Fakultät II – Informatik, Wirtschafts- und Rechtswissenschaften
Department für Informatik

Bachelorstudiengang Informatik

Bachelorarbeit

Servicebasierte Refactorings

vorgelegt von

Sergej Tihonov

Matrikelnummer: 1436957

Gutachter:

Prof. Dr. Andreas Winter
M.Sc. Jan Jelschen

Oldenburg, 14. November 2013

Inhalt

1 Vision **1**
1.1 Problemstellung . . . 1
1.2 Zielsetzung . . . 1
1.3 Lösungsansatz . . . 1
1.4 Kapitelübersicht . . . 2

2 Grundlagen **3**
2.1 Service . . . 3
2.2 Refactoring . . . 10
2.3 TGraph . . . 12

3 Anforderungen **15**
3.1 Technische Anforderungen . . . 15
3.2 Funktionale Anforderungen . . . 15
3.3 Nichtfunktionale Anforderungen . . . 16

4 Architektur **19**
4.1 Refactoring Analyse . . . 19
4.2 Vereinfachte Architektur . . . 20

5 Erweiterte Architektur mit Anforderungen **25**
5.1 parseJavaToGraph . . . 25
5.2 unparseGraphToJava . . . 26
5.3 executeBadSmells . . . 26
5.4 getChoice . . . 28
5.5 executeRestructurings . . . 30
5.6 combineRefactorings . . . 32

6 Service Katalog **35**

7 Implementierung **43**
7.1 Architekturumsetzung . . . 43
7.2 Compositezerlegung . . . 44
7.3 Service Implementierung . . . 46

8 Ansätze zu Erweiterbarkeit **51**
8.1 Refactoringerstellung . . . 51
8.2 Refactoringerweiterung . . . 51

9 Validierung **55**

9.1 Servicewiederverwendbarkeit als Metrik 55
9.2 Refactoring Verknüpfung 56

10 Zusammenfassung **59**

CD-Verzeichnis **61**

Abkürzungen **63**

Abbildungen **65**

Literatur **67**

Index **69**

Anhang **71**

1 Vision

In dem folgenden Kapitel wird die Problemstellung dieser Bachelor-Arbeit beschrieben. Resultierend aus der Problemstellung wird die Zielsetzung für diese Arbeit definiert und der erste Lösungsansatz aufgezeigt.

1.1 Problemstellung

Softwareentwicklung mit einer Intergrierten Entwicklungsumgebung, im Folgenden mit IDE abgekürzt, bietet viele Vorteile gegenüber einem Texteditor ohne Hilfsmittel. Ein Compiler, ein Linker und ein Debugger sind bereits in die IDE integriert und erleichtern somit die Fehlersuche. Tippfehler werden durch die Autovervollständigung wesentlich reduziert. Außerdem werden Schlüsselwörter vorgeschlagen und durch farbliche Markierung im Quelltext hervorgehoben.
Eines der wichtigsten Features ist das Refactoring[Fow00]. Es ermöglicht eine effiziente Veränderung der Codestruktur, wodurch der Quelltext beispielsweise zu Gunsten der Lesbarkeit, der Performance oder der Erweiterbarkeit optimiert werden kann.
Neu entwickelte IDE können nicht auf Tools von bereits existierenden IDEs zugreifen, da diese in sich geschlossen und nicht modular sind[Jel13]. Somit müssen bereits etablierte Tools wiederholt neu geschrieben werden. Da dies sehr zeit- und kostenintensiv ist, soll eine Alternative entwickelt werden. Diese soll den Einbau und die Wiederverwendung der bereits existierenden Tools ermöglichen.

1.2 Zielsetzung

Ziel dieser Arbeit basiert auf der servicebasierten Betrachtung von Refactorings. Dabei ist ein Service ein Dienst, der genau eine Aufgabe erfüllt. Somit soll der Refactoring-Prozess in einzelne Aufgaben zerlegt werden. Aus den Aufgaben soll daraufhin eine Architektur aufgebaut werden. Diese muss flexibel sein und ohne großen Aufwand weitere Refactorings in sich aufnehmen können.
Bei der Zerlegung des Refactoring-Prozesses in einzelne Services, soll aufgezeigt werden, dass sie Überschneidungen in der Funktionalität aufweisen. Bei Überschneidungen sollen die Services nach Möglichkeit angeglichen werden, damit sich keine geklonte Funktionalität sich in der Architektur befinden.
Die Services sollen nicht spezifisch für den Refactoring-Tool geschrieben werden. Sie sollen die Möglichkeit bieten sie in anderen Tools wiederzuverwenden. Diese Eigenschaft soll ebenfalls an einem Beispiel demonstriert werden.
Alle erstellten Services sollen in einem Katalog aufgeführt werden. Dies soll die Übersicht über die Services erleichtern und doppelte Implementierung verhindern. Die richtige Art der Beschreibung eines Services muss im Rahmen der Arbeit noch definiert werden.

