse) und Test (JUnit) zu evaluieren.

Meilensteine:

- Einarbeiten in die Eclipse Entwicklungsumgebung, Eclipse Modeling Framework(EMF), Topcased und JUnit.
- Konzeption und Umsetzung eines allgemeinen Datenschemas für Entwicklungsartefakte und deren Beziehungen.
- Konzeption und Umsetzung des Frameworks zur Integration von Entwicklungswerkzeugen.
- Evaluierung des Frameworks.

- Schreiben der Ausarbeitung.

Literatur:

- Eclipse Project Webseite, http://www.eclipse.org.
- Topcased Project Webseite, http://www.topcased.org.
- Eclipse Modeling Framework http://www.eclipse.org/modeling/emf.
- Object Management Group: Unified Modeling Language (UML) Specification, Version 2.4.1, Aug. 2011, http://www.omg.org/spec/UML.
- JUnit Webseite, http://www.junit.org.

Betreuung:

- Prof. Dr. Dirk Riehle
- Dr.-Ing. Martin Jung

Bearbeiter:

- Iuliia Mailova

Kurzfassung

Die heutigen Software-Entwicklungsprojekte werden immer größer und der Bedarf nach einer End-To-End Traceability - der Nachverfolgbarkeit der Entwicklungsartefakte im gesamten Softwarelebenszyklus - steigt dabei rasant. Aufgrund des Einsetzen unterschiedlicher Entwicklungswerkzeuge bei verschiedenen Software-Entwicklungsprojekten wird immer wieder eine neue projektspezifische Lösung benötigt, die die End-To-End Traceability im konkreten Projekt abdeckt. Daher soll eine allgemeine Lösung entwickelt werden, die in vielen Projekten unabhängig von den eingesetzten Entwicklungswerkzeugen für die End-To-End Traceability sorgt.

In dieser Arbeit wird ein Framework konzipiert und prototypisch umgesetzt, welches beliebige Entwicklungswerkzeuge verbinden und deren Entwicklungsdaten zwecks einer End-To-End Traceability austauschen kann. Dafür wird ein Traceability-Modell entworfen, welches Entwicklungsartefakte jeglicher Art und deren Verbindungen (Tracelinks) verknüpft. Anschließend wird das Framework an einem Beispiel-Projekt evaluiert.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listingverzeichnis

Algorithmenverzeichnis

Abkürzungsverzeichnis

OMG Object Management Group
MOF MetaObject Facility
UML Unified Modeling Language
URI Uniform Resource Identifier
EMF Eclipse Modeling Framework
XML Extensible Markup Language
XMI XML Metadata Interchange
IDE Integrated Development Environment
RCP Rich Client Platform
SDK Software Development Kit
JDT Java Development Tools
PDE Plug-in Development Environment
SWT Standard Widget Toolkit
CDO Connected Data Objects
PNG Portable Network Graphics
PDF Portable Document Format

1. Einleitung

1.1. Problemstellung

Die heutigen Softwareprojekte nehmen stark an der Komplexität zu und bestehen daher aus mehreren Phasen wie Anforderungsanalyse, Design, Implementierung und Test. Für jede Phase wird typischerweise mindestens ein eigenes Werkzeug eingesetzt. Die Nachvollziehbarkeit aller Werkzeugdaten über den gesamten Entwicklungsprozess, dem sogenannten End-To-End-Traceability, nimmt in vielen Softwareprojekten immer mehr zu. Da die Werkzeuge häufig keine Schnittstellen zueinander anbieten, ist es schwierig den gesamten Entwicklungsprozess durch die entstehenden Werkzeugdaten nachzuverfolgen. Ein manueller Abgleich der Daten würde nur bei kleinen Softwareprojekten in Frage kommen, da bei großen komplexen Systemen dieses Vorgehen sehr aufwändig und fehleranfällig ist. Daher ist hier eine werkzeuggestütze Lösung notwendig.

Viele bereits existierende Tools und Ansätze decken keine End-To-End-Traceability ab [14, 24], sind nur für begrenzte und/oder spezifische Werkzeuge bzw. Entwicklungsumgebungen einsetzbar [8, 13, 6] oder haben gewisse Rahmenbedingungen, die schwer zu erfüllen sind [18].

Daher wird eine Lösung benötigt, die End-To-End-Traceability abdeckt, für beliebige verschiedene Werkzeuge in unterschiedlichen Entwicklungsumgebungen eingesetzt werden kann und keine schwer einzuhaltenden Rahmenbedingungen hat.

1.2. Zielsetzung

Ziel dieser Arbeit ist die Konzeption und prototypische Umsetzung eines allgemeinen Frameworks, welches beliebige Werkzeuge in einem Softwareprojekt verbinden und deren Daten zwecks einer End-To-End-Traceability austauschen kann. Dabei muss ein allgemeines Datenmodell für die extrahierten Werkzeugdaten und deren Beziehungen untereinander entworfen werden. Anschließend ist das Framework anhand eines konkreten Softwareprojekts zu evaluieren.

1.3. Aufbau dieser Arbeit

Im Kapitel 2 wird erklärt was Traceability ist und warum diese benötigt wird. Anschließend wird auf die Metamodellierung eingegangen. Im Kapitel 3 wird das Traceability-Modell und das Framework konzipiert. Das Kapitel 4 befasst sich mit den technischen Grundlagen, die für die Implementierung benötigt werden. Im Kapitel 5 wird die prototypische Implementierung des Konzepts aus dem Kapitel 3 vorgestellt. Das Kapitel 6 befasst sich mit der Evaluation des implementieren Frameworks an einem Beispiel-Projekt. Im Kapitel 7 werden die geleisteten Arbeiten zusammengefasst, die bestehenden Probleme offen gelegt und die Arbeit wird mit einem Ausblick abgeschloßen.

2. Grundlagen

2.1. Traceability

2.1.1. Definitionen

Traceability (Deutsch: *Nachverfolgbarkeit*):

> „Traceability ist der Grad, in dem Zusammenhänge zwischen Erzeugnissen im Entwicklungsprozess von Software festgestellt werden können.“ [5]

Einfacher ausgedrückt ist Traceability der Grad an Nachverfolgbarkeit der Projektinformationen und bildet im Grunde einen roten Faden über den gesamten Softwarelebenszyklus. So lässt sich eine Anforderung auf zugehörige Modelle, Source-Code, bis hin zum Test verfolgen. Umgekehrt kann z.B. festgestellt werden, auf welche Anforderung sich ursprünglich ein Testfall bezieht.

Traceability-Informationen (kurz *Trace-Informationen*):

> „Traceability-Informationen beschreiben Zusammenhänge zwischen Erzeugnissen im Entwicklungsprozess von Software und erhöhen somit dessen Traceability.“ [5]

Laut dieser Definition lässt sich Traceability anhand von Trace-Informationen messen.

Artefakt:

> „Ein Artefakt ist ein eindeutig identifizierbares Erzeugnis im Prozess der Softwareentwicklung von unterschiedlichster Granularität.“ [5]

Ein Artefakt kann sich in weitere Artefakte zerlegen lassen oder kann ein Teil eines anderen Artefakts sein. Ein Artefakt muss global eindeutig identifizierbar sein [5]. Beispiele für Artefakte sind Anforderungen, Modell-Elemente, Source-Code und Testfälle.

Ein *Tracelink* verbindet zwei Artefakte, wobei eines Quell- und das andere Ziel-Artefakt ist. Ein Tracelink kann implizit oder explizit vorhanden sein [6] (genaue

Definition dazu erfolgt im nächsten Kapitel) und muss ebenso wie die beiden beteiligten Artefakte eindeutig identifizierbar sein [5].
Jeder Tracelink hat eine bestimmte Bedeutung wie beispielsweise „ist abgeleitet von“, „testet“ oder „realisiert“. Tracelinks können unidirektional oder bidirektional sein. Im ersten Fall wird nur von der Quelle bis zum Ziel navigiert. Im zweiten Fall wird sowohl von der Quelle bis zum Ziel navigiert sowie umgekehrt vom Ziel bis zur Quelle [22].

Es gibt mehrere Arten von Traceability wie *Requirements Traceability* und *End-To-End Traceability*. Die Requirements Traceability bezieht sich auf die Anforderungen in einem Software-Entwicklungsprozess. Hier liegt der Fokus auf der Verfolgung einer Anforderung in einem Softwarelebenszyklus sowohl vorwärts als auch rückwärts [17]. Die End-To-End Traceability bezieht sich auf alle Phasen in einem Software-Entwicklungsprozess, von der Anforderungsphase und bis zur Testphase [8] (Die Phasen des Softwareentwicklungsprozesses werden in einem Vorgehensmodell definiert, wie z.B. das V-Modell aus der Abbildung 2.1).

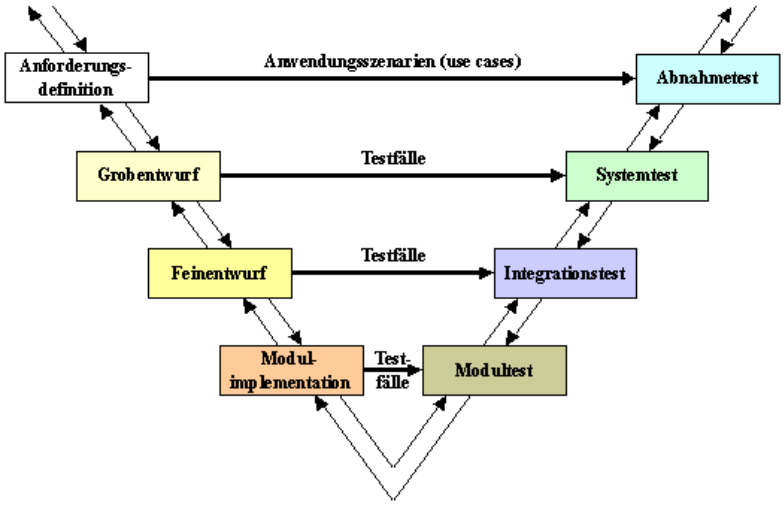


Abbildung 2.1.: V-Modell [3]

Die Traceability kann mithilfe eines Traceability-Modells dargestellt werden. Ein Traceability-Modell vermittelt die Bedeutung der Artefakte sowie deren Beziehungen zueinander [28]. Dies wird in folgendem Beispiel (siehe Abbildung 2.2) veranschaulicht.

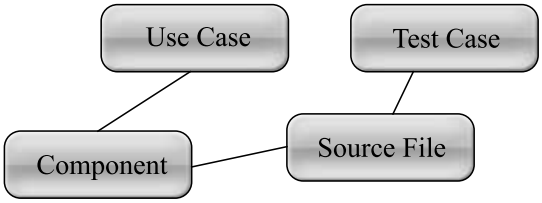


Abbildung 2.2.: Traceability-Modell (ein informelles Beispiel)

2.1.2. Arten von Traceability-Links

Die Tracelinks können sowohl zwischen Artefakten innerhalb einer Phase als auch zwischen Artefakten unterschiedlicher Phasen entstehen. Es werden in [5] zwei Arten von Links unterschieden:

- *Intra-Level-Links*: Tracelinks zwischen zwei Artefakten einer Phase.
- *Inter-Level-Links*: Tracelinks zwischen zwei Artefakten unterschiedlicher Phasen.

Die Abbildung 2.3 zeigt die unterschiedlichen Arten von Tracelinks.

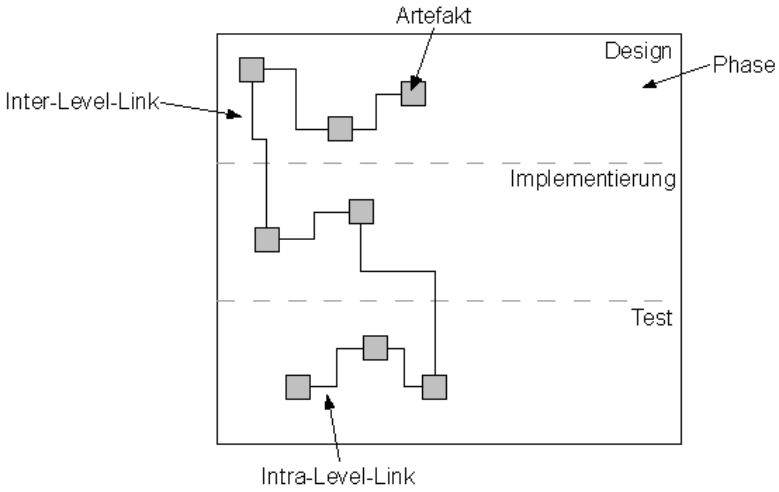


Abbildung 2.3.: Intra-Level-Links und Inter-Level-Links

Die Intra-Level-Links sind meistens innerhalb eines einzigen Werkzeugs vorhanden und werden häufig durch das Werkzeug selbst gewährleistet. Im Gegensatz dazu entstehen die Inter-Level-Links zwischen unterschiedlichen Werkzeugen und müssen explizit dokumentiert werden, da die Werkzeuge zueinander in den meisten Fällen keine Schnittstellen anbieten.

Diese Arbeit beschäftigt sich daher ausschließlich mit Inter-Level-Links und nimmt kaum Bezug auf die Intra-Level-Links.

Ein Tracelink kann explizit oder implizit vorhanden sein.
Ein expliziter Link ist ein Tracelink, welcher explizit dokumentiert wird [22, 6]. Angenommen es gibt ein Modell-Element M, welches in einer Source-Datei S realisiert wird. Der Tracelink zwischen M und S muss explizit dokumentiert werden, um diese Beziehung festzuhalten.
Ein impliziter Link wird aus mehreren expliziten Links abgeleitet [6, 22]. Angenommen es gibt eine Anforderung A, ein Modell-Element M und eine Source-Datei S. Das Modell-Element M wird aus der Anforderung A abgeleitet und die Source-Datei S basiert auf dem Modell-Element M. Daraus ergibt sich transitive Abhängigkeit zwischen der Anforderung A und der Source-Datei S. Diese transitive Beziehung zwischen Anforderung A und Source-Datei S wird nicht explizit dokumentiert, sondern aus zwei expliziten Tracelinks zwischen A und M sowie M und S erschlossen (siehe Abbildung 2.4).

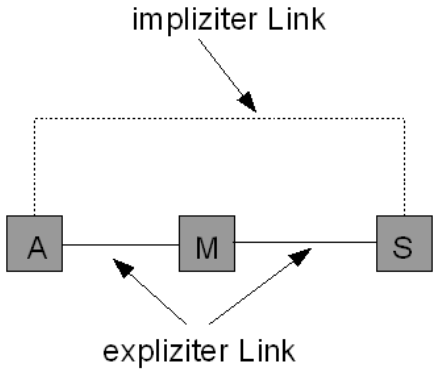


Abbildung 2.4.: explizite und implizite Links

Ergänzung

Unter impliziten Links werden nicht nur die Links verstanden, die sich aus mehreren expliziten Links ableiten lassen. Ein impliziter Link ist auch ein Link, welcher ohne zusätzlichen Aufwand bestimmbar ist. Beispielsweise zeigt die Verwendung vom gleichen *„Identifier“* für mehrere Artefakte die Abhängigkeit zwischen diesen Artefakten. Daher ist die Erstellung von expliziten Links zwischen diesen Artefakten überflüssig [19].

2.1.3. Vorteile und Nachteile von Traceability

Vorteile

> „Through traceability, each project engineer has access to contextual information that can assist them in determining where a requirement came from, its importance, how it was implemented, and how it was tested.“ [25]

Traceability hilft eine Übersicht über den gesamten Softwarelebenszyklus zu behalten. Dank ihr können beispielsweise folgende Fragen beantwortet werden:

- Wie wirkt sich die Änderung einer Anforderung aus?
- Wurden alle Anforderungen umgesetzt?
- Wurden alle Anforderungen getestet?
- Auf welche Anforderung bezieht sich ein Testfall?
- Auf welche Modelle bezieht sich eine Source-Datei?
- Welche Testfälle sind gescheitert?
- ...

Dank Traceability können auch bestimmte Metriken berechnet werden (z.B. Wie viel Prozent aller Testfälle wurden bereits erfolgreich getestet). Durch Traceability lässt sich abschätzen wie viel Zeit für die Umsetzung einer geänderten Anforderung aufgewendet werden muss. Der Nutzen von Traceability ist groß und daher in einem Software-Entwicklungsprozess nicht wegzudenken.

Nachteile

Der Hauptnachteil von Traceability ist vor allem die sehr aufwändige Erstellung und Pflege von Tracelinks, insbesondere wenn es sich um die Inter-Level-Links handelt. Hier müssen Projektinformationen aus unterschiedlichen Werkzeugen verknüpft werden [22]. Traceability bringt nur dann einen Nutzen, wenn sie fehlerfrei, vollständig und up-to-date gehalten wird [25].

2.1.4. Realisierung von Traceability

In diesem Kapitel werden sowohl manuelle als auch toolgestützte Lösungen für Traceability beschrieben und diskutiert.

Traceability kann durch eine sogenannte *Traceability Matrix* realisiert werden. Eine Traceability Matrix ist eine Tabelle die Beziehungen zwischen Artefakten meistens in Form einer Kreuztabelle darstellt [30]. Eine Traceability Matrix kann am besten in einem kleinen Projekt mit relativ stabilen Anforderungen eingesetzt werden [28]. In [11] wurde herausgefunden, dass die Anzahl von Tracelinks, die extrahiert und gepflegt werden müssen, exponentiell zur Größe und Komplizität des Projekts anwachsen. Folglich ist das manuelle Pflegen einer Traceability Matrix für große Projekte ungeeignet. Hier bedarf es eines toolgestützten Ansatzes.

Im Weiteren werden einige existierende Software-Lösungen für Traceability vorgestellt und diskutiert.

Es gibt diverse kommerzielle Requirements Management Tools, wie beispielsweise DOORS[1] und RequisitePro[2]. Mit diesen Tools können die Anforderungen verwaltet und gegenseitig verlinkt werden. Solche Tools können keine End-To-End Traceability abdecken.

Ein von der Firma Wonderware entwickeltes Tool [8] sorgt für die komplette End-To-End Traceability. Es bietet ein zentrales Repository für die Speicherung und Verwaltung der Trace-Informationen, eine Visualisierungskomponente und MS-Word Macros für die Dokument-Automatisierung. Das Tool integriert einige Werkzeuge (wie MS SharePoint), die die Firma bereits besitzt und verwendet. Außerdem kann das Repository nur bestimmte Typen von Artefakten beinhalten und verwalten. Dieses Tool ist nicht flexibel genug um es in beliebigen Projekten einzusetzen.

Es gibt auch einige Forschungsprojekte, die sich mit der Traceability beschäftigen.

Alexander Egyed hat ein Tool entwickelt [14], welches vorhandene Trace-Informationen validiert und zusätzlich neue generiert. Dazu werden die Test-Szenarios, die auf dem Ziel-System laufen, beobachtet und dabei sogenannte *footprints* ermittelt. Footprints sind Zugriffe eines Testfalls auf das zu testende System. Zusätzlich verlangt das Tool die manuelle Eingabe der Tracelinks zwischen den Modell-Elementen und Source-Dateien (*Hypothesen*). Diese stammen oft aus der Projekt-Dokumentation und müssen weder vollständig noch richtig sein. Anhand von Beobachtungen und eingegebenen Daten generiert das Tool neue Trace-Informationen und validiert die bereits existierenden Hypothesen. Auf diese Art und Weise können die fehlenden Trace-Informationen vervollständigt und vorhandene ausgebessert werden. Das Tool integriert keine externen Werkzeuge und deckt keine End-To-End Traceability ab.

[1] Vgl. http://www-142.ibm.com/software/products/us/en/ratidoor
[2] Vgl. http://www-01.ibm.com/software/awdtools/reqpro/

In einem Forschungsprojekt OPHELIA [18] wurde ein Tool (Ophelia), welches für die Abdeckung von End-To-End Traceability einsetzbar ist. Das Tool bietet einen Notifikation-Mechanismus. Sobald eine Änderung im Projekt auftritt wird eine Notifikation erzeugt und an die zuständigen Mitarbeiter versendet. Außerdem ermöglicht Ophelia durch die vielen Schnittstellen die Integration von Entwicklungs-Werkzeugen. Im Prinzip können beliebige Werkzeuge an Ophelia angebunden werden, sofern diese eine passende Schnittstelle zu Ophelia bereitstellen. Daraus ergibt sich eine Rahmenbedingung, die für den produktiven Einsatz von Ophelia eine schwer überwindbare Hürde darstellt.

Ein weiteres Eclipse-basiertes Tool wird in [6] vorgeschlagen und unterstützt Traceability in einer modellgetriebenen Softwareentwicklung [31]. Ein besonderer Fokus liegt hier auf dem hohen Automatisierungsgrad. Da der Code automatisch aus den Modellen generiert wird, lassen sich Tracelinks zwischen Modell- und Code-Elementen ohne manuellen Aufwand korrekt und vollständig erstellen. Das Tool hat eigene Modellierungsumgebung für die Erstellung der Modell-Elemente. Dort können auch die Anforderungen zu den Modell-Elementen über einen Auswahl-Dialog verlinkt werden. Jedoch werden bei diesem Tool keine Entwicklungs-Werkzeuge integriert.

