Institut für Parallele und Verteilte Systeme

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Synchronisation eingebetteter Architekturentscheidungen mit Architekturentscheidungs-programmen

Tobias Boceck

Studiengang:	Softwaretechnik
Prüfer/in:	Prof. Dr. Bernhard Mitschang
Betreuer/in:	Dr. rer. nat. Oliver Kopp
Beginn am:	25. Januar 2017
Beendet am:	25. Juli 2017
CR-Nummer:	D.2.7

Kurzfassung

Diese Arbeit behandelt das Thema der Synchronisation von quellcode-intern und quellcode-extern dokumentierten Softwarearchitekturentscheidungen. Einerseits gibt eine Java-Annotation, welche es erlaubt Architekturentscheidungen im Java-Quellcode zu hinterlegen, andererseits können Architekturentscheidungen auch mit dem Modellierungswerkzeug AD-Mentor modelliert und in einem Versionierungsrepository hinterlegt werden. Da Architekturentscheidungen somit an verschiedenen Stellen dokumentiert werden können, ist es sinnvoll diese zu synchronisieren. In dieser Arbeit wird ein Konzept zur Synchronisation von eingebettete Architekturentscheidungen mit modellierten Architekturentscheidungen anhand eines Versionierungsrepository vorgestellt und implementiert. Zur Verifikation des erstellten Konzepts wurde eine Fallstudie anhand des implementierten Synchronisationsprogramms durchgeführt und ausgewertet.

Inhaltsverzeichnis

1. Einleitung **13**

2. Architekturentscheidungen **15**
2.1. Dokumentation von Architekturentscheidungen 15
2.2. Eingebettete Architekturentscheidungen 19

3. Toolunterstützung zum Verwalten von Architekturentscheidungen **23**
3.1. AD-Mentor 23
3.2. SE-Repo 25

4. Synchronisation von eADL mit einem SE-Repo **29**
4.1. Übersicht der Synchronisationskomponenten 29
4.2. Anforderungen an die Synchronisation 30
4.3. Herausforderungen der Synchronisation 31
4.4. Einschränkungen der Synchronisation 32

5. Konzept der Synchronisation **35**
5.1. Zentrale Datenverwaltung 35
5.2. Synchronisationswerkzeug 35
5.3. Fallunterscheidung bei geänderten Architekturentscheidungen 36
5.4. Änderungen pullen 39
5.5. Änderungen committen 41

6. Implementierung des Synchronisationsprogramms **43**
6.1. Architektur und Implementierung 43
6.2. Voraussetzungen 45
6.3. Bedienung 45
6.4. Funktionsumfang 45

7. Fallstudie **55**
7.1. Vorbereitung und Durchführung 55
7.2. Auswertung 55

8. Zusammenfassung und Ausblick **57**

A. Anhang **59**
A.1. Fallstudie 59

Literaturverzeichnis **65**

Abbildungsverzeichnis

2.1. Grafische Darstellung des Modells eines Y-Statements 16

3.1. Inhalt eines Problem-Knotens bei einem modellierten Y-Statements durch mit AD-Mentor beim Beispiel des Caching der Darstellungen von Literaturentscheidungen in JabRef 24
3.2. Inhalt eines Options-Knotens bei einem modellierten Y-Statements durch mit AD-Mentor beim Beispiel des Caching der Darstellungen von Literaturentscheidungen in JabRef 25
3.3. Abbildung einer modellierten Architekturentscheidung auf die Felder eines Y-Statements 26
3.4. Modellierung des ausgewählten Options-Knoten mit AD-Mentor beim Beispiel des Caching der Darstellungen von Literaturentscheidungen in JabRef 27
3.5. Modellierung eines verworfenen Options-Knoten mit AD-Mentor beim Beispiel des Caching der Darstellungen von Literaturentscheidungen in JabRef 28
3.6. Benutzerinterface des SE-Repo Connectors zum einchecken von Änderungen der Architekturentscheidungen 28

4.1. Die Grafik zeigt, welche Rollen die Architekturentscheidungen beeinflussen und wie die benutzten Komponenten miteinander in Beziehung stehen 30
4.2. Die Grafik zeigt detailliert, welche Rollen die Architekturentscheidungen beeinflussen und wie die benutzten Komponenten kommunizieren 31

5.1. Bildliche Darstellung, wie lokale Änderungen an Architekturentscheidungen gegenüber den entfernten Architekturentscheidungen im SE-Repo interpretiert werden sollen 37
5.2. Bildliche Darstellung, wie entfernten Änderungen an den Architekturentscheidungen im SE-Repo gegenüber den lokalen Architekturentscheidungen interpretiert werden sollen 38
5.3. Bildliche Darstellung, wie lokale Änderungen an Architekturentscheidungen gegenüber entfernten Änderungen der Architekturentscheidungen im SE-Repo interpretiert werden sollen 39
5.4. Konzept zum Pullen von Architekturentscheidungen für die unterschiedlichen Fälle **F2** bis **F4** 40
5.5. Konzept zum Committen von Architekturentscheidungen für die unterschiedlichen Fälle **F2** bis **F4** 42

6.1. Das UML Paketdiagramm zeigt die Trennung zwischen dem Kommando Paket, welches die Benutzerschnittstelle darstellt, und dem Logik Paket, welches mit den Model Paketen arbeitet 44
6.2. Der Konflikt Manager bietet eine grafische Benutzeroberfläche zum komfortablen Beheben von Konflikten, die beim automatischen Zusammenführen aufgetreten sind 48
6.3. Der Diff Viewer bietet eine grafische Benutzeroberfläche zum Visualisieren der Unterschiede der ausgewählten Architekturentscheidungen 52

Tabellenverzeichnis

2.1. Tabellarische Form eines Y-Statements für das Caching von Zitierstilen in JabRef 19

7.1. Die Tabelle zeigt eine Übersicht der Befehle und gibt deren durchschnittliche Ausführungsdauer, sowie die Interaktionen mit dem Quellcode bzw. SE-Repo an 56

Verzeichnis der Listings

2.1. Klassendefinition des Y-Statements als eADL in Java [Kop16] 20
2.2. Eingebettete Architekturentscheidung in JabRef für die Benutzung eines Caches zum Zwischenspeichern von Zitierstilen 21

6.1. Konfigurationsoptionen und Bedienungsübersicht des Synchronisationsprogramms 46
6.2. Parameterüberblick des Init-Befehls 47
6.3. Parameterüberblick des Pull-Befehls 47
6.4. Parameterüberblick des Commit-Befehls 49
6.5. Parameterüberblick des Sync-Befehls 49
6.6. Parameterüberblick des Merge-Befehls 50
6.7. Parameterüberblick des Reset-Befehls 50
6.8. Parameterüberblick des Status-Befehls 50
6.9. Parameterüberblick des Statistic-Befehls 51
6.10. Parameterüberblick des Diff-Befehls 51
6.11. Parameterüberblick des Config-Befehls 53
6.12. Parameterüberblick des Deinit-Befehls 53

1. Einleitung

Architekturentscheidungen sind in der Softwareentwicklung von Bedeutung, da sie den Aufbau der Software bestimmen [VAC+08]. Oft werden Architekturentscheidungen von Softwareentwicklern getroffen, welche sich in einem späteren Stadium der Entwicklung nicht mehr vollständig zurückverfolgen lassen [LL13a]. Es muss somit sichergestellt werden, dass Architekturentscheidungen nachhaltig verfügbar sind [ZCTZ13]. Hierzu gibt es bereits Werkzeuge, wie AD-Mentor, welches ein Add-In für das Modellierungswerkzeug Sparx Enterprise Architect[1] ist [ZWKG15]. AD-Mentor soll das Modellieren von Architekturentscheidungen vereinfachen und die Entscheidungsfindung unterstützen [ZWKG15]. Die modellierten Architekturentscheidungen können zudem in ein Repository exportiert werden, welches eine Nachverfolgbarkeit der Entscheidungen gewährleistet. Durch sie kann die Wiederverwendbarkeit von bereits getroffenen Architekturentscheidungen erhöht werden, da die Softwarearchitektur leichter zu verstehen ist [GE11].

Eine neue Art der Annotation („embedded ADL“[2], kurz eADL) erlaubt es, Quelltext mit Architekturentscheidungen zu versehen. Durch AD-Mentor und das Verwenden der eADL können Software Architekturentscheidungen an mehreren Stellen hinterlegt sind. Da sich im Laufe der Softwareentwicklung Anforderungen ändern können, kann es vorkommen, dass der Softwarearchitekt oder der Softwareentwickler die getroffenen Architekturentscheidungen ändern muss [LL13b]. Da der Softwarearchitekt und der Softwareingenieur die Architekturentscheidungen an unterschiedlichen Stellen anpassen, können Inkonsistenzen auftreten. Da diese zu Unklarheiten führen können, muss sichergestellt werden, dass die Annotationen im Quellcode mit den Architekturentscheidungen im Repository konsistent sind.

In dieser Arbeit wird diese Herausforderung durch Synchronisation der eingebetteten Architekturentscheidungen mit denen des Repositories adressiert. Dazu wird ein Programm vorgestellt, mit welchem Architekturentscheidungen des Repositories als auch der Quellcode Dateien automatisiert abgeglichen werden können. Zusätzlich sollen Statistiken generiert werden, welche die Unterschiede der doppelt vorhandenen Architekturentscheidungen aufzeigen.

[1] https://www.sparxsystems.de
[2] https://github.com/adr/embedded-adl

Gliederung

Die Arbeit ist in folgender Weise gegliedert:

Kapitel 2 – Architekturentscheidungen: Hier werden die Grundlagen von Architekturentscheidungen beschrieben.

Kapitel 3 – Toolunterstützung zum Verwalten von Architekturentscheidungen: Dieses Kapitel gibt eine Einführung in die Werkzeuge zur Dokumentation von Architekturentscheidungen.

Kapitel 4 – Synchronisation von eADL mit einem SE-Repo: Dieses Kapitel beschreibt die Anforderung der Synchronisierung von den eingebetteten Architekturentscheidungen mit denen, eines Repositories.

Kapitel 5 – Konzept der Synchronisation: In diesem Kapitel wird die Herausforderung der Synchronisation beschrieben.

Kapitel 6 – Implementierung des Synchronisationsprogramms: Hier wird die Architektur und Implementierung der Software vorgestellt.

Kapitel 7 – Fallstudie: Hier wird eine durchgeführte Fallstudie an einem Open-Source Projekt beschrieben.

Kapitel 8 – Zusammenfassung und Ausblick fasst die Ergebnisse der Arbeit zusammen und stellt Anknüpfungspunkte vor.

Als durchgängiges Beispiel wird in dieser Arbeit die Open-Source Software JabRef[3] betrachtet. Diese ist ein Literaturverwaltungsprogramm welche das BibTeX-Format verwendet. Das Programm ist in Java geschrieben und bietet eine grafische Benutzeroberfläche. Es verwendet einige Standardkonzepte, welche gut durch Architekturentscheidungen modelliert werden können, wie das Caching von Annotationen und Zitierstilen.

[3] https://www.jabref.org/

2. Architekturentscheidungen

Die Architektur eines Softwareprogramms setzt sich aus mehreren Komponenten, der Beschreibung der Funktionalität der Komponente und deren Schnittstellen zusammen [CGB+02]. Beim Modellieren der Softwarearchitektur werden diese Komponenten und Schnittstellen durch verschiedene Pfeile zusammengefügt [Cle96]. Am Ende bekommt man ein Modell bei welchem, die Architektur als eine Ansammlung von modellierten Komponenten, Schnittstellen und deren Beziehungen dargestellt wird. Die Entscheidungen und Abwägungen, weshalb eine Komponente oder Schnittstelle definiert wurde, ist hierbei jedoch nicht mehr enthalten. Dabei ist die komplette Architektur eigentlich nur eine Schlussfolgerung von Architekturentscheidungen [Bos04].

In Abschnitt 2.1 wird das Modell des Y-Statements eingeführt und beschrieben wie Architekturentscheidungen in dieser Form dokumentiert werden können. In Abschnitt 2.2 wird eine neue Art der Annotation vorgestellt, welche es erlaubt Softwarearchitekturentscheidungen direkt im Quellcode zu hinterlegen.

2.1. Dokumentation von Architekturentscheidungen

Softwarearchitekten müssen eine Softwarearchitektur erstellen, die auch bei geänderten Anforderungen besteht oder angepasst werden kann. Um dies zu erreichen, müssen die Architekturentscheidungen ebenfalls im Kontext der Erweiterbarkeit getroffen und daher auch dokumentiert werden. Daraus formen sich neue Probleme, wie z.B. der Aufwand der Dokumentation, die Darstellung von Abhängigkeiten der unterschiedlichen Entscheidungen und das Dokumentieren von wiederkehrenden Architekturentscheidungen. [ZCTZ13]

Es existieren viele unterschiedliche Formate, wie IEEE 42010, IBM UMF AD Table, Tyree/ Akerman, Bredemeyer Key Decisions, Nygard ADRs, arc42 Hruschka/Starke und Y-Statements welche Architekturentscheidungen kontextabhängig dokumentieren können [ZWKG15]. Die Formate unterscheiden sich in den Attributen, welche auszufüllen sind und haben somit nicht nur einen unterschiedlichen Informationsgehalt, sondern auch einen unterschiedlichen Dokumentationsaufwand [ZWKG15]. In dieser Arbeit wurde der Argumentation von Zdun et al. [ZCTZ13] gefolgt, weshalb sich diese Arbeit auf das Konzept des Y-Statements fokussiert.

Das Konzept des Y-Statements stellt einen Ansatz der Dokumentation von Architekturentscheidungen und deren Abhängigkeiten dar. Das Ziel ist es, dass der Weg zur Entscheidungsfindung

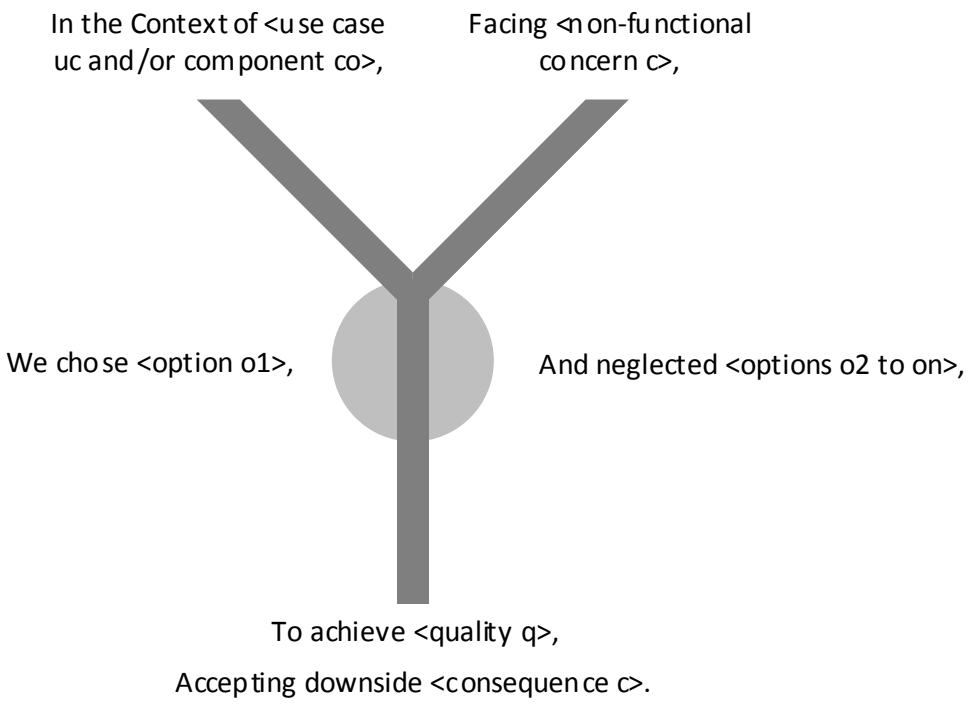


Abbildung 2.1.: Grafische Darstellung des Modells eines Y-Statements

mit dokumentiert wird. Es beinhaltet die Felder *Context*, *Facing*, *Chosen*, *Neglected*, *Achieving* und *Accepting*. Das Feld *Context* beschreibt einen Use-Case als Ausgangslage. Das Feld *Facing* beinhaltet mögliche Herausforderungen und Hindernisse. *Chosen* und *Neglected* dokumentieren die ausgewählte bzw. verworfenen Optionen. Das Feld *Achieving* dokumentiert die resultierenden Vorteile der ausgewählten Option aus und das Feld *Accepting*, die möglichen, akzeptierten Nachteile. Alle Felder zusammen entsprechen nun einer Architekturentscheidung. Abbildung 2.1 zeigt eine visuelle Darstellung des gerade beschriebenen Y-Statements. [ZCTZ13]

Da die abhängigen Architekturentscheidungen nur in Textform dargestellt werden, benötigt ein Softwarearchitekt zusätzlichen Aufwand, um diese zu finden und in Bezug zu stellen. Aus diesem Grund wurden Werkzeuge zur Dokumentationsunterstützung erstellt, welche die Dokumentation von Softwarearchitekturentscheidungen erheblich erleichtern sollen (z.B. AD-Mentor). [ZCTZ13]

2.1.1. Architekturentscheidung aus JabRef in Form eines Y-Statements

Das Literaturverwaltungsprogramm Gliederung wird verwendet um gesammelte Literatur zu verwalten. Die gespeicherten Literatureinträge können ausgewählt und in einer Vorschau angezeigt werden. Der Benutzer kann zudem aus einer großen Auswahl von unterschiedlichen Layouts, die Vorschau der Literaturangaben auswählen. Die Layouts werden dabei aus dem Internet heruntergeladen und können so dem Benutzer in den Einstellungen präsentiert und von diesem ausgewählt werden. Damit die ausgewählten Darstellungen nicht bei jedem Wechsel der Literatureinträge neu geladen werden müssen, haben sich die Entwickler für Caching entschieden. Um dies zu dokumentieren kann das Feld *Context* ausgefüllt werden.

