Institut für Softwaretechnologie
Abteilung Software Engineering
Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Diplomarbeit

Software Repositories Mining von Issue Tasks und Coupled File Changes

Deniz Alakus

Studiengang:	Informatik
Prüfer:	Prof. Dr. rer. nat. Stefan Wagner
Betreuer:	M. Sc Jasmin Ramadani
Begonnen am:	29. September 2016
Beendet am:	31. März 2017
CR-Nummer:	D.2.7, H.2.8

Inhaltsverzeichnis

Abkürzungsverzeichnis

CLI	Command Line Interface
IDE	Integrated Development Environment
GUI	Graphical User Interface
SRM	Software-Repository Mining
SQL	Structured Query Language
SWT	Standard Widget Toolkit
RCP	Rich Client Platform
VCS	Version Control System
CVS	Concurrent Versions System
DMIO	Data Mining Input Objekt
DMRO	Data Mining Result Objekt

1 Einleitung

Immer mehr Softwareprojekte werden über Versionsverwaltungssysteme (VCS) wie CVS und Git verwaltet. Mit der Zeit häufen sich viele Daten in einem durch VCS verwalteten Software-Repository an. Bugtracker wie Bugzilla sind ebenfalls Software-Repositories, die zusammen mit ihrer zugehörigen Software wachsen, aber sie sind trotzdem getrennte Entitäten. SRM (Software Repository Mining) kann helfen, um aus diesen Daten nützliche Informationen zu gewinnen. In diesem Zusammenhang sind Coupled Changes [1] besonders interessant. Indem Data Mining, wie Frequent Pattern Mining, auf den Software-Repositories ausgeführt wird, können Dateien identifiziert werden, die oft miteinander geändert wurden. Diese Dateien stellen Coupled Changes dar und basieren auf der Software-Historie einer Software. Coupled Changes können unerfahrenen Entwicklern bei immer komplexer werdender Software helfen, ihre Wartungsaufgaben durchzuführen.

Im Rahmen dieser Arbeit soll ein auf Eclipse basierendes Tool erstellt werden, welches die „Maintenance Task Issues“ eines Software-Projekts anzeigen und Coupled Changes auf Basis der Issues extrahieren kann. Als Data Mining Algorithmen kommen der Frequent Itemset Mining Algorithmus FPGrowth und der Sequential Pattern Mining Algorithmus Prefixspan zum Einsatz.

1.1 Gliederung

Diese Arbeit ist wie folgt gegliedert:

Kapitel 1 – Einleitung:

Einführung in das Themengebiet

Kapitel 2 – Grundlagen:

Erläuterung der Grundlagen

Kapitel 3 – Verwandte Arbeiten:

Beschreibung von Arbeiten mit Bezug auf das Themengebiet

Kapitel 4 – Anforderungen und Analyse:

Anforderungen an das Tool und Analyse

Kapitel 5 – Konzept und Architektur:

Konzept und Architektur des zu erstellenden Tools

Kapitel 6 – Implementierung:

Beschreibung der Implementierung des Tools

Kapitel 7 – Evaluierung:

Evaluierung des erstellten Tools

Kapitel 8 – Zusammenfassung:

Abschließende Worte und Ausblick

2 Grundlagen

In diesem Kapitel werden die Grundlagen und verwendeten Technologien erläutert.

2.1 Pattern Mining

Data Mining beschäftigt sich mit dem Sammeln, Verarbeiten und Analysieren von Daten, um nützliche Einsichten zu gewinnen [2]. Ein Teilbereich des Data Minings ist Pattern Mining, welches sich mit interessanten, nützlichen und unerwarteten Mustern in Datenbanken beschäftigt. Mit Hilfe dieser Technik können versteckte Muster in großen Datenbanken zum Vorschein gebracht werden [3].
In diesem Kapitel werden die Pattern Mining Techniken Frequent Itemset Mining und Sequential Pattern Mining erläutert und verglichen.

2.1.1 Frequent Itemset Mining

Frequent Itemset Mining berechnet häufig vorkommende Itemsets in einer Transaktionsdatenbank. Eine Transaktionsdatenbank DB setzt sich zusammen aus einer Menge von Tupeln (tid, T), wobei T für eine Transaktion steht und aus einer Menge von Items besteht (ein sogenanntes Itemset). Die Transaktions ID dient der eindeutigen Identifikation der Transaktion. Ein Itemset A kommt dann häufig in einer Transaktionsdatenbank DB vor, wenn die Anzahl der Transaktionen, in denen A enthalten ist (entspricht dem Support-Wert), einen vorbestimmten Wert, den Minimum-Support-Wert, übersteigt. In diesem Fall ist A ein Frequent Pattern. [4]

Um die Frequent Patterns einer Transationsdatenbank DB effizient zu berechnen, benutzt FPGrowth eine kompakte Datenstruktur mit dem Namen FP-tree.

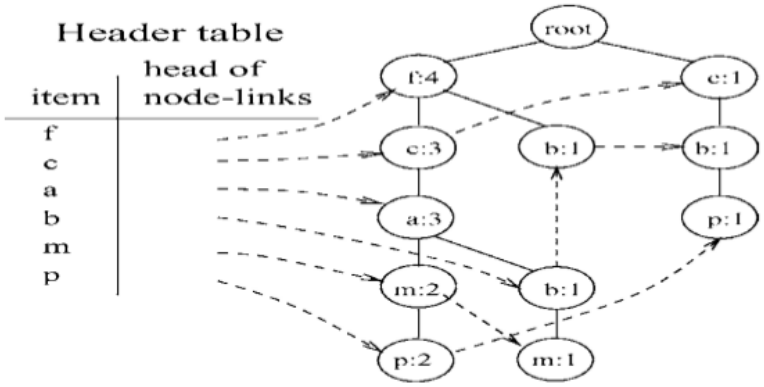

Abbildung 2.1: Beispiel für ein FP-tree [4]

Die Konstruktion des FP-tree wird durch zwei Durchläufe von DB bewerkstelligt. Im ersten Durchlauf wird der Support aller Items gezählt. Es entsteht eine Liste FList aus Items und ihrem Support. Items mit einem Support kleiner als Minimum-Support werden nicht in die Liste aufgenommen. Anschließend wird die Liste absteigend nach Support sortiert.

Im zweiten Durchlauf wird der FP-tree aus den Transaktionen in DB erzeugt. Dies wird anhand des Beispiel FP-trees in Abbildung 2.1 erläutert. Nicht häufig vorkommende Items werden aus den Transaktionen aussortiert. Frequent Items werden nach den Vorkommen der Items in FList sortiert. Werden als Frequent Items z.B. die Items in Abbildung 2.1 angenommen, bedeutet dies für eine Transaktion mit Items <b, c, p> eine Sortierung nach <c, b, p>. Die sortierten Transaktionen werden anschließend der Reihe nach in den Baum eingefügt. Transaktionen, welche einen Prefix einer bereits im Baum vorhandenen Transaktion enthalten, werden entlang des Prefixes in den Baum eingefügt. An der Stelle, wo sie sich unterscheiden, entsteht ein neuer Ast im Baum und der Rest der Items der Transaktion werden in den Ast eingefügt.

Damit die Frequent Itemsets berechnet werden können, wird eine Zeiger-Tabelle für die Frequent Items aufgebaut. Sie zeigen zu den ersten im FP-tree vorkommenden Items mit demselben Namen. Dies ist in Abbildung 2.1 gut zu sehen. Der Zeiger für das Item f z.B. zeigt auf das erste hinzugefügte Item f in FP-tree. Dasselbe gilt für die anderen Frequent Items. Die Knoten im FP-tree zeigen mit ihren Zeigern ebenfalls auf neu hinzugefügte Items mit demselben Namen.

Um die Frequent Patterns, an denen ein Frequent Item beteiligt ist, zu berechnen, wird zunächst über die Zeiger-Tabelle alle Transaktionen im FP-tree verfolgt, in denen das Frequent Item vorkommt. Daraufhin werden nur die Prefixpfade ohne das Item und dem Suffix betrachtet. Aus diesen wird ein neuer sogenannter Conditional FP-tree erzeugt und es werden die in diesem Baum vorkommenden Frequent Itemsets berechnet. Dies wird rekursiv solange wiederholt, bis alle Frequent Patterns gefunden sind. [4]

2.1.2 Sequential Pattern Mining

Für Daten, in denen die Reihenfolge eine wichtige Rolle spielt wie z.B. DNA-Sequenzen [5], ist Sequential Pattern Mining das bevorzugte Tool, denn es wurde für die Entdeckung von sequenziellen Mustern entworfen.

Sequential Pattern Mining nutzt eine sogenannte Sequenzdatenbank als Input. Eine Sequenzdatenbank besteht aus einer Menge von Tupeln <sid, s>, wobei sid eine eindeutige Sequenz ID ist und s eine Sequenz darstellt. Eine Sequenz besteht aus einer Liste von Itemsets (vgl. Kapitel 2.1.1) [5]. Aus dem Namen Sequenz lässt sich ableiten, dass die Reihenfolge, in der die Itemsets vorkommen, von äußerster Wichtigkeit sind.

Eine Sequenz $\alpha=\langle a_1 a_2 ... a_n \rangle$ ist eine Subsequenz von einer Sequenz $\beta=\langle b_1 b_2 ... b_m \rangle$, falls alle Itemsets in α als Untermenge von Itemsets in β in einer Sequenz vorkommen. Formal bedeutet dies, falls Zahlen $1 \leq j_1 < j_2 < ... < j_n \leq m$ existieren, sodass $a_1 \subseteq b_{j1,} a_2 \subseteq b_{j2}, ..., a_n \subseteq b_{jn}$ gilt, dann ist α eine Subsequenz von β. Wenn die Anzahl der Sequenzen (in der Sequenzdatenbank), in denen α als Subsequenz vorkommt, mindestens dem Minimum-Support-Wert entspricht, dann ist α ein Sequential Pattern. [5]

Das Ziel des Sequential Pattern Mining ist bei gegebenem Minimum-Support-Wert alle Sequential Patterns einer Sequenzdatenbank zu finden.

Prefixspan ist ein schneller Algorithmus designt zum Mining von Sequential Patterns. Der Mining Prozess findet statt, indem zuerst alle frequent length-1 Sequential Patterns gesucht werden. Hierzu wird die Datenbank einmal komplett gescannt. Danach wird der Suchraum nach den gefundenen length-1 Sequential Patterns partitioniert. Die Subsequenzen der Sequential Pattern können dann

gemint werden, indem ihre projizierten Datenbanken erstellt und rekursiv gemint werden [5].

2.1.3 Vergleich der Methoden

Sequential Pattern Mining und Frequent Itemset Mining haben unterschiedliche Ziele. Frequent Itemset Mining versucht die häufig vorkommenden Pattern einer Transaktionsdatenbank zu finden. Transaktionen bestehen aus Itemmengen, was bedeutet, dass Items jeweils nur einmal in einer Transaktion vorkommen können. Sequential Pattern Mining versucht hingegen häufig vorkommende Sequential Patterns in einer Sequenzdatenbank zu finden. Im Gegensatz zu Frequent Pattern Mining besteht eine Sequenz aus einer Folge von Itemsets. Das bedeutet, dass Items auch mehrfach in einer Sequenz vorkommen können. Des Weiteren ist die Reihenfolge der Itemsets in einer Sequenz wichtig. Prefixspan und FPGrowth als Repräsentanten ihrer Klassen haben gemeinsam, dass sie beide Pattern-Growth Methoden ohne Kandidatengenerierung sind [4][5].

2.2 CSV-Dateien

CSV steht für „comma seperated values“ [6] und ist ein Format zur Speicherung von Tabellen, Datenbanken u.ä. Datensätze. So lassen sich mit diesem Format beispielsweise Tabellen aus Tabellenkalkulationsprogrammen auf einfache Art exportieren, um diese anderen Personen zur Verfügung zu stellen. Auch Kontakte aus E-Mail-Programmen lassen sich auf diese Weise exportieren und sichern.

CSV-Dateien sind simple Textdateien, die sich mit herkömmlichen Texteditoren öffnen und bearbeiten lassen. Auch Tabellenkalkulationsprogramme wie Microsoft Excel sind in der Lage sie zu öffnen. Excel zeigt die Dateien dabei in tabellarischer Form an. Im Gegensatz dazu werden CSV-Dateien von Texteditoren in ihrer eigentlichen Form angezeigt.

2.2.1 Aufbau einer CSV-Datei

Obwohl der Aufbau einer CSV-Datei meist recht einfach ist, gibt es für CSV-Dateien keinen einheitlichen Standard. Programme können die Struktur von CSV-Dateien frei bestimmen, was dadurch zu Inkompatibilitäten bei Software führen kann, welche eine andere Struktur erwarten. Trotz dieser Vielzahl von

möglichen Formaten hat sich ein allgemeiner Aufbau durchgesetzt, so wie sie von einer großen Mehrheit von Software akzeptiert wird. Der allgemeine Aufbau, ist in RFC 4180 [6] von der IETF (Internet Engineering Task Force) dokumentiert worden. In Listing 2.1 wird der Aufbau von CSV-Dateien als ABNF Grammatik beschrieben.

```
file = [header CRLF] record *(CRLF record) [CRLF]
header = name *(COMMA name)
record = field *(COMMA field)
name = field
field = (escaped / non-escaped)
escaped = DQUOTE *(TEXTDATA / COMMA / CR / LF / 2DQUOTE) DQUOTE
non-escaped = *TEXTDATA
```

Listing 2.1: ABNF Grammatik von CSV-Dateien [6]

Eine CSV-Datei besteht also aus einem optionalem Header, welcher die Spalten der Datei beschreibt. Jede darauf folgende Zeile ist ein Record. Die Records selbst sind in Felder unterteilt, welche Text enthalten. Das Trennzeichen, das die Felder voneinander trennt, ist das Komma. Die Anzahl der Felder pro Record muss einheitlich sein und auch mit der Anzahl der Felder im Header (falls vorhanden) übereinstimmen.

Als Trennzeichen kommen neben dem Komma oft auch Doppelpunkte oder Semikola vor.

2.3 Eclipse Rich Client Platform

Eclipse RCP ist eine Software Plattform für die Entwicklung von Desktop Anwendungen. Hierfür bietet RCP ein Grundgerüst aus Komponenten, welche beliebig erweitert werden können.

Das wohl bekannteste Anwendungsbeispiel von Eclipse RCP ist die Eclipse IDE. Sie demonstriert die Mächtigkeit von RCP, mit ihrer Erweiterbarkeit und sinnvollen Nutzung von Eclipse Plugins und Komponenten.

Mit RCP lassen sich in kurzer Zeit komplexe Anwendungen realisieren, die zudem Plattformunabhängig sind [7]. Die bereitgestellten grafischen Komponenten entsprechen den grundlegenden Bedürfnissen einer Benutzeroberfläche. RCP

Anwendungen lassen sich über die Eclipse IDE einfach programmieren, da sie Plugins für ihre Erstellung bereitstellt. Diese nehmen dem Programmierer einen großen Teil der Entwicklungslast ab.

Es gibt eine große Community, die Eclipse basierende Plugins und Tools entwickeln und diese der Allgemeinheit zur Verfügung stellen. Entwickler können sich auf diese Weise um die eigentliche Software kümmern, ohne sich um die Implementation dieser Funktionalitäten kümmern zu müssen.

