Friedrich-Alexander-Universität Erlangen-Nürnberg
Technische Fakultät, Department Informatik

MARTIN WAGNER

BACHELOR THESIS

VISUELLE ANNOTATION VON QUELLCODE MIT TRACEABILITY-INFORMATIONEN

Eingereicht am 2. Februar 2020

Betreuer:
Prof. Dr. Dirk Riehle, M.B.A.
Julia Krause
Professur für Open-Source-Software
Department Informatik, Technische Fakultät
Friedrich-Alexander-Universität Erlangen-Nürnberg

Dr.-Ing. Dipl.-Inf. Martin Jung, Lehrbeauftragter am Department für Informatik
der Friedrich-Alexander-Universität Erlangen-Nürnberg
Dipl.-Ing. Niko Pollner
develop group

Versicherung

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.

Erlangen, 2. Februar 2020

License

Erlangen, 2. Februar 2020

Abstract

Poor requirements management is one of the primary reasons for project failure. Managing and tracing of requirements involve additional effort for the developers. The tracing of requirements involves the creation of links between requirements and source code files. These links are also referred to as traceability information. The developers' development environments do not yet support visualisation of the traceability information at source code level.

Due to the lack of visualisation at source code level, the developer sees no benefit in updating the traceability information. The traceability information is neglected, which inevitably leads to poor requirements management.

The absence of the visualisation of traceability information at source code level is quintessential here. The missing visualisation of the connections between requirements and source code at source code level raises the core question of this thesis.

How to highlight, visualise and manage requirements at source code level?

This thesis on visual annotation of source code with traceability information documents the development of a prototype. This prototype supports the highlighting, visualisation and management of requirements at source code level and thus the visualisation of traceability information. The relevance of the prototype is evaluated in a survey in a university environment.

Zusammenfassung

Schlechtes Anforderungsmanagement ist einer der Hauptgründe für das Scheitern von Projekten. Das Verwalten und Nachverfolgen von Anforderungen ist mit Aufwand für die Entwickler verbunden. Für die Nachverfolgung von Anforderungen werden Verbindungen zwischen Anforderungen und Quellcodedateien angelegt. Diese Verbindungen werden auch als Traceability-Informationen bezeichnet. Die Entwicklungsumgebungen der Entwickler unterstützen die Visualisierung dieser Traceability-Informationen noch nicht auf Quellcode-Ebene. Aufgrund der fehlenden Visualisierung auf Quellcode-Ebene sieht der Entwickler keinen Nutzen in der Aktualisierung der Traceability-Informationen. Die Traceability-Informationen werden vernachlässigt und dadurch kommt es zu schlechtem Anforderungsmanagement.

Ausschlaggebend ist hierbei die fehlende Visualisierung der Traceability-Informationen auf Quellcode-Ebene. Die fehlende Visualisierung der Verbindungen zwischen Anforderungen und Quellcode auf Quellcode-Ebene zieht die zentrale Frage der vorliegenden Arbeit nach sich.

Wie können Anforderungen auf Quellcode-Ebene markiert, visuell dargestellt und verwaltet werden?

Die vorliegende Arbeit zur visuellen Annotation von Quellcode mit Traceability-Informationen dokumentiert die Entwicklung eines Prototyps. Dieser Prototyp ermöglicht die Markierung, visuelle Darstellung und Verwaltung von Anforderungen auf Quellcode-Ebene, also die Visualisierung der Traceability-Informationen. In einer Umfrage im universitären Umfeld wird die Relevanz des Prototyps bewertet.

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung meiner Bachelorarbeit unterstützt und motiviert haben. Zuerst gebührt mein Dank Dr.-Ing. Dipl.-Inf. Martin Jung, Dipl.-Ing. Niko Pollner und Julia Krause, die mich während der gesamten Bachelorarbeit betreut haben. Für die hilfreichen Anregungen und die konstruktive Kritik bei der Erstellung dieser Arbeit möchte ich mich herzlich bedanken.

Ein besonderer Dank gilt allen Teilnehmern und Teilnehmerinnen meiner Umfrage, die einen großen Beitrag zu dieser Arbeit beisteuert. Mein Dank gilt ihrer Informationsbereitschaft und ihren interessanten Beiträgen und Antworten auf meine Fragen. Danke auch allen fleißigen Korrekturlesern und -leserinnen. Besonderer Dank gilt auch meiner Freundin, die mich immer unterstützt und auch während dieser Bachelorarbeit immer für mich da war. Abschließend möchte ich mich bei meiner Familie bedanken, die mein Studium durch ihre Unterstützung ermöglicht hat und mir auch während dieser Arbeit den Rücken freihielt.

Martin Wagner

Erlangen, 2. Februar 2020

Gender-Erklärung

In dieser Arbeit wird aus Gründen der besseren Lesbarkeit die Sprachform des generischen Maskulinums angewandt. Weibliche und anderweitige Geschlechteridentitäten werden dabei ausdrücklich mitgemeint, sofern es für die Aussage erforderlich ist.

Inhaltsverzeichnis

1 Einleitung **1**
- 1.1 Motivation . . . 1
- 1.2 Forschungsfragen . . . 3
- 1.3 Aufbau der Arbeit und Umfrage . . . 3

2 Grundlagen und Begriffsklärungen **5**
- 2.1 Requirements-Management . . . 5
- 2.2 Traceability-Daten . . . 6
- 2.3 Tracing Overlay . . . 7
- 2.4 Begriffe für die Umfrage . . . 10

3 Verwandte Arbeiten und Stand der Technik **12**
- 3.1 Verwandte wissenschaftliche Arbeiten . . . 12
- 3.2 Verwandte Tools . . . 17
- 3.3 Vergleich bestehender Ansätze und Stand der Technik . . . 18

4 Requirements **20**
- 4.1 Stakeholder-Analyse . . . 20
- 4.2 Aufgabe des Prototyps . . . 21
- 4.3 Anwendungsfälle des Prototyps . . . 22
- 4.4 Requirements an den Prototyp . . . 22

5 Architektur und Entwurf **28**
- 5.1 Kontextabgrenzung des Tools . . . 28
- 5.2 Speicherung der Traceability-Daten . . . 31
- 5.3 Datengrundlage . . . 33
- 5.4 Dynamische Sicht . . . 34
- 5.5 Verwendete Technologien . . . 37

6 Umsetzung **39**
- 6.1 Realisierung der Komponenten . . . 39
- 6.2 Implementierung des Tracing Overlays . . . 40
- 6.3 Implementierung der eigenen Klassen . . . 45

6.4 Entwicklung der programmatischen Tests 47

7 Evaluation **49**
7.1 Evaluation des Prototyps 49
7.2 Auswertung der Umfrage 57

8 Zusammenfassung und Ausblick **62**

Anhänge **65**
Anhang A Screenshots für die Evaluation des Prototyps 65

Literaturverzeichnis **71**

Abbildungsverzeichnis

2.1 Lücke zwischen Quellcodedatei und Quellcode 7
2.2 Exemplarischer Python-Quellcode mit Traceability-Informationen 9
2.3 Beispielhafte Benutzeroberfläche für ein *Tracing Overlay für UML und SysML* 11

5.1 Kontextsicht auf die Softwarearchitektur 28
5.2 Logische Komponenten der Tools mit externen Interaktionspartnern 30
5.3 Visualisierung der Datengrundlage 34
5.4 Sequenzdiagramm für das Starten und Beenden des Tools 35
5.5 V1 - Sequenzdiagramm für R-06 36
5.6 V2 - Sequenzdiagramm für R-06 36
5.7 *EditBox* Beispiel 37

6.1 Realisierungssicht für das *TRC Overlay* 40
6.2 Layer-System R-03, R-02, R-01 42
6.3 Layer-System R-01, R-02, R-03 43
6.4 Layer-System mit hervorgehobenen Requirements 44

7.1 *TRC View* mit fünf Requirements in zweizeiliger Darstellung . . . 51
7.2 Box wird korrekt angezeigt 53
7.3 Box wird nicht korrekt angezeigt 53
7.4 Nach dem Hinzufügen der führenden Leerzeichen in Zeile 47 . . . 53
7.5 Beschäftigungsverhältnisse der Umfrageteilnehmer 57
7.6 Durchschnittliches Interesse an Software Architektur Features . . 58
7.7 Durchschnittliches Interesse an Features für Programmierer 59
7.8 Durchschnittlicher Prozentualer Mehraufwand 60

8.1 Test von verschiedenen Box Formen 65
8.2 Test von Boxen in schlecht formatiertem Quellcode 66
8.3 Nutzereingabe der anzuzeigenden Requirements 67
8.4 Anzeige der Requirements in der *TRC View* 68
8.5 Zu verwendende Requirements sind in der *TRC View* ausgewählt 69
8.6 Neue Box wurde erstellt 70

1 Einleitung

Am 13.7.2019 fiel das europäische Satellitensystem Galileo für mehrere Tage komplett aus. Grund für den Ausfall war ein technischer Vorfall. Das Galileo System ist das europäische Äquivalent zum amerikanischen GPS. Es befindet sich derzeit in einer Pilotphase und soll später auch für selbstfahrende Autos verwendet werden (Lücker, 2019). Das Auftreten von technischen Störungen kann verringert werden, indem man die möglichen Ursachen behandelt. Ein Grund für technische Vorfälle ist schlechtes Requirements-Management. Das Requirements-Management bezeichnet die Verwaltung und Aktualisierung von Requirements, also Anforderungen an das System, im Verlauf des Projekts. Das Requirements-Management kann durch das Aktuellhalten der Zuordnungen zwischen Requirements und Artefakten, den sogenannten Traceability-Informationen, unterstützt werden. Dieses Aktualisieren fordert die Mitarbeit des „low-end“ Nutzers, des Ingenieurs. Um den Ingenieur bei der Mitarbeit zu unterstützen und so zum besseren Requirements-Management beizutragen untersucht diese Arbeit einen Ansatz zur visuellen Annotation von Quellcode mit Traceability-Informationen.

1.1 Motivation

Zur Motivation der Arbeit wird das folgende beispielhafte Szenario betrachtet:

Für die Planung eines großen Systems treffen sich fünf Stakeholder, Vertreter einer Interessensgruppe, mit unterschiedlichen Interessen. Aus den unterschiedlichen Ideen und Wünschen der Stakeholder wird eine Spezifikation gebildet, mit der alle Stakeholder einverstanden sind. Die Spezifikation besteht aus den Requirements, den Anforderungen an das System. Einige Systemarchitekten erstellen einen Systementwurf und Sichten auf die Systemarchitektur. Anhand dieses Systementwurfs wird von vielen Ingenieuren ein Endprodukt entwickelt. Der anfänglich gebündelte Informations- und Arbeitsfluss teilt sich ab dem Systementwurf auf viele Mitarbeiter auf. Die Mitarbeiter sind in diesem Fall Ingenieure, die an der Entwicklung des Systems arbeiten. Damit alles reibungslos ablaufen kann, ist sowohl im Vorfeld, als auch insbesondere *während der Umsetzung* Kommunikation und Planung unabdingbar.

In diesem Beispiel verläuft aufgrund der Planung im Vorfeld alles reibungslos. Das System wird getestet, ausgeliefert und installiert. Das folgende Problem entsteht nach der erfolgreichen Auslieferung der ersten Version des Systems.

Nach einem Jahr soll eine zusätzliche Funktion eingebaut werden. Mittlerweile sind von den ursprünglich an der Entwicklung beteiligten Ingenieuren nicht mehr alle involviert. Von den verbliebenen Ingenieuren erinnert sich kaum jemand, warum bei der Entwicklung bestimmte Aspekte auf eine bestimmte Art umgesetzt wurden. Dokumentation ist nur in Form von sporadischen Dokumentationskommentaren vorhanden. Aus diesem Grund ist eine lange Einarbeitungszeit notwendig. Das System hat sich durch die kleinen, schlecht dokumentierten Änderungen soweit vom ursprünglichen Systementwurf entfernt, dass dieser kaum noch zur Einarbeitung beitragen kann. Die beste Planung im Vorfeld wird wertlos, wenn Systementwurf und Sichten auf die Systemarchitektur nicht aktualisiert und mitgeführt werden.

Um die Zuordnung zwischen Systementwurf und System gewährleisten zu können, sind gute Requirements notwendig. Doch gute Requirements allein sind nicht ausreichend. Zusätzlich ist sogenannte Traceability, Nachverfolgbarkeit, der Requirements notwendig. Die Verbindungen zwischen Requirements und Artefakten, wie beispielsweise Quellcodedateien, ermöglichen diese Traceability. Diese Verbindungen werden hierbei als Traceability-Daten oder Traceability-Informationen bezeichnet.

Es ist nicht überraschend, dass der Systementwurf häufig nicht aktualisiert wird. Für das Aktualisieren ist die Mitarbeit des Ingenieurs gefragt. Der Ingenieur kann jedoch zumeist keinen direkten Nutzen beobachten, wenn er die Traceability-Daten aktualisiert (Arkley & Riddle, 2005). Dadurch neigt er dazu, das Aktualisieren der Traceability-Daten zu vernachlässigen. Aus diesem Grund divergieren System und Systementwurf häufig nach der initialen Planungsphase.

Durch visuelle Annotation von Quellcode mit Traceability-Informationen kann der Ingenieur, der die Traceability-Daten anlegen muss, direkt von den angelegten Daten profitieren. Dadurch wird er dazu motiviert, die Traceability-Daten korrekt anzulegen und aktuell zu halten. Aktuelle Traceability-Daten erleichtern das Aktuellhalten der Sichten auf die Softwarearchitektur und das Nachvollziehen von Änderungen. Dies wiederum reduziert vorbeugend die Wahrscheinlichkeit, dass System und Systementwurf im Entwicklungszyklus divergieren.

Insbesondere bei der Benutzung und Visualisierung von Traceability-Informationen gibt es nach Winkler und Pilgrim (2010) noch Bedarf, denn in bestehenden Tools können Traceability-Informationen oft keinen Nutzen beisteuern, da keine Visualisierung erfolgt.

1.2 Forschungsfragen

Ausschlaggebend ist die fehlende Visualisierung der Traceability-Informationen auf Quellcode-Ebene. Durch die fehlende Visualisierung sieht der Anwender, in diesem Fall der Programmierer, keinen Nutzen in der Aktualisierung der Traceability-Informationen. Die Traceability-Informationen werden vernachlässigt und verlieren somit gänzlich ihren Nutzen. Das macht die Aktualisierung in den Augen des Nutzers noch sinnloser. Um diese Abwärtsspirale zu durchbrechen ist eine Sensibilisierung des Programmierers für Traceability notwendig. Diese Sensibilisierung kann beispielsweise mithilfe kleiner Annehmlichkeiten für den Programmierer, wie dem in dieser Arbeit umgesetzten Tool, erfolgen.

Die aus dieser Motivation resultierenden Forschungsfragen sind:

1. *Wie können Requirements auf Quellcode-Ebene markiert, visuell dargestellt und verwaltet werden?*
2. *Wie lässt sich eine derartige Software realisieren?*

Die Fragen verdeutlichen, dass es neben dem in der vorliegenden Arbeit vorgestellten Ansatz weitere gibt. Zudem setzen sie den Fokus auf die Ingenieursseite dieser wissenschaftlichen Arbeit. Durch Frage 1 wird sichergestellt, dass mögliche Ansätze zur Umsetzung gegeneinander abgewogen werden sollen. Frage 2 stellt diese abwägende Herangehensweise auch bei der programmatischen, visuellen Umsetzung sicher. Die Forschungsfragen sind die Grundlage für die Software, die in dieser Arbeit implementiert wird.

1.3 Aufbau der Arbeit und Umfrage

Diese studentische Arbeit basiert auf der Forschungsrichtung zur visuellen Annotation von Quellcode mit Traceability-Daten. Aus diesem Grund wird eine dynamische, iterative Herangehensweise auf Basis der Forschungsfragen gewählt.

Aufbau der Arbeit Zu Beginn werden für diese Arbeit relevante Grundlagen und Begriffe erklärt. Anschließend werden verwandte Arbeiten und Tools kurz vorgestellt und der Stand der Technik dargelegt. Danach werden die Requirements dieser Arbeit erläutert. Eine Stakeholderanalyse klassifiziert die Wünsche der Interessensgruppen. Die Aufgabe des Prototyps ist die Arbeit des Programmierers zu erleichtern. Es folgen die Anwendungsfälle und daraus resultierenden Requirements an den Prototyp. Auf welche Art diese Requirements überprüft werden, wird dabei mit einem Evaluierungsschema vorgegeben. Verschiedene Sichten ermöglichen einen Einblick in die Softwarearchitektur und den Entwurf des Prototyps. Nach der Erklärung der prototypinternen Zusammenhänge anhand einer detaillierten Sicht auf Komponenten-Ebene wird die Umsetzung der einzelnen

Komponenten beschrieben. Diese Umsetzung ist in drei Bestandteile unterteilt: Erstens die Realisierung des *Tracing Overlays* durch Anpassung bestehender Software, zweitens die Implementierung eigener Klassen und drittens die Entwicklung der Tests für den Prototyp. Es folgt die Evaluation des Prototyps, dessen Requirements auf Erfüllung geprüft werden. Die anschließende Auswertung der Umfrage bestätigt die Notwendigkeit des Tools. Abschließend wird eine Zusammenfassung über die Ergebnisse dieser Arbeit und ein Ausblick über weitere Forschungsmöglichkeiten gegeben.

Umfrage Um auch die Wahl der Requirements an das Tool evaluieren zu können, wird zu Beginn der Entwicklung des Prototyps eine Umfrage gestartet. Die Umfrage richtet sich an die Zielgruppen, die von einem Tool zur visuellen Annotation von Quellcode mit Traceability-Informationen profitieren könnten. Sie wird über E-Mail-Verteiler der technischen Fakultät der Friedrich-Alexander-Universität Erlangen-Nürnberg (FAU) verteilt und online ausgefüllt. Die E-Mails werden dabei an Studenten des Studiengangs Informatik und wissenschaftliche Mitarbeiter der FAU versendet. Auch Mitarbeiter aus Industrieprojekten werden zur Teilnahme an der Umfrage eingeladen. Die Erhebung der Umfrageantworten erfolgt in einem viermonatigen Zeitraum mit anschließender Auswertung. In der Umfrage sind Beispiele für das Tool zur Verbesserung der Traceability von Projekten gegeben. Umfrageteilnehmer werden um eine Einschätzung gebeten, ob sie Tools dieser Art verwenden würden, und welche Vorteile oder Nachteile sie durch die Verwendung erwarten. Die in der Umfrage vorgestellten Ideen umfassen den Hauptgegenstand dieser Arbeit, das sogenannte *Tracing Overlay*, und einige Erweiterungsmöglichkeiten, die in Kapitel 2 genauer vorgestellt werden.

2 Grundlagen und Begriffsklärungen

Die Forschungsrichtung zur visuellen Annotation von Quellcode mit Traceability-Daten basiert auf grundlegenden Elementen und Begriffen. Um die vorliegende Arbeit weiter ausführen zu können, werden diese Grundlagen in diesem Kapitel erklärt.

2.1 Requirements-Management

Die Dokumentation von Änderungen und Entwicklungsentscheidungen eines Systems kann durch Requirements-Management umgesetzt oder unterstützt werden. Requirements-Management umfasst Maßnahmen zur Planung, Analyse, Kommunikation und Steuerung, sowie Kontrolle und Verwaltung von Requirements (PMI, 2014a).

Auch das Zuordnen von Requirements zu Artefakten des Systems ist ein Bestandteil des Requirements-Management. Diese Zuordnungen werden Traceability-Daten genannt. Solche Traceability-Daten werden besonders in sicherheitsrelevanten Systemen der Automobil- und Bahnindustrie gefordert (Automotive SPICE und EN 50128). Die Forderung nach sorgfältigem Requirements-Management ist nicht unbegründet. Nach PMI (2014a) ist schlechtes Requirements-Management der zweithäufigste Grund für das Scheitern von Projekten.

Zudem häuft sich bei schlechtem Requirements-Management oft toter Code an. Toter Code ist Quellcode, der nicht entfernt wird, obwohl er nicht mehr vom Programm benötigt wird. Dadurch erhöht sich die Einlesezeit und häufen sich Missverständnisse.

2.2 Traceability-Daten

Das Bindeglied zwischen Requirements und Umsetzung sind sogenannte Traceability-Daten oder Traceability-Informationen. Traceability-Daten sind wichtige Informationen zur Nachvollziehbarkeit und Verfolgbarkeit in Systemen. Sie sollen die Zuordnung von Requirements zu Artefakten über den gesamten Entwicklungsprozess eines Systems ermöglichen. Zudem sollen sie mit zur Begründung verwendet werden können, wenn es um die Frage geht, *warum* etwas im fertigen System auf eine bestimmte Art und Weise gelöst wurde. In dieser Arbeit werden die Traceability-Daten hauptsächlich zur Zuordnung von Requirements zu Quellcode verwendet.

Traceability-Daten haben, wie jede Art von Daten, intrinsische, zugriffsrelevante, kontextrelevante und darstellungsrelevante Eigenschaften (Wang, Ziad & Lee, 2002, S. 5). Aus diesen Eigenschaften gehen einige Merkmale hervor, die besonders wichtig für Traceability-Daten sind: Vollständigkeit, Darstellbarkeit, Eindeutigkeit, Gültigkeit und Konsistenz. Im Kontext dieser Arbeit bedeutet Vollständigkeit, dass die Traceability-Daten den gesamten Quellcode und alle Requirements abdecken müssen. Die Darstellbarkeit betrifft die Art, wie die Traceability-Daten für den Benutzer sichtbar gemacht werden können. Die Eindeutigkeit von Traceability-Daten besagt, dass jeder Link zwischen Quellcode und Requirement eindeutig sein muss. Gültige Traceability-Daten benötigen korrektes Änderungsmanagement. Zu keinem Zeitpunkt darf es zwei unterschiedliche Versionen von Traceability-Daten zu einem Link geben, damit die Daten konsistent sind.

In bereits bestehenden Systemen werden Traceability-Informationen häufig vernachlässigt (Larson, 2014, Why We Need to Manage Requirements), (Ramesh, 1998), da sie nicht ausreichend visualisiert werden (Winkler & Pilgrim, 2010). Ein System mit vernachlässigten Traceability-Informationen wird mit fortschreitender Zeit zunehmend fehleranfällig und damit kostspielig. In großen Systemen nachträglich Traceability einzuführen ist zeit- und kostenintensiv. Von Beginn an korrekt angelegte und mitgeführte Traceability-Daten ermöglichen Zeit- und Kostenersparnis (PMI, 2014b).

Die direkten Verbindungen zwischen Requirements und Quellcode können vom Softwarearchitekten dazu verwendet werden, die im Systementwurf angelegten Sichten auf die Softwarearchitektur aktuell zu halten. Mit aktuellen Traceability-Informationen kann verifiziert werden, ob der Quellcode dem Systementwurf entspricht und umgekehrt. Der Ingenieur kann die aktuellen Sichten und die Traceability-Daten verwenden um den Quellcode nachzuvollziehen.

2.3 Tracing Overlay

Warum sich Traceability-Daten dazu eignen, Kosten und Zeit zu sparen, und was *Tracing Overlay* im Rahmen dieser Arbeit bedeutet, verdeutlicht die folgende Erläuterung. Abbildung 2.1 visualisiert den Ausgangspunkt und das Grundproblem dieser Arbeit:

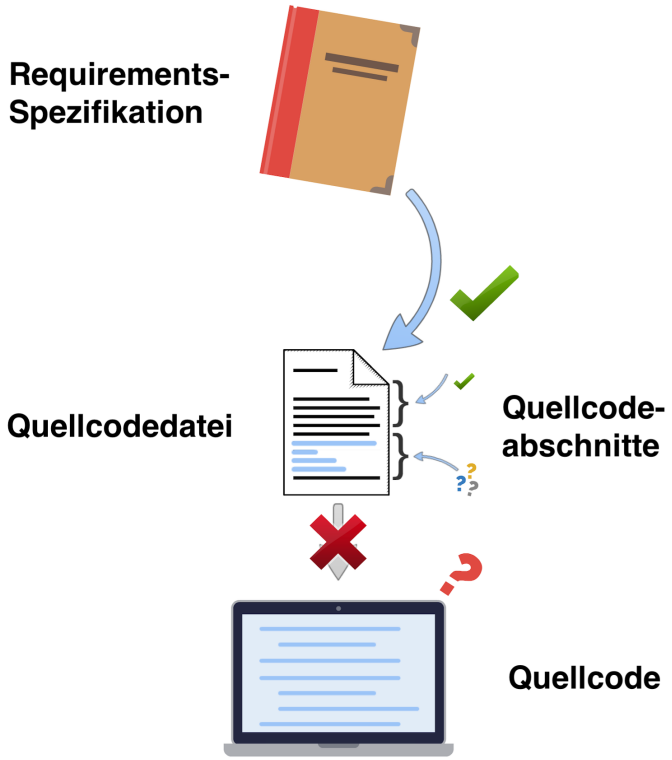

Abbildung 2.1: Lücke zwischen Quellcodedatei und Quellcode

Die Requirements aus der Spezifikation werden mit Traceability-Informationen zu Quellcodedateien zugeordnet. Nach einer (blau angedeuteten) Änderung in der Quellcodedatei, müssten die Zuordnungen jedoch aktualisiert werden. Die Traceability-Informationen sind allerdings eines der ersten nicht-funktionalen Requirements, das beispielsweise unter Zeitdruck vernachlässigt wird (Ramesh, 1998). So entstehen Quellcode-Abschnitte, die nicht mehr zur ursprünglichen Zuordnung der Requirements passen. Dadurch hängen die Requirements aus der Spezifikation nicht mehr direkt mit der Implementierung zusammen.