2.2. Metamodellierung

Das Ziel der *Metamodellierung* ist eine Modellierungssprache mit einer bestimmten Syntax und Semantik in Form eines Modells (*Metamodells*) zu beschreiben. Gemäß dieser Modellierungssprache werden Modelle erstellt, die eine bestimmte Domäne beschreiben. Daher ist ein Modell eine Instanz seines Metamodells und wiederum ist ein Metamodell ein Modell eines Modells. Das Metamodell selbst wird ebenso durch eine (Meta-)Modellierungssprache definiert. Diese wird in einem sogenannten Meta-Metamodell beschrieben [29].

Dazu hat OMG[3] vier Metaschichten definiert. Von diesen vier Schichten ist Meta Object Facility (MOF)[4] die oberste Instanz der Metamodellierung. Die Abbildung 2.5 zeigt die Metamodell-Hierarchie der OMG.

[3]Vgl. http://www.omg.org/
[4]Vgl. http://www.omg.org/mof/

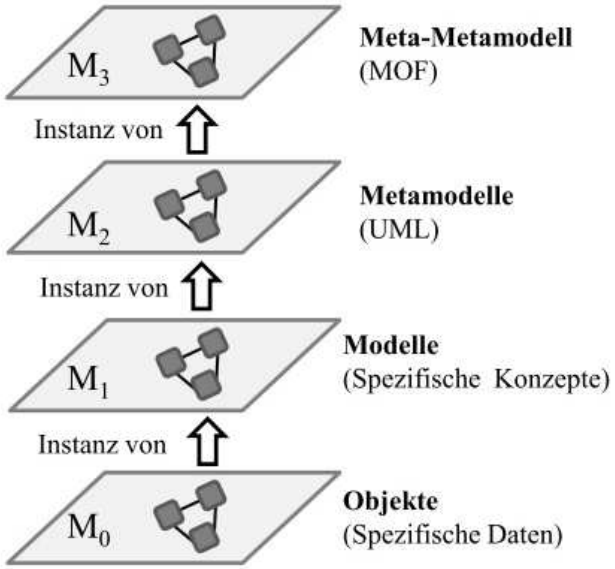


Abbildung 2.5.: Metamodell-Hierarchie der OMG [5]

M_0 ist die Ebene der *Objekte*, die Instanzen der *Modelle* der Ebene M_1 sind. Die Modelle werden durch eine Sprache beschrieben, die in der Ebene M_2 durch *Metamodelle* definiert wird. Die Metamodelle werden durch das *Meta-Metamodell* aus der Ebene M_3 beschrieben [5].

Das Beispiel für ein Metamodell ist Unified Modelling Language (UML)[5]. UML-Modelle werden durch die UML-Metamodelle [4] beschrieben, die wiederum selbst UML-Modelle sind.

UML-Profile

UML-Profile [29] sind spezielle Mechanismen der UML um das UML-Metamodell benutzerspezifisch zu erweitern. UML-Profile enthalten Elemente vom Typ *Stereotype*. Ein *Stereotype* ist eine Spezialisierung einer Metaklasse aus dem UML-Metamodell. Darüber hinaus kann ein *Stereotype* Attribute (*Tagged Values*) enthalten. Die Abbildung 2.6 zeigt die Spezialisierung (Stereotype *process* mit dem Attribut *PID*) der Metaklasse *Class*.

[5]Vgl. http://www.uml.org/

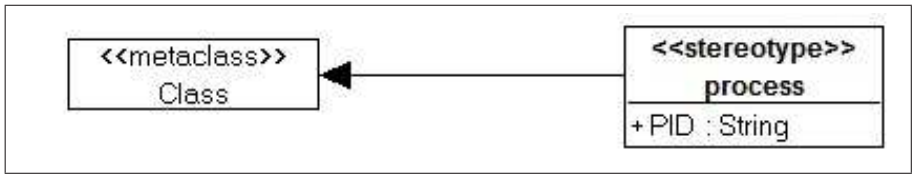


Abbildung 2.6.: UML Stereotyp mit dem Tagged Value

Sobald die Stereotypen die Bestandteile des UML-Metamodells sind, können diese wie alle anderen Elemente in einem UML-Modell verwendet werden. Nur wenn ein Stereotyp verwendet wird, kann sein Tagged Value, falls einer im UML-Profil definiert wurde, gesetzt werden. Um einen Stereotypen in einem UML-Modell verwenden zu können, muss das UML-Profil, das diesen Stereotypen definiert, auf das UML-Modell angewendet werden.

3. Konzeption

3.1. Verlinkung von Traceability-Informationen

Die Inter-Level-Links (siehe dazu Kapitel 2.1.2), also die Tracelinks zwischen den Artefakten unterschiedlicher Werkzeuge, müssen zwecks Traceability explizit dokumentiert werden. Dies kann extern oder intern erfolgen [5]. Extern bedeutet das Dokumentieren der Tracelinks außerhalb der Werkzeuge z.B. in einer Traceability Matrix (siehe Kapitel 2.1.4). Wenn das Dokumentieren intern erfolgt, dann werden die Tracelinks in den Werkzeugen selbst gehalten. Im zweiten Fall werden die Entwicklungsdaten mit zusätzlichen Informationen belastet, die zwar nicht zum Entwicklungszweck gehören, doch macht es gewissen Sinn diese zusammen mit den Artefakten zu halten: Die Erstellung und Pflege der Tracelinks wird einfacher, da dies gleich am Artefakt im selben Werkzeug erfolgt.
In dieser Arbeit erfolgt das Verlinken der Trace-Informationen intern: In Werkzeugen werden neben den Artefakten auch Tracelinks zu den Artefakten anderer Werkzeuge dokumentiert. Beispielsweise könnten bei jeder Anforderung in einem Text-Dokument URIs [9] (siehe dazu Kapitel 3.2.3) der Modell-Elemente stehen oder jedes Modell-Element in einem graphischen Editor enthält URIs zugehöriger Source-Dateien. Wie die Trace-Informationen verlinkt werden, hängt ausschließlich davon ab, welche Werkzeuge verwendet werden und welche Möglichkeiten es gibt in den Werkzeugen selbst zusätzliche Information wie URIs anderer Artefakte einzutragen. Daher kann das Verlinken der Trace-Informationen zwischen den Werkzeugen beliebig sein und wird erst im konkreten Projekt bekannt. In diesem Fall spricht man von einer sogenannten ***Tool-Kette***: eine Kette von Werkzeugen, welche anhand ihrer Artefakte zueinander in Bezug stehen. Eine mögliche Kette wäre beispielsweise *MS Word*[1] $\rightarrow$ *UML Editor* $\rightarrow$ *Eclipse*[2] $\rightarrow$ *JUnit*[3], in der MS Word Tracelinks zu den Elementen aus dem UML-Editor enthält, der wiederum Tracelinks zu Elementen aus Eclipse enthält und so weiter. Solche Ketten können beliebig sein.

Da sich das Konzept auf keine konkreten Werkzeuge stützt, können die Werkzeuge generell in folgende Gruppen eingeteilt werden:

[1] Ein Werkzeug für die Erfassung von Anforderungen
[2] Ein Werkzeug für die Erstellung von Source-Code
[3] Ein Werkzeug für die Erstellung von Tests (siehe dazu Kapitel 4.5)

- Textbasierte Werkzeuge wie MS Excel, MS Word, DOORS, etc (meistens für die Anforderungsphase).
  Möglichkeit zum Dokumentieren der Tracelinks: Hier können die Tracelinks als Freitext an den Entwicklungsdaten dokumentiert werden.
- Graphische Werkzeuge wie Topcased, Enterprice Architekt, etc (für die Designphase).
  Möglichkeit zum Dokumentieren der Tracelinks: Hier können die Tracelinks am Modell eingetragen werden. Beispielsweise als Stereotype (siehe Kapitel 2.2 Abschnitt **UML-Profile**) mittels UML-Profile.
- Programmierwerkzeuge wie Eclipse, Visual Studio, etc (für die Implementierung- und Testphase).
  Möglichkeit zum Dokumentieren der Tracelinks: Hier können Tracelinks als Kommentare im Entwicklungs-Code stehen.

Es können auch viele weitere Werkzeuge verwendet werden (z.B. für die Simulation, Konfigurationsmanagement, etc). Welche und wieviele Werkzeuge verwendet werden, hängt ausschließlichen vom konkreten Projekt ab.

3.2. Traceability-Modell

Die Entwicklungsdaten, die für Traceability relevant sind, werden im Traceability-Modell verknüpft. Das Traceability-Modell schreibt vor, welche Daten es abbildet und wie diese zueinander in Beziehung stehen. Mittels Traceability-Modell können die Trace-Informationen ausgewertet werden.

3.2.1. Anforderungen

In [10] wurden bereits Anforderungen an ein Traceability-Modell festgelegt: Universalität, Ausdrucksmächtigkeit, Offenheit und Einfachheit. Diese sind zu allgemein gehalten und können prinzipiell für jedes Traceability-Modell gelten.

Das Traceability-Modell in dieser Arbeit muss wie in [10] zwingend generisch und zugleich aussagekräftig sein. Es muss Daten aus beliebigen unterschiedlichen Datenquellen verknüpfen können ohne dass es Änderungen an der Modellstruktur bedarf. Gleichzeitig müssen die aus dem Traceability-Modell gewonnenen Informationen für die Projektmitarbeiter eine Aussagekraft besitzen.

Das Traceability-Modell muss für andere Datenquellen offen (dies wird bereits durch die Universalität gewährleistet) und möglichst einfach und übersichtlich sein.

Abgesehen von den allgemeinen Anforderungen muss das Traceability-Modell noch spezielle Kriterien, die aus der Universalität und Ausdrucksmächtigkeit hervorgehen, erfüllen:

- Das Traceability-Modell muss beliebige Typen von Artefakten und deren Beziehungen abbilden können.
- Die Navigation zwischen den Artefakten muss bidirektional erfolgen.
- Die Artefakte und Tracelinks müssen global eindeutig sein und eine Bedeutung haben.
- Das Traceability-Modell muss die Tool-Kette (siehe dazu Kapitel 3.1) abbilden können. Dies ist insbesondere wichtig, da die Tool-Kette erst im konkreten Projekt bekannt wird und daher beliebig sein kann.

3.2.2. Entwurf

In dieser Phase wird das Traceability-Modell entworfen. Da das Traceability-Modell generisch sein muss, wird es als ein Metamodell konzipiert (siehe Kapitel 2.2). Das konkrete Traceability-Modell wird durch sein Metamodell für die konkrete Entwicklungsumgebung (mit konkreten Werkzeugen) instanziiert.

Als Ausgangsbasis wurden *EAV-CR*[4] und das *UML-Metamodell* [4, 21] betrachtet.

EAV-CR ist ein Ansatz zur generischen Datenbankmodellierung. EAV-CR ist so eine Art Metamodell, welches beliebige konkrete Datenbankschemata „instanzieren“ kann. Jedoch wird dieser Ansatz in dieser Arbeit nicht weiter verfolgt, weil es auf relationale Datenbanken zwecks Persistierung eng zugeschnitten und nicht objektorientiert ist.

UML-Metamodell ist objektorientiert und dient zum Teil als Grundlage für den Entwurf des Traceability-Modells.

Ausgehend aus den Anforderungen im Kapitel 3.2.1 muss das Traceability-Modell Artefakte jeglicher Art beinhalten. Daher wird ***Arteface*** (siehe Abbildung 3.1) als eine eigene Klasse repräsentiert. Mittels des Attributs `uri` (siehe Kapitel 3.2.3) wird ein Artefakt global eindeutig. Ein Artefakt kann Tracelinks zu anderen Artefakten enthalten. Ein ***Tracelink*** (siehe Abbildung 3.1) wird jedoch, wie bei der Metamodellierung üblich ist, nicht als eine rekursive Referenz vom Artefakt zu sich selbst definiert, sondern als eine eigene Klasse (wie z.B. die Generalisierung, die im UML-Metamodell eine eigene Metaklasse ist). Über `id` wird ein Tracelink global eindeutig (`id` ist eine Konkatenation von `uri` zugehöriger Quell- und Ziel-Artefakte).

[4] Vgl. http://ycmi.med.yale.edu/nadkarni/eav_cr_contents.htm

Die Tracelinks, die ein Artefakt besitzt, unterscheiden sich in zwei Arten: Sourcelinks und Targetlinks (siehe Abbildung 3.1), die beide vom Typ *Tracelink* sind. Targetlinks sind die ausgehenden Tracelinks und die Sourcelinks sind die eingehenden Tracelinks. Diese Vorgehensweise gewährleistet die Bidirektionalität: Es kann vom Quell- zum Ziel-Artefakt und umgekehrt vom Ziel- zum Quell-Artefakt navigiert werden.

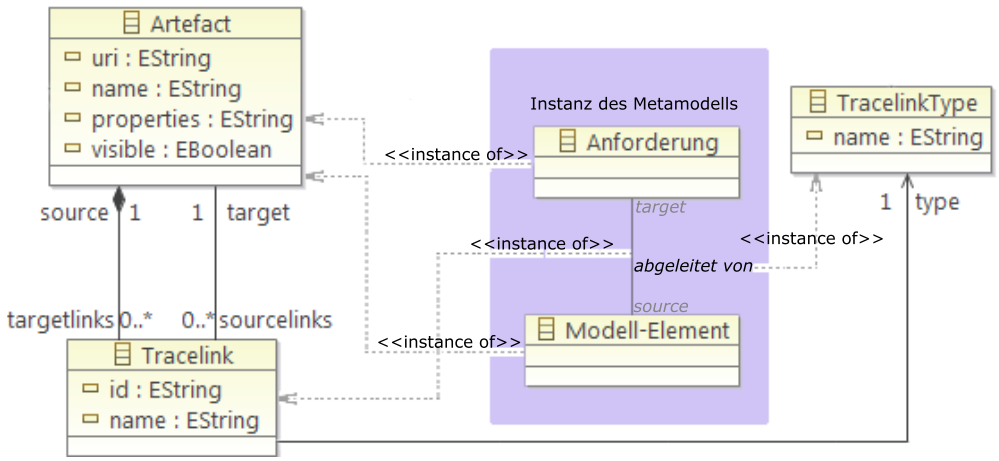


Abbildung 3.1.: *Artefact* und *Tracelink* im Traceability-Modell (mit einer beispielhaften Instanz)

Da das Traceability-Modell die Tool-Kette abbilden muss, gibt es dafür die Klasse ***Tool*** (siehe Abbildung 3.2), die ein Werkzeug repräsentiert. Jedes Tool wird fortlaufend nummeriert (`id` vom Tool). Ein ***Toollink*** (siehe Abbildung 3.2) ist eine Verbindung zwischen zwei Tools der Tool-Kette und wird ebenso wie Tracelink als eigene Klasse dargestellt. Mittels Toollinks kann festgestellt werden, wie die konkrete Tool-Kette in einem Projekt aussieht. Alle Toollinks sind ausgehende Kanten: Es kann lediglich vom Quell- zum Ziel-Tool navigiert werden. Diese Information genügt um festzustellen, wie die Tool-Kette aussieht. Über `id` (Konkatenation von `id` zugehöriger Quell- und Ziel-Tools) wird ein Toollink eindeutig.

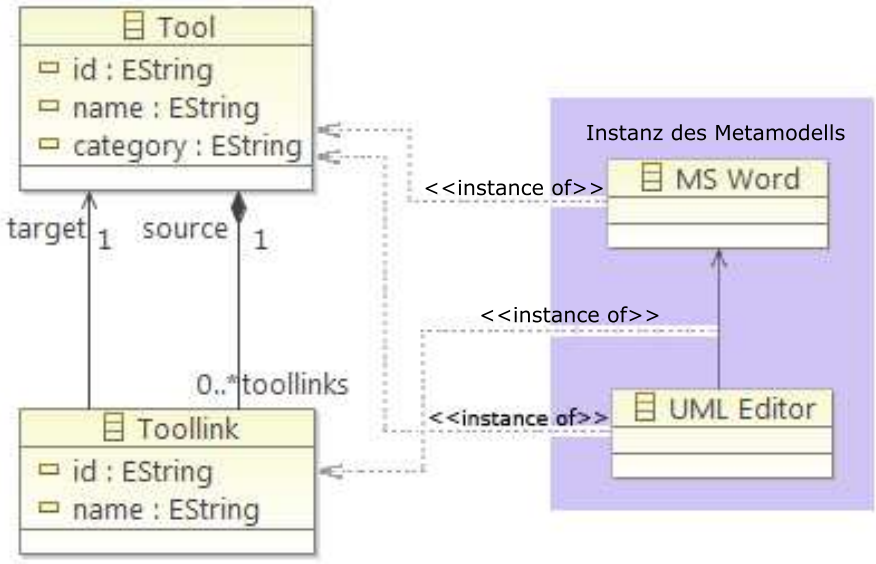


Abbildung 3.2.: *Tool* und *Toollink* im Traceability-Modell (mit einer beispielhaften Instanz)

Artefakte und Tracelinks bedürfen einer Bedeutung damit diese an Aussagekraft gewinnen. Dafür enthält ein Tracelink einen ***TracelinkType*** (siehe Abbildung 3.1), welcher als eigene Klasse dargestellt wird (analog zur Darstellung der Typen im UML-Metamodell). Mittels TracelinkType hat ein Tracelink eine ihm zugeordnete Rolle bzw. Bedeutung. Die Bedeutung eines Artefakts wird durch sein Tool bestimmt, welches dafür ein Attribut `category` hat. Wird im Projekt ein Tool mehrfach verwendet, kann dem Artefakt noch eine weitere Bedeutung mittels ***ArtefactType*** zugewiesen werden.

In [5] und [10] wurden bereits Traceability-Modelle dargestellt, die dem in dieser Arbeit beschriebenen Traceability-Modell ähnlich sind: Es gibt ebenso eine Klasse *Artefact* um Artefakte jeglicher Art abzubilden und eine Klasse *Link* für die Beziehungen zwischen den Artefakten. Was restliche Modell-Elemente angeht, sind diese Traceability-Modelle anders aufgebaut als das Traceability-Modell aus dieser Arbeit.

3.2.3. Bezeichner für Artefakte

Die Artefakte werden in einem Repository abgelegt und benötigen daher einen global eindeutigen Bezeichner. Dieser muss ein Artefakt nicht nur global identifizieren können, sondern auch einfach interpretierbar sein und möglichst viele sinnvolle Informationen beinhalten. Dies wird mit der Verwendung von `uri` vorgeschlagen.

Uniform Resource Identifier

Ein Uniform Resource Identifier (URI) ist eine kompakte Zeichenkette, die dazu dient eine Ressource eindeutig zu identifizieren.

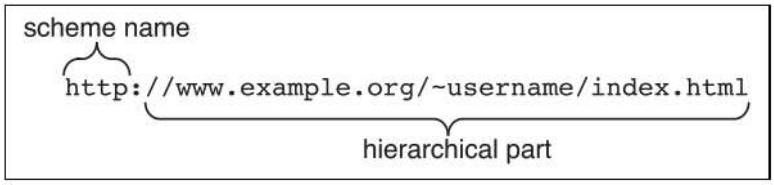


Abbildung 3.3.: Aufbau von URI [10]

Ein URI besteht aus *scheme name* und *hierarchical part*. Der *scheme name* bezeichnet den Namensraum der Ressource und der *hierarchical part* benennt die Ressource hierarchisch innerhalb des Namensraums [10]. Die Abbildung 3.3 zeigt ein beispielhaftes URI und skizziert dessen Aufbau.

URI für Artefakte

Ein Artefakt wird nach dem Benennungsschema der URIs bezeichnet. Der Aufbau vom URI für Artefakte wird größtenteils aus [10] übernommen:

artefact://<SourceDomain>/<SourceDomainType>/<HierarchicalName>

Schema 3.1: URI für Artefakte

- *SourceDomain* ist der Bereich aus dem das Artefakt stammt (z.B. uml oder code). Wenn *SourceDomain* nicht definiert ist, dann entfällt dieser Betandteil.
- *SourceDomainType* ist die konkrete Ausprägung innerhalb einer Domäne (z.B. class in uml). Dieser Bestandteil kann entfallen, falls dieser nicht definiert ist. Er muss aber entfallen, wenn *SourceDomain* entfällt.
- *HierarchicalName* besitzt die Informationen über ein Artefakt selbst (siehe Schema 3.2).

<ParentResourceName>/<ElementName>˜<uniqueID>

Schema 3.2: HierarchicalName

- *ElementName* ist der Name vom Artefakt selbst (z.B. XMLReader.java).
- *ParentResourceName* ist der Name vom Artefakt, welches *ElementName* als Verschachtlung umgibt (z.B. PackageName). Dieser Bestandteil kann entfallen, fass dieser nicht definiert ist.
- *uniqueID* ist eine künstliche Zusatz-ID, welche URI global eindeutig macht. Dieser Bestandteil kann entfallen, falls URI bereits ohne *uniqueID* global eindeutig ist.

Die *uniqueID* darf nicht willkürlich bzw. bei jedem Auslesen immer neu vergeben werden. Es wird einmal vergeben und muss unverändert bleiben, damit ein Artefakt immer global eindeutig ist. Die genaue Vergabe von URIs für Artefakte wird im Kapitel 5 gezeigt.

3.2.4. Definitionen

In diesem Kapitel wird das Traceability-Modell als ein EMF-Metamodell (siehe Kapitel 4.2) umgesetzt. Das EMF-Metamodell wird in der Abbildung 3.4 dargestellt.