2.3.1 Dependency Injection

Eines der größten und wichtigsten Neuerungen in RCP 4 ist die Einführung eines serviceorientierten Programmiermodells [8]. Im Gegensatz zu RCP 3, wird in RCP 4 bevorzugt Dependency Injection (DI) verwendet.

```
BundleContext ctx=FrameworkUtil.getBundle(Eventsender.class)
.getBundleContext();
ServiceReference<EventAdmin> sref =
ctx.getServiceReference(EventAdmin.class);
EventAdmin eventAdmin = ctx.getService(sref);
Map<String,Object> properties = new HashMap<String, Object>();
properties.put("date", new Date());
Event event = new Event("dateEvent", properties);
eventAdmin.postEvent(event);
```

Listing 2.2: Beispiel zum Senden von Events in RCP 3

Um ein EventAdmin Objekt zu erhalten ist es in RCP 3 erforderlich, einen Umweg über mehrere Klassen zu machen. Erst muss über die FrameworkUtil ein BundleContext erlangt werden. Über diesen und einer ServiceReference auf die EventAdmin Klasse kann schließlich ein EventAdmin Objekt erhalten werden. Das gewünschte Event kann nun über das EventAdmin Objekt versendet werden. In Listing 2.3 ist der Code abgebildet, der dasselbe Ergebnis mit RCP 4 erreicht.

```
@Inject
IEventBroker eventBroker;
eventBroker.postEvent("date", new Date());
```

Listing 2.3: Beispiel zum Senden von Events in RCP 4

Im Vergleich zu RCP 3 wird in RCP 4 ein IEventBroker Objekt verwendet. Auf den ersten Blick fällt auf, dass dieselbe Funktionalität, die in Listing 2.3 abgebildet ist, mit RCP 4 in nur 3 Zeilen ausgedrückt werden kann. Der Umweg über die

FramworkUtil und dem BundleContext fällt weg. Erreicht wird dies über die „@Inject“ Annotation, welche dem Eclipse Framework die Bürde des Auffindens des IEventBroker Objekts auflädt. Sobald es gefunden ist, wird es in die eventBroker Variable „injiziert“. Das Senden eines Events erfolgt schließlich über den Aufruf der postEvent Methode mit den zugehörigen Parametern.

2.3.2 Annotationen

In RCP 4 existieren verschiedene Annotationen, die von der Plattform für Dependency Injection und Callbacks eingesetzt werden. In Tabelle 2.1 ist eine Auswahl der von RCP 4 unterstützten Annotationen aufgelistet.

Annotation	**Beschreibung**
@Inject	Zur Injektion von Feldern, Methoden und Konstruktoren
@Optional	Markiert Felder, Methoden und Parameter als Optional
@UIEventTopic("TOPIC")	Registriert eine Methode für den Empfang von bestimmten Ereignissen
@PostConstruct	Methoden die mit @PostConstruct markiert sind, werden aufgerufen nachdem alle ausstehenden DI auf dem Objekt durchgelaufen sind
@PreDestroy	Methoden die mit @PreDestroy markiert sind, werden bei Zerstörung eines Objekts aufgerufen

Tabelle 2.1: Auswahl an von RCP 4 unterstützten Annotationen [9]

In Kapitel 2.3.1 wurde bereits der Einsatz von Annotationen im Zusammenhang von Events anhand eines Beispiels (siehe Listing 2.3) geschildert. In diesem Beispiel wurde ein Event mit dem Topic „date“ versendet. Um auf dieses Event reagieren zu können ist es erforderlich, eine Methode zu registrieren, die bei Empfangen des Events aufgerufen wird. Dies kann über ein IEventBroker Objekt erreicht werden oder über die Annotation einer Methode mit der @UIEventTopic("date") Annotation.

Die @Optional Annotation hat je nach Definierungsort unterschiedliches Verhalten und wird zusammen mit der @Inject Annotation verwendet. Sie kommt überall zum Einsatz, wo eine erfolgreiche Injektion von Seiten des Frameworks nicht immer garantiert werden kann. So wird bei einer nicht erfolgreichen Injektion bei

- Feldern die gewünschten Objekte nicht injiziert
- Methoden, der Aufruf übersprungen
- Parametern, der „null“ Wert übergeben

Dadurch lassen sich beispielsweise unnötige NullPointer Exceptions vermeiden.

Für das Management des Lebenszyklus sind Annotationen wie @PostConstruct und @PreDestroy zuständig, wie sich auch aus Tabelle 2.1 entnehmen lässt.

Die Eclipse Plattform beherbergt noch etliche weitere Annotationen mit verschiedensten Einsatzzwecken.

2.4 SPMF

SPMF ist eine Open Source Data Mining Software mit dem Fokus auf Pattern Mining [10]. Es sind 129 Data Mining Algorithmen aus verschiedenen Kategorien implementiert. Darunter befinden sich z.B. Itemset Mining und Sequential Pattern Mining. Die Algorithmen können über eine Benutzeroberfläche oder über die Kommandozeile ausgeführt werden.

Für die Ausführung der Algorithmen werden algorithmenspezifische Input-Dateien und Parameter benötigt. Ein solcher Parameter kann z.B. ein Minimum-Support-Wert sein. Nachdem der Benutzer die Parameter gesetzt und eine Output-Textdatei definiert hat, werden die Ergebnisse in diese geschrieben.

Durch die modulare Art der Software ist es möglich die Algorithmen ohne GUI in anderer Software zu verwenden und für eigene Zwecke anzupassen.

3 Verwandte Arbeiten

In diesem Abschnitt werden Arbeiten vorgestellt, die der Funktionalität des zu erstellenden Tools ähnlich sind. Die Arbeiten die in Kapitel 3.2 und 3.3 vorgestellt werden, stellen ein großes Fundament für das Tool dar, da das Tool in weiten Zügen auf diesen Arbeiten aufbaut.

3.1 Release History Database

Von Fischer et. al [11] wurde eine „Release History Database“ entwickelt, mit dessen Hilfe die Evolution einer Software analysiert werden kann. Das System kombiniert dabei die Informationen, die in CVS Version Control Systemen und im Bugzilla Bugtracker vorhanden sind. Um die Daten aus CVS und Bugzilla zu extrahieren, verwenden sie verschiedene Skripte.

Wie aus dem Namen des Systems abzuleiten ist, wird eine SQL Datenbank zur Speicherung der Daten verwendet. In Abbildung 3.1 ist ein UML-Diagramm der Datenbank zu sehen.

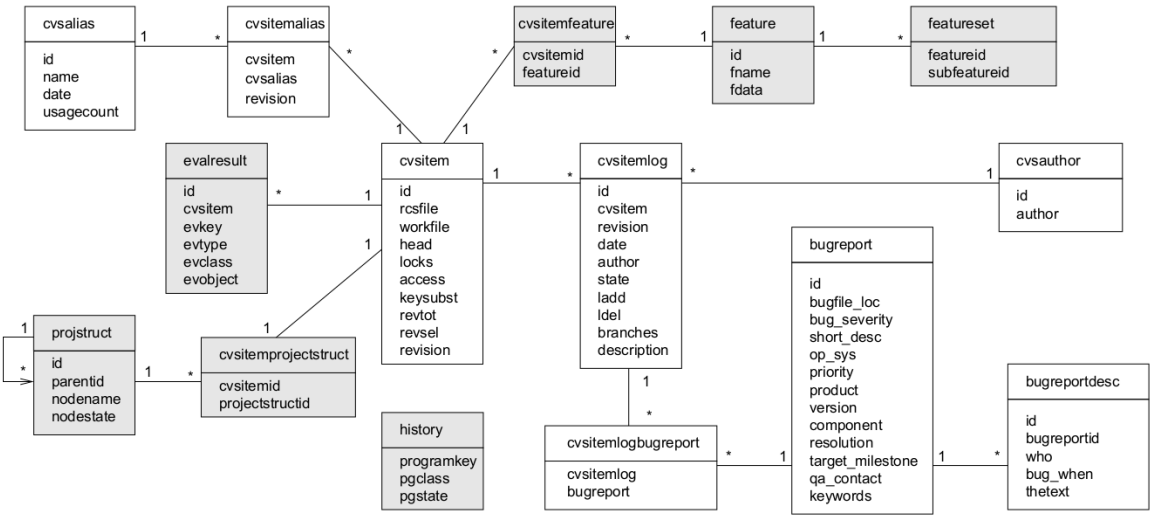

Abbildung 3.1: Release History Database [11]

In der Datenbank werden die CVS Informationen aller Dateien abgespeichert. Die

Informationen werden aus den CVS Log-Dateien gewonnen. Dabei besitzt jede Datei in CVS Log-Informationen. Ändert sich die Revision einer Datei, nachdem Veränderungen an ihr durchgeführt wurden, wird dies in der Log-Datei protokolliert. Zu den protokollierten Daten gehören z.B. Autor, der die Änderung durchgeführt hat, und das Datum, an dem die Änderung stattgefunden hat. Auch eine textuelle Beschreibung, welche die Änderungen kurz zusammenfasst, werden erfasst. Diese Daten werden jeweils in die *author*, *date* und *description* Felder der *cvsitemlog* Tabelle geschrieben.

Jeder Eintrag in *cvsitemlog* gehört zu einem Eintrag in *cvsitem*. Sie stellt die zentrale Einheit der Datenbank dar. Für jede Datei im Software-Repository wird ein Eintrag in *csvitem* erstellt, welches u.a. den Pfad zur Datei im Feld *rcsfile* abspeichert. Die Beziehung zwischen *cvsitem* und *cvsitemlog* ist dabei eine 1-zu-n-Beziehung, da jede Datei in CVS mehrere Revisionen und damit Log-Informationen besitzt.

Wie aus Abbildung 3.1 ersichtlich ist, sind die Informationen aus den Bugreports in Bugzilla ebenfalls in der Datenbank vertreten. Die Bugreports werden über die Tabelle *cvsitemlogbugreport* mit der *cvsitemlog* Tabelle verknüpft. Erfasst werden u.a. Bug ID, Bug Status und Bug Severity als Bugreportdaten. Damit die Datenbank gefüllt werden kann, müssen die notwendigen Informationen aus einem CVS Repository und zugehörigem Bugzilla ausgelesen werden.

Der Importprozess von CVS Daten und Bugreports ist in Abbildung 3.2 dargestellt.

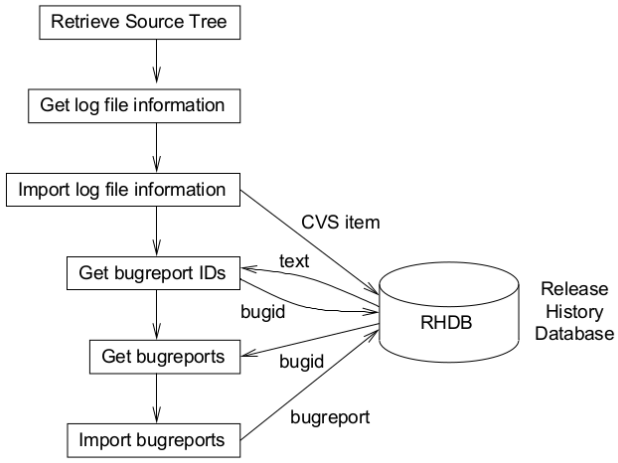

Abbildung 3.2: Database History Database Importprozess[11]

Damit die Daten in einem CVS Repository extrahiert werden können, muss das CVS Repository z.B. durch auschecken des Repositories heruntergeladen werden. Daraufhin können die Log-Informationen der Dateien im Repository mithilfe des CVS Programms extrahiert und in die Datenbank geschrieben werden. Der nächste Schritt ist Bugreports aus Bugzilla herunterzuladen und sie mit den zugehörigen *cvsitems* zu verknüpfen. Hierzu wird zunächst versucht, aus den Log-Informationen der Dateien die Bug ID ausfindig zu machen. Diese haben kein festes Format und das System versucht deshalb über einen regulären Ausdruck die Bug ID zu finden. Ist sie gefunden, wird sie in die Tabelle *cvsitemlogbugreport* geschrieben. Dieselbe Bug ID wird nun verwendet, um den Bugreport von Bugzilla herunterzuladen. Die so erhaltenen Bugreports werden daraufhin in die Datenbank geschrieben.

Mit der erstellten „Release History Database“ können Evulotionäre Aspekte einer Software, wie z.B. Systemwachstum oder Änderungsrate des Systems analysiert werden. Ebenso ist es möglich, logisch gekoppelte Dateien über die Release Historie ausfindig zu machen. Die gekoppelten Dateien hängen auch oft mit Bugreports zusammen und so lassen sich Gruppen von Bugreports finden, welche ein ähnliches Problem beschreiben. [11]

3.2 Automatische Transformation von Daten aus Software Repositories

Die Software ATSR [12] hat die Funktion, die in Software-Repositories gespeicherten Metadaten zu extrahieren und in einer Datenbank zu speichern. Die so gespeicherten Daten können von anderen Programmen eingelesen und weiterverarbeitet werden. Als Software-Repository kommen bei ATSR Git Repositories in Frage.

Das Tool bietet dem Benutzer eine Benutzeroberfläche, welche hauptsächlich in Java Swing entwickelt wurde. Zur Speicherung der Transformationsdaten in der Datenbank verwendet es als SQL-Backend das MySQL-Datenbanksystem.

Die GUI enthält Felder, über die der Benutzer die gewünschte Transformation durchführen kann. Darüber hinaus können Einstellungen bezüglich der Datenbank vorgenommen werden.

3.2.1 Input- und Outputformat

Metadata	Datenquelle	Format
Issue	CSV	Parameter 1;Parameter 2;…;Parameter 22
Docu	CSV	Parameter 1;Parameter 2;Parameter 3
Commit	Kommandozeilen-output	<Hash>#<Urheber>#<Uhrzeit+Datum>#<Commit-Titel> <Dateiliste>

Tabelle 3.1: Datenquellen und -format [13]

Als Input nutzt ATSR Issue- und Docudaten, welche jeweils im CSV Format vorliegen müssen. Die Commitdaten werden aus der Terminalausgabe des Git Kommandozeilenprogramms ausgelesen und transformiert. Die Commitausgabe umfasst dabei wichtige Daten wie den Commithash bzw. Commit ID, Committitel und eine Liste der Dateien, welche in dem Commit enthalten sind.

Die CSV Dateien werden zeilenweise geparst und die Daten werden Zeile für Zeile in ihre jeweiligen Datenbanktabellen geschrieben. Für Issuedaten ist es die „issuetable“, für Docudaten ist es die „docutable“.
Da die Commitdaten die umfangreichsten Daten beinhaltet, ist das Datenbankformat dementsprechend komplex. In Abbildung 3.3 ist das Outputformat für die Commitdaten dargestellt.

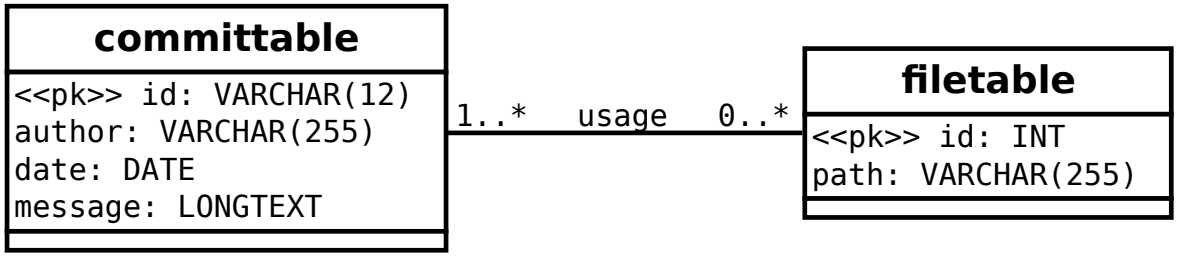

Abbildung 3.3: Tabellen für die Commit-Metadaten [13]

Die Commitdaten werden in drei Tabellen gespeichert: committable, usage, filetable. „Hash“, „Urheber“, „Datum“ und „Committitel“ aus Tabelle 3.1 werden in dieser Reihenfolge in die committable geschrieben. Die Dateien werden in der filetable gespeichert. Die filetable speichert alle Dateien, die in allen Commits vorkommen. Keine Datei kommt doppelt vor. Die Commits werden mit der filetable über die usage Tabelle verbunden. Sie verknüpft die Commits und die in ihnen enthaltenen Dateien.

3.3 SRM Plugin

SRMP [14][15] ist ein Tool, welches entwickelt wurde, um Data Mining auf Basis der Metadaten eines Software Repositories zu betreiben und auf Basis dieses Data Minings Coupled Changes zu identifizieren. Die Metadaten liegen in Datenbanken im Format der ATSR Software vor. Genauer sind es die Commit Metadaten, welche für das Data Mining verwendet werden. Als Data Mining Algorithmus kommt FPGrowth zum Einsatz, was zum Ziel hat, die Frequent Itemsets aus der Commitdatenbank zu extrahieren und diese wiederum in einer Datenbank zu speichern, um weitere Zugriffe zu beschleunigen. Diese Frequent Itemsets werden benutzt um Coupled Changes zu identifizieren. Der Benutzer legt die Identifikation in Gang, indem er eine Datei in der GUI auswählt und das SRMP startet daraufhin eine Suche in der für die Frequent Itemsets angelegten Datenbank. Es werden alle Frequent Itemsets ausgelesen, die die ausgewählte Datei beinhalten und alle Dateien dieser Frequent Itemsets werden in der GUI gelistet. Diese Menge an Dateien definiert sich als Coupled Changes, denn es sind die Dateien, welche zusammen mit der ausgewählten Datei geändert wurden.