Mit den derzeit etablierten Tools, wie beispielsweise Rational DOORS, können bereits Traceability-Informationen angelegt werden. Das kleinste visuell zugeordnete Objekt ist dabei die Quellcodedatei. Diese etablierte Umsetzung wird oft zum Problem. Entwickler neigen häufig dazu, verschiedene Konzepte zu vermischen und so viel Funktionalität wie möglich in einer einzigen Klasse zu implementie-

ren. Sie tun dies um so schnell wie möglich neue Funktionen umsetzen zu können (Marcus & Poshyvanyk, 2005) (Moha, Gueheneuc, Duchien & Le Meur, 2009). Dadurch müssen viele überlappende Verbindungen zwischen Requirements und Quellcodedateien erstellt werden (Egyed & Grunbacher, 2004). Im schlimmsten Fall verweisen dann alle Requirements der Spezifikation auf die einzige Quellcodedatei. In diesem Fall sind die Traceability-Informationen kaum nützlich. Die entstandene Lücke in den Traceability-Informationen zwischen der Quellcodedatei und dem eigentlichen Quellcode ist das resultierende Problem.

Um auch in einem solchen Fall noch von Traceability-Informationen profitieren zu können, müssen die Traceability-Informationen nicht an der einzelnen Quellcodedatei, sondern an Quellcode-Abschnitten verankert werden. Durch die Visualisierung der Traceability-Informationen kann der Nutzer direkt von den Traceability-Daten profitieren. Mithilfe eines *Tracing Overlay*s kann das Verankern der Requirements an Quellcode-Abschnitte realisiert und visualisiert werden.

Das Tool, das Quellcode-Abschnitte einfärbt und durch die Farbe Zuordnungen zu Requirements visualisiert, wird im weiteren Verlauf dieser Arbeit als *Tracing Overlay* bezeichnet. Diese Zuordnungen werden durch farbliches Hervorheben der Quellcode-Abschnitte und der zugehörigen Requirements in der gleichen Farbe visualisiert.

Abbildung 2.2 stellt eine mögliche Darstellungsweise für Programmierer dar. Auf der linken Seite ist der exemplarische Python-Quellcode dargestellt. Die farbig markierten Abschnitte stellen die Zugehörigkeit zu einem bestimmten Requirement dar. Die zugehörigen Requirements werden auf der rechten Seite als kleine Informationsboxen in der gleichen Farbe dargestellt. So kann von Programmierern einfacher die Funktion eines Quellcode-Abschnittes nachvollzogen werden.

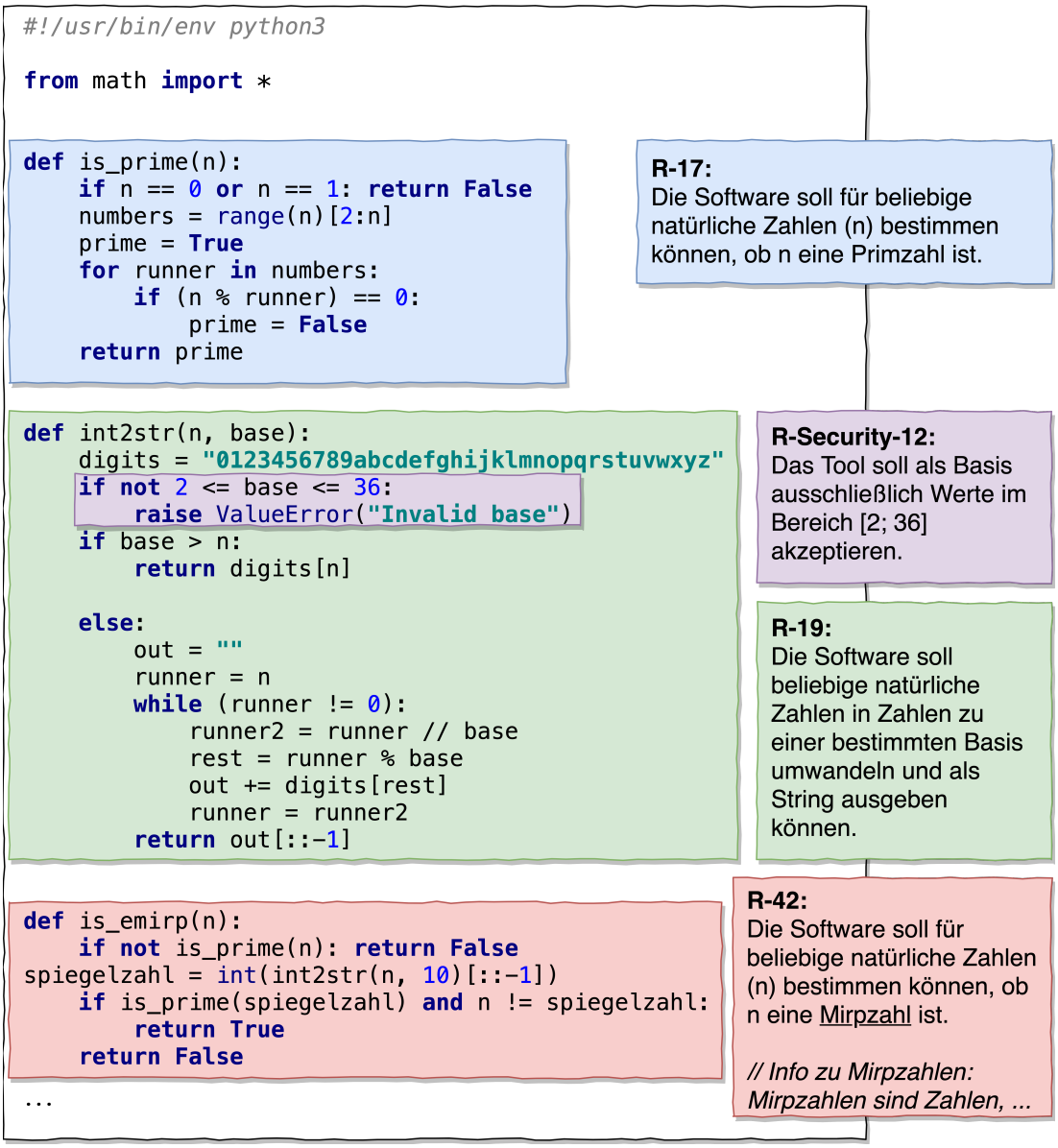

Abbildung 2.2: Exemplarischer Python-Quellcode mit Traceability-Informationen

2.4 Begriffe für die Umfrage

Die folgenden Begriffe wurden in der in Abschnitt 1.3 erwähnten Umfrage verwendet. Sie bezeichnen mögliche Erweiterungen des *Tracing Overlay*. Sie wurden nicht im Rahmen dieser Arbeit umgesetzt, aber im Hinblick auf weitere mögliche Forschung mit berücksichtigt. Die eigenen Ideen für diese Erweiterungen entstanden während der anfänglichen Überlegungen zur Entwicklung des Prototyps zur visuellen Annotation von Quellcode mit Traceability-Informationen.

Tracing Overlay für UML und SysML Diese Erweiterung unterscheidet sich vom *Tracing Overlay* (Abschnitt 2.3), das in dieser Arbeit umgesetzt wird, soll aber als mögliche Erweiterung dieser Arbeit berücksichtigt werden. Das *Tracing Overlay für UML und SysML* ermöglicht das Zuweisen von Requirements zu einzelnen UML / SysML Objekten während der Designphase. Die Zuweisung kann beispielsweise per *Drag and Drop*, oder per Stichwortsuche, oder auch per Suche nach Requirement Kategorie/Art erfolgen. Der Softwarearchitekt kann so bereits im UML / SysML Diagramm festlegen, welche Requirements in welchen Modulen realisiert werden sollen. Sollen aus dem Systementwurf in einem nächsten Schritt Codebausteine generiert werden, so können auch gleich die umzusetzenden Requirements mit eingearbeitet werden. Wenn so ein generiertes Artefakt dann implementiert werden soll, muss der Programmierer nur noch aus den bereits zugewiesenen Requirements das auswählen, welches er gerade umsetzen will. Beim Programmieren fügt er somit gleich die Traceability-Information auf niedrigster Ebene hinzu. Eine mögliche Umsetzung könnte beispielsweise wie in Abbildung 2.3 aussehen. Abbildung 2.3 wurde eigens zur Visualisierung des *Tracing Overlay*s für UML und SysML erstellt.

Tracing Info Ein Overlay, das farbige Diagramme und prozentuale Angaben in UML oder SysML Objekten, wie beispielsweise UML Klassen, erzeugen kann. Hierbei werden prozentuale Angaben über verschiedenen markierte „Requirements Klassen“ oder „Gruppen“ farblich visualisiert. Die Visualisierung kann beispielsweise mit Kreisdiagrammen erfolgen (2.3). „Requirement Klassen oder Gruppen“ sind Requirements, die eine gleiche Eigenschaft besitzen. Die zum Beispiel alle sicherheitsrelevant sind und somit in die Gruppe „Security“ fallen. Alle Requirements einer Requirement Gruppe werden zusammengefasst und als ein Kreissektor dargestellt.

White-Spot Metrik Eine Metrik, die den Anteil von Quellcode ohne Traceability-Informationen im Verhältnis zu Quellcode mit Traceability-Informationen liefert. Im Falle von Abbildung 2.3, würde diese Metrik beispielsweise auf die zu 51 % fehlenden Traceability-Informationen in der Klasse BoxBuilderImpl hinweisen.

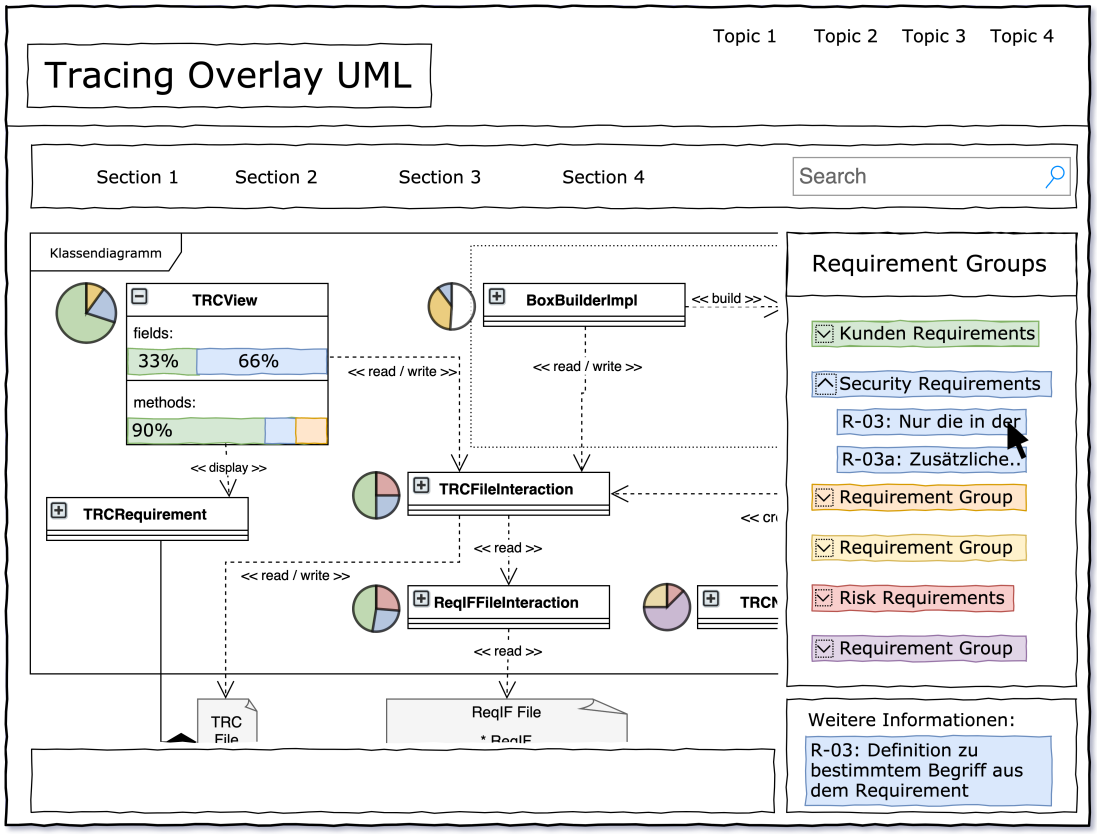

Abbildung 2.3: Beispielhafte Benutzeroberfläche für ein *Tracing Overlay für UML und SysML*

3 Verwandte Arbeiten und Stand der Technik

Mit der Notwendigkeit der Verbesserung der Visualisierung und Verwaltung von Traceability-Informationen von Projekten haben sich bereits Wissenschaftler auseinandergesetzt. Auch gibt es etablierte Tools, die Visualisierung von Abhängigkeiten unterstützen. Im Folgenden wird die vorliegende Arbeit von existierenden Ansätzen abgegrenzt und einzelne Aspekte der Arbeit werden durch bestehende Forschung begründet.

3.1 Verwandte wissenschaftliche Arbeiten

Um sicherzustellen, dass die Entwicklung eines Tools zur visuellen Annotation von Traceability-Informationen sinnvoll ist, wurden die folgenden wissenschaftlichen Arbeiten analysiert.

Tracing Requirements and Source Code During Software Development
So lautet der Titel der Dissertation von Alexander Delater (2013), die unter anderem auf das Problem der fehlenden Traceability von Software eingeht.

Alexander Delater hat in seiner Dissertation die Probleme und Stärken von Traceability wissenschaftlich dargelegt:

- Akkurate, konsistente, vollständige, aktuelle Traceability-Daten sind für verschiedene Gruppen von Stakeholdern notwendig. Aktuelle Techniken zur Verwaltung von Traceability-Verbindungen sind immer noch arbeitsaufwändig und fehleranfällig (zum Beispiel wegen schlechter Qualität der Dokumentation, Detaillierungsgrad, et cetera).
- Traceability Verbindungen können nur unter der Bedingung, die aktuellen Abhängigkeiten zwischen den Artefakten wiederzugeben, brauchbar sein. Die Verbindungen während der Entwicklung zu aktualisieren ist mühsam und folglich zerfallen die Verbindungen oft in einen ungültigen Zustand.

- Traceability Verbindungen müssen sich synchron mit ihren zugehörigen Artefakten entwickeln. Aktuelle Veränderungsmanagement-Systeme und Link Semantiken sind nicht ausreichend ausgereift um eine effektive Entwicklung von Traceability Verbindungen zu unterstützen.

Alexander Delater entwickelte ein Tool, das die Aktivitäten des Programmierers während des Programmierprozesses aufzeichnet. Sämtliche betrachtete Requirements werden dem Programmierer im Anschluss gezeigt. Dann muss der Programmierer auswählen, welche der Requirements auch tatsächlich beim Programmieren verwendet wurden.

Auf diesem Grundgerüst konnte die vorliegende Arbeit jedoch nicht aufbauen, da das entwickelte Tool nicht mehr aufzufinden ist und mehrere verwendete Programme veraltet sind. Ein solches Modell wäre eine gewünschte Grundlage für das Bindeglied zwischen Requirements und Quellcode. Der Ansatz dieser Arbeit soll nicht das bestehende Konzept verändern, er soll lediglich den Ansatz der semi-automatisierten Erfassung von Verbindungen zwischen Quellcode und Requirement um eine spezielle Art der Darstellung und Umsetzung bereichern.

Die Dissertation von Alexander Delater sieht keine Visualisierung auf Quellcode-Ebene vor und unterscheidet sich somit einerseits von dem Ansatz der vorliegenden Arbeit und belegt andererseits die Wichtigkeit von Traceability-Daten für Projekte.

Nachverfolgbarkeit zwischen Requirements und Code Die Masterarbeit von Julia Krause (2019) geht auf die Kernprobleme von Traceability ein, wie auch den Umstand, dass Traceability besonders in sicherheitsrelevanten Bereichen der Softwareentwicklung an Wichtigkeit gewinnt. In ihrer Arbeit verweist sie auf eine Studie von Lilienthal (2019) die besagt, dass Programmierer 70 % ihrer Zeit zum Verstehen des Quellcodes benötigen, 20 % auf das Lösen des Problems anfallen und nur 10 % der Zeit effektiv für das Schreiben des Quellcodes benötigt werden. Gut gepflegte Traceability-Informationen tragen insbesondere zum einfacheren Verständnis des Quellcodes bei. Somit bedeuten gut gepflegte Traceability-Informationen eine Zeit- und Kosteneinsparung (Lilienthal, 2019). Julia Krause entwickelte im Rahmen ihrer Arbeit eine eigene Anwendung, in der mehrere Ansätze zur Visualisierung von Traceability-Informationen kombiniert wurden. So werden verschiedene Darstellungsweisen kombiniert, um die Traceability-Links zwischen Quellcodedateien und Requirements wiederherzustellen (Krause, 2019).

Die Masterarbeit von Julia Krause sieht ebenfalls keine Visualisierung auf Quellcode-Ebene vor und divergiert somit vom Ansatz der vorliegenden Arbeit. Sowohl Alexander Delater, als auch Julia Krause belegen in ihren Arbeiten den Nutzen von Traceability-Informationen. Im Folgenden wird jedoch neben dem allgemeinen Nutzen auch auf den konkreten Nutzen für die verschiedenen Stakeholder

von Traceability-Informationen eingegangen.

Stakeholder von Traceability-Informationen Laut Ramesh (1998) und Ramesh und Jarke (2001) gibt es zwei unterschiedliche Interessensgruppen für Traceability-Informationen: *low-end* und *high-end* Nutzer.

Nach ihren Ergebnissen neigen *low-end* Nutzer dazu, Traceability als Belastung wahrzunehmen. Diese Belastung wird ihnen durch Vorschriften oder Kunden auferlegt. In den Augen der *low-end* Nutzer ist die Aufgabe des Aufzeichnens von Traceability-Informationen arbeitsaufwändig und nicht nützlich. Aus diesem Grund betreiben sie kaum Aufwand für das Bestimmen oder Befolgen von Richtlinien. Auch das nötige Beschaffen oder Konfigurieren von Tools oder das Anlegen von Umgebungen, in denen Traceability-Informationen effektiv benutzt werden kann, wird nicht bewältigt. Folglich benutzen *low-end* Nutzer die Traceability-Informationen nicht, was wiederum die Meinung, dass Traceability-Informationen sinnlos sind, weiter stärkt.

Die *high-end* Nutzer hingegen sind davon überzeugt, dass Traceability nützlich ist und investieren somit Zeit. Sie befassen sich mit Technologien, Strategien und Richtlinien, die zum Anlegen und zur Nutzung von Traceability-Informationen verwendet werden können. Auch widmen sie sich der Konfiguration von Tools. Außerdem benutzen sie Traceability erst nach Abwägen verschiedener Alternativen, um den größten Nutzen der Traceability-Informationen zu erhalten. Sie passen die Traceability an ihre Bedürfnisse an. Mit der Zeit sammeln *high-end* Nutzer ein Set von Erfolgserlebnissen, Erfolgsmodellen und allgemeiner Erfahrung, die sie dann benutzen, um den Traceability-Vorgang weiter zu optimieren (Ramesh, 1998) (Ramesh & Jarke, 2001).

Auch Arkley und Riddle (2005) benennen das „Traceability-Nutzen Problem“. Das Aufzeichnen der Traceability-Informationen muss idealerweise von den Ingenieuren durchgeführt werden, die direkt im Entwicklungsprozess involviert sind. Gerade diese Ingenieure scheinen aber keinen Vorteil aus dieser Mehrarbeit zu ziehen (Arkley & Riddle, 2005).

Die vorliegende Arbeit richtet sich insbesondere an den *low-end* Nutzer von Traceability-Informationen. Der Nutzer erhält direktes visuelles Feedback bei Verwendung des *Tracing Overlays*. Wodurch die Motivation zum Aufzeichnen von Traceability-Informationen bei den *low-end* Nutzern steigt. Aus den Informationen, die das *Tracing Overlay* bereitstellt, können auch *low-end* Nutzer direkt bei ihrer Arbeit profitieren und somit einen Nutzen aus den Traceability-Informationen ziehen.

A survey of traceability in requirements engineering and model-driven development Winkler und Pilgrim (2010) geben einen weiteren Überblick über bestehende Ansätze, die sich bereits mit Traceability befassen. So gibt es bestehende Ansätze zum Anlegen, Warten und Benutzen von Traceability-Informationen. Die Ansätze werden auf ihren Nutzen geprüft. So gibt es beispielsweise Ansätze, die Kosten und Nutzen der Traceability-Informationen berücksichtigen. Traceability-Informationen können positiv zum Management beitragen. Zudem kann Traceability an die jeweiligen Bedürfnisse von Projekten angepasst werden.

Insbesondere bei der Benutzung und Visualisierung von Traceability-Informationen gibt es laut Winkler und Pilgrim noch Bedarf. In bestehenden Tools können Traceability-Informationen oft keinen Nutzen beisteuern, da keine Visualisierung erfolgt (Winkler & Pilgrim, 2010). Einen möglichen Lösungsansatz zur Visualisierung von Traceability-Informationen stellt die vorliegende Arbeit vor.

Cross-Cutting Concerns Nicht modularisierbare Requirements werden auch *Cross-Cutting Concerns* genannt (Khaled, Noble & Biddle, 2003). Für die Realisierung solcher Requirements, die in vielen Softwareartefakten einzeln umgesetzt werden müssen, gibt es einen eigenen Ansatz. Dieser Ansatz wird als Aspektorientiertes Programmieren bezeichnet (Kiczales et al., 1997). Mit diesem Ansatz kann die Auswirkung eines *Cross-Cutting Concerns* durch Zentralisierung auf wenige Requirements beschränkt werden. Trotz des bestehenden Lösungsansatzes kommt es immer wieder vor, dass Requirements eine Auswirkung auf viele Quellcodedateien haben. Im schlimmsten Fall hat ein solches Requirement Auswirkungen, die sich in allen Quellcodedateien des Projekts niederschlagen. Werden ausschließlich Traceability-Daten zwischen Requirements und Quellcodedateien visualisiert, so bedeutet diese Visualisierung keinen Informationsgewinn für den Betrachter. Doch durch die visuelle Annotation von Quellcode mit Traceability-Informationen können auch diese Requirements auf eine sinnvolle Art mit Quellcode-Abschnitten verknüpft werden. So kann gezeigt werden, wo ein *Cross-Cutting Concern* im Quellcode umgesetzt wurde.

Bestehende Ansätze zum visuellen Anreichern von Quellcode Ein erster Ansatz ist das in vielen Entwicklungsumgebungen implementierte Syntax-Highlighting, Syntax-Colouring oder Syntaxhervorhebung. Bei der Syntaxhervorhebung werden vielerlei Quellcode-Elemente auf verschiedene Art farbig codiert. Ein Beispiel ist die Codierung von Kommentaren in einer anderen Farbe als Datentypen, um die visuelle Abgrenzung zu erleichtern. In mehreren wissenschaftlichen Studien wurde untersucht, ob Syntax-Highlighting einen Vorteil gegenüber dem blanken schwarz-weißen Quellcode bietet. Die Ergebnisse aus Studien von Beelders und du Plessis (2016) und Hakala, Nykyri und Sajaniemi (2006) zeigen, dass Syntax-Highlighting bei der Lesegeschwindigkeit und Suchgeschwindigkeit keinen signifikanten Vorteil oder Nachteil zu schwarz-weißem Quellcode bietet.

Die Vorlieben für bestimmte Arten der Syntaxhervorhebung waren dabei vom jeweils befragten Individuum abhängig.

Bei Syntax-Highlighting wird häufig die Schriftfarbe des Quellcodes und nicht der Hintergrund eingefärbt. Somit sollte das in der vorliegenden Arbeit entwickelte Tool den Hintergrund einfärben, um das etablierte Syntax-Highlighting nicht zu stören. So können beide Ansätze zur Visualisierung auf Quellcode-Ebene zugleich benutzt werden.

Ein weiterer Ansatz ist die Markierung von Quellcode während der Ausführung eines Programms, um das Debugging von Quellcode zu erleichtern. Brandt, Pattamatta, Choi, Hsieh und Klemmer (2010) geben ein Beispiel, bei dem kürzlich ausgeführter Programmcode dunkelgrün eingefärbt wird. Wenn die Ausführung des Programms weiter fortschreitet, verblasst die Farbe der länger nicht mehr ausgeführten Programmcode-Abschnitte. Somit lässt sich der Verlauf eines Programms während des Debuggings leichter nachvollziehen. Das Markieren von Programmcode während der Ausführung ist ein gänzlich verschiedenes Konzept zum Gegenstand der vorliegenden Arbeit. Da nicht in der Debugging View programmiert wird, stellt der ähnliche Ansatz des Einfärbens der Hintergrundfarbe keinen kollidierenden Ansatz dar. Beim Debugging könnte beispielsweise das in der vorliegenden Arbeit entwickelte Tool deaktiviert und danach wieder aktiviert werden.

Eine gänzlich andere Art der Darstellung von Informationen im Quellcode stellt die Masterarbeit von Roque (2007) vor. Die dort entwickelte Software *OpenBlocks* ist ein Open Source Projekt, bei dem Programmierern eine Art Puzzle-Baukasten zur Verfügung gestellt wird. Die einzelnen Puzzleteile repräsentieren Programminstruktionen. Mit Projekten wie *OpenBlocks* sollen besonders Nachwuchsprogrammierer gefördert werden. Da typischerweise eine sehr farbenfrohe Art der Darstellung der verschiedenen Puzzleteile verwendet wird, wäre eine Kombination von *OpenBlocks* und der visuellen Annotation von Quellcode mit Traceability-Informationen nicht sinnvoll umsetzbar. Die Zielgruppe von *OpenBlocks* entspricht jedoch nicht der Zielgruppe des in dieser Arbeit entwickelten Tools. *OpenBlocks* gibt den einzelnen Puzzleteilen durch die Farbzuweisung einen erhöhten Wiedererkennungswert. Diese Erhöhung des Wiedererkennungswertes durch Farbzuweisung lässt sich auch auf die vorliegende Arbeit übertragen.