> Context:
> User wants to quickly change the citation style for an entry

Das Feld *Facing* beinhaltet hingegen die auftretenden Herausforderungen und muss nicht unbedingt ausgefüllt werden. Einfachheitshalber wird behauptet, dass die Implementierung des Caches schnell gehen muss, weshalb dies in dem Feld *Facing* dokumentiert wird.

> Facing:
> Not much time

Das bedeutet, dass die Informationen für die ausgewählten Darstellungen der Literaturangaben in den Arbeitsspeicher geschrieben werden und somit schneller abrufbar sind. Das Ersetzen der zwischengespeicherten Dateien kann auf unterschiedliche Weise umgesetzt werden. Nach Wikipedia [Wik17] gibt es mindestens fünfzehn verschiedene Verfahren um den Cache zu erneuern. Diese sind potentielle Optionen um das Caching der Darstellungen der Literaturangaben zu realisieren:

1. Bélády's Algorithm
2. First In First Out (FIFO)
3. Last In First Out (LIFO)
4. Least Recently Used (LRU)
5. Most Recently Used (MRU)
6. Pseudo-LRU (PLRU)
7. Random Replacement (RR)
8. Segmented LRU (SLRU)
9. Least-Frequently Used (LFU)
10. LFU with Dynamic Aging (LFUDA)

11. Low Inter-reference Recency Set (LIRS)
12. Adaptive Replacement Cache (ARC)
13. Clock with Adaptive Replacement (CAR)
14. Multi Queue (MQ) caching algorithm | Multi Queue (MQ)
15. Pannier: Container-based caching algorithm for compound objects

Bei dem Caching Verfahren in JabRef werden jedoch für jeden Literatureintrag die gleichen Darstellungen solange gespeichert, bis der Benutzer die Darstellungsart ändert. Dieses Verfahren ist in der obigen Aufzählung nicht aufgeführt. Im weiteren Verlauf dieser Arbeit wird es als *Clear All On Request* (CAOR) bezeichnet. Das ausgewählte Verfahren wird in dem Feld *Chosen* dokumentiert.

> Chosen:
> Clear All On Request (CAOR)

Die Weiteren verfügbaren Optionen werden in dem Feld *Neglected* aufgezählt.

> Neglected:
> FIFO, LIFO, LRU, MRU, PLRU, RR, SLRU, LFU, LFUDA, LIRS, ARC, CAR, MQ

Der Vorteil des CAOR Verfahren in JabRef ist, dass somit sichergestellt wird, dass die ausgewählten Darstellungen für jeden Literatureintrag solange verfügbar sind, wie dieser existiert. Dies wird mit dem Feld *Achieving* dokumentiert.

> Achieving:
> Selected citation styles should be cached as long as entry exists

Allerdings bedeutet dies auch, dass der Cache bei vielen Einträgen sehr groß werden kann. Dies ist ein Nachteil der getroffenen Entscheidung und kann mit dem Feld *Accepting* dokumentiert werden.

> Accepting:
> Big cache

Tabelle 2.1 zeigt wie die Dokumentation dieser Entscheidung als Y-Statement in Tabellenform aussehen kann.

Y-Statement: Cache replacement	
ID	Y001
Context	User wants to quickly change the citation style for an entry
Facing	Not much time
Chosen	Clear All On Request(CAOR)
Neglected	FIFO, LIFO, LRU, MRU, PLRU, RR, SLRU, LFU, LFUDA, LIRS, ARC, CAR, MQ
Achieving	Selected citation styles should be cached as long as entry exists
Accepting	Big cache

Tabelle 2.1.: Tabellarische Form eines Y-Statements für das Caching von Zitierstilen in JabRef

2.2. Eingebettete Architekturentscheidungen

Eine neue Art der Annotation erlaubt es Softwarearchitekturentscheidungen nun auch direkt in Java Quellcode zu hinterlegen. Somit ist es möglich Architekturentscheidungen mit den resultierenden Komponenten zu verknüpfen. Dadurch erhält der Softwareentwickler ein besseres Verständnis, warum bestimmte Komponenten erstellt wurden und welche Abhängigkeiten diese haben.

Die Annotation ist speziell für Java Klassen und Methoden erstellt worden, lässt sich aber auch auf andere Sprachen übertragen. Das eADL Repository, welches eingebettete Architekturentscheidungen für die Programmiersprache Java definiert, beinhaltet zwei Annotations-Klassen: *DecisionMade* und *YStatementJustification*. *DecisionMade* stellt dabei eine unabhängig getroffene Architekturentscheidung dar, wobei *YStatementJustification* eine Architekturentscheidung nach dem Y-Statement Modell dokumentiert. Listing 2.1 zeigt, wie die *YStatementJustication* Annotations-Klasse definiert wurde.

Zunächst ist die Klasse selbst durch einige Annotationen annotiert worden. *@Retention(RetentionPolicy.RUNTIME)* bedeutet, dass die Annotationen auch zur Laufzeit in den Java Klassendateien vorhanden sein sollen. *@Target(Element.TYPE)* bestimmt, wo die definierte Annotations Klasse auftreten darf. In diesem Fall handelt es sich um eine Klassenannotation. Alle nicht speziell definierten Annotationen für eine Annotationsklasse werden, durch die *@Inherited* Annotation, von den Überklassen übernommen. Damit die Instanziierungen der *YStatementJustification* Klasse auch als JavaDoc für die Softwareentwickler verfügbar sind, wird noch die Annotation *@Documented* benötigt. Eine Annotationsklasse wird in Java durch die *@interface* Notation definiert. Das Feld *id()* ist nicht explizit in dem Modell des Y-Statements beschrieben, wird aber zur Unterscheidung der Architekturentscheidungen benötigt. Das Feld *moreInformation()* wird auch nicht im Modell des Y-Statements beschrieben und stellt eine Erweiterung von diesem dar, wird aber im Folgenden nicht weiter betrachtet. Die übrigen Felder entsprechen genau den Feldern des Y-Statement Modells aus Abbildung 2.1.

Listing 2.1 Klassendefinition des Y-Statements als eADL in Java [Kop16]

```
1  @Retention(RetentionPolicy.RUNTIME)
2  @Target({ElementType.TYPE})
3  @Inherited
4  @Documented
5  public @interface YStatementJustification {
6
7        String id() default ‘\AD-xx";
8
9        String context() default "In the context of [functional requirement and current
             design stage/evolution state]";
10
11       String facing() default "facing [non-functional requirements such as quality
             attributes and constraints]";
12
13       String chosen() default "we decided for [selected solution option]";
14
15       String neglected() default "and neglected [alternate solution options]";
16
17       String achieving() default "to achieve [positive consequences of chosen solution
             (quality attribute fulfillmment?)]";
18
19       String accepting() default "accepting that [negative consequences of chosen
             solution (quality attribute impact?)]";
20
21       String moreInformation() default ""; // this could take URI
22
23 }
```

2.2.1. Instanziierung eines Y-Statements in JabRef durch eADL

Die Klasse *CitationStyleCache* ist in JabRef für das Caching der Zitierstile zuständig. Diese Klasse lässt sich somit auf die bereits beschriebene Architekturentscheidung zurückführen. Der Codeblock in Listing 2.2 zeigt, wie die Architekturentscheidung für das Caching von Zitierstilen durch eine Annotation mit der resultierenden Klasse verknüpft werden kann.

Listing 2.2 Eingebettete Architekturentscheidung in JabRef für die Benutzung eines Caches zum Zwischenspeichern von Zitierstilen

```
@YStatementJustification(
      id = "Caching",
      context = "User wants to quickly change the citation style for an entry",
      facing = "Not much time",
      chosen = "CAOR",
      neglected = "FIFO, LIFO, LRU, MRU, PLRU, RR, SLRU, LFU, LFUDA, LIRS, ARC, CAR, MQ",
      achieving = "Selected citation styles should be cached as long as entry exists",
      accepting = "Big cache")
public class CitationStyleCache {
      ...
}
```

3. Toolunterstützung zum Verwalten von Architekturentscheidungen

Um aus Architekturentscheidungen etwas über die Softwarearchitektur zu lernen, müssen diese hinreichend dokumentiert werden [HP06]. Besonders wichtig ist es, dass sowohl die akzeptierten als auch zurückgewiesenen Entscheidungsoptionen dokumentiert und in Bezug gestellt werden [HP06]. Hierzu wurde in Kapitel 2 bereits die *YStatementJustification* Klasse vorgestellt, welche diese Anforderungen erfüllt.

Allerdings bedeutet dies zusätzlichen Dokumentationsaufwand für den Softwarearchitekten [Zim09]. Um die benötigte Zeit zum Dokumentieren dieser Entscheidungen in Grenzen zu halten, wurden unterschiedliche Programme erstellt, die eine Hilfestellung zur Dokumentation der Entscheidungen geben sollen [JVAH07].

In Abschnitt 3.1 wird speziell das Architekturentescheidungs Guidance-Tool AD-Mentor betrachtet. In Abschnitt 3.2 wird zudem ein Versionierungsrepository vorgestellt.

3.1. AD-Mentor

AD-Mentor ist ein Add-In für das UML-Modellierungswerkzeug Sparx Enterprise Architekt. Somit werden die Architekturentscheidungen nicht in schriftlicher Form, sondern in einem Abhängigkeitsgraphen dokumentiert und visualisiert. Das Tool besteht aus zwei unterschiedlichen Bereichstypen, dem Problem Space und dem Solution Space. Im Problem Space können Problem-Knoten definiert werden, welche für die Architektur der Software relevant sind. Zu jedem Problem können mehrere Options-Knoten erstellt und mittels einer Relation mit dem Problem-Knoten verknüpft werden. Da sich Probleme zum Teil auch untereinander beeinflussen, können auch diese Knoten verknüpft werden. Für Options-Knoten untereinander gilt das Gleiche. [ZWKG15]

Um die Entscheidungsfindung für ein Problem zu erleichtern, wurde unter anderem das Y-Statement [ZCTZ13] als Template in AD-Mentor implementiert [ZWKG15]. Somit beinhaltetet ein Y-Statement Problem-Knoten die Felder *Context* und *Facing* welche das Problem auf funktionale bzw. nicht-funktionale Anforderungen zurückführen [ZWKG15]. In Abbildung 3.1 kann man die modellierte Architekturentscheidung des Caching von Zitierstilen in JabRef

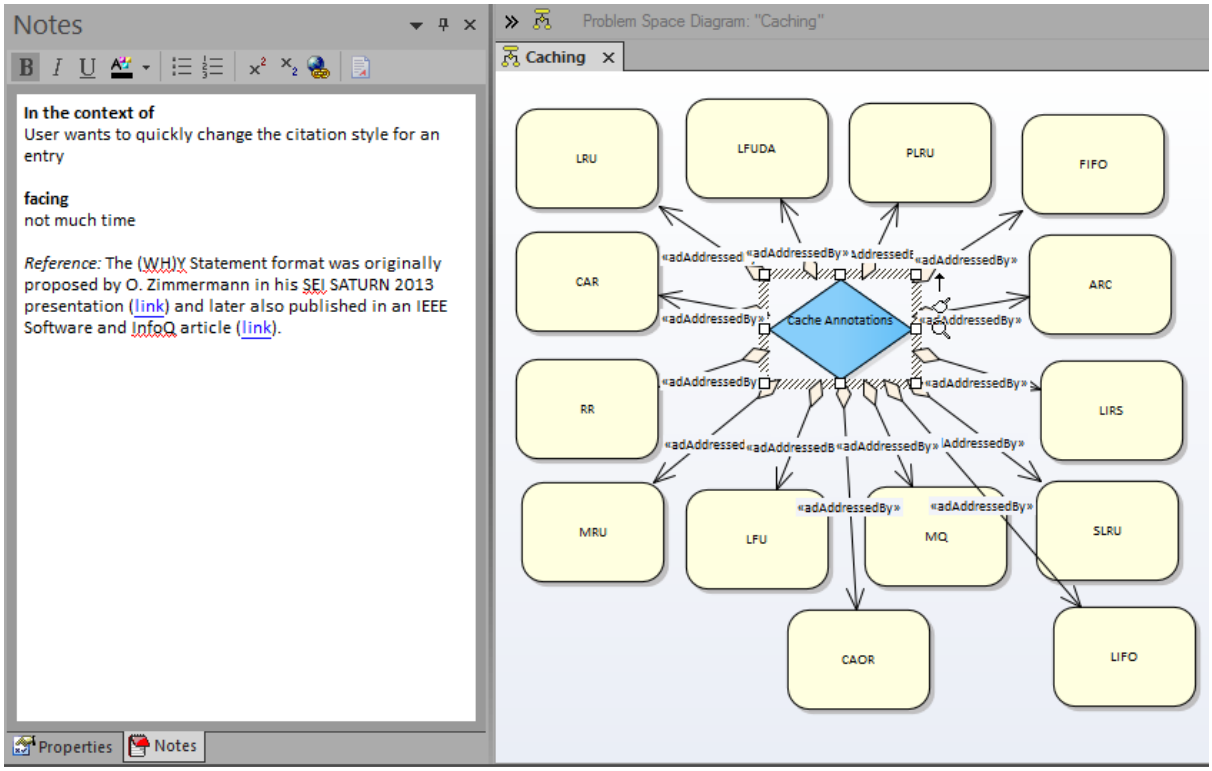


Abbildung 3.1.: Inhalt eines Problem-Knotens bei einem modellierten Y-Statements durch mit AD-Mentor beim Beispiel des Caching der Darstellungen von Literaturentscheidungen in JabRef

sehen. Der Problem-Knoten ist ausgewählt und auf der linken Seite ist sein Inhalt zu sehen. Dort stehen die ausgefüllten Felder *Context* und *Facing*.

Die Felder *Achieving* und *Accepting* sind im Inhalt aller Options-Knoten und beschreiben welche positiven bzw. negativen Auswirkungen die Option hat [ZWKG15]. Jeder Options-Knoten besitzt zudem einen Status, welcher den Entscheidungsprozess der Option beschreibt. In Abbildung 3.2 ist ein Options-Knoten ausgewählt, welcher genau diese Felder beinhaltet. Die Verknüpfungen zwischen Knoten können vom Typ *Addressed By*, *Raises*, *Suggests*, *Conflicts With* oder *Bound To* sein [ZWKG15]. Abbildung 3.3 zeigt konkret, wie die Felder der Problem-, und Options-Knoten auf das Modell des Y-Statements abgebildet werden.