In der aktuellsten Fassung des SRMP [13] wurden die ATSR Funktionen in das Plugin in Form eines Wizards integriert. Dem Benutzer wurde es damit ermöglicht, direkt im Plugin die notwendigen Datenbanktabellen für das Data Mining zu generieren. Der Minimum-Support-Wert kann nun über den Wizard eingestellt werden. Falls gewünscht, kann die Größe der Input-Daten für das Data Mining über die Auswahl eines Committers eingeschränkt werden. Hierfür werden die Committer mit den meisten Commits im Wizard aufgelistet. Durch Auswahl eines Committers werden lediglich die Commits dieses Committers für die Frequent Itemset Analyse in Betracht gezogen. Das MySQL-Backend wurde für eine besser integrierte, eingebettete Datenbank ausgetauscht. Der Benutzer des Plugins muss sich dank dieser Änderung nicht auch noch um das Aufsetzen eines SQL Servers kümmern, womit die Installation des Plugins erleichtert wurde.

4 Anforderungen und Analyse

In diesem Kapitel wird erläutert, was die Anforderungen am Tool sind und es wird analysiert, wie diese Anforderungen durchgesetzt werden können.

4.1 Anforderungen an das Tool

Das Tool hat folgende Anforderungen zu erfüllen:

1. **Eclipse basierend:** Eine Eclipse basierende Anwendung erleichtert das Design einer GUI und vereinfacht die Wiederverwendung von Komponenten aus anderen Eclipse Produkten oder Nutzung von vorhanden GUI Komponenten entwickelt für die Eclipse Plattform. Ein weiterer Bonus ist, dass das Tool ohne viel Aufwand auf anderen Plattformen benutzt werden kann.

2. **Import von Issuedaten:** Für die Arbeit des Tools sind Issuedaten eine Voraussetzung. Einlesen von Issuedaten im CSV Format muss möglich sein.

3. **Import von Git Logs in eine Datenbank:** Commit-Daten gehören zu den wichtigsten Daten und erfordern eine persistente Speicherung in einer Datenbank. Über Anwendungsneustarts können so dieselben Daten ohne Neuberechnung wiederverwendet werden.

4. **Nutzung von Frequent Itemsets Mining Algorithmus FPGrowth:** Data Mining Algorithmus benötigt für die Generierung von Coupled Changes.

5. **Nutzung von Sequence Pattern Mining Algorithmus Prefixspan:** Alternative Data Mining Methode für die Generierung von Coupled Changes.

6. **Anzeigen von Issues:** Die Issues sind im Tool passend aufzulisten.

7. **Anzeigen von Coupled Changes in Relation zu Issues und zugehörige Attribute:** Die generierten Coupled Changes sollen im Tool mit ihren Attributen wie beispielsweise Commit ID, Commit message und Autor angezeigt werden.

8. **Export der generierten Coupled Changes:** Es soll möglich sein, generierte Coupled Changes über die Benutzeroberfläche zu exportieren.

9. **Coupled Changes nach Zeit, Committer und anderen Commit Attributen filtrieren.**

4.2 Analyse

Werden die Anforderungen analysiert, lassen sich gewisse Parallelitäten zwischen dem Tool und SRMP aufzeigen, welche bei der Entwicklung des Tools ausgenutzt werden können. Anforderungen zwei, drei und vier sind in ähnlicher Form bereits in SRMP integriert und können mit minimalen Änderungen wiederverwendet werden. Die Nutzung von Prefixspan im Tool (Punkt 5) lässt sich in das bestehende Framework SRMPs integrieren, da mit FPGrowth bereits der Grundpfeiler für weitere Data Mining Algorithmen gelegt wurde. SRMP hat auch Funktionen integriert, welche zum Anzeigen von Issues und Coupled Changes verwendet werden. Diese sind jedoch SRMP spezifisch und deshalb nicht auf das Tool anwendbar.

5 Konzept und Architektur

In diesem Kapitel wird neben dem Lösungsansatz zur Erzeugung von Coupled Changes für Issue Tasks auch das Workflow und die Architektur des Tools beschrieben.

5.1 Lösungsansatz zum Erzeugen von Coupled Changes für Issue Tasks

Der Ansatz, der von der Release History Database (siehe Kapitel 3) genutzt wird, um Bugreport IDs aus den CVS Log-Dateien zu extrahieren, kann auch für das Finden von Issue IDs in Git-Logs genutzt werden. Extrahiere hierzu erst alle Commit IDs, die eine Referenz zu einer bestimmten Issue besitzen. Die erstellte Liste kann dann zur Erzeugung von Coupled Changes für die Issue genutzt werden. Dieser Prozess wird in Kapitel 6.3 ausführlich beschrieben.

5.2 Workflow

In Abbildung 5.1 ist der Workflow des Tools abgebildet.

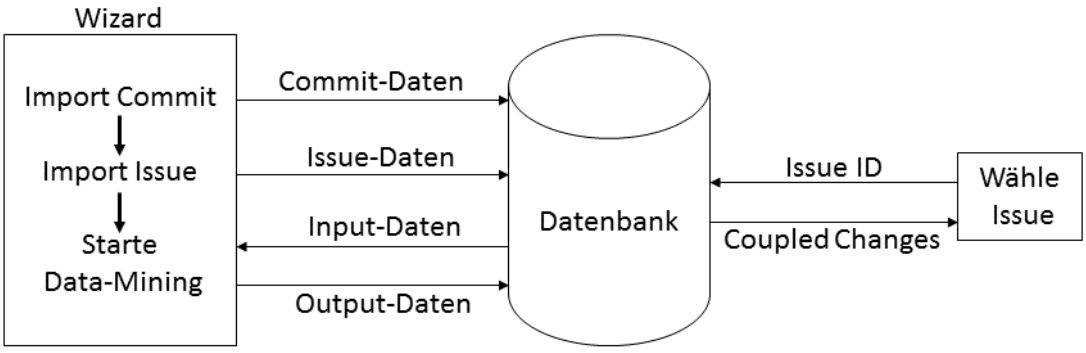

Abbildung 5.1: Workflow des Tools

Über den Wizard werden die Commits und danach die Issues importiert. Die Daten werden in die Datenbank geschrieben. Daraufhin wird das Data Mining mit

den Input-Daten, die aus der Datenbank ausgelesen werden, gestartet. Die Output-Daten des Mining-Prozesses werden zurück in die Datenbank geschrieben.

Durch Auswahl einer Issue wird dessen ID in die Datenbank übergeben und es werden Coupled Changes mit Referenz zur ID ausgegeben.

5.3 Framework

Wie bereits in Abschnitt 4.2 erwähnt, enthält SRMP einen Unterbau für die Verwendung von FPGrowth, welches sich eignet, um weitere Data Mining Algorithmen, wie Prefixspan, zu integrieren. Prefixspan lässt sich durch Nutzung dieses Unterbaus in das Framework integrieren. Das resultierende Framework ist in Abbildung 5.2 abgebildet.

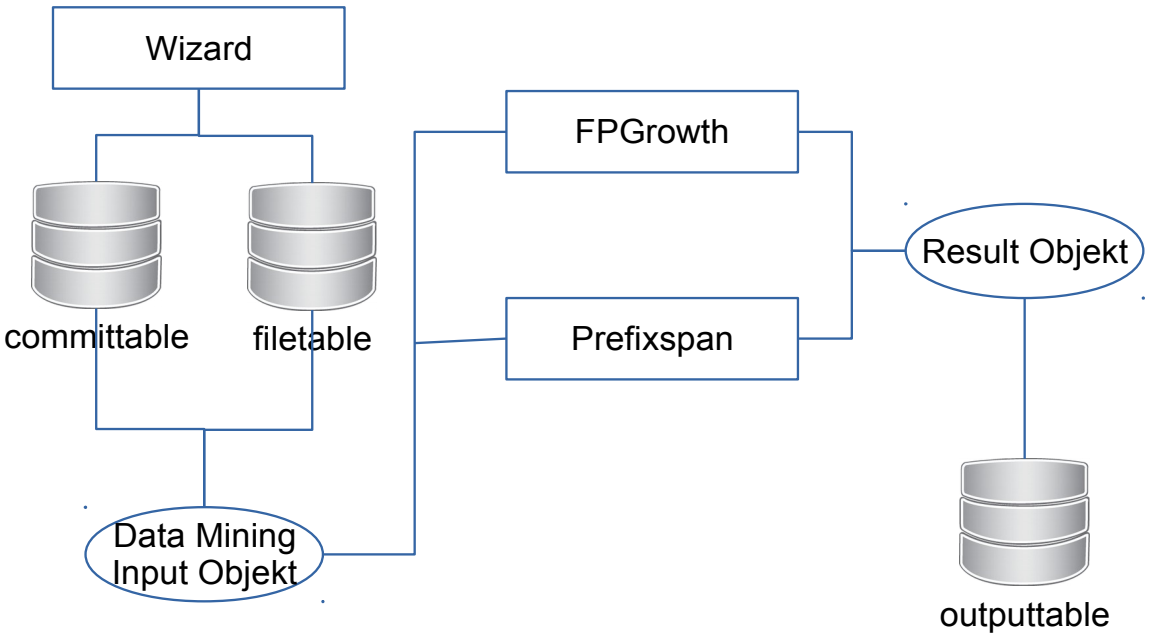

Abbildung 5.2: Framework für das Data-Mining

Über das Framework kann das Tool die Metadaten für die Commits, welche in einer Datenbank abgespeichert sind, auslesen und weiterverwenden. Die dabei verwendeten Tabellen sind die committable und filetable (vgl. Abbildung 3.3). Damit der Data Mining Algorithmus die Daten verwenden kann, werden diese aus den Tabellen ausgelesen und in ein *Data Mining Input Objekt* (DMIO) umgeformt (siehe Kapitel 5.4.1). Als Output wird ein *Data Mining Result Objekt*

(DMRO) von den Algorithmen erzeugt. Dieses Objekt wird im letzten Schritt in eine Output-Tabelle geschrieben. Durch diese Tabelle sind die Ergebnisse des Data Minings für die Berechnung von Coupled Changes zugänglich.

5.4 Input und Output Objekte für das Data Mining

Sowohl das *Data Mining Input Objekt* als auch das *Data Mining Result Objekt* sind Objekte, welche für beide Algorithmen ein identisches Format haben. Dies erleichtert die Integration von Prefixspan in das bestehende Framework, da keine algorithmenspezifische Funktionen und Pfade implementiert werden müssen.

5.4.1 Data Mining Input Objekt

Das DMIO speichert alle relevanten Informationen, die für das Arbeiten der Data Mining Algorithmen notwendig sind. Die Informationen werden vom Wizard (siehe Kapitel 3.3) in Form der committable und filetable bereitgestellt (siehe Abbildung 3.3).

Commit ID i, $Item_i$ 1, …., $Item_i$ l
Commit ID j, $Item_j$ 1, …., $Item_j$ m
Commit ID k, $Item_k$ 1, …., $Item_k$ n
….

Tabelle 5.1: Format eines Data Mining Input Objekts

Die aus den Datenbanktabellen extrahierten Daten sind eine Liste aller Commits und die ihnen zugehörigen, in den Commits geänderten Dateien (siehe Tabelle 5.1). Die Data Mining Algorithmen nutzen nun dieses Objekt als Transaktionsdatenbank im Fall FPGrowth und als Sequenzdatenbank im Falle Prefixspans.

5.4.2 Data Mining Result Objekt

Die Resultate der Data Mining Algorithmen müssen in einer Form vorliegen, welche die Berechnung von Coupled Changes vereinfacht und eine persistente Speicherung möglich macht.

Das DMRO speichert hierzu die Ergebnisse der Data Mining Algorithmen als

Vorbereitung für die persistente Speicherung in einer Datenbank ab. Die Algorithmen geben die Ergebnisse als Liste zurück, deren Format in Tabelle 5.2 zu sehen ist.

$Support_i$, $Length_i$, Commit ID_i 1 , .. , Commit ID_i l_1 , $Item_i$ 1 , … $Item_i$ l_2
$Support_j$, $Length_j$, Commit ID_j 1 , .. , Commit ID_j m_1 , $Item_j$ 1 , … $Item_j$ m_2
$Support_k$, $Length_k$, Commit ID_k 1 , .. , Commit ID_k n_1 , $Item_k$ 1 , … $Item_k$ n_2
….

Tabelle 5.2: Format eines Data Mining Result Objekts

Es ist zu erkennen, dass ein jeder Eintrag in der Liste mit Support und Length beginnt, welche beide Zahlen sind. Die erste Zahl stellt den Support dieses Eintrags dar. Die Interpretation dieser Zahl ist sowohl bei FPGrowth als auch bei Prefixspan ähnlich. In FPGrowth bedeutet die Zahl, in wievielen Transaktionen die gefundenen Frequent Pattern vorkommen. In Prefixspan bedeutet sie, in wievielen Sequenzen die gefundenen Pattern vorkommen. Die zweite Zahl beschreibt jeweils die Anzahl der gefundenen Pattern. Auf diese beiden Zahlen folgt eine Liste von Commit IDs, der Länge von der ersten Zahl Support. Die Commit IDs entsprechen der Transaktions ID bzw. Sequenz ID von FPGrowth und Prefixspan. Sie bekräftigen die Zugehörigkeit der Patterns zu den entsprechenden Commits. Ist diese Relation nicht vorhanden, ist eine Berechnung der Coupled Changes nicht möglich. Dies ist der Grund, weshalb die Liste der Commit IDs zwingend für jeden Eintrag in der Ergebnismenge vorhanden sein muss. Auf die Liste der Commit IDs folgt die Liste der gefundenen Pattern. Diese Liste hat die Länge der zweiten Zahl, also der Anzahl der gefundenen Pattern. Die ersten zwei Zahlen dienen somit als Index und beschreiben die Position und Länge von Commit IDs und Patterns.

Weitere wichtige Variablen, die von dem DMRO bereitgestellt werden, sind maxsupport und maxlength. Sie definieren, wie aus dem Namen abgeleitet werden kann, den größten Support und Länge, welche in der Ergebnismenge gefunden werden konnte. Sie werden für die Erzeugung der Output-Tabelle benötigt.

5.5 Format der Output-Tabelle

Das DMRO, welches durch die Anwendung der Data Mining Algorithmen auf das DMIO entsteht, wird zur persistenten Speicherung in eine Datenbank geschrieben. Die entstehende Output-Tabelle beinhaltet die vollständigen Informationen des DMRO.

Da die Einträge im DMRO jeweils eine variable Anzahl von Commit IDs und Patterns besitzen können, muss die Output-Tabelle in Anbetracht dieser Variabilität erstellt werden. Hier kommen die maxsupport und maxlength Variablen des DMRO zum Einsatz (siehe Abschnitt 5.4.2). Der maxsupport Wert bestimmt die Anzahl der Commit ID Spalten, wohingegen der Wert von maxlength die Anzahl der Spalten für die gefundenen Items, also den Dateien, bestimmt. Dadurch können die Einträge im DMRO in die Output-Tabelle abgebildet werden.

5.5.1 Schreiben des Output Objekts in die Output-Tabelle

Jeder Datensatz der Output-Tabelle entspricht einem Eintrag im DMRO. Es werden zuerst der Support und Length Wert in die Tabelle geschrieben. Darauf folgen Commit IDs und Items.

Wie bereits beschrieben, enthält nicht jeder Eintrag die volle Anzahl von Commit IDs und Items, deshalb werden die Commit IDs nach der Reihe in die Commit ID Spalten der Output-Tabelle geschrieben. Falls die Anzahl der Commit IDs geringer ist als maxsupport, werden die restlichen Commit ID Spalten der Tabelle mit null Werten beschrieben, um kenntlich zu machen, dass diese Spalten leer sind und nicht verwendet werden sollen. Dasselbe Vorgehen wird analog für die Items angewandt, nur werden sie in die Item Spalten geschrieben und nicht in die Commit ID Spalten.