Schnelleres Suchen dank farbiger Boxen Um Aussagen über die Suchgeschwindigkeit treffen zu können werden die folgenden Punkte angenommen: Dem Programmierer ist die Farbe des Requirements bekannt, nach der er im Quellcode suchen muss. Er muss durchschnittlich fünf Wörter lesen, um einen Quellcodeabschnitt zu einem Requirement zuordnen zu können. Unter diesen Annahmen gilt:

Die durchschnittliche Lesezeit für fünf Wörter liegt bei 1,4 Sekunden (Brysbaert, 2019). Die durchschnittliche Erkennungszeit für ein farbiges Objekt liegt dagegen bei 0,5 Sekunden (Quinlan & Humphreys, 1987). Somit ist durch Farbcodierung von Quellcode-Abschnitte eine schnellere Zuordnung möglich. Mit Farbcodierung von Quellcode-Abschnitten wird durchschnittlich 30 % der Zeit benötigt, die ohne Farbcodierung zum Zuordnen nötig wäre. Die Zeit, die benötigt wird um den Text zu verstehen, wird dabei nicht beachtet.

Durch die Visualisierung der Traceability-Informationen ist für den Nutzer eine kürzere Suchzeit von Quellcode-Abschnitten, die zu bestimmten Requirements gehören, zu erwarten.

3.2 Verwandte Tools

Für Requirements-Management gibt es einige etablierte Tools, zum Beispiel Rational DOORS, Eclipse Capra, oder ReqIF Studio. Im Folgenden werden Tools, die sich im Rahmen des Requirements-Management bereits mit Traceability-Informationen befassen kurz vorgestellt.

Eclipse Capra ist ein Open Source Tool zum Requirements-Management.[1] Es erlaubt die Visualisierung von Abhängigkeiten zwischen verschiedenen Requirements und zwischen Requirements und Codedateien. Die kleinste nachverfolgte Einheit ist somit die Quellcodedatei.

Rational DOORS wurde von IBM Corp. (2014) aufgekauft und ist ein etabliertes kommerzielles Tool für Requirements-Management (IBM Corp., 2014). Es ermöglicht die Verwaltung von Requirements im Dokumentenstil. In einer tabellarischen Übersicht werden die Requirements angezeigt und können vom Nutzer bearbeitet werden. Auch das Anlegen von Verbindungen zwischen den Requirements und Artefakten, wie anderen Requirements oder Quellcodedateien, ist möglich.

ReqIF Studio ist ein kostenloses Tool zur Verwaltung von Requirements im gängigen Austauschformat für Requirements (ReqIF). Auch hier können Links zwischen Requirements und Quellcodedateien erstellt werden (Jastram, 2017). Im grundlegenden Aufbau ähnelt dieses Tool Rational DOORS.

[1] Eclipse Capra. Verfügbar unter https://projects.eclipse.org/projects/modeling.capra

YAKINDU Traceability ist ein kommerzielles von der itemis AG entwickeltes Tool zum Requirements-Management in Projekten.[2] Dabei werden Requirements an ein Softwareprojekt auf abstrakten Ebenen, wie zum Beispiel in UML Views, verwaltet. YAKINDU Traceability erlaubt das Zuweisen von Quellcode-Abschnitten zu Requirements. Allerdings wurde noch keine Visualisierung umgesetzt, die dem Tool der vorliegenden Arbeit gleicht. Eine Integration des in dieser Arbeit programmierten Tools, in eine Umgebung wie YAKINDU Traceability, ist denkbar.

3.3 Vergleich bestehender Ansätze und Stand der Technik

Die Tools sind zum Teil grundlegend unterschiedlich in ihrer Art, die Traceability-Daten zu visualisieren. Aber alle vorgestellten Tools unterstützen das einheitliche ReqIF Datenformat für Requirements-Spezifikationen.

Das Hinzufügen dieser Daten während der Entwicklung erfordert einerseits erhöhten Entwicklungsaufwand. Andererseits gilt, dass Traceability den Softwareentwicklungsprozess auf verschiedene Weise unterstützt. Unter anderem zum Veränderungsmanagement, zur Wartung von Software und zur Vermeidung von Missverständnissen trägt Traceability bei. Verbindungen zwischen Requirements und Quellcode sind wichtig zur Unterstützung dieser Entwicklungsaktivitäten, zum Beispiel für das Navigieren von einer Anforderung zu ihrer Umsetzung im Quellcode, und umgekehrt (Delater, 2013).

Trotz der erwiesenen Wichtigkeit gibt es bis heute keinen konkreten wissenschaftlichen Ansatz zur Visualisierung von Traceability-Informationen auf Quellcode-Ebene.

Der in dieser Arbeit vorgestellte und entwickelte Prototyp schließt mit der Markierung von Traceability-Informationen auf Quellcode-Ebene die Lücke zwischen Quellcodedatei und Quellcode. Diese Lücke kann beispielsweise bei nicht dokumentierten Änderungen im Quellcode entstehen. Diese nicht dokumentierten Änderungen haben ihren Ursprung wiederum in der geringen Motivation der *low-end* Nutzer, Traceability-Informationen zu aktualisieren. Der Ansatz der vorliegenden Arbeit gibt auch dem *low-end* Nutzer der Traceability-Daten einen Grund, sich aktiv am Traceability-Prozess zu beteiligen. Mit dem Prototyp dieser Arbeit kann er die Traceability-Informationen sehen und somit als Informationsquelle nutzen.

Die Vorteile der visuellen Annotation von Requirements im Quellcode sind unter anderem: Die schnellere Suche von zu Requirements zugehörigem Quellcode. Die-

[2] YAKINDU Traceability – itemis' professional traceability tool. Verfügbar unter https://www.itemis.com/en/yakindu/traceability/

se schnellere Suche ist durch Möglichkeit begründet, beim Lesen von Quellcode Abschnitte überspringen zu können. Des Weiteren die Motivation des *low-end* Nutzers zum Anlegen von Traceability-Informationen, da er auch selbst direkt davon profitieren kann. Zudem ist ein leichteres Auswechseln von ganzen Systemkomponenten und eine günstigere Wartung möglich. Darüber hinaus ist ein verringerter Einleseaufwand belegt. Die visuelle Annotation von Requirements auf Quellcode-Ebene stellt auch eine mögliche sinnvolle Art der Visualisierung von „Cross-Cutting Concerns“ dar. Diese Vorteile sind zusätzliche Motivation für die vorliegende Arbeit.

4 Requirements

Aus den vorangegangenen Kapiteln werden die Requirements für diese Arbeit abgeleitet. Diese Requirements umfassen die Zielgruppe, den Zweck, die Anwendungsfälle und die Requirements für das Tool.

4.1 Stakeholder-Analyse

Das Tool soll vor allem die Arbeit der Programmierer erleichtern. Der Programmierer nimmt somit die Rolle des Kunden dieses Projektes ein. Es gibt aber auch Nutzen für Softwarearchitekten und Projektmanager, weswegen diese Gruppe von potentiellen Kunden nicht explizit ausgeschlossen wird. Für zukünftige Entwickler ist insbesondere die Dokumentation des für den Prototyp entwickelten Quellcodes wichtig.

Der Prototyp wird nicht zum fertigen Einsatz in einem großen System implementiert, sondern soll zeigen, ob eine Umsetzung der visuellen Annotation von Quellcode mit Traceability-Informationen auf diese Art Sinn ergibt. Somit werden zukünftige Stakeholder wie Betreiber, Qualitätsbeauftragte, das Management oder etwaige Konkurrenten im Rahmen dieser Arbeit nicht berücksichtigt.

Der Prototyp erleichtert dem Programmierer das Einlesen in Quellcode. Das Anzeigen der wichtigsten Informationen zu den Requirements ermöglicht dem Programmierer einen Überblick über die relevanten Requirements zu erhalten, ohne die Entwicklungsumgebung (IDE) verlassen zu müssen. Der Programmierer sieht hier auch auf einen Blick, welches Requirement in welcher Farbe markiert wird. Die farbliche Markierung von Requirements im Quellcode erleichtert die Navigation. Der Nutzer kann zu den Stellen im Quellcode springen, die gerade von Interesse sind. Ähnlich wie bei einem mit Textmarker farblich annotierten Text. Ohne zuerst den Quellcode lesen zu müssen, können durch die visuelle Zuordnung eingefärbte Quellcode-Abschnitte beim Lesen übersprungen werden. Insbesondere dem Programmierer gänzlich unbekannter Quellcode kann durch die visuellen Annotationen schneller verstanden werden.

Mögliche Erweiterungen des in dieser Arbeit realisierten Tools richten sich haupt-

sächlich an Softwarearchitekten und Projektmanager. Um die Erweiterung zu einem späteren Zeitpunkt zu erleichtern, werden diese Stakeholder nicht ausgeschlossen. Der potentielle Nutzen für Softwarearchitekten und Projektmanager besteht auch in einer denkbaren *White-Spot Metrik* (Abschnitt 2.4). Eine *White-Spot Metrik* kann mit den vom Tool angelegten Traceability-Daten den Anteil von „nicht zu Requirements zugeordnetem Code“ in Quellcodedateien bestimmen. Die Visualisierung kann in UML Tools, wie dem *Tracing Overlay für UML und SysML* (Abschnitt 2.4) verwendet werden. Der Softwarearchitekt kann den Klassen die dort umzusetzenden Requirements visuell zuweisen. Bei der Generierung von Codegerüsten aus UML Diagrammen ist danach eine Übernahme der zugeordneten, umzusetzenden Requirements möglich. Mittels *Tracing Info* (Abschnitt 2.4) können Informationen aus den mit Traceability-Informationen versehenen Quellcodedateien abgeleitet und für Softwarearchitekten und Projektmanager dargestellt werden.

Der wichtigste Stakeholder für diese Arbeit ist der Programmierer als Kunde und Nutzer des Tools.

4.2 Aufgabe des Prototyps

Die Aufgabe des resultierenden Tools ist die visuelle Annotation von Quellcode mit Traceability-Informationen. Das bedeutet, den Nutzer bei der Zuordnung von Requirements zu Code-Abschnitten zu unterstützen und diese Zuordnungen zu visualisieren. Die erwarteten Vorteile sind unter anderem Zeitersparnis beim Einlesen in einen auf diese Art annotierten Quellcode, Förderung der Übersicht und Verbesserung der Langlebigkeit und Wiederverwendbarkeit des Quellcodes.

In vielen Softwareprojekten wird Dokumentation nicht ausreichend relevant eingestuft, um fortlaufend aktualisiert zu werden (Larson, 2014). Durch das visuell ansprechende Darstellungsformat wird Programmierern ein zusätzlicher Anreiz gegeben, das Hinzufügen und Aktualisieren von Traceability-Informationen zum Code nicht zu vernachlässigen.

Der visuelle Anreiz wird insbesondere durch die Anzeige von zusätzlichen Informationen gegeben. Beim Programmieren nützliche Informationen sind beispielsweise die Identifikationsnummern (IDs) und grundlegende textuelle Information zu den Requirements.

Das Anlegen dieser Informationen erfolgt im Tool. Es kann während des Programmierens erfolgen. Zudem können Nachträglich Traceability-Informationen durch Auswählen, Markieren von Quellcode und Setzen der Requirements angelegt werden.

Zur besseren Übersichtlichkeit werden nicht alle Requirements des gesamten Pro-

jekts angezeigt. Lediglich die Anzeige von in der Quellcodedatei verwendeten Requirements ist erwünscht. Die Bestimmung der Requirements, die angezeigt werden sollen, erfolgt durch den Nutzer.

Die Daten für die Requirements (IDs und textuelle Kurzbeschreibung) sollen aus Dateien im gängigen Standard für Requirements-Management ReqIF (OMG, 2016) bezogen werden können. Die Verwendung von ReqIF als bereits etabliertes Standartformat für Requirements-Management ist naheliegend.

Diese Aufgaben eines Tools zur visuellen Annotation von Quellcode mit Traceability-Informationen werden zu Anwendungsfällen zusammengefasst und im folgenden Abschnitt aufgezählt.

4.3 Anwendungsfälle des Prototyps

Die dargelegten Überlegungen (Abschnitt 4.2) ermöglichen die Bestimmung der folgenden Anwendungsfälle (UC). Zu diesem Zeitpunkt der Entwicklung der vorliegenden Arbeit wird auch ein erstes visuelles Beispiel (Abschnitt *2.3*) erstellt.

UC 1: Der Programmierer möchte markierten Quellcode-Abschnitten visuell Requirements zuordnen.

UC 2: Der Programmierer möchte die Zuordnungen zwischen Quellcodeabschnitten und Requirements modifizieren.

UC 3: Der Programmierer möchte Informationen zu den Requirements erhalten.

UC 4: Der Programmierer, Softwarearchitekt oder Projektmanager möchte Requirements zu Quellcodedateien zuordnen.

Um die Umsetzung zu ermöglichen werden diese Anwendungsfälle weiter konkretisiert.

4.4 Requirements an den Prototyp

Aufgrund der vorhergehenden Anwendungsfälle (Abschnitt 4.3) können die folgenden Requirements an das zu entwickelnde Tool aufgestellt werden.

Um die Umsetzung der Requirements evaluieren zu können, werden konkrete Abnahmekriterien (AK) für einige Requirements formuliert.

Funktionale Requirements

Nachfolgend werden die funktionalen Requirements vorgestellt, die für die Funktion des Tools im Sinne der Use Cases (Abschnitt 4.3) essentiell sind.

Informationen zu den Requirements

Das Erhalten von Informationen zu den in der Quellcodedatei relevanten Requirements ist ein Wunsch des Nutzers. Aus diesem Wunsch resultieren die folgenden Requirements.

R-01: Die IDs und die textuellen Kurzbeschreibungen der einer Quellcodedatei zugeordneten Requirements sollen in einem zusätzlichen Fenster angezeigt werden.
Begründung: UC 3

R-02: Die Zuordnung von Requirements zu einer Quellcodedatei soll mit dem Tool durch den Nutzer erfolgen können.
Begründung: UC 4

R-03: Die textuelle Kurzbeschreibung der Requirements soll aus einer Datei im gängigen Standardformat für Requirements-Management, ReqIF bezogen werden.
Begründung: UC 3

R-04: Das zusätzliche Fenster soll Knöpfe zum Auslösen von Funktionen des Tools enthalten.
Begründung: UC 1, UC 2, UC 4

Boxen

Diese folgenden Requirements definieren die Art der Darstellung der Annotationen im Quellcode.

R-05: Die Bereiche im Quellcode, die mit Traceability-Informationen annotiert sind, werden durch rechteckige Boxen repräsentiert.
Begründung: UC 1

R-06: Boxen sollen durch Selektion von Quellcode durch den Nutzer und anschließenden Knopfdruck angelegt werden können. Dabei sollen die Boxen automatisch vom Tool zu ausgewählten Requirements zugeordnet werden.
AK: Auf Knopfdruck sollen zur derzeitigen Selektion im Quellcode Boxen angelegt werden, entsprechend der im zusätzlichen Fenster ausgewählten Requirements.
Begründung: UC 1, UC 2

R-07: Eine Box soll genau einem Requirement zugeordnet sein.
Begründung: UC 1

R-08: Die Boxen sollen beliebige Start und Endpunkte im Quellcode haben können.
AK: Die Boxen sollen vom ersten Zeichen der Quellcodedatei bis zum Ende der Quellcodedatei reichen können. Alle Intervalle, die dazwischen liegen, sind explizit *eingeschlossen*.
Begründung: UC 1

R-09: Die Boxen sollen beim ersten Zeichen, das kein Leerzeichen oder ein Tabulator ist, in der ersten Zeile der Box beginnen.
AK: Führende Leerzeichen und Tabulatoren sollen zu einer Einrückung der Box führen.
Begründung: UC 1

R-10: Die Boxen sollen beim letzten markierten Zeichen einer Zeile, das kein Leerzeichen oder Tabulator ist, enden.
AK: Leerzeichen oder Tabulatoren am Ende einer Zeile sollen die Box nicht verlängern.
Begründung: UC 1

R-11: Boxen sollen eine Mindestgröße von einem Zeichen haben.

R-12: Die Boxen sollen sich automatisch an die maximale Breite des eingeschlossenen Textes anpassen.
AK: Wenn die erste Zeile länger als die zweiten und dritte Zeile ist, muss die Box die Breite der ersten Zeile annehmen. Entsprechendes muss für eine längere zweite, bzw. dritte Zeile gelten.
Begründung: UC 1

Änderungen

Die folgenden Requirements sind durch den Wunsch des Nutzers begründet, Traceability-Informationen bereits während des Programmierens bearbeiten zu können.

R-13: Bei einer Änderung im Quellcode sollen die von der Änderung betroffen Boxen automatisch vom Tool angepasst werden.
AK: Beim Löschen von Quellcode müssen vollständig in der Änderung enthaltene Boxen gelöscht werden und teilweise enthaltene Boxen verkürzt werden. Boxen, die durch das Löschen nahtlos aneinander angrenzen, müssen fusioniert werden. Beim Hinzufügen von Quellcode müssen die Boxen entsprechend der im zusätzlichen Fenster ausgewählten Requirements erweitert oder gesplittet werden.
Begründung: UC 2

R-14: Gibt es noch keine Box an der Stelle der Änderung, so sollen neue Boxen angelegt werden, entsprechend der im zusätzlichen Fenster ausgewählten Requirements.
Begründung: UC 2

R-15: Wird eine Box durch eine Änderung auf eine Länge kleiner gleich 0 reduziert, so soll die Box entfernt werden.
Begründung: UC 2, R-11

Visuelle Zuordnung

Die Requirements unter diesem Stichwort konkretisieren die visuelle Zuordnung von Requirements zu Quellcode-Abschnitten.

R-16: Für die visuelle Zuordnung zwischen Boxen und Requirements wird das Medium *Hintergrundfarbe* gewählt.
Begründung: UC 1

R-17: Das Tool soll farbige Boxen im Editor der IDE anzeigen können.
Begründung: UC 1

R-18: Angezeigte Requirements und die zugehörigen Boxen müssen in der gleichen Farbe gekennzeichnet werden.
Begründung: Dies ermöglicht dem Nutzer eine visuelle Zuordnung zwischen Requirements und Quellcode-Abschnitten herzustellen, wie in UC 1 gefordert.
Beispiel: Abbildung 2.2.

R-19: Die Farbe von Requirements und deren zugehörigen Boxen soll vom Nutzer geändert werden können.
Begründung: UC 1

Nicht-funktionale Requirements

Es folgen nicht-funktionale Requirements, die Richtlinien für Useability, Wartbarkeit, Erweiterbarkeit und Performanz des resultierenden Tools definieren.

R-20: Wenn ein neues Requirement zu einer Quellcodedatei hinzugefügt wird, soll dem Requirement vom Tool einmalig eine zufällige Farbe zugewiesen werden.
Begründung: Dieses initiale Zuweisen erleichtert dem Nutzer den sofortigen Einstieg und verringert den Einrichtungsaufwand. Es fördert die Useability.

R-21: Die Farbe eines Requirements soll nach dem ersten Zuweisen gespeichert werden.
Begründung: So kann der Wiedererkennungswert der Requirements gesteigert werden, was der Useability zuträglich ist.

R-22: Die zufällige, einem hinzugefügten Requirement zugewiesene Farbe (siehe R-20) soll eine Pastellfarbe sein.
Begründung: Pastellfarben sind hellere Farben, die den Nutzer beim Lesen des Quellcodes nicht behindern sollen. Werden zufällig beliebige Farben ausgewählt, könnte beispielsweise auch Schwarz gewählt werden, was den Quellcode unlesbar machen würde. Durch die Verbesserung der Lesbarkeit mit diesem Requirement kann die Useability verbessert werden.

R-23: Das Tool soll den Benutzer motivieren, ebendieses benutzen zu wollen. Dazu soll es die Benutzeroberfläche des Nutzers visuell bereichern.
AK: Eine anonyme Studie soll zeigen, dass mindestens 75 % der Teilnehmer das Tool nach einer dreiwöchigen Eingewöhnungszeit weiterhin regelmäßig benutzen.
Begründung: Diese Studie ist im Rahmen der Entwicklung dieses Prototyps nicht geplant. Das Requirement soll aber trotz Entwicklung eines *Prototyps* nicht außer acht gelassen werden, um eine gute Useability, Performanz und grundlegende Verlässlichkeit zu fördern.

R-24: Die Antwortzeiten für die unterschiedlichen Funktionen des Tools sollen jeweils maximal 0,5 s betragen.
AK: Dieses Requirement gilt als erfüllt, wenn die Funktionen des Prototypen zu den Requirements R-06, R-13 und R-19 diese Antwortzeit einhalten können.
Begründung: Der Nutzer soll nicht durch Wartezeiten in der Benutzung eingeschränkt sein. Da das entwickelte Tool ein Prototyp ist, wird hier ein relativ großer Wert gewählt. Ein Grundmaß an Performanz wird jedoch somit sichergestellt.

R-25: Implementierter Quellcode dieses Prototyps soll dokumentiert sein.
AK: Die Dokumentation soll sowohl durch Dokumentationskommentare realisiert sein, als auch durch visuelle Annotation des Quellcodes dieses Tools mit Traceability-Informationen.
Begründung: Der zukünftige Entwickler wünscht eine einfache Erweiterbarkeit und Wartbarkeit.

R-26: Das Tool soll unabhängig von der Programmiersprache sein, in welcher der Programmierer arbeitet.
AK: Dieser Prototyp soll mit Java- und Python-Quellcodedateien arbeiten können.
Begründung: Die Unabhängigkeit von der Programmiersprache erhöht die Kompatibilität des Prototyps.

Im Verlauf der vorliegenden Arbeit zur Entwicklung eines ersten Prototyps zur visuellen Annotation von Quellcode mit Traceability-Informationen wird auf einige Requirements bewusst verzichtet. So bestehen keine Erwartungen an die Sicherheit, Portierbarkeit, wie auch die Verlässlichkeit und die Wartbarkeit dieses ersten Prototyps. Im Fokus stehen die oben aufgeführten Requirements zu Funktionalität, Useability und Kompatibilität. Useability bedingt des Weiteren auch ein Minimum an Performanz. Bei der Entwicklung eines Produkts, das über einen Prototyp hinaus geht, sind die hier vernachlässigten Qualitätsanforderungen obligatorisch.

Evaluationsschema für die Requirements

Für die Evaluation ist zunächst vorgesehen, mit den vorgestellten Requirements einen Prototyp zu implementieren. Der Prototyp soll anschließend in dokumentierten Tests auf Einhaltung der Requirements getestet werden. Für jedes Requirement sollen Tests durchgeführt werden, die das Tool auf Erfüllung des Requirements untersuchen. Zusätzlich wird eine Umfrage (vgl. Abschnitt 1.3) durchgeführt. Die Ergebnisse der Umfrage sollen zeigen, wie groß das Interesse an einem solchen Tool ist und in wie weit die Requirements sinnvoll gestellt sind. Darüber hinaus wird in der Umfrage um eine Einschätzung zum möglichen Mehraufwand während des Programmierens gebeten. Auch eine Abschätzung zur möglichen Einleseersparnis wird abgefragt, um die Erwartungen an das Tool ermitteln zu können.

Um diese Requirements an die Arbeit zur visuellen Annotation von Quellcode mit Traceability-Informationen in einer Software zu verwirklichen, ist eine grundlegende Planung der Architektur der Software nützlich.

5 Architektur und Entwurf

Dieses Kapitel beinhaltet die Softwarearchitektur und den Entwurf des Prototypen zur Annotation von Requirements im Quellcode, der im Rahmen dieser Arbeit entwickelt wird.

5.1 Kontextabgrenzung des Tools

Zunächst wird mittels einer statischen Kontextsicht (Abb. 5.1) gezeigt, welche Personen und welche Software in Interaktion mit dem *TRC Overlay* stehen.

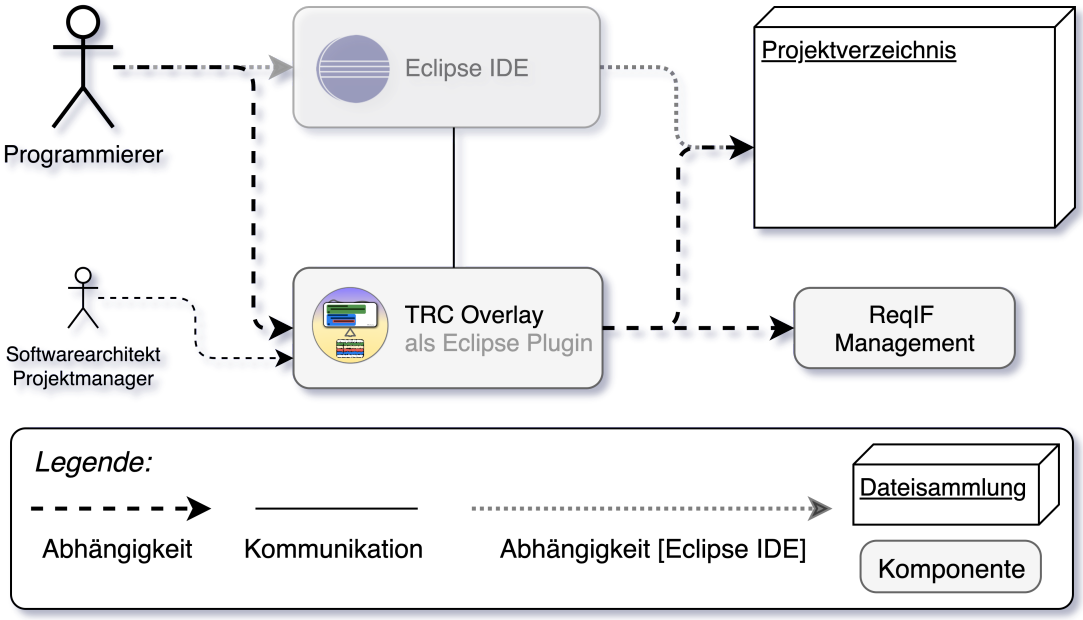

Abbildung 5.1: Kontextsicht auf die Softwarearchitektur

Die Interaktionspartner des *TRC Overlays* sind primär Programmierer und sekundär Softwarearchitekten und Projektmanager. Das *TRC Overlay* ist als Plugin in die *Eclipse IDE* integriert und kann somit auf Benutzereingaben in der IDE reagieren. Das Tool bezieht textuelle Informationen aus einer *ReqIF Datei*,

die nicht von diesem Tool verwaltet wird. Des Weiteren interagiert das Tool mit Daten im Projektverzeichnis eines *Eclipse IDE* Projekts.