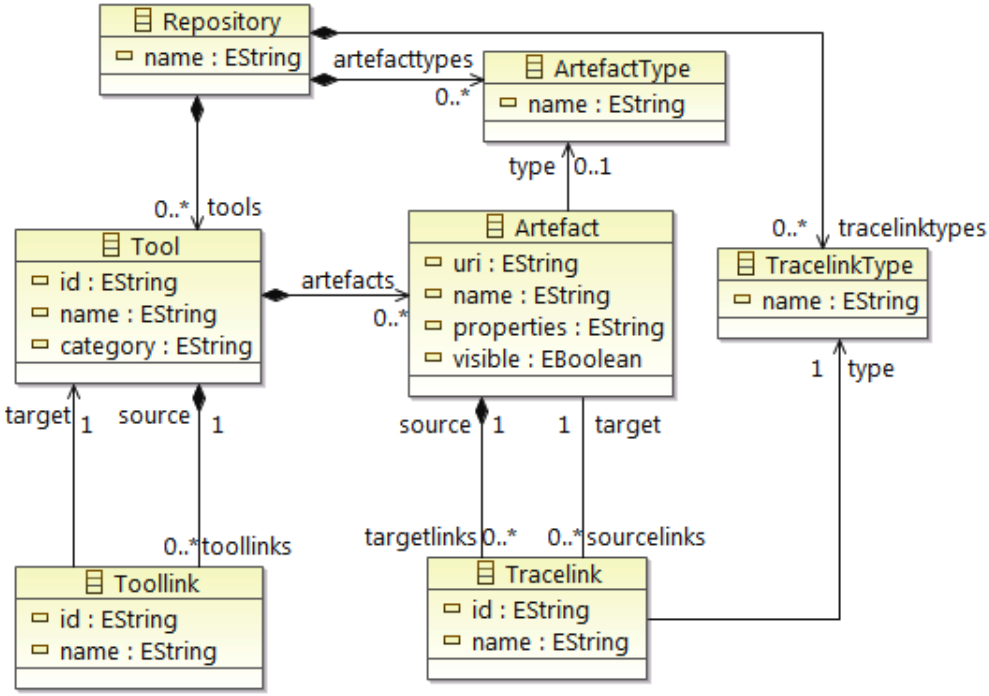


Abbildung 3.4.: Traceability-Modell

- Ein **Repository** stellt ein Wurzel-Element dar und kann beliebig viele Tools, ArtefactTypes und TracelinkTypes beinhalten. Ein *Repository* wird über `name` identifiziert.
- Ein **Tool** kann beliebig viele Artefakte und Toollinks beinhalten und wird über `id` identifiziert. Es hat außerdem noch die Attribute `name` und `category`. Das Attribut `category` dient zur Zuordnung einer Software-Entwicklungsphase zu ihrem Tool.
- Ein **Toollink** ist eine unidirektionale Beziehung zwischen zwei Tools und wird über `id` eindeutig identifiziert. Es besitzt noch das Attribut `name`.
- Ein **Artefact** kann beliebig viele Targetlinks, Sourcelinks und einen ArtefactType beinhalten und wird über `uri` eindeutig identifiziert. Außerdem hat es noch die Attribute `name`, `properties` und `visible`. Das Attribut `properties` ist ein Textfeld für zusätzliche Informationen. Mit dem Attribut `visible` kann ein Artefakt bei der späteren Analyse ausgeblendet werden.
- Ein **Tracelink** hat ein Quell- sowie Ziel-Artefakt und einen TracelinkType. Es hat die Attribute `id` für die eindeutige Identifizierung und `name`.
- Ein **TracelinkType** repräsentiert den Typ vom Tracelink und wird durch das Attribut `name` bezeichnet.
- Ein **ArtefactType** repräsentiert den Typ vom Artefakt und wird durch das Attribut `name` bezeichnet.

3.3. Framework

Nachdem das Traceability-Modell für die Artefakte und deren Beziehungen bereits definiert wurde, wird in diesem Kapitel das Framework konzipiert. Das Framework extrahiert die Daten aus den Werkzeugen und baut das Traceability-Modell auf.

3.3.1. Frameworks & Design Patterns

Ein Framework besteht aus den zusammenarbeitenden Komponenten, die eine wiederverwendbare Architektur für einen speziellen Anwendungsbereich bereitstellen [16, 20]. Ein Framework wird mittels Unterklassenbildung für eine spezielle Anwendung adaptiert. Darüber hinaus wird die Anwendung durch die Architektur des Frameworks beeinflusst [16]. Die Adaptierung des Frameworks für eine spezielle Anwendung erfolgt über die *Hot Spots*, die meist abstrakte Klassen oder Schnittstellen sind [15].

Frameworks nehmen immer mehr an Bedeutung zu, da diese *„den höchsten Grad an Wiederverwendung“* [16] der Softwaresysteme ermöglichen. Eine zentrale Rolle spielen dabei die Entwurfsmuster (Englisch: Design Patterns), die folgendermaßen definiert sind:

> „Beschreibungen zusammenarbeitender Objekte und Klassen, die maßgeschneidet sind, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen [16].“

Das Verwenden der Entwurfsmuster in einem Framework hat einen großen Vorteil, da solche Frameworks *„viel leichter einen hohen Grad an Entwurfs- und Codewiederverwendung erreichen“* [16]. Mit anderen Worten sind Frameworks, die auf Entwurfsmustern basieren, viel einfacher erweiterbar und für neue Anwendungen anpassbar.

3.3.2. Anforderungen

Das Framework muss ebenso wie das Traceability-Modell universell sein. Folglich wäre so ein Framework in jedem Projekt unabhängig von den konkreten Werkzeugen, Artefakten und der Tool-Kette einsetzbar. Der einzige Aufwand wäre dabei die konkreten Extraktoren (Parser) für das Auslesen der Artefakte zu implementieren und dem Framework bekannt zu machen, wie die konkrete Tool-Kette definiert ist. Daraus ergeben sich folgende Anforderungen:

- Das Framework muss eine Art Adapter zur Verfügung stellen, über den, das Anbinden konkreter Parser möglich ist.

- Es muss eine Möglichkeit geben in dem Framework die Tool-Kette projektspezifisch zu konfigurieren.

  Mit dieser Information kann festgestellt werden, wie die Werkzeuge anhand ihrer Artefakten zueinander in Beziehung stehen. Somit kann folgende Frage beantwortet werden: Welche Artefakte welches Werkzeugs beinhalten URIs zu den Artefakten welches anderen Werkzeugs?

- Das Framework muss das Traceability-Modell aufbauen und dieses anschließend in einem Repository (Datenbank) abbilden.

Die Anforderungen an das Framework werden in der Abbildung 3.5 skizziert.

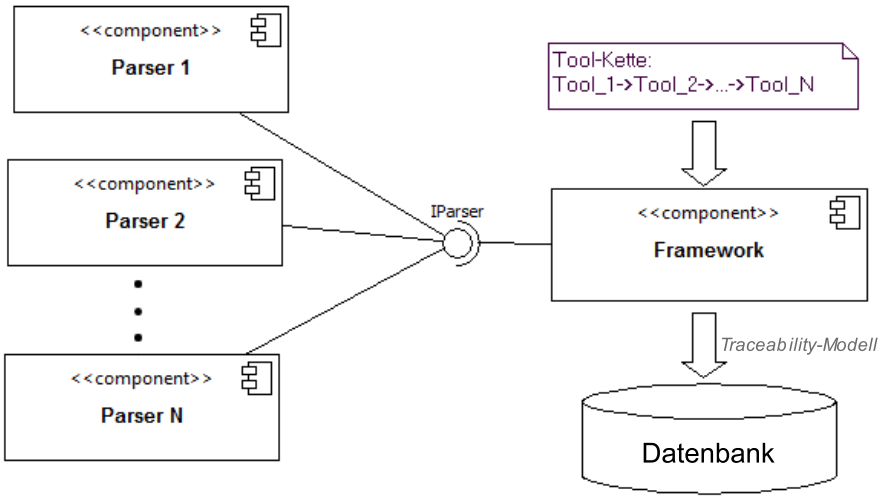


Abbildung 3.5.: **Framework:** Anforderungen

3.3.3. Entwurf

Das Framework setzt sich aus vier Komponenten zusammen (siehe Abbildung 3.6): **ParserCreator**, **TraceController**, **TraceLinker** und **DatabaseWriter**.

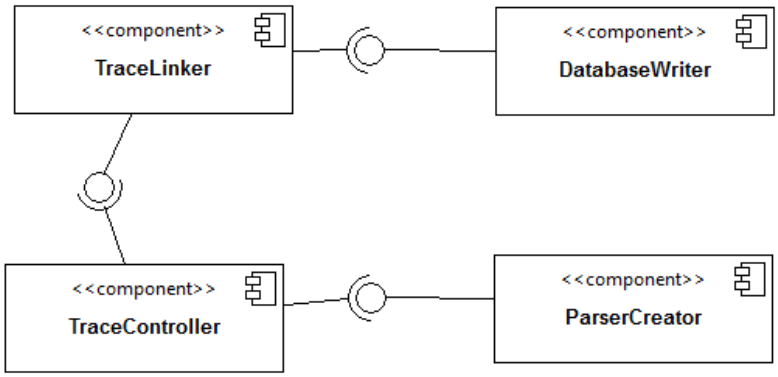


Abbildung 3.6.: **Framework:** Komponenten

Die Komponente **ParserCreator** beinhaltet einen Hot Spot, über den die konkreten Parser angebunden werden. Zugleich dient diese Komponente zur Erzeugung der konkreten Parser. Da das Framework die konkreten Parser nicht im Voraus kennt, muss die Komponente so gestaltet werden, damit das Austauschen von Parsern ohne großen Aufwand gemacht werden kann. Ausgehend von den im Kapitel 3.3.1 beschrieben Vorteilen wird diese Komponente als ein Entwurfsmuster konzipiert. Dabei kommen speziell die Erzeugungsmuster in Frage, da die Kom-

ponente **ParserCreator** für das Erzeugen der Objekt verantwortlich ist. Es gibt eine Reihe von Erzeugungsmustern [16], die für das Konzept betrachtet wurden:

- **Abstrakte Fabrik** dient zum Erzeugen von Produktfamilien verwandter Objekte. Die Produktfamilien können elegant ausgetauscht werden.
- **Erbauer** dient zum Erzeugen eines komplexen Objekts mit unterschiedlichen Repräsentationen, welche elegant ausgetauscht werden können ohne das Erzeugen des Objekts zu beeinträchtigen.
- **Fabrikmethode** dient zum Erzeugen eines Produkts, welches elegant ausgetauscht werden kann.
- **Prototyp** dient zum Erzeugen neuer Objekte durch das Kopieren (Klonen) bestehender Prototypen (Exemplare).

Abstrakte Fabrik kommt wenig in Frage, weil es nur ein Produkt *Parser* gibt und keine Produktfamilie. Das zu erzeugende Objekt *Parser* ist nicht komplex und hat immer die gleiche Repräsentation, daher ist Erbauer ungeeignet. Das Klonen bestehender Objekte *Parser* ist unnötig, weil die neuen *Parser* neu implementiert und an das Framework angebunden werden müssen. Daher ist Prototyp auch ungeeignet. Folglich wird für **ParserCreator** die Fabrikmethode eingesetzt: Es gibt nur ein zu erzeugendes Produkt *Parser*, welches einfach ausgetauscht werden kann. Die Abbildung 3.7 illustriert den Aufbau vom **ParserCreator** mit dem Fabrikmethode-Muster.

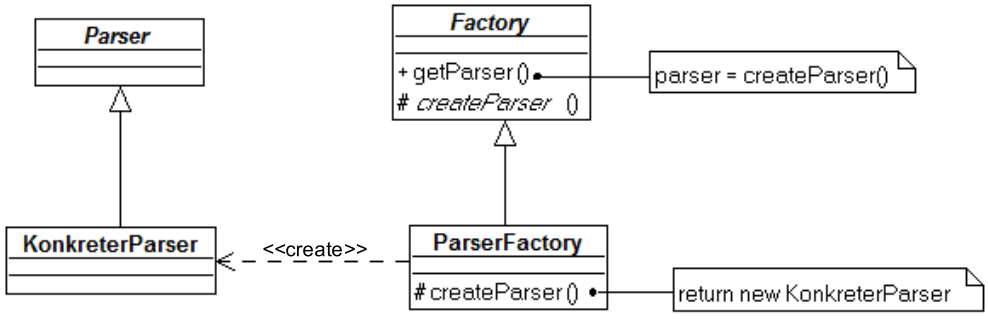


Abbildung 3.7.: **ParserCreator**: Aufbau

Ein konkreter Parser spezialisiert die abstrakte Klasse *Parser*. Die abstrakte Klasse *Factory* delegiert die Erzeugung konkreter Parser an ihre Subklasse *ParserFactory*. Das Interaktionsdiagramm in der Abbildung 3.8 zeigt die Zusammenarbeit der Klassen. Die konkreten Parser können flexibel hinzugefügt werden. Das

Hinzufügen konkreter Parser erfordert lediglich das Anpassen der Klasse *ParserFactory*, damit der neue Parser erzeugt werden kann. Die restliche Implementierung bleibt unverändert und ist komplett entkoppelt.

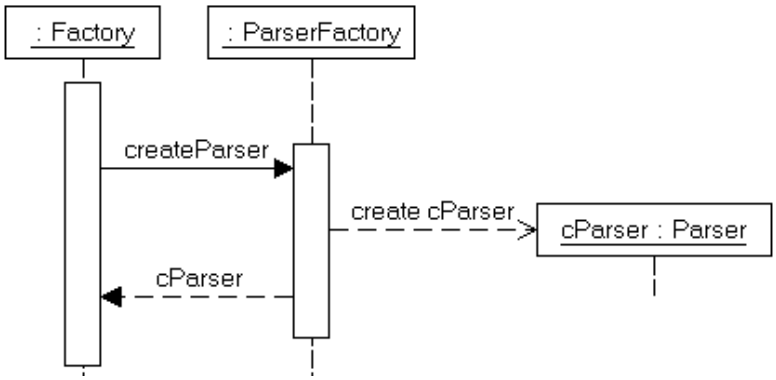


Abbildung 3.8.: **ParserCreator**: Interaktion der Klassen

Das Framework erwartet eine konkrete Tool-Kette als Eingabe. Diese wird in einer XML-Datei konfiguriert. Die Struktur dieser XML-Datei wird vorgegeben und muss lediglich manuell mit konkreten Daten befüllt werden. Das Codefragment 3.1 stellt eine solche Konfigurationsdatei dar. Die Datei beinhaltet das Wurzel-Element `<tool-chain>`, welches wiederum beliebig viele Elemente `<tool>` beinhalten kann. Das Element `<tool>` steht für ein Werkzeug, dessen Artefakte extrahiert werden müssen. Es besitzt die Attribute `name` und `path`. Das Attribut `name` ist der Name des Werkzeugs und `path` ist der Pfad zu den Quell-Dateien des Werkzeugs, die geparst werden. Weiterhin beinhaltet `<tool>` ein weiteres Element `<containedReferences>`. Dieses Element kann ein Element `<reference/>` mit dem Attribut `name` beinhalten. Das Element `<containedReferences>` lässt sich am besten anhand eines Beispiels erklären: Das Element mit dem Namen *Tool2* aus dem Codefragment 3.1 hat unter `<containedReferences>` das Werkzeug mit dem Namen *Tool1*. Das bedeutet, dass die Artefakte vom Tool2 zu den Artefakten vom Tool1 Tracelinks haben, was im Tool2 explizit dokumentiert wird. Mit anderen Worten hat das Tool2 ***„Referenzen“*** auf das Tool1. Ein Werkzeug kann beliebig viele solche *„Referenzen“* enthalten, also auch keine wie im Fall von Tool1. Wenn ein Werkzeug keine *„Referenzen“* hat, dann werden in diesem Werkzeug keine Tracelinks dokumentiert. Die Konfigurationsdatei spiegelt so die konkrete Tool-Kette wieder. Damit kennt das Framework die konkreten Werkzeuge, die Pfade zu den Quell-Dateien und die konkrete Tool-Kette.

Listing 3.1: Konfigurationsdatei: *config-file.xml*

```
<?xml version="1.0" encoding="ASCII"?>
<tool-chain>
  <tool name="Tool1" path ="pfad zu Dateien von Tool1">
    <containedReferences>
    </containedReferences>
  </tool>
  <tool name="Tool2" path="pfad zu Dateien von Tool2">
     <containedReferences>
      <reference name = "Tool1"/>
     </containedReferences>
  </tool>
  <tool name="Tool3" path="pfad zu Dateien von Tool3">
     <containedReferences>
      <reference name = "Tool2"/>
     </containedReferences>
  </tool>
  <tool name="Tool4" path="pfad zu Dateien von Tool4">
     <containedReferences>
      <reference name = "Tool2"/>
      <reference name = "Tool3"/>
     </containedReferences>
  </tool>
</tool-chain>
```

Um die geladene Konfigurationsdatei zu analysieren und die zugehörigen Parser zu erzeugen, gibt es die Komponente **TraceController**. Diese Komponente ist eine zentrale Steuerungseinheit, die den gesamten Prozess startet und kontrolliert.

Das Auslesen der Konfigurationsdatei liefert Informationen, die der Tabelle 3.1 entnommen werden können.

Werkzeug (Name)	Pfad	*„Referenzen“*
Tool1	Pfad zu Dateien von Tool1	—
Tool2	Pfad zu Dateien von Tool2	Tool1
Tool3	Pfad zu Dateien von Tool3	Tool2
Tool4	Pfad zu Dateien von Tool4	Tool2, Tool3

Tabelle 3.1.: Informationen aus der Konfigurationsdatei

TraceController übergibt an *Factory* (aus **ParserCreator**) die Namen der Werkzeuge sowie die Pfade und bekommt die konkreten Parser zurück. Das Sequenzdiagramm in der Abbildung 3.9 zeigt die Interaktion zwischen dem **TraceController** und *Factory*.

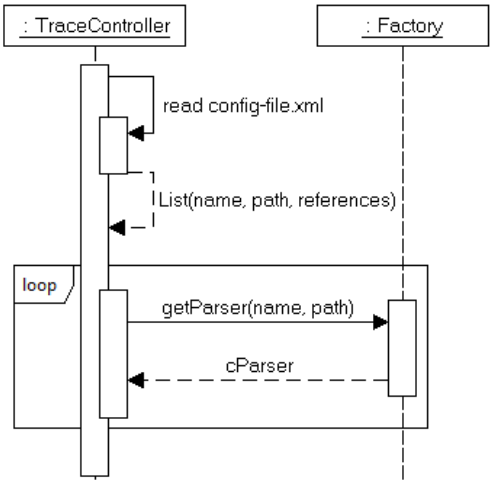


Abbildung 3.9.: Interaktion zwischen **TraceController** und *Factory*

Nun können jedem Werkzeug seine konkreten Parser zugeordnet werden:

Werkzeug (Name)	Instanzen
Tool1	Konkreter Parser1
Tool2	Konkreter Parser2
Tool3	Konkreter Parser3
Tool4	Konkreter Parser4

Tabelle 3.2.: Zuordnung konkreter Instanzen zu den Werkzeugen

Die gewonnenen Daten (siehe Tabelle 3.2) werden vom **TraceController** an die Komponente **TraceLinker** übergeben. Die Komponente **TraceLinker** kennt nun die Werkzeuge, deren *„Referenzen“* und die konkreten Parser. Anhand dieser Informationen kann **TraceLinker** weitere Informationen erschließen (siehe Tabelle 3.3).

Instanz	Instanzen von *„Referenzen“*
Konkreter Parser1	—
Konkreter Parser2	Konkreter Parser1
Konkreter Parser3	Konkreter Parser2
Konkreter Parser4	Konkreter Parser2, Konkreter Parser3

Tabelle 3.3.: Zuordnung konkreter Instanzen zu *„Referenzen“*

Nun werden jeder konkreten Instanz ihre *„Referenzen“* als Instanzen zugeordnet. Die Komponente **TraceLinker** stellt die Querbezüge zwischen den ausgelesenen

Daten zwei konkreter Parser (zwei Instanzen) her. Anschließend wird die Instanz des Traceability-Modells anhand der gewonnenen Daten erzeugt bzw. überarbeitet. Somit ist die Hauptaufgabe vom **TraceLinker** der Aufbau und Pflege der Instanz des Traceability-Modells. Das Sequenzdiagramm in der Abbildung 3.10 zeigt die Interaktion zwischen **TraceController** und **TraceLinker**.

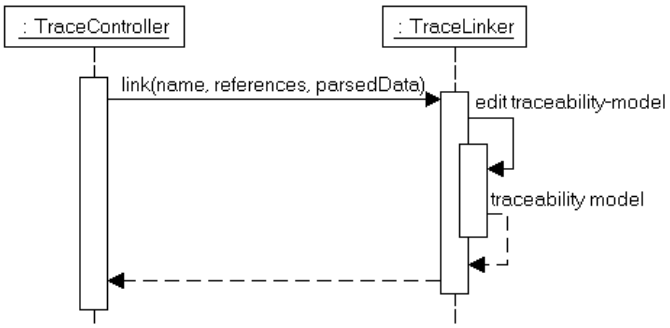


Abbildung 3.10.: Interaktion zwischen **TraceController** und **TraceLinker**

Die Komponente **DatabaseWriter** lädt die Instanz des Tracability Modells aus der Datenbank und übergibt diese an die **TraceLinker** Komponente. Nachdem **TraceLinker** die Instanz des Traceability-Modells überarbeitet hat, wird diese anschließend vom **DatabaseWriter** in die Datenbank geschrieben. Das Sequenzdiagramm in der Abbildung 3.11 illustriert die Interaktion zwischen **DatabaseWriter** und **TraceLinker**.