Nachdem jeder Eintrag im DMRO in einem Datensatz der Output-Tabelle gespeichert ist, ist das Schreiben in die Tabelle beendet.

5.6 Prefixspan in SPMF

Als Implementation für Sequential Pattern Mining wurde eine Implementation

von Prefixspan aus dem SPMF Framework (siehe Kapitel 2.4) ausgewählt. In SPMF sind verschiedene Implementationen des Algorithmus integriert. Die erste basiert auf einem Input aus Zahlen und die zweite auf einem Input von Strings. Im Folgenden soll diskutiert werden, wie der Input für Prefixspan (SPMF) aufgebaut ist und auf Basis dieser Erkenntnisse, die Auswahl der Implementation begründet werden.

5.6.1 Input von Prefixspan

Wie bereits in Kapitel 2.4 beschrieben, benutzt das SPMF Framework als Input für die implementierten Daten herkömmliche Textdateien. Das Format und der Inhalt der Textdateien sind je nach Algorithmus unterschiedlich. Für Prefixspan gilt das in Tabelle 5.3 definierte Format.

<Itemset 1> -1 <Itemset 2> -1 . . . . . <Itemset i-1> -1 <Itemset i> -1 -2
<Itemset 1> -1 <Itemset 2> -1 . . . . . <Itemset j-1> -1 <Itemset j> -1 -2
<Itemset 1> -1 <Itemset 2> -1 . . . . . <Itemset k-1> -1 <Itemset k> -1 -2
….

Tabelle 5.3: Prefixspan Input-Textdateiformat

Die Textdatei besteht aus einer Sequenzdatenbank mit je einer Sequenz pro Zeile. Jede Sequenz besteht aus einem oder mehreren Itemsets. Itemsets wiederum bestehen aus mindestens einem Item. Die Items werden mit einem Leerzeichen voneinander getrennt. Bei Prefixspan_int bestehen Items ausschließlich aus positiven Zahlen, wohingegen bei Prefixspan_string die Items aus einer Zeichenkette bestehen. Jedoch gilt bei beiden, dass Items in Itemsets mit Leerzeichen voneinander getrennt werden. Würde man Itemsets auch mit einem Leerzeichen voneinander trennen, gäbe es keine Möglichkeit die verschiedenen Itemsets voneinander zu unterscheiden. Deshalb werden sie mit einer Zahl getrennt, die nicht in Itemsets vorkommen kann oder vorkommen sollte. Da Itemsets in Prefixspan_int aus positiven Zahlen bestehen, wird zur Trennung eine negative Zahl, „-1“, verwendet. Damit der Algorithmus weiß, wann eine Sequenz beendet ist, erwartet es eine weitere negative Zahl, „-2“.

Die Trennzeichen für Itemsets („-1“) und Sequenzen („-2“) gelten sowohl für Prefixspan_int als auch Prefixspan_string.

5.6.2 Auswahl der Implementierung

Da Prefixspan als Input mit Dateipfaden arbeiten muss und diese als Zeichenketten vorliegen, ist eine Verwendung von Prefixspan basierend auf Items mit Strings naheliegend.

Bei der Verwendung von Prefixspan_int müssten alle Dateipfade, die dem Algorithmus als Input übergeben werden, auf positive Zahlen abgebildet werden. Ist das Data Mining durchgelaufen, müssen diese Dateien wieder von Zahlen zurück in Strings umgewandelt werden, damit die Daten im Rest des Tools weiterverwendet werden können. Dies stellt einen unnötigen Mehraufwand dar, der mit Strings wegfällt. Außerdem stellt die Translation von Strings in Integers und wieder zurück eine weitere potenzielle Fehlerquelle dar.

6 Implementierung

In diesem Kapitel wird die Implementierung des Tools unter Berücksichtigung des angestrebten Konzepts und der Architektur beschrieben.

6.1 CommitTableData

Die CommitTableData wird mit dem Lesen der Input-Datenbank erzeugt und bildet das Format der Input-Datenbank in einer Instanzvariable *res* ab (siehe Tabelle 5.1). Zusätzlich besitzt sie ein Feld (*list_commitID*) zum seperaten abspeichern einer Commit ID Liste für den Data Mining Algorithmus Prefixspan. Eine weitere Funktion dieser Klasse ist, das Objekt *res* für die Verwendung als Sequenzdatenbank vorzubereiten (siehe 6.2.1).

6.2 Integrierung von Prefixspan in das Framework

Da Prefixspan ein bestimmtes Input-Format besitzt (vgl. Kapitel 5.6.1), müssen die Input-Daten vor der Übergabe an Prefixspan transformiert werden und Prefixspan bringt die Daten dann in sein internes Format. Die gefundenen Pattern müssen ebenfalls transformiert werden, um die Ergebnisse im Rest des Frameworks nutzen zu können.

6.2.1 Transformation der Input-Daten für Prefixspan

Die DMIO Daten des CommitTableData Objekts, können im Gegensatz zu FPGrowth nicht direkt verwendet werden. Wie in Kapitel 5.4.1 beschrieben, besteht ein Eintrag im DMIO aus einer Commit ID und den im Commit geänderten Dateien. FPGrowth nimmt die Daten des CommitTableData Objekts ohne Veränderung an. Bei Prefixspan ist dies nicht ohne weiteres möglich, da Prefixspan das in Tabelle 5.3 beschriebene Inputformat erwartet.

Um die Daten in das gewünschte Format zu bringen ist es zunächst nur nötig eine neue Liste zu generieren, bei dem jeweils alle Commit IDs fehlen. Die zuständige

Methode der CommitTableData *get_fileIDs()* ist in Listing 6.1 abgebildet.

```
1  public List<List<String>> get_fileIDs() {
2        List<List<String>> ret = new ArrayList<>(res.size());
3        list_commitID = new ArrayList<>();
4        for(List<String> it :  res) {
5              ArrayList<String> t = new ArrayList<>();
6              for(int i=0; i<it.size(); i++) {
7                    if(i==0)
8                          list_commitID.add(it.get(0));
9                    else
10                         t.add(it.get(i));
11             }
12             ret.add(t);
13       }
14       return ret;
15 }
```

Listing 6.1: Transformation der Input-Daten für Prefixspan

Die Variable *ret* speichert die transformierten Input-Daten, welche wie oben beschrieben umgewandelt werden. In Zeile 8 werden die zu einem Commit gehörende Commit ID in einer seperaten Instanzvariable *list_commitID* gespeichert. Die restlichen Daten werden in Zeile 12 der Variablen *ret* übergeben. Somit sind in *ret* nur noch Listen aus Dateipfaden gespeichert. Die Instanzvariable *list_commitID* ist von besonderer Wichtigkeit, denn sie ist erforderlich um den berechneten Sequential Patterns die zugehörigen Commit IDs hinzuzufügen. Dies ist erforderlich, damit sich Prefixspan in das in Kapitel 5.3 beschriebene Framework integrieren kann.

6.2.2 Laden von transformierten Input-Daten

In Kapitel 5.6.1 wurde das Input-Format für Prefixspan und in Kapitel 6.2.1 die Transformation der Input-Daten für Prefixspan erläutert. In der ursprünglichen SPMF Implementation unterstützt Prefixspan lediglich das Einlesen von Textdateien. Die Input-Daten liegen jedoch als DMIO vor, welche nicht von Prefixspan verstanden werden. Deshalb wurde Prefixspan um eine Methode erweitert, welche die transformierten Input-Daten laden und diese in das Prefixspan Input-Format überführen kann.

Die SequenceDatabase Klasse ist unter anderem für das Laden von Textdateien zuständig. Sie wurde um die Methode *load_files()* erweitert, welche die transformierten Input-Daten akzeptiert (siehe Listing 6.2).

```
1 public void load_files(List<List<String>> files) {
2       for(List<String> lfiles : files) {
3             String[] tokens = format_files(lfiles);
4             addSequence(tokens);
5       }
6 }
```

Listing 6.2: Laden und Umwandeln der Prefixspan Input-Daten

Für jede Liste aus Dateipfaden (Zeilen 2-5) wird eine Methode *format_files()* aufgerufen. Die *format_files()* Methode überführt die in *lfiles* beinhalteten Strings in das von Prefixspan erwartete Format. Nach jedem Item/ Dateipfad wird ein „-1“ String hinzugefügt um das Itemset als beendet zu markieren. Am Schluss wird zusätzlich noch eine „-2“ angehängt, um das Ende der Sequenz zu markieren. Nachdem *format_files()* die Überführung beendet hat, gibt er die umgewandelte Sequenz als String Array *tokens* zurück. Die Sequenz wird zum Schluss an die SequenceDatabase hinzugefügt (Zeile 4).

Ist die Methode *load_files()* durchgelaufen, sind die Input-Daten in der Sequenzdatenbank von Prefixspan im benötigten Format vorliegend.

6.2.3 Transformation der gefundenen Sequential Patterns

Nachdem Prefixspan über die *runAlgorithm()* Methode ausgeführt wurde, werden die gefundenen Sequential Patterns von der Methode als SequentialPatterns Objekt zurückgegeben.

Für die Berechnung der Coupled Changes ist es erforderlich zu Wissen, aus welcher Sequenz der Sequenzdatenbank die gefundenen Sequential Patterns stammen. Prefixspan etikettiert jede Sequenz in einer Sequenzdatenbank mit einer eindeutigen internen Nummer. Angefangen bei Null, welches der ersten Sequenz entspricht, wird so jede Sequenz durchnummeriert. Diese Nummer wird auch Sequenz ID genannt. Sie ist vergleichbar mit der Transaktions ID FPGrowths.

Für jedes gefundene Pattern wird gespeichert in welchen Sequenzen es jeweils vorkommt. Somit besitzt jedes Pattern eine Liste aus Sequenz IDs über die der Ursprung des Patterns in der Sequenzdatenbank ersichtlich ist. Da jedoch für die Berechnung der Coupled Changes ein Objekt benötigt wird, welches dem Format eines DMRO entspricht, müssen die Patterns angepasst werden.

In Kapitel 6.2.1 wurde beschrieben, welche Form die Input-Daten für Prefixspan haben. Bevor sie für Prefixspan transformiert werden, besteht jede Liste in den Input-Daten aus einer Commit ID und Dateipfaden. Nach der Transformation fehlt jeder Liste die Commit ID. Die Listen stellen Sequenzen der Sequenzdatenbank dar und werden wie beschrieben durchnummeriert. Wie aus Listing 6.1 ersichtlich ist, werden die Commit IDs der Sequenzen in die ArrayList *list_commitID* kopiert. Somit herrscht eine Eins-zu-eins-Beziehung zwischen Position der Commit ID in *list_commitID* und der Sequenz ID der Sequenz. Dieser Umstand wird ausgenutzt, um aus der Sequenz ID die zugehörige Commit ID zurückzugewinnen.

```
1  public List<List<String>> getPatterns(){
2          List<List<String>> patterns = new ArrayList<>();
3          maximum_support = 0;
4          maximum_item_length = 0;
5          for(List<SequentialPattern> level : levels) {
6                  for(SequentialPattern sequence : level) {
7                          List<String> subp = new LinkedList<>();
8                          subp.add(Integer.toString(sequence.getAbsoluteSupport()));
9                          subp.add(Integer.toString(sequence.size()));
10                         Set<Integer> sequenceIDs = sequence.getSequencesID();
11                         List<Itemset> itemset = sequence.getItemsets();
12                         List<String> items_string = new ArrayList<>(itemset.size());
13                         itemset.forEach(set -> set.getItems().forEach(item ->
14                                 items_string.add(item)));
15                         List<String> sequenceIDs_string =
16                                 new ArrayList<>(sequenceIDs.size());
17                         if(mapSeqIDtoCommitID != null)
18                                 for(int i : sequenceIDs) {
19                                         sequenceIDs_string
20                                                 .add(mapSeqIDtoCommitID.get(i));
21                                 }
22                         subp.addAll(sequenceIDs_string);
23                         subp.addAll(items_string);
24                         if(maximum_support < sequence.getAbsoluteSupport())
25                                 maximum_support =
26                                         sequence.getAbsoluteSupport();
27                         if(maximum_item_length < itemset.size())
28                                 maximum_item_length = itemset.size();
29                         patterns.add(subp);
30                 }
31         }
32         return patterns;
33 }
```

Listing 6.3: Transformation der Sequential Patterns

In Listing 6.3 ist der Quellcode abgebildet der aus den gefundenen Sequential Patterns des SequentialPatterns Objekts ein DMRO generiert. Es wird über jedes Pattern iteriert und für diese ein Eintrag für das DMRO zusammengebaut. Hierfür wird der Support-Wert und die Anzahl der Items im Pattern in der Liste *subp* gespeichert. Daraufhin müssen laut dem DMRO-Format die Commit IDs und die Items folgen.

Wie beschrieben sind die Commit IDs als Zahl codiert. Die Codierung wird in den Zeilen 18-21 mit Hilfe der Instanzvariable *mapSeqIDtoCommitID* rückgangig gemacht. Die Instanzvariable entspricht der *list_commitID* Liste. Durch Ausnutzung der Eins-zu-eins-Beziehung wird aus den Sequenz IDs zugehörige Commit IDs gewonnen und der Liste *subp* hinzugefügt (Zeile 22).

Die Items werden aus den Itemsets über den Ausdruck in Zeile 13-14 gewonnen. Es wird über alle Itemsets iteriert und die Items der Reihe nach in die Liste *items_string* kopiert. Die items in *items_string* werden nach den Commit IDs in die Liste *subp* eingefügt. Dadurch ist ein vollständiger Eintrag für das DMRO zusammengebaut, welcher dann in die *patterns* Liste eingefügt wird. Ist die Iteration beendet enthält *patterns* das fertige DMRO und wird mit Beendigung der Methode *getPatterns()* zurückgegeben.

Neben der Erzeugung der Einträge wird während der Iteration der maximale Support (Zeile 24-26) und die maximale Item-Anzahl (Zeile 27-28) in den gefundenen Sequential Patterns berechnet. Diese werden für die Erstellung und das Beschreiben der Output-Datenbank benötigt (siehe Kapitel 6.7.2).

6.2.4 Konstruktor und Ausführung des Algorithmus

Wie in Kapitel 6.2.3 beschrieben, wird eine Liste der Commit IDs benötigt um die Codierung der Sequenz IDs rückgangig zu machen. Die Liste wird bei Erzeugung eines Prefixspan Objekts dem Konstruktor als Parameter übergeben und in der Instanzvariable *mapSeqIDtoCommitID* gespeichert (Listing 6.4, Zeile 2).

Der Algorithmus wird über die *runAlgorithm()* Methode gestartet. Als Parameter erhält sie transformierte Input-Daten (Variable *files*) und ein Minimum-Support-Wert. Die SequenceDatabase wiederum transformiert *files* in das Prefixspan Format und der Algorithmus wird mit der SequenceDatabase ausgeführt (Zeile 10).

```
1  public AlgoPrefixSpan_with_Strings(ArrayList<String> mapSeqIDtoCommitID) {
2        this.mapSeqIDtoCommitID = mapSeqIDtoCommitID;
3  }
4
5  public SequentialPatterns runAlgorithm(List<List<String>> files,
6  double minsupRelative) throws IOException {
7        SequenceDatabase sequenceDatabase = new SequenceDatabase();
8        sequenceDatabase.load_files(files);
9        ...
10       prefixSpan(sequenceDatabase, null);
11       ...
12       return patterns;
13 }
```

Listing 6.4: Prefixspan – Konstruktor und runAlgorithm()

6.3 Generierung von Coupled Changes für Issue Tasks

Im SRMP existiert eine Implementation zur Generierung von Coupled Changes (siehe Kapitel 3.3). Die Coupled Changes werden jedoch über Quellcodepfade im Eclipse Package Explorer generiert und derselbe Prozess ist deshalb nicht für das Tool geeignet. Die Generierung der Coupled Changes von Issue Tasks besteht aus einer Folge von Schritten, die aufeinander aufbauen. In diesem Abschnitt werden diese Schritte und ihre Implementation erläutert.