Sobald der Problem Space alle Software Probleme und Optionen des Softwarearchitekten beinhaltet, kann dieser Instantiiert werden. Dies bedeutet, dass aus dem vorhanden Problem Space ein neuer Solution Space generiert wird. Dort kann sich der Softwarearchitekt für jeweils eine Option, pro Problem-Knoten entscheiden. Sobald die Entscheidung für eine Option gefallen ist, werden automatisch alle widersprüchlichen Optionen vernachlässigt. Dies wird solange wiederholt, bis alle Probleme durch eine Option gelöst wurden. Das fertige Diagramm spiegelt nicht nur die getroffenen Softwareentscheidungen wider, sondern auch die Verworfenen. Die ausgewählten Entscheidungen können direkt mit Klassen, eines UML Klassen-Diagrams im

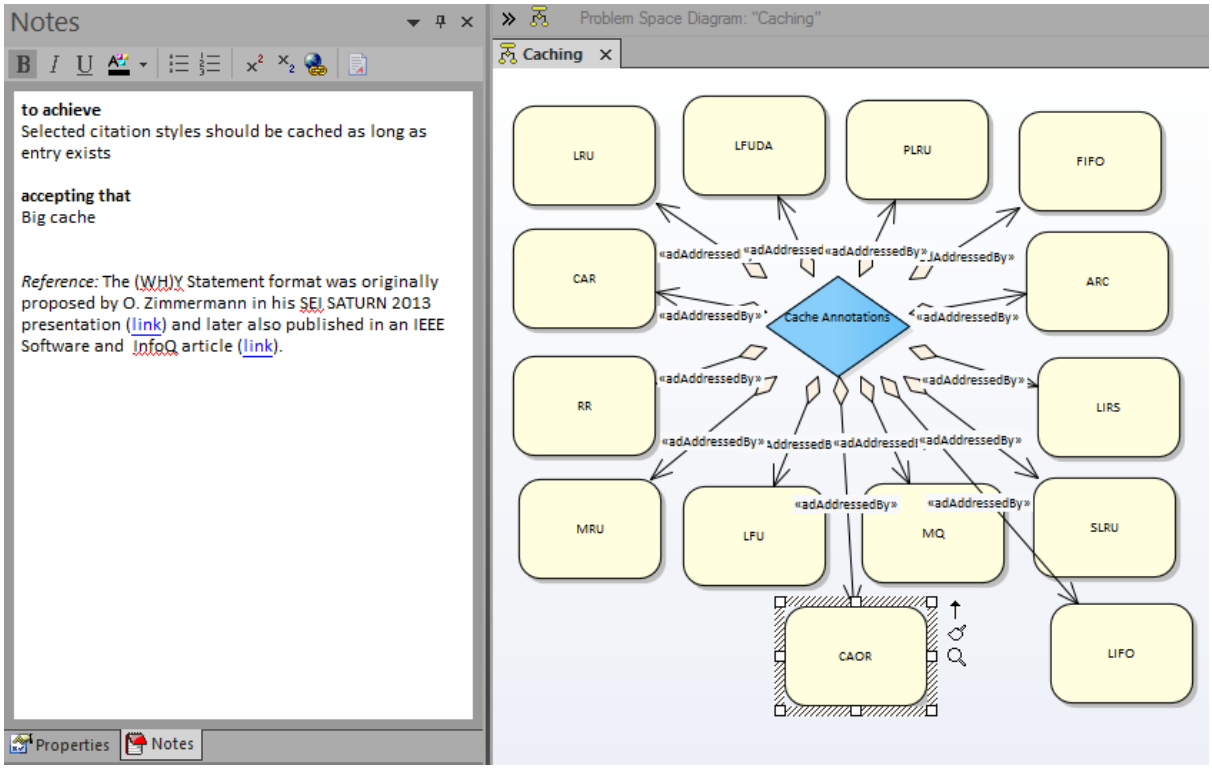


Abbildung 3.2.: Inhalt eines Options-Knotens bei einem modellierten Y-Statements durch mit AD-Mentor beim Beispiel des Caching der Darstellungen von Literaturentscheidungen in JabRef

Sparx Enterprise Architekten verlinkt oder durch weitere Inhalte ergänzt werden. Somit entsteht eine Softwarearchitektur bei der man genau nachvollziehen kann, welche Entscheidung maßgebend für die entsprechende Umsetzung war. [ZWKG15]

Die Abbildungen 3.4 bis 3.5 zeigen eine ausgewählte und eine verworfene Option. Man sieht, dass anhand der Farbe direkt erkennbar ist, welche Option verworfen und welche ausgewählt wurde. Außerdem sieht man in den Eigenschaften den Typ des Knotens und den Status von diesem.

3.2. SE-Repo

Das Software Engineering Repository (SE-Repo) wurde erstellt um Architekturentscheidungen an einem zentralen Ort zu speichern. Die Entscheidungen stehen dann unter Versionskontrolle. Wie bei einem Git Repository werden Änderungen durch Commits[1] hinzugefügt. Damit der

[1]Schnappschuss eines Repositories mit Änderungen [CS14]

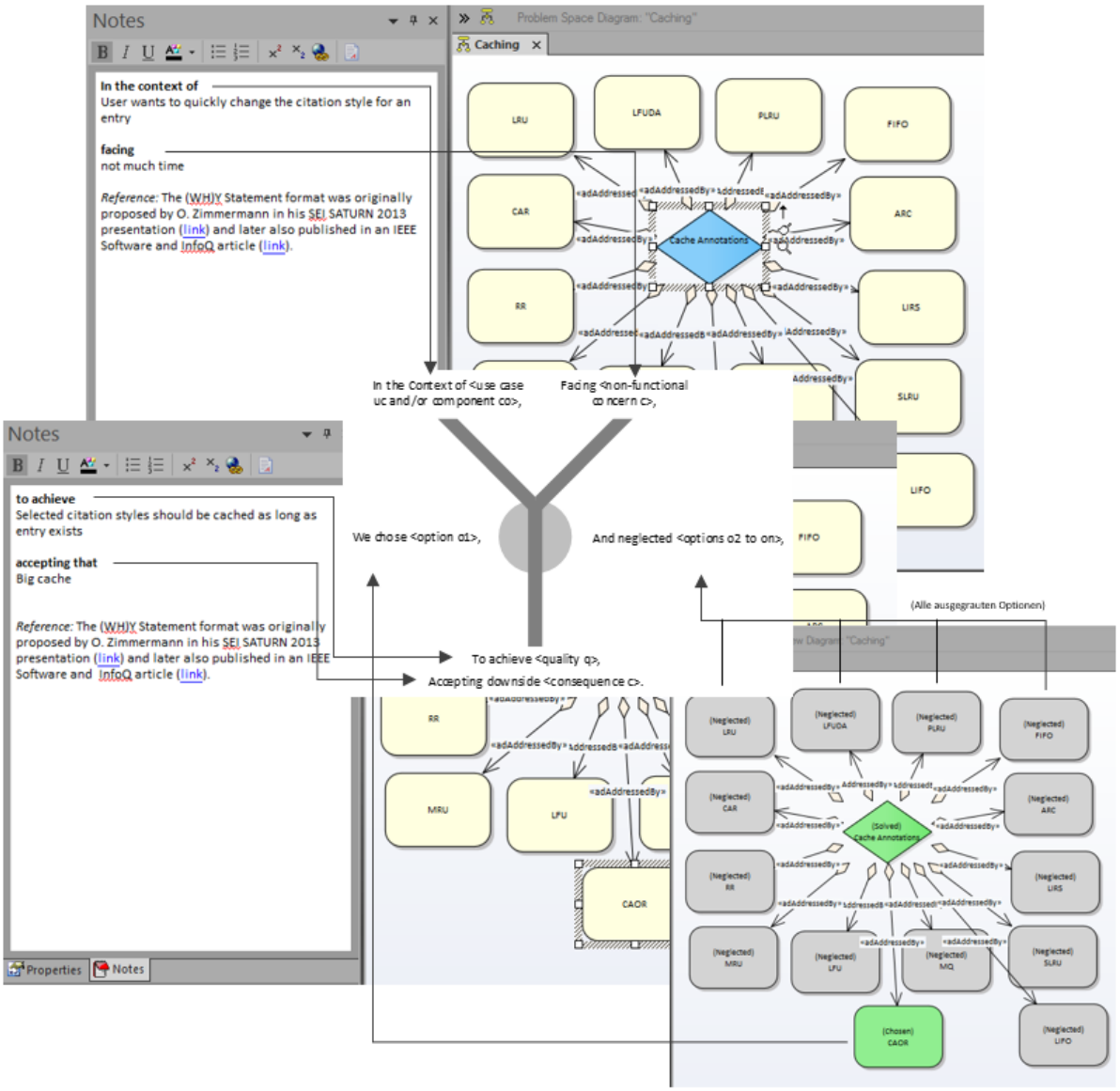


Abbildung 3.3.: Abbildung einer modellierten Architekturentscheidung auf die Felder eines Y-Statements

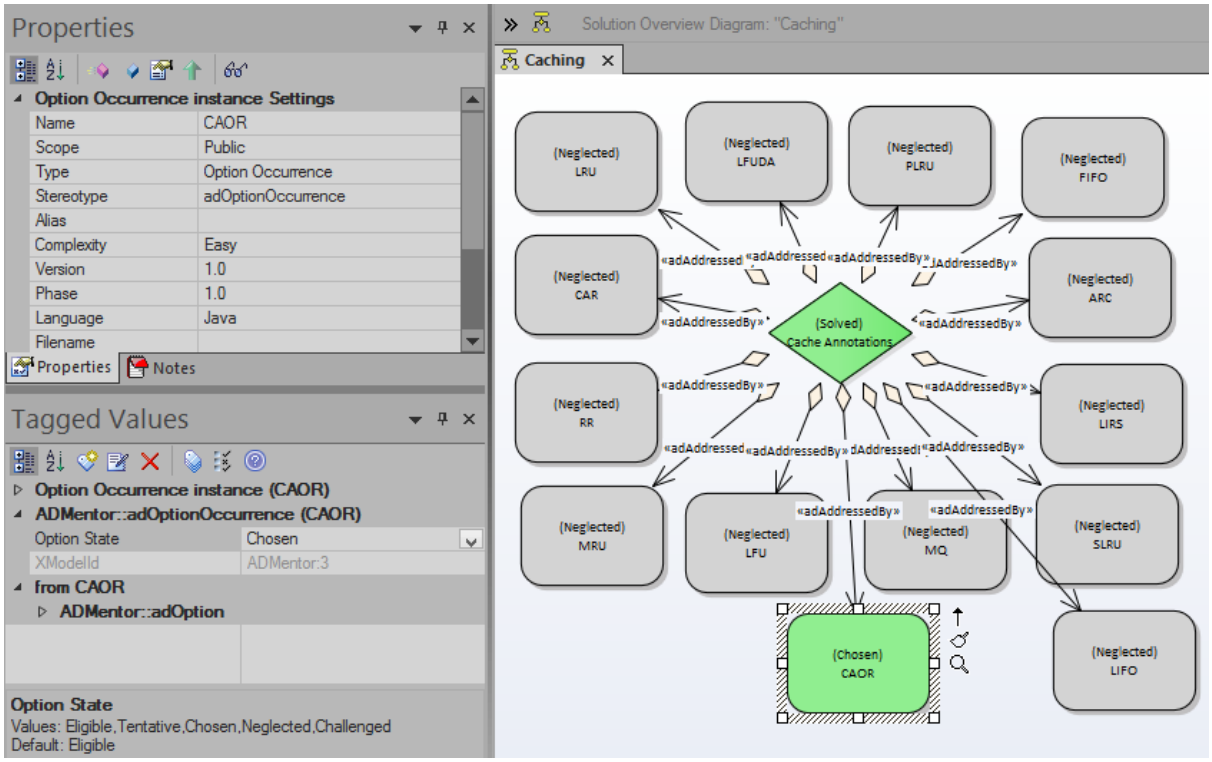


Abbildung 3.4.: Modellierung des ausgewählten Options-Knoten mit AD-Mentor beim Beispiel des Caching der Darstellungen von Literaturentscheidungen in JabRef

Softwarearchitekt Änderungen an den Softwarearchitekturentscheidungen direkt aus dem Architekturentscheidungswerkzeug in das SE-Repo einchecken kann, wurde ein SE-Repo Connector erstellt. Dieser ist ein weiteres Add-In für den Sparx Enterprise Architekten und erlaubt es, das geänderte Modell der Architekturentscheidungen automatisch in ein SE-Repo zu exportieren. Neben dem Inhalt der Architekturentscheidungen werden zudem die Relationen, Metadaten sowie weitere hinzugefügte Werte im Repository als Software Engineering Items (SE-Items) abgespeichert. [Büc17]

Abbildung 3.6 zeigt das Benutzerinterface des SE-Repo Connectors nach der Eingabe der URL des SE-Repos. Der Softwarearchitekt kann ein Repository auswählen und eine Nachricht zum einchecken angeben. Die restlichen Felder werden von dem SE-Repo Connector ausgefüllt.

Zum Zeitpunkt dieser Arbeit wurde der SE-Repo Connector zudem um eine Importfunktion erweitert. Somit könnten Architekturentscheidungen nicht nur im SE-Repo hinterlegt werden, sondern auch zu einem späteren Zeitpunkt wieder abgerufen und mit AD-Mentor modifiziert werden.

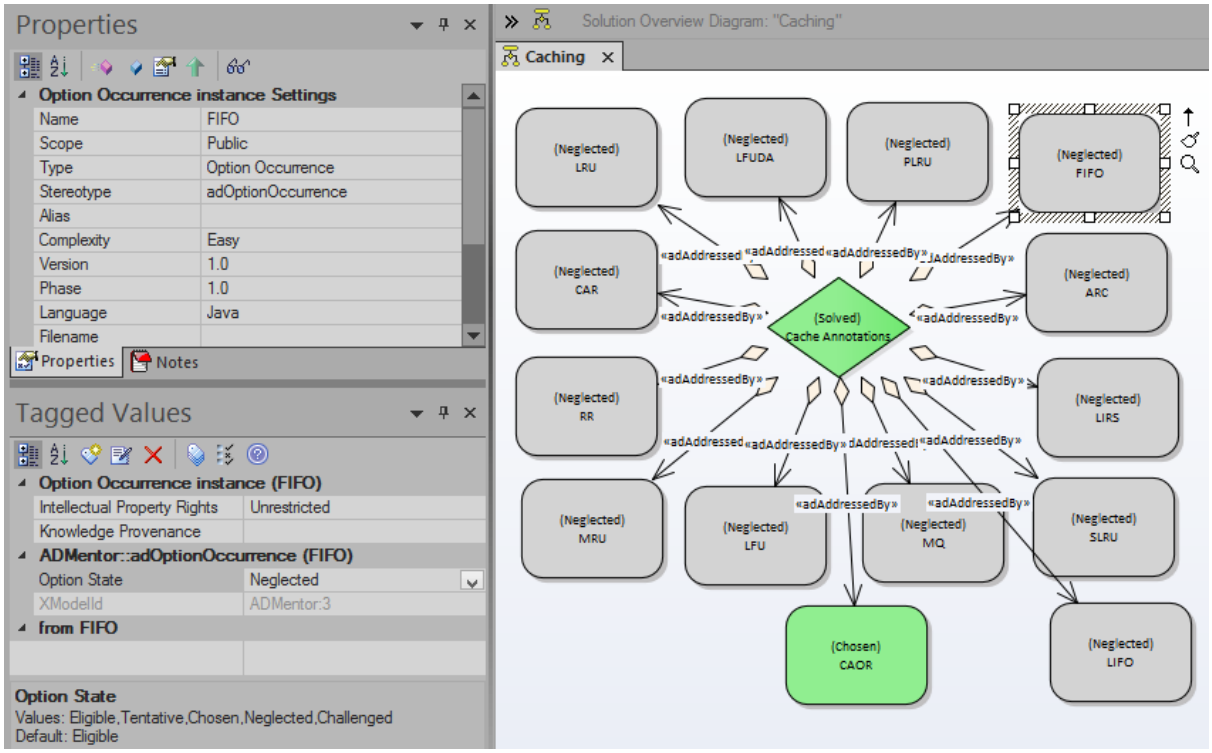


Abbildung 3.5.: Modellierung eines verworfenen Options-Knoten mit AD-Mentor beim Beispiel des Caching der Darstellungen von Literaturentscheidungen in JabRef

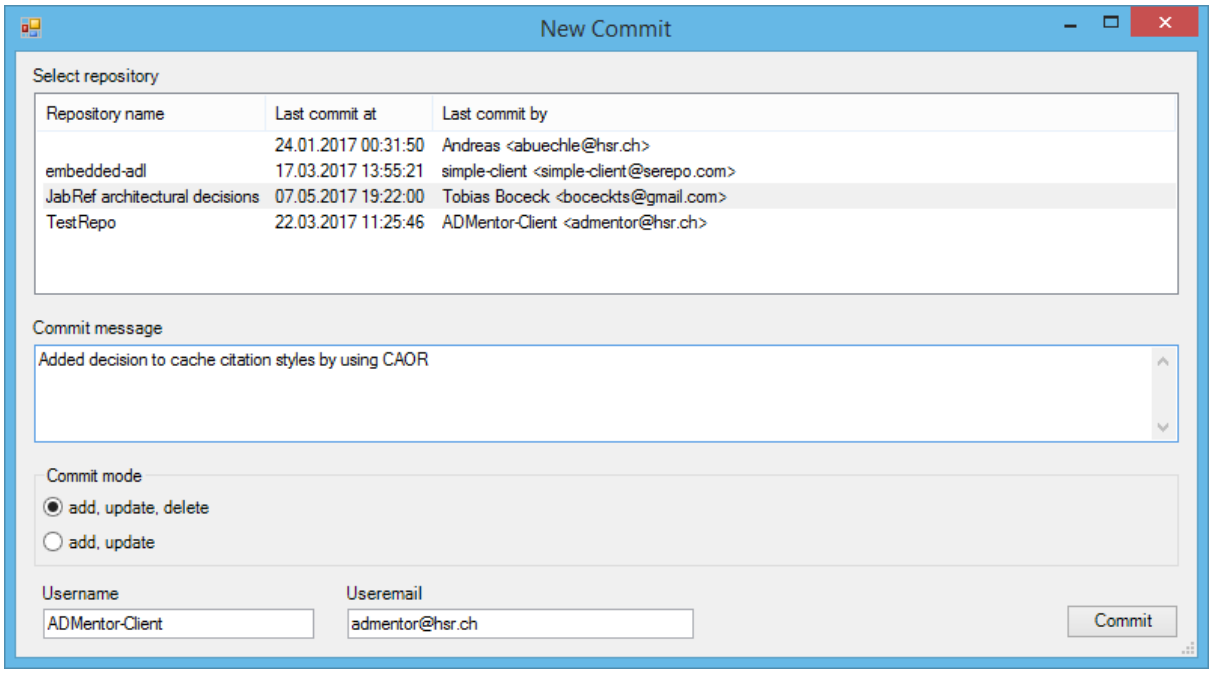


Abbildung 3.6.: Benutzerinterface des SE-Repo Connectors zum einchecken von Änderungen der Architekturentscheidungen

4. Synchronisation von eADL mit einem SE-Repo

In Kapitel 2 wurde die Notation der eADL vorgestellt, welche es erlaubt Y-Statements direkt im Quellcode zu dokumentieren. In Kapitel 3 wurden das Architekturentscheidungswerkzeug AD-Mentor und das SE-Repo vorgestellt. Beide sollen dem Softwarearchitekten helfen, Architekturentscheidungen einfach und nachhaltig zu dokumentieren.