Aus den in Kapitel 4 gebildeten Requirements werden im nächsten Schritt funktionale Gruppen gebildet und nach der Component-Responsibility-Collaboration (CRC) (nach Beck, Cunningham & Services, 1989) mit Responsibilities, also Verantwortlichkeiten, und Collaborations, also Abhängigkeiten, angereichert.

Die Software soll in den Komponenten *Traceability-Data Interaction*, *Tracing Overlay*, *TRC View* und *TRC New Wizard* umgesetzt werden. *TRC* steht dabei für *Traceability-Daten*. Nachfolgend werden die Komponenten als CRC Karten dargestellt.

Traceability-Data Interaction	
Responsibilities:	Collaborations:
• Lesen und Schreiben von Traceability-Informationen • Lesen des ReqIF Datensatzes, der dem Projekt zu Grunde liegt	• keine

Tracing Overlay	
Responsibilities:	Collaborations:
• Erstellen von farbigen Boxen • Anzeigen von farbigen Boxen	• *Traceability-Data Interaction*

TRC View	
Responsibilities:	Collaborations:
• Lesen von Traceability-Informationen • Anzeige der gelesenen Informationen zu den Requirements	• *Traceability-Data Interaction*

TRC New Wizard	
Responsibilities:	Collaborations:
• Anlegen von Traceability-Informationen • Bearbeiten von Traceability-Informationen	• *Traceability-Data Interaction*

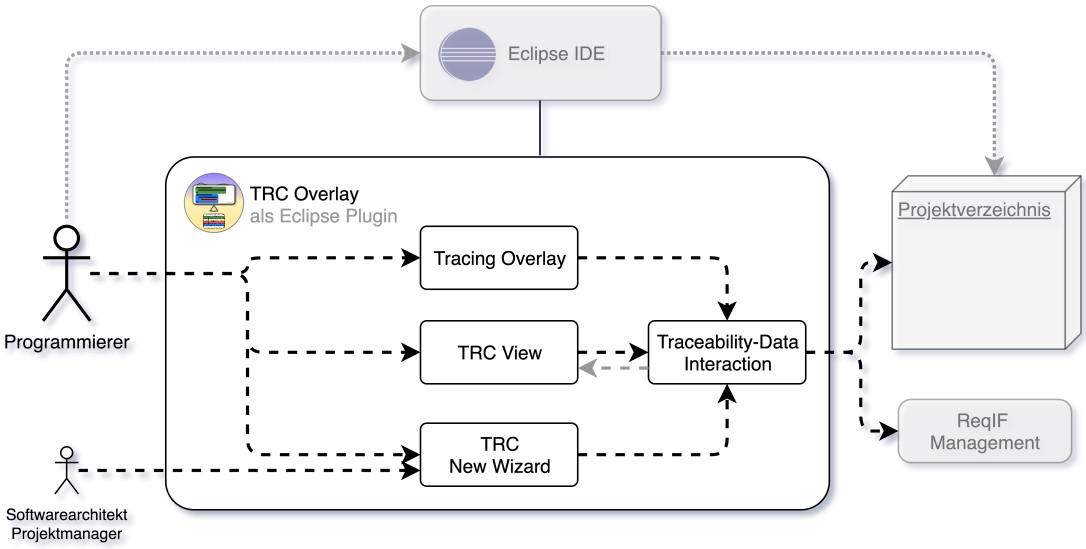

Abbildung 5.2: Logische Komponenten der Tools mit externen Interaktionspartnern

Die funktionalen Gruppen der CRC Karten und deren Abhängigkeiten sind in Abbildung 5.2 in Zusammenhang mit dem *TRC Overlay* dargestellt. Der Programmierer kann mit *Tracing Overlay*, *TRC View* und *TRC New Wizard* interagieren. Der Softwarearchitekt oder Projektmanager kann die Traceability-Informationen beeinflussen. Er kann mit Hilfe des *TRC New Wizard* Requirements zu einer Quellcodedatei zuweisen. Die einheitliche Schnittstelle für das Schreiben und Lesen der Traceability-Informationen wird in der Komponente *Traceability-Data Interaction* realisiert. Die graue Abhängigkeit zwischen *Traceability-Data Interaction* und *TRC View* wird aus Performanz-Gründen hinzugefügt.

Die Schnittstelle zwischen Programmierer und *Tracing Overlay* ist der Editor der *Eclipse IDE*. Die Schnittstelle zwischen Programmierer und *TRC View* ist ein Menü, in dem der Benutzer durch Knöpfe den Inhalt der View manipulieren kann. Die Schnittstelle zwischen Programmierer, bzw. Softwarearchitekt, oder Projektmanager und *TRC New Wizard* ist die Benutzeroberfläche des Wizard, in dem vom Benutzer Daten abgefragt und in Systemdaten übersetzt werden.

Die *Traceability-Data Interaction* Komponente stellt Methoden zum Manipulieren der Traceability-Daten bereit. Diese können von den anderen Komponenten verwendet werden. Die *Traceability-Data Interaction* Komponente greift auf die Traceability-Daten, die im Projektverzeichnis des Eclipse Projekts liegen, zu. Darüber hinaus bezieht sie textuelle Informationen aus dem ReqIF Datensatz für dieses Projekt.

5.2 Speicherung der Traceability-Daten

Um die Sichten auf die Softwarearchitektur weiter vervollständigen zu können, wird eine weitere Entscheidung, die spätere Umsetzung betreffend, getroffen. Die Requirements und die zugehörigen Boxen müssen gespeichert werden. Die Requirements für diese Arbeit (Abschnitt 4.4) fordern, dass nur die in der aktuellen Datei relevanten Requirements verwaltet werden. Dabei soll die Antwortzeit der Software kurz gehalten werden um eine flüssige Bedienung zu gewährleisten.

Zur möglichen Umsetzung der Speicherung werden im Folgenden zwei mögliche Ansätze betrachtet.

***Ansatz 1:* Magische Anmerkungen im Code** Magische Anmerkungen bezeichnen die Speicherung von Traceability-Informationen in Form von speziellen Kommentaren. Diese Kommentare werden von der Software erkannt und in Daten zur Darstellung der Boxen übersetzt.

Die Vorteile dieses Ansatzes umfassen:

- Die Traceability-Informationen sind in der gleichen Datei gespeichert. Somit sind sie leicht mit bereits bestehenden Versionskontrollsystemen und Austauschplattformen wie *git* (sh. Chacon & Straub, 2014) verwaltbar.
- Der Anwender ist bei diesem Ansatz nicht an die IDE, in der das Tool installiert ist, gebunden. Änderungen am Quellcode können somit problemlos in einem fremden Editor durchgeführt werden, ohne die Traceability-Informationen zu kompromittieren. Auch könnten die Traceability-Informationen selbst bearbeitet werden. Die magischen Anmerkungen müssen hierfür händisch eingefügt werden, dazu müssen sie für Menschen lesbar sein.
- Die Beständigkeit der gespeicherten Traceability-Informationen in diesem Ansatz ist automatisch gewährleistet.

Nachteile dieses Ansatzes sind:

- Die magischen Kommentare müssen für unterschiedliche Programmiersprachen einzeln definiert werden.

- Die magischen Kommentare tragen nicht zum eigentlichen Ziel bei, die Programmieroberfläche übersichtlicher zu machen. Beispielsweise können mehrere unterschiedlich lange, sich überlappende Boxen in einer einzelnen Zeile nicht leicht durch magische Kommentare umgesetzt werden. Die Zeile würde bereits bei einer geringen Anzahl von Requirements mehr Kommentare als Quellcode enthalten.
- Die Speicherung der zur Datei zugeordneten Requirements ist nicht inbegriffen und muss separat behandelt werden.

Ansatz 2: **Die Verwaltung der Information in einer separaten Datei**
Dies ist der zweite denkbare Ansatz zur Umsetzung der Fragen. Diese separate Datei wird im weiteren Verlauf als *Tracingdatei* bezeichnet. Separate Dateien mit gleichem Dateinamen aber unterschiedlicher Dateiendung werden auch als „Sidecar-Dateien“ bezeichnet und häufig in der Bildverarbeitung verwendet (Dyer et al., 2017), (Gerber-Menz, 2014).

Vorteile dieser Variante sind:

- Die Speicherung der Requirements, die einer Quellcodedatei zugeordnet sind, ist inbegriffen.
- Die Unabhängigkeit von jeglicher Programmiersprache ist gewährleistet.
- Die Traceability-Daten sind auch über *git* verwaltbar. Durch das Versionsverwaltungssystem können mehrere Menschen nacheinander an dem Code programmieren. Lediglich das zusätzliche Hochladen der geänderten *Tracingdatei* ist notwendig.
- Eine mögliche Sicherheitsstufe bietet der Umstand, dass die Traceability-Daten nicht in der Quellcodedatei gespeichert sind. Ohne die *Tracingdatei* hat ein Angreifer lediglich Zugriff auf den Quellcode und kann nicht direkt auf die Requirements schließen oder umgekehrt.
- Durch das gezielte Ablegen von Requirements in der separaten Datei kann der Leseaufwand beim Einlesen von Informationen zu Requirements aus dem ReqIF Datensatz reduziert werden. Lediglich neu hinzugefügte Requirements werden eingelesen.
- Dieser Ansatz lässt sich durch Anpassung einer bestehenden Architektur eines Open Source Tools umsetzen.

Nachteile dieses Ansatzes sind:

- Die Verwaltung in separaten Dateien ist umständlich. Statt einer zu verwaltenden Datei gibt es zwei.

- Die Konsistenz der *Tracingdateien* muss gewährleistet werden. Es darf nur maximal eine *Tracingdatei* pro Quellcodedatei geben, um Kompromittierungen zu vermeiden.
- Es gibt für diese Art von Datei noch keine Merge-Verwaltung. Konflikte müssten also vermieden werden, hierfür dürfen nicht mehrere Anwender gleichzeitig an der gleichen Quellcodedatei arbeiten.

Das fundamentale Ziel dieser Arbeit ist die visuelle Verbesserung der Entwicklerumgebung. Durch Verwendung von Ansatz 1, also magischen Kommentaren, würde Quellcode im Rahmen dieses Prototyps an Lesbarkeit einbüßen (R-23). Es sollen insbesondere überlappende Requirements, die oft nur eine Zeile des Quellcodes einnehmen, dargestellt werden können. Die Existenz eines bestehenden Open Source Tools, das zur Darstellung der Boxen auf die gewünschte Art erweitert werden kann, ist für diese Arbeit ein weiterer Grund, Ansatz 2 zu bevorzugen. Durch den etwas geringeren Aufwand bei der Visualisierung können mehr Ressourcen auf die Umsetzung anderer Requirements des Tools verwendet werden. Des Weiteren bestimmt Requirement R-26: Das Tool soll unabhängig von der Programmiersprache sein, in welcher der Nutzer arbeitet. Diese Unabhängigkeit kann durch Verwendung von Ansatz 2 erreicht werden. Aus diesen Gründen wird Ansatz 2 gewählt.

Somit wird eine *ReqIF Datei* zur Verwaltung der Projektanforderungen verwendet, aber für jede Quellcodedatei eine eigene *Tracingdatei* mit den dort relevanten Informationen gehalten.

Mit diesem Ansatz und mit den vorher gebildeten CRC Karten wird der Architekturentwurf erweitert.

5.3 Datengrundlage

Die Daten, mit denen das in dieser Arbeit entwickelte Tool somit arbeiten soll, sind *TRC Dateien*, eine *ReqIF Datei*, ein Projektverzeichnis mit Quellcodedateien und die *Eclipse IDE* (Abbildung 5.3).

Eine *TRC Datei* ist eine spezielle *Tracingdatei*, die den gleichen Dateinamen hat wie die dazu gehörige Quellcodedatei. Lediglich die Endung weicht mit „.trc“ ab. In der *TRC Datei* werden sogenannte *TRCRequirement* Objekte gespeichert. Diese *TRCRequirements* werden selbst definiert und speichern die Zugehörigkeiten zwischen Requirements und Boxen im Quellcode. Die *ReqIF Datei* ist ein Datensatz im gängigen ReqIF Format, mit den Requirements des gesamten Projektes. Die *TRC Dateien* erhalten ihre *textuelle Kurzbeschreibung* aus dieser zentralen *ReqIF Datei*.

Das Projektverzeichnis, in dem das Tool arbeitet, besitzt eine beliebig verschach-

telte Ordnerstruktur, in der die Quellcodedateien abgelegt sind. Die Umgebung, in der dieses Tool arbeitet, ist die *Eclipse IDE*. Als Plugin für diese IDE wird das Tool programmiert.

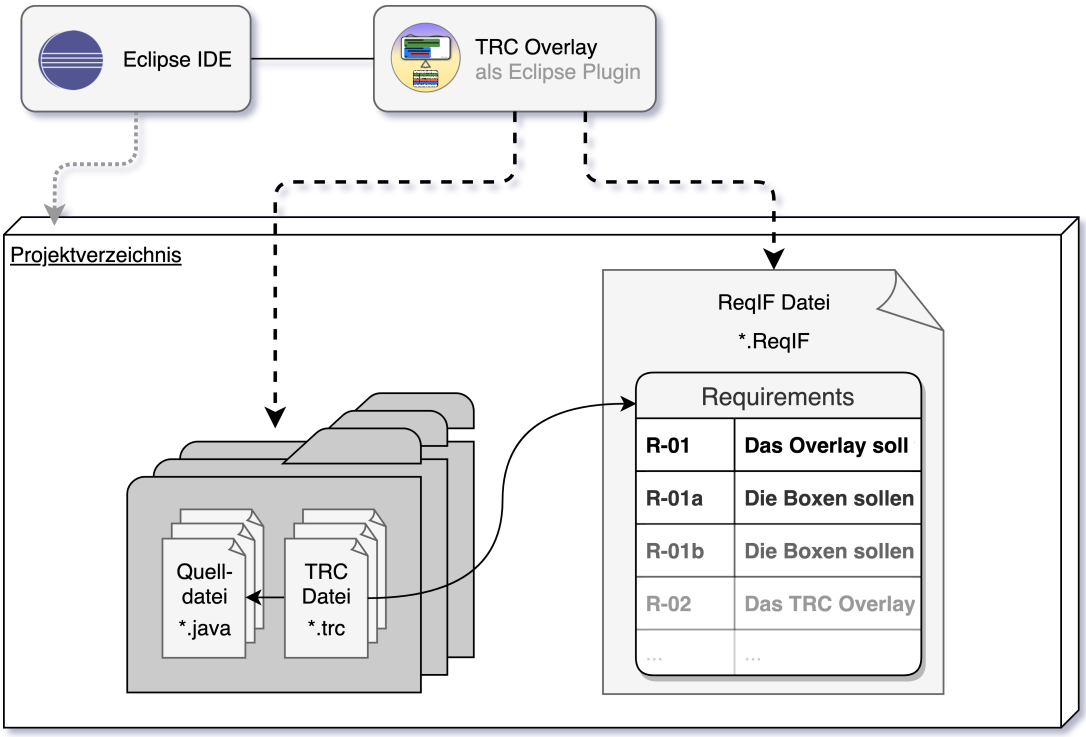

Abbildung 5.3: Visualisierung der Datengrundlage

5.4 Dynamische Sicht

Die wichtigsten dynamischen Abläufe des Tools werden im Folgenden erläutert. Anschließend werden einige Besonderheiten vorgestellt. Das Tool wird dabei häufig als Plugin bezeichnet, da es als Plugin für die *Eclipse IDE* implementiert wird.

Abbildung 5.4 visualisiert Starten und Beenden des *TRC Overlays*, wie auch die ablaufenden Funktionen, welche die farbigen Boxen für den Benutzer sichtbar machen. Nach dem Start der *Eclipse IDE* kann der Nutzer das *TRC Overlay* aktivieren. Beim ersten Aktivieren wird das Plugin aktiviert und verbleibt bis zum Beenden von Eclipse im aktiven Status. Eclipse speichert den Status mit dem das Plugin beendet wird ab, sodass bei einem erneuten Start der *Eclipse IDE* kein manuelles Starten des *TRC Overlays* mehr nötig ist.

Nach dem Start des *TRC Overlays* erfragt das *TRC Overlay* von der IDE den Pfad zur aktuellen geöffneten Quellcodedatei. Es bestimmt die zugehörige *TRC*

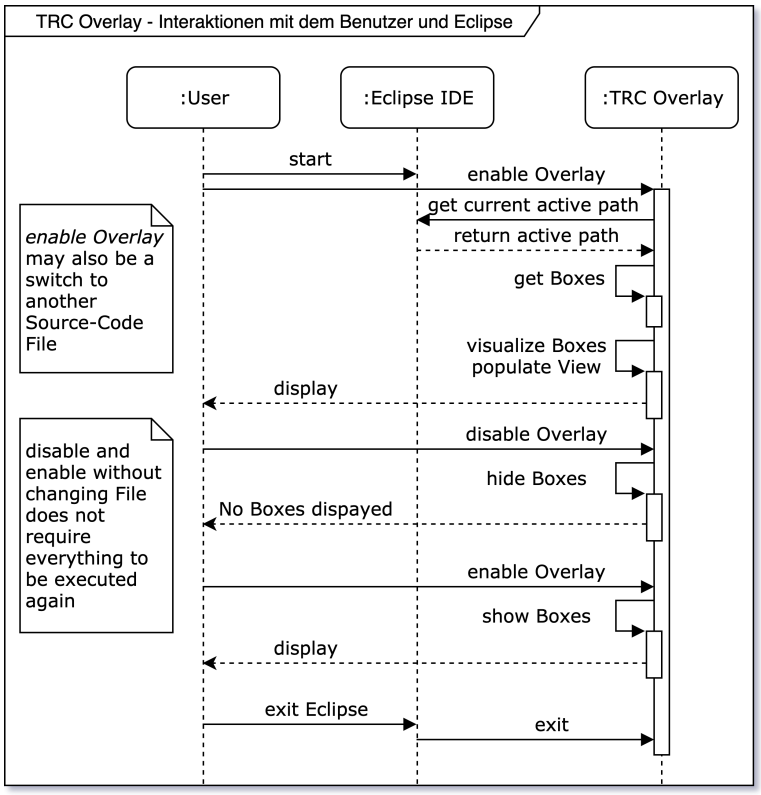

Abbildung 5.4: Sequenzdiagramm für das Starten und Beenden des Tools

Datei und liest die Daten aus. Das Tool erzeugt dann anhand der Daten einzelne Boxen, färbt diese ein und befüllt die integrierte *TRC View* mit den Requirements. Zu diesem Zeitpunkt sieht der Nutzer die Boxen und die gleichfarbigen Requirements in der *TRC View*.

Falls das *TRC Overlay* nicht benötigt wird, kann es mit Hilfe des Knopfs zur Aktivierung des Plugins ausgeblendet und wieder eingeblendet werden. Beim erneuten Einblenden des *TRC Overlays* wird nicht die *TRC Datei* eingelesen, sondern nur die ausgeblendeten Boxen wieder eingeblendet.

Beim Modellieren der dynamischen Sicht werden auch Qualitätsanforderungen wie R-24 beachtet. Bei der Planung der Umsetzung von R-06 wird V1 als eine Variante integriert. Diese Variante bedeutet für 99 % der lesenden Aufrufe der *Traceability-Data Interaction* eine vollkommene Ersparnis des Lesevorgangs aus der *TRC Datei*. Der Vorgang trifft auf alle Lesevorgänge zu, wird aber in den Abbildungen 5.5 und 5.6 exemplarisch an dem Sequenzdiagramm für R-06 erklärt.

In V2 (Abbildung 5.6) selektiert der Nutzer einen Codeabschnitt im Quellcode. Er wählt dann den Knopf zum Setzen der aktiven Requirements. Das *Traceability-Data Overlay* erfragt die aktiven Daten von der *Traceability-Data Interaction*,

diese prüft ob die *TRC View* bereits die benötigten Daten enthält. Beim ersten Aufruf für eine Quellcodedatei enthält die View nicht die benötigten Daten, somit muss die zugehörige *TRC Datei* eingelesen werden. Sobald V2 einmal durchgeführt wurde, enthält die *TRC View* alle benötigten Daten und es kann somit bei allen folgenden Lesevorgängen auf das zeitaufwändige Lesen der *TRC Datei* verzichtet werden (Abbildung 5.5). Nach jeder Änderung werden die Daten auch in der *TRC Datei* gespeichert, um die Daten in der *TRC View* und der *TRC Datei* konsistent zu halten.

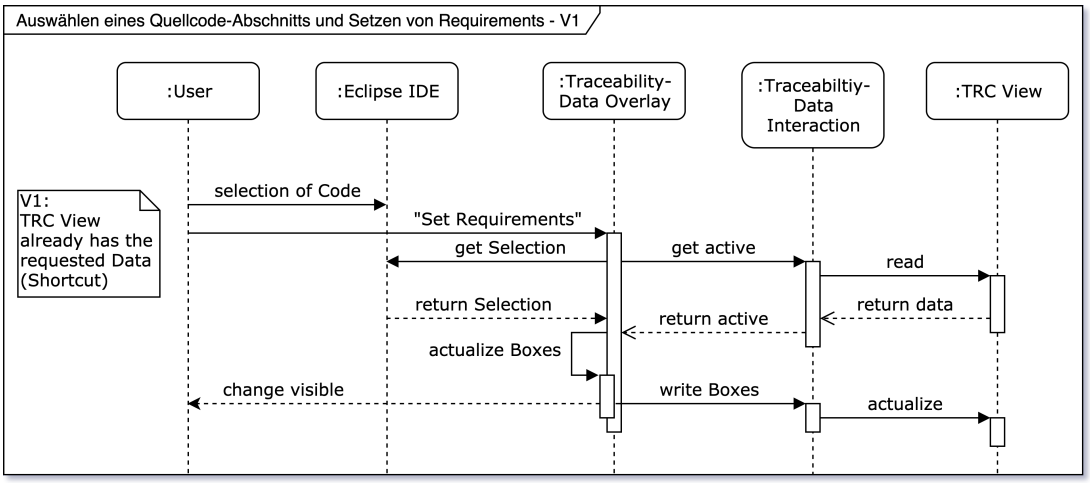

Abbildung 5.5: V1 - Sequenzdiagramm für R-06

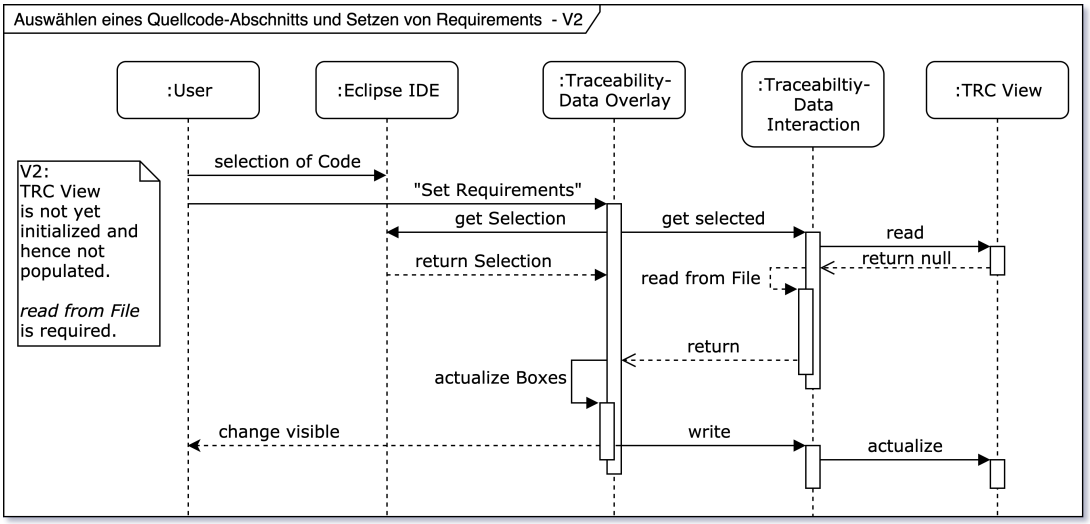

Abbildung 5.6: V2 - Sequenzdiagramm für R-06

Anhand der diesen Sichten zugrunde liegenden Softwarearchitektur wird das Tool zur visuellen Annotation von Quellcode mit Traceability-Informationen umgesetzt.

5.5 Verwendete Technologien

Die folgenden Technologien werden im Rahmen dieser Arbeit nicht selbst implementiert, sondern angepasst, beziehungsweise verwendet.

EditBox ist ein Tool für die *Eclipse IDE*.[1] *EditBox* legt farbige Boxen mit unterschiedlichen Farben für unterschiedliche Einrückungen des Quellcodes an (Abbildung 5.7).

EditBox erzeugt in seiner Originalform Boxen je nach Einrückung des Quellcodes. Automatisch eingerückte Boxen werden für Quellcode-Blöcke wie beispielsweise Schleifen, Abfragen oder Methoden angelegt.

Der Quellcode wird von *EditBox* als 0 indizierter Zeichenstrom interpretiert. Das erste Zeichen in einer Datei hat also den Index 0, das zweite den Index 1, et cetera. Die Boxen werden an diesen Indizes „befestigt“. Dadurch kann eine Box innerhalb einer Zeile nur wenige Zeichen umfassen, oder vom Beginn der Datei bis zum Ende der Datei reichen.

EditBox setzt die Darstellung der Boxen bereits auf eine Art um, die für den Zweck dieser Arbeit adaptiert werden kann. Durch Verwendung und Anpassung von *EditBox* können mehr Ressourcen auf andere Requirements an den Prototypen verwendet werden.