1.3 Lösungsansatz

Damit das Endergebnis den Anforderungen entspricht, wird eine solide Grundlage benötigt, welche die nötige Modularität und Stabilität bereitstellt. Die Service-Oriented Architecture, folgend SOA genannt, ist ein modernes und weit ausgereiftes Architekturmuster, welches vor allem auf die strikte Trennung zwischen der Schnittstelle und der Funktionalität setzt. Für SOA gibt es mehrere Implemen-

tierungen. Johannes Meier zeigte bereits in seiner Bachelorarbeit[Mei12], dass Service Component Architecture (SCA) eine leichte Anbindung bietet und sich damit auch umfassende Projekte realisieren lassen. Durch den modularen Aufbau und die festgeschriebenen Schnittstellen lassen sich die einzelnen Komponenten leicht ersetzen und gut wiederverwenden[Cha07]. Somit erfüllt SCA alle Anforderungen, um das Grundgerüst des Refactoring-Tools zu realisieren.
Vorbereitend für die Realisierung müssen die Refactorings aufbereitet werden. Dazu wird die Top-down Methode verwendet. Hierbei werden sie schrittweise in Teilfunktionen zerlegt, bis nur noch grundlegende Funktionen vorhanden sind. Dabei ergibt sich eine hierarchische Anordnung, die auf unterer Ebene sehr technisch ist und aus mehreren elementaren Services besteht und zur Spitze hin immer mehr Services bündelt und dadurch den Überblick über die interne Funktionsweise erleichtert. Die abstrakten Services lassen sich leichter handhaben und die technischen sich leichter wiederverwenden und mit anderen Services verknüpfen.
Um den Lösungsansatz in der Praxis zu testen, wird das Verfahren im kleinen Rahmen als Prototyp implementiert. Dabei werden gezielt einige Refactorings ausgewählt, um das Verhalten bei Funktionsüberschneidungen und den Aufwand des Hinzufügens weiterer Refactorings zu untersuchen. Desweiteren wird eine Fallunterscheidung zwischen der automatischen Erkennung von problematischen Stellen im Quelltext und der Benutzeninteraktion stattfinden, um sich der richtigen Balance und der daraus resultierenden Benutzerfreundlichkeit anzunähern. Die daraus gewonnen Informationen werden zur Optimierung der Architektur verwendet.

1.4 Kapitelübersicht

In Kapitel 2 werden zunächst die Grundlagen, die für diese Arbeit benötigt werden erläutert.
In Kapitel 3 werden Anforderungen gesammelt, um das Ziel der Arbeit zu konkretisieren und einzugrenzen.
In Kapitel 4 wird die Architektur für das Refactoring-Tool erstellt.
In Kapitel 5 wird die Architektur mit den Anforderungen ergänzt, wodurch eine implementierungsnahe Servicezerlegung durchgeführt werden kann.
In Kapitel 6 werden alle erstellten Services alphabetisch aufgelistet.
In Kapitel 7 wird die Implementierung der Services beschrieben.
In Kapitel 8 wird vorgestellt, wie die bestehenden Infrastruktur um einen weiteren Refactoring erweitert werden kann.
In Kapitel 9 werden einige Ergebnisse der Arbeit getestet und bewertet.
In Kapitel 10 werden die Ergebnisse der Arbeit zusammengefasst.

2 Grundlagen

Ziel des Kapitels ist es die verwendeten Technologien vorzustellen. Von jeder Technologie wird die Funktionsweise in kurzer Form dargestellt und an einem Beispiel demonstriert. Dadurch wird ein Grundverständnis aufgebaut um den nachfolgenden Text folgen zu können ohne bereits diese Technologien zu kennen.

2.1 Service

Ziel des Kapitels ist es ein Grundverständnis für Services aufzubauen. Es werden die wichtigsten Eigenschaften von Services und ihre Umsetzung in der Praxis vorgestellt.