Werden Änderungen an den Architekturentscheidungen in das SE-Repo committet, müssen diese bisher manuell in den Quellcode übertragen werden. Da man interne Dokumentation leichter anpassen kann als externe, kann es ebenso vorkommen, dass die Architekturentscheidungen des Quellcodes nicht mehr mit denen des SE-Repos übereinstimmen (vgl. [Hap14]).

In Abschnitt 4.1 werden zunächst die beteiligten Rollen und Komponenten der Synchronisation vorgestellt. In Abschnitt 4.2 werden die Anforderungen an die Synchronisation definiert und Abschnitt 4.3 beschreibt die hierbei auftretenden Herausforderungen. In Abschnitt 4.4 werden die Einschränkungen der Synchronisation aufgezeigt.

4.1. Übersicht der Synchronisationskomponenten

Zunächst muss festgelegt werden, welche Komponenten und Anwender auf gemeinsame Ressourcen zugreifen, und wie diese in Verbindung zu einander stehen. Es wird davon ausgegangen, dass nur die Rolle des Softwarearchitekten und die des Softwareentwicklers an den Architekturentscheidungen einer Software mitwirken. Die Softwarearchitekten modellieren ihre getroffenen Softwarearchitekturentscheidungen mit AD-Mentor (Kapitel 3), während die Softwareentwickler eingebettete Architekturentscheidungen verwenden, um ihre getroffene Softwarearchitekturentscheidungen im Quellcode zu dokumentieren. Außerdem steht ein SE-Repo zur Verfügung, in welchem modellierte Architekturentscheidungen hinterlegt werden können. In Abbildung 4.1 sind die genannten Rollen und Komponenten zu sehen.

Nachdem die Rollen und Komponenten nun definiert sind, müssen die zugehörigen Schnittstellen festgelegt werden. Um die modellierten Softwarearchitekturentscheidungen in das SE-Repo zu exportieren, wurde bereits ein SE-Repo Connector implementiert, durch den diese als Commit in das SE-Repo committet werden können. Damit die im SE-Repo hinterlegten Architekturentscheidungen auch nachträglich noch geändert werden können, bietet

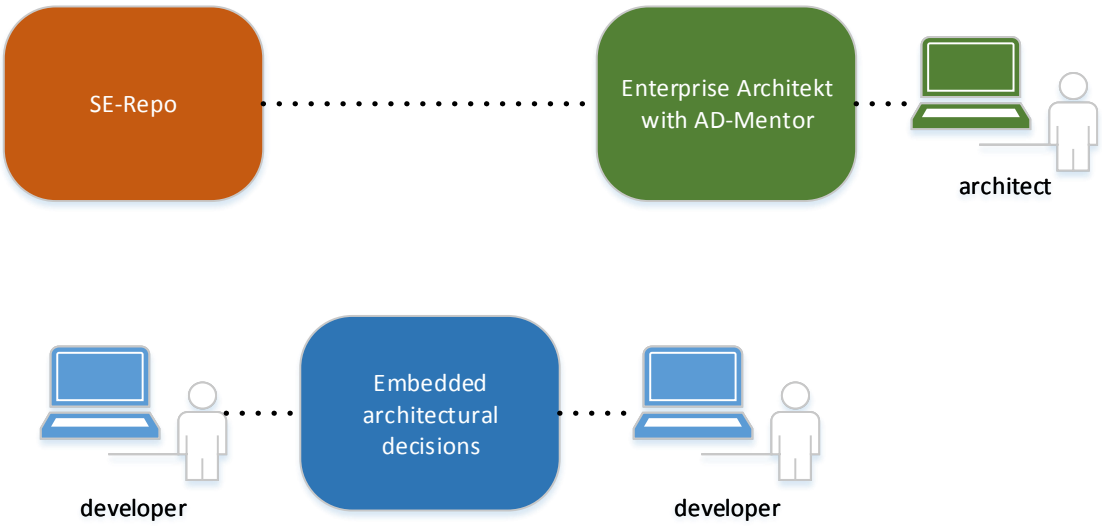


Abbildung 4.1.: Die Grafik zeigt, welche Rollen die Architekturentscheidungen beeinflussen und wie die benutzten Komponenten miteinander in Beziehung stehen

der SE-Repo Connector auch eine Import Funktion. Da die Architekturentscheidungen der Softwarearchitekten und die, der Softwareentwickler an unterschiedlichen Stellen stehen, muss ein Synchronisationswerkzeug erstellt werden, das die Synchronisation mit dem SE-Repo ermöglicht. Das Synchronisationsprogramm muss die eingebetteten Architekturentscheidungen mit denen aus dem SE-Repo aktualisieren und lokale Änderungen in das SE-Repo committen können. In Abbildung 4.2 wird der Synchronisationsprozess zwischen den Komponenten grafisch dargestellt.

4.2. Anforderungen an die Synchronisation

Die Synchronisation muss sicherstellen, dass alle Architekturentscheidungen im SE-Repo mit denen des Code Repositories übereinstimmen. Dazu müssen zunächst mehrere Fälle betrachtet werden.

A1 Architekturentscheidungen im Code entfernen

Im Quellcode existieren Architekturentscheidungen, welche nicht im SE-Repo stehen. Dies bedeutet, dass die Architekturentscheidung aus dem Quellcode gelöscht wurde und nun auch aus dem SE-Repo gelöscht werden muss.

A2 Architekturentscheidungen im SE-Repo löschen

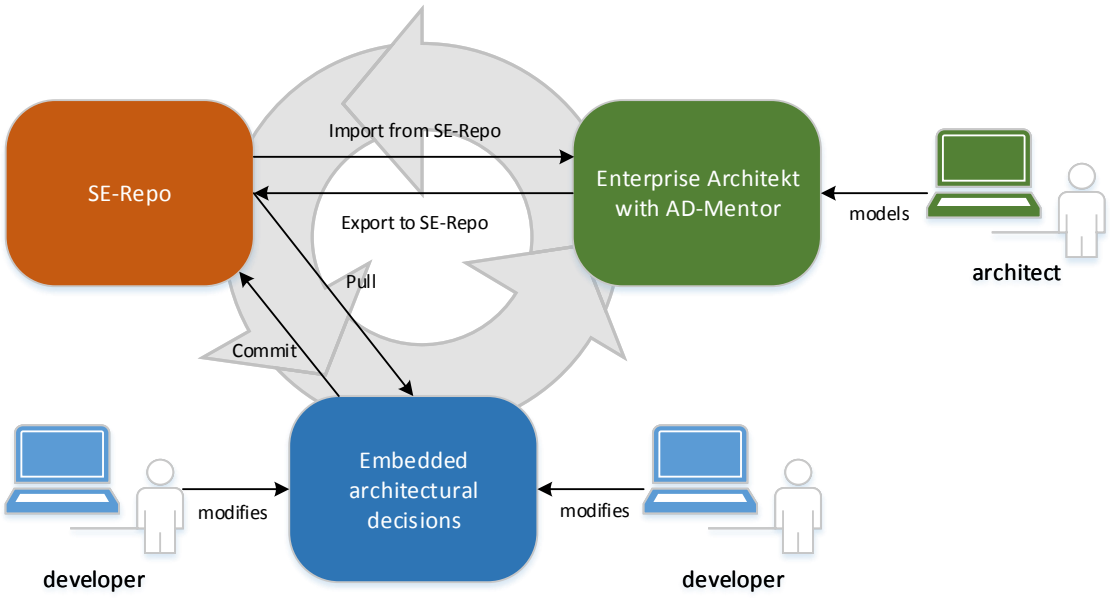


Abbildung 4.2.: Die Grafik zeigt detailliert, welche Rollen die Architekturentscheidungen beeinflussen und wie die benutzten Komponenten kommunizieren

Gleich dem ersten Fall, jedoch existieren nun im SE-Repo Architekturentscheidungen, welche nicht im SE-Repo gespeichert sind.

A3 Unterschiedliche Felder abgleichen

Im SE-Repo stehen Architekturentscheidungen, welche auch im Quellcode hinterlegt sind, jedoch unterscheidet sich der Inhalt einzelner Felder. Daher muss die Synchronisation eine Architekturentscheidung nicht nur als Ganzes sehen, sondern auch den Inhalt dieser abgleichen und wenn nötig aktualisieren.

Zusammenfassend muss die Synchronisation fehlende Architekturentscheidungen im SE-Repo, sowie im Code Repository hinzufügen können. Außerdem müssen vorhandene Architekturentscheidungen verglichen und wenn nötig aktualisiert werden. Da keines der Repositories bevorzugt werden soll, muss der Benutzer des Synchronisationswerkzeugs auswählen können, welche Entscheidungen in welchem Repository aktualisiert werden sollen.

4.3. Herausforderungen der Synchronisation

Im folgenden werden die Herausforderungen der Synchronisation von eADL mit einem SE-Repo aufgelistet.

H1 Unterschiedliche Formate der Architekturentscheidungen

Eine Herausforderung der Synchronisation besteht darin, dass die Daten im SE-Repo nicht in Textform, sondern durch Relationen gespeichert sind. Da die Daten nicht auf einer gemeinsamen Grundlage hinterlegt sind, müssen diese erst für die Synchronisation vorbereitet werden. Nach der Synchronisation müssen die Daten wieder zurück in ihre ursprüngliche Form gebracht werden, sodass diese in ihrer jeweiligen Umgebung gespeichert werden können.

H2 Felder aus HTML Body parsen

Das Template des Y-Statements ist in AD-Mentor so definiert, dass alle benötigten Felder in den Inhalten des Problem-Knotens und dem ausgewählten Options-Knotens stehen. Der Inhalte der Knoten werden im SE-Repo (Kapitel 3) im Body des HTML Dokuments gespeichert. Dort sind die Felder jedoch als einfache Aufzählung ohne eindeutige Abgrenzung definiert. Daher ist eine weitere Herausforderung, den vollständigen Inhalt der im SE-Repo gespeicherten Knoten zu parsen. Bei der Synchronisation muss sichergestellt sein, dass der komplette Inhalt eines Feldes aus dem SE-Repo mit dem Feld einer eADL verglichen wird und sich keine Unterschiede ergeben, wenn keine vorhanden sind.

H3 Java Source Code generieren

Um Architekturentscheidungen aus dem SE-Repo als eADL zu speichern muss Java Quellcode generiert werden. Da eine Programmiersprache nicht einfach eigene Klassen seiner eigenen Sprache schreiben kann, stellt dies eine weitere Herausforderung dar.

H4 IDs der eADL auf IDs der SE-Items abbilden

Jede Architekturentscheidung muss mit einer eindeutigen ID gekennzeichnet sein, damit diese unterschieden werden können. Im SE-Repo hat jedes SE-Item eine eigene ID, welche die URL des SE-Items ist. Bei den eADL hat nur jedes *YStatementJustification* eine ID. Da dieses durch mehrere SE-Items im SE-Repo dargestellt wird, muss eine Abbildung der IDs des SE-Repo auf die einer eADL geschehen.

H5 Konflikte beim Synchronisieren

Die Synchronisierung soll die Felder der Architekturentscheidungen des Code-Repositories mit denen des SE-Repos automatisch zusammenführen. Bei auftretenden Konflikten soll jedoch der Benutzer auswählen können, welche Felder er von wo übernehmen möchte.

4.4. Einschränkungen der Synchronisation

Die durch AD-Mentor modellierten Architekturentscheidungen können zur besseren Übersicht in Ordnern organisiert werden. Diese bilden aber nicht notwendigerweise die Ordnerstruktur

des Code Repositories ab. Somit kann auch kein Zusammenhang zwischen Architekturentscheidung und resultierender eADL hergestellt werden. Dies bedeutet, dass Architekturentscheidungen welche im SE-Repo, aber noch nicht als eADL existieren, nicht automatisch im Code erstellt werden können. Auf der anderen Seite ist es auch nicht möglich die Ordnerstruktur der eADLs automatisch auf die Ordnerstruktur der modellierten Architekturentscheidungen abzubilden. Daher können Architekturentscheidungen nicht hinzugefügt werden und die Synchronisation beschränkt sich nur auf vorhandene Architekturentscheidungen

5. Konzept der Synchronisation

In diesem Kapitel wird das Konzept der Synchronisation vorgestellt. Dazu wird zunächst in Abschnitt 5.1 das Prinzip der zentralen Datenverwaltung bei Versionsverwaltungssystemen beschrieben. In Abschnitt 5.2 werden die elementaren Befehle, welche das Synchronisationswerkzeug unterstützen muss, vorgestellt. Abschnitt 5.3 zeigt, welche Fälle bei der Synchronisation unterschieden werden müssen. Abschnitte 5.4 bis 5.5 stellen das Konzepte für die elementaren Synchronisationsbefehle vor.

5.1. Zentrale Datenverwaltung

Bei der zentralen Datenverwaltung durch ein Versionierungsrepository werden die zu verwaltenden Daten auf einem zentralen Server abgelegt. Dieser ist entweder über ein lokales Netzwerk, oder über das Internet angeschlossen und für die Entwickler erreichbar. Alle Entwickler die an einem versionierten Projekt mitarbeiten, können Daten vom Versionierungsserver abrufen, oder geänderte Daten auf dem Versionierungsserver ablegen. Legt man mehrere Daten zur gleichen Zeit im Versionierungsrepository ab, so nennt man dies einen Commit. Durch das Speichern der Commitreihenfolge können auch Änderungen zu früheren Versionen eingesehen werden. Bei der zentralen Datenverwaltung werden die Commits auf dem zentralen Sever gespeichert und es wird, im Gegensatz zu dezentralisierter Datenverwaltung, keine lokale Kopie der Commits benötigt. [PCF09]

Da das SE-Repo zur zentralen Datenverwaltung entworfen wurde, muss lediglich ein Programm erstellt werden, welches die nötigen Operationen zum Zugreifen bzw. Abrufen der Daten eines Versionierungsreposiotries bereitstellt.

5.2. Synchronisationswerkzeug

Das Synchronisationswerkzeug soll anhand des Standes der letzten Synchronisation eine Diff jeweils für die lokalen und entfernten Architekturentscheidungen erstellen. Dazu muss die Commit ID der letzten Synchronisation im Softwareprojekt hinterlegt werden. Es wird zudem einerseits ein Parser zum Parsen der eADL aus dem Quellcode benötigt, andererseits muss ein Mapper für das Mapping von SE-Items auf das eADL Format existieren. Dabei muss der Inhalt der SE-Items aus nicht strukturierten HTML Code extrahiert werden, wofür ein weiterer

Parser benötigt wird. Die Diff basiert auf dem Datenobjekt der *YStatementJustification* Klasse (**H1**) und soll Änderungen an den einzelnen Feldern beinhalten. Die Diff soll nun auf Feldbasis verglichen und bei nicht auftretenden Konflikten automatisch zusammengeführt werden. Da ein Versionierungsrepository verwendet wird muss die Richtung der Synchronisation angegeben werden. Alternativ soll jedoch auch eine Option zur automatischen beidseitigen Synchronisation existieren.