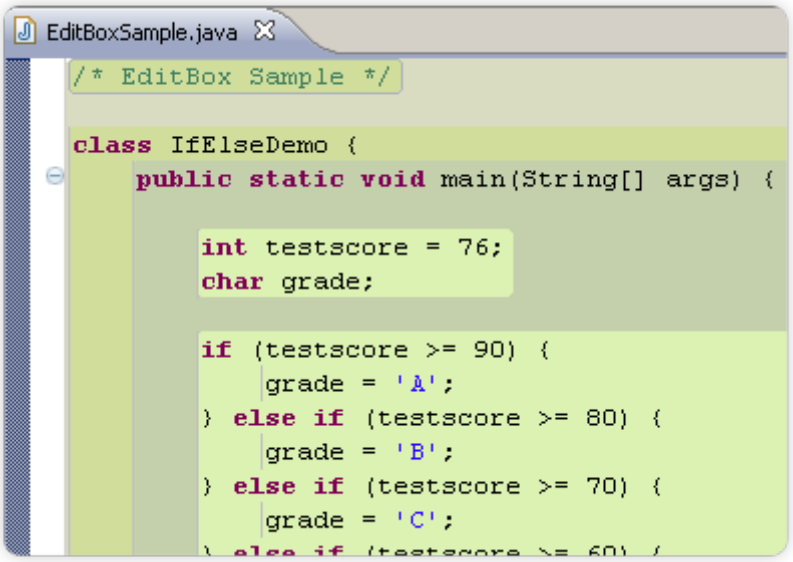

Abbildung 5.7: *EditBox* Beispiel

Nachdem *EditBox* von Piotr Metel und Paul Verest unter der EPL 1.0[2] lizenziert

[1] Eclipse Plugin. Verfügbar unter http://editbox.sourceforge.net/
[2] Eclipse Public License - Version 1.0. Verfügbar unter https://spdx.org//licenses/EPL-1.0

ist, kann der Quellcode des Tools als Grundlage für die Umsetzung der farbigen Boxen benutzt werden. Das Tool wurde ursprünglich von Piotr Metel entwickelt und dann von Paul Verest auf *Github* hochgeladen und weiter gepflegt.

Im Rahmen dieser Arbeit wird die Implementierung des Box Builders, der die Boxen erstellt, komplett überarbeitet und die Implementierung des Box Decorators, der die Boxen grafisch umsetzt, angepasst.

ReqIf Parser Zum Parsen der zu jedem Projekt zugehörigen *ReqIF Datei* werden die Tools von Kay Erik Muench verwendet.[3] Mit dessen ReqIF Toolset kann performant über einen ReqIF Datensatz iteriert werden. Dabei lassen sich Objekte erzeugen, die in Eclipse benutzt werden können. Da auch der ReqIf Parser unter der EPL 1.0 veröffentlicht wurde, kann er für dieses Tool verwendet werden.

Eclipse IDE Das Tool wird in der *Eclipse IDE* umgesetzt. Die genaue Version der IDE, mit der dieses Tool als Plugin erstellt wird, ist die *Eclipse IDE* der Version 2019-06 (4.12.0), mit der Build id 20190614-1200. Eclipse ist Open Source Software, ausführlich dokumentiert und bietet zahlreiche Beispiele für die Plugin Entwicklung. Aus diesen Gründen wird der Prototyp als Plugin für die *Eclipse IDE* programmiert.

[3] reqiftools. Verfügbar unter https://github.com/KayErikMuench/reqiftools

6 Umsetzung

Dieses Kapitel erläutert die genauere Umsetzung des in Kapitel 5 vorgestellten Entwurfs. Das Ziel der Umsetzung ist kein verkaufsfertiges Tool, sondern ein Prototyp als *Proof of Concept* der visuellen Annotation von Quellcode mit Traceability-Informationen.

6.1 Realisierung der Komponenten

Die Verteilung der Komponenten auf einzelne Softwareartefakte beziehungsweise Klassen wird auf folgende Art und Weise umgesetzt. Bei der Umsetzung dieses speziellen Prototyps gibt es für jede der umgesetzten Klassen ein eigenes Softwareartefakt. Die Sicht auf Klassenebene (Abbildung 6.1) gibt einen Gesamtüberblick über das zu entwickelnde Tool für den Entwickler. Nicht eingezeichnet sind hier die Schnittstellen für die Nutzerinteraktion, da der Fokus hier auf systeminternen Abhängigkeiten und Abläufen liegt.

Traceability-Data Interaction Die Responsibilities dieser Komponente werden auf zwei Klassen aufgeteilt: *TRCFileInteraction* und *ReqIFFileInteraction*. Die Klasse *ReqIFFileInteraction* behandelt das Lesen des ReqIF Datensatzes. Die Klasse *TRCFileInteraction* ist ausschließlich für das Lesen und Schreiben der *TRC Dateien* verantwortlich, in denen die *TRCRequirement* Objekte als LinkedList gespeichert sind.

TRCRequirement Diese Klasse definiert, wie ein *TRCRequirement* Objekt aussieht.

ReqIFFileInteraction Die Methoden der Klasse *ReqIFFileInteraction* werden von der *TRCFileInteraction* aus aufgerufen und lesen den ReqIF Datensatz. Die Klasse *ReqIFFileInteraction* verwendet zum Lesen des ReqIF Datensatzes die bereits implementierten Methoden von Kay Erik Muench (Abschnitt 5.5).

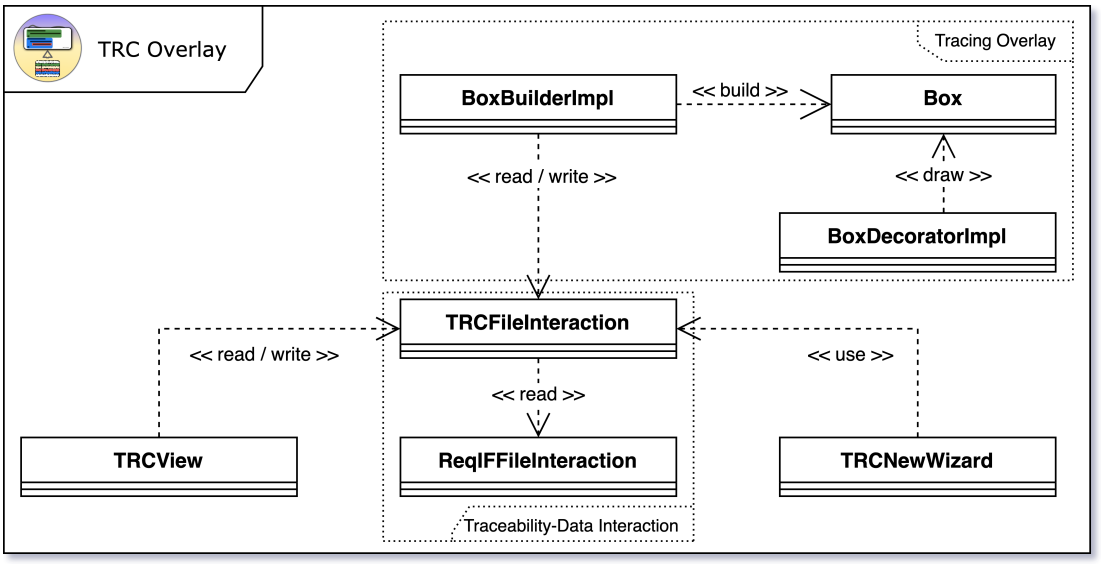

Abbildung 6.1: Realisierungssicht für das *TRC Overlay*

TRCFileInteraction Das Lesen und Schreiben der *TRC Dateien* wird in der Klasse *TRCFileInteraction* realisiert. Die Klasse *ReqIFFileInteraction* wird von hier aus aufgerufen, um die Kurzbeschreibungen zu den Requirement IDs zu erhalten.

Tracing Overlay Das eigentliche *Tracing Overlay*, also die Komponente, die Annotationen im Code sichtbar machen soll, wird innerhalb der bestehenden Klassen des *EditBox* Plugins (siehe Abschnitt 5.5) umgesetzt. Die Klasse *BoxBuilderImpl* wird neu geschrieben und die Klasse *BoxDecoratorImpl* angepasst. In der *BoxBuilderImpl* werden die farbigen Boxen erstellt, und in der *BoxDecoratorImpl* werden sie für den Anwender sichtbar gemacht.

TRCNewWizard Mit den Funktionen der Klasse *TRCNewWizard* können *TRC Dateien* bearbeitet und erstellt werden. Für die Bearbeitung der *TRC Dateien* verwendet der *TRCNewWizard* die Funktionen der Klasse *TRCFileInteraction*. Die *TRCView* fragt die aktuell anzuzeigenden Requirements mittels der *TRCFileInteraction* ab und zeigt sie an.

6.2 Implementierung des Tracing Overlays

Die bestehende *EditBox* Struktur wird umgeformt, um das *Tracing Overlay* zu realisieren. Insbesondere die Klassen *BoxBuilderImpl*, die für das Anlegen der Boxen verantwortlich ist, und *BoxDecoratorImpl*, die ihrerseits für die graphische Darstellung der Boxen verantwortlich ist, werden modifiziert.

Das *Tracing Overlay* kann mit einem Knopf in der Eclipse Toolbar aktiviert, aus- und wieder eingeblendet werden.

BoxBuilderImpl In der *BoxBuilderImpl* wird der Algorithmus der die Boxen erstellt neu implementiert. Die Boxen werden jetzt nicht mehr, wie in *EditBox* zuvor üblich, durch Scannen des Quellcodes erstellt. Stattdessen werden die Requirements aus den *TRC Dateien* ausgelesen und die zugehörigen dort abgespeicherten Boxen verwendet. In dieser Klasse wird dafür gesorgt, dass sich die Boxform automatisch an den in der Box enthaltenen Quellcode anpasst.

Eine Box wird nicht angezeigt, wenn sie nur Leerzeichen oder Tabulatoren enthält. Sie wird aber intern trotzdem angelegt und als Box behandelt. Der Grund für dieses Verhalten ist die Unabhängigkeit der Traceability-Daten vom Quellcode. Die Boxen „wissen“ sozusagen nicht, welche Art von Zeichen sie beinhalten. Sie werden intern als Start- und Endwert Paare gespeichert. Die Werte sind dabei Indices von Zeichen in der Quellcodedatei. Bei der Erstellung der Boxen durch *BoxBuilderImpl* wird dann erst beim Anlegen der visuellen Boxen bestimmt, wo eine Box beginnt und ob sie angezeigt wird.

Die Box soll visuell mit dem ersten Zeichen beginnen, das kein Leerzeichen oder Tabulator ist (siehe R-09). Diese visuelle Box entspricht nicht der intern gespeicherten, vom Nutzer markierten Box. Intern gespeicherte Boxen können führende und nachgestellte Leerzeichen oder Tabulatoren mit einschließen. Das erleichtert das Markieren von Quellcode-Abschnitten. Der Nutzer muss nicht genau das erste und letzte Zeichen des gewünschten Codeabschnitts selektieren.

Zudem verhindert diese Art der Umsetzung, dass viele kleine einzelne Boxen angelegt werden. Beim Anlegen der Boxen während des Programmierens müsste eine Box für jedes einzelne Wort angelegt werden, wenn keine führenden und nachgestellten Leerzeichen intern erlaubt sind. Diese einzelnen Boxen müssten dann erkannt und zusammengefügt werden. Diese umständliche Art der Umsetzung wird für die Umsetzung des Prototyps nicht gewählt.

Die Einhaltung von R-09 und R-10 ist trotzdem gewährleistet. Die visuelle Darstellung der Boxen beginnt mit dem ersten Zeichen der ersten Zeile, das kein Leerzeichen oder Tabulator ist, und endet beim letzten markierten Zeichen, das kein Leerzeichen oder Tabulator ist.

BoxDecoratorImpl Die Klasse *BoxDecoratorImpl* wird weitgehend von *EditBox* übernommen. Lediglich der Darstellungsalgorithmus wird überarbeitet, um den Requirements des *Tracing Overlay* zu entsprechen. Zudem wird das Aufgabenfeld des *Change Listeners* um die notwendigen Funktionen für diesen Prototypen erweitert. Der *Change Listener* reagiert auf Benutzereingaben im Editor der Eclipse IDE. Er sorgt bei einer Änderung im Quellcode für die Aktivierung der

Funktionen, die die Änderung auch in den Traceability-Daten umsetzen. Die Anpassung des Darstellungsalgorithmus beschränkt sich überwiegend auf die Farbzuweisung. Anstatt eines einheitlichen Farbschemas werden nach dieser Änderung die zu den Requirements gespeicherten Farben verwendet. Bei der Verwendung des EditBox Overlays zur Darstellung mehrerer, verschiedener Requirements im gleichen Code, offenbart sich der folgende Seiteneffekt:

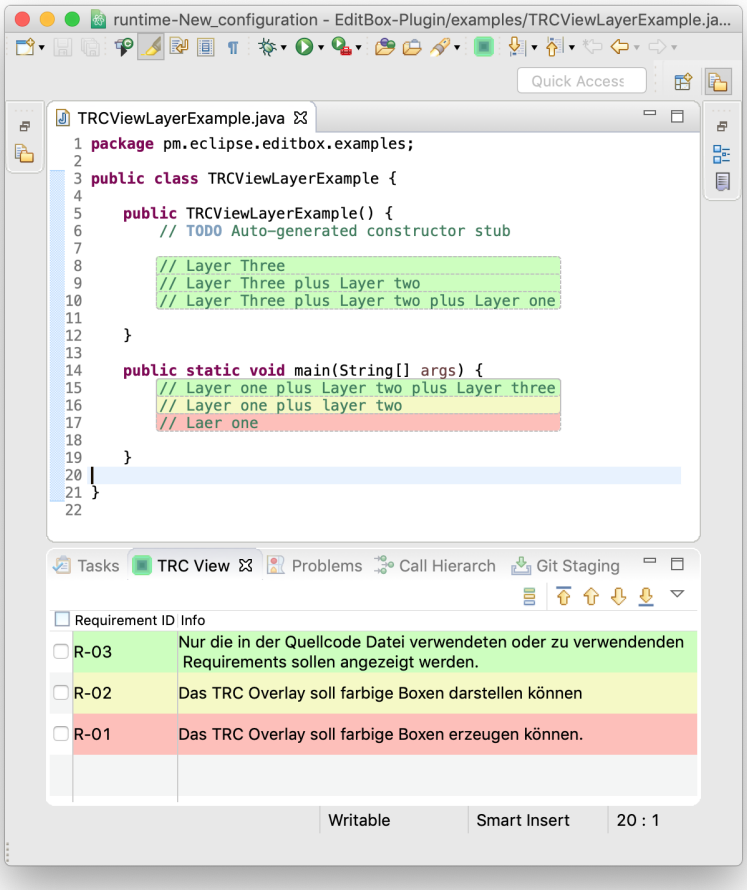

Abbildung 6.2: Layer-System R-03, R-02, R-01

Die Boxen werden durch Verwendung des bestehenden Algorithmus nacheinander, Requirement für Requirement im Editor „gemalt“. Das bedeutet, später gezeichnete Boxen „überlappen“ solche, die früher gezeichnet wurden. Der erste Lösungsansatz ist, die betreffenden späteren Boxen in einer Farbmischung einzufärben. Diese Idee lässt sich mit dem bestehenden Tool jedoch nur schwer umsetzen und erschwert das visuelle Zuordnen von Boxen in Mischfarben zu Requirements in reinen Farben. Bei genauerer Betrachtung einer solchen Kollision (vgl. Abbildung 6.2) kann zudem festgestellt werden, dass ein solches verdeckendes Verhalten auch nützlich und dem Zweck des Tools dienlich sein kann.

Deshalb wird beschlossen, die Visualisierung von verschiedenen Requirements im Quellcode mittels eines Layer-Systems, eines Ebenen-Systems, darzustellen. Der Vorteil an einer solchen Einteilung in Ebenen ist, dass der Fokus vom Anwender auf beliebige Requirements gelegt werden kann.

Die Information, zu welchem Requirement eine „verschattete“ Box gehört, geht bei dieser Art der Darstellung nicht verloren. Sie kann in der *TRC View* eingesehen werden. Dieses zusätzliche Feature wird nachträglich als Requirement hinzugefügt.

R-27: Der Anwender soll seinen Fokus mittels eines Layer-Systems auf vom Anwender ausgewählte Requirements setzen können.
Begründung: Durch Ermöglichung dieser Priorisierung kann die Useability weiter erhöht werden.

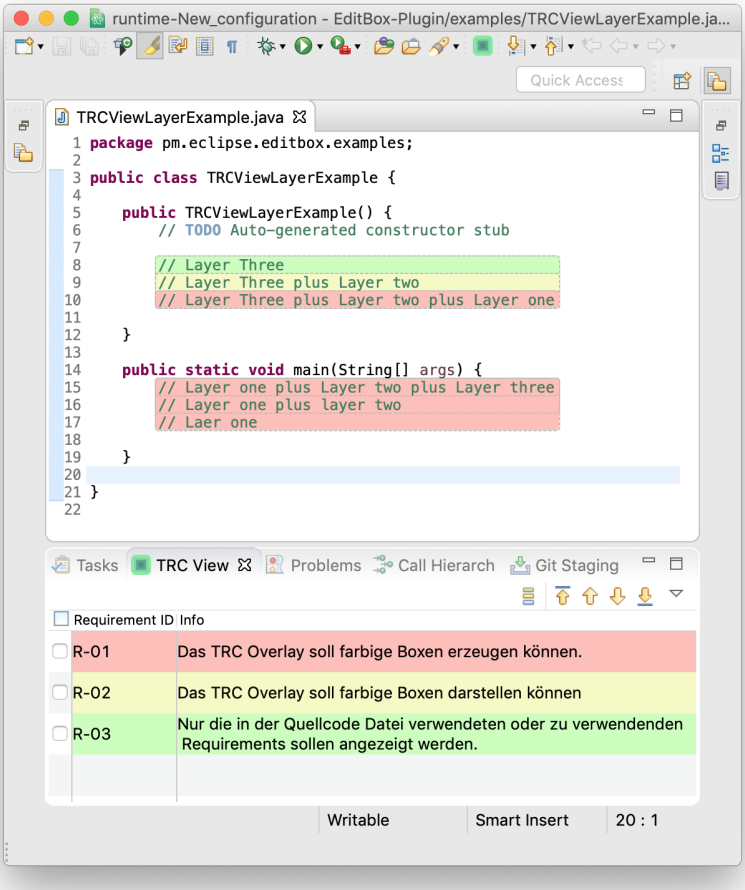

Abbildung 6.3: Layer-System R-01, R-02, R-03

In Abbildung 6.2 sind drei Requirement Boxen übereinander angelegt worden. Die Boxen für Requirement R-03 sind hierbei beispielsweise Quellcode Zeile acht bis zehn und Zeile 15. In Abbildung 6.3 ist die Reihenfolge der Requirements in

der *TRC View* umgekehrt worden. Dadurch ist die zuvor gut sichtbare Box für R-03 in Zeile acht bis zehn nun in Zeile neun und zehn von anderen Boxen verdeckt. Möchte der Anwender wissen, welche weiteren Boxen unter verdeckenden Boxen verborgen sind (vgl. Zeile 15-17 Abbildung 6.4), so kann er dies bei fokussierter *TRC View* erfahren. Er muss dazu nur mit dem Mauszeiger über eine beliebige Box im Quellcode fahren. Die Requirements zu den Boxen unter der derzeitigen Mausposition (Zeile 16 in Abbildung 6.4), werden in der *TRC View* weiß hervorgehoben.

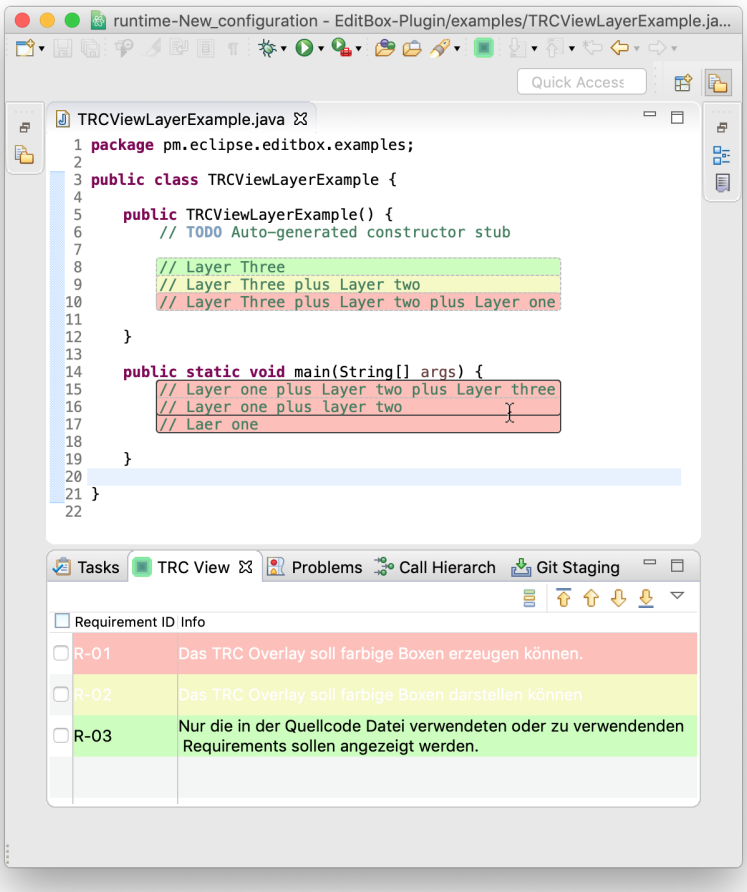

Abbildung 6.4: Layer-System mit hervorgehobenen Requirements

6.3 Implementierung der eigenen Klassen

TRCRequirement Diese *TRCRequirements* sind selbstdefiniert und haben die folgenden Attribute. Die *ID* des Requirements, eine *Farbe*, eine *Liste von zugehörigen Boxen*, eine *textuelle Kurzbeschreibung* und einen Status *aktiv*.

Ein TRCRequirement hat zu jedem Zeitpunkt seiner Existenz genau eine *ID* die als String gespeichert wird.

Die *Farbe* dieses *TRCRequirements* ist gleichzeitig auch die Farbe der zugeordneten Boxen. Diese Farbe wird beim Neuanlegen eines *TRCRequirement* Objekts automatisch zugewiesen. Sie wird als RGB Werte gespeichert.

Die *Liste von zugehörigen Boxen* entspricht der Liste der im Quellcode annotierten Bereiche von diesem Requirement. Diese annotierten Bereiche werden als Liste von Startwert-Endwert Paaren gespeichert. Die Liste ist in aufsteigender Reihenfolge sortiert.

Die textuelle Kurzbeschreibung wird von der Klasse *ReqIFFileInteraction* zugewiesen und als String gespeichert.

Der Status *aktiv* ist ein Wahrheitswert, der angibt, ob das Requirement in der View markiert ist. Diese Markierung wird mittels einer Checkbox zu Beginn der Zeile in der *TRCView* visualisiert.

TRC View Die *TRC View* wird als eigene *Eclipse View* implementiert. Die *Eclipse IDE* besteht in den fundamentalsten Zügen aus einer *Workbench* und einem *Workspace*. Die *Workbench* ist die Arbeitsoberfläche des Benutzers. In der *Workbench* können mittels Editoren Dateien geöffnet und bearbeitet werden. Im *Workspace* können beliebige *Views*, ähnlich wie Tabs in einem Browser, geöffnet werden. Durch Klicken auf die Reiter kann der Nutzer dann zur gewünschten *View* wechseln.

Die *TRC View* liest mittels der *TRCFileInteration* die *TRC Datei* der derzeit geöffneten Quellcodedatei. Dabei ist die Reihenfolge der Requirements relevant, da die Boxen im Quellcode im Layer-System dargestellt werden. Das Layer-System wird durch die Reihenfolge der Requirements in der *TRC View* widergespiegelt. Die weiteren Informationen, die aus der *TRC Datei* ausgelesen werden, sind die Farbe, in der die *TRC View* das Requirement hinterlegen soll und die textuelle Kurzbeschreibung.

In der *TRC View* können Requirements mittels Checkboxen ausgewählt werden. Bei Änderungen im Quellcode werden automatisch Boxen für die ausgewählten Requirements erstellt oder bestehende Boxen entsprechend der Änderung erweitert oder verkürzt. Des Weiteren wird implementiert, dass die *Tracingdatei* bei einer Änderung im Text der Quellcodedatei angepasst wird.

Von der Änderung nicht betroffene Boxen werden dabei nicht „verschoben“.

So muss beispielsweise der Start- und Endindex aller Boxen, die im Quellcode hinter einer Änderung liegen, beim Einfügen eines Zeichens um 1 erhöht werden. Andernfalls würden die Boxen ein Zeichen nach „vorne“ wandern.

Zusätzlich können, mittels eines weiteren Knopfes in der *TRC View*, Requirements „gesetzt“ werden. Der Benutzer kann einen Codeabschnitt selektieren und mit einem Klick den Requirements zuordnen, die zuvor in der *TRC View* gesetzt wurden. Ist kein Requirement ausgewählt, so werden bestehende Boxen im Bereich der Selektion geteilt, gelöscht oder verkürzt. Durch Änderungen kann es auch dazu kommen, dass bestehende Boxen verschmolzen, erweitert oder neu angelegt werden müssen. Diese Fälle sind durch eigene Implementierung abgedeckt.

TRCFileInteraction In der Klasse *TRCFileInteraction* sind die Methoden zum Lesen und schreiben der *TRC Dateien* implementiert.

Beim Lesen werden zunächst die *TRCRequirements* aus der *TRC Datei* eingelesen. Anschließend wird im ReqIF Datensatz nach den IDs der *TRCRequirements* gesucht, die noch keine Kurzbeschreibung besitzen. Für gefundene Requirements, wird die Kurzbeschreibung eingelesen und im *TRCRequirement* gespeichert. Einmal eingelesen bleibt die Information gespeichert, auch wenn sich der ReqIF Datensatz zu einem der Requirements ändert. Es wurde keine Funktionalität zur Verwaltung von Änderungen im ReqIF Datensatz implementiert. Um die textuelle Kurzbeschreibung eines *TRCRequirements* zu ändern, muss das *TRCRequirement* mit allen zugeordneten Boxen gelöscht werden. Anschließend kann es mittels *TRCNewWizard* erneut hinzugefügt werden.