6.3.1 Filterung von Commits die im Zusammenhang mit Issues stehen

Da eine Verbindung zwischen Issues und Commits hergestellt werden muss, um Coupled Changes zu generieren, ist der erste Schritt folglich das Herausfiltern der Commits, welche im Zusammenhang mit Issues stehen. Die Commitdaten sind in der Datenbank abgespeichert und müssen auch von dort abgefragt werden.

Ein Commit hängt mit einem Issue zusammen, falls sich in der Commitmessage ein Hinweis auf eine Issue finden lässt. Ein solcher Hinweis ist meistens die Issue ID der Issue. Die Commitmessage lässt sich in der Datenbanktabelle *committable* finden und über eine SQL-Abfrage lassen sich die Commits aus der Tabelle herausfiltern. Die SQL-Abfrage ist in Listing 6.5 dargestellt.

```
1  SELECT id
2  FROM committable
3  WHERE message LIKE '%#<Issue ID>%'
   OR message LIKE '%refs <Issue ID>%'
```

Listing 6.5: Auslesen von Commits die in Relation zu Issues stehen

Über die SQL-Abfrage werden aus der *committable* alle Commit IDs selektiert, welche in ihrem *message* Feld eine Issue ID referenzieren. Hierzu wird der LIKE Operator verwendet mit dessen Hilfe sich Muster in Feldern finden lassen. Kommt eine Issue ID in der Form „#<Issue ID>“, oder „refs <Issue ID>“ im *message* Feld der Tabelle vor, dann referenziert der Commit eine Issue. Das in der Where-Klausel vorkommende „<Issue ID>“ stellt offensichtlich eine Zahl dar, die eine Issue referenziert.

6.3.2 Bestimmung der Commitspaltenanzahl

Die Bestimmung der Commitspaltenanzahl ist erforderlich, da eine SQL-Abfrage über alle CommitID Spalten der Output-Tabelle für das weitere Vorgehen nötig sind. Hierfür wird zuerst lediglich eine Zeile mit allen Feldern aus der Output-Tabelle ausgelesen. Dies wird über die SQL-Abfrage „SELECT * FROM outputtable LIMIT 1“ bewerkstelligt. Das zurückgegebene ResultSet wird an die *buildWhereClause()* (siehe Listing 6.6) Methode der DBConnection Klasse übergeben. Diese Methode baut aus dem ResultSet, welches wie beschrieben aus lediglich einem Record besteht.

```
1  private String buildWhereClause(String val, int[] icount, ResultSet
2  result) throws SQLException {
3        ResultSetMetaData metaData = result.getMetaData();
4        ...
5        int count = 0;
6        for (int i = 4; i <= metaData.getColumnCount(); i++) {
7              if (metaData.getColumnName(i).contains(val))
8                    count++;
9        }
10       ...
11       for (int i = 1; i < count; i++)
12             whereclause.append(val + i + " LIKE ? OR ");
13       whereclause.append(val + count + " LIKE ?");
14       ...
15       return whereclause.toString();}
```

Listing 6.6: Aufbau der WHERE-Klausel zur Auslesung von Commit IDs

Der Parameter *val* entspricht „CommitID“. Aus dem ResultSet werden nach CommitID Spalten gesucht und die Anzahl gezählt (Listing 6.6). Mit diesen Informationen kann die WHERE-Klausel gebaut werden. Hierfür sind die Zeilen 11-13 zuständig. Es entsteht eine WHERE-Klausel die das Format „CommitID1 LIKE ? OR … CommitIDN LIKE ?“, je nachdem wie viele Commit ID Spalten in

der Output-Tabelle existieren.

6.3.3 Auslesen der Data Mining Resultate aus der Datenbank

Als Vorschritt zur Generierung von Coupled Changes, müssen nun die Ergebnisse des Data Mining Prozesses aus der Datenbank ausgelesen werden, in die sie geschrieben wurden (vgl. Kapitel 5.5 & 5.5.1). Da das DMRO (vgl. Kapitel 5.4.2) in die Output-Tabelle geschrieben wird, müssen die Daten auch wieder von dort ausgelesen werden.

Im vorigen Kapitel(6.3.2) wurde die WHERE-Klausel für die benötigte Abfrage gebaut. Der WHERE-Klausel fehlen jedoch noch die notwendigen Parameter für die SQL-Abfrage. Eine Beispielhafte SQL-Abfrage ist in Listing 6.7 abgebildet. Die Parameter ersetzen die Fragezeichen in der Abfrage.

```
1 SELECT *
2 FROM outputtable
3 WHERE CommitID1 LIKE ?
4 OR CommitID2 LIKE ?
5 OR CommitID3 LIKE ?
```

Listing 6.7: Beispiel SQL-Abrage zum Auslesen der Output-Tabelle

In diesem Beispiel hat die Output-Tabelle lediglich drei Commit ID Spalten, welche von eins bis drei durchnummeriert sind. Selektiert werden alle Felder der Datenbanktabelle, welche die gesuchten Parameter enthalten. Die Commit IDs, welche über die SQL-Abfrage in Listing 6.5 bereitgestellt werden, stellen die benötigten Parameter dar.

```
1 SELECT *
2 FROM outputtable
3 WHERE CommitID1 LIKE ?
4 ...
5 OR CommitIDN LIKE ?
```

Listing 6.8: SQL-Abrage zum Auslesen der Output-Tabelle

Die SQL-Abfrage in Listing 6.7 ist, wie bereits dargelegt, ein Beispiel. Im Allgemeinen hängt die genau Form der SQL-Abfrage von der WHERE-Klausel ab. Im Beispiel besteht sie aus einer Abfrage über drei Commit ID Spalten. Im allgemeinen Fall wird sie wie in Kapitel 6.3.2 beschrieben zusammengestellt und

hat eine, wie in Listing 6.8 abgebildete Form. Die Form ist bedingt durch den Minimum-Support-Wert. Verschiedene Minimum-Support-Werte bedeuten unterschiedliche Output Tabellen und unterschiedliche Output Tabellen bedeuten eine unterschiedliche Anzahl an Commit ID Spalten.

Um nun die mit den Issues gekoppelten Dateiänderungen aus der Output-Tabelle zu berechnen, ist es erforderlich, die in Listing 6.8 abgebildete SQL-Abfrage mit Commit ID Parametern zu bestücken. Wie beschrieben werden diese über die in Listing 6.5 abgebildete SQL-Abfrage erhalten. Da die SQL-Abfrage eine Liste von Commit IDs zurückgibt, muss über diese Liste iteriert werden und die Parameter der SQL-Abfrage in jeder Iteration angepasst werden. Der zuständige Quellcode ist in Listing 6.9 zu sehen.

```
1  while (commit_result.next()) {
2        String commitID = commit_result.getString(1);
3        for (int i = 1; i <= itemcount[0]; i++)
4              output_statement.setString(i, commitID);
5        output_statement.executeQuery();
6        output_result = output_statement.getResultSet();
7        ResultSetMetaData data = output_result.getMetaData();
8        int numcols = data.getColumnCount();
9        while (output_result.next()) {
10             List<String> row = new ArrayList<>(numcols); int i = 2;
11             while (i <= numcols) {
12                   if (output_result.getString(i) != null && !
13 output_result.getString(i).equals("null"))
14                         row.add(output_result.getString(i));
15                   i++;
16             }
17             res.add(row);}}
```

Listing 6.9: Erzeugung der Coupled Changes

Die äußerste While-Schleife iteriert, wie beschrieben, über die Commit IDs. In den Zeilen 3-4 werden die Parameter zur SQL-Abfrage (Listing 6.8) gesetzt. Alle Parameter der SQL-Abfrage werden auf den aktuellen Wert der iterierten gesetzt. Nach diesem Schritt ist die Abfrage vollständig und kann dem Datenbanksystem zum ausführen übergeben werden (Zeile 5). Das Resultat der Abfrage wird in der Variablen *output_result* gespeichert. Es besteht aus allen Datensätzen der Output-Tabelle, welche die aktuelle Commit ID in einer ihrer Commit ID Spalten enthält. In den Zeilen 9-17 wird über das Ergebnis der Abfrage bzw. *output_result* iteriert. Die Daten der Records werden in der Variablen *row* gespeichert (Zeile 14). Da die Felder der Output-Tabelle nicht alle mit validen, brauchbaren Daten beschrieben

sind, werden nur diejenigen Felder gespeichert, welche nicht *null* sind. Schließlich wird *row* in der Liste *res* gespeichert und die Iteration fährt wieder in Zeile 9 fort, solange bis alle Datensätze von *output_result* kopiert sind.

Die Tatsache, dass über mehrere Commit IDs iteriert wird und mit diesen Commit IDs jeweils SQL-Abfragen in der Output-Tabelle getätigt werden, kann zu Duplikaten in *res* führen. Es kann nicht ausgeschlossen werden, dass ein Teil der Datensätze identisch ist. Um Duplikate zu vermeiden, ist die Variable *res* vom Datentyp Set, denn es können keine zwei identischen Objekte in ihr vorkommen.

Die Variable *res* ist ein Teil des DMRO, da es dasselbe Format besitzt (vgl. Tabelle 5.2). Die in res gespeicherten Datensätze aus der Output-Tabelle repräsentieren die Coupled Changes die zu einer Issue gehören. Um auf die berechneten Coupled Changes von anderen Klassen zugreifen zu können, wird die Instanzvariable *sqlprocedureInput* der DBConnection Klasse auf die Referenz der lokalen Variable *res* gesetzt.

6.3.4 Zusammenfassung

Zur Erzeugung von Coupled Changes für Issue Tasks sind mehrere aufeinander aufbauende Schritte erforderlich. Die erforderlichen Schritte wurden in den Kapiteln 6.3.1-6.3.3 erläutert. Diese Schritte sind Teil der *ReadOutputTable2()* Methode der DBConnection Klasse. Als Input erhält sie die Issue ID einer Issue, die zur Generierung einer Liste von Commits aus der Datenbank benutzt wird. Die Commits stehen dabei im Zusammenhang mit der Issue. Die Commit IDs der Commits wird benutzt, um in der Output-Tabelle alle Sequenzen bzw. Transaktionen herauszufiltern, an denen sie beteiligt sind. Das Ergebnis ist ein DMRO, dass die Informationen über die Coupled Changes beherbergt.

6.4 Implementierung der Benutzeroberfläche

Das Design der Bedienoberfläche folgt den Anforderungen an das Tool. Die Oberfläche wurde in drei verschiedene Bereiche unterteilt:

- Bereich zur Anzeige von Issues
- Bereich zur Anzeige von Informationen zur ausgewählten Issue
- Bereich zur Anzeige von Coupled Changes und zugehörigen Informationen

Für die Repräsentation von verschiedenen Informationen werden SWT-Table

Objekte verwendet. Die Implementierung der Oberfläche und den Änderungen am Wizard werden in den nächsten Kapiteln beschrieben.

6.4.1 IssuesPart

Die Issues, welche über die Wizard Komponente des SRMPs in die Datenbank geschrieben werden, werden in der IssuesPart Part-Komponente angezeigt. Zur Repäsentation der Daten wird eine SWT Tabelle verwendet, welche zwei Spalten besitzt. Die erste Spalte trägt den Namen „ID“, die zweite „Issue Title“. Jede vollständige Zeile in der Tabelle beinhaltet ein Paar aus einer Issue ID und dem zugehörigen Issue Titel.

Wird eine Zeile vom Benutzer selektiert, dann wird der SelectionListener der Tabelle ausgeführt. Dieser ist in Listing 6.10 abgebildet.

```
1 tableIssue.addSelectionListener(new SelectionAdapter() {
2       @Override
3       public void widgetSelected(SelectionEvent e) {
4             broker.post("DisplayIssues",
5                   issues.get(tableIssue.getSelectionIndex()));
6       }
7 });
```

Listing 6.10: IssuesPart - SelectionListener der Tabelle

Die Instanzvariable *issues* beinhaltet eine Liste aus allen Issue-Informationen. Die zugehörigen Issue-Informationen der ausgewählten Tabellenzeile werden über ein Ereignis mit dem Topic „DisplayIssues“ versendet (Zeile 4). Da eine Eins-zu-eins-Beziehung zwischen dem Tabellenindex und der Position der Issue in der Instanzvariable *issues* besteht, wird dieser Index verwendet um die zugehörigen Issue-Informationen aus *issues* zu extrahieren.

6.4.2 IssueInformationsPart

In der IssueInformationsPart werden Issue-Informationen detailliert aufgelistet. Um die Informationen anzuzeigen, wird wieder eine SWT Tabelle verwendet. Die Tabelle hat fünf Spalten mit den Namen:

- Issue ID
- Issue Status
- Issue Type
- Issue Date

- Issue Description

Des Weiteren ist im IssueInformationsPart ein Button mit der Aufschrift „SRM Settings“ enthalten, über den der SRMP-Wizard aufgerufen werden kann. Wird der Button ausgewählt, wird der in Listing 6.11 abgebildete SelectionListener ausgeführt.

```
1  btn_srmsettings.addSelectionListener(new SelectionAdapter() {
2        @Override
3        public void widgetSelected(SelectionEvent e) {
4              SRMSettings.project_name = "maint_tools";
5              DBConnection.getDBConnection()
6                    .setDatabase(SRMSettings.project_name);
7              Wizard w = new Wizard(ctx);
8              NWizardDialog wizardDialog = new NWizardDialog(shell,w);
9              ContextInjectionFactory.inject(wizardDialog, ctx);
10             ...
11             wizardDialog.open();
12       }
13 });
```

Listing 6.11: IssueInformationsPart - SelectionListener von Button „SRM Settings“

Zunächst wird der Projektname in Zeile 4 auf „maint_tools“ gesetzt. Die DBConnection wird daraufhin aufgefordert eine Verbindung zu der Datenbank mit dem gesetzten Projektnamen aufzubauen. Die Instanzvariable *ctx* ist vom Typ IEclipseContext und wird für Dependency Injection im Wizard benötigt (Zeile 7). Im Wizard wird zur Kommunikation mit der IssuesPart ein IEventBroker benötigt. Dieser kann ohne den IEclipseContext nicht direkt in den Wizard injiziert werden. Ist die Konfiguration des Wizards beendet, wird der Wizard-Dialog geöffnet (Zeile 11).

6.4.3 CoupledChangesPart

Für Issue Tasks berechnete Coupled Changes werden in der CoupledChangesPart angezeigt. Dieser Part ist in zwei Tabellen unterteilt. Die erste Tabelle dient der Anzeige von Coupled Changes, die zweite zur Anzeige von Commit-Information. Dies sind die Commit-Informationen zu allen Commit IDs, die in der aktuell ausgewählten Coupled-Changes-Gruppe vorkommen. Die Coupled-Changes-Gruppen bestehen aus den Daten welche über die *ReadOutputTable2()* Methode der DBConnection Klasse bereitgestellt werden (vgl. Kapitel 6.3). Jede Gruppe wird über eine Liste von gekoppelten Dateipfaden definiert. Da die Gruppen jeweils zu einem Coupled Change gehören, sind die Commit-Informationen für die Einträge

in einer Gruppe identisch. Gruppen sind durch leere Zeilen voneinander getrennt.

Die in der Commit Tabelle enthaltenen Spalten sind:

- Commit ID
- Commit Author
- Commit Date
- Commit Message

Des Weiteren enthält die CoupledChangesPart ein Button zum Export der Coupled Changes (siehe Kapitel 6.6).

```
1  List<String> commit_ids = (List<String>) item.getData();
2  DBConnection db = DBConnection.getDBConnection();
3  String where_clause="WHERE id IN ("+buildWhereClause(commit_ids+")";
4  List<List<String>> commit_data=
5        db.ReadTable("committable",where_clause);
6  for(List<String> l : commit_data) {
7        TableItem ti = new TableItem(tableCommitData, SWT.NONE);
8        for(int i=0; i<tableCommitData.getColumnCount(); i++)
9              ti.setText(i, l.get(i));
10 }
```

Listing 6.12: CoupledChangesPart – Ausschnitt SelectionListener

Ist keine Tabellenzeile in der Coupled-Changes-Tabelle ausgewählt, dann ist die Commit Tabelle leer, da nichts anzuzeigen ist. Wird eine Zeile ausgewählt, dann wird in jedem Fall die Commit Tabelle geleert. Ansonsten würde die Tabelle bei hinzufügen von neuen Elementen ständig wachsen und alte Elemente wären immer noch vorhanden.