2.1.1 Service-Oriented Architecture

Service-Oriented Architecture, abgekürzt SOA, ist ein Architekturmuster. Sein Ziel ist es Software aus wiederverwendbaren Services zu erstellen, um kosteneffizientes Entwickeln zu ermöglichen [ABB+09].
Jeder Service ist ein Dienst, der genau eine fest definierte Aufgabe erledigt. Dazu hat der Service eine Schnittstelle, welche die Eingabe- und Ausgabe-Parameter definiert. Ansonsten ist er eine Blackbox [HBBK07]. Das bedeutet, dass das Innere des Services nicht eingesehen werden kann. Die einzigen Informationen, die nach außen zur Verfügung stehen sind die Beschreibung der Serviceaufgabe und die Schnittstelle, über die der Service angesprochen werden kann.
Intern kann ein Service seine Aufgabe direkt erledigen oder sich weiterer Services bedienen, um seine Aufgabe zu lösen. Wenn sich ein Service eines anderen Services bedient, dann hat er eine Referenz auf ihn. Der obere Service kennt ebenfalls nur die Beschreibung der Serviceaufgabe und die Schnittstelle von dem unteren Service. Nach diesem Prinzip kann es beliebig viele Verschachtelungen geben.
Die beschriebene Verbindung besteht aus einer Referenz und einem Service. Somit ist es eine „*1 zu 1*“ Verbindung. Mit Services kann auch eine „*1 zu N*“ Beziehung erstellt werden. Dazu wird ein abstrakter Service benötigt. Der obere Service referenziert in diesem Fall den abstrakten Service und die unteren Services erben von dem abstrakten Service, wodurch eine „*1 zu N*“ Beziehung entsteht.
Eine „*N zu 1*“ Verbindung ist mit Services in dieser Form nicht möglich. Dies beruht auf der Lebensspanne eines Services. Denn beim Aufruf eines Services wird eine neue Instanz von ihm erstellt. Diese Instanz bleibt bestehen, bis der Service seine Aufgabe erfüllt und das Ergebnis zurückgibt. Danach wird die Instanz zerstört. Wird ein Service mehrfach aufgerufen, dann wird für jeden Aufruf parallel eine eigene Instanz erstellt. Somit entstehen mehrere „*1 zu 1*“ Verbindungen, aber keine „*N zu 1*“ Verbindung. In der SOA Implementierung, welche in Kapitel 2.1.2 vorgestellt wird, gibt es Möglichkeiten auf bereits erstellte Service-Instanzen zuzugreifen.
Zur Beschreibung von Verbindungen zwischen Referenzen und Services wird das „Service Assembly Model“ -Diagramm, im Folgenden mit SAM abgekürzt, eingeführt. Zur Darstellung des Diagramms werden Elemente aus dem UML 2.4.1 Klassendiagramm [Obj] verwendet. Services werden mit Klassen repräsentiert. Referenzen werden mit Compositionen abgebildet, da beim Zerstören des oberen Services auch der untere Service zwangsläufig wegen seiner Lebensspanne zerstört wird. Damit die Beziehung zwischen zwei Services leichter erkennbar ist, werden zusätzlich die Multiplizitäten ergänzt.

Die Abb. 2.1 zeigt ein Beispiel von zwei Services. Dabei hat der *ServiceA* eine Referenz auf den *ServiceB* und es besteht eine „*1 zu 1*“ Beziehung zwischen diesen beiden Services.
Zur Darstellung einer „*1 zu N*“ Beziehung wird ein abstrakter Service benötigt. Der obere Service

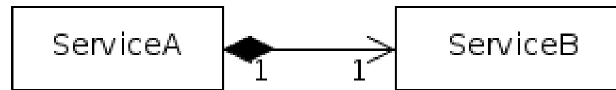

Abbildung 2.1: SAM-Diagramm Beispiel 1

referenziert den abstrakten Service und die unteren Service werden an den abstrakten Service gebunden. Dadurch bekommt der obere Service den Zugriff auf alle an den abstrakten Service gebundenen Services. Die Abb. 2.2 zeigt dazu ein Beispiel. Der *ServiceA* hat eine Referenz auf den *«abstract» ServiceB*. An den *«abstract» ServiceB* werden die Services *ServiceB1* bis *ServiceB3* gebunden. Dadurch besteht eine „*1 zu N*“ Beziehung zum *ServiceA*.