Die Begründung für dieses Verhalten ist, dass eine Änderung eines Requirements meistens Änderungen im Quellcode erfordert. Zur korrekten Änderungsverwaltung müssten die *TRCRequirements*, die von der Änderung betroffen sind, als ungültig gekennzeichnet werden. Der Nutzer kann dann für die einzelnen Boxen überprüfen, ob die Zuordnungen noch korrekt sind. Andernfalls kann er Korrekturen an Quellcode und Traceability-Informationen durchführen. Abschließend wird das *TRCRequirement* wieder als gültig gekennzeichnet. Da dieser Ansatz im Falle dieses Prototyps nicht umgesetzt wurde, müssen nach einer Änderung in einem Requirement die manuellen Schritte des Löschens und Neuanlegens zur Korrektur erfolgen.

ReqIFFileInteraction Diese Klasse liest den ReqIF Datensatz des Projektes. Dabei ist folgendes für die Funktionalität zu beachten.

Das ID-Feld der verwendeten ReqIF Datei muss „requirementID“ heißen und das Feld, in dem die Kurzbeschreibung gespeichert wird, muss „PlainText“ genannt werden. Auch muss die ReqIF Datei im Projektordner „reqif“ liegen und ihr Name

muss „TRCOverlay.reqif“ sein. Diese Benennungen werden für diesen Prototyp in den Quellcode eingebettet und können in der Klasse *ReqIFFileInteraction* angepasst werden. Ein Beispiel für einen solchen ReqIF Datensatz gibt die dem Quellcodeverzeichnis dieser Arbeit beiliegende Datei „TRCOverlay.reqif“. In ihr sind exemplarisch die Requirements dieses Prototyps eingetragen. Der ReqIF Datensatz kann weit mehr Daten umfassen, als in diesem Beispiel. Die für die Funktion dieses Prototyps wichtigen Daten sind jedoch ausschließlich die IDs und die textuelle Kurzbeschreibung.

TRCNewWizard Der *TRCNewWizard* wird implementiert, um das manuelle Anlegen und Bearbeiten von *TRC Dateien* zu ermöglichen. Der Wizard übernimmt beim Start die letzte Selektion des Benutzers. Wenn der Nutzer eine Quellcodedatei im Projektverzeichnis ausgewählt hat, dann wird der entsprechende Dateipfad automatisch eingefügt. Danach wird der Benutzer um die Eingabe der Requirement IDs gebeten. Hier können beliebige IDs angegeben werden. Die textuelle Kurzbeschreibung kann jedoch nur zu Requirements, die auch im ReqIF Datensatz des Projektes enthalten sind, gefunden werden. Die Farbe des einer Datei neu zugeordneten Requirements wird einmalig zufällig ausgewählt. Dieses Auswählen geschieht nach dem Ändern der *TRC Datei* mit dem *TRCNewWizard*. Die zufällige Auswahl wird hierbei auf hellere Farben eingeschränkt, um den Code nicht unleserlich zu machen. Der Benutzer kann durch Doppelklick auf ein Requirement in der *TRC View* die Farbe nachträglich anpassen.

6.4 Entwicklung der programmatischen Tests

Der Umstand, dass das Tool als Plugin für die Eclipse IDE entwickelt wird, erschwert programmatische Integrationstests. Es müssen Mocks, also simulierte kontrollierte Platzhalter, für komplexe Bestandteile der Eclipse IDE angelegt werden. Das Anlegen dieser komplexen Mocks erfordert eine hohe Einarbeitungszeit. Das Testen auf Systemebene ist in diesem Fall mit geringerem Zeitaufwand verbundenen. Manuelles Testen ist jedoch weniger zuverlässig als programmatische Tests. Da viele der Requirements visuell prüfbar sind, wird an dieser Stelle auf einen ausführlichen Integrationstest verzichtet und direkt auf Systemebene getestet.

Die Natur dieses Plugins erschwert auch das Testen auf Modulebene. Wichtige Methoden, die manuell nicht leicht prüfbar sind, werden in eigenen Helfer-Klassen isoliert und programmatisch mit JUnit Tests geprüft. Das Anlegen der Boxen, und das Anpassen der Boxen nach einer Änderung wird beispielsweise auf diese Art getestet. Die Anzahl der dabei möglichen Situationen ist nicht manuell prüfbar. Die Tests beachten Grenzfälle und Normalfälle und prüfen viele mögliche Kombinationen von verschiedenen Situationen.

Im Falle dieser Prototypentwicklung wird bei verwendeten Komponenten wie dem ReqIF Parser und Komponenten der *EditBox* eine korrekte Programmfunktion angenommen und nicht getestet.

Mit den gesamten vorhergehenden Schritten ist die Planung und Umsetzung des Tools abgeschlossen. Es bleibt durch Tests zu zeigen, dass das Tool die Requirements erfüllt.

7 Evaluation

In diesem Kapitel wird zunächst auf die Evaluation des Prototyps und anschließend auf die Auswertung der Umfrage eingegangen.

7.1 Evaluation des Prototyps

Die Evaluation zeigt durch die Prüfung der Requirements mit Modul- und Systemtests, ob das Tool die Requirements erfüllt, Die Modultests wurden als JUnit Tests umgesetzt. Die Natur dieses Plugins bedingt einen hohen Mehraufwand bei Integrationstests (siehe Abschnitt 6.4). Aus diesem Grund, und wegen des vergleichsweise geringen Nutzens, wird auf Integrationstests verzichtet. Die korrekte Zusammenarbeit der Module wird durch Systemtests sichergestellt, deren Prüfung manuell erfolgt.

R-01: Die IDs und die textuellen Kurzbeschreibungen der einer Quellcodedatei zugeordneten Requirements sollen in einem zusätzlichen Fenster angezeigt werden.

Dieses Requirement ist als *TRC View* umgesetzt, die auf den Screenshots (Abbildungen 8.4 und 8.5) im unteren Bereich zu erkennen ist. In ihr werden die IDs der Requirements und deren zugehörige textuelle Kurzbeschreibung angezeigt. Die Anzeige der textuellen Beschreibung erfolgt ein- oder Zweizeilig, entsprechend der Länge der längsten textuellen Beschreibung. Der Text von Beschreibungen, die über zwei Zeilen der TRC View hinaus gingen wird nicht angezeigt. Die textuelle Beschreibung kann in voller Länge in Form einer Tooltip-Information eingeblendet werden, indem der Nutzer den Mauszeiger über dem Requirement positioniert. Verschiedene IDs und unterschiedlich lange textuelle Beschreibungen wurden im manuellen Systemtest in der *TRC View* getestet. Es traten keine Fehler auf.

R-02: Die Zuordnung von Requirements zu einer Quellcodedatei soll mit dem Tool durch den Nutzer erfolgen können.

Der Nutzer kann zu bestehenden Quellcode Dateien beliebige Requirements durch Eingabe deren IDs hinzufügen. Wenn bereits eine *TRC Datei* zur Quellcodedatei existiert, so kann der Nutzer die bereits zugewiesenen Requirement IDs bearbeiten. In manuellen Systemtests wurden neue *TRC Dateien* zu verschiedenen Quellcodedateien angelegt. Auch wurde die Bearbeitung bestehender *TRC Dateien* durch nachträgliches Entfernen und Hinzufügen von IDs getestet. Es traten keine Fehler auf. In Abbildung 8.3 ist der Wizard zu sehen, in dem der Nutzer eine Liste von IDs von Requirements angegeben hat. Die Anzeige ebendieser Requirements erfolgt nach der Bestätigung in der *TRC View* (Abbildung 8.4).

R-03: Die textuelle Kurzbeschreibung der Requirements soll aus einer Datei im gängigen Standardformat für Requirements-Management ReqIF bezogen werden.

Wie in R-03 gefordert, wird die textuelle Kurzbeschreibung der Requirements aus einer ReqIF Datei ausgelesen. Die manuellen Tests für diese Requirements erfolgten mit unterschiedlichen ReqIF Dateien, deren Felder genau wie in Abschnitt 6.1 spezifiziert benannt wurden. Da die Funktionalität für dieses Requirement nicht selbst entwickelt wurde, wird hier auf Modultests in Form von JUnit-Tests verzichtet. Die korrekte Funktion der Funktionalität wird angenommen und nur im Systemtest überprüft. Auch hier konnte im Systemtest bei sachgemäßer Bedienung kein Fehlverhalten festgestellt werden. Die Verwendung von kompromittierten ReqIF Dateien, oder ReqIF Dateien, deren Felder nicht wie erwartet benannt waren führt dazu, dass in der *TRC View* keine Kurzbeschreibungen angezeigt werden.

R-04: Das zusätzliche Fenster soll Knöpfe zum Auslösen von Funktionen des Tools enthalten.

Die *TRC View* hat Knöpfe zum Manipulieren des Layer-Systems und einen Knopf zum Setzen von Boxen im Quellcode, die in manuellen Tests die richtigen Funktionen starten. Die Knöpfe sind in der oberen rechten oberen Ecke der *TRC View* (Abbildung 7.1) angebracht. Der Knopf links vom Separator löst die Funktion zum Setzen der Boxen im Quellcode aus, und die vier Knöpfe nach dem Separator ermöglichen das Bearbeiten des Layer-Systems.

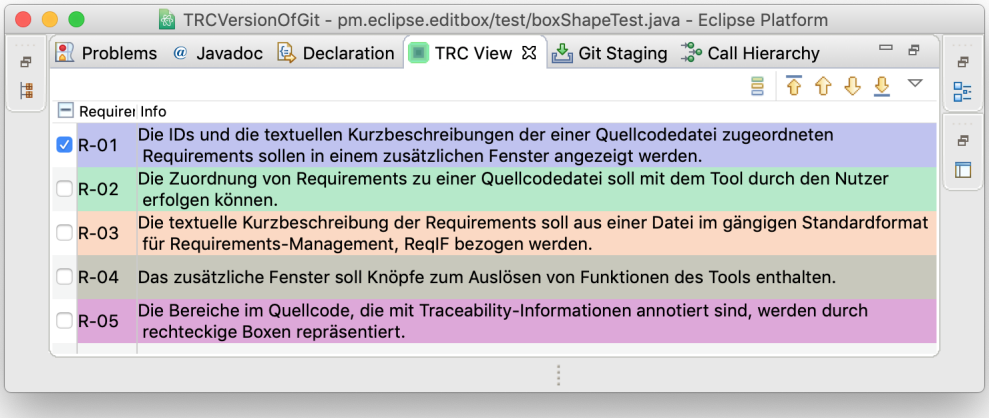

Abbildung 7.1: *TRC View* mit fünf Requirements in zweizeiliger Darstellung

R-05: Die Bereiche im Quellcode, die mit Traceability-Informationen annotiert sind, werden durch rechteckige Boxen repräsentiert.

Bei Tests verschiedenster Kombinationen von Start und Endpunkten der Boxen wurden immer rechteckige Boxen angezeigt, es traten keine unerwarteten Formen auf (Abbildung 8.1).

R-06: Boxen sollen durch Selektion von Quellcode durch den Nutzer und anschließenden Knopfdruck angelegt werden können. Dabei sollen die Boxen automatisch vom Tool zu ausgewählten Requirements zugeordnet werden.

Der Nutzer kann in der *TRC View* mit den Checkboxen Requirements auswählen (Abbildung 8.5). Danach kann er einen Quellcodeabschnitt markieren und mit einem Klick auf den Knopf in der *TRC View* Boxen zu dem ausgewählten Requirement anlegen (Abbildung 8.6). Durch die Umsetzung dieser Requirements sind Korrekturen an den Traceability-Informationen, und das Hinzufügen von Traceability-Informationen im Nachhinein möglich. Existieren bereits Boxen im Quellcode, so gibt es mehrere Fälle, die auftreten können. Unter Umständen ist das Neuanlegen, Verschmelzen, Anpassen, Aufteilen oder Entfernen von Boxen nötig. Die möglichen Szenarien bei mehreren involvierten Boxen wurden im Modultest mit Hilfe von JUnit Tests geprüft. Dabei wurden Randfälle wie auch Normalfälle beachtet. Es konnte kein Fehlverhalten festgestellt werden. Auch beim folgenden manuellen Systemtest traten keine Fehler beim Testen der Funktionalität auf.

R-07: Eine Box soll genau einem Requirement zugeordnet sein.

Die Implementierung des Box Objekts enthält genau eine Requirement ID. Der Nutzer hat keine Möglichkeit diese Boxen beliebig zu bearbeiten. Die Bearbeitung erfolgt ausschließlich über die implementierten Klassen, die keine multiplen Zuweisungen erlauben. Möchte der Anwender einen Quellcodeabschnitt mit mehreren Requirements zugleich annotieren, so wird für jedes Requirement eine eigene Box angelegt. Ein Modultest dieses Requirements ist somit in diesem Fall obsolet. Beim manuellen Systemtest traten keine Boxen auf, die keinem Requirement oder mehreren Requirements zugeordnet waren. Dieses Requirement ist visuell verletzt, wenn zwei Requirements vom Nutzer oder vom Tool die gleiche Farbe zugewiesen wird. Die Boxen verlieren durch die gleiche Farbzuweisung jedoch nicht die Zugehörigkeit. Die visuelle Verletzung kann durch Zuweisung einer neuen Farbe durch den Nutzer aufgehoben werden. Auch tritt die zufällige Zuweisung der gleichen Farbe durch das Tool nur selten auf. Aus diesem Grund wurden keine Funktionen implementiert, die eine visuelle Verletzung verhindern.

R-08: Die Boxen sollen beliebige Start und Endpunkte im Quellcode haben können.

Die Boxen können beim ersten Zeichen der Quellcodedatei beginnen und bis zum letzten Zeichen reichen. Bei der Durchführung von manuellen Tests und Modultests für verschiedene Größen der Boxen treten keine Fehler auf. Die Modultests wurden als JUnit Tests durchgeführt. Dabei wurde geprüft ob die Boxen beliebige Start und Endpunkte haben können. Bei den manuellen Tests wurden Boxen verschiedener Größen angelegt (siehe Abbildung 8.1)

R-09: Die Boxen sollen beim ersten Zeichen, das kein Leerzeichen oder ein Tabulator ist, in der ersten Zeile der Box beginnen.
R-10: Die Boxen sollen beim letzten markierten Zeichen einer Zeile, das kein Leerzeichen oder Tabulator ist, enden.

R-09 und R-10 werden in verschiedenen Quellcodedateien mit verschiedenen Requirements getestet. Dabei fällt auf, dass formatierter, modularisiert strukturierter Quellcode problemlos dargestellt werden kann (Abbildung 8.1). Bei der Verwendung von schlecht formatiertem wahllos eingerücktem Quellcode stößt der Prototyp jedoch an seine Grenzen (Abbildung 8.2). Die Visualisierung von Traceability-Informationen in schlecht formatiertem Quellcode entspricht nicht R-23 und dafür ist der Prototyp auch nicht ausgelegt. Aufgrund von R-09 werden Boxen, die ausschließlich Leerzeichen oder Tabulatoren enthalten nicht angezeigt. Diese Boxen sind aber, wie in Abschnitt 6.2 erläutert, intern trotzdem als Boxen angelegt und gespeichert. Im manuellen Systemtest begannen und endeten alle Boxen visuell bei den vom Nutzer gewünschten Start- und Endpunkten.

R-11: Boxen sollen eine Mindestgröße von einem Zeichen haben.
R-15 Wird eine Box durch eine Änderung auf eine Länge kleiner gleich 0 reduziert, so soll die Box entfernt werden.

Diese beiden Requirements wurden durch Modultests mittels JUnit Tests geprüft. Einerseits wurde versucht Boxen der Größe 0 anzulegen, was nicht gelang. Andererseits wurde durch Bearbeitung bestehender Boxen erfolglos versucht, eine Box auf die Größe 0 zu bringen. Sobald eine Box die Mindestgröße von einem Zeichen unterschreitet, wird sie automatisch vom Tool entfernt. Auch beim Systemtest konnten keine Boxen der Größe 0 erzeugt werden.

R-12: Die Boxen sollen sich automatisch an die maximale Breite des eingeschlossenen Textes anpassen.

Durch visuelle Tests am laufenden Prototyp mit verschiedenen Kombinationen von Boxen wird die Funktion dieses Requirements geprüft. Die Boxbreite wird wie erwartet angepasst für verschiedenste Kombinationen. Mit einer Ausnahme: Werden in der ersten Zeile der Box nicht alle Zeichen ab dem ersten Zeichen der Zeile markiert, so ist die Berechnung der Boxbreite um diese fehlenden Zeichen verfälscht. In allen anderen Zeilen werden diese beispielsweise führenden Leerzeichen mitgezählt, wodurch es zu fehlerhaften Darstellungen kommen kann (Abbildungen 7.2 und 7.3). Dieser Programmfehler kann erstens durch Hinzufügen der führenden Zeichen in der ersten Zeile leicht behoben werden 7.4. Zweitens hat der verwendete Darstellungsalgorithmus seinen Ursprung in *EditBox*. Aus diesen beiden Gründen wird die nicht eigens programmierte Darstellung hier nicht angepasst.

```
47      public void setRequirement(TRCRequirement requirement) {
48          this.requirement = requirement; //123456789
49      }
```

Abbildung 7.2: Box wird korrekt angezeigt

```
47      public void setRequirement(TRCRequirement requirement) {
48          this.requirement = requirement; //1234567890
49      }
```

Abbildung 7.3: Box wird nicht korrekt angezeigt

```
47      public void setRequirement(TRCRequirement requirement) {
48          this.requirement = requirement; //1234567890
49      }
```

Abbildung 7.4: Nach dem Hinzufügen der führenden Leerzeichen in Zeile 47

R-13: Bei einer Änderung im Quellcode sollen die von der Änderung betroffen Boxen automatisch vom Tool angepasst werden.

Die Umsetzung dieses Requirements ist aufgrund der hohen Anzahl von möglichen Kombinationen und Randfälle fehleranfällig. Im Modultest wird diese Funktion des Prototyps deshalb gründlich geprüft. Das Löschen und Einfügen von Quellcode wird mit unterschiedlichen bestehenden Boxen verschiedener Requirements simuliert. Um die korrekte Funktionalität sicherzustellen wurden dabei in JUnit Tests alle möglichen Kombinationen von aktivierten und nicht aktivierten Requirements simuliert. Nach dem Testen und anschließendem erfolgreichem Debuggen der Funktion konnten beim Systemtest keine Fehler mehr festgestellt werden.

R-14: Gibt es noch keine Box an der Stelle der Änderung, so sollen neue Boxen angelegt werden, entsprechend der im zusätzlichen Fenster ausgewählten Requirements.

Durch Modul- und Systemtests wird gezeigt, dass das Tool auch diese Anforderung umsetzt. So wurde in den Modultests das Anlegen der Boxen für verschiedene, zufällig bestimmte Änderungen geprüft. Auch beim Systemtest konnten durch Änderungen Boxen wie erwartet angelegt werden.

R-16: Für die visuelle Zuordnung zwischen Boxen und Requirements wird das Medium Hintergrundfarbe *gewählt.*
R-17: Das Tool soll farbige Boxen im Editor der IDE anzeigen können.
R-18: Angezeigte Requirements und die zugehörigen Boxen müssen in der gleichen Farbe gekennzeichnet werden.

Sowohl in der *TRC View* als auch im *Tracing Overlay* wurden diese Anforderungen umgesetzt. Die Requirements werden in der *TRC View* in der Farbe hinterlegt, in der die zugehörigen Boxen im Quellcode eingefärbt sind. Dadurch lassen sich den Boxen visuell Requirements zuordnen und vice versa. Abbildung 8.5 zeigt einen Screenshot vom laufenden Prototyp in seiner eigenen Entwicklungsumgebung. Der Entwickler möchte eine neue Box im Quellcode einfügen, um die Zugehörigkeit von Quellcode und Requirement zu dokumentieren. Hierfür wurde bereits die Checkbox für das zusätzlich hinzuzufügende Requirement aktiviert. Nach der Auswahl des Bereichs im Editor und einem Klick auf den „Requirements setzen“ Knopf am oberen rechten Rand der *TRC View* wurde eine zusätzliche Box erstellt. (vgl. Abbildung 8.6)

Beim Systemtest des Tools wird deutlich, dass durch das Farbspektrum eine gewisse Grenze gegeben ist. Ab mehr als 20 Requirements in einer einzelnen Quellcodedatei steigt der Aufwand ausreichend visuell voneinander abgrenzbare Farben zu finden. Auch kann es beim initialen zufälligen Auswählen der Farben vorkommen, dass zwei unterschiedliche Requirements die gleiche, oder eine sehr

ähnliche Farbe zugewiesen bekommen. Dies kann jedoch leicht vom Anwender nachträglich korrigiert werden, indem durch Doppelklick auf ein Requirement in der *TRC View* die Farbe des selbigen geändert wird. Für diesen Prototyp wurden kein komplexer Farbzuweisungsalgorithmus programmiert, die den Kontrast zwischen benachbarten Boxen durch intelligente Farbzuweisung optimieren. Das ist eine der Aufgaben die für nachfolgende wissenschaftliche Arbeiten bestimmt sind.

R-19: Die Farbe von Requirements und deren zugehörigen Boxen soll vom Nutzer geändert werden können.

Durch Doppelklicken auf ein Requirement in der *TRC View* kann vom Nutzer ein Farbdialog geöffnet werden, in dem beliebige Farben ausgewählt werden können. Das Schließen des Farbdialogs führt zur Übernahme der gewählten Farbe in *TRC View* und *Tracing Overlay*. Im Systemtest wurden unterschiedliche Farbänderungen mit verschiedenen Requirements durchgeführt. Es liegt dabei im Ermessen des Nutzers, den Requirements ausreichend unterschiedliche Farben zuzuweisen um die Boxen unterscheiden zu können.

R-27: Der Anwender soll seinen Fokus mittels eines Layer-Systems auf vom Anwender ausgewählte Requirements setzen können.

Dieser Seiteneffekt ergab sich, wie bereits unter Abschnitt 6.2 dargelegt, während der Umsetzung. Dadurch, dass die Farbe des Requirements in der *TRC View* gleich der Farbe der zugehörigen Box im *TRC Overlay* im Editor sein soll, ist eine Umsetzung mit halbdurchsichtigen Boxen und Mischfarben nicht sinnvoll. Durch die iterative Erstellung der Boxen ergibt sich automatisch ein Layer-System, das vom Anwender manipuliert werden kann. Die Reihenfolge der Requirements lässt sich hierbei beliebig mittels der Knöpfe am oberen Ende der *TRC View* anpassen.

R-20: Wenn ein neues Requirement zu einer Quellcodedatei hinzugefügt wird, soll dem Requirement vom Tool einmalig eine zufällige Farbe zugewiesen werden.
R-22: Die zufällige, einem hinzugefügten Requirement zugewiesene Farbe (siehe R-20) soll eine Pastellfarbe sein.
R-21: Die Farbe eines Requirements soll nach dem ersten Zuweisen gespeichert werden.

Durch Systemtests mit verschiedenen neu angelegten Requirements kann bestätigt werden, dass diese Requirements vom Prototyp realisiert werden. Beim erstmaligen Anlegen eines *TRCRequirement* Objektes, das in der *TRC View* angezeigt wird, wird dem *TRCRequirement* eine zufällige Farbe zugewiesen. Diese Farbe wird gespeichert und überdauert so Dateiwechsel und Neustarts der IDE, was den Requirements einen höheren Wiedererkennungswert verschafft.

R-23: Das Tool soll den Benutzer motivieren, ebendieses benutzen zu wollen. Dazu soll es die Benutzeroberfläche des Nutzers visuell bereichern

Dieses Requirement müsste mit einer anonymen Studie belegt werden, die im Rahmen dieser Arbeit nicht vorgesehen ist. Es wurde jedoch immer unter der Prämisse einer vorzeigbaren Useability gearbeitet. Aus diesem Grund ist der Prototyp mit dem Abschluss dieser Arbeit für eine solche Studie geeignet.

R-24: Die Antwortzeiten für die unterschiedlichen Funktionen des Tools sollen jeweils maximal 0,5 s betragen.

Die Einhaltung dieser Antwortzeiten wurde durch Modul und Systemtests belegt. Der Prototyp konnte für die Funktionen zu R-06, R-13 und R-19 diese Antwortzeit einhalten. Beim Testen ergab sich, dass der zeitaufwändigste Vorgang das Zeichnen der Boxen in der IDE ist. Dabei liegt die dafür benötigte Zeit mit durchschnittlich 0,3 Sekunden jedoch noch im zulässigen Rahmen. Da die Methoden zum Zeichnen der Boxen nicht selbst implementiert, sondern von *EditBox* übernommen wurden, wird hier nicht weiter optimiert.

R-25: Implementierter Quellcode dieses Prototyps soll dokumentiert sein.

Die Stellen an denen die Requirements im Quellcode umgesetzt wurden, werden einerseits mit dem Prototyp markiert. Der Quellcode wird dabei visuell mit Traceability-Informationen annotiert. Andererseits tragen Dokumentationskommentare zur Wartbarkeit und Erweiterbarkeit bei und stellen Dokumentation für Nutzer, die die *TRC Dateien* nicht lesen können sicher.

R-26: Das Tool soll unabhängig von der Programmiersprache sein, in welcher der Programmierer arbeitet.

Mit Systemtests wird die Funktionalität für verschiedene Quellcodedateien in Java, wie auch Python geprüft. Da die *Eclipse IDE* primär für die Entwicklung von Java Applikationen gedacht ist, wird in der Standardversion kein Syntax-Highlighting für Python Quellcodedateien unterstützt. Python Quellcode kann aber angezeigt, und mit dem Tool mit Traceability-Informationen versehen werden.

Alle funktionalen Requirements konnten im Prototyp umgesetzt werden. Auch die nicht-funktionalen Requirements wurden bis auf R-23 eingehalten.

7.2 Auswertung der Umfrage

Um die selbst bestimmten Requirements an den Prototyp besser begründen zu können wurde während der Ausarbeitung des Tools eine Umfrage im universitären Umfeld durchgeführt. Sowohl Studenten, als auch wissenschaftliche Mitarbeiter der Friedrich-Alexander-Universität Erlangen-Nürnberg wurden zu der Umfrage eingeladen. Des Weiteren wurden einige Mitarbeiter von Industriepartnern der FAU eingeladen.