Weiterhin wird überprüft ob die Zeile ein Daten-Objekt besitzt (vgl. Kapitel 6.5.5), falls nicht dann ist die Zeile leer und es wird nichts gemacht (außer dem Leeren der Tabelle). Ein Daten-Objekt ist hierbei eine Liste von Commit IDs. Besitzt die Zeile jedoch ein Daten-Objekt, dann wird der in Quellcode in Listing 6.12 ausgeführt. Das Daten-Objekt wird aus dem aktuell ausgewählten Tabellenzeile entnommen und in die *commit_ids* Variable gespeichert (Zeile 1). Die *buildWhereClause()* Methode baut aus der *commit_ids* Variablen einen String, in dem die Commit IDs mit Kommas getrennt sind. Dieser String wird als Parameter für die WHERE-Klausel in Zeile 3 eingefügt. Die WHERE-Klausel selektiert alle Commit IDs, welche in der Variablen *commit_ids* vorkommen. Zusammen mit dem Aufruf von *db.ReadTable()* (Zeile 5) entsteht eine SQL-Anweisung welche aus der committable alle Commit-Informationen zurückgibt, welche die Commit IDs

enthalten. Die so erhaltenen Commit-Daten, werden Zeilenweise in die Commit Tabelle eingefügt (Zeilen 6-10).

6.5 Kommunikation zwischen den Komponenten

In Kapitel 6.4 wurde die Benutzeroberfläche beschrieben und wie die verschiedenen Parts ihre Komponenten mit Daten füllen, aber nicht woher sie diese Daten erhalten. Dies wird in den nachfolgenden Kapitel erläutert.

6.5.1 Events

Die Parts reagieren auf Events die von dem Tool versendet werden. Die Events und von welcher Komponente sie versendet werden, sind in Abbildung 6.1 zu sehen.

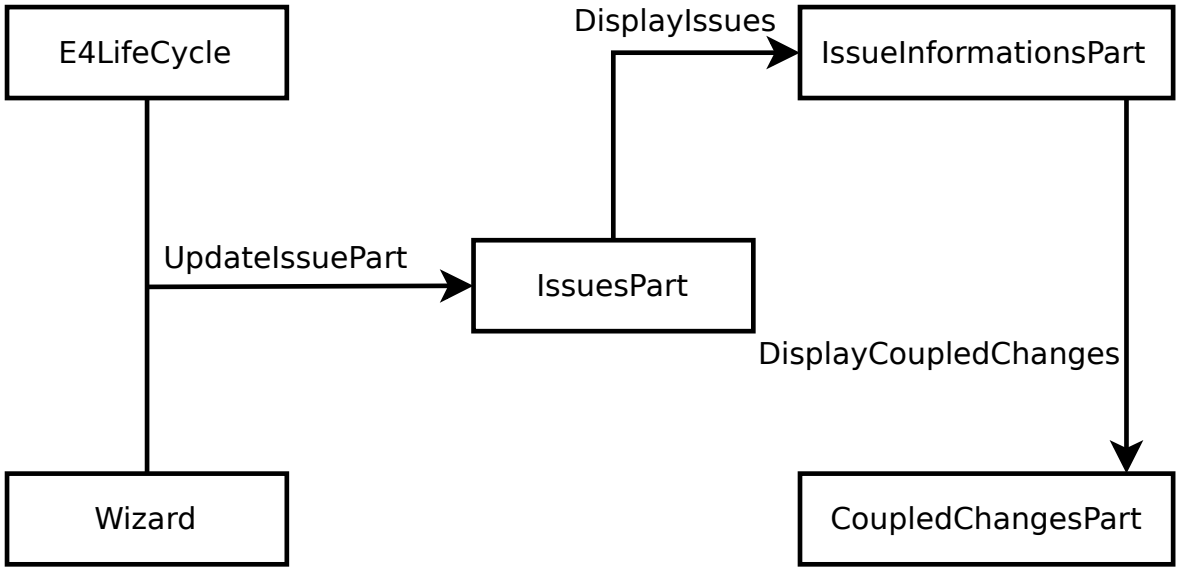

Abbildung 6.1: Von den Komponenten gesendete und empfangene Events

Wie aus der Abbildung erkennbar ist, wird von dem Wizard ein Event mit dem Topic „UpdateIssuePart“ an die IssuesPart Komponente versendet. Dieses Event wird von dem Wizard in dem Zeitpunkt versendet, wenn die Transformation einer Issue-CSV-Datei in die Datenbank beendet ist. Das Event signalisiert also, dass die Issue-Daten in der Datenbank gespeichert sind und das auf diese über SQL-Abfragen zugegriffen werden kann. Es wird auch von der E4LifeCycle Klasse, beim Start der Anwendung versendet.

Von der IssuesPart wird ein Event beim Auswählen einer Tabellenzeile versendet (vgl. Listing 6.10). Das Topic lautet bei diesem Event „DisplayIssues“. Zusammen mit dem Event werden die Issue-Informationen des ausgewählten Elements versendet.

Die IssueInformationsPart versendet ein „DisplayCoupledChanges“ Event, mit einem Coupled Changes Objekt als Parameter.

6.5.2 EventHandler

Die von der Anwendung versendeten Events werden über EventHandler-Methoden empfangen. Jede EventHandler-Methode registriert sich hierfür, für die Events, die für sie von Belang sind. Im Tool existieren EventHandler in den IssuesPart, IssueInformationsPart und CoupledChangesPart Klassen. Über Annotationen werden diese in der Eclipse Platform als EventHandler markiert. Dependency Injection kümmert sich dann um die Injektion der zu empfangenden Parameter. In Tabelle 6.1 sind die im Tool benutzten EventHandler gelistet.

Klasse	**EventHandler**

`IssuesPart`	`@Inject @Optional`
	`void updateListing(`
	`  @UIEventTopic("UpdateIssuePart") String obj)`

`IssueInformationsPart`	`@Inject @Optional`
	`void updateIssueInformationTable(`
	`  @UIEventTopic("DisplayIssues") List<String> issue)`

`CoupledChangesPart`	`@Inject @Optional`
	`void UpdateCoupledChanges(@UIEventTopic("DisplayCoupledChanges")`
	`  List<List<String>> output)`

Tabelle 6.1: EventHandler

Über die Annotation UIEeventTopic() wird die Klasse

- IssuesPart zum Empfang von „UpdateIssuePart“ Events,
- IssueInformationsPart zum Empfang von „DisplayIssues“ Events,
- CoupledChangesPart zum Empfang von „DisplayCoupledChanges“ Events registriert.

6.5.3 EventHandler IssuesPart

Der EventHandler *updateListing()* (siehe Listing 6.13) empfängt „UpdateIssuePart“

Events und ein String Objekt *obj*. Es wird überprüft, ob das Objekt ein valides Objekt ist (Zeile 3). Trifft dies zu, dann wird der Rest des EventHandlers ausgeführt.

```
1  @Inject @Optional
2  private void updateListing(@UIEventTopic("UpdateIssuePart") String obj) {
3        if(obj != null) {
4              DBConnection conn = DBConnection.getDBConnection();
5              conn.setDatabase(SRMSettings.project_name);
6              issues = conn.ReadTable("issuetable", null);
7              tableIssue.removeAll();
8              for (List<String> issue : issues) {
9                    TableItem ti = new TableItem(
10                         tableIssue, SWT.NONE);
11                   ti.setText(new String[] {
12                         issue.get(0), issue.get(5)});
13             }
14       }
15 }
```

Listing 6.13: EventHandler updateListing()

In Kapitel 6.5.1 wurde beschrieben, dass dieses Event nach dem Schreiben der Issue-Daten aus einer CSV-Datei versendet wird. Ausgelöst wird das Schreiben durch den Benutzer. Nachdem er im Wizard eine CSV-Datei ausgewählt und diese mit einem Klick auf den Transformationsbutton in die Datenbank geschrieben hat, müssen die Issues auch in der Benutzeroberfläche im IssuesPart angezeigt werden. Diese Aufgabe übernimmt der *updateListint()* EventHandler.

Der EventHandler setzt den Datenbanknamen und verbindet sich mit der Datenbank (Zeile 5). Die in der Datenbank gespeicherten Issue-Daten werden über die *ReadTable()* Methode ausgelesen und in die Instanzvariable *issues* gespeichert. Als Parameter erhält diese den Namen der Issue Tabelle „issuetable“ (Zeile 6). Schließlich werden eventuell in der Tabelle vorhandene Issues gelöscht, da sie mit neuen ersetzt werden sollen (Zeile 7). In der For-Schleife (Zeile 8-13) wird die Issue SWT-Tabelle der Benutzeroberfläche mit Issue IDs und Issue Titeln der Issues gefüllt (vgl. Kapitel 6.4.1).

6.5.4 EventHandler IssueInformationsPart

Die Issue-Daten, welche von der IssuesPart gesendet werden nachdem ein Element in der Issue Tabelle ausgewählt wurde, werden über den EventHandler *updateIssueInformationTable()* (siehe Listing 6.14) im IssueInformationsPart empfangen. Das Event auf das gewartet wird, ist das „DisplayIssues“ Event und

als Parameter wird per Dependency Injection die Issue-Daten *issue* in den EventHandler injiziert.

```
1  @Inject @Optional
2  private void updateIssueInformationTable(@UIEventTopic("DisplayIssues")
3         List<String> issue) {
4  if (issue != null) {
5         tableIssue.removeAll();
6         TableItem ti = new TableItem(tableIssue, SWT.NONE);
7         for (String s : issue) {
8                ...
9                ti.setText(i++, s);
10               ...
11        }
12        DBConnection.getDBConnection()
13               .ReadOutputTable2(issue.get(0));
14        List<List<String>> sqlprocedureInput =
15               DBConnection.getDBConnection().sqlprocedureInput;
16        broker.send("DisplayCoupledChanges", sqlprocedureInput);
17 }}
```

Listing 6.14: EventHandler updateIssueInformationTable()

Bevor die Issue-Daten übernommen werden, wird die Tabelle geleert (Zeile 5). In den Zeilen 7-11 werden die Issue-Daten nacheinander in die Tabelle eingefügt. Das Resultat ist nach dem Einfügen eine Zeile in der Tabelle, die die Issue-Daten enthält.

Der EventHandler hat auch die Aufgabe die Coupled Changes über die Issue ID zu beschaffen (siehe Kapitel 6.3). Hierfür übergibt er der *ReadOutputTable2()* Methode der DBConnection Klasse die Issue ID der im IssuesPart ausgewählten Issue (Zeile 12-13). Die Coupled Changes werden von der Instanzvariable *sqlprocedureInput* der DBConnection Klasse ausgelesen (Zeile 14-15), welche nach Beendigung der *ReadOutputTable2()* Methode gesetzt wird.

Der EventHandler wiederum sendet selber ein Event mit dem Topic „DisplayCoupledChanges“ und den zuvor generierten Coupled Changes Daten.

6.5.5 EventHandler CoupledChangesPart

Die von dem EventHandler *updateIssueInformationTable()* versendeten „DisplayIssues“ Events kommen im EventHandler *updateCoupledChanges()* (siehe Listing 6.15) der CoupledChangesPart an. Wie in Kapitel 6.5.4 dargelegt wurde, werden zusammen mit dem Event die Coupled Changes Daten gesendet. Diese werden in den Methodenparameter *output* per Dependency Injection eingefügt.

```
1  @Inject @Optional
2  private void updateCoupledChanges(@UIEventTopic("DisplayCoupledChanges")
3        List<List<String>> output) {
4  if(output != null) {
5  tableCoupledChanges.removeAll();
6  tableCommitData.removeAll();
7  cchanges = new ArrayList<>();
8  for(List<String> l : output) {
9        TableItem ti = null;
10       int support = Integer.valueOf(l.get(0));
11       int length  = Integer.valueOf(l.get(1));
12       int index_commits = 2;
13       int index_files = index_commits + support;
14       cchanges.add(l.subList(index_files, l.size()));
15       List<String> commit_ids = new ArrayList<>(support);
16       for(int i=0; i<support; i++)
17             commit_ids.add(l.get(index_commits++));
18       for(int i=0; i<length; i++) {
19             String s = l.get(index_files++);
20             ti = new TableItem(tableCoupledChanges, SWT.NONE);
21             ti.setText(0, s);
22             ti.setData(commit_ids);
23       }
24       new TableItem(tableCoupledChanges, SWT.NONE);
25 }}}
```

Listing 6.15: EventHandler updateCoupledChanges()

Bei Ankommen eines Events mit Eventdaten *output*, werden die Coupled Changes und Commit-Informationen Tabellen geleert (Zeile 5 und 6). Die Instanzvariable *cchanges* speichert die Dateipfade aus dem Inputparameter *output*.

Aus jeder Coupled-Changes-Gruppe (vgl. Format in Tabelle 5.2) im *output* Objekt werden zunächst der Supportwert und die Anzahl der Items bzw. Dateipfade extrahiert (Zeile 10 und 11). Die Variable *l* repräsentiert hierbei die Coupled-Changes-Gruppe. Die Variablen *index_commits* und *index_files* sind Zeiger auf den Start der Commit-Daten und Dateipfaden in den Coupled-Changes-Gruppen. Die Variable index_commits hat den Anfangswert von zwei, da die ersten beiden Elemente der Gruppe *support* und *length* sind und die Liste von Commits nach ihnen beginnt.

Es werden die Commit IDs aus den Coupled-Changes-Gruppen über *index_commits* in die Liste *commit_ids* kopiert (Zeile 16-17). Danach folgen die Dateipfade. Jedoch werden diese nicht in eine Variable kopiert wie bei den Commit IDs, sondern sie werden in die Coupled Changes Tabelle geschrieben. Dabei werden Dateipfade die zu einer Coupled-Changes-Gruppe gehören untereinander in die Tabelle eingefügt. Ist das Einfügen für eine Gruppe von

Dateipfaden beendet, werden diese von anderen Gruppen mit einer leeren Tabellenzeile getrennt (Zeile 24). Zusätzlich erhalten die in einer Gruppe vorkommenden Coupled Changes die zuvor aus der Variablen *output* kopierten Commit IDs, indem die Liste *commit_ids* in die Tabellenzeilen eingebettet wird (Zeile 22). Somit besitzt jede Gruppe aus Dateipfaden in der Tabelle dasselbe *commit_ids* Objekt. Leere Tabellenzeilen erhalten hingegen kein Daten-Objekt.

Das Einbetten der Commit-Informationen in die Tabellenzeilen der Coupled Changes Tabelle ist erforderlich, um sie in der Commit Tabelle anzeigen zu können. Die Commit-Informationen werden nämlich über den SelectionListener der Coupled Changes Tabelle in die Commit Tabelle geschrieben (siehe Kapitel 6.4.3).

6.6 Export von Coupled Changes

In Kapitel 6.4.3 wurde die Oberfläche der CoupledChangesPart beschrieben. Neben der Anzeige von Coupled Changes und Commit-Informationen in SWT-Tabellen existiert auch eine Schaltfläche für den Export der in der Benutzeroberfläche angezeigten Coupled Changes. Der SelectionListener für den Button ist in Listing 6.16 zu sehen.

```
1  btn_exportcchanges.addSelectionListener(new SelectionAdapter() {
2        @Override
3        public void widgetSelected(SelectionEvent e) {
4              FileDialog select_file = new FileDialog(shell,SWT.SAVE);
5              select_file.setFilterExtensions(new String[] {".txt"});
6              String path = select_file.open();
7              if(path != null)
8                    writeCoupledChanges(path);
9        }
10 });
```

Listing 6.16: SelectionListener für den Export von Coupled Changes

Beim Auswählen des Buttons wird ein SWT-FileDialog mit der Option *SWT.SAVE* initialisiert (Zeile 4). Wie der Name vermuten lässt, konfiguriert dies den FileDialog sich im Speichermodus zu öffnen. Dies ermöglicht neben der Navigation durch Ordner, auch die Eingabe eines Dateinamens für die zu exportierenden Coupled Changes. Als Dateiendung sind Dateien mit der Dateiendung „txt“ zugelassen.

Ist die Konfiguration des Speicherdialogs beendet, wird er geöffnet und der Benutzer kann zu dem Verzeichnis navigieren, in der er die Datei abspeichern möchte. Ist ein Dateiname gewählt, kann der Speicherdialog beendet werden.