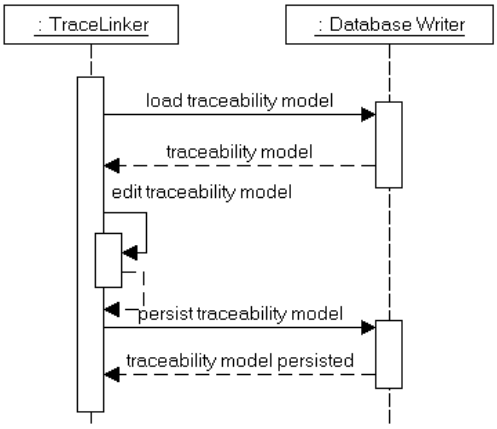


Abbildung 3.11.: Interaktion zwischen **TraceLinker** und **DatabaseWriter**

3.4. Analyse der Traceability-Informationen

Die Instanz des Traceability-Modells bildet einen sogenannten ungerichteten *Traceability-Graphen*, welcher in der Datenbank persistiert wird. Die Knoten sind dabei Artefakte und die Kanten sind die Tracelinks. Die Trace-Informationen werden durch den Traceability-Graphen repräsentiert. Diese können automatisch analysiert werden. Es hat einen großen Vorteil: Wenn der Traceability-Graph ziemlich groß ist und keine manuelle Auswertung mehr möglich ist.

Generell kann der Traceability-Graph auf der **Metamodell-Ebene** und auf der **Modell-Ebene** analysiert werden (die Implementierung dazu wird im Kapitel 5.6 besprochen). Die Analyse auf der Metamodell-Ebene bedeutet, dass es lediglich die Struktur des Graphen analysiert wird, jedoch nicht dessen Inhalt (Inhalt der Knoten und die Bedeutung der Kanten). Auf der Metamodell-Ebene werden folgende Analysen durchgeführt:

- **Entdecken verwaister Knoten.**
- **Entdecken von Zyklen.**

Unter verwaisten Knoten versteht man Knoten, die keine Kanten haben. Auf der Abbildung 3.12a ist ein verwaister Knoten (mit rot markiert) dargestellt. Solche Knoten deuten auf eine Unvollständigkeiten hin. Es kann auch halbverwaiste Knoten geben, bei welchen einige obligatorische Kanten fehlen. Auf den Abbildungen 3.12b und 3.12c sind halbverwaiste Knoten (mit rot markiert) dargestellt. Da nicht jeder halbverwaiste Knoten auf eine Unvollständigkeit hinweist (siehe die grün markierten Knoten auf den Abbildungen 3.12b und 3.12c), ist die Suche nach halbverwaisten Knoten allein auf der Metamodell-Ebene nicht möglich. Hierfür muss die konkrete Tool-Kette (die Bedeutung von Tools und Toollinks) untersucht werden.

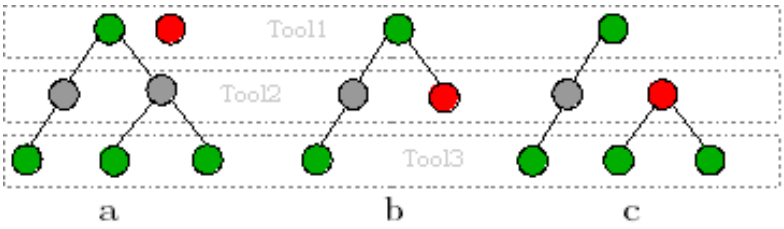


Abbildung 3.12.: Verwaiste und halbverwaiste Knoten

Angenommen die Tool-Kette sieht so aus: Tool3 → Tool2 → Tool1. In diesem Fall müssen die Artefakte vom Tool2 Tracelinks zu den Artefakten vom Tool1 und Tool3 haben. Dies wird in der Abbildung 3.12a dargestellt (Knoten vom Tool2 haben Kanten zu den Knoten vom Tool1 und Tool3). Haben die Knoten vom

Tool2 gar keine Kanten zu den Knoten vom Tool1 oder Tool3, dann sind solche Knoten halbverwaist (siehe die Abbildungen 3.12b und 3.12c).
Die Artefakte vom Tool1 müssen lediglich Tracelinks zu den Artefakten vom Tool2 haben. Daher genügt es, wenn es nur die Kanten zwischen den Knoten vom Tool1 und Tool2 gibt. Dies wird in den Abbildungen 3.12b und 3.12c dargestellt (die grün markierten Knoten vom Tool1). Das gleiche Prinzip gilt für die Artefakte vom Tool3. Diese müssen nur Tracelinks zu den Artefakten vom Tool2 haben (siehe auf den Abbildungen 3.12b und 3.12c die grün markierten Knoten vom Tool3). Die genaue Implementierung dazu wird im Kapitel 5.6 beschrieben.

Die Zyklen im Traceability-Graphen deuten vermutlich auf eine mögliche Anomalie hin, wenn über mehrere verschiedene Pfade zu demselben Knoten gelangt werden kann. Beispielsweise gibt es von einer Anforderung aus zwei separate Pfade zu demselben Testfall. Folglich müssen solche Zyklen entdeckt werden. Dabei ist jede Kante in einem ungerichteten Graphen bereits ein Zyklus (ein Zyklus zwischen Vater- und Kindknoten). Solche *lokale* Zyklen dürfen nicht berücksichtigt werden. Theoretisch könnten zwischen zwei Knoten mehrere Kanten existieren, die einen Zyklus zwischen diesen Knoten bilden. Solche Zyklen dürften genauso wie die *lokale* Zyklen nicht berücksichtigt werden. Die Existenz mehrerer Kanten zwischen zwei Knoten wird aber durch das Konzept (siehe dazu Kapitel 3.2) in dieser Arbeit ausgeschlossen.
Da der Traceability-Graph höchstwahrscheinlich nicht zusammenhängend (es gibt Artefakte, die über keinen Pfad verbunden sind), muss der Graph vorher in seine Zusammenhangskomponenten zerlegt werden. Erst dann kann in den separaten Komponenten nach Zyklen gesucht werden (siehe dazu Kapitel 5.6).

Die Analysen auf der Modell-Ebene beziehen sich auf den Inhalt der Knoten und Bedeutung der Kanten. In dieser Arbeit werden folgende Analysen auf der Modell-Ebene duchgeführt:

- **Impact Analyse**
- **Coverage Analyse**
- **Testfall Analyse**
- **Kopplung Analyse**

Die **Impact Analyse** [7] zeigt die Auswirkungen einer Änderung. Sollte sich beispielsweise eine Anforderung später ändern, würde die Impact Analyse alle von ihr abhängigen Artefakte liefern. Dank der Impact Analyse können die Auswirkungen zukünftiger Änderungen besser abgeschätzt werden.

Die **Coverage Analyse** [7] zeigt, ob ein Artefakt erfüllt wurde. Beispielsweise kann die Coverage Analyse feststellen, ob es zu einer Source-Datei mindestens einen

Testfall gibt. Dank der Coverage Analyse können Unvollständigkeiten entdeckt werden.

Die **Testfall Analyse** zeigt die Anzahl der erfolgreichen und nicht erfolgreichen Testfälle, die sich auf eine Anforderung beziehen. Dank dieser Analyse kann schnell festgestellt werden, welche Testfälle bereits erfolgreich gelaufen sind und welche noch wiederholt werden müssen.

Die **Kopplung Analyse** analysiert die Kopplung zwischen den Artefakten zweier Tools. Hier werden die Artefakte in Gruppen aufgeteilt:

- Erste Gruppe: Anzahl der Artefakte ersten Tools, die jeweils einen Tracelink zu Artefakten zweiten Tools haben.
- Zweite Gruppe: Anzahl der Artefakte ersten Tools, die jeweils zwei Tracelinks zu Artefakten zweiten Tools haben.
- Dritte Gruppe: Anzahl der Artefakte ersten Tools, die jeweils drei Tracelinks zu Artefakten zweiten Tools haben.
- Vierte Gruppe: Anzahl der Artefakte ersten Tools, die jeweils mehr als drei Tracelinks zu Artefakten zweiten Tools haben.

Mit dieser Analyse kann festgestellt werden, wenn ein Artefakt zu viele Tracelinks zu den Artefakten eines anderen Tools hat. Beispielsweise kann eine Anforderung fünf Tracelinks zu Modell-Elementen haben, was darauf hinweist, dass die Anforderung eventuell zu generell ist und besser in mehrere Anforderungen aufgeteilt werden soll.

Prinzipiell sind noch viele andere Analysen denkbar. Die Analysen in dieser Arbeit sind lediglich einige Beispiele möglicher Analysen.

Analyse Pattern & Constraints

Die Analysen auf dem Traceability-Graphen können auch durch sogenannte Analyse Pattern und Constraints weiter verfeinert werden. Es sind eine Reihe vordefinierter Regeln, anhand deren die Ergebnisse der Analyse bestimmt werden. Solche Regeln können für beliebige Analysen angewendet werden.

Wie bereits erwähnt, deuten die verwaisten Knoten auf eine Unvollständigkeit hin. Jedoch ist diese Aussage allgemein gehalten. In einigen konkreten Projekten können verwaiste Knoten erlaubt sein, wenn beispielsweise die Anforderungen hierarchisch angeordnet sind, indem es Ober- und Unterkapitel gibt. Die Unterkapitel haben Tracelinks zu Modell-Elementen und die Oberkapitel dienen lediglich dazu die Struktur der Anforderungen hierarchisch darzustellen. Daher müssen die Anforderungen, die Oberkapitel sind, keine Tracelinks haben. Diese Bedingung kann als ein Analyse Pattern spezifiziert werden. Sobald die Analyse verwaiste Knoten

findet, die diesem Pattern entsprechen, ist dies keine Unvollständigkeit. Ein Analyse Pattern dazu kann beispielsweise so aussehen: $A-\overset{*}{A}-M$. Das bedeutet: Ein Oberkapitel steht in Beziehung mit beliebig vielen Unterkapiteln, die wiederum in Beziehung mit Modell-Elementen stehen.

Ebenso wie bei verwaisten Knoten können Zyklen in einigen konkreten Projekten erlaubt sein. Es kann mittels eines Analyse Pattern spezifiziert werden, welche Zyklen erlaubt sind bzw. nicht erlaubt sind. Beispielsweise wird eine Anforderung in mehreren Modell-Elementen abgebildet, welche dann in mehreren Source-Dateien umgesetzt und anschließend in einem einzigen Testfall getestet werden. In diesem Fall kann von einer Anforderung aus über mehrere Pfade zum gleichen Testfall gelangt werden. Das Analyse Pattern dazu könnte so aussehen: A−T−A. Dieses Pattern erlaubt Zyklen zwischen einer Anforderung und einem Testfall. Es können auch Constraints spezifiziert werden, die beispielsweise einige Zyklen verbieten: !A−T−A. Das heißt Zyklen zwischen einer Anforderung und einem Test sind nicht erlaubt.

Die bisher besprochenen Analyse Pattern und Constraints beziehen sich auf die Metamodell-Ebene. Auch auf der Modell-Ebene lassen sich Analyse Pattern und Constraints definieren. Beispielsweise läss es sich mit einem Analyse Pattern bei der Coverage Analyse festlegen, dass es zu einer Anforderung mindestens zwei Testfälle geben muss: A−2T.

Die Analyse Pattern und Constraints können nur projekspezifisch bestimmt werden. Deswegen empfiehlt es sich einen projektspezifischen Katalog aus Analyse Pattern bzw. Constraints zu definieren und diese in die Analyse mit einzubetten.

4. Technische Grundlagen

4.1. Eclipse

Eclipse[1] ist ein Open-Source-Framework zur Entwicklung von Software unterschiedlicher Art. Anfangs (bis zur einschließlich Version 2.1) wurde Eclipse als eine integrierte Entwicklungsumgebung (IDE) für Java-Programmierung konzipiert. Seit Version 3.0 besteht Eclipse aus einem kleinen Kern, bekannt als *Platform Runtime*. Die volle Funktionalität von Eclipse wird durch die Plug-Ins bereitgestellt, die durch den Kern geladen werden. Somit kann Eclipse leicht erweitert werden. Das nennt sich Rich Client Platform (RCP). Sowohl der Kern als auch die Plug-Ins sind in Java implementiert [12].

Unter Eclipse wird oft Eclipse Software Development Kit (SDK) verstanden. Eclipse SDK besteht aus der Eclipse Plattform selbst, den Werkzeugen für die Java-Entwicklung (Java Development Tools (JDT)) und der Umgebung zur Entwicklung von Plug-Ins (Plug-in Developer Environment (PDE)) [2]. Die Abbildung 4.1 zeigt die Architektur von Eclipse SDK .

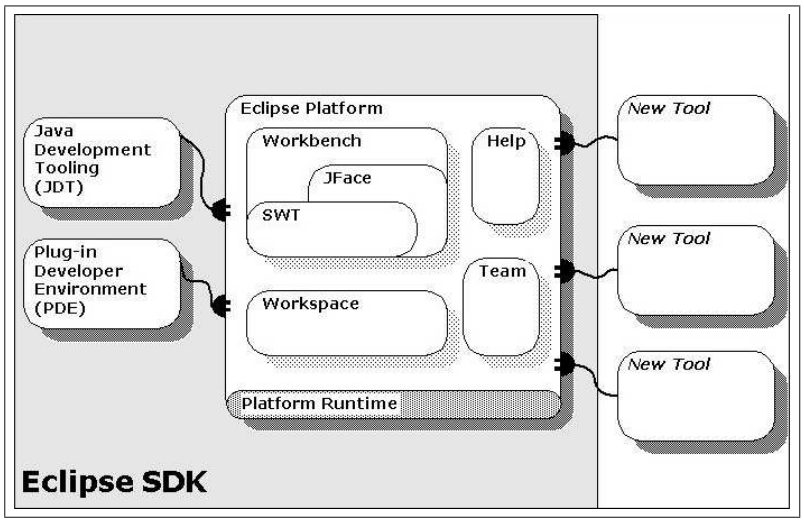


Abbildung 4.1.: Architektur von Eclipse SDK [2]

[1] Vgl. http://www.eclipse.org/

Eclipse verwendet Standard Widget Toolkit (SWT) als GUI-Framework für die graphische Oberfläche. Da SWT auf den nativen Komponenten eines Betriebssystems basiert und von Eclipse als graphische Basis verwendet wird, ist Eclipse daher plattformspezifisch [12].

In dieser Arbeit wird Eclipse Indigo 3.7.2 verwendet.

4.2. Eclipse Modeling Framework

Eclipse Modeling Framework[2] (EMF) ist ein in Eclipse Werkzeugplattform integriertes Modellierungs-Framework, welches zur Generierung von Quellcode aus strukturierten Datenmodellen verwendet wird. Das Modell, welches die Modelle in EMF repräsentiert, heißt *Ecore*. Ecore ist selbst ein EMF-Modell und somit sein eigenes Metamodell. Ein EMF-Modell kann aus anderen UML-Diagrammen, XML-Dateien oder annotierten Java-Interfaces generiert werden [26]. Mit einem zur Verfügung gestellten graphischen Editor können EMF-Modelle auch manuell erstellt werden.

Der Ecore-Ansatz ist allgemein gehalten. Damit lassen sich nicht nur spezifische Objektmodelle erzeugen, sondern auch spezifische Metamodelle (wie das Traceability-Modell aus dem Kapitel 3.2), von denen ebenso Instanzen erzeugt werden können. Deswegen könnte Ecore-Metamodell auch als Ecore-Meta-Metamodell angesehen werden [23].

Genutzte Vorteile von EMF:

- Automatische Codegenerierung.
- Bei Änderungen am Modell kann der Code neu generiert werden ohne die bereits implementierten Code-Teile zu überschreiben.
- Änderungen an der Code-Struktur, werden im Modell aktualisiert.
- Modell-Instanzen werden nicht direkt, sondern über eine Fabrik erzeugt.
- Einfache und schnelle Persistierung von Modell-Instanzen in einer XML-Datei mittels Ecore-Klassenbibliothek.
- Es wird automatisch um die Konsistenz von Assoziationen gekümmert.

Darüber hinaus gibt es noch weitere Vorteile:

- Bereitstellen von Notifikations-Mechanismen für die Modell-Änderungen.

[2] Vgl. http://www.eclipse.org/modeling/emf

- Ecore-Klassenbibliothek bietet eine mächtige reflective API für generischen Zugriff auf Modelle.

- ... und vieles mehr.

4.3. Connected Data Objects

Connected Data Objects[3] (CDO) ist ein verteiltes Model Repository für EMF-Modelle und ermöglicht das Persistieren von EMF-Modellen in allen Arten von Datenbanken. Dank CDO werden die Objekte in einer Datenbank komplett automatisch abgebildet. So bleibt der Entwicklungs-Code von spezifischen Zuordnungen zu den Datenbank-Tabellen frei [1].

Um CDO zu verwenden müssen der CDO Server und der CDO-Client eingerichtet werden. Der CDO-Server muss vorerst konfiguriert werden. Dies erfolgt mittels einer *cdo-server.xml* Konfigurationsdatei. Der CDO-Client kann mit einem graphischen Editor eingerichtet und als Client-Code programmiert werden.

4.4. Topcased

Topcased[4] (Toolkit in Open source for Critical Applications & Systems Development) ist eine Open-Source Werkzeugsammlung für die Entwicklung sicherheitskritischer Anwendungen und Systeme. Topcased wurde in die Eclipse-Entwicklungsumgebung integriert und nutzt dessen IDE. Mit Topcased können alle Phasen im Software-Entwicklungsprozess unterstützt werden.

In dieser Arbeit wird Topcased als Modellierungsumgebung für die Erstellung von UML-Diagrammen verwendet. Mit den graphischen Editoren von Topcased können viele UML-Diagramme wie beispielsweise Klassendiagramme oder Sequenzdiagramme erstellt werden. Die interne Repräsentation von UML-Diagrammen in Topcased erfolgt in einem XMI (XML Metadata Interchange)[5] Format. XMI ist ein Austauschformat zwischen den Software-Entwicklungswerkzeugen. Da XMI eine Erweiterung von XML ist, können XMI-Dateien leicht verarbeitet bzw. geparst werden (siehe dazu Kapitel 5.2.2).

[3] Vgl. http://www.eclipse.org/cdo/
[4] http://www.topcased.org/
[5] Vgl. http://www.omg.org/spec/XMI/

4.5. JUnit

JUnit[6] ist ein Framework zum Testen von Java-Programmen. Damit werden insbesondere automatische Tests von Klassen, Methoden und kleinen Modulen durchgeführt. Die Testfälle werden als Methoden innerhalb einer Klasse definiert.
JUnit nutzt eine Reihe an Annotationen und Assert-Statements. Annotationen werden über den Methoden definiert und spezifizieren damit deren Zweck. Assert-Statements werden dazu verwendet um die Tests zu evaluieren. In dem Code-Beispiel 4.1, was Multiplikation zweier Zahlen testet, gibt es eine Annotation *@Test* und ein Assert-Statement *assertEquals*. Die Annotation *@Test* bedeutet, dass die Methode *testMult()* ein Testfall ist (siehe Codefragment 4.1). Das Assert-Statement *assertEquals* prüft die eigentliche Multiplikation, indem es das erwartete Verhalten mit dem tatsächlichen vergleicht.

Listing 4.1: JUnit Test-Methode

```
1 @Test
2 public void testMult() {
3    // CalculatorClass is tested
4    Calculator calc = new Calculator();
5    // Check if 2*5 returns 10
6    assertEquals(10, calc.mult(2,5));
7     }
```

Es gibt auch eine Möglichkeit die Test-Ergebnisse in einem Protokoll (XML-Datei) abzuspeichern.

[6] Vgl. http://www.junit.org/

5. Implementierung

In diesem Kapitel wird die prototypische Implementierung des Frameworks beschrieben. Die Implementierung basiert auf konkreten Werkzeugen: MS Excel, Topcased, Eclipse und JUnit.

5.1. Allgemeine Struktur

Die Struktur der Implementierung (entsprechend dem Konzept aus dem Kapitel 3) wird in der Abbildung 5.1 dargestellt. Das gesamte Klassendiagramm des Frameworks befindet sich im Anhang A (Abbildung A.1).