5.3. Fallunterscheidung bei geänderten Architekturentscheidungen

Typischerweise werden mit AD-Mentor modellierte Architekturentscheidungen in das SE-Repo exportiert. Dazu kann ein erstellter Solution Space durch den SE-Repo Connector exportiert und schlussendlich im SE-Repo committet werden. Der Commit beinhaltet Metadaten, Relationen und Inhalte der modellierten Architekturentscheidungen.

F1 Keine lokalen oder entfernten Änderungen

In diesem Fall sind die lokalen und entfernten Architekturentscheidungen identisch und es ist keine Synchronisation notwendig. Daher dürfen jegliche Ausführungen mit dem Synchronisationsprogramm keine Auswirkungen auf die vorhandenen Architekturentscheidungen haben. Das Programm soll zudem eine entsprechende Meldung ausgeben, welche den Benutzer darauf hinweist, dass keine Synchronisation notwendig ist.

F2 Lokale Änderungen

In diesem Fall haben sich die eingebetteten Architekturentscheidungen geändert aber die entfernten Architekturentscheidungen im SE-Repo sind auf dem Stand der letzten Synchronisation. Abbildung 5.1 visualisiert das Szenario. Das Synchronisationsprogramm muss, durch das Erstellen einer Diff der aktuellen Architekturentscheidungen zu denen vom Stand der letzten Synchronisation, entscheiden, ob dieser Fall eingetreten ist und eine Funktion bereitstellen, welche die lokalen Architekturentscheidungen in das SE-Repo committet. Danach müssen die lokalen und entfernten Architekturentscheidungen identisch sein.

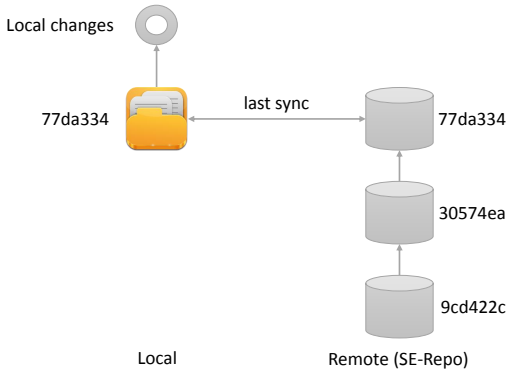


Abbildung 5.1.: Bildliche Darstellung, wie lokale Änderungen an Architekturentscheidungen gegenüber den entfernten Architekturentscheidungen im SE-Repo interpretiert werden sollen

F3 Entfernte Änderungen

In diesem Fall haben sich die entfernten Architekturentscheidungen geändert und es ist ein neuer Commit im SE-Repo hinterlegt. Die lokalen Architekturentscheidungen haben sich jedoch nicht weiterentwickelt und sind auf dem Stand der letzten Synchronisation. Abbildung 5.2 visualisiert das Szenario. Durch das Vergleichen der letzten Synchronisations ID mit der Commit ID des letzten Commits im SE-Repo, kann entschieden werden ob man sich nicht in diesem Fall befindet. Ist die Synchronisations ID gleich der Commit ID des letzten Commits im SE-Repo, so gibt es definitiv keine entfernten Änderungen. Andererseits muss bei unterschiedlicher ID eine Diff der entfernten Architekturentscheidungen zum Stand der letzten Synchronisation erstellt werden. Denn es kann durchaus sein, dass Architekturentscheidungen hinzugefügt, alle bereits vorhandenen jedoch nicht geändert wurden. Da wir als Einschränkung definiert haben, dass nur bereits vorhandene Architekturentscheidungen synchronisiert werden können, werden diese auch nicht in der Diff berücksichtigt und es wird keine Synchronisation benötigt, obwohl sich die Synchronisations ID von der letzten Commit ID im SE-Repo unterscheidet. Das Synchronisationsprogramm kann somit entscheiden ob dieser Fall eingetreten ist und eine Funktion bereitstellen, welche die Änderungen der entfernten Architekturentscheidungen aus dem SE-Repo abruft und auf die lokalen Architekturentscheidungen anwendet. Danach müssen die lokalen und entfernten Architekturentscheidungen identisch sein.

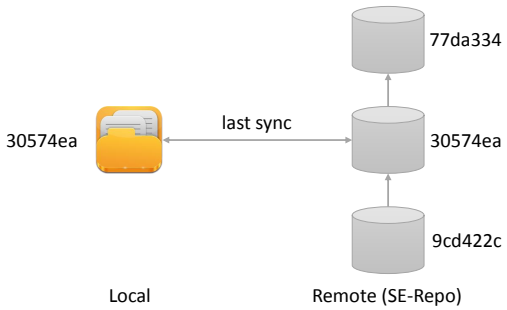


Abbildung 5.2.: Bildliche Darstellung, wie entfernten Änderungen an den Architekturentscheidungen im SE-Repo gegenüber den lokalen Architekturentscheidungen interpretiert werden sollen

F4 Lokale und Entfernte Änderungen

In diesem Fall haben sich die lokalen und entfernten Architekturentscheidungen zum Stand der letzten Synchronisation geändert. Abbildung 5.3 visualisiert das Szenario. Das Synchronisationsprogramm soll eine neue Funktion bereitstellen, welche lokale und entfernte Änderungen automatisch zusammenführen kann. Diese muss außerdem kompatibel zu den Funktionen aus **F2** und **F3** sein. Für die Synchronisation von lokalen und entfernten Änderungen soll somit eine Funktion verfügbar sein, die entfernte Änderungen aus dem SE-Repo abruft und mit den lokalen Architekturentscheidungen zusammenführt. Nach erfolgreicher Ausführung befindet man sich auf dem selben Stand wie in Fall **F3**. Die vorgestellte Funktion für diesen Fall soll darauffolgend automatisch ausgeführt werden.

Hierbei können Konflikte auftreten (**H5**), da es durchaus vorkommen kann, dass sich lokal und entfernt die gleichen Felder bei den gleichen Architekturentscheidung ändern. In diesem besonderen Fall sollen nur die Architekturentscheidungen automatisch zusammengeführt werden, bei denen keine Konflikte auftreten. Zusätzlich soll das Synchronisationsprogramm eine grafische Oberfläche bereitstellen, die es dem Benutzer erlaubt für alle Architekturentscheidungen mit Konflikten, komfortabel und gezielt Felder aus den lokalen und entfernten Architekturentscheidung auszuwählen. Nachdem der Benutzer alle Konflikte manuell gelöst hat, soll die Funktion aus Fall **F2** angewendet werden können. Danach müssen die lokalen und entfernten Architekturentscheidungen identisch sein.

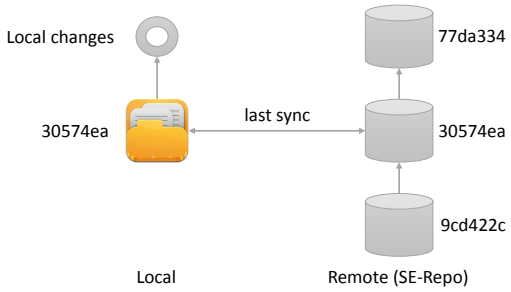


Abbildung 5.3.: Bildliche Darstellung, wie lokale Änderungen an Architekturentscheidungen gegenüber entfernten Änderungen der Architekturentscheidungen im SE-Repo interpretiert werden sollen

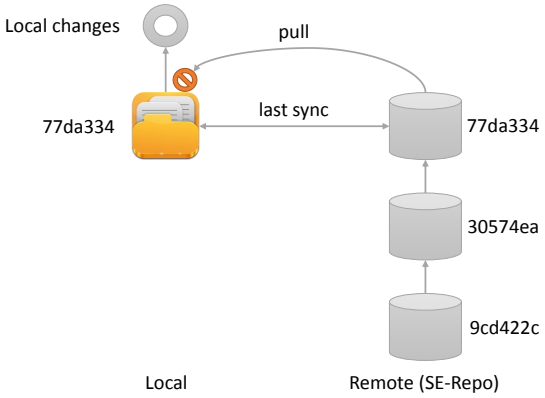
5.4. Änderungen pullen

Um die lokalen Änderungen auf den aktuellen Stand zu bringen, soll das Synchronisationswerkzeug einen Befehl zum Abrufen (pull) von Änderungen aus dem SE-Repo bereitstellen. Dabei soll zunächst eine Diff der lokalen und entfernten Architekturentscheidungen zu denen vom Stand der letzten Synchronisation erstellt werden. Tritt der Fall **F1** ein und es existieren weder lokale noch entfernte Unterschiede, so soll der Pull Befehl beendet werden, da die Architekturentscheidungen bereits synchronisiert sind.

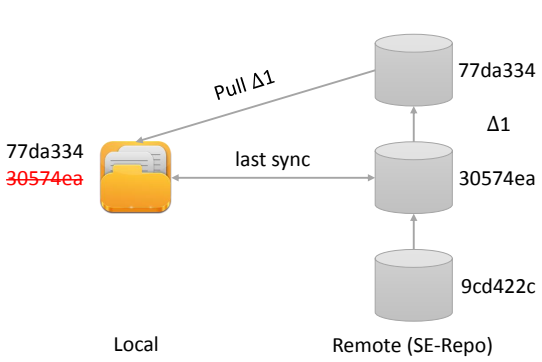
Falls die Diff der entfernten Architekturentscheidungen leer ist, oder die Synchronisations ID mit der Commit ID des letzten Commits im SE-Repo übereinstimmt und wie in Fall **F2** nur lokale Änderungen existieren, soll ein Pull verweigert und die lokalen Architekturentscheidungen nicht verändert werden. In Abbildung 5.4a wird dieser Fall bildlich dargestellt.

Falls allerdings die Diff der lokalen Architekturentscheidungen leer ist und stattdessen wie im Fall **F3** eine Diff der entfernten Architekturentscheidungen existiert, soll ein Pull die Diff dieser, auf die lokalen Architekturentscheidungen anwenden. Außerdem muss die ID der letzten Synchronisation auf die Commit ID des letzten Commits im SE-Repo geändert werden.

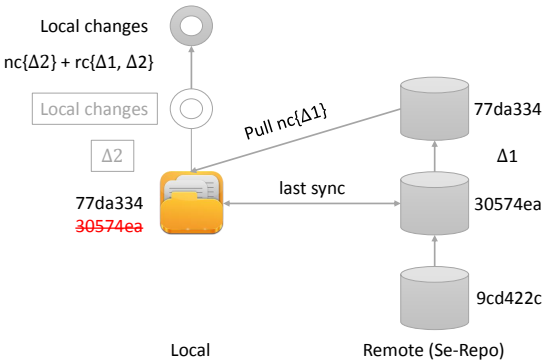
Existiert eine lokale und eine entfernte Diff, tritt Fall **F4** ein. Hierbei soll zunächst die Diff der entfernten Architekturentscheidungen, welche keine Konflikte zur lokalen Diff aufweist, auf die Architekturentscheidungen vom Stand der letzten Synchronisation angewandt werden. Bei Konflikten, soll der Benutzer nun aufgefordert werden diese in einem bereitgestellten Programmfenster zu lösen. Die gelösten Konflikte werden nun, auf die Architekturentscheidungen vom Stand der letzten Synchronisation mit angewandter entfernter Diff ohne Konflikte, angewandt. Zum Schluss muss noch die Diff der lokalen Architekturentscheidungen, welche keine Konflikte mit der entfernten Diff aufweist, auf den aktuellen Stand angewandt werden. Der Pull Befehl erfüllt daher die Anforderung **A1** und **A3** aus Kapitel 4.



(a) Konzept zum Pullen von Architekturentscheidungen, wenn nur lokale Änderungen vorhanden sind (**F2**)



(b) Konzept zum Pullen von Architekturentscheidungen, wenn nur entfernte Änderungen vorhanden sind (**F3**)



(c) Konzept zum Pullen von Architekturentscheidungen, wenn lokale und auch entfernte Änderungen vorhanden sind (**F4**)

Abbildung 5.4.: Konzept zum Pullen von Architekturentscheidungen für die unterschiedlichen Fälle **F2** bis **F4**

Nach dem Ausführen des Pull Befehls müssen die lokalen Architekturentscheidungen entweder alle Änderungen der entfernten Architekturentscheidung enthalten oder der Benutzer hat sich explizit gegen einzelne Entscheidungen entschieden. Nach der ersten erfolgreichen Ausführung des Befehls, müssen sich alle weiteren Ausführungen, in Fall **F1** oder **F2** befinden, womit diese Ausführungen keine Auswirkungen auf die lokalen und entfernten Architekturentscheidungen haben.

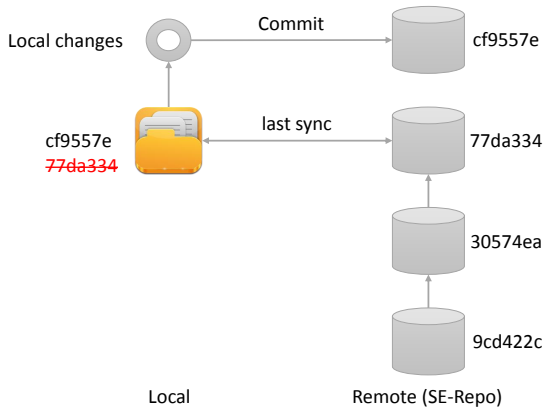
5.5. Änderungen committen

Um die entfernten Änderungen auf den aktuellen Stand zu bringen, soll das Synchronisationswerkzeug einen Befehl zum Einstellen (commit) von lokalen Änderungen bereitstellen. Dabei soll zunächst eine Diff der lokalen und entfernten Architekturentscheidungen zu denen vom Stand der letzten Synchronisation erstellt werden. Tritt der Fall **F1** ein und es existiert weder eine lokale noch eine entfernte Diff, so soll der Commit Befehl beendet werden, da die Architekturentscheidungen bereits synchronisiert sind.

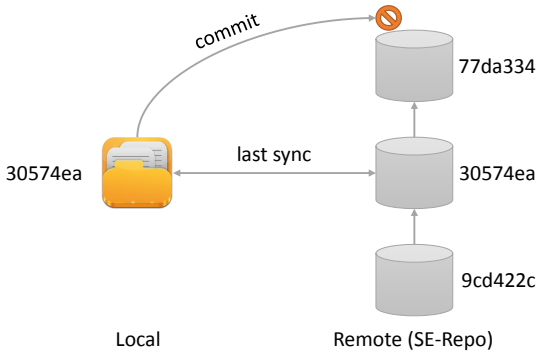
Falls die Diff der entfernten Architekturentscheidungen leer ist, oder die Synchronisations ID mit der Commit ID des letzten Commits im SE-Repo übereinstimmt und wie in Fall **F2** nur lokale Änderungen existieren, soll die Commit Operation die lokalen Architekturentscheidungen im SE-Repo comitten. Nun muss zudem die Synchronisations ID durch die Commit ID des Commits ersetzt werden. In Abbildung 5.4a wird dieser Fall bildlich dargestellt.

Sobald jedoch eine entfernte Diff existiert, soll, unabhängig von lokalen Änderugen, ein Commit verweigert werden. Denn sonst würden die Änderungen aus den Commits, welche nach dem Commit mit der Synchronisations ID gemacht wurden, verloren gehen. Dies betrifft die Fälle **F3** und **F4** und ist andernfalls in den Abbildung 5.5b und 5.5c bildlich dargestellt. Durch den Commit Befehl werden die Anforderung **A2** und **A3** aus Kapitel 4 erfüllt.

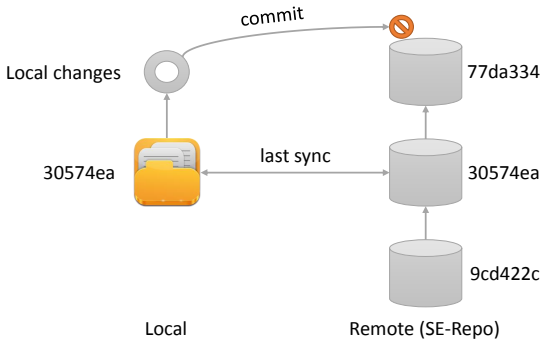
Nach dem Ausführen des Commit Befehls müssen die entfernten Architekturentscheidungen alle Änderungen der lokalen Architekturentscheidung enthalten. Nach der ersten erfolgreichen Ausführung des Befehls müssen sich alle weiteren Ausführungen in Fall **F1** befinden, da die lokalen und entfernten Architekturentscheidungen nun synchronisiert sind. Diese Ausführungen haben somit keine Auswirkungen auf die lokalen und entfernten Architekturentscheidungen.