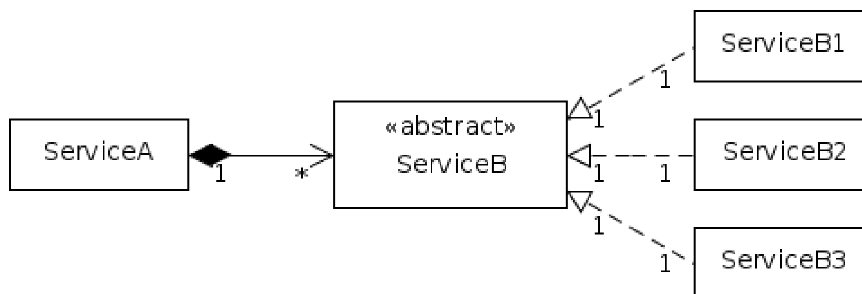

Abbildung 2.2: SAM-Diagramm Beispiel 2

Jeder Service erfüllt eine eigenständige Aufgabe. Die Kopplung zwischen den einzelnen Services ist sehr locker und sie können einzeln ausgeführt werden. Es entsteht ein Pool aus Services. Aus diesem Pool können Services für eine benötigte Aufgabe herausgenommen werden, falls diese dort bereits existieren. Oder es können Services mit neuer Funktionalität hinzugefügt werden. Damit eine Auswahl möglich ist und damit keine Services doppelt auftreten, sollte es eine einheitliche Beschreibung von Services geben. Mit Hilfe dieser Beschreibung werden alle Services in ein Katalog aufgenommen und alphabetisch sortiert.
Jeder Service benötigt einen Namen. Über diesen Namen kann jeder Service identifiziert werden. Somit muss der Name einzigartig sein.
Jeder Service benötigt eine Beschreibung. Die Beschreibung beschreibt die Funktionalität des Services. Die Funktionalität des Services kann nur der Beschreibung entnommen werden, da jeder Service eine Blackbox ist.
Jeder Service benötigt eine Eingabe. Die Eingabe ist eine Menge aus Eingabeparametern, die benötigt werden um den Service zu starten. Die Eingabe kann auch aus einer leere Menge bestehen. Dann werden keine Eingabeparameter benötigt.

Jeder Service benötigt eine Ausgabe. Die Ausgabe ist eine Menge aus Ausgabeparametern, die der Service nach der Beendigung seiner Ausgabe ausgibt. Die Ausgabe kann auch aus einer leeren Menge bestehen. Dann werden eine Ausgabeparameter zurückgegeben.
Jeder Service benötigt eine besondere Eigenschaft. Diese wird als *Eigenschaft* abgekürzt, damit die Einrückungen in der Übersicht einheitlich sind. Die Eigenschaft beschreibt weitere Möglichkeiten des Services, die die Funktionalität erweitern.
Das Format für die Serviceübersicht sieht folgendermaßen aus:

- Servicename

Beschreibung:	Beschreibung der Funktionalität
Eingabe:	Menge der Eingabeparameter
Ausgabe:	Menge der Ausgabeparameter
Eigenschaft:	Besonderheiten des Services

2.1.2 Service Component Architecture

Service Component Architecture ist eine mögliche Implementierung von SOA. Johannes Meier zeigte bereits in seiner Bachelorarbeit[Mei12], dass Service CSA eine stabile Grundlage für die SOA Implementierung bietet und sich damit für Realisierung von umfangreichen Projekten eignet. Beruhend auf dieser Erkenntnis wird SCA als Implementierung von SOA verwendet.
SCA beschreibt wie die einzelnen Konzepte von SOA auf der Implementierungsebene realisiert werden können. Die Implementierung wird in mehreren Programmiersprachen unterstützt. Im Rahmen dieser Bachelor-Arbeit wird ausschließlich Java zur Implementierung von Services und XML zur Erstellung von Verbindungen zwischen den einzelnen Services verwendet. Somit beruhen alle folgenden Beschreibungen auf diesen beiden Programmiersprachen.
SCA besteht grundlegend aus sechs Elementen: Composite, Component, Service, Referenz, Property und Wire. Gleichzeitig zur Klärung der einzelnen Begriffe wird das „Assembly Model“ -Diagramm eingeführt, im Folgenden mit AM abgekürzt, welches die einzelnen Elemente grafisch darstellt. Eine Übersicht der Elemente wird in der Abb. 2.3 dargestellt.
Als Beginn der Beschreibung wird die Component gewählt. Es ist der dunkelblaue Rechteck mit abgerundeten Ecken in der Abb. 2.3. Die Component kann als eine abstrakte Hülle gesehen werden, die alle Elemente für die Umsetzung eines Services zusammenhält. Diese wird in XML beschrieben und benötigt einen einzigartigen Namen, damit auf sie von überall referenziert werden kann. Die Component benötigt eine Implementierung und mindestens einen Service. Der Service muss mit der Annotation @*Service* explizit bei der Implementierung angegeben werden. Zusätzlich dazu können Referenzen und Propertys hinzugefügt werden.
Der Service in SCA unterscheidet sich von dem SOA-Service. Der SOA-Service beschreibt den gesamten Dienst in seiner abstrakten Form. Der SCA-Service beschreibt nur das Interface, welches an die Component angehängt wird. Um diese Begriffsüberschneidung zu beseitigen wird der Service Begriff nur für den SOA-Service verwendet. Der SCA-Service bekommt den „SCA-“ Präfix und kann somit von dem Service eindeutig unterschieden werden.
Der SCA-Service wird als Interface realisiert. Es ist der grüne Pfeil in der Abb. 2.3. Der Name, die Beschreibung und die Parameter können dem Service entnommen werden. In SCA gibt es verschiedene Möglichkeiten ein SCA-Service anzusprechen. Bei der Standardeinstellung kann ein SCA-Service nur lokal angesprochen werden. Mit der Annotationen @*Remotable* kann ein SCA-Service zu einem