Die Intention der Wahl dieser Zielgruppe ist unter anderem durch die Stakeholder-Analyse in Abschnitt 4.1 begründet. Die Umfrage richtet sich an industrielle und akademische Programmierer, Softwarearchitekten und Studenten die überlegen, zukünftig in diesem Feld aktiv zu sein.

Es sind insgesamt 80 Rückmeldungen eingegangen, davon waren 78 verwendbare Antworten. Zwei Personen entsprachen nicht der Zielgruppe.

Die Umfrage beginnt mit einer Abfrage des Beschäftigungsverhältnisses. Zur Auswahl stehen Informatik Student(in), Projektmanager(in), Softwarearchitekt(in), Programmierer(in), Wissenschaftler(in) und Sonstiges. Bei einer Eingabe in das Feld *Sonstiges* wird der Benutzer jedoch darum gebeten, eine weniger spezifische Auswahl zu Treffen, oder die Umfrage zu verlassen.

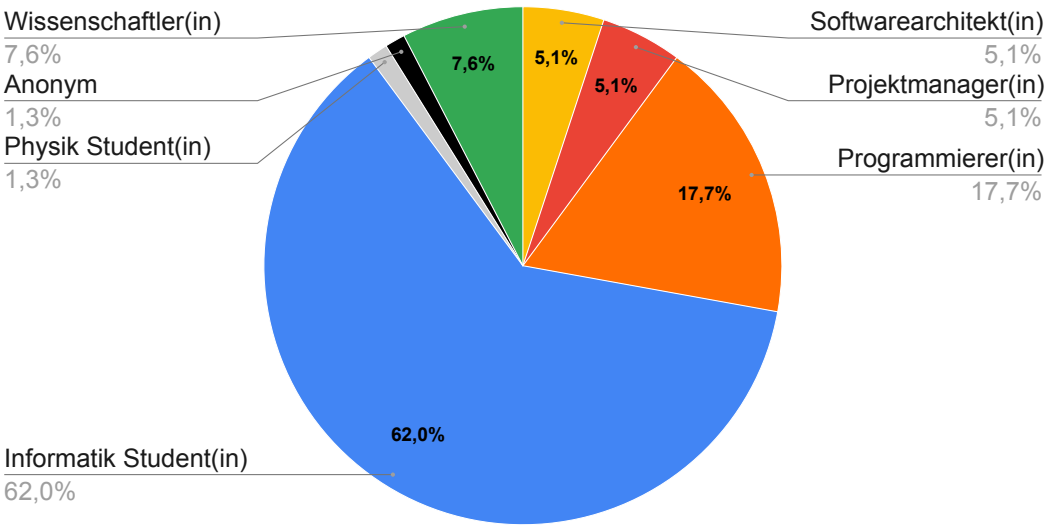

Abbildung 7.5: Beschäftigungsverhältnisse der Umfrageteilnehmer

Die Teilnehmer setzen sich aus 62,0 % Studenten und 35,3 % aus Personen, die bereits im Umfeld der Informatik arbeiten zusammen. 2,7 % Der Teilnehmer haben die Umfrage verlassen, ohne die Fragen zu beantworten. (vgl. Abbildung 7.5)

Im weiteren Verlauf wird das Interesse für bestimmte Features abgeprüft, die mittels des implementierten Tools generell umsetzbar oder denkbar sind. Die Umfrage gab den Teilnehmern die Wahl auf einer Skala von 0 (nicht interessant) bis 10 (sehr interessant) anzugeben, für wie interessant sie ein bestimmtes Feature hielten.

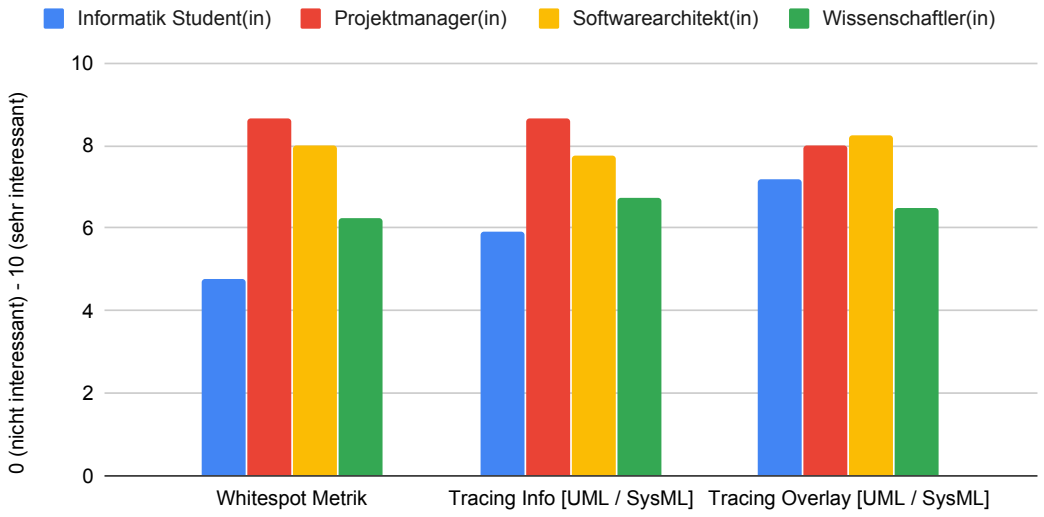

Abbildung 7.6: Durchschnittliches Interesse an Software Architektur Features

Bei genauerer Betrachtung von Abbildung 7.6, die das durchschnittliche Interesse an der Whitespot Metrik, der Tracing Info [UML / SysML] und dem Tracing Overlay [UML / SysML] der einzelnen Gruppen darstellt, fällt Folgendes auf:

Die Studenten, die weniger Erfahrung im Umfeld der industriellen Informatik haben als die anderen befragten Gruppen, zeigen weniger Interesse an einer Whitespot Metrik, als die Gruppen, die in der Industrie vertreten sind. Bei Projektmanagern und Software Architekten zum Beispiel, ist die Whitespot Metrik hingegen sehr beliebt. Die Studenten fanden ihrerseits das Tracing Overlay für UML oder SysML unter den Tools für Software Architekten am interessantesten. Die Gesamtdurchschnittswerte für alle Befragten sind 6,1 bei der Whitespot Metrik, 6,7 bei der Tracing Info und 7,3 von 10 Punkten beim Tracing Overlay für UML oder SysML.

Es lässt sich also ein mäßiges Interesse der Befragten an den Features für Software Architekten verzeichnen.

In Abbildung 7.7 ist das durchschnittliche Interesse an Features des Tracing Overlays für die IDE der einzelnen Gruppen visualisiert. Am interessantesten finden die Gruppen die Möglichkeit, während des Programmierens Informationen zu den gerade relevanten Requirements erhalten zu können. Das durchschnittliche Interesse liegt hier bei 7,6 von 10 möglichen Punkten. Auch die Gruppen der

Softwarearchitekten, Programmierer und Wissenschaftler zeigen hier ihr größtes Interesse.

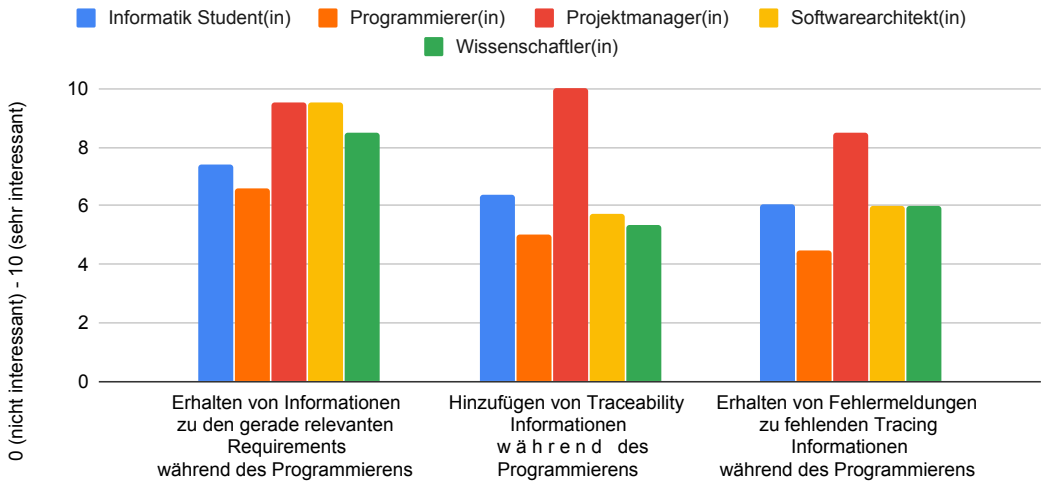

Abbildung 7.7: Durchschnittliches Interesse an Features für Programmierer

Das Hinzufügen von Traceability-Informationen während des Programmierens ist besonders bei Projektmanagern hoch angesehen, und Programmierer stehen der Option neutral gegenüber. Nachdem die Programmierer die Leidtragenden eines solchen Features wären, da sie den Mehraufwand hätten, ist ein neutrales Ergebnis an dieser Stelle als positiv zu vermerken. Das durchschnittliche Interesse liegt hier bei 6,2 Punkten. Insgesamt am wenigsten interessant wird das Erhalten von Informationen zu fehlenden Traceability-Informationen im Quellcode angesehen. Der Durchschnitt liegt hier bei 5,8 Punkten.

Insgesamt wird somit das Erhalten von Informationen zu relevanten Requirements beim Programmieren als am interessantesten angesehen.

Im weiteren Verlauf der Umfrage wird um eine prozentuale Abschätzung gebeten, wie viel Zeit es kosten würde, die Traceability-Informationen mit einem zu dem Zeitpunkt noch fiktiven Tracing Overlay während des Programmierens hinzuzufügen.

Die durchschnittlichen Ergebnisse der einzelnen Gruppen sind in Abbildung 7.8 visualisiert. Die genaue prozentuale Angabe dieses Aufwands ohne ein konkretes Beispiel zu nennen erwies sich als schwer abschätzbar. Mögliche Eingaben waren positive Ganzzahlen für den geschätzten zeitlichen Mehraufwand, und negative wie positive Ganzzahlen für die geschätzte Einlesezeitersparnis in Prozent. Dennoch kann aus den erhaltenen Durchschnittswerten Folgendes geschlossen werden:

Der Gesamtdurchschnitt aller Gruppen für den geschätzten Mehraufwand war 17,9 % und somit niedriger als die von allen Gruppen insgesamt abgeschätzten

24,2 % der erwarteten Ersparnis an Zeit, die beim Einlesen von mit visuellen Informationen angereichertem Quellcode nötig ist. Die Gruppen sind sich alle einig, dass es vermutlich eine Erleichterung beim Einlesen in visuell angereicherten Quellcode gibt. Beim Schreiben des Quellcodes wird ein Mehraufwand erwartet, wenn die Informationen gleich mit hinzugefügt werden sollen. Da sich der abgeschätzte prozentuale Mehraufwand in etwa mit der abgeschätzten Ersparnis deckt, ist dies ein weiteres positives Ergebnis.

Unter der Annahme, dass gleiche Quellcode-Abschnitte öfter eingelesen werden müssen, als sie programmiert werden, ist eine erwartete Ersparnis von 24,2 % eine große Zahl. Angenommen ein Programmierer bräuchte normalerweise 60 Minuten um sich in eine Quellcodedatei einzulesen, so könnte er unter Annahme der 24,2 % 15 Minuten an Zeit sparen. Auch die Nachfolger des Programmierers und seine Kollegen würden aber von dieser Ersparnis profitieren, somit ist anzunehmen, dass insgesamt weit mehr Zeit gespart, als investiert werden würde.

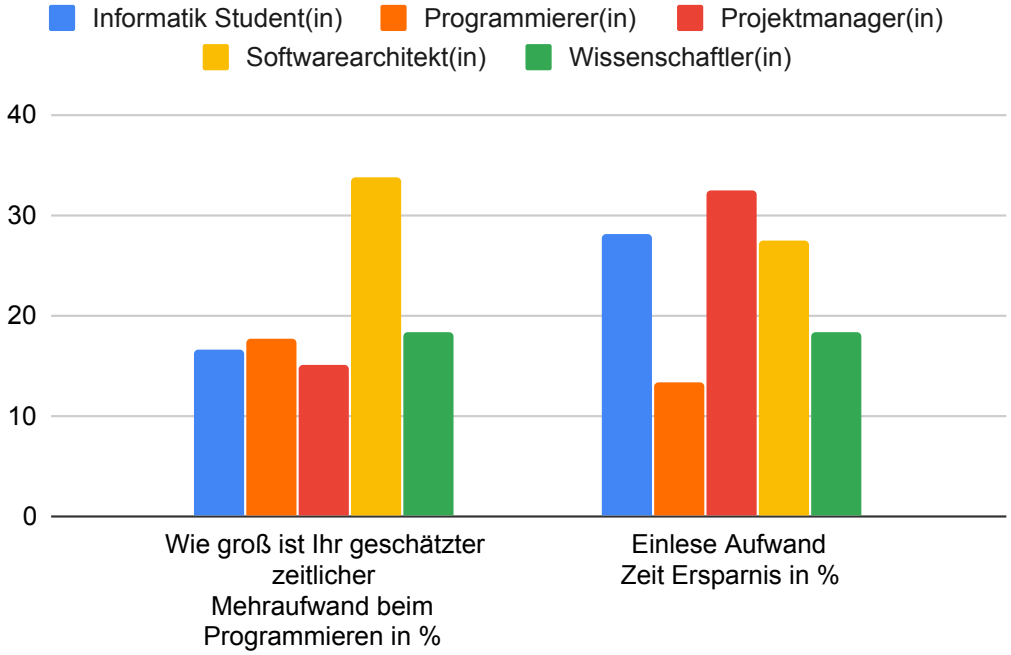

Abbildung 7.8: Durchschnittlicher Prozentualer Mehraufwand

Auf der letzten Seite der Umfrage hatte der Nutzer die Möglichkeit, beliebiges Feedback zur Umfrage zu geben. Beispielsweise wurde dort die Aussage laut, dass Requirements oft nicht existent oder zu lückenhaft sind, um sinnvoll damit verknüpfen zu können. Dieses Tool soll gerade das Vorhandensein und die Vollständigkeit der Requirements unterstützen, indem es Programmierer dazu motiviert Traceability-Informationen direkt bei der Entwicklung mit anzulegen. Außerdem soll das Tool nach Möglichkeit bereits in der Design Phase eines Projektes mit in Betracht gezogen werden. Beim Generieren von Quellcode aus UML

Diagrammen könnten bereits .trc Dateien mit generiert werden. Dies bedeutet eine Erleichterung beim Programmieren, als auch beim Hinzufügen der Traceability-Informationen.

Einige Begrifflichkeiten sollten genauer erklärt werden und mehr Beispielgrafiken wurden gewünscht. Es war jedoch niemand unter den Antworten der erklärt hat, dass die Umfrage dadurch unverständlich wurde, somit sind Einflüsse auf die Ergebnisse durch die teilweise fehlenden Begriffsklärungen unwahrscheinlich. Unter Verwendung des jetzt entwickelten Tools können zukünftig beliebig viele Beispielgrafiken erstellt werden.

In mehreren Antworten wurden Bedenken laut, was passiert, wenn einem Quellcode-Abschnitt mehrere Requirements zugeordnet werden müssen. An dieser Stelle kann nun auf das umgesetzte Layer-System (Abbildungen 6.3 und 6.4) verwiesen werden. Es können problemlos beliebig viele Requirements auf den gleichen Quellcode-Abschnitt verweisen.

Ich bedanke mich herzlich für die Glückwünsche und für weitere Ideen die an dieser Stelle eingereicht wurden, auf die unter Kapitel 8 mit eingegangen wird.

Es kann somit zusammenfassend unter Berücksichtigung der kleinen Stichprobengröße gesagt werden, dass es ein grundlegendes Interesse an dem Tracing Overlay gibt (Abb. 7.7). Des Weiteren besteht Interesse an möglichen Erweiterungen des Tools, insbesondere für Softwarearchitekten.

8 Zusammenfassung und Ausblick

Die Forschungsfragen dieser Arbeit können jetzt beantwortet werden:

1. *Wie können Requirements auf Quellcode-Ebene markiert, visuell dargestellt und verwaltet werden?*

   Die in dieser Arbeit dargelegten Grundlagen und Überlegungen geben eine mögliche Antwort auf diese Frage.

2. *Wie lässt sich eine derartige Software realisieren?*

   Der im Rahmen dieser Arbeit umgesetzt Prototyp stellt eine Möglichkeit dar, visuelle Annotation von Quellcode mit Traceability-Informationen zu implementieren.

Um diese Forschungsfragen wissenschaftlich begründet beantworten zu können wurden in Kapitel 2 grundlegende Begrifflichkeiten zum Thema Traceability-Daten erklärt. Zudem wurden einige Ideen zu Verbesserungsmöglichkeiten in der Domäne der Traceability vorgestellt. In Kapitel 3 erfolgte die Klassifizierung bestehender Ansätze zum Thema visuelle Annotation von Quellcode mit Traceability-Informationen und die Abgrenzung dieser Ansätze von der vorliegenden Arbeit. Es wurden allgemeine Ansätze zur Anreicherung von Quellcode mit Farben als zusätzliche Informationsquelle betrachtet. Darüber hinaus wurden bestehende Ansätze zur Visualisierung von Traceability-Informationen analysiert. Im Verlauf der Recherche wurden keine Ansätze gefunden, die bereits Requirements auf Quellcode-Ebene visualisieren.

Auf der Basis dieser Informationen wurde in Kapitel 4 eine Stakeholder-Analyse durchgeführt. Die Stakeholder-Analyse ergab Programmierer und Softwarearchitekten als direkte Zielgruppe und Projektmanager als erweiterte Zielgruppe. Die erweiterte Zielgruppe kann besonders von dem Gegenstand dieser Arbeit profitieren, wenn weitere Forschung in diesem Umfeld erfolgt. Die Aufgabe des Prototyps, Quellcode-Abschnitten visuell Requirements zuzuordnen, wurde bestimmt. Aus der textuellen Beschreibung der Aufgabe des Tools wurden zunächst Anwendungsfälle und dann konkrete Requirements an den Prototyp abgeleitet. Unter Berücksichtigung der Requirements wurde die Architektur für den Prototyp ent-

wickelt und in Kapitel 5 dokumentiert. Auf der Basis dieser Architektur konnte die Umsetzung des Prototyps basieren. Das Tool konnte bis auf kleinere Abweichungen, die in Kapitel 6 näher erläutert wurden, wie geplant umgesetzt werden.

Abschließend wurden in Kapitel 7 die zuvor aufgestellten Requirements gegen den Prototyp getestet und die Testergebnisse dokumentiert. Die Testergebnisse zeigten, dass der Prototyp die Requirements erfüllte. Die Auswertung der Umfrage ergab, dass schnelleres Einlesen von Quellcode durch die verbesserte Übersicht erwartet wird. Es ist also eine Erleichterung für Programmierer zu erwarten. Somit ist das Ziel der Arbeit, Programmierern einen zusätzlichen Anreiz zu geben, mehr auf Traceability-Informationen zu achten theoretisch gelungen. Damit dieser Ansatz auch praktisch gelingen kann, ist weitere Forschung im Umfeld der visuellen Annotation von Quellcode mit Traceability-Informationen erforderlich.

Ausblick Die weitere Forschung in diesem Feld sollte die folgenden Ansätze mit berücksichtigen: Die unter Abschnitt 2.4 erläuterten Erweiterungsmöglichkeiten des Prototyps, als auch darüber hinaus folgende Gedanken.

An dieser Stelle sei auch erneut auf *(R-23) Das Tool soll den Benutzer motivieren, ebendieses benutzen zu wollen. Dazu soll es die Benutzeroberfläche des Nutzers visuell bereichern* verwiesen. Zum Testen dieses Requirements ist eine anonyme Studie nötig, die die Funktion dieses Prototypen in einem Feldtest analysiert.

Einer der Vorteile davon, den Quellcode visuell mit Traceability-Informationen anzureichern, ist die dadurch für den Nutzer sichtbare Modularisierung. Beispielsweise im Falle von Änderungsanfragen an das gerade entwickelte Projekt können so schnell die betroffenen Stellen im Quellcode ausgemacht werden, ohne überhaupt Quellcode einlesen zu müssen. R-23 und die objektive Useability des Tools bleiben aber wissenschaftlich zu prüfen.

Für diesen Prototypen wurden keine komplexen Farbzuweisungsalgorithmen programmiert, die den Kontrast zwischen benachbarten Boxen durch intelligente Farbzuweisung optimieren. Auch Farbkollisionen durch gleiche Farben in unterschiedlichen Requirements werden nicht programmatisch verhindert. Die Implementierung solcher Funktionalitäten könnte die Useability des Prototypen weiter verbessern. Auch sei an dieser Stelle erwähnt, dass der vorgestellte Ansatz zu magischen Kommentaren im Quellcode (5.2) hauptsächlich aus Kostengründen nicht verfolgt wurde. Dieser Ansatz stellt eine weitere mögliche Forschungsrichtung dar.

In der Umfrage wurde auch eine Idee eingereicht, die nicht vorenthalten werden soll: Ein Tool, das sämtliche Quellcode-Abschnitte, die zu einem Requirement zugeordnet wurden, zusammengefasst anzeigt. Dazu können die vom *TRC Overlay* angelegten Informationen verwendet werden. Diese Darstellung könnte ähnlich der Übersicht bei *git* aussehen, die sämtliche Änderungen aus einem Commit anzeigt. Mit einem solchen Tool kann ein Überblick über den zu einem Requirement

zugehörigen Quellcode gewonnen werden.

Mittels Mustererkennung könnten wiederkehrende Programmieransätze für ähnliche Requirements gefunden werden, um sie beispielsweise während des Programmierens vorzuschlagen. Insbesondere bei sehr feingranularen Requirements würde sich ein solches Tool bereits in den ersten Monaten der Laufzeit als nützlich erweisen. Feingranulare Requirements sind zumeist in wenigen Zeilen Quellcode realisierbar. Mit einer zunehmend wachsenden Datenbasis können dann womöglich auch Informationen zu komplexeren oder verschachtelten Requirements geliefert werden. Zusätzlich ist denkbar, alternative Umsetzungsmöglichkeiten zu bestimmten Requirements im Quellcode finden zu können. Diese könnten beispielsweise aus älteren Projekten oder anderen Projekten stammen. Solche Alternativen könnten dann im „Back-to-Back-Test“ verglichen oder ausgetauscht werden.

Die gerade vorgestellten Ideen lassen sich am wirkungsvollsten in einem von Beginn an dokumentierten und geplantem Umfeld umsetzen. Auch die Einführung des in dieser Arbeit entwickelten Prototyps ist in einem solchen Umfeld leichter realisierbar. Somit sind das Tool und die Ideen besonders für neue Systeme, die gerade erst in der Entstehungsphase sind, interessant. Mit großem manuellem initialen Aufwand kann dieses Tool aber auch in bestehenden Softwareprojekten zum Einsatz kommen. Die vorgestellten Ansätze zur möglichen Erweiterung des Tools können dabei von den großen Mengen bereits vorhandenen Quellcodes profitieren.