(a) Konzept zum Committen von Architekturentscheidungen, wenn nur lokale Änderungen vorhanden sind (**F2**)



(b) Konzept zum Committen von Architekturentscheidungen, wenn nur entfernte Änderungen vorhanden sind (**F3**)



(c) Konzept zum Committen von Architekturentscheidungen, wenn lokale und auch entfernte Änderungen vorhanden sind (**F4**)

Abbildung 5.5.: Konzept zum Committen von Architekturentscheidungen für die unterschiedlichen Fälle **F2** bis **F4**

6. Implementierung des Synchronisationsprogramms

In diesem Kapitel wird das implementierte Synchronisationsprogramm[1] beschrieben. Zunächst wird in Abschnitt 6.1 die Architektur und Implementierung des Synchronisationsprogramms vorgestellt. Dabei werden insbesondere die wichtigsten Klassen und Pakete vorgestellt. Der Abschnitt 6.2 beschreibt welche Bedingungen erfüllt sein sollten, um eine fehlerfreie Ausführung des Programms zu gewährleisten. Die Abschnitte 6.3 und 6.4 beschreiben wie das Programm bedient wird und welche Befehle dem Anwender zur Verfügung stehen.

6.1. Architektur und Implementierung

Das Synchronisationsprogramm wurde als Kommandozeilenprogramm entworfen. Somit wurde ein Kommando Paket erstellt, welches die Schnittstelle des Anwenders zum Programm darstellt. Die Befehle zur Synchronisation wurden dagegen in einem Logik Paket implementiert, welches ein Interface mit den benötigten Methoden zur Synchronisation für das Kommando Paket bereitstellt. Die Logik Implementierung benutzt Klassen aus dem Model Paket um die Synchronisationsbefehle auszuführen und gibt einen Status zurück. Dieser wird dann von dem Kommando Paket interpretiert und dem Anwender wird eine informative Rückmeldung angezeigt. Abbildung 6.1 visualisiert die Paketstruktur.

[1] https://github.com/boceckts/eadlsync

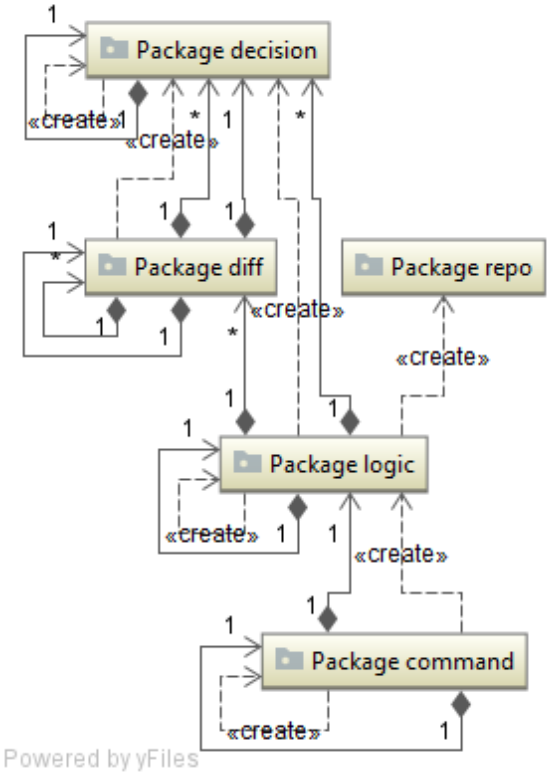


Abbildung 6.1.: Das UML Paketdiagramm zeigt die Trennung zwischen dem Kommando Paket, welches die Benutzerschnittstelle darstellt, und dem Logik Paket, welches mit den Model Paketen arbeitet

Eine Klasse *YStatementJustificationWrapper* wurde erstellt, um die Felder des Y-Statements abzubilden und Programm intern eine gemeinsame Klasse, zum Vergleichen von Architekturentscheidungen zu haben (**H1**). Zum Parsen der eingebetteten Architekturentscheidungen aus dem Quellcode (**H3**) wurde die Hilfsklasse *JavaDecisionParser* erstellt, welche die Open-Source Bibliothek Roaster[2] zum Parsen des Quellcodes benutzt. Zum Abrufen der modellierten Architekturentscheidungen aus dem SE-Repo wurden weitere Hilfsklassen erstellt. Die Klasse *SeRepoConnector* bildet die Schnittstelle des SE-Repos ab und bietet Methoden zum abrufen bzw. committen von SE-Items. Die separate Klasse *SeItemContentParser* wird dabei zum Parsen der Inhalte der SE-Items benutzt (**H2**). Da die interne Datenstruktur auf der Klasse *YStatementJustificationWrapper* basiert, existiert die Klasse *YStatementSeItemHelper*, welche die SE-Items in die interne Objektstruktur umwandelt. Dabei wird auch die ID der SE-Items auf die ID der eingebetteten Architekturentscheidungen abgebildet (**H4**). Die zentrale Klasse des Synchronisationsprogramms heißt *CodeRepo* und ist durch das Interface *IRepo* definiert. Dieses bietet alle notwendigen Methoden zur Synchronisation an. In der zentralen Klasse werden die lokale

[2] https://github.com/forge/roaster

und entfernte Diff in einem Objekt der Klasse *DiffManager* gespeichert. Durch dieses kann unterschieden werden, in welchem Fall aus Kapitel 5 sich das Synchronisationsprogramm gerade befindet. Zum Lösen von auftretenden Konflikten wurde ein ConflictManager in JavaFX implementiert (**H5**).

6.2. Voraussetzungen

Das Programm ist in Java geschrieben und verwendet unter anderem die Oberflächensprache JavaFX, weshalb Java in der Version 8u121 oder höher benötigt wird.

Das Programm ist als Kommandozeilenprogramm entworfen worden und verfügt über entsprechende Wrapper für Windows und Unix Plattformen. Diese sollten als Umgebungsvariable im Betriebssystem hinzugefügt werden, damit das Synchronisationsprogramm von jedem Verzeichnis aus, auf einer Kommandozeile über den Name des Synchronisationsprogramms ausgeführt werden kann.

6.3. Bedienung

Das Synchronisationsprogramm wird von der Kommandozeile aus mit den gewünschten Parametern und einem Befehl aufgerufen. Listing 6.1 zeigt die verfügbaren Parameter. Die wichtigsten Parameter sind jene, die zum Debuggen und zum Anzeigen von erweiterten Informationen im Fehlerfall existieren. Ansonsten können mit den restlichen Parametern lediglich Informationen über das Synchronisationsprogramm selbst angezeigt werden. Eine ausführliche Liste, der verfügbaren Befehle mit deren Beschreibung findet sich im Abschnitt 6.4.

6.4. Funktionsumfang

Im Folgenden werden die unterstützten Befehle des Synchronisationsprogramms vorgestellt. Dabei sind die Befehle *Init*, *Deinit*, und *Config* für die Konfiguration des Synchronisationsprogramms notwendig. Die Befehle *Commit*, *Pull*, *Sync*, *Merge* und *Reset* wurden erstellt um die eingebetteten Architekturentscheidungen mit denen des SE-Repos zu synchronisieren und werden daher auch Synchronisationsbefehle genannt. Da das Synchronisationsprogramm keine grafische Oberfläche zur Navigation bietet, wurden die weiteren Synchronisationsbefehle *Status* und *Diff* erstellt, um den Anwender über den aktuellen Stand der Synchronisation zu informieren. Um die Benutzung von eingebetteten Architekturentscheidungen in Bezug zu den verfügbaren Klassen eines Projektes zu stellen wurde noch der Statistic-Befehl erstellt.

Listing 6.1 Konfigurationsoptionen und Bedienungsübersicht des Synchronisationsprogramms

```
$ eadlsync -h
Usage: <main class> [options] [command] [command options]
      Options:
        -c, --copyright, --license
            Show copyright information of this program
        -d, --debug
            Debug mode
            Default: false
        -a, --author, --authors, -dev, --developers
            List the authors of this program
        -h, --help
            Show the usage of this program
        -l, --log
            Set the log level
            Default: OFF
        -s, --stacktrace
            Print stacktrace
            Default: false
        -v, --version
            Show version information of this program
```

6.4.1. Init

Der Init-Befehl wird benötigt um das Synchronisationsprogramm in einem Code Repository zu initialisieren. Wie in Listing 6.2 zu sehen ist, müssen die URL zum SE-Repo und der Name des zu synchronisierenden Repositories als Parameter angegeben werden. Optional kann zudem der Pfad zum Source Code angegeben werden. Dadurch liest das Programm nur Dateien unter diesem Verzeichnis ein und nicht relevante Dateien, wie z.B. Programmcode Dateien für das Testen werden nicht berücksichtigt. Nach dem Ausführen des Befehls ist ein neuer Ordner im aktuellen Verzeichnis angelegt worden, welcher die Konfigurationsdatei für die Synchronisation beinhaltet. Außerdem wird im aktuellen Verzeichnis die Datei *.eadlsync-commitid* angelegt, welche die Commit ID der letzten Synchronisation beinhaltet.

6.4.2. Pull

Der Pull-Befehl holt die neuesten Änderungen an den Architekturentscheidungen und wendet diese auf die lokalen an. Wie im Konzept aus Abschnitt 5.4 bereits vorgestellt, wird immer der neuste Commit aus dem SE-Repo verwendet. Daher benötigt dieser Befehl auch keine weiteren Parameter, wie in Listing 6.3 zu sehen ist. Bei Konflikten mit den lokalen Änderungen wird ein Fenster mit den Konflikten geöffnet. Die grafische Benutzeroberfläche visualisiert die Konflikte und der Anwender kann diese mit wenigen Klicks beseitigen. Abbildung 6.2 zeigt diese grafische Oberfläche anhand von Architekturentscheidungen in JabRef. Um die lokalen Architekturentscheidungen mit einem beliebigen Commit zusammen zu führen, kann

Listing 6.2 Parameterüberblick des Init-Befehls

```
$ eadlsync init -h
use 'eadlsync init -u <base-url> -p <project>' to initialize eadlsync in this directory
Usage: init [options]
      Options:
        -h, --help
             Show the usage of this command
      * -p, --project
             name of the project in the se-repo
        -s, --source
             specify the source directory of the project as relative or absolute path
             Default: <directory where program is started>
      * -u, --url
             url of the se-repo, has to end with '/se-repo' except if localhost is
                 specified
```

Listing 6.3 Parameterüberblick des Pull-Befehls

```
$ eadlsync pull -h
use 'eadlsync pull' to update the local decisions
Usage: pull [options]
      Options:
        -h, --help
             Show the usage of this command
```

der Merge-Befehl aus dem Abschnitt 6.4.5 verwendet werden. Um die lokalen Architekturentscheidungen, ungeachtet der lokalen Änderungen auf den Stand eines bestimmten Commits zurückzusetzen, kann der Reset-Befehl aus dem Abschnitt 6.4.6 verwendet werden.

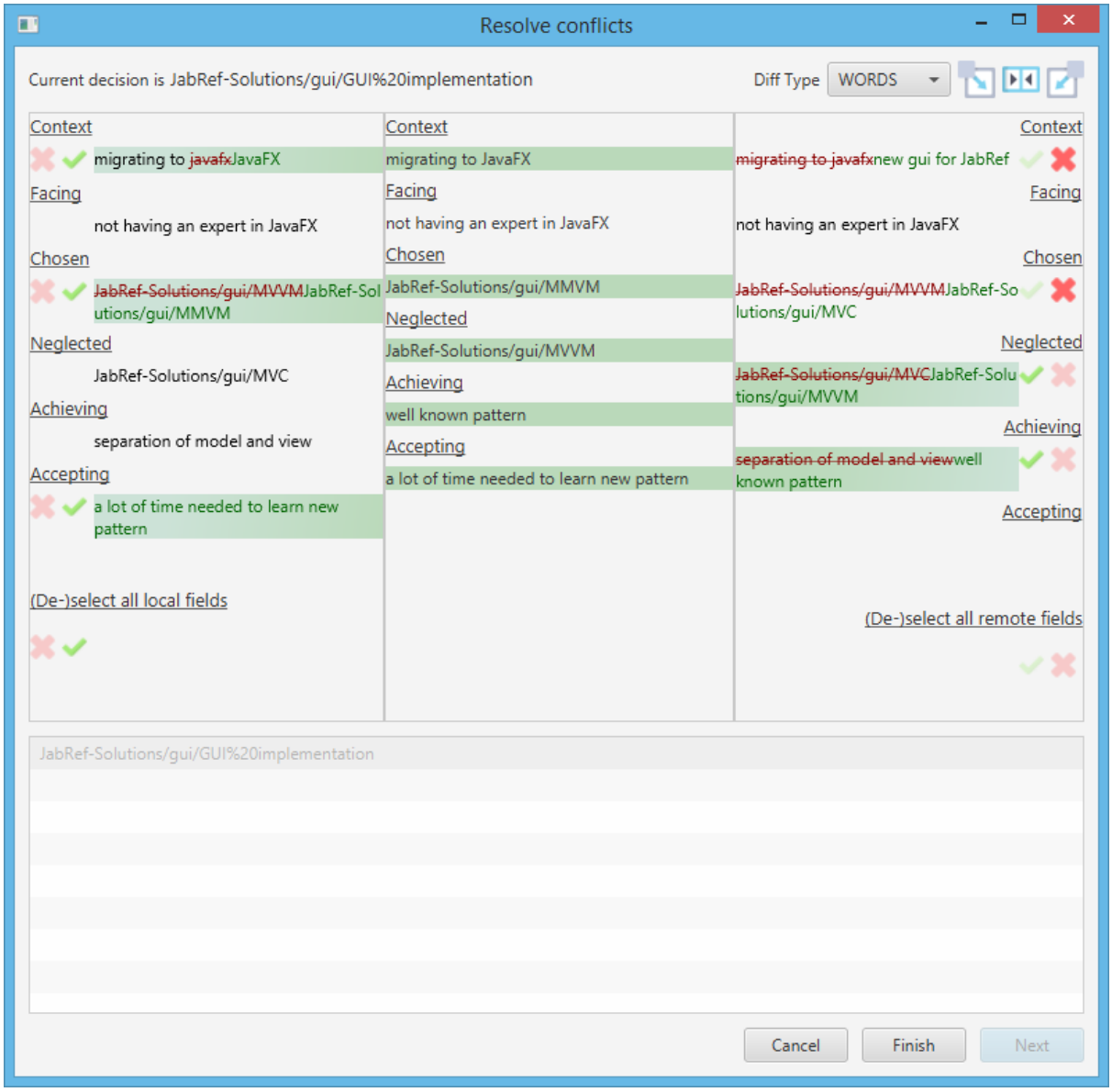


Abbildung 6.2.: Der Konflikt Manager bietet eine grafische Benutzeroberfläche zum komfortablen Beheben von Konflikten, die beim automatischen Zusammenführen aufgetreten sind

6.4.3. Commit

Der Commit-Befehl committet die lokalen Architekturentscheidungen in das SE-Repo und bildet das Konzept aus Abschnitt 5.5 ab. Beim Ausführen des Befehls muss eine Commit Nachricht angegeben werden, welche im SE-Repo hinterlegt wird. Listing 6.4 zeigt, dass ein optionaler Parameter angegeben werden kann, wodurch man erzwingen kann, dass beim Committen der lokalen Architekturentscheidungen Änderungen im SE-Repo ignoriert werden.

Listing 6.4 Parameterüberblick des Commit-Befehls

```
$ eadlsync commit -h
Usage: commit [options]
      Options:
        -f, --force
             force a commit even if the se-repo has changes
             Default: false
        -h, --help
             Show the usage of this command
      * -m, --message
             the commit message that appears in the se-repo
```

Listing 6.5 Parameterüberblick des Sync-Befehls

```
$ eadlsync sync -h
use 'eadlsync sync' to update the local decisions and afterwards update the decisions of
    the se-repo
Usage: sync [options]
      Options:
        -h, --help
             Show the usage of this command
```

6.4.4. Sync

Der Sync-Befehl wurde erstellt um den normalen Synchronisationsweg durch einen kurzen Befehl abzubilden. Beim normalen Synchronisationsweg werden die Änderungen vom SE-Repo gepullt, Änderungen (automatisch) zusammengeführt und anschließend die vorhandenen lokalen Änderungen, mit einer Commit Nachricht, in das SE-Repo committet. Der Sync-Befehl führt die Befehle *Pull* und *Commit* hintereinander aus und ist somit ein Shortcut für diese. Der Commit wird mit einer automatisch generierten Commit Nachricht in das SE-Repo committet. Auf diese Weise benötigt der Sync-Befehl keine weiteren Parameter, wie auch in der Beschreibung in Listing 6.5 zu sehen ist.