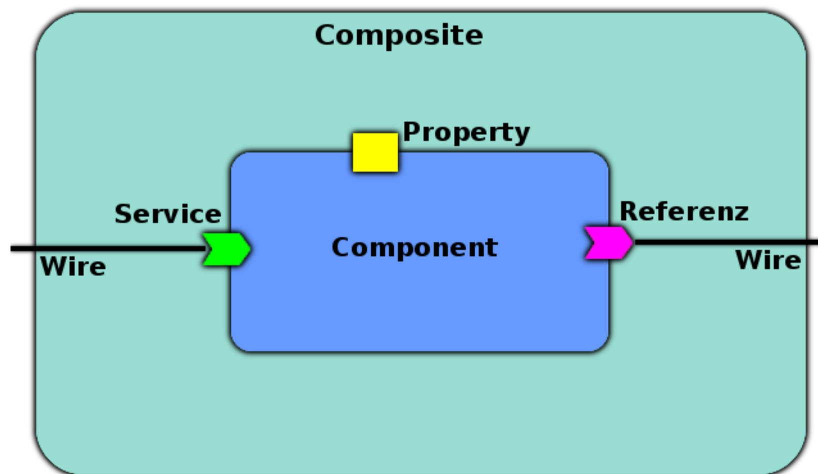

Abbildung 2.3: AM-Diagramm Übersicht

WebService umgewandelt werden. Danach kann der SCA-Service über HTTP angesprochen werden. Es gibt eine Reihe weiterer Annotationen [LCF$^+$11], auf diese wird aber im Rahmen dieser Bachelor-Arbeit nicht eingegangen.
Die Referenz ist ein Platzhalter eines SCA-Services, der für die Realisierung der Funktionalität benötigt wird. Es ist der rosarote Pfeil. Die Referenz wird an die Component angehängt und im Quelltext mit der Annotation @*Reference* markiert. Der Name der Referenz muss im Quelltext und in der Component äquivalent sein. Es ist zu beachten, dass die Referenz nicht *private* sein darf.
Bei dem Beschreiben der Referenz in der Component kann ein Zielservice angegeben werden. Somit wird eine Verbindung zwischen der Referenz und dem SCA-Service erstellt. Diese Verbindung wird Wire genannt. Es ist die schwarze Linie in der Abbildung. Die Wire stellt siecher, dass beim Aufrufen der Referenz der dazugehörige SCA-Service und seine Component gestartet werden.
Die Property bietet die Möglichkeit der Component Informationen bei der Instanziierung mitzugeben. Es ist der gelbe Viereck in der Abbildung. Somit kann die gleiche Component in mehreren Instanzen verschiedene Informationen beinhalten. Die Property wird an die Component angehängt und im Quelltext mit der Annotation @*Property* markiert. Die Property kann zum Beispiel dazu genutzt werden Login-Daten einzutragen.
Das letzte SCA-Element ist die Composite. Es ist der hellblaue Rechteck mit abgerundeten Ecken in der Abbildung. Die Composite ist das hierarchisch oberste Element und beinhaltet die Informationen von der SCA-Version. Die Composite beinhaltet mindestens eine Component und wird ebenfalls in XML beschrieben.