Anhang A Screenshots für die Evaluation des Prototyps

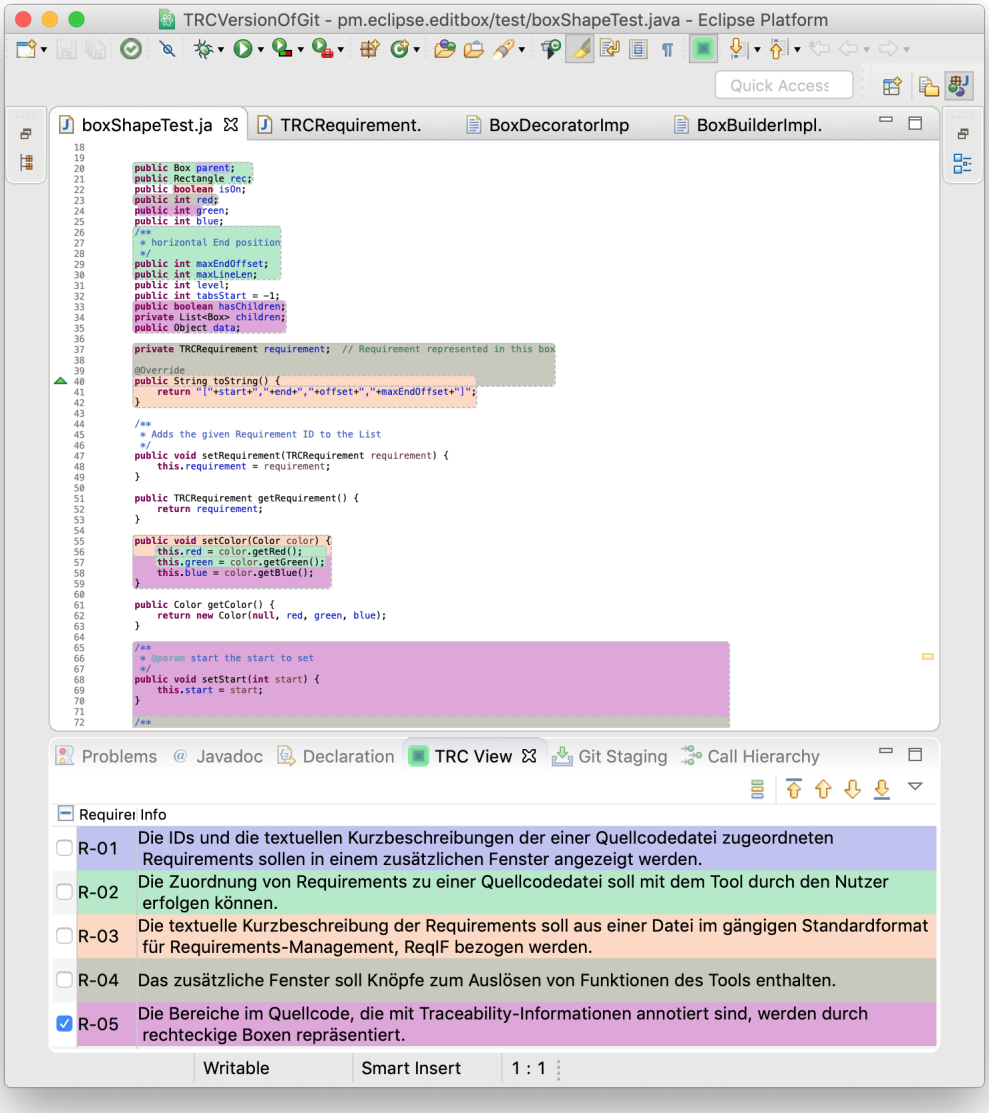

Abbildung 8.1: Test von verschiedenen Box Formen

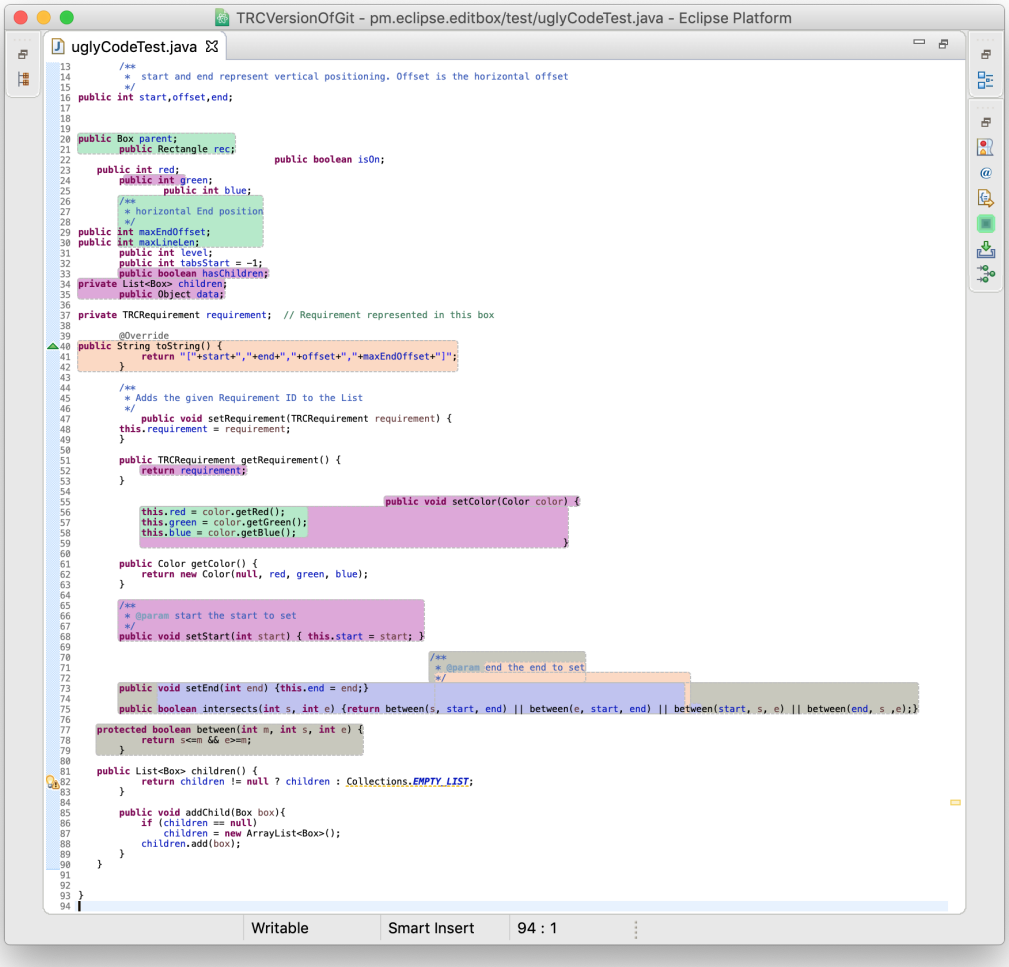

Abbildung 8.2: Test von Boxen in schlecht formatiertem Quellcode

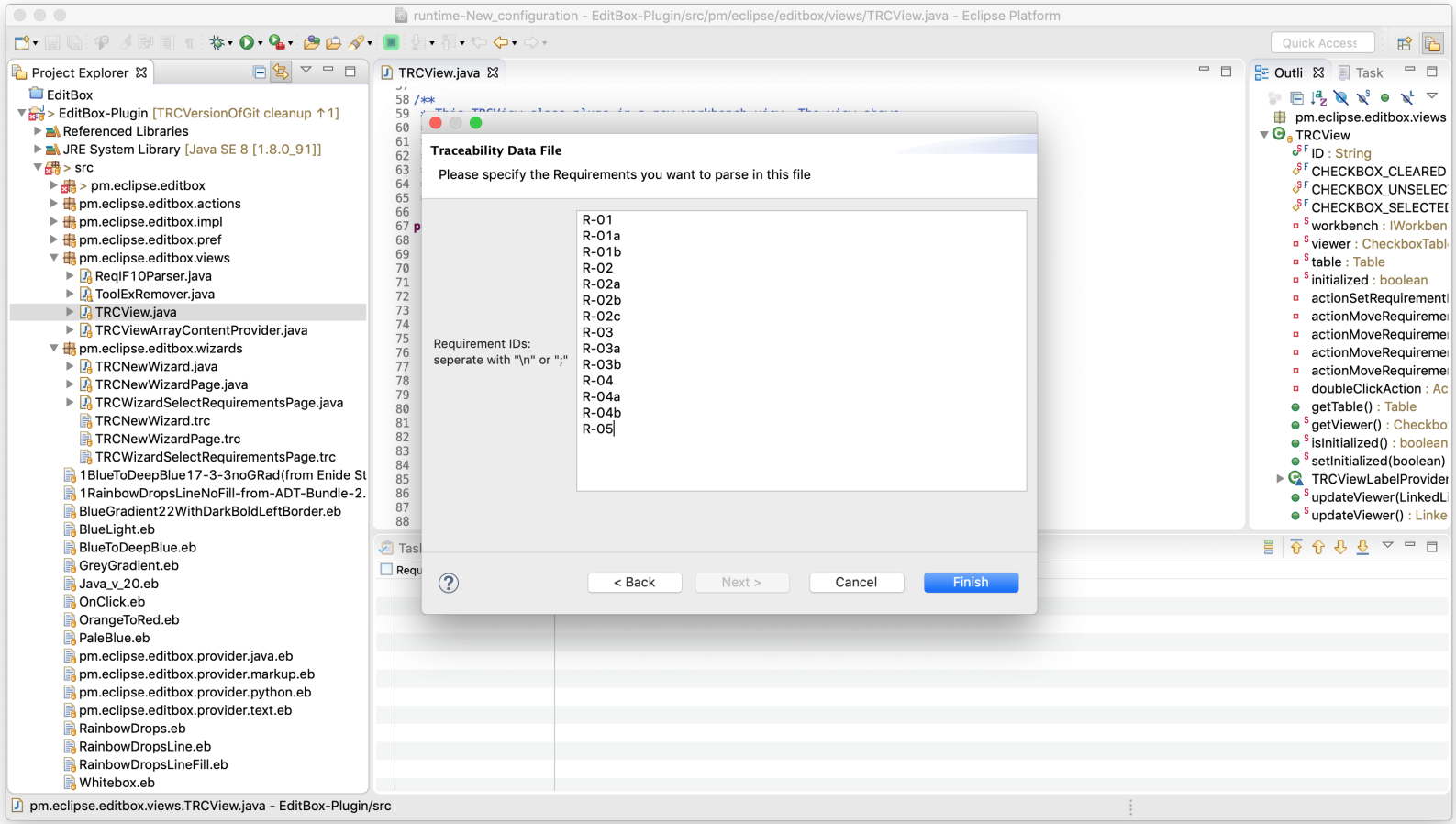

Abbildung 8.3: Nutzereingabe der anzuzeigenden Requirements

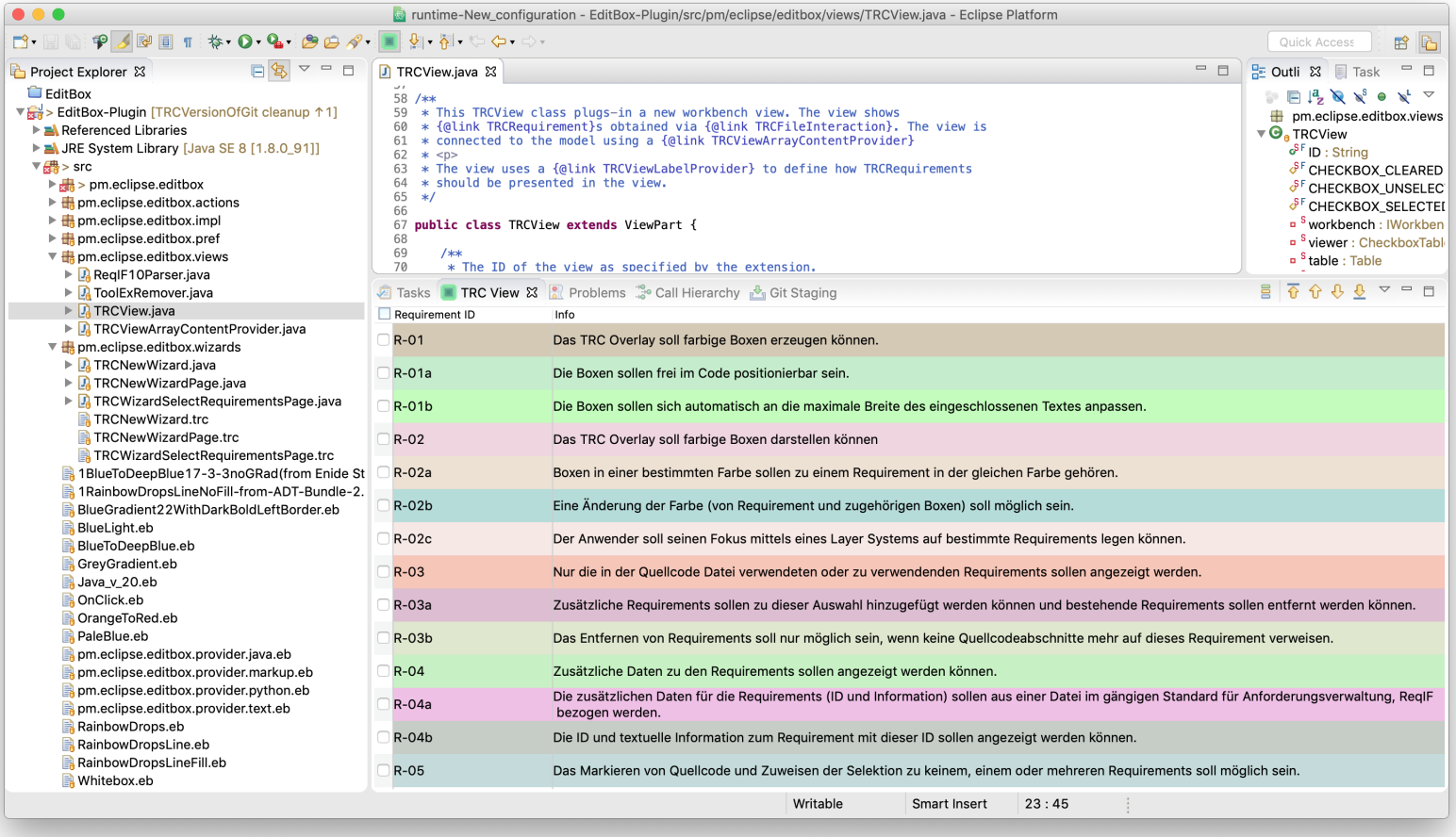

Abbildung 8.4: Anzeige der Requirements in der *TRC View*

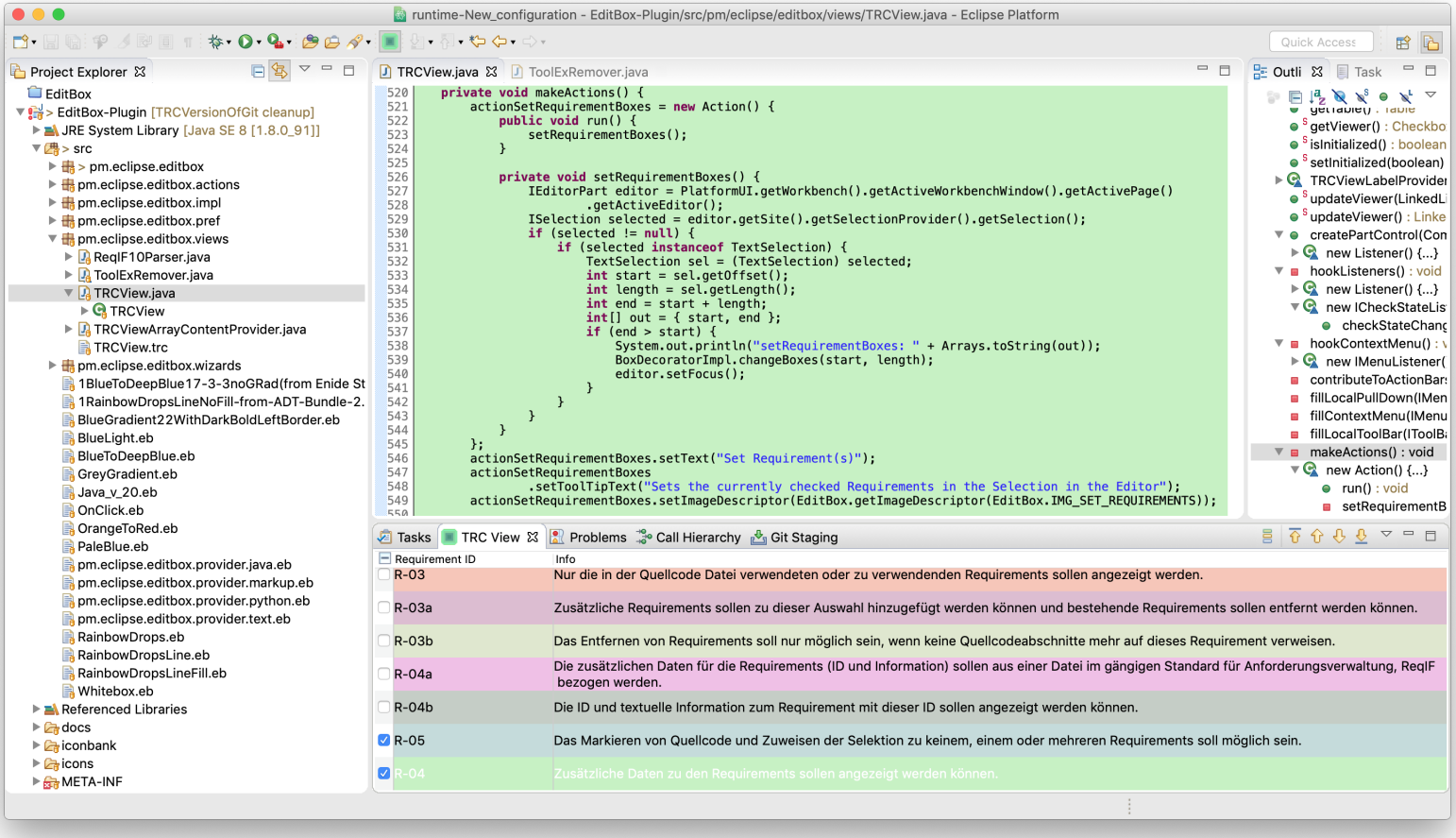

Abbildung 8.5: Zu verwendende Requirements sind in der *TRC View* ausgewählt

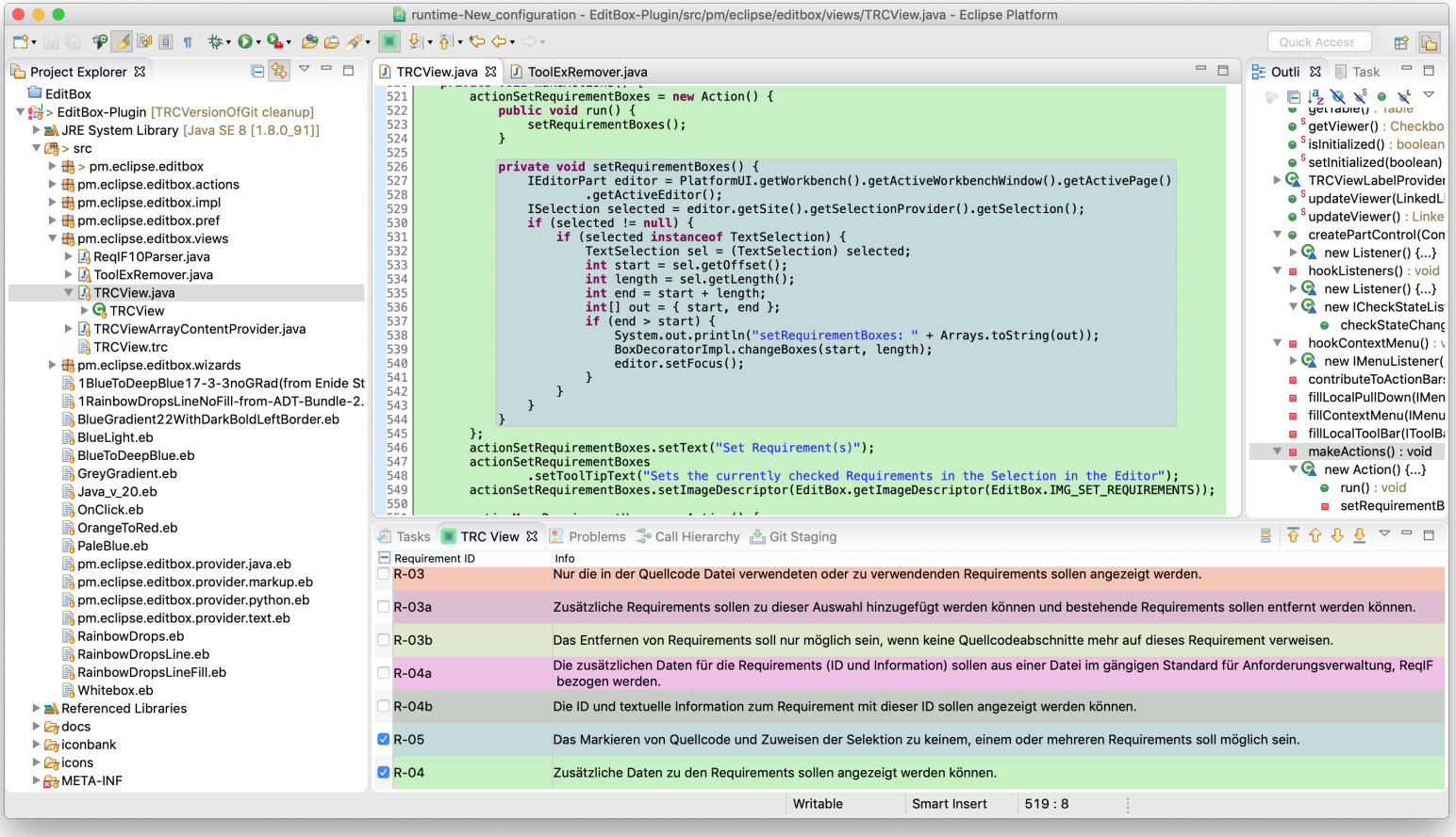

Abbildung 8.6: Neue Box wurde erstellt

Literaturverzeichnis

Arkley, P. & Riddle, S. (2005). Overcoming the Traceability Benefit Problem. In *13th IEEE International Conference on Requirements Engineering (RE'05)* (S. 385–389). IEEE.

Beck, K., Cunningham, W. & Services, W. S. (1989). A Laboratory for Teaching Object-Oriented Thinking. (S. 1–6).

Beelders, T. R. & du Plessis, J.-P. L. (2016). Syntax Highlighting as an Influencing Factor When Reading and Comprehending Source Code. *Journal of Eye Movement Research*, *9*(1), 11. doi:10.16910/jemr.9.1.1

Brandt, J., Pattamatta, V., Choi, W., Hsieh, B. & Klemmer, S. R. (2010). *Rehearse: Helping Programmers Adapt Examples by Visualizing Execution and Highlighting Related Code.*

Brysbaert, M. (2019). How Many Words Do We Read per Minute? A Review and Meta-Analysis of Reading Rate.

CENELEC. (2011). EN 50128: Railway Applications: Software for Railway Control and Protection Systems. *European Committee for Electrotechnical Standardization.* Verfügbar unter https://standards.globalspec.com/std/1678027/EN%5C%2050128

Chacon, S. & Straub, B. (2014). *Pro Git.* Apress. Verfügbar unter https://git-scm.com/

Delater, A. (2013). *Tracing Requirements and Source Code During Software Development* (Dissertation, Naturwissenschaftlich-Mathematische Gesamtfakultät, Ruprecht-Karls-Universität, Heidelberg, Deutschland).

Dyer, S., Forsythe, A., Maltz, A., Morin, D., Tobenkin, S. & Walker, D. (2017). ACES Leadership Response to "ACES – Retrospectives and Enhancements". Verfügbar unter https://acescentral.com/uploads/default/original/1X/14adba28da022a506541e9d717203e75dc207faa.pdf

Egyed, A. & Grunbacher, P. (2004). Identifying Requirements Conflicts and Cooperation: How Quality Attributes and Automated Traceability Can Help. *IEEE software*, *21*(6), 50–58.

Gerber-Menz, V. (2014). *Übernahme von Born-Digital Fotobeständen Und Fotografennachlässen Ins Archiv.* Hochschule für Technik und Wirtschaft, Arbeitsbereich Informationswissenschaft.

Hakala, T., Nykyri, P. & Sajaniemi, J. (2006). An Experiment on the Effects of Program Code Highlighting on Visual Search for Local Patterns. In *PPIG* (S. 10).

IBM Corp. (2014). Übersicht über Rational DOORS. Verfügbar 20. Dezember 2019 unter https://www.ibm.com/support/knowledgecenter/de/SSYQBZ_9.6.1/com.ibm.doors.requirements.doc/topics/c_welcome.html

Jastram, M. (2017). *ReqIF Studio: Requirements Engineering Platform Based on Eclipse*. CreateSpace Independent Publishing Platform.

Khaled, R., Noble, J. & Biddle, R. (2003). InspectJ: Program Monitoring for Visualisation Using AspectJ. In *Proceedings of the 26th Australasian Computer Science Conference* (Bd. 16, S. 359–368). Australian Computer Society, Inc.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J.-M. & Irwin, J. (1997). Aspect-Oriented Programming. In *European Conference on Object-Oriented Programming* (S. 220–242). Springer.

Krause, J. (2019). *Nachverfolgbarkeit zwischen Anforderungen und Code* (Masterarbeit, Technische Universität Dresden).

Larson, E. (2014). I Still Don't Have Time to Manage Requirements: My Project Is Later than Ever, Nordamerika, Phoenix: Project Management Institute. Verfügbar unter https://www.pmi.org/learning/library/poor-requirements-management-source-failed-projects-9341

Lilienthal, C. (2019). *Langlebige Software-Architekturen: Technische Schulden Analysieren, Begrenzen Und Abbauen*. dpunkt. verlag.

Lücker, M. (2019). Keine Infos Aus Dem All: Welche Folgen Der Galileo-Ausfall Hat. Verfügbar unter https://www.tagesspiegel.de/gesellschaft/panorama/keine-infos-aus-dem-all-welche-folgen-der-galileo-ausfall-hat/24595488.html

Marcus, A. & Poshyvanyk, D. (2005). The Conceptual Cohesion of Classes. In *21st IEEE International Conference on Software Maintenance (ICSM'05)* (S. 133–142). IEEE.

Moha, N., Gueheneuc, Y.-G., Duchien, L. & Le Meur, A.-F. (2009). Decor: A Method for the Specification and Detection of Code and Design Smells. *IEEE Transactions on Software Engineering*, *36*(1), 20–36.

OMG. (2016). *Requirements Interchange Format (ReqIF) Specification* (Technischer Bericht Nr. Version 1.2). Object Management Group. Verfügbar unter https://www.omg.org/spec/ReqIF/

PMI. (2014a). *Requirements Management: Core Competency for Project and Program Success*. Project Management Institute. Nordamerika, Newtown Square. Verfügbar unter https://www.pmi.org/learning/thought-leadership/pulse/core-competency-project-program-success

PMI. (2014b). *The High Cost of Low Performance*. Project Management Institute. Nordamerika, Newtown Square. Verfügbar unter https://www.pmi.org/learning/thought-leadership/pulse/the-high-cost-of-low-performance-2014

Quinlan, P. T. & Humphreys, G. W. (1987). Visual Search for Targets Defined by Combinations of Color, Shape, and Size: An Examination of the Task Constraints on Feature and Conjunction Searches. *Perception & psychophysics*, *41*(5), 455–472.

Ramesh, B. (1998). Factors Influencing Requirements Traceability Practice. *Communications of the ACM*, *41*(12), 37–44.

Ramesh, B. & Jarke, M. (2001). Toward Reference Models for Requirements Traceability. *IEEE transactions on software engineering*, *27*(1), 58–93.

Roque, R. V. (2007). *OpenBlocks: An Extendable Framework for Graphical Block Programming Systems* (Master Thesis, Massachusetts Institute of Technology).

VDA QMC Working Group 13 & Automotive SIG. (2017). *Automotive SPICE® Process Assessment / Reference Model*. Verfügbar unter http://www.automotivespice.com/

Wang, R. Y., Ziad, M. & Lee, Y. W. (2002). *Data Quality*. New York: Kluwer Academic Publishers.

Winkler, S. & Pilgrim, J. (2010). A Survey of Traceability in Requirements Engineering and Model-Driven Development. *Software & Systems Modeling*, *9*(4), 529–565. doi:10.1007/s10270-009-0145-0