6.4.5. Merge

Der Merge-Befehl bietet eine Ergänzung zum bereits vorgestellten Pull-Befehl. Durch den Merge-Befehl ist es möglich die lokalen Architekturentscheidungen nicht nur mit dem aktuellen Stand des SE-Repos zusammenzuführen, sondern mit einem beliebigen Commit. Dazu muss zusätzlich zum Befehl die gewünschte Commit ID angegeben werden. Die lokalen Architekturentscheidungen werden dann automatisch mit den Architekturentscheidungen vom SE-Repo des angegebenen Commits automatisch zusammengeführt. Wie bei dem Pull-Befehl wird auch hier bei auftretenden Konflikten eine grafische Benutzeroberfläche angezeigt, die ein einfaches Lösen der Konflikte erlaubt.

Listing 6.6 Parameterüberblick des Merge-Befehls

```
1 $ eadlsync merge -h
2 use 'eadlsync merge <commit-id>' to merge the local decisions with the decisions of the
      selected commit from the se-repo
3 Usage: merge [options] commit-id
4       Options:
5         -h, --help
6              Show the usage of this command
```

Listing 6.7 Parameterüberblick des Reset-Befehls

```
1 $ eadlsync reset -h
2 use 'eadlsync reset <commit-id>' to reset the local decisions to the decisions of the
      selected commit from the se-repo
3 Usage: reset [options] commit-id
4       Options:
5         -h, --help
6              Show the usage of this command
```

6.4.6. Reset

Ähnlich dem Merge-Befehl, ist auch der Reset-Befehl eine Ergänzung des Pull-Befehls. Durch den Reset-Befehl können die lokalen Architekturentscheidungen auf die eines ausgewählten Commits des SE-Repos zurückgesetzt werden. Somit ersetzt der Reset-Befehl einen optionalen Zwangsparameter beim Merge-, bzw. Pull-Befehl. Wie Listing 6.7 zeigt, sind neben der Commit ID keine weiteren Parameter nötig.

6.4.7. Status

Der Status-Befehl wurde implementiert, um dem Anwender einen Überblick über den aktuellen Status der Synchronisation zu verschaffen. Es wird auf der Konsole angezeigt, ob lokale bzw. entfernte Änderungen vorliegen und falls nötig, ob diese automatisch zusammengeführt werden können. Zudem werden Vorschläge angezeigt, welche Befehle in der jeweiligen Situation ausgeführt werden sollten, damit die lokalen Architekturentscheidungen mit denen des letzten Commits im SE-Repo übereinstimmen. Wie Listing 6.8 zeigt, müssen keine weiteren Parameter angegeben werden.

Listing 6.8 Parameterüberblick des Status-Befehls

```
1 $ eadlsync status -h
2 use 'eadlsync status' to show the status of the eadlsync
3 Usage: status [options]
4       Options:
5         -h, --help
6              Show the usage of this command
```

Listing 6.9 Parameterüberblick des Statistic-Befehls

```
$ eadlsync statistic -h
use 'eadlsync statistic' to display statistics about the decisions in this code repository
Usage: statistic [options]
      Options:
        -h, --help
             Show the usage of this command
```

Listing 6.10 Parameterüberblick des Diff-Befehls

```
$ eadlsync diff -h
use 'eadlsync diff <commit-id>' to view the diff of the specified decisions compared to
    the local decisions
Usage: diff [options] commit-id
      Options:
        -g, --gui
             display the diff in a JavaFX window
             Default: false
        -h, --help
             Show the usage of this command
```

6.4.8. Statistic

Der Statistic-Befehl listet einige Statistiken über die eingebetteten (lokalen) Architekturentscheidungen. Es wird für jedes Paket angezeigt, wie viele Klassen existieren sind und in wie vielen von diesen, eingebettete Architekturentscheidungen vorhanden sind. Für jedes Paket und einmal für alle Pakete, wird zudem die Entscheidungsüberdeckung in Prozent angegeben. Auch berücksichtigt wird, in wie vielen Entscheidungen einzelne Felder, wie *Context*, *Facing*, *Achieving* oder *Accepting* nicht angegeben sind. Zudem werden weitere Metriken über diese Felder angezeigt. Listing 6.9 zeigt, dass keine weiteren Parameter spezifiziert werden müssen.

6.4.9. Diff

Der Diff-Befehl zeigt Änderungen der Architekturentscheidungen eines bestimmten Commits zu den lokalen Architekturentscheidungen an. Dazu muss lediglich die gewünschte Commit ID als Parameter angegeben werden. Die Unterschiede der Architekturentscheidungen werden dann übereinander in der Konsole angezeigt. Wie Listing 6.10 zeigt, kann durch einen optional Parameter die Diff der Architekturentscheidungen in grafischen Benutzeroberfläche geöffnet werden. Abbildung 6.3 zeigt wie die Oberfläche des Diff Viewers aussieht, anhand von Beispiel Architekturentscheidungen im Open-Source Projekt JabRef.

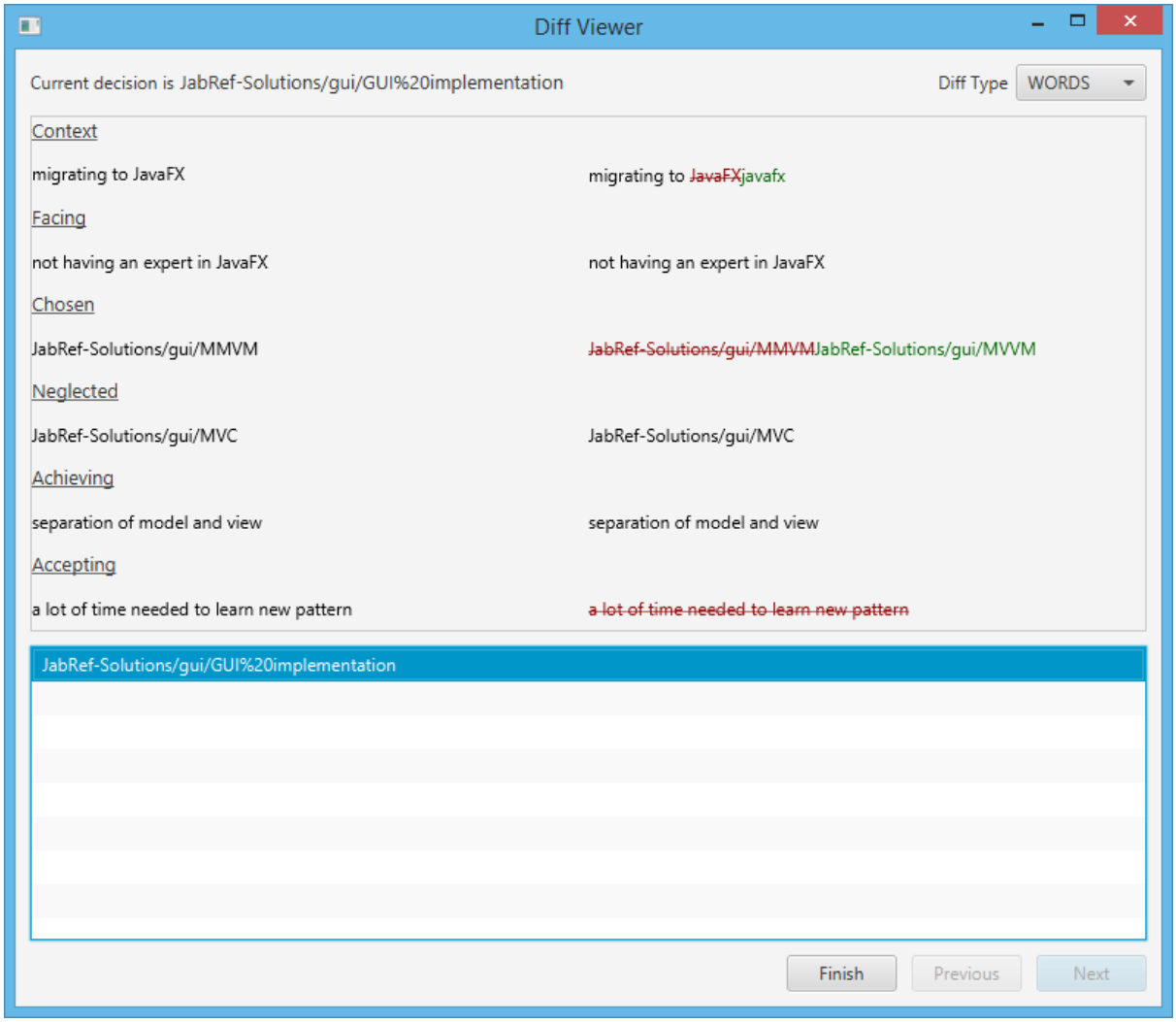


Abbildung 6.3.: Der Diff Viewer bietet eine grafische Benutzeroberfläche zum Visualisieren der Unterschiede der ausgewählten Architekturentscheidungen

6.4.10. Config

Durch den Config-Befehl können die Werte in der Konfigurationsdatei, welche das Synchronisationsprogramm benutzt, geändert werden. Somit lassen sich auch die Werte, der beim initialisieren gesetzten Parameter nachträglich noch ändern. In Listing 6.11 sind alle Werte aufgezählt, die sich durch diesen Befehl nachträglich noch ändern lassen.

Listing 6.11 Parameterüberblick des Config-Befehls

```
$ eadlsync config -h
use 'eadlsync config --[user|core].<key>=<value>' to update the decisions in the se-repo
Usage: config [options]
      Options:
        --core.project
            set the project to work with, this shall match a se-repo repository name
        --core.root
            set the root directory of the code base
        --core.url
            set the base url for the se-repo, the url shall be in a format of
                '<host>/serepo' or 'localhost'
        -h, --help
            Show the usage of this command
        --user.email
            change the commit email of this program
        --user.name
            change the commit name of this program
```

Listing 6.12 Parameterüberblick des Deinit-Befehls

```
$ eadlsync deinit -h
use 'eadlsync deinit' de-initialize eadlsync in this directory
Usage: deinit [options]
      Options:
        -h, --help
            Show the usage of this command
```

6.4.11. Deinit

Der Deinit-Befehl ist das Pendant zum Init-Befehl und de-initialisiert das Synchronisationsprogramm in dem Verzeichnis. Dabei werden die Dateien, welche das Synchronisationsprogramm benötigt, gelöscht. Lokale Architekturentscheidungen bleiben jedoch unberührt. Der Befehl wurde hauptsächlich erstellt, da es in der Anfangsphase des Testens zu falschen Commit IDs in der Synchronisationsdatei gekommen ist. Durch das De-initialisieren und anschließendem Ausführen des Init-Befehls, kann solch ein falscher Programmzustand behoben werden. Der Deinit-Befehl muss in dem Verzeichnis ausgeführt werden, in welchem der vorausgegangene Init-Befehl auch ausgeführt wurde, denn wie Listing 6.12 zeigt, können keine weiteren Parameter angegeben werden.

7. Fallstudie

Zur Verifizierung des Synchronisationsprogramms wurde eine Fallstudie am Beispiel von JabRef durchgeführt. Die Open-Source Literaturverwaltungssoftware hat ungefähr 200.000 Codezeilen (LOC). Es wurde nicht nur auf die korrekte Arbeitsweise der Befehle geachtet, sondern auch auf die Dauer der Ausführung.

In Abschnitt 7.1 wird beschrieben, wie der Aufbau der Fallstudie aussieht und wie die Durchführung abläuft. Im Abschnitt 7.2 wird das Synchronisationsprogramms beurteilt.

7.1. Vorbereitung und Durchführung

Für die Fallstudie wurde das SE-Repo auf dem Localhost ausgeführt und mit Hilfe des SE-Repo Webinterfaces ein Repository für JabRef erstellt. Es wurden jeweils drei Architekturentscheidungen mit AD-Mentor modelliert bzw. als Y-Statement Annotation im Quellcode hinterlegt. Im SE-Repo wurde ein Commit erstellt, in dem die modellierten Architekturentscheidungen enthalten sind. Zudem wurden zwei weitere Commits erstellt, in denen diese modifiziert, bzw. gelöscht wurden. Die Daten sind echten Architekturentscheidungen nachempfunden und sind öffentlich auf GitHub[1] verfügbar.

Im Laufe der Fallstudie wurde die ID der letzten Synchronisation manuell verändert, sodass alle Befehle mindestens einmal ausgeführt werden konnten. Bei der Initialisierung des Synchronisationsprogramms wurde *src/main/java* als oberstes Verzeichnis des Quellcodes angegeben. Auf diese Weise konnte die Anzahl der relevanten LOC auf ungefähr 130.000 gesenkt werden. Alle Programmaufrufe wurden mit der Option *--log info* ausgeführt, sodass die Ausführungsdauer nach Beendigung auf der Konsole angezeigt wird.

7.2. Auswertung

Alle Befehle wurden erfolgreich ausgeführt und lieferten das gewünschte Ergebnis. Die Programmaufrufe variierten jedoch stark in ihrer Ausführungszeit, wie Tabelle 7.1 zeigt. Zieht man dabei in Betracht, ob ein Befehl auf die lokalen Quellcodedateien oder auf das SE-Repo

[1] https://github.com/boceckts/jabref/tree/eadl

Befehl	Dauer in [s]	Code-Repo Interaktion	SE-Repo Interaktion
Init	1	Nein	Nein
Deinit	1	Nein	Nein
Config	1	Nein	Nein
Statistic	8	Ja	Nein
Status	13	Ja	Ja
Diff	11	Ja	Ja
Pull	12	Ja	Ja
Merge	12	Ja	Ja
Reset	11	Ja	Ja
Commit	18	Ja	Ja
Sync	31	Ja	Ja

Tabelle 7.1.: Die Tabelle zeigt eine Übersicht der Befehle und gibt deren durchschnittliche Ausführungsdauer, sowie die Interaktionen mit dem Quellcode bzw. SE-Repo an

zugreift, so sind die Unterschiede in der Ausführungszeit verständlicher. Listing A.1 im Anhang zeigt, dass in ungefähr 8 Sekunden 992 Klassen in 158 Paketen durchsucht werden können. Da jeder der Synchronisationsbefehle zunächst eine Diff der lokalen Architekturentscheidungen zu dem Stand der Architekturentscheidungen der letzten Synchronisation erstellt, benötigen diese Befehle in unserem Beispiel mindestens 8 Sekunden. Abgesehen von dem Sync- bzw. Commit-Befehl, benötigten die Synchronisationsbefehle zwischen 3 und 5 Sekunden für die Interaktion mit dem SE-Repo. Der große Unterschied zu der Ausführungsdauer des Commit-Befehls kommt vermutlich daher, dass dieser Befehl nicht nur die Daten des SE-Repo abruft, sonder auch Daten in dieses committet. Die Ausführungsdauer des Sync-Befehls entspricht in etwa der des Pull-Befehls addiert mit der des Commit-Befehls, da der Sync-Befehl eben diese hintereinander ausführt.

8. Zusammenfassung und Ausblick

In Kapitel 2 wurde beschrieben, wie Architekturentscheidungen in Form eines Y-Statements extern dokumentiert werden können. Außerdem wurde eine neue Art der Annotation vorgestellt, welche es erlaubt die Architekturentscheidungen direkt in Java Quellcode Dateien zu schreiben. In Kapitel 3 wurde das UML Modelierungswekzeug AD-Mentor vorgestellt, durch welches ein Y-Statement in Form von abhängigen Knoten abstrakt modelliert werden kann. Zudem wurde das SE-Repo vorgestellt, in welchem die so modellierten Architekturentscheidungen versioniert werden können.

Kapitel 4 stellt das Problem der nicht vorhandenen Synchronisation zwischen modellierten und eingebetteten Architekturentscheidungen dar. Zudem werden Anforderungen für die Synchronisation spezifiziert und zusätzliche Herausforderungen für diese aufgezählt. In Kapitel 5 wurde das Konzept für die Synchronisation beschrieben und insbesondere auf die unterschiedlichen Fälle bei lokalen bzw. entfernten Änderungen der Architekturentscheidungen eingegangen. In Kapitel 6 wurde zunächst die Architektur und Implementierung des Synchronisationsprogramms beschrieben. Außerdem wurden die implementierten Funktionen vorgestellt. Im Rahmen der Arbeit wurde zudem eine Fallstudie durchgeführt, welche die Funktionalität des Synchronisationsprogramms verifizieren soll. Die Ergebnisse dieser werden als Abschluss in Kapitel 7 vorgestellt.

Ausblick

Aktuell sind die eingebettete Architekturentscheidungen für die Programmiersprache Java in Form einer Annotation vorhanden. Diese kann bisher nur auf Typ Ebene angewendet werden. Im Weiteren Schritt sollten jedoch auch Methoden und Felder unterstützt werden. Auf diese Weise können die eingebetteten Architekturentscheidungen genau auf die resultierenden Komponenten abgebildet werden. Außerdem sollten eingebettete Architekturentscheidungen für weitere Programmiersprachen erstellt werden. Da die modellierten Architekturentscheidungen im SE-Repo sprachunabhängig sind, funktioniert das Konzept der Synchronisation auch bei anderen Programmiersprachen. Das Synchronisationsprogramm muss dazu lediglich weitere Parser für das Parsen von eingebetteten Architekturentscheidungen in anderen Programmiersprachen implementieren.