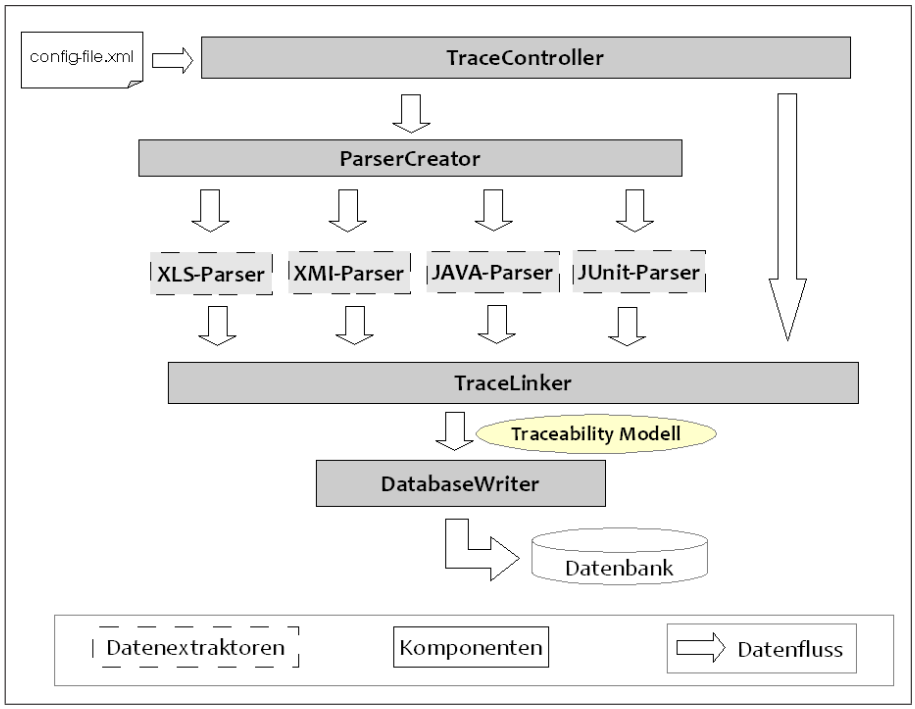


Abbildung 5.1.: Komponenten des Frameworks

Es wurden für die Implementierung vier konkrete Datenquellen gewählt: MS Excel für eine Liste von Anforderungen, Topcased für UML-Diagramme, Eclipse für Source-Dateien (*.java) und JUnit für die Testfälle.

5.2. ParserCreator

5.2.1. Aufbau

Die Komponente *ParserCreator* beinhaltet eine abstrakte Klasse *Factory*, die eine Methode `getParser(toolName, path)` zur Verfügung stellt. Dabei wird der Name des Werkzeugs und der Pfad zu dessen Quell-Dateien übergeben. Die *Factory* liefert einen konkreten Parser zurück, indem sie die Erzeugung eines konkreten Parsers an ihre Subklasse *ParserFactory* delegiert. Die Klasse *ParserFactory* entscheidet anhand vom übergebenen Namen des Werkzeugs welchen konkreten Parser sie zurück liefert. Mit dem übegebenen Pfad wird der konkrete Parser bei der Erzeugung initialisiert.

5.2.2. Konkrete Parser

Für diese Arbeit wurden vier konkrete Parser implementiert. Alle Parser leiten die abstrakte Klasse *Parser* ab. Jeder Parser legt die ausgelesenen Daten in einer eigenen Datenstruktur (einer Hilfsklasse *ExtractedData*) ab. Diese Datenstruktur (siehe Tabelle 5.1) ist für alle Parser einheitlich und ist eine eigene Klasse.

Type	Name
`Map<String,String>`	uri$Artefact
`Map<String,List<String>>`	uri$ReferencedURIs
`String`	id
`String`	tool
`String`	category
`String`	type

Tabelle 5.1.: Datenstruktur (Hilfsklasse *ExtractedData*)

Alle Artefakte und deren *URI*s werden in *uri$Artefact* abgelegt. In *uri$ReferencedURIs* werden alle *URI*s aus *uri$Artefact* und zusätzlich zu jedem *URI* eine Liste mit URIs verlinkter Artefakte abgespeichert. Durch *id* wird das Werkzeug identifiziert, *tool* beinhaltet den Namen des Werkzeugs, *category* beinhaltet den Namen der zugehörigen Entwicklungs-Phase und *type* legt die Bedeutung der Tracelinks fest, die von den Artefakten des Werkzeugs ausgehen.

XLS-Parser

Der XLS-Parser dient dazu, eine Excel-Datei zu parsen und die darin enthaltenen Anforderungen auszulesen. Die Anforderungen befinden sich in einer Excel-Tabelle,

die aus zwei Spalten besteht. Die erste Spalte beinhaltet eine Liste von Identifiern (IDs) und die zweite Spalte die Liste von den zugehörigen Anforderungen. Das Parsen von einem Excel-Dokument erfolgt mittels einer Java Excel API[1]. Damit kann auf die Excel-Tabellen und deren Zellen zugegriffen werden. Die ausgelesenen Inhalte werden in die Datenstruktur (siehe Tabelle 5.1) abgelegt. Da die Anforderungen keine URIs zu anderen Artefakten beinhalten, bleibt *uri$ReferencedURIs* leer. Die Attribute *id*, *tool*, *category* werden im XLS-Parser als Konstanten *0, „Excel“, „Requirements“* definiert. Hat die zu parsende Datenquelle keine dokumentierten URIs zu anderen Artefakten, dann bleibt *type* leer.

Vergabe von URIs erfolgt gemäß dem Schema 3.1:

artefact://xls/<ElementName>˜<uniqueID>

Schema 5.1: URI für Anforderungen in Excel

ElementName ist der Name der Anforderung, so wie sie im Excel-Dokument steht und *uniqueID* ist die zugehörige ID dieser Anforderung. Die anderen Bestandteile entfallen, da *URI* bereits global eindeutig ist. Ein Beispiel dazu gibt es im Kapitel 6.1.

XMI-Parser

Der XMI-Parser dient dazu die UML-Elemente aus den UML-Modellen in Topcased zu extrahieren. Da die UML-Modelle in einem XMI Format gespeichert werden, werden diese wie gewöhnliche XML-Dateien geparst. Extrahiert werden folgende Modell-Elemente:

uml:Class
uml:Component
uml:UseCase
uml:Package
uml:Interface

Tabelle 5.2.: UML-Elemente, die extrahiert werden

Jeder dieser Modell-Elemente steht in einer *1:N-Beziehung* zu den Anforderungen: Ein Modell-Element kann aufgrund einer oder mehrerer Anforderungen entstehen. Die Anforderungen werden mittels Stereotypen des UML-Profiles (siehe Ka-

[1] Vgl. http://jexcelapi.sourceforge.net/

pitel 2.2 Abschnitt UML-Profile) aus der Abbildung 5.2 zu den Modell-Elementen verlinkt.

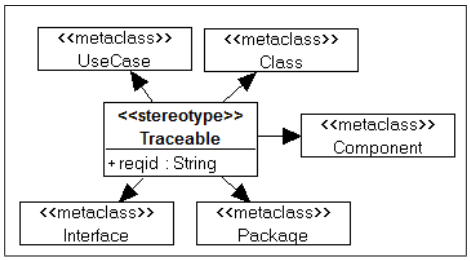


Abbildung 5.2.: UML-Profil für die Verlinkung von Anforderungen

Das UML-Profil hat einen Stereotypen *Traceable* mit dem Tagged Value *reqid*. Mit diesem Tagged Value können an einem Modell-Element die URIs zugehöriger Anforderungen separiert durch die Kommata eingetragen werden. Diese URIs werden zusammen mit den Modell-Elementen aus der XMI-Datei extrahiert. Die extrahierten Inhalte werden in der Datenstruktur aus der Tabelle 5.1 abgespeichert. Jedes Modell-Element enthält eine Liste zugehöriger Anforderungen. Diese Beziehung wird in *uri$ReferencedURIs* abgespeichert. Das Attribut *type* wird als eine Konstante *„abgeleitet von“* deklariert. Dies bedeutet, dass die Modell-Elemente aus den Anforderungen abgeleitet sind. Die Attribute *id*, *tool*, *category* werden im XMI-Parser als Konstanten *1, „Topcased“, „Design“* definiert.

Vergabe von URIs erfolgt gemäß dem Schema 3.1:

artefact://uml/<SourceDomainType>/<HierarchicalName>

Schema 5.2: URI für Modell-Elemente in Topcased

SourceDomainType ist der Name des Modell-Elements (Class, Component, etc). *HierarchicalName* wird wie folgt anhand vom Schema 3.2 aufgebaut:

<ParentResourceName>/<ElementName> ˜XMI:ID

Schema 5.3: HierarchicalName

ParentResourceName ist der Name des übergeordenen Modell-Elements (z.B. Package), *ElementName* ist der Name des ausgelesenen Modell-Elements und *xmi:id* ist die eindeutige Bezeichnung des Modell-Elements. Da ein UML-Modell in einem XMI Format gespeichert ist, besitzt jedes Modell-Element dieses UML-Modells

ein *xmi:id*, durch welches es innerhalb von diesem UML-Modell eindeutig ist. Die *xmi:id*s werden vom Topcased automatisch vergeben und bleiben beim mehrmaligen Auslesen der Modell-Elemente unverändert. Ein Beispiel dazu gibt es im Kapitel 6.1.

JAVA-Parser

Der JAVA-Parser wird zum Auslesen von Source-Code-Elementen benötingt. Die Inhalte der **.java*-Dateien werden mit einem speziellen JavaParser[2] extrahiert. Dabei wird der Ordner mit den JAVA-Dateien durchsucht und es werden zu jeder gefundenen Datei folgende Inhalte extrahiert:

Package-Name
Class-Name
Imports
Comments

Tabelle 5.3.: Source-Code-Elemente, die extrahiert werden

Als Artefakt wird nur der Klassen-Name jeder Source-Datei extrahiert, wobei auch die Methoden-Namen und sogar einzelne Code-Zeilen als Artefakte ausgelesen werden können. Für die prototypische Implementierung dieser Arbeit sind die Klassen-Namen vollkommen ausreichend. In einem echten Projekt kann die Granularität von extrahierenden Source-Code-Elementen projektspezifisch bestimmt werden.
Wozu Imports? Die Source-Dateien werden anhand von Imports herausgefiltert (z.B. der Import `org.junit.Test` ist nur bei den JUnit-Testfällen vorhanden).
Wozu Kommentare? Jede Source-Datei kann auf einem oder mehreren Modell-Elementen basieren, was eine *1:N-Beziehung* ist. Die URIs der zugehörigen Modell-Elemente werden separiert durch die Kommata in einem Kommentar im Source-Code über dem Package (siehe dazu das Codefragment 5.1) eingegeben.

Listing 5.1: Referenzierte Modell-Elemente zu einer Source-Datei

```
1 /*referencedURIs: //uml/Class/PackageName/ClassName~_DxlCAH64Ed-
      bMvOI7kjLMQ,//uml/Interface/PackageName/InterfaceName~_i-
      niQH63Ed-bMvOI7kjLMQ*/
2 package de.fau.cs.trace.example;
3 import java.io.*;
4 public class ExampleClass {
5 // put your methods here
```

[2] Vgl. http://code.google.com/p/javaparser/

```
6 }
```

Die URIs der zugehöriger Modell-Elemente werden ohne *scheme name* eingetragen. Die Klasse *ExampleClass* realisiert Modell-Elemente *ClassName* und *InterfaceName*.

JAVA-Parser befüllt die Struktur aus der Tabelle 5.1. Das Feld *type* wird als eine Konstante *„realisiert“* deklariert. Die Attribute *id*, *tool*, *category* werden im JAVA-Parser als Konstanten *2, „Eclipse“, „Coding“* definiert.

Vergabe von URIs erfolgt gemäß dem Schema 3.1:

artefact://code/<ParentResourceName>/<ElementName>

Schema 5.4: URI für Source-Code-Elemente in Eclipse

ParentResourceName ist der Name vom Package und *ElementName* ist der Klassenname der Source-Datei. Andere Bestandteile entfallen, da URI bereits global eindeutig ist. Ein Beispiel dazu gibt es im Kapitel 6.1.

JUnit-Parser

Der JUnit Parser funktioniert analog zum JAVA-Parser, da JUnit-Dateien auch JAVA-Dateien sind. Wie bereits bei dem JAVA-Parser erwähnt, werden die JUnit-Dateien anhand des Imports `org.junit.Test` als Tests erkannt. Im Gegensatz zum JAVA-Parser liest JUnit-Parser auch die Methoden-Namen innerhalb jeder Test-Klasse. Dies ist erforderlich, da jeder Testfall separat extrahiert werden muss. Ein Testfall kann anhand beliebig vieler Source-Dateien oder auch UML-Komponenten entstehen. Die URIs von zugehörigen Source-Dateien oder UML-Komponenten werden separiert durch die Kommata in einem Kommentar über dem *@Test* eingegeben:

Listing 5.2: Referenzierte Source-Datei und UML-Komponente zu einer Test-Datei

```
1    /*referencedURIs: //code/de.fau.cs.trace.example/ExampleClass,//
         uml/Component/PackageName/ComponentName~
         _5k5asFaFEeKAtYQJnEXfnQ*/
2    @Test
3    public void testCase() {
4    //write your testcase here
5    }
```

Der Testfall *testCase* ist zu einer Source-Datei *ExampleClass* und einer UML-Komponente *ComponentName* über ihre URIs verlinkt.

Außerdem liest der JUnit-Parser das Ergebnis jedes Testfalls aus dem Test-Protokoll *test-protokol.xml*.
JUnit-Parser befüllt die Struktur aus der Tabelle 5.1 analog zu anderen Parsern. Das Feld *type* wird als eine Konstante *„testet“* deklariert. Die Attribute *id*, *tool*, *category* werden im JUnit-Parser als Konstanten *3, „JUnit“, „Test“* definiert.

Vergabe von URIs erfolgt gemäß dem Schema 3.1:

artefact://test/<ParentResourceName>/<ElementName>[test-result]

Schema 5.5: URI für Test-Elemente in JUnit

ParentResourceName ist der Name vom Package plus der Klassenname der Test-Datei und *ElementName* ist der Name der Methode aus der Test-Datei. Andere Bestandteile entfallen, da URI bereits global eindeutig ist. Es wird noch das Ergebnis vom Testfall am Ende von URI als *[test-result]* eingefügt. Ein Beispiel dazu gibt es im Kapitel 6.1.

5.3. TraceController

Die Komponente *TraceController* startet und steuert den gesamten Prozess: Von der Erzeugung der konkreten Parser bis zum Persistieren der Trace-Informationen in der Datenbank.
Der *TraceController* lädt die *config-file.xml* (siehe das Codefragment 5.3), extrahiert dessen Inhalte und legt diese in einer Datenstruktur (siehe Tabelle 5.4) ab.

Type	Name
`Map<String,String>`	tool$Path
`Map<String,List<String>>`	tool$ReferencedTools

Tabelle 5.4.: Datenstruktur (Hilfsklasse *ConfigurationData*)

Die Namen der Werkzeuge und deren Pfade zu den Quell-Dateien werden in *tool$Path* abgelegt. Die Namen der Werkzeuge und die Liste dessen *„Referenzen“* - die Namen der verlinkten Werkzeuge - werden in *tool$ReferencedTools* abgespeichert.

Listing 5.3: Konfigurationsdatei mit einer konkreten Tool-Kette: *config-file.xml*

```
<?xml version="1.0" encoding="ASCII"?>
<parser-chain>
  <parser name="Excel" path ="project-files/test.xls">
    <containedReferences>
    </containedReferences>
  </parser>
  <parser name="Topcased" path="project-files/models">
     <containedReferences>
      <reference name = "Excel"/>
    </containedReferences>
  </parser>
  <parser name="Eclipse" path="project-files/src">
     <containedReferences>
      <reference name = "Topcased"/>
    </containedReferences>
  </parser>
  <parser name="JUnit" path="project-files/src">
     <containedReferences>
      <reference name = "Topcased"/>
      <reference name = "Eclipse"/>
    </containedReferences>
  </parser>
</parser-chain>
```

Nach dem Auslesen von *config-file.xml* werden konkrete Parser erzeugt: Aus *tool$Path* bekommt die Klasse *Factory* den Namen des Werkzeugs sowie den Pfad übergeben und liefert die konkreten Parser zurück, die dann auch gleich die Datenquellen auslesen. Die gewonnenen Daten werden in *tool$Artefacts* vom Typ `java.util.Map<String,ExtractedData>` abgelegt. Der Datentyp *ExtractedData* entspricht der Datenstruktur aus der Tabelle 5.1. Das heißt, der *TraceController* ordnet jedem Werkzeug eine Struktur *ExtractedData* zu, die bereits die extrahierten Daten enthält.
Die Komponente *TraceLinker* bekommt *tool$Artefacts* und *tool$ReferencedTools* vom *TraceController* übergeben.

5.4. TraceLinker

EMF-Modell

Das Traceability-Modell aus der Abbildung 3.4 wird als EMF-Modell *traceModel.ecore* umgesetzt. Der aus diesem Modell generierte Code hat eine Struktur, die in der Tabelle 5.5 dargestellt ist.

traceModel
traceModel.impl
traceModel.util

Tabelle 5.5.: Struktur vom generierten Code: *Packages*

Das Paket *traceModel* beinhaltet die Schnittstellen der Objekte und die Schnittstelle *TraceModelFactory* für die Erzeugung der Objekte. Das Paket *traceModel.impl* beinhaltet konkrete Realisierung der Schnittstellen aus *traceModel*. Die Schnittstelle *TraceModelFactory* besitzt Methoden `create[ObjektName]` für die Erzeugung der Objekte. Alle Schnittstellen der Objekte werden von der Schnittstelle `EObject` abgeleitet. Das heißt alle über *TraceModelFactory* erzeugten Objekte sind automatisch auch vom Typ `EObject`.

Besonderheit von Containment Referenzen in EMF

Die Containment Referenzen werden durch ein schwarzes Diamant-Zeichen am Assoziations-Ende gekennzeichnet. Diese Beziehung stellt in EMF einen Eltern-Kind-Zusammenhang dar. Wenn eine Containment Referenz besteht, dann ist ein Objekt der Vater des Objekt, welches referenziert wird. Dies ist insbesondere wichtig für die Persistierung des EMF-Modells. Beim Persistieren des Vater- bzw. Container-Objekts werden alle seine Kinder-Objekte automatisch innerhalb vom Vater-Objekt mitpersistiert [26]. In der Abbildung 3.4 ist *Repository* ein Container für alle *Tool*-, *ArtefactType*- und *TracelinkType*-Objekte. Wiederum ist *Tool* ein Container für *Artefact*- sowie *Toollink*-Objekte und *Artefact* ist ein Container für *Tracelink*-Objekte. Folglich werden alle Objekte des Traceability-Modells innerhalb von *Repository* persistiert.

Traceability-Modell: Laden und Aufbau

Bevor eine Instanz des Traceability-Modells erzeugt wird, wird überprüft, ob in der Datenbank eine Instanz bereits vorhanden ist. In diesem Fall wird diese Instanz geladen und keine neue erzeugt. Hier wird unter der Instanz ein *Repository*-Objekt verstanden, da dieses ein Container für alle anderen Objekte ist. Um alle Kind-Objekte von *Repository* zu bekommen, stellt die Klasse *EcoreUtil*[3] eine spezielle Methode *getAllContents* zur Verfügung. Mit den ausgelesenen Objekten wird ein *cache* vom Typ `Map<String,EObject>` aufgebaut und der Zugriff auf die Objekte erfolgt ausschließlich über den *cache*.

Die Hauptaufgabe der *TraceLinker* Komponente ist die Herstellung der Querbezüge zwischen den Artefakten verschiedener Werkzeuge und darauffolgend die Erzeugung bzw. die Überarbeitung der Instanz des Traceability-Modells. Dies erfolgt anhand von übergebenen Daten (*tool$Artefacts*, *tool$ReferencedTools*) von der Komponente *TraceController*. Die Abbildung 5.3 zeigt am Beispiel mit *Topcased* und *Excel* wie die Instanz des Traceability-Modells aufgebaut wird.

[3] Vgl. http://download.eclipse.org/modeling/emf/emf/javadoc/2.5.0/org/eclipse/emf/ecore/-util/EcoreUtil.html

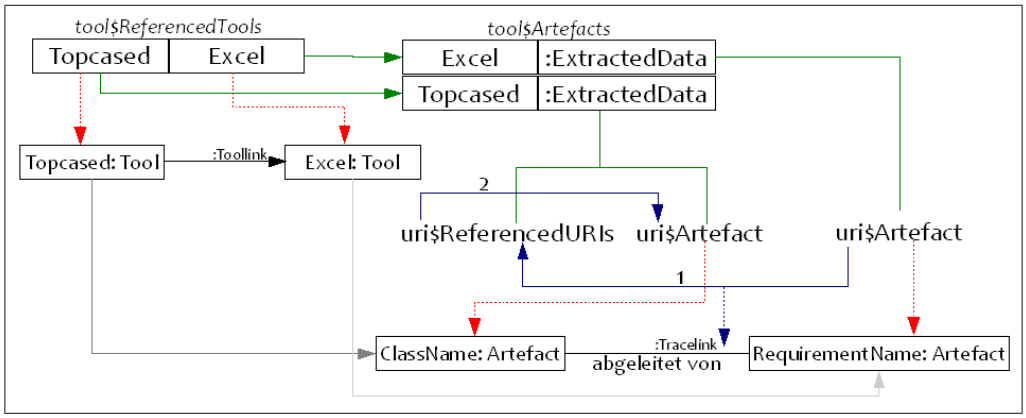


Abbildung 5.3.: Aufbau der Instanz vom Traceability-Modell (Beispiel mit Topcased und Excel)

Anhand *tool$ReferencedTools* kann festgestellt werden, welche Werkzeuge zueinander verlinkt sind (Topcased hat *„Referenzen“* auf Excel). Mittels *tool$Artefacts* können jedem Werkzeug seine Entwicklungsdaten, die sich in der Datenstruktur vom Typ `ExtractedData` befinden, zugeordnet werden. Die Datenstruktur wird in der Tabelle 5.1 dargestellt. Die *uri$ReferencedURIs* aus der Datenstruktur vom Excel bleibt leer, da Artefakte aus Excel keine URIs zu anderen Artefakten beinhalten. Hingegen beinhalten die Artefakte aus Topcased URIs zu Artefakten aus Excel, was im *uri$ReferencedURIs* gespeichert wird. Wie in der Abbildung 5.3 zu sehen ist, sucht zuerst der *TraceLinker* nach URIs aus Excel in den ReferencedURIs aus Topcased. Sobald ein Treffer gefunden wird, werden die zugehörigen Artefakte aus Topcased und Excel als Instanzen des Traceability-Modells erzeugt, falls diese nicht bereits im *cache* vorhanden sind. Diese Artefakte werden durch ein Tracelink-Objekt verbunden. Die Abbildung 6.4 zeigt partiell die Instanz des Traceability-Modells.