In Kapitel 2.1.1 wurde erwähnt, dass ein Service eine begrenzte Lebensspanne besitzt. In der Java-Implementirung von SCA kann die Instanziierung eines Services beeinflusst werden. Dazu gibt es die Annotation @*Scope* [LCF$^+$11] und drei mögliche Eingaben: STATELESS, COMPOSITE und CONVERSATION.
Das Stateless-Scope ist die Standardeinstellung in jedem Service und kann daher weggelassen werden. Mit diesem Scope wird bei jedem Aufruf des Services eine neue Instanz erstellt. Nach der Beendigung der Aufgabe wird die Instanz zerstört und ist nicht mehr erreichbar.
Das Composite-Scope erstellt eine Instanz des Services und leitet alle Referenzen auf diese Instanz

um. Beim ersten Aufruf des Services wird eine statische Instanz erstellt, welche von allen dazugehörigen Referenzen erreicht und nicht mehr zerstört wird. Mit der Annotation *@EagerInit* kann sofort eine statische Instanz des Services erstellt werden ohne den ersten Aufruf abzuwarten. Dies hat den Vorteil, dass alle Informationen vorzeitig geladen werden können und beim Aufruf diese Zeit erspart bleibt.
Das Conversation-Scope ermöglicht die Kontrolle der Lebensspanne und des Zugriff auf den Service. Dabei kann eine Instanz des Services kontrolliert erstellt werden. Die Referenzen werden dann auf die bestehende Instanz umgeleitet. Wenn die Instanz nicht mehr benötigt wird, kann diese kontrolliert zerstört werden.

2.1.3 SCA Beispiel

In diesem Kapitel wird die Implementierung von SCA an einem Beispiel demonstriert. Als Beispiel wird ein Ausschnitt aus der Implementierung des Parsers aus dem Kapitel 7 vorgegriffen. Anhand des „*parseJavaToGraph*"-Services soll die Verwendung von SCA-Annotationen in der Implementierung und die Erstellung der XML-Datei mit der Composite gezeigt werden.

Als erstes muss das *ParseJavaToGraph* Interface erstellt werden. Die Abbildung 2.4 zeigt den dazugehörigen Quelltext. Wie an dem Interface zu sehen ist, handelt es sich um einen normalen Java-Interface. Das liegt daran, dass der „*parseJavaToGraph*"-Service ein lokaler Service ist. Um diesen Service zu einem WebService umzuwandeln, muss über die Zeile 1 die Annotation *@Remotable* hinzugefügt werden.

```
1 public interface ParseJavaToGraph {
2     public Graph javaToGraph(String[] pathJavaFiles, String[]
         pathClassFiles) throws FileNotFoundException,
3     ClassNotFoundException, XMLStreamException, GraphIOException;
4 }
```

Abbildung 2.4: ParseJavaToGraph Interface

Als zweites muss die *ParseJavaToGraphImpl* Klasse erstellt werden. Die Abbildung 2.5 zeigt den dazugehörigen Quelltext. Der erste Unterschied zu einem normalen Java-Quelltext ist die Annotation in der Zeile 1 *@Service*. An dieser Stelle muss das implementierte Interface als Service markiert werden. In Zeile 4 wird eine Referenz auf den *ParseJavaToXML*-Service erstellt. Dafür wird eine Variable von dem *ParseJavaToXML*-Interface erstellt. Die Sichtbarkeit wird auf *protected* gesetzt, da *private* verboten ist. Damit das als Referenz von SCA erkannt wird, muss die Annotation *@Reference* eine Zeile darüber hinzugefügt werden. Dasselbe wird auch bei der Referenz auf den *GraphImport*-Service gemacht. Im Quelltext können die beiden Variablen ganz normal benutzt werden und alle Methoden aus dem Interface verwendet werden, wie zum Beispiel in der Zeile 14. Die Instanz wird über die XML-Datei angebunden und erst zur Laufzeit instanziiert.

Als drittes muss die XML-Datei mit der Composite erstellt werden. Die Abbildung 2.6 zeigt den dazugehörigen Quelltext.

```
1  @Service(ParseJavaToGraph.class)
2  public class ParseJavaToGraphImpl implements ParseJavaToGraph {
3
4      @Reference
5      protected ParseJavaToXML parseJavaToXML;
6
7      @Reference
8      protected GraphImport graphImport;
9
10     @Override
11     public Graph javaToGraph(String[] pathJavaFiles, String[]
           pathClassFiles)
12             throws FileNotFoundException, ClassNotFoundException,
13             XMLStreamException, GraphIOException {
14         String xmlPath = parseJavaToXML.parseJavaData(pathJavaFiles
               ,
15                 pathClassFiles);
16         return graphImport.importGraph(xmlPath);
17     }
18
19 }
```