Soll das Synchronisationsprogramm jedoch weitere Annotationen mit anderen Felder unterstützen, so muss die interne Speicherstruktur angepasst werden.

A. Anhang

A.1. Fallstudie

```
Statistic of the decisions in this code
      project 'JabRef' at http://localhost:8080/serepo
Processing decision statistic
      source folder is C:\Users\Tobias\git\jabref\src\main\java

decisions by packages that have at least one class
   0 decisions for 1 classes (0,0%) in package radar\ad\annotations
   0 decisions for 3 classes (0,0%) in package oracle\jdbc
   0 decisions for 5 classes (0,0%) in package org\jabref\shared\exception
   0 decisions for 1 classes (0,0%) in package org\jabref\gui\dbproperties
   0 decisions for 9 classes (0,0%) in package
       org\jabref\logic\formatter\casechanger
   0 decisions for 2 classes (0,0%) in package org\jabref\gui\contentselector
   0 decisions for 7 classes (0,0%) in package org\jabref\logic\bibtex
   0 decisions for 3 classes (0,0%) in package org\jabref\logic\undo
   0 decisions for 4 classes (0,0%) in package
       org\jabref\gui\fieldeditors\contextmenu
   0 decisions for 2 classes (0,0%) in package org\jabref\gui\autosaveandbackup
   0 decisions for 1 classes (0,0%) in package org\jabref\shared\security
   0 decisions for 10 classes (0,0%) in package org\jabref\model\database
   0 decisions for 8 classes (0,0%) in package org\jabref\gui\exporter
   0 decisions for 4 classes (0,0%) in package org\jabref\logic\citationstyle
   0 decisions for 8 classes (0,0%) in package org\jabref\shared
   1 decisions for 5 classes (20,0%) in package org\jabref\logic\pdf
   0 decisions for 2 classes (0,0%) in package org\jabref\shared\listener
   0 decisions for 5 classes (0,0%) in package org\jabref\model\metadata
   0 decisions for 2 classes (0,0%) in package org\jabref\logic\logging
   0 decisions for 3 classes (0,0%) in package org\jabref\model\util
   0 decisions for 9 classes (0,0%) in package org\jabref\gui\externalfiles
   0 decisions for 11 classes (0,0%) in package org\jabref\gui\journals
   0 decisions for 2 classes (0,0%) in package org\jabref\gui\protectedterms
   0 decisions for 15 classes (0,0%) in package org\jabref\gui\entryeditor
   0 decisions for 7 classes (0,0%) in package org\jabref\gui\mergeentries
   0 decisions for 41 classes (0,0%) in package org\jabref\gui\fieldeditors
```

```
33     0 decisions for 2 classes (0,0%) in package org\jabref\logic\remote
34     0 decisions for 4 classes (0,0%) in package org\jabref\gui\util\comparator
35     0 decisions for 8 classes (0,0%) in package org\jabref\logic\util
36     0 decisions for 3 classes (0,0%) in package org\jabref\gui\cleanup
37     0 decisions for 5 classes (0,0%) in package org\jabref\model
38     0 decisions for 29 classes (0,0%) in package org\jabref\model\entry
39     0 decisions for 3 classes (0,0%) in package org\jabref\shared\event
40     0 decisions for 2 classes (0,0%) in package org\jabref\logic\auxparser
41     0 decisions for 4 classes (0,0%) in package
           org\jabref\logic\search\rules\describer
42     0 decisions for 12 classes (0,0%) in package org\jabref\logic\autocompleter
43     0 decisions for 1 classes (0,0%) in package org\jabref\logic\remote\shared
44     0 decisions for 8 classes (0,0%) in package org\jabref\gui\worker
45     0 decisions for 5 classes (0,0%) in package org\jabref\model\search\matchers
46     0 decisions for 1 classes (0,0%) in package org\jabref\gui\auximport
47     0 decisions for 4 classes (0,0%) in package org\jabref\gui\customentrytypes
48     0 decisions for 3 classes (0,0%) in package org\jabref\model\bibtexkeypattern
49     0 decisions for 1 classes (0,0%) in package org\jabref\logic\remote\client
50     0 decisions for 10 classes (0,0%) in package org\jabref\preferences
51     0 decisions for 6 classes (0,0%) in package org\jabref\model\search\rules
52     0 decisions for 1 classes (0,0%) in package org\jabref\logic\specialfields
53     0 decisions for 5 classes (0,0%) in package org\jabref\gui\desktop\os
54     0 decisions for 3 classes (0,0%) in package oracle\jdbc\dcn
55     0 decisions for 2 classes (0,0%) in package
           org\jabref\model\entry\specialfields
56     0 decisions for 1 classes (0,0%) in package org\jabref\gui\remote
57     0 decisions for 2 classes (0,0%) in package org\jabref\gui\search\matchers
58     0 decisions for 1 classes (0,0%) in package org\jabref\model\metadata\event
59     0 decisions for 1 classes (0,0%) in package oracle\jdbc\driver
60     0 decisions for 7 classes (0,0%) in package org\jabref\gui\search
61     0 decisions for 14 classes (0,0%) in package org\jabref\gui\importer
62     0 decisions for 7 classes (0,0%) in package org\jabref\gui\specialfields
63     0 decisions for 10 classes (0,0%) in package org\jabref\gui\keyboard
64     0 decisions for 4 classes (0,0%) in package org\jabref\gui\errorconsole
65     0 decisions for 23 classes (0,0%) in package org\jabref\logic\importer
66     0 decisions for 4 classes (0,0%) in package org\jabref\logic\xmp
67     0 decisions for 6 classes (0,0%) in package org\jabref\gui\externalfiletype
68     0 decisions for 21 classes (0,0%) in package org\jabref\gui\actions
69     0 decisions for 1 classes (0,0%) in package org\jabref\shared\prefs
70     0 decisions for 6 classes (0,0%) in package org\jabref\logic\bibtex\comparator
71     0 decisions for 13 classes (0,0%) in package
           org\jabref\logic\formatter\bibtexfields
72     0 decisions for 9 classes (0,0%) in package org\jabref\gui\util\component
73     0 decisions for 1 classes (0,0%) in package
           org\jabref\logic\importer\fileformat\mods
```

```
74    0 decisions for 1 classes (0,0%) in package org\jabref\gui\desktop
75    0 decisions for 20 classes (0,0%) in package org\jabref\logic\exporter
76    0 decisions for 10 classes (0,0%) in package org\jabref\gui\documentviewer
77    0 decisions for 11 classes (0,0%) in package org\jabref\gui\maintable
78    0 decisions for 1 classes (0,0%) in package org\jabref\logic\groups
79    0 decisions for 13 classes (0,0%) in package org\jabref\logic\bst
80    0 decisions for 4 classes (0,0%) in package org\jabref\gui\shared
81    0 decisions for 3 classes (0,0%) in package org\jabref\model\strings
82    0 decisions for 31 classes (0,0%) in package org\jabref\logic\integrity
83    0 decisions for 1 classes (0,0%) in package org\jabref\logic\formatter\minifier
84    0 decisions for 8 classes (0,0%) in package org\jabref\logic\msbib
85    0 decisions for 1 classes (0,0%) in package org\jabref\model\groups\event
86    0 decisions for 9 classes (0,0%) in package org\jabref\logic\importer\util
87    0 decisions for 7 classes (0,0%) in package org\jabref\model\entry\identifier
88    0 decisions for 42 classes (0,0%) in package org\jabref\gui
89    0 decisions for 15 classes (0,0%) in package org\jabref\gui\util
90    0 decisions for 5 classes (0,0%) in package org\jabref\model\database\event
91    0 decisions for 5 classes (0,0%) in package org\jabref\gui\importer\actions
92    0 decisions for 22 classes (0,0%) in package
          org\jabref\logic\importer\fileformat
93    0 decisions for 6 classes (0,0%) in package org\jabref
94    0 decisions for 13 classes (0,0%) in package org\jabref\model\groups
95    0 decisions for 2 classes (0,0%) in package org\jabref\logic\formatter
96    0 decisions for 11 classes (0,0%) in package org\jabref\logic\util\io
97    0 decisions for 8 classes (0,0%) in package org\jabref\cli
98    0 decisions for 4 classes (0,0%) in package org\jabref\gui\autocompleter
99    0 decisions for 2 classes (0,0%) in package org\jabref\model\search
100   0 decisions for 4 classes (0,0%) in package org\jabref\logic\search
101   1 decisions for 4 classes (25,0%) in package org\jabref\model\entry\event
102   0 decisions for 12 classes (0,0%) in package org\jabref\gui\undo
103   0 decisions for 13 classes (0,0%) in package org\jabref\logic\cleanup
104   0 decisions for 4 classes (0,0%) in package org\jabref\logic\protectedterms
105   0 decisions for 6 classes (0,0%) in package org\jabref\logic\journals
106   0 decisions for 2 classes (0,0%) in package org\jabref\gui\logging
107   0 decisions for 2 classes (0,0%) in package org\jabref\gui\plaintextimport
108   0 decisions for 12 classes (0,0%) in package org\jabref\gui\push
109   0 decisions for 3 classes (0,0%) in package org\jabref\gui\renderer
110   0 decisions for 7 classes (0,0%) in package org\jabref\logic\util\strings
111   0 decisions for 17 classes (0,0%) in package org\jabref\collab
112   0 decisions for 3 classes (0,0%) in package org\jabref\migrations
113   0 decisions for 5 classes (0,0%) in package org\jabref\gui\filelist
114   0 decisions for 8 classes (0,0%) in package org\jabref\logic\layout
115   0 decisions for 1 classes (0,0%) in package osx\macadapter
116   0 decisions for 21 classes (0,0%) in package org\jabref\gui\preftabs
117   0 decisions for 2 classes (0,0%) in package org\jabref\model\pdf
```

```
118     0 decisions for 22 classes (0,0%) in package org\jabref\gui\groups
119     0 decisions for 12 classes (0,0%) in package org\jabref\gui\importer\fetcher
120     1 decisions for 6 classes (16,7%) in package org\jabref\gui\help
121     0 decisions for 4 classes (0,0%) in package org\jabref\gui\bibtexkeypattern
122     0 decisions for 12 classes (0,0%) in package org\jabref\gui\openoffice
123     0 decisions for 1 classes (0,0%) in package org\jabref\logic
124     0 decisions for 3 classes (0,0%) in package org\jabref\gui\menus
125     0 decisions for 2 classes (0,0%) in package org\jabref\logic\bibtexkeypattern
126     0 decisions for 9 classes (0,0%) in package org\jabref\logic\openoffice
127     0 decisions for 3 classes (0,0%) in package org\jabref\pdfimport
128     0 decisions for 24 classes (0,0%) in package org\jabref\logic\importer\fetcher
129     0 decisions for 4 classes (0,0%) in package org\jabref\model\cleanup
130     0 decisions for 6 classes (0,0%) in package org\jabref\logic\net
131     0 decisions for 2 classes (0,0%) in package org\jabref\logic\autosaveandbackup
132     0 decisions for 7 classes (0,0%) in package org\jabref\logic\l10n
133     0 decisions for 66 classes (0,0%) in package org\jabref\logic\layout\format
134     0 decisions for 1 classes (0,0%) in package org\jabref\logic\help
135     0 decisions for 4 classes (0,0%) in package org\jabref\logic\remote\server
136     --------------------
137     3 total decisions for 992 classes in 158 packages resulting in a decision
            coverage of 0,3%
138
139 decision field CONTEXT
140     0 decisions (0,0%) have the field CONTEXT not specified
141     minimum content length is 19
142     maximum content length is 49
143     average content length is 31,0
144     content length median is 26,0
145
146 decision field FACING
147     2 decisions (66,7%) have the field FACING not specified
148     minimum content length is 30
149     maximum content length is 30
150     average content length is 10,0
151     content length median is 30,0
152
153 decision field NEGLECTED
154     minimum content length is 1
155     maximum content length is 3
156     average neglected options 1,0
157     content length median is 1,0
158
159 decision field ACHIEVING
160     1 decisions (33,3%) have the field ACHIEVING not specified
161     minimum content length is 19
```

```
162     maximum content length is 28
163     average content length is 15,0
164     content length median is 23,0
165
166  decision field ACCEPTING
167     1 decisions (33,3%) have the field ACCEPTING not specified
168     minimum content length is 10
169     maximum content length is 41
170     average content length is 17,0
171     content length median is 25,0
172
173  18:24:24.202 [main] INFO com.eadlsync.cli.EADLSyncMain - Program finished in 8
        seconds
```

Listing A.1: Ausgabe des Statistik-Befehls

Literaturverzeichnis

[Bos04] J. Bosch. „Software architecture: The next step“. In: *European Workshop on Software Architecture*. Springer. 2004, S. 194–199 (zitiert auf S. 15).

[Büc17] A. Büchler. „Software Engineering Repository (SE-Repo) Gesamtkonzept, Umsetzung mit Git und Validierung“. Magisterarb. HSR Hochschule für Technik Rapperswil, 2017 (zitiert auf S. 27).

[CGB+02] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, R. Little. *Documenting software architectures: views and beyond*. Pearson Education, 2002 (zitiert auf S. 15).

[Cle96] P. C. Clements. „A survey of architecture description languages“. In: *Software Specification and Design, 1996., Proceedings of the 8th International Workshop on*. IEEE. 1996, S. 16–25 (zitiert auf S. 15).

[CS14] S. Chacon, B. Straub. *Pro git*. Apress, 2014, S. 6 (zitiert auf S. 25).

[GE11] F. Gilson, V. Englebert. „Rationale, decisions and alternatives traceability for architecture design“. In: *Proceedings of the 5th European Conference on Software Architecture: Companion Volume*. ACM. 2011, S. 4 (zitiert auf S. 13).

[Hap14] M. Happel. „Synchronisierung der Dokumentation von Software-Modulen“. B.S. thesis. Universität Stuttgart, 2014 (zitiert auf S. 29).

[HP06] A. Herrmann, B. Paech. „Lernen aus dokumentierten Architektur-Entscheidungen“. In: *Softwaretechnik-Trends* 26.4 (2006) (zitiert auf S. 23).

[JVAH07] A. Jansen, J. Van Der Ven, P. Avgeriou, D. K. Hammer. „Tool support for architectural decisions“. In: *Software Architecture, 2007. WICSA'07. The Working IEEE/IFIP Conference on*. Ieee. 2007, S. 4–4 (zitiert auf S. 23).

[Kop16] O. Kopp. *Embedded Architectural Decisions*. Accessed: 23.07.2017. 2016. URL: https://github.com/adr/embedded-adl (zitiert auf S. 20).

[LL13a] J. Ludewig, H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2013, S. 373 (zitiert auf S. 13).

[LL13b] J. Ludewig, H. Lichter. *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*. dpunkt. verlag, 2013, S. 366 (zitiert auf S. 13).

[PCF09] C. M. Pilato, B. Collins-Sussman, B. W. Fitzpatrick. *Versionskontrolle mit Subversion*. O'Reilly Germany, 2009 (zitiert auf S. 35).

[VAC+08] O. Vogel, I. Arnold, A. Chughtai, E. Ihler, T. Kehrer, U. Mehlig, U. Zdun. *Software-Architektur: Grundlagen-Konzepte-Praxis*. Springer Science & Business Media, 2008, S. 501 (zitiert auf S. 13).

[Wik17] Wikipedia. *Cache replacement policies — Wikipedia*. Accessed: 23.07.2017. 2017. URL: https://en.wikipedia.org/wiki/Cache_replacement_policies (zitiert auf S. 17).

[ZCTZ13] U. Zdun, R. Capilla, H. Tran, O. Zimmermann. „Sustainable architectural design decisions". In: *IEEE software* 30.6 (2013), S. 46–53 (zitiert auf S. 13, 15, 16, 23).

[Zim09] O. Zimmermann. „An architectural decision modeling framework for service-oriented architecture design". Diss. Stuttgart, Univ., Diss., 2009, 2009 (zitiert auf S. 23).

[ZWKG15] O. Zimmermann, L. Wegmann, H. Koziolek, T. Goldschmidt. „Architectural decision guidance across projects-problem space modeling, decision backlog management and cloud computing knowledge". In: *Software Architecture (WICSA), 2015 12th Working IEEE/IFIP Conference on*. IEEE. 2015, S. 85–94 (zitiert auf S. 13, 15, 23–25).

Alle URLs wurden zuletzt am 19. 07. 2017 geprüft.

Erklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum, Unterschrift