Mit Beendigung des Speicherdialogs wird der Pfad, welcher den Dateinamen enthält, zurückgegeben (Zeile 6). Nun ist es auch möglich, dass der Benutzer den Speicherdialog abgebrochen hat. In diesem Fall wird „null“ zurückgegeben.

Die Coupled Changes werden nur dann in eine Datei geschrieben, wenn der Pfad im Speicherdialog gesetzt wurde. Trifft dies zu wird die *writeCoupledChanges()* Methode (siehe Listing 6.17) mit dem Pfad als Parameter aufgerufen (Zeile 8).

```
1  private void writeCoupledChanges(String path) {
2        try (BufferedWriter bw =
3              Files.newBufferedWriter(Paths.get(path),
4                    StandardOpenOption.CREATE,
5                    StandardOpenOption.TRUNCATE_EXISTING)) {
6              for(List<String> l : cchanges) {
7                    for(String s : l)
8                          bw.write(s+"\n");
9                    bw.write("\n");
10             }
11       }
12 }
```

Listing 6.17: Schreiben der Coupled Changes in eine Datei

Der übergebene Pfad wird von einem BufferedWriter geöffnet. Existiert die Datei nicht wird sie erstellt. Die Datei kann von einem vorigen Export stammen, deshalb ist es wichtig etwaige vorhandene Daten zu löschen. Dies geschieht mit der *StandardOpenOption.TRUNCATE_EXISTING*.

Wurde die Datei erfolgreich für das Beschreiben geöffnet, können die Coupled Changes Daten in die Datei geschrieben werden. Hiefür wird die zuvor vom EventHandler *updateCoupledChanges()* befüllte *cchanges* Instanzvariable (vgl. Listing 6.15) verwendet.

Die Struktur der exportierten Datei ist der Struktur der Coupled Changes Tabelle im CoupledChangesPart ähnlich. Dies folgt aus der Tatsache, dass die Coupled Changes als Liste von Dateipfaden in *cchanges* abgelegt sind. Dateipfade die in

einer Liste enthalten sind, werden Zeile für Zeile in die Datei geschrieben (Zeile 7-8). Sind jeweils alle Elemente in einer Liste in die Datei geschrieben, wird eine leere Zeile eingefügt (Zeile 9). In der Coupled Changes Tabelle wird ähnlich vorgegangen. Der Unterschied ist, dass leere Tabellenzeilen statt leeren Zeilen eingefügt werden. Am Ende des Vorgans entspricht die Datei der angezeigten Coupled Changes in der Coupled Changes Tabelle.

6.7 Änderungen am Wizard

Durch Integrierung von Prefixspan in das Framework muss der bestehende Wizard mit Prefixspan erweitert werden.

6.7.1 Auswahlmöglichkeit zwischen den Data Mining Algorithmen

Der Wizard im SRMP hatte bisher nur mit einem Data Mining Algorithmus zu tun. In dieser Form genügt der Wizard nicht den Anforderungen. Er muss in der Lage sein auch mit Prefixspan umzugehen. Hierzu wurde der Wizard in der SelectCommitersPage um eine Auswahlmöglichkeit zwischen den Algorithmen erweitert. Hierfür wurde der WizardPage eine SWT-ToolBar eingefügt. In diese ToolBar können ToolItems eingefügt werden. Es wird ein ToolItem für FPGrowth und ein weiteres für Prefixspan eingefügt. Sie tragen die Aufschrift „FPGrowth“ und „Prefixspan“.

Mit Auswahl eines ToolItems wird dessen SelectionListener ausgeführt. Dieser setzt den Data Mining Algorithmus in der SRMSettings Klasse des Wizards. Hierzu wurde die SRMSettings Klasse um ein Enum DataMiningAlgorithm erweitert (siehe Listing 6.18).

```
1  public class SRMSettings {
2        public static enum DataMiningAlgorithm {
3              FPGrowth, PrefixSpan
4        }
5        ...
6        public static DataMiningAlgorithm data_mining_algorithm =
7              DataMiningAlgorithm.FPGrowth;
8  }
```

Listing 6.18: SRMSettings Enum für verwendeten Data Mining Algorithmus

Die Elemente im Enum sind *FPGrowth* und *Prefixspan*. Standardmäßig, falls noch

kein Data Mining Algorithmus gesetzt wurde, wird als Algorithmus FPGrowth vorgegeben (Zeile 6-7).

Wenn das FPGrowths ToolItem ausgewählt wird, dann wird *data_mining_algorithm* auf FPGrowth gesetzt. Bei Prefixspans ToolItem wird es wiederum auf Prefixspan gesetzt. Beim Öffnen des Wizards, wird automatisch das ToolItem ausgewählt, welches in der Variable data_mining_algorithm gesetzt ist. Wird im Wizard z.B. FPGrowth ausgewählt, dann bleibt dieses solange in der ToolBar automatisch ausgewählt, bis FPGrowth ausgewählt wird.

6.7.2 Durchführung des Data Minings

Nach Auswahl des Data Mining Algorithmus und des Minimum-Support-Werts kann der Wizard abgeschlossen werden. Daraufhin werden der ausgewählte Minimum-Support-Wert und Data Mining Algorithmus im Configuration Scope des Eclipse Preference Service gespeichert (siehe Listing 6.19).

```
1 IEclipsePreferences node =
2       ConfigurationScope.INSTANCE.getNode("com.maint_tools");
3 node.put("Algorithm", SRMSettings.data_mining_algorithm.toString());
4 node.putDouble("minimum_support", SRMSettings.minsupport);
5 ...
6 node.flush();
```

Listing 6.19: Speichern der Wizard-Daten

Der Ausgewählte Data Mining Algorithmus wird als String mit dem Schlüssel „Algorithm“ abgelegt. Der Minimum-Support-Wert hingegen wird, da es eine reelle Zahl ist, als Double mit dem Schlüssel „minimum_support“ abgelegt.

Nachdem die Daten geschrieben sind folgt die Durchführung des Data Minings. Dieser Prozess folgt dem im Framework (vgl. Abbildung 5.2) vorgegebenen Ablauf. Aus der „committable“ und „filetable“ wird ein DMIO erzeugt, welches den Algorithmen als Input dient. Diese wiederum produzieren ein DMRO, welches in die Datenbanktabelle „outputtable“ geschrieben wird.

Die Schritte im Quellcode, die dem Vorgang im Framework entsprechen, sind für FPGrowth und Prefixspan sehr ähnlich (siehe Listing 6.20).

```
commit_data = dataBaseConn.ReadInputTable(null);
switch(SRMSettings.data_mining_algorithm) {
case FPGrowth :
       FPGrowthAlgorithmus.input = commit_data.get_res();
       fpgaAlg.runAlgorithm(SRMSettings.minsupport);
       output = FPGrowthAlgorithmus.output;
       maxsupport = FPGrowthAlgorithmus.maxSupport;
       maxlength = FPGrowthAlgorithmus.maxLength;
       break;
case PrefixSpan :
       List<List<String>> files = commit_data.get_fileIDs();
       AlgoPrefixSpan_with_Strings prefixspan =
       new AlgoPrefixSpan_with_Strings(commit_data.get_list_commitID());
       SequentialPatterns patterns = null;
       patterns = prefixspan.runAlgorithm(files, SRMSettings.minsupport);
       output = patterns.getPatterns();
       maxsupport = patterns.get_maximum_support();
       maxlength = patterns.get_maximum_item_length();
       break;
}
dataBaseConn.CreateOutputTable("outputtable",maxsupport,maxlength);
dataBaseConn.WriteIntoOutputTable("outputtable",output,maxsupport,maxlength)
```

Listing 6.20: Ausführung der Data Mining Algorithmen

Als Erstes wird die Input Tabelle ausgelesen und der Inhalt wird in *commit_data* vom Typ CommitTableData gespeichert. Der Inhalt repräsentiert die Daten, die den Data Mining Algorithmen als Input dienen. Danach werden die Input-Parameter für die Algorithmen vorbereitet und der Algorithmus, mit dem im Wizard ausgewählten Minimum-Support-Wert, ausgeführt. Als Resultat generieren die Algorithmen Output Objekte in Form eines DMRO. Das Format der Output Objekte ist für beide Algorithmen identisch. Wie im Framework ersichtlich ist, wird das DMRO zur persistenten Speicherung in die Datenbanktabelle „outputtable“ geschrieben.

Zusätzlich zu den generierten Output Objekten, berechnen die Data Mining Algorithmen den maximalen Support-Wert und die maximale Länge eines im Output Objekt vorkommenden Patterns. Der maximale Support-Wert ist die maximale Anzahl von Commit IDs in allen Patterns. Die maximale Länge hingegen bezeichnet die maximale Anzahl an Dateipfaden in allen Patterns. Diese Informationen werden für die Erstellung der Output-Tabelle und dem Schreiben des Output Objekts in selbiges benötigt (siehe Kapitel 5.5 & 5.5.1). Der Wert von *maxsupport* stellt die obere Grenze für die Commit ID Spalten dar und *maxlength* die obere Grenze für die Item Spalten.

Die Vorbereitung der Input-Parameter erfolgt für die Algorithmen auf

unterschiedliche Weise. FPGrowth erhält das DMIO so wie es ist (Zeile 4) (siehe Kapitel 5.4.1). Bei Prefixspan müssen die in *commit_data* gespeicherten Input-Daten zuerst transformiert werden (siehe Kapitel 6.2.1). Die Transformation wird über *commit_data.get_fileIDs()* (Zeile 11) bewerkstelligt. In der Variablen *files* sind nur noch Dateipfade des DMIO enthalten. Um nach Durchführung von Prefixspan wieder ein vollständiges DMRO zu erhalten, wird dem Algorithmus eine Liste der Commit IDs übergeben. Somit sind die Vorbereitungen beendet und der Algorithmus kann mit *files* als Sequenzdatenbank und dem Minimum-Support-Wert gestartet werden. Das DMRO wird über die getPatterns() Methode mit Hilfe der Commit ID Liste zusammengebaut und in *output* gespeichert (Zeile 16).

6.8 Programmstart

Beim Öffnen des Tools wird die in der E4LifeCycle Klasse definierten Methoden ausgeführt. Diese sind je nach zu erfüllender Funktion mit bestimmten Annotationen annotiert. Damit der Benutzer bei Neustart des Tools nicht immer dieselben Vorschritte zum Berechnen der Coupled Changes durchführen muss, sollen diese nach Programmstart automatisch ausgeführt werden. Diese Vorschritte sind alle Schritte, die zur Erzeugung und Befüllung der Output-Tabelle führen (siehe Kapitel 6.7.2).

Die Methode *processAdditions()* mit der Annotation *@ProcessAdditions* wird aufgerufen nachdem das Applikationsmodell, also alle Komponenten welche die Benutzeroberfläche darstellen, geladen wurde [16]. Nicht nur die Schritte bis zur Befüllung der Output-Tabelle sollen durchgeführt werden, sondern auch die Anzeige der Issue-Daten im IssuesPart. Deshalb ist es erforderlich diese Änderungen in der *processAdditions()* Methode durchzuführen.

```
1  @ProcessAdditions
2  void processAdditions(IEclipseContext workbenchContext) {
3        broker.subscribe(UIEvents.UILifeCycle.APP_STARTUP_COMPLETE,event -> {
4        //Durchführung des Data Minings
5        }
6  }
```

Listing 6.21: Warten auf den vollständigen Programmstart

Damit die erforderlichen Schritte durchgeführt werden können, muss auf den vollständigen Programmstart gewartet werden. Hierzu wird ein IEventBroker

registriert, welcher auf das zugehörige UIEvent (*APP_STARTUP_COMPLETE*, Listing 6.21, Zeile 3) wartet.

Damit die IssuesPart mit Issue-Daten befüllt werden kann, muss erst überprüft werden, ob die Datenbanktabelle „issuetable“ existiert. Nur dann kann davon ausgegangen werden, dass zuvor eine Issue CSV-Datei über den Wizard transformiert und in die Datenbank geschrieben wurde. Wurde in der Datenbank eine „issuetable“ gefunden, muss nur noch ein „UpdateIssuePart“ Event gesendet werden. Der EventHandler der IssuesPart, der für dieses Event registriert ist, kümmert sich dann um die Befüllung der Tabelle für die Issue-Daten (siehe Kapitel 6.5.3).

Da zur Durchführung des Data Minings Commit-Daten vorhanden sein müssen wird überprüft, ob die „committable“ Datenbanktabelle existiert. Ist dies der Fall wird versucht den in der Configuration Scope gesicherten Minimum-Support-Wert und den ausgewählten Data Mining Algorithmus (siehe Listing 6.19) über die Schlüssel „minimum_support“ und „Algorithm“ zu laden. Diese werden dann, wenn sie existieren, in die zugehörigen Instanzvariablen der SRMSettings Klasse kopiert. Andernfalls werden Standardwerte für die Instanzvariablen angenommen. Es folgen daraufhin die Schritte zur Ausführung der Data Mining Algorithmen (vgl. Listing 6.20).

7 Evaluierung

In diesem Kapitel wird die Evaluation des Tools beschrieben. An der Evaluation haben insgesamt acht Personen teilgenommen: Fünf Informatik Studenten und drei Absolventen des Studiengangs Informatik.

7.1 Vorbereitungen und Testumgebung

Als Testumgebung wurde ein Windows-Laptop mit vorinstallierter Eclipse IDE bereitgestellt. Damit das Tool arbeiten kann, muss seine Datenbank mit den Daten eines Git-Repositories gefüllt werden. Für diesen Zweck wurde die Software A-STPA [17] und dessen Git-Repository ausgewählt. Neben dem Git-Repository wird eine zugehörige CSV-Datei mit Issue-Daten benötigt. Damit die Berechnung der Coupled Changes funktionieren kann, müssen die Issue-Daten offensichtlich dem Git-Repository zugehörig sein. Die Issue-Daten wurden für diesen Zweck bereitgestellt, da sie nicht öffentlich zugänglich sind. Die Datenbank des Tools wurde befüllt, indem das Git-Repository und die Issue-Datei über den Wizard importiert wurden.

Weiterhin wurde die A-STPA Software in Eclipse importiert und Eclipse so konfiguriert, dass die Software auch kompiliert und ausgeführt werden kann. Dadurch sind die Teilnehmer in der Lage ihre Änderungen am Quellcode direkt zu testen.

7.2 Testaufbau

Für die Evaluation des Tools wurden die Teilnehmer in zwei gleich große Gruppen unterteilt. Beide Gruppen haben zwei gleiche Aufgaben (Tasks) bekommen, die sie mithilfe des Tools lösen sollten. Zwar sind die Tasks gleich, aber die Parameter, die im Tool gesetzt werden müssen, sind jeweils unterschiedlich.

Die zu lösenden Tasks sind:

- **Task 1:** Ändern der Shortcuts zum Hinzufügen von neuen Items für die CommonTableViewer Klassen von Keycode „n“ in Keycode „i“.
- **Task 2:** Anfügen des Strings „_EditPart“ an die Tooltips für die Komponenten-Elemente der Control Structure.

Die Tasks sind so konstruiert, dass die Lösung der Tasks eine Änderung an der Benutzeroberfläche hervorruft. Dadurch ist kein tiefgehendes Verständnis des A-STPA Quellcodes erforderlich und es müssen keine gravierenden Änderungen von den Teilnehmern durchgeführt werden.

Gruppe 1 führt für Task 1 das Data Mining mit dem Frequent Pattern Mining Algorithmus FPGrowth durch und für Task 2 wird der Sequential Pattern Mining Algorithmus Prefixspan verwendet. Für Gruppe 2 gilt das Gegenteil: Task 1 wird mit Prefixspan und Task 2 mit FPGrowth durchgeführt. Dadurch lassen sich die Ergebnisse zur Lösung der Tasks mit beiden Algorithmen vergleichen. Der Minimum-Support-Wert für Task 1 und Task 2 ist vorgegeben und ist für beide Algorithmen identisch. Für Task 1 wurde ein Minimum-Support-Wert von vier Prozent und für Task 2 ein Minimum-Support-Wert von zwei Prozent festgelegt. Es wird ein identischer Wert vorgegeben um die Vergleichbarkeit der Algorithmen zu gewährleisten.