Abbildung 2.5: ParseJavaToGraphImpl Klasse

In Zeile 1 wird die XML gestartet. Es wird die Version und die Codieren angegeben.
In Zeile 2 wird die Composite erstellt. Diese benötigt einen beliebigen *targetNamespace* und einen beliebigen *name*. Der *autowire* in Zeile 3 wird auf *false* gestellt, da die Wire manuell eingetragen werden. In Zeile 4 wird mit *xmlns* die SCA Version angegeben.
In Zeile 6 wird die Component für den „*parseJavaToGraph*“-Services erstellt. Diese benötigt einen beliebigen Namen.
In Zeile 7 wird die Implementierung an die Component angehängt. Dabei muss der Typ der Implementierung angegeben werden.
In Zeile 8 wird der Service an die Component angehängt. Diese benötigt den Namen des Interfaces.
In Zeile 9 wird das Interface an der Service angehängt. Bei dem Interface muss der Typ der Implementierung angegeben werden.
In Zeile 11 und 14 werden Referenzen an die Component angehängt. Bei beiden muss der Name der Variable und das Ziel auf den referenzierten Service angegeben werden. Das Ziel besteht aus dem Componentennamen in dem sich das Service befindet und dem eigentlichen Servicenamen.
Eine Zeile später muss das Interface von dem referenzierten Service angegeben werden. Dabei muss ebenfalls der Typ der Implementierung ergänzt werden.

```
1  <?xml version="1.0" encoding="UTF-8"?>
2  <composite targetNamespace="http://ParseServices"
3    name="ParseServices" autowire="false"
4    xmlns="http://www.osoa.org/xmlns/sca/1.0">
5
6    <component name="ParseJavaToGraphComponent">
7      <implementation.java class="datahandler.ParseJavaToGraphImpl"/>
8      <service name="ParseJavaToGraph">
9        <interface.java interface="datahandler.ParseJavaToGraph"/>
10     </service>
11     <reference name="parseJavaToXML" target="
         ParseJavaToXMLComponent/ParseJavaToXML">
12       <interface.java interface="datahandler.parser.ParseJavaToXML"
           />
13     </reference>
14     <reference name="graphImport" target="GraphImportComponent/
         GraphImport">
15       <interface.java interface="datahandler.importgraph.
           GraphImport"/>
16     </reference>
17   </component>
18
19 </composite>
```

Abbildung 2.6: ParseServices Composite

2.2 Refactoring

Refactoring beschreibt nach Martin Fowler [Fow00] einen Prozess, bei dem die Software umstukturiert wird ohne ihr beobachtbares Verhalten zu ändern. Dabei wird nicht unterschieden ob dieser Prozess automatisch oder manuell geschieht. Der Refactoring Prozess besteht aus zwei Teilen.
Der erste Teil ist für das finden von Schwachstellen im Quelltext zuständig und wird BadSmell genannt. Der BadSmell Prozess untersucht den Quelltext gezielt nach bestimmten Eigenschaften und liefert die gefundenen Schwachstellen.
Der zweite Teil ist für das beseitigen von den gefundenen Schwachstellen zuständig und wird Restructuring genannt. Der Restructuring Prozess nimmt die gefundenen Schwachstellen aus dem Quelltext entgegen und beseitigt diese. Dabei gibt es meistens mehrere Wege ein BadSmell zu beseitigen.
Ein Refactoring wird immer mit einem bestimmten Ziel durchgeführt. Das Ziel kann es sein die Lesbarkeit zu verbessern, leichtere Erweiterbarkeit zu ermöglichen oder die Wartung des Quelltextes zu erleichtern. In Abhängigkeit des Ziels werden andere BadSmells gefunden und unterschiedlich durch das Restructuring beseitigt. Diese können auch widersprüchlich sein. Denn wenn sich etwas positive auf die Lesbarkeit auswirkt, kann es sich gleichzeitig negative auf die Performance auswirken.

2.2.1 Refactoring Auswahl

Im Rahmen der Bachelor-Arbeit müssen Refactoring gewählt werden, um mit diesen die Architektur anhand eines Prototypen zu demonstrieren. Die Refactorings müssen bestimmte Eigenschaften erfüllen um in die Auswahl zu gelangen. Alle Refactorings werden aus dem Buch von Martin Fowler [Fow00] gewählt.
Der wichtigste Eigenschaft ist, dass die Refactorings automatisch ablaufen müssen um diese mit möglichst wenig Benutzerinteraktionen implementiert zu können. Die restlichen Eigenschaften werden in den jeweiligen Unterkapitel beschrieben.
Es werden insgesamt vier Refactorings gewählt. Drei davon werden sofort implementiert und in die Architektur aufgenommen. Der letzte wird zur Demonstration der Erweiterbarkeit der Architektur verwendet. Die ersten drei Refactorings bestehen aus drei Bad Smells und vier Restructurings, da der dritte Refactoring zwei Restructurings zur Beseitigung des Bad Smells anbietet. Der vierte Refactoring besteht aus einem Bad Smell und einem Restructuring.