Alle angefragten Objekte werden in *queriedObjects* vom Typ `java.util.Set< EObject>` abgespeichert. Nachdem die Instanz des Traceability-Modells erzeugt bzw. überarbeitet wurde, werden alle nicht angefragten Objekte (Objekte, die im *cache* vorhanden sind, im *queriedObjects* aber nicht vorkommen) aus dem *Repository* gelöscht. Für das Löschen der Objekte stellt *EcoreUtil* die Methode *Delete* zur Verfügung. Diese Methode garantiert, dass keine Referenzen mehr auf das gelöschte Objekt zeigen.

5.5. DatabaseWriter

Die Komponente *DatabaseWriter* speichert die Instanz des Traceability-Modells in der H2-Datenbank[4]. Das Persistieren des Traceability-Modells erfolgt mit CDO (siehe Kapitel 4.3). Die Instanz des Traceability-Modells befindet sich innerhalb einer Ressource vom Typ `CDOResource`. Diese Ressource wird aus der Datenbank geladen und enthält das Vater-Objekt *Repository*. Dieses Vater-Objekt wird der Ressource entnommen und weiterverarbeitet. Anschließend wird das Vater-Objekt der Ressource wieder hinzugefügt und die Ressource wird in die Datenbank geschrieben. Falls die Ressource leer ist, wird ein neues *Repository*-Objekt durch *TraceModelFactory* instanziiert (siehe Abbildung 5.4).

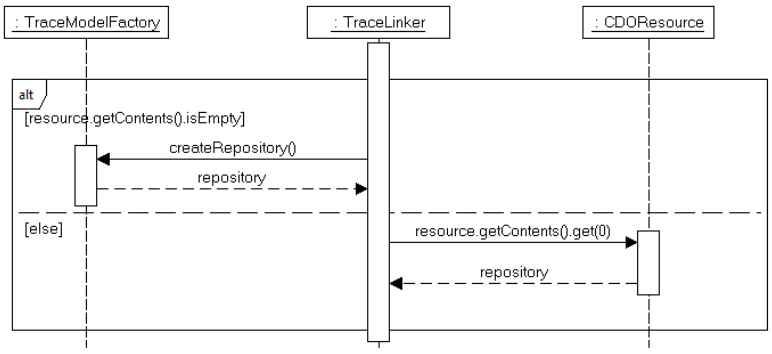


Abbildung 5.4.: Erzeugen bzw. Laden des Objekts *repository*

Der *DatabaseWriter* erledigt alle Konfigurationsaufgaben mittels CDO komplett automatisch: Das Herstellen bzw. Schließen der Verbindung zur Datenbank, das Erzeugen des Datenbankschemas, das Managen der Transaktionen, etc.

5.6. Analysen

Durch die Analysen wird der eigentliche Nutzen aus Traceability gezogen. Die Analysen können z.B. die Fragen aus dem Kapitel 2.1.3 beantworten, indem implizite Links (siehe Kapitel 2.1.2) erschlossen werden.

In diesem Kapitel wird die Implementierung von Analysen aus dem Kapitel 3.4 beschrieben. Besondere Herausforderung stellt die Tatsache dar, dass die Analysen unabhängig von einem konkreten Projekt und der spezifischen Tool-Kette funktionieren müssen.
Bevor die Analysen durchgeführt werden können, muss die Instanz des Traceability-Modells aus der Datenbank geladen werden.

[4]Vgl. www.h2database.com/

IncQuery Base

Für die Implementierung von Analysen werden bestehende Lösungen betrachtet. Eine Lösung davon ist *IncQuery Base*[5]. *IncQuery Base* ist eine kleine Java-Bibliothek und kann leicht in jedes EMF-basierte Tool integriert werden. Die Bibliothek erlaubt einige Abfragen auf dem EMF-Modell:

1. Inverse Navigation entlang der *EReferences* (Referenzen im EMF-Modell).
2. Finden aller Modell-Elemente anhand eines Attribut-Wertes.
3. Berechnen der transitiven Hülle auf dem EMF-Modell entlang gegebener Referenzen.
4. Ermitteln aller Instanzen von einem *EClass* (eine Klasse im EMF-Modell).

Für die Berechnung der transitiven Hülle auf dem EMF-Modell bietet *IncQuery Base* Methoden wie *getAllReachableTargets*, *getAllReachableSources* und *isReachable*. Die Implementierung dieser Methoden ist sehr effizient und kann auf große EMF-Modell-Instanzen angewendet werden. Beispielsweise könnte die Impact Analyse mit Hilfe von *getAllReachableTargets* durchgeführt werden, indem ausgehend von einer Anforderung alle von ihr abhängigen Elemente ermittelt werden.
Beim Evaluieren von *IncQuery Base* wurde festgestellt, dass die Berechnung der transitiven Hülle auf dem EMF-Modell, welches Containment Referenzen enthält, scheitert. Daher wurden eigene Implementierungen durchgeführt. Die anderen möglichen Abfragen von *IncQuery Base* auf dem EMF-Modell haben einwandfrei funktioniert und konnten als Unterstützung für die eigene Implementierung verwendet werden.

Eigene Implementierung

Wie bereits im Kapitel 3.4 erwähnt wurde, bildet die Instanz des Traceability-Modells einen sogenannten Traceability-Graphen. Daher werden für die Implementierung die Methoden der Graphentheorie [27] verwendet.

Für die Implementierung der **Impact Analyse** wird die *Breitensuche* angewendet. Mithilfe der Breitensuche wird ein Teil[6] des Traceability-Graphen ausgehend von einem Knoten traversiert. Dabei wurde auf die rekursive Implementierung der Breitensuche verzichtet, da bei großen Graphen der Stack viel zu sehr beansprucht wird. Viel effizienter hat sich die iterative Implementierung mit `java.util.Queue <EObject>` erwiesen. Die Ergebnisse werden in der Datenstruktur aus der Tabelle

[5] Vgl. http://incquery.net//incquery/documentation/base
[6] Der Traceability-Graph ist nicht zusammenhängend

5.6 abgespeichert. Dabei ist *artefact* der Artefakt von dem aus die Breitensuche begonnen wird und *dependency$List* ist die Liste der vom *artefact* erreichten Elemente.

Type	Name
`Artefact`	*artefact*
`List<Artefact>`	*dependency$List*

Tabelle 5.6.: Datenstruktur (Hilfsklasse *AnalysisResults*)

Die **Coverage Analyse** ermittelt mindestens einen Testfall zu einer Anforderung. Das Traversieren des Traceability-Graphen erfolgt analog zur Impact Analyse mittels der iterativen Breitensuche. Im Gegensatz zur Impact Analyse müssen nur die Test-Artefakte in die Ergebnis-Menge (Tabelle 5.6) aufgenommen werden. Dabei sind nicht alle Testfälle, die ausgehend von einer Anforderung erreicht werden, für die Coverage Analyse interessant. Als Beispiel gibt es in der Abbildung 5.5 zu der Anforderung a_1 zwei Testfälle t_1 und t_2, die diese Anforderung testen. Der Testfall t_3 ist ebenso transitiv abhängig von der Anforderung a_1, testet aber die Anforderung a_2. Daher ist in diesem Fall ausreichend, wenn der Graph nur in drei Schritten traversiert wird: Im ersten Schritt wird m_1, im zweiten s_1 sowie s_2 und im dritten letztendlich t_1 sowie t_2 erreicht. An der Stelle wird das Traversieren des Graphen abgebrochen (siehe die zwei roten Striche, die den Graphen „durchschneiden“).
Folglich wird bei der Coverage Analyse eine notwendige Anzahl an Schritten L für das Traversieren des Graphen ausgehend von einer Anforderung ermittelt.

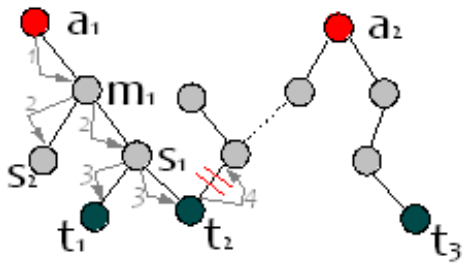


Abbildung 5.5.: Beispiel eines Traceability-Graphen (L=3)

Wie wird L bestimmt?
Da die Tool-Kette beliebig sein kann, muss L ausgehend von der konkreten Tool-Kette ausgerechnet werden. Die Art und Weise wie die Artefakte im Traceability-Graphen verbunden sind, hängt direkt davon ab, wie die Tools über die Toollinks

verknüpft werden (siehe Abbildung 5.6). Der größtmögliche Abstand (*Graphendurchmesser*[7]) zwischen zwei Tools, falls alle Tools linear[8] verbunden sind, beträgt $N-1$, wobei N die Anzahl der Tools ist. In der Abbildung 5.6 ist $N=4$ und damit ist $N-1=3$ der größtmögliche Abstand zwischen zwei Tools. Daher um alle direkten Testfälle aus der Abbildung 5.5 zu bestimmen, wird L gleich dem größtmöglichen Abstand zwischen zwei Tool gesetzt ($L=3$).
Da es von vorne an nicht bekannt ist, wie die Tools verbunden sind, kann der maximale größtmögliche Abstand zwischen zwei Tools $N-1$ betragen, da die längste mögliche Tool-Kette linear ist. Daher wird standardmäßig für jede Tool-Kette $L=N-1$ gewählt.

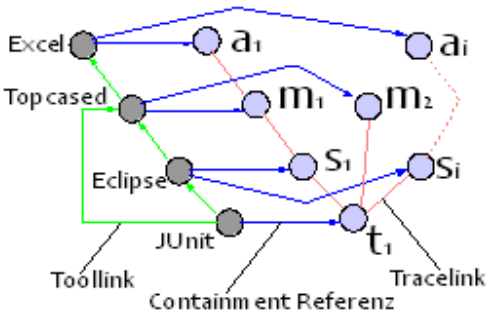


Abbildung 5.6.: Zusammenhang zwischen Tools und Artefakten (Ein Beispiel)

Einschränkung: Nur wenn die Tools linear verbunden sind (wie in den Abbildungen 5.5 und 5.6), beträgt der größtmögliche Abstand tatsächlich $N-1$. Wenn die Tools nicht linear (baumförmig, sternförmig, etc) verbunden sind, ist der größtmögliche Abstand kleiner als $N-1$, aber niemals größer. Daher können auch unerwünschte Testfälle in die Ergebnis-Menge geraten. Dies kann passieren, wenn L zu groß ist (zu viele Tools) und der tatsächlich größtmögliche Abstand viel kleiner als L ist (Tools werden z.B. sternförmig verbunden).

Theoretisch kann auch mittels Tools und Toollinks herausgefunden werden, wie groß L sein soll. Dies ist aber schwierig, da die Tool-Kette beliebig sein kann und alle möglichen Varianten vorkommen können, die nicht unbedingt vorhersehbar sind. Daher ist die oben beschriebene Variante mit dem Grapchendurchmesser zwar nicht optimal, aber dennoch sicherer um die Coverage Analyse universell zu gestalten.

Die **Test Analyse** wird nach demselben Prinzip wie Coverage Analyse durchgeführt, wobei die Ergebnis-Menge (Tabelle 5.6) anschließend in zwei Teil-Mengen

[7] Vgl. http://www.staff.hs-mittweida.de/ rjentsch/70durchm.html
[8] Der Weg vom ersten zum letzten Tool, der alle bereits besuchten Kanten genau einmal enthält (dabei wird die Richtung der Toollinks ignoriert)

aufgeteilt wird. Eine Menge erhält Testfälle, die erfolgreich sind und andere die Testfälle, die durchgefallen sind.

Die **Kopplung Analyse** wird für jeden Trace-Typ (im konkreten Fall: Excel-Topcased, Topcased-Eclipse, JUnit-Eclipse und JUnit-Topcased) bestimmt.
Zuerst wird über die Toollinks iteriert und vom jedem Toollink wird Source- und Target-Tool bestimmt. Danach wird über die Artefakte des Source-Tools iteriert und für jeden Artefakt einzeln berechnet, wie viele ausgehende Tracelinks dieses Artefakt zu den Artefakten aus dem Target-Tool hat. Je nach Ergebnis bei jedem einzelnen Artefakt wird die entsprechende globale Variable (gemäß den Gruppen aus dem Kapitel 3.4) für jeden Trace-Type hochgezählt.

Der Traceability-Graph beinhaltet **Zyklen**, wenn es zu einem Knoten mehr als nur einen möglichen Weg von einem anderen Knoten gibt (siehe Abbildung 5.7).

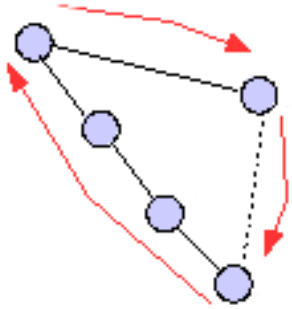


Abbildung 5.7.: Ein Zyklus im Traceability-Graphen

Der Traceability-Graph ist nicht zusammenhängend und muss vorerst in seine Zusammenhangskomponenten[9] zerlegt werden. Dies wird mit einer iterativen Tiefensuche mit Hilfe von `java.util.Stack<EObject>` durchgeführt:

INPUT: Graph G.
Setze die Markierung $c = 0$ und markiere alle Knoten v aus G mit c.
while Es gibt Knoten aus G mit Markierung 0 **do**
  Setze $c = c + 1$, wähle einen Startknoten w mit Markierung 0 und starte die Tiefensuche mit w.
  Ermittele mit der Tiefensuche alle von w erreichten Knoten und markiere diese mit c.
end while
OUTPUT: Alle Knoten sind markiert mit der Nummer ihrer Zusammenhangskomponente.

Algorithm 5.1: Zerlegen des Graphen in seine Zusammenhangskomponenten

[9]Vgl. http://www.inf.fh-flensburg.de/lang/algorithmen/graph/connected-components.htm

Das Suchen nach Zyklen kann bereits während der Tiefensuche durchgeführt werden. Wenn die Tiefensuche jedes mal mit dem neuen Startknoten w gestartet wird, wird eine neue Zusammenhangskomponente erkundet. Dabei werden alle gefundenen Knoten in *visited* vom Typ `java.util.Set<EObject>` abgelegt. Ist ein gefundener Knoten in *visited* bereits vorhanden und sein Vorgänger ist dessen Kind-Knoten, dann handelt es sich um die Kante, die in einem ungerichteten Graphen zwei mal gefunden wird: Einmal vom Vater- zum Kind-Knoten und umgekehrt vom Kind- zum Vater-Knoten. Dies ist kein Zyklus und wird daher ignoriert. Ansonsten, wenn ein Knoten gefunden wird und bereits in *visited* ist, ohne dass es sich um die nochmal gefundene Vater-Kind-Kante handelt, dann gibt es einen Zyklus innerhalb der Zusammenhangskomponente.
Einschränkung: Der Algorithmus macht lediglich einen Test[10], ob der Graph azyklisch ist. Das heißt es wird nur der Knoten ausgegeben, in dem der Zyklus gefunden wurde und zusätzlich werden alle Aktionen auf dem Stack geloggt: Ausgabe der besuchten Knoten inklusive der Nummer ihrer Zusammenhangskomponente.

Die **verwaisten Knoten** lassen sich leicht finden, indem bei jedem Artefakt im Traceability-Graphen überprüft wird, ob er keine Targetlinks und keine Sourcelinks (siehe Kapitel 3.2.4) hat. Mit den halbverwaisten Knoten sieht das Ganze etwas komplizierter aus: Nicht immer müssen Artefakte eingehende (Sourcelinks) und ausgehende (Targetlinks) Kanten haben. Dies läßt sich anhand von Tools feststellen: Hat ein Tool A keine *„Referenzen“* auf ein anderes Tool bzw. andere Tools, wird jedoch selbst von einem anderen Tool B *„referenziert“*, dann haben alle Artefakte von A nur Sourcelinks zu Artefakten vom Tool B. Hat ein Tool C *„Referenzen“* auf ein anderes Tool D bzw. andere Tools D und E , wird aber selbst von keinem Tool *„referenziert“*, dann haben alle Artefakte von C nur Targetlinks zu den Artefakten von D bzw. D und E. Hat ein Tool F *„Referenzen“* und wird selbst *„referenziert“*, dann haben seine Artefakte sowohl Targetlinks als auch Sourcelinks. Dies wird in einer Tabelle (siehe Tabelle 5.7) festgehalten:

[10] Vgl. http://alda.iwr.uni-heidelberg.de/index.php/Graphen_und_Graphenalgorithmen#-Test.2C_ob_ein_ungerichteter_Graph_azyklisch_ist

Tool	Muster
Excel	S
Topcased	TS
Eclipse	TS
JUnit	T

Tabelle 5.7.: Muster zum Evaluieren von Artefakten

Jedes Tool hat eins ihm zugeordnetes Muster: S bedeutet, dass die Artefakte von Excel nur Sourcelinks zu anderen Artefakten haben müssen. TS heißt, dass Artefakte von Topcased und Eclipse sowohl Targetlinks als auch Sourcelinks zu anderen Artefakten haben müssen. Anschließend bedeutet T, dass die Artefakte von JUnit nur Targetlinks zu anderen Artefakten haben müssen.

Die Inhalte der Tabelle 5.7 werden in einem *pattern* vom Typ `java.util.Map <EObject,String>` abgelegt. Das Muster für das Evaluieren der Artefakte wird folgenderweise aufgebaut:

INPUT: Graph G.
Extrahiere alle Tools aus G in *tools*.
Extrahiere alle Toollinks aus G in *toollinks*.
for tool : *tools* **do**
  for toollink : *toollinks* **do**
    if toollink.getSource().equals(tool) **then**
      isSource = true;
    end if
    if toollink.getTarget().equals(tool) **then**
      isTarget = true;
    end if
  end for
  if isTarget & isSource **then**
    pattern.put(tool, TS);
  end if
  if isTarget bzw. isSource **then**
    pattern.put(tool, T) bzw. pattern.put(tool, S);
  end if
end for
OUTPUT: *pattern*.

Algorithm 5.2: Aufbauen von *pattern*

Anhand von *pattern* können nun die Artefakte evaluiert werden:

INPUT: Graph G, *pattern*.
Extrahiere alle Artefakte aus G in *artefacts*.
for artefact : *artefacts* **do**
  tool = artefact.eContainer() *#EObject.eContainer() liefert dem EObject sein Vater-Objekt: Ein Tool-Objekt eines Artefact-Objekts*
  Berechne t_size (Anzahl an Targetlinks) und s_size (Anzahl an Sourcelinks) von artefact
  if t_size = 0 & s_size = 0 **then**
    PRINT: Artefakt [Name] ist verwaist.
  end if
  if t_size = 0 & $s\_size \neq 0$ **then**
    if $pattern.get(tool) \neq S$ **then**
      PRINT: $Artefakt[Name]$ *hat keine* $Targetlinks \rightarrow$ *halbverwaist*
    end if
  end if
  if s_size = 0 & $t\_size \neq 0$ **then**
    if $pattern.get(tool) \neq T$ **then**
      PRINT: $Artefakt[Name]$ *hat keine* $Sourcelinks \rightarrow$ *halbverwaist*
    end if
  end if
end for

Algorithm 5.3: Evaluieren von Artefakten

Einschränkung:

In die vorgestellen Analysen wurden keine Analyse Pattern bzw. Constraints eingebettet. Daher sind diese erstmals allgemeingültig.

6. Evaluation

In diesem Kapitel wird das Framework anhand eines konkreten Beispiel-Projekts evaluiert. Die daraus gewonnenen Ergebnisse werden anschließend diskutiert.

6.1. Beispiel-Projekt

Bei dem Beispiel-Projekt handelt es sich um ein Bibliotheksverwaltungssystem, welches wie die Implementierung aus dem Kapitel 5 auf den vier Werkzeugen MS Excel, Topcased, Eclipse und JUnit basiert.
Die Tool-Kette ist in der Abbildung 6.1 dargestellt und wird im *config-file.xml* (siehe das Codefragment 5.3) konfiguriert.

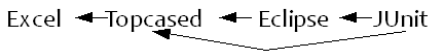


Abbildung 6.1.: Tool-Kette im Beispiel-Projekt

Die Anforderungen an das Bibliotheksverwaltungssystem werden in Excel folgendermaßen spezifiziert:

	A	B
1	ID	Description
2	UC01	Exemplar anlegen
3	UC02	Medium anlegen
4	UC03	Nach Medium suchen
5	UC04	Medium ausleihen
6	UC05	Medium zurücknehmen
7	UC06	Benutzer anlegen
8	UC07	Medium reservieren
9	UC08	Benutzer löschen
10	UC09	System verwalten

Abbildung 6.2.: Anforderungen an das Bibliotheksverwaltungssystem

Das Design des Bibliotheksverwaltungssystems erfolgt in Topcased. Das gesamte Klassendiagramm befindet sich im Anhang B (Abbildung B.1). Jedes relevante

UML-Element wird mit einem Stereotypen *Traceable* (siehe Abbildung 5.2) versehen. Mittels Tagged Value *Reqid* erfolgt die Verlinkung zu den Anforderungen. Die Abbildung 6.3 zeigt wie das *Reqid* von *Traceable* der Klasse *Exemplar* mit dem URI der Anforderung *Exemplar anlegen* gesetzt wird.