Damit Coupled Changes generiert werden können, werden für Task 1 und Task 2 Issues vorgegeben, die mit der zu lösenden Issue-Task zusammenhängen.

Die Issue-Tasks und der Testaufbau orientieren sich an denen, die in [18] angegeben sind.

7.3 Testdurchführung

Die Teilnehmer erhalten einen Zettel mit der Aufgabenbeschreibung und welche Schritte sie durchführen müssen. Diese Schritte sind wie sie die Einstellungen bezüglich der Data Mining Algorithmen im Tool vorzunehmen haben und welche Issue sie im Tool auswählen müssen (Issue 854 für Task 1 und Issue 834 für Task 2), um Coupled Changes für die zu lösende Aufgabe zu erstellen.
Fragen bei Unklarheiten bezüglich der Aufgabenstellung und der durchzuführenden Schritte wurden bei Anfragen beantwortet.

Die Teilnehmer haben nun die Aufgabe beide Tasks zu lösen, indem sie die Anleitung benutzen, um Coupled Changes für die Tasks zu generieren. Unter Zuhilfenahme dieser Coupled Changes sollen die Tasks gelöst werden. Im Anschluss füllen die Teilnehmer einen Fragebogen aus (siehe Tabelle 7.1).

Fragen	
Programmiererfahrung in Java*	
Erfahrung mit Issue-Tracking-Systemen (bsp. Jira, Bugzilla, launchpad)	
Die Bedienung des Tools ist verständlich	
Die Anordnung der Views in der Benutzeroberfläche ist gut strukturiert	
Die dargestellten Informationen sind übersichtlicht	Issue- & Commit-Informationen
	Coupled Changes
Das Tool war bei Lösung der Tasks hilfreich mit	Frequent Patterns
	Sequential Patterns
Nützlichkeit von Coupled Changes	

Tabelle 7.1: Fragebogen

7.4 Auswertung

Die Zeit, die ein Teilnehmer für das Lösen einer Task benötigt, wird festgehalten. Des Weiteren wird überprüft, ob eine korrekte Lösung gefunden wurde indem der vom Teilnehmer veränderte Quellcode analysiert wird. Es wird gezählt, wie viele der notwendigen Veränderungen vom Teilnehmer umgesetzt wurden.

7.5 Ergebnisse

In Abbildung 7.1 sind die Umfrageergebnisse abgebildet.

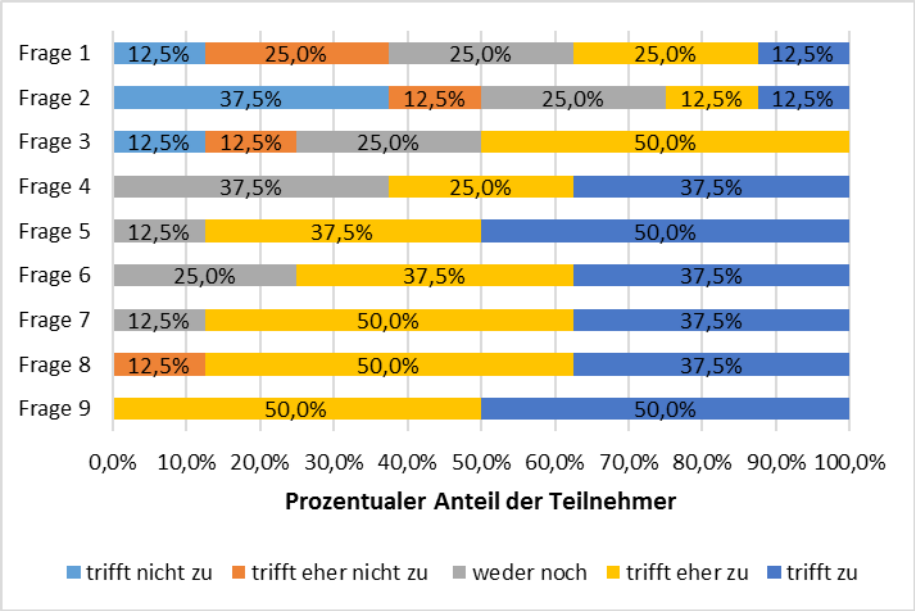

Abbildung 7.1: Umfrageergebnisse

Aus der ersten Frage kann die Verteilung der Programmiererfahrung in Java abgelesen werden. Es herrscht beinahe eine homogene Verteilung. Die Legende ist hier als 1 Jahr, 1-2 Jahre, 2-3 Jahre, 3-4 Jahre und 5+ Jahre zu interpretieren.

Ein großer Prozentsatz (37,5%) hat keinerlei Erfahrung mit Issue-Tracking Systemen, wobei der Mehrheit (62,5%) Issue-Tracking Systeme zumindest ein Begriff ist. Dies zeigt, dass sie mit der Thematik von Issues-Tracking Systemen vertraut sind und eventuell mit den im Tool angezeigten Issue- und Commit-Informationen besser umgehen können. Die Fragen drei bis sechs betreffen das Design des Tools. Die Bedienung des Tools ist für die Mehrheit verständlich, aber es gibt auch Teilnehmer, die diesen Punkt negativ bewertet haben. Die restlichen Fragen bezüglich des Designs wurden positiv bewertet.

Mit den Fragen sieben und acht wurde ermittelt, ob die Teilnehmer die Nutzung der Data Mining Algorithmen zur Lösung der Tasks und somit die von ihnen generierten Coupled Changes als hilfreich empfunden haben. Aus Abbildung 7.1 kann leicht abgelesen werden, dass beide Data-Mining-Algorithmen ein sehr hilfreiche Rolle bei der Lösung der Tasks gespielt haben. In Abbildung 7.2 ist Frage sieben und acht nach den Gruppen aufgeschlüsselt.

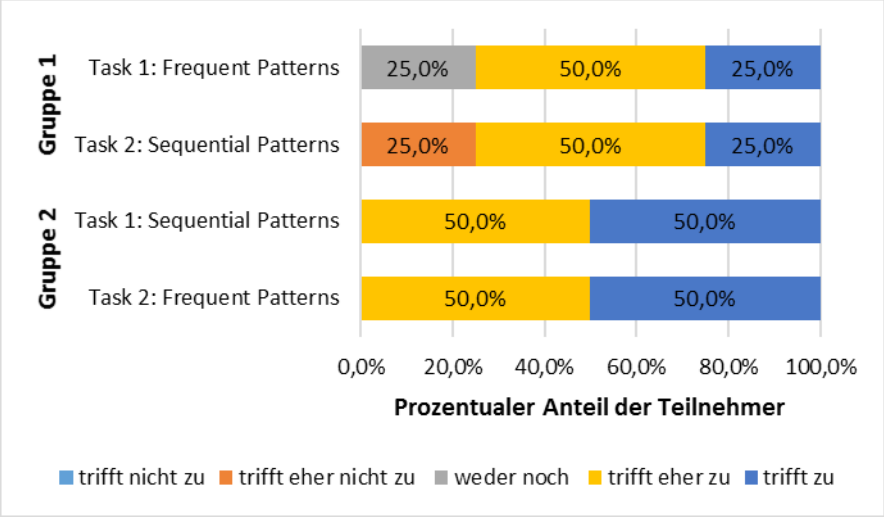

Abbildung 7.2: Frage 7 und 8 nach Gruppen aufgeschlüsselt

Wie zu sehen ist, wurde von Gruppe 2 die Nutzung von sowohl Sequential Pattern Mining als auch Frequent Pattern Mining insgesamt sehr positiv bewertet. In Gruppe 1 wurde Frequent Pattern Mining von neutral bis positiv bewertet, wobei der positive Anteil bei Sequential Pattern Mining identisch ist. Jedoch wurde es von 25% der Teilnehmer eher negativ bewertet. Die letzte Frage betrachtet die Bewertung der Nützlichkeit von Coupled Changes insgesamt, also für beide Algorithmen. Das Ergebnis zeigt, dass die Teilnehmer Coupled Changes als nützlich bis sehr nützlich bewerten.

	Durchschnittliche benötigte Zeit		Standardabweichung	
Gruppe	Task 1	Task 2	Task 1	Task 2
1	12:42	15:39	1:53	5:07
2	13:45	15:09	7:19	4:18

Tabelle 7.2: Durchschnittlich benötigte Zeit für Tasks

In Tabelle 7.2 ist die durchschnittliche Zeit dargestellt, die jede Gruppe für Task 1 und Task 2 benötigten. Gruppe 1 benötigte für Task 1 weniger Zeit als Gruppe 2 und mit geringerer Abweichung zwischen den Teilnehmern. Bei Task 2 sind beide Gruppen praktisch gleich schnell gewesen.

	Korrektheit		Standardabweichung	
Gruppe	Task 1	Task 2	Task 1	Task 2
1	83,3%	93,75%	13,61%	12,5%
2	83,3%	100%	0%	0%

Tabelle 7.3: Durchschnittliche Korrektheit für Tasks

Aus der Tabelle 7.3 lässt sich ableiten, dass für die Lösung von Task 1 beide Gruppen insgesamt denselben Korrektheitsgrad erreichen. Während in Gruppe 2 jeder Teilnehmer dasselbe Ergebnis erreicht (Standardabweichung 0%), ist die Korrektheit bei Gruppe 1 nicht uniform (Standardabweichung 13,61%).

Bei Task 2 erreicht jeder Teilnehmer in Gruppe 2 die vollständige Lösung, während Gruppe 1 hier mit 93,75% trotzdem einen sehr hohen Korrektheitsgrad erreicht.

Sowohl die Ergebnisse aus Tabelle 7.3 als auch den Umfrageergebnissen sind ein Indiz dafür, dass es keinen großen Unterschied macht, welcher Data-Mining-Algorithmus für die Generierung der Coupled Changes verwendet wird, zumindest für dieses Paar an Tasks.

Im allgemeinen könnte Sequential Pattern Mining bessere Ergebnisse aufgrund der sequentiellen Natur der Input-Daten bieten, denn in Git werden die Dateipfade eines Commits lexikographisch ausgegeben. Dies bedeutet, dass potenziel viele Subsequenzen in der Sequenzdatenbank als Muster auftauchen können.

8 Zusammenfassung

Das Ziel dieser Arbeit war, ein Eclipse basierendes Tool zu entwickeln welches in der Lage ist „Maintenance Tasks Issues“ eines Softwareprojekts anzuzeigen und zugehörige Coupled Changes zu extrahieren. Dafür muss das Tool die Daten in seine Datenbank importieren, zusätzlich zu dem Git-Repository, zu dem die Issue Tasks gehören. Mit der internen Repräsentation der Daten können die Git-Logs in Verbindung mit Issue IDs gebracht und Coupled Changes berechnet werden. Die Anforderungen an das Tool wurden bis auf Punkt neun (vgl. Kapitel 4.1) vollständig umgesetzt, da für diesen Punkt leider keine Zeit mehr übrig blieb. Durch Wegfall von Punkt neun sind keine negativen Auswirkungen auf die Evaluation des Tools zu erwarten, denn es hätte keinen Einfluss auf die erzeugten Coupled Changes.

Um die Maintenance-Aufgaben von Software Projekten zu koordinieren, werden Issue Tracking Systeme eingesetzt. Um die Issues zu bearbeiten, sind meist mehrere Änderungen am Quellcode durchzuführen. Für Issue Tasks, die sich ähnlich sind, müssen oft dieselbe Menge an Dateien geändert werden, da sie logisch miteinander gekoppelt sind. Zur Durchführung von neuen Maintenance Tasks kann mit dem Tool durch Auswahl einer ähnlichen Issue Task (falls vorhanden) Coupled Changes bezüglich diesem Task gefunden und die Arbeit des Entwicklers vereinfacht werden. Die Evaluation des Tools hat gezeigt, dass die Lösung von Maintenance Tasks durch Nutzung des Tools mit einem hohen Korrektheitsgrad durchgeführt werden kann.

8.1 Weitere Schritte

Um eine Verbindung zwischen den Issue-Daten und den Git-Logs herzustellen, wurde in den Logs nach Hinweisen für eine Issue gesucht. Als Hinweis dienten Strings wie „#<IssueID>“ und „refs <IssueID>“. Dies hat für die Issues von A-STPA gut funktioniert, denn es wurden hauptsächlich solche Verweise auf Issues in den Commit-Logs verwendet. Doch kann diese Annahme nicht für alle möglichen Issue CSV-Dateien gelten, die dem Tool als Import dienen. Eine Verbesserung

wäre es, die Suchkriterien nach einer IssueID mit regulären Ausdrücken zu erweitern, um eine bessere Trefferquote zu erreichen. Doch ist es meist auch nicht möglich alle Referenzen zu finden, da Entwickler u.a. oft vergessen, beim Committen eine Referenz zu einer Issue ID zu setzen und es gehen somit viele mögliche Referenzen verloren [19].

Das Tool kann zudem mit einer Suchleiste erweitert werden, mit der nach Issues gesucht werden kann, indem Stichwörter eingegeben werden. Denkbar wäre auch der Einsatz von Methoden der maschinellen Sprachverarbeitung um semantisch ähnliche Issues vorzuschlagen.

Literaturverzeichnis

[1] Hassan, A.E., The Road Ahead for Mining Software Repositories, Queen's University, Canada, 2008

[2] Aggarwal, C. C., Data Mining: The Textbook, 2015

[3] Fournier-Viger, P., Lin, C. W., Kiran, R. U., Koh, Y. S., Thomas, R., A Survey of Sequential Pattern Mining, Ubiquitous International, vol. 1, no. 1, 2017

[4] Han, J., Pei, J., Yin, Y., Mao, R., Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach, Data Mining and Knowledge Discovery, vol. 8(1), pp. 53–87, 2004

[5] Pei, J., Han, J., Mortazavi-Asl, B., Dayal, U., Hsu, M.-C., Mining Sequential Patterns by Pattern-Growth: The PrefixSpan Approach, IEEE Transactions on knowledge and data engineering, vol. 16, no. 10, 2004

[6] Common Format and MIME Type for Comma-Separated Values (CSV) Files URL: https://tools.ietf.org/html/rfc4180

[7] Rich Client Platform/FAQ URL: https://wiki.eclipse.org/Rich_Client_Platform/FAQ

[8] White Paper: e4 Technical Overview URL: https://www.eclipse.org/e4/resources/e4-whitepaper-20090729.pdf

[9] Eclipse4/RCP/Dependency Injection URL: https://wiki.eclipse.org/Eclipse4/RCP/Dependency_Injection

[10] Fournier-Viger, P., Lin, C.W., Gomariz, A., Gueniche, T., Soltani, A., Deng, Z., Lam, H. T. (2016). The SPMF Open-Source Data Mining Library Version 2. Proc. 19th European Conference on Principles of Data Mining and Knowledge Discovery (PKDD 2016) Part III, Springer LNCS 9853, pp. 36-40.

[11] Fischer, M., Pinziger, M., Gall, H., Populating a Release History Database from Version Control and Bug Tracking Systems, Vienna University of Technology

[12] Lehmann, S., Automatisierte Transformation von Daten aus Software Repositories und ihre Vorbereitung für Data Mining, Universität Stuttgart, 2015

[13] Alakus, D., Integration von Data Mining in einem Eclipse Plugin, Universität Stuttgart, 2016

[14] Cicek, M. F., Präsentation von Software Repository in Eclipse, Universität Stuttgart, 2015

[15] Demir, Y., Visualisierungsoptimierung von Repository Data Mining in Eclipse, Universität Stuttgart, 2015

[16] Eclipse4/RCP/Lifecycle URL: https://wiki.eclipse.org/Eclipse4/RCP/Lifecycle

[17] A-STPA URL: https://sourceforge.net/projects/astpa/

[18] Ramadani, J., Wagner, S., Are coupled file changes suggestions useful?, 2016 URL: https://doi.org/10.7287/peerj.preprints.2492v1

[19] Ayari, K., Meshkinfam, P., Antoniol, G., Di Penta, M., Threats on Building Models from CVS and Bugzilla Repositories: the Mozilla Case Study. In Proceedings of the 2007 conference of the center for advanced studies on Collaborative research (CASCON '07), Bruce Spencer, Margaret-Anne Storey, and Darlene Stewart (Eds.). IBM Corp., Riverton, NJ, USA, 215-228., 2007

Alle Links wurden zuletzt am 27.03.17 besucht

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