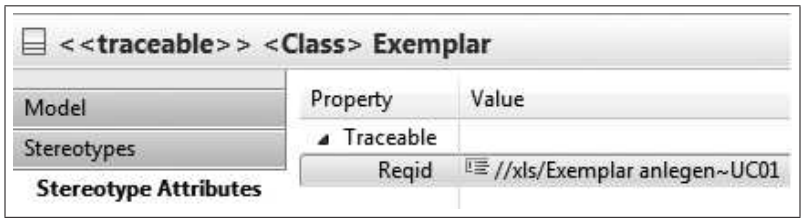


Abbildung 6.3.: Stereotyp *Traceable*: Setzen von *Reqid*

Der Source-Code des Bibliotheksverwaltungssystems ist in JAVA (Eclipse) geschrieben. Die Implementierung ist nicht vollständig und dient lediglich als Beispiel. Die Struktur der Implementierung kann dem Anhang B (Tabelle B.1) entnommen werden.

Das Verlinken von URIs der UML-Elemente zu den Source-Dateien erfolgt anhand des Codefragments 5.1 im Source-Code in Form von Kommentaren. Das Codefragment 6.1 demonstriert das Verlinken der Klasse *Suchmaske* zur Source-Datei *Controlls.java*.

Listing 6.1: Die UML-Klasse *Suchmaske* verlinkt zur Source-Datei *Controlls*

```
1 /*referencedURIs: //uml/Class/Package:Klassenmodell/Suchmaske~
      _cT7oYH62Ed-bMvOI7kjLMQ*/
2 package de.fau.cs.bibliothek.common.controll;
3
4 public class Controlls implements IApplication {
5 }
```

Die Testfälle werden mit JUnit implementiert. Es gibt eine Klasse *ModulTest.java*, die zehn Testfälle beinhaltet. Die Übersicht der Testfälle befindet sich im Anhang B (Tabelle B.2).
Die Verlinkung der Source-Dateien bzw. UML-Komponenten zu den Testfällen (siehe Codefragment 5.2) erfolgt analog zu Verlinkung der UML-Elemente zu den Source-Dateien in Form von Kommentaren. Das Codefragment 6.2 demonstriert die Verlinkung von zwei Source-Dateien *Bibliothek* und *Bibliotheksverwalter* zum einem Testfall *test_legeNutzerAn*.

Listing 6.2: Die Source-Dateien *Bibliothek* und *Bibliotheksverwalter* verlinkt zum Testfall *test_legeNutzerAn*

```
1 /*referencedURIs: //code/de.fau.cs.bibliothek.common.controll/
      Bibliothek,//code/de.fau.cs.bibliothek.common.persistence/
      Bibliotheksverwalter*/
2     @Test
3     public void test_legeNutzerAn() {
4
5     }
```

Das Test-Protokoll befindet sich im Anhang B (Codefragment B.1). Die durchgefallenen Testfälle beinhalten einen Tag *failure* mit der Fehlermeldung.

6.2. Ergebnisse

Dank EMF und CDO können die Inhalte der Datenbank in einem Editor hierarchisch angezeigt werden. Die Abbildung 6.4 zeigt einen Ausschnitt aus der Datenbank.

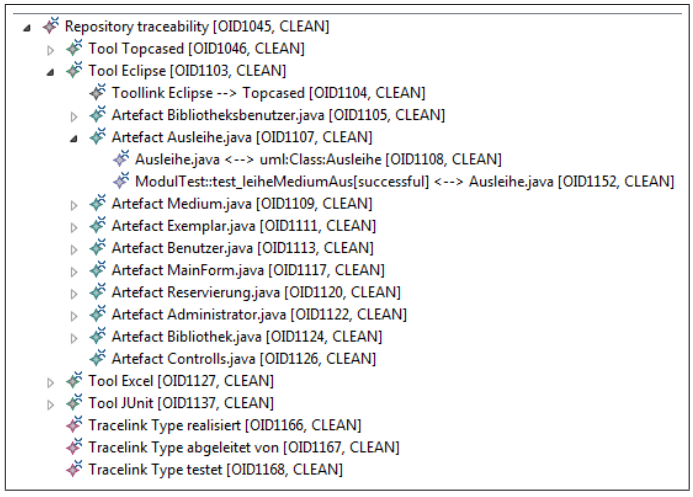


Abbildung 6.4.: Trace-Informationen in der Datenbank

Es gibt in der Datenbank ein Repository-Objekt *traceability*, welches Tool-Objekte und TracelinkType-Objekte beinhaltet. Alle Tool-Objekte beinhalten Artefact-Objekte und Toollink-Objekte. Am Beispiel von Eclipse wird die Liste seiner Artefakten angezeigt. Das Artefakt *Ausleihe.java* hat zwei Tracelinks: Targetlink zur UML-Klasse *Ausleihe* und einen Sourcelink vom Testfall *test_leiheMediumAus*.

Die Ergebnisse der Analysen liegen graphisch vor (siehe Abbildungen 6.5 und 6.6). Die Graphiken wurden mit Hilfe von BIRT[1]-Bibliothek erstellt und sind als PNG-Dateien abgespeichert. Für die Impact Analyse wurde auch ein Bericht in Form einer PDF-Datei erstellt, welcher sich im Anhang C (Abbildung C.1) befindet.

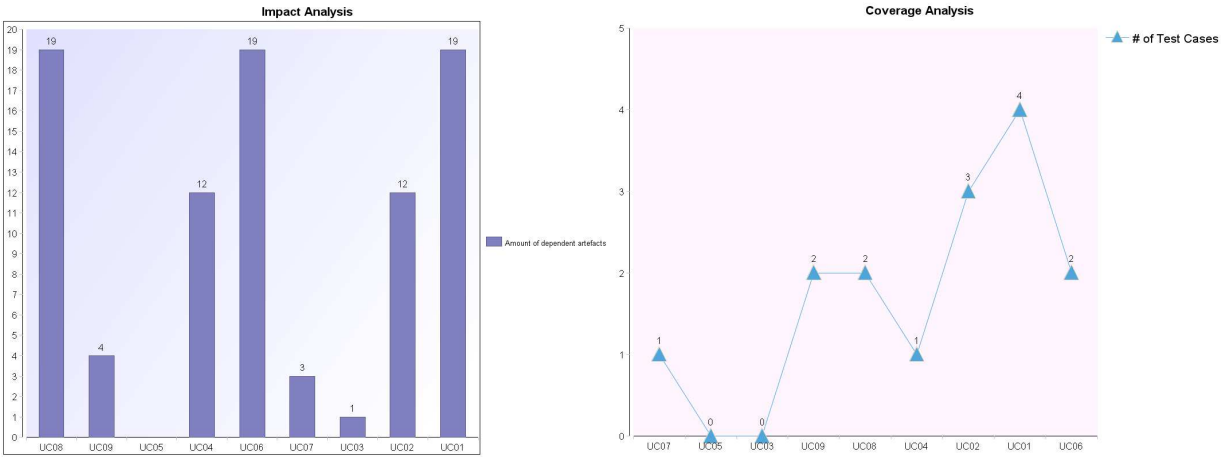


Abbildung 6.5.: Ergebnisse der Impact und Coverage Analyse

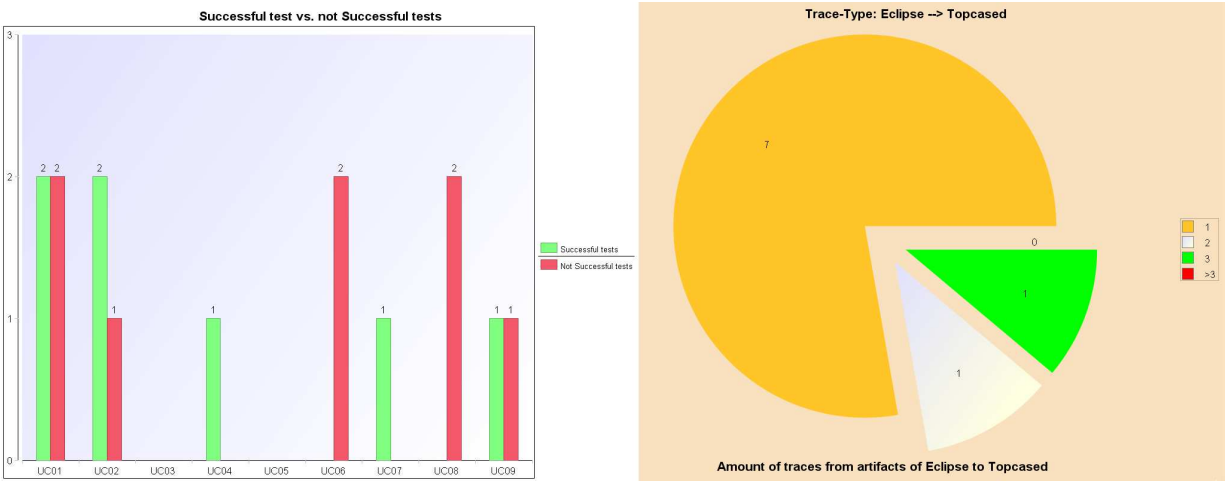


Abbildung 6.6.: Ergebnisse der Test und Kopplung Analyse

Aus Übersichtlichkeitsgründen werden nicht die ganzen URIs von Anforderungen angezeigt, sondern nur ihre IDs, welche in der Excel-Datei stehen und als *UniqueID* beim URI hinzugefügt werden.

[1] http://www.eclipse.org/birt/

Die Ergebnisse der Analysen auf der Metamodell-Ebene (Zyklen und verwaiste Knoten) werden auf der Console als Text ausgegeben.

6.3. Bewertung

Anhand von dem Beispiel-Projekt wird gezeigt, wie das Framework verwendet werden kann um die Trace-Informationen im Projekt zu sammeln und zu analysieren.

Die Impact Analyse zeigt, dass von den Anforderungen *UC08, UC06* und *UC01* ziemlich viele Artefakte abhängen. Sollten sich diese Anforderungen in der Zukunft ändern, würde es einen erheblichen Aufwand für die Projektmitarbeiter bedeuten. Im Gegensatz dazu haben einige Anforderung wie *UC07* und *UC03* sehr wenig abhängige Artefakte. Dies weist darauf hin, dass diese Anforderungen nicht vollständig umgesetzt wurden. Die Anforderung *UC05* wurde gar nicht umgesetzt. Generell ist die Verteilung der abhängigen Artefakte zwischen den Anforderungen nicht besonders gut, da einige zu viele und andere zu wenig abhängige Artefakte haben. Die Anforderungen *UC08, UC06* und *UC01* sind eventuell zu allgemein gehalten und könnten eventuell in mehrere konkretere Anforderungen zerlegt werden.

Laut den Ergebnissen der Coverage Analyse existieren zu den Anforderungen *UC03* und *UC05* überhaupt keine Testfälle. Beim Betrachten der Anforderungen *UC09* und *UC08* wird festgestellt, dass *UC09* vier abhängige Artefakte hat, wobei zwei davon Testfälle sind. Die *UC08* und *UC06* haben neunzehn abhängige Artefakte und lediglich zwei Testfälle davon. Dies ist ein weiterer Hinweis darauf, dass *UC08*, *UC06* und gegebenenfalls auch *UC01* überprüft werden sollen.

Anhand der Test Analyse sind acht Testfälle von insgesamt fünf Anforderungen fehlgeschlagen, was eine relativ hohe Fehlerquote darstellt.

Die Kopplung Analyse zeigt, dass es insgesamt neun Artefakte aus Eclipse gibt, die Tracelinks zu den Artefakten aus Topcased haben. Davon haben sieben Artefakte jeweils einen Tracelink zu einem Artefakt aus Topcased, ein Artefakt hat zwei Tracelinks zu den Artefakten aus Topcased und ein weiteres Artefakt hat drei Tracelinks zu den Artefakten aus Topcased. Die Artefakte, die keine Tracelinks haben, werden durch die Analyse auf der Metamodell-Ebene als verwaiste Knoten entdeckt. Es gibt jedoch keine Artefakte, die mehr als drei Tracelinks zu den Artefakten aus Topcased haben. Wäre dies der Fall, müssten solche Artefakte nochmal in Betracht genommen werden, ob diese in mehrere Artefakte zerlegt werden können.

Generell werden die Ergebnisse der Analysen auf einem relativ einfachen Niveau gehalten. Idealerweise gibt es zu jeder Graphik einen textuellen Bericht (wie im Anhang C), welcher sich zusammen mit der Graphik in einer Datei befindet.

7. Zusammenfassung

In diesem Kapitel werden die Ergebnisse dieser Arbeit zusammengefasst und diskutiert.

7.1. Ergebnisse dieser Arbeit

In dieser Arbeit wurde ein allgemeines Framework konzipiert und prototypisch umgesetzt, welches eine End-To-End Traceability abdeckt. Dabei ist das Framework projektunabhängig, indem es beliebige Entwicklungswerkzeuge integrieren kann. Dafür stellt es eine abstrakte Klasse zum Anbinden konkreter Parser bereit. Die konkreten Parser extrahieren die Entwicklungsdaten aus den jeweiligen Entwicklungswerkzeugen. Die Abhängigkeiten zwischen den Artefakten (siehe Kapitel 3.1) der Entwicklungswerkzeuge werden in den Werkzeugen selbst gehalten (mittels Stereotypen, Kommentare, etc).
Um den Nutzen aus der Traceability zu ziehen, wurden anschließend mehrere Analysen implementiert.

Konzeption und prototypische Implementierung

Zuerst wurde das Traceability-Modell (siehe Kapitel 3.2) entworfen. Der Zweck dieses Modells ist die Artefakte und deren Beziehungen abzubilden. Das Traceability-Modell wurde als ein Ecore-Metamodell umgesetzt und ist allgemein gehalten, indem es Entwicklungsdaten aus sehr unterschiedlichen Datenquellen verknüpfen kann. Dafür beinhaltet das Modell die Elemente *Artefact* und *Tracelink*. Mittels *Artefact* werden die konkreten Artefakte und durch den *Tracelink* die Verbindungen zwischen zwei Artefakten unterschiedlicher Entwicklungswerkzeuge abgebildet. Anhand von *TracelinkType* und *ArtefactType* kann den Tracelinks und Artefakten eine Bedeutung zugeordnet werden. Außerdem besitzt das Modell weitere Elemente wie *Tool* und *Toollink*. Mithilfe von *Tool* werden die Tools abgebildet, welche die Artefakte beinhalten. Über *Toollink* werden die Tools mit einander verbunden. Alle oben genannten Elemente befinden sich in einem Wurzel-Element *Repository*.

Da die Artefakte aus unterschiedlichen Werkzeugen stammen, muss darauf geachtet werden, dass alle Artefakte innerhalb einer Modell-Instanz global eindeutig

sind. Dies wird mit einem URI (siehe Kapitel 3.2.3) gewährleistet. Dazu trägt URI die wesentlich sinnvolle und aussagekräftige Informationen über die Artefakte.

Das Framework (siehe Kapitel 3.3) beinhaltet vier Komponenten. Die Komponente *ParserCreator* ist für das Erzeugen der konkreten Parser verantwortlich. Die Komponente wurde als ein Entwurfsmuster *Fabrikmethode* konzipiert. Somit können beliebige Parser an das Framework leicht angebunden werden. Die Komponente *TraceController* ist eine zentrale Steuerungseinheit. Diese liest die Konfigurations-Datei, in der die konkrete Tool-Kette konfiguriert wurde und erzeugt anhand dieser Informationen die konkreten Parser mit Hilfe von *ParserCreator*. Die Komponente *TraceLinker* erzeugt bzw. überarbeitet die Instanz des Traceability-Modells, indem es die dazu benötigten Daten vom *TraceController* übergeben bekommt. Anschließend wird die Instanz in die Datenbank geschrieben. Dafür gibt es die Komponente *DatabaseWriter*, die die Instanz des Traceability-Modells lädt bzw. speichert.

Es wurden einige beispielhafte Analysen (siehe Kapitel 3.4) konzipiert: Die Analysen auf der Metamodell- und Modell-Ebene. Die Analysen auf der Metamodell-Ebene sind das Auffinden von verwaisten Knoten und Zyklen. Unter verwaisten Knoten werden die Knoten verstanden, die keine Kanten haben. Außerdem kann auch nach halbverwaisten Knoten gesucht werden. Solchen Knoten fehlen in der Regel einige obligatorische Kanten. Verwaiste und halbverwaiste Knoten deuten auf eine Unvollständigkeit im Traceability-Graphen hin.
Die Zyklen im Traceability-Graphen entstehen dann, wenn über mehrere verschiedene Pfade zu demselben Knoten gelangt werden kann. Die Zyklen deuten auf eine eventuelle Anomalie hin.
Die Analysen auf der Modell-Ebene sind Impact Analyse, Coverage Analyse, Test Analyse und Kopplung Analyse. Die Impact Analyse ermittelt alle von einer Anforderung abhängigen Artefakte. Dank dieser Analyse kann z.B. der Aufwand für die Änderung einer Anforderung besser abgeschätzt werden. Die Coverage Analyse prüft, ob es zu jeder Anforderung mindestens einen Testfall gibt. Dabei werden möglichst die Testfälle ermittelt, die diese Anforderung testen und nicht die Testfälle, die über eine andere Anforderung transitiv abhängig sind (wie bei der Impact Analyse). Die Test Analyse ermittelt zu jeder Anforderung deren bestandene und gescheiterte Testfälle. Die Kopplung Analyse analysiert die Kopplung zwischen den Artefakten zweier Tools.

Das oben beschriebene Konzept wurde prototypisch implementiert (siehe Kapitel 5) um die Praxistauglichkeit des Konzepts zu beweisen. Dafür wurden vier konkrete Parser programmiert: für MS Excel, Topcased, Eclipse und JUnit. Alle Parser leiten die abstrakte Klasse *Parser* ab und speichern die extrahierten Daten in einer einheitlichen Datenstruktur. Die Verlinkung der Trace-Informationen

zwischen den Werkzeugen erfolgt mittels eines Stereotypen *Traceable* (siehe Abbildung 5.2) für UML-Elemente und Kommentare im Source-Code.
Die implementierten Analysen (siehe Kapitel 5.6) basieren auf der Graphentheorie, da die Instanz des Traceability-Modells einen Traceability-Graphen mit Artefakten als Knoten und Tracelinks als Kanten darstellt.

Evaluation

Anhand eines (einfachen) Beispiel-Projekts (siehe Kapitel 6) wurde die Implementierung evaluiert. Die Ergebnisse der Analysen konnten Rückschluss darauf geben, ob die Anforderungen zu allgemein bzw. zu spezifisch gehalten sind. Es konnte festgestellt werden, ob diese bereits getestet wurden und wie die Testergebnisse ausgefallen sind. Es wurde auch die Kopplung zwischen den Artefakten zweier Werkzeuge bestimmt.

7.2. Nachteile beim Einsetzen des Frameworks

Beim Einsetzen des Frameworks müssen einige Nachteile in Kauf genommen werden:

1. Alle expliziten Tracelinks müssen manuell dokumentiert werden (mittels Stereotypen oder Kommentare im Source-Code). Dabei muss die eintragende Person auf die Korrektheit der URIs referenzierter Artefakten achten. Falsche Eingaben bedeuten, dass der Tracelink zwischen den verlinkten Artefakte im Traceability-Modell nicht erstellt wird.
   Das manuelle Einpflegen der expliziten Tracelinks ist aber generell notwendig um eine Traceability zu erzielen.

2. Die Eingaben in der Konfigurationsdatei erfolgen ebenso manuell. Falsche oder widersprüchliche Eingaben erzeugen keine vollständigen Trace-Informationen. Es könnte automatisch partiell geprüft werden, ob die Eingaben falsch bzw. inkonsistent sind. Eine komplette automatische Prüfung ist unmöglich, da nur die Projektmitarbeiter die richtigen Werkzeugnamen sowie die Pfade zu den Quell-Dateien kennen und die vollständige valide Tool-Kette bestimmen können.
   Dies ist aber recht übersichtlich und lässt sich von einem Mitarbeiter, der die Konfigurationsdatei richtig pflegt, erledigen.

3. Da es sich bei dem Framework um ein sogenanntes *White-Box* Framework [15] handelt, müssen die Projektmitarbeiter die Kenntnisse über den inneren Aufbau des Frameworks besitzen um es in einem konkreten Projekt zu verwenden. Somit ist ein größerer Einarbeitungsaufwand notwendig.

7.3. Ausblick

Evaluation des Frameworks in mehreren reellen Industrieprojekten ist notwendig um festzustellen, ob sich das Framework in der Praxis etablieren kann. Dabei könnte festgestellt werden, was dem Framework noch fehlt und was in dem Framework überflüssig ist. Außerdem kann erforscht werden, wie schwerwiegend die oben beschriebenen Nachteile sich in konkreten Projekten auswirken.

Die Analysen sollten in der Zukunft mit *IncQuery Base* umgesetzt werden, sobald das Problem mit den Containment Referenzen vom Entwickler gelöst wird. Die Bibliothek *IncQuery Base* bietet sehr effiziente Implementierung für die Bestimmung der transitiven Hülle, was für viele Analysen verwendet werden kann. Bei großen Projekten ist dies von erheblichem Vorteil.
Die Ergebnisse der Analysen sollten neben den graphischen Informationen auch die textuelle Beschreibung enthalten.
Ein Mechanismus, welches das Anbinden von Analyse Pattern und Constraints projektunabhängig ermöglicht, würde die Analysen auf die möglichen konkreten Pojekt-Zwecke enger zuschneiden. Hierfür wäre ein Konzept notwendig, welches die Analysen um die Eingabe und das Evaluieren der Analyse Pattern bzw. Constraints (z.B. mit einer Anfragesprache) erweitert.

Den Artefakten soll künftig eine vordefinierte Semantik zugewiesen werden. Beispiele dafür sind z.B. *rejected* oder *expired*. Dies kann mithilfe vom Attribut `properties` vom *Artefact* (siehe Kapitel 3.2.4) erfolgen. Mittels des Attributs `visible` sollen die Artefakte je nach ihrer semantischen Bedeutung für die Analyse eingeblendet bzw. ausgeblendet werden.

Damit die Trace-Informationen in der Datenbank immer up-to-date gehalten werden, soll ein Automatismus geschaffen werden, welcher den Trace-Prozess nächtlich neu ausführt um die über den Tag erfolgten Änderungen im Repository zu aktualisieren. Dabei soll der alte Stand nicht einfach überschrieben, sondern separat in einer History abgespeichert werden. Es soll auch möglich sein automatische Vergleiche zwischen dem aktuellen Stand und den Trace-Informationen aus der History durchführen zu können.

Da die manuelle Erstellung von Tracelinks einen erheblichen Aufwand mit sich bringt, soll die Usability diesbezüglich verbessert werden. Jeder Mitarbeiter soll in seinem Tool arbeiten und dort die Tracedaten erzeugen, pflegen und analysieren können. Weitere Ausbaustufen wären dann z.B. Impact Analysen in den einzelnen Tools zu ermöglichen. Einfachere Varianten sind Autocompletion von Schlüsselinformationen, die dem Mitarbeiter automatische Hilfestellung zum Erstellen bzw. Einpflegen der Tracelinks ermöglichen.

Literaturverzeichnis

[1] CDO Model Repository Overview. http://www.eclipse.org/cdo/documentation. Letzter Zugriff: 06.02.2013.

[2] Platform Plug-in Developer Guide. http://help.eclipse.org/indigo/topic/org.eclipse.platform.doc.isv/guide/arch.htm. Letzter Zugriff: 04.02.2013.

[3] V-Modell. http://moodle.fhsg.ch/mw_swe/images/3/3f/V-Modell_Resultate.gif. Letzter Zugriff: 04.04.2013.

[4] OMG Unified Modeling Language (UML) Infrastructure, Version 2.1.2, 2007.

[5] Josef Adersberger. *Modellbasierte Extraktion, Repräsentation und Analyse von Traceability-Informationen.* Dissertation, Friedrich-Alexander-Universität, 2012.

[6] Markus Aleksy, Tobias Hildenbrand, Claudia Obergfell, and Michael Schwind. A pragmatic approach to traceability in model-driven development. In Armin Heinzl, Hans-Jrgen Appelrath, and Elmar J. Sinz, editors, *PRIMIUM*, volume 328 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2008.

[7] Andreas Willert. Newsletter No 24 - Willert Software. www.willert.de/assets/Newsletter/NL24-Richtigesrichtigtun-v4r0.pdf, 2010. Letzter Zugriff: 07.10.2012.

[8] Hazeline U. Asuncion, Frédéric François, and Richard N. Taylor. An end-to-end industrial software traceability tool. In *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ESEC-FSE '07, pages 115–124, New York, NY, USA, 2007. ACM.

[9] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform Resource Identifiers (URI): Generic Syntax. 1998.

[10] Christian Neuhaus. Traceability von Komponentenmodellen zu Code über in UML integrierte Reflexion-Modelle. Diplomarbeit, Friedrich-Alexander-Universität, Erlangen-Nürnberg, 2009.

[11] Jane Cleland-Huang, Carl K. Chang, and Mark Christensen. Event-based traceability for managing evolutionary change. *IEEE Transactions on Software Engineering*, 29(9):796–810, 2003.

[12] International Business Machines Corp. Eclipse Platform Technical Overview. http://eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf, 2006. Letzter Zugriff: 04.02.2013.

[13] Ralf Dömges and Klaus Pohl. Adapting traceability environments to project-specific needs. *Commun. ACM*, 41(12):54–62, December 1998.

[14] Alexander Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Trans. Softw. Eng.*, 29(2):116–132, February 2003.

[15] Marcus Fontoura, Wolfgang Pree, and Bernhard Rumpe. *The Uml Profile for Framework Architectures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[16] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software*. Addison-Wesley Longman, Amsterdam, 1 edition, 1995. 37. Reprint (2009).

[17] Orlena C. Z. Gotel and Anthony C. W. Finkelstein. An analysis of the requirements traceability problem. pages 94–101, 1994.

[18] Krzysztof Kowalczykiewicz and Dawid Weiss. Traceability: Taming uncontrolled change in software development, 2002.

[19] Patrick Mäder, Ilka Philippow, and Matthias Riebisch. Customizing traceability links for the unified process. In *Proceedings of the Quality of software architectures 3rd international conference on Software architectures, components, and applications*, QoSA'07, pages 53–71, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] G. Montalto, A. Pasetti, and N. Salerno. Application of Software Framework Technology to an Antenna Pointing Controller. In *Data System in Aerospace (DASIA) Conference*, Dublin, May 2002.

[21] S. Nolte. *QVT - Operational Mappings: Modellierung mit der Query Views Transformation*. Springer, 2009.

[22] Jens Palluch. Traceability: Tipps zur Realisierung nachvollziehbarer Anforderungen. *SQ Magazin*, Dezember 2010.

[23] Thomas Pohl. Komfortabel und leicht: Modellgetriebene Softwareentwicklung mit EMF und Xtext. *Heise Developer*, 2010(26.01.2010), 2010.

[24] Balasubramaniam Ramesh and Matthias Jarke. Toward reference models for requirements traceability. *IEEE Trans. Softw. Eng.*, 27(1):58–93, January 2001.

[25] Dr. Hossein Saiedian and Andrew Kannenberg. Why software requirements traceability remains a challenge, 2009.

[26] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley, Boston, MA, 2. edition, 2009.

[27] Volker Turau. *Algorithmische Graphentheorie*. Oldenbourg, München, Germany, 3rd edition, October 2009.

[28] Klaudia Dussa-Zieger und Michael Gerdom. Traceability - Anspruch und Realität. *MQ Management und Qualität*, April 2008.

[29] Markus Völter. Metamodellierung. http://www.voelter.de/data/presentations/metamodelling-paper.pdf, 2004. Letzter Zugriff: 07.02.2013.

[30] K.E. Wiegers. *Software Requirements*. Microsoft Press, 2009.

[31] Frank Zimmermann. Nutzung und Erfolg modellgetriebener Softwareentwicklung. Arbeitspapiere der Nordakademie 2009-10, Elmshorn, 2009.

A. Anhang A

A.1.

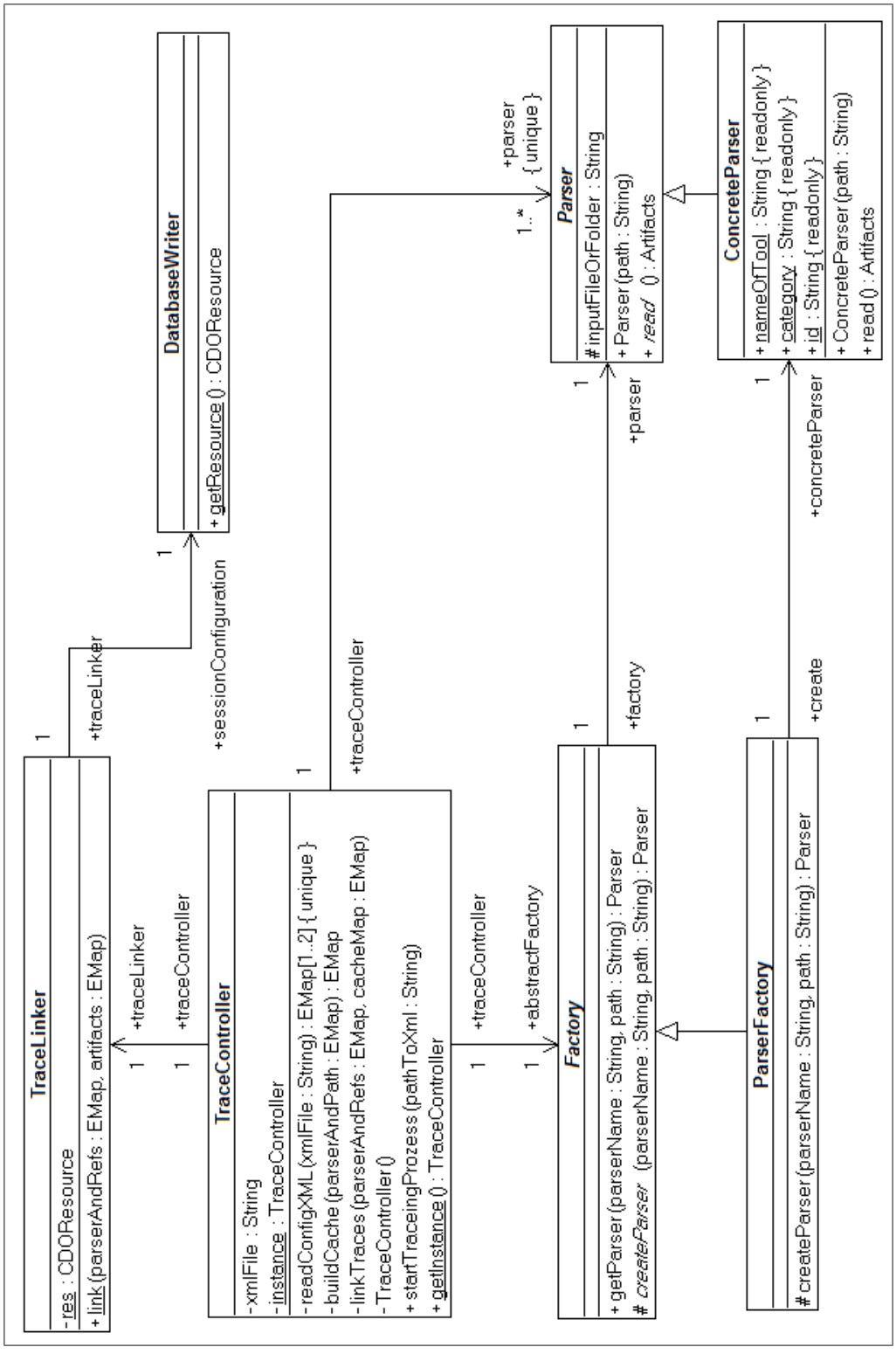


Abbildung A.1.: Klassendiagramm des Frameworks

B. Anhang B

B.1.

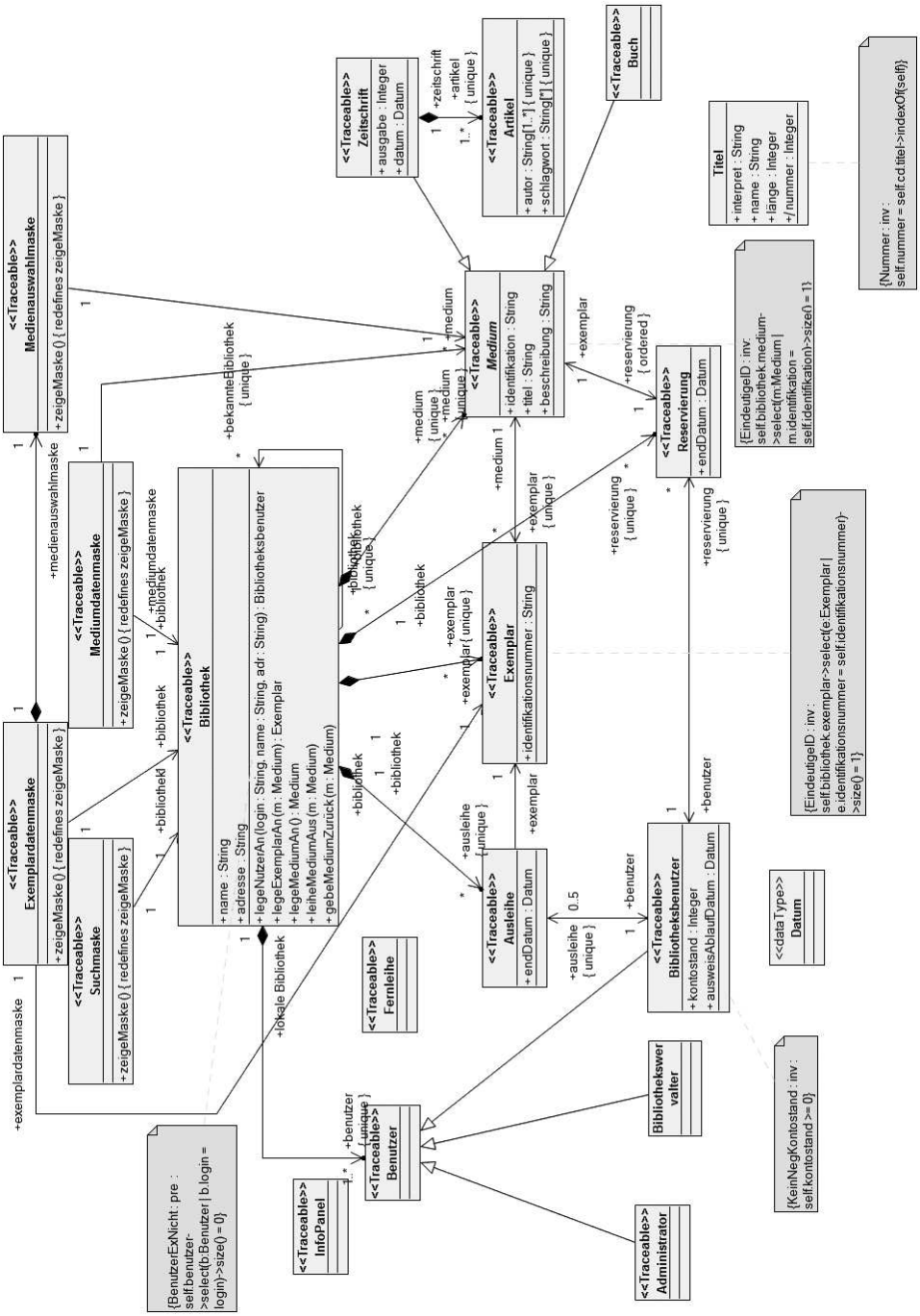


Abbildung B.1.: Klassendiagramm des Bibliotheksverwaltungssystems

B.2.

Package	JAVA-Klassen
de.fau.cs.bibliothek.common.controll	Bibliothek
de.fau.cs.bibliothek.common.gui	Controlls, MainForm
de.fau.cs.bibliothek.common.persistence	Medium, Exemplar, Reservierung, Ausleihe, Administrator, Bibliotheksbenutzer, Bibliotheksverwalter

Tabelle B.1.: Source-Dateien des Bibliotheksverwaltungssystems

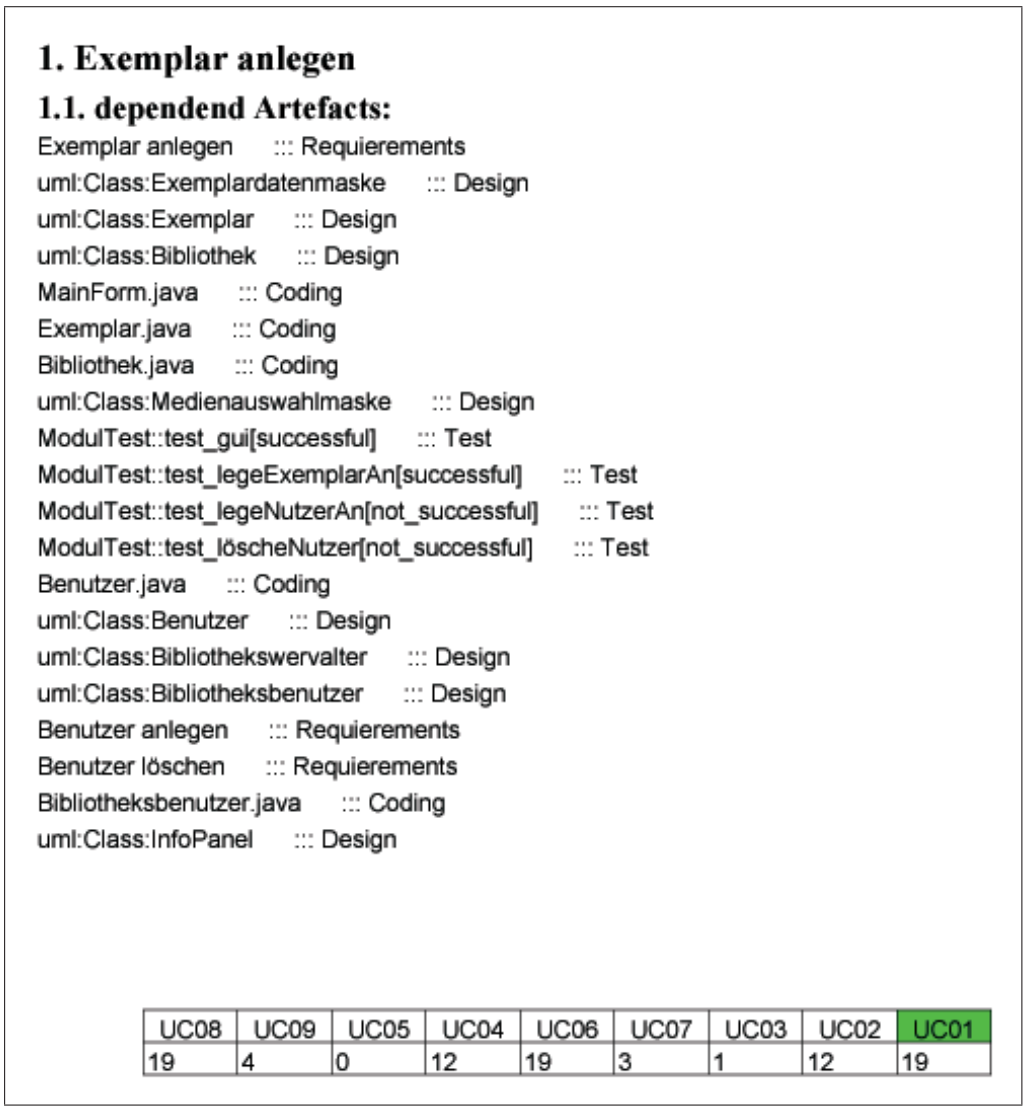
Testfall
test_legeExemplarAn
test_legeNutzerAn
test_legeNutzerAn
test_leiheMediumAus
test_gebeMediumZurück
test_test_gui
test_mapper
test_lösche Admin
test_reserviereMedium
test_lösche Nutzer

Tabelle B.2.: Testfälle des Bibliotheksverwaltungssystems

Listing B.1: JUnit-Test-Protokoll

```
<testsuite>
    <testcase classname="ModulTest" name="test_legeNutzerAn">
     <failure type=""> details about failure </failure>
    </testcase>
    <testcase classname="ModulTest" name="test_legeExemplarAn"/>
    <testcase classname="ModulTest" name="test_legeMediumAn"/>
     <testcase classname="ModulTest" name="test_leiheMediumAus"/>
       <testcase classname="ModulTest" name="test_gui"/>
    <testcase classname="ModulTest" name="test_mapper">
       <failure type=""> details about failure </failure>
    </testcase>
    <testcase classname="ModulTest" name="legeAdmin_An"/>
     <testcase classname="ModulTest" name="löscheAdmin">
       <failure type=""> details about failure </failure>
    </testcase>
    <testcase classname="ModulTest" name="reserviereMedium"/>
    <testcase classname="ModulTest" name="test_löscheNutzer">
       <failure type=""> details about failure </failure>
    </testcase>
    <testcase classname="ModulTest" name="gebeMediumZurück">
       <failure type=""> details about failure </failure>
    </testcase>
</testsuite>
```

C. Anhang C



1. Exemplar anlegen

1.1. dependend Artefacts:

Exemplar anlegen ::: Requierements
uml:Class:Exemplardatenmaske ::: Design
uml:Class:Exemplar ::: Design
uml:Class:Bibliothek ::: Design
MainForm.java ::: Coding
Exemplar.java ::: Coding
Bibliothek.java ::: Coding
uml:Class:Medienauswahlmaske ::: Design
ModulTest::test_gui[successful] ::: Test
ModulTest::test_legeExemplarAn[successful] ::: Test
ModulTest::test_legeNutzerAn[not_successful] ::: Test
ModulTest::test_löscheNutzer[not_successful] ::: Test
Benutzer.java ::: Coding
uml:Class:Benutzer ::: Design
uml:Class:Bibliothekswervalter ::: Design
uml:Class:Bibliotheksbenutzer ::: Design
Benutzer anlegen ::: Requierements
Benutzer löschen ::: Requierements
Bibliotheksbenutzer.java ::: Coding
uml:Class:InfoPanel ::: Design

UC08	UC09	UC05	UC04	UC06	UC07	UC03	UC02	UC01
19	4	0	12	19	3	1	12	19

Abbildung C.1.: Impact Analyse: PDF-Bericht (Ausschnitt)