RenameInappropriateVariable

RenameInappropriateVariable ist das erste Refactoring. Es besteht aus dem *InappropriateVariableName*-BadSmell und dem *RenameVariable*-Restructuring.
Der *InappropriateVariableName*-BadSmell sucht nach Variablennamen, welche der festgelegten Definition nicht entsprechen. Die Definitionen werden von dem Benutzer festgelegt. Dabei gibt es drei Kategorien, die betrachtet werden müssen.
Die erste Kategorie beinhaltet schwarze Listen. Die schwarzen Listen beinhalten Variablennamen, die ungültig sind. Dazu gehören zum Beispiel die Java-Schlüsselwörter. Die schwarzen Listen sind die erste Instanz bei der Überprüfung der Variablennamen. Diese Variablen werden als ungültig markiert.
Die zweite Kategorie beinhaltet die weißen Listen. Die weißen Listen beinhalten Variablennamen, die gültig sind. Dazu gehören Variablen, die vor der Überprüfung herausgenommen werden sollen.

Die weißen Listen sind die zweite Instanz bei der Überprüfung der Variablennamen. Diese Variablen werden als gültig markiert.
Die dritte Kategorie beinhaltet Strukturüberprüfungen. Diese überprüfen die Variablennamen auf bestimmte Muster hin. Dazu gehören zum Beispiel die Variablenlängen oder Worttrennungen durch Großbuchstaben. Entspricht der Variablenname allen Regeln, so wird dieser als gültig markiert. Entspricht der Variablenname einer oder mehreren Regeln nicht, so wird dieser als ungültig markiert.
Der *RenameVariable*-Restructuring nimm eine Variable entgegen und benennt diese um. Bei der Auswahl des neuen Namen sollte darauf geachtet werden, dass dieser den festgelegten Definitionen entspricht. Ansonsten wird keine BadSmell durch dieses Restructuring beseitigt, sondern es entsteht ein neues BadSmell.

SplitMultipleLocalVariableDeclaration

SplitMultipleLocalVariableDeclaration ist das zweite Refactoring. Es besteht aus dem *MultipleLocalVarialbeDeclaration*-BadSmell und dem *SplitMultipleDeclaration*-Restructuring.
Der *MultipleLocalVarialbeDeclaration*-BadSmell sucht nach Variablen, denen mehrfach neue Werte zugewiesen werden. Der *SplitMultipleDeclaration*-Restructuring nimmt eine Variablen entgegen, der mehrfach neue Werte zugewiesen wurden. Jede dieser Zuweisungen soll durch eine neue Variable ersetzt und die nachfolgenden Verwendungen angepasst werden. Dadurch sollen nach Fowler [Fow00] die Zwischenschritte durch den neuen Variablennamen dokumentiert werden. Gleichzeitig können die einzelnen Zwischenschritte einzeln weiter verwendet werden.

ResolveWasteLocalVariable

ResolveWasteLocalVariable ist das dritte Refactoring. Es besteht aus dem *WasteLocalVariable*-BadSmell, dem *ResolveLocalVariable*-Restructuring und dem *ReplaceLocalVariableWithMethod*-Restructuring. Dieser Refactoring soll demonstrieren, dass es mehrere Möglichkeiten zur BadSmell-Beseitigung existieren können. Desweiteren haben beide Restructurings eine gewissen Überschneidung in der Funktionalität. Es soll untersucht werden, ob sich dieser Aspekt positive auf die Entwicklungszeit auswirkt.
WasteLocalVariable-BadSmell sucht nach lokalen Variablen denen einmal ein Wert zugewiesen wurde und diese Variable anschließend als Platzhalten für diesen Wert im Quelltext verwendet wird.
Der *ResolveLocalVariable*-Restructuring nimmt die lokale Platzhaltervariable und setzt den ihr zugewiesenen Wert überall an ihrer Stelle ein. Die eigentliche Variable wird danach nirgendwo mehr verwendet und kann gelöscht werden.
Der *ReplaceLocalVariableWithMethod*-Restructuring nimmt die lokale Platzhaltervariable entgegen. Der zugewiesene Wert wird in eine eigene Methode verschoben und alle Verwendungen der Variable werden durch den Methodenaufruf ersetzt. Die eigentliche Variable wird danach nirgendwo mehr verwendet und kann gelöscht werden. Bei diesem Restructuring muss dynamisch eine neue Methode erstellt und verwendet werden. Damit kann eine größere Erweiterung des Quelltextes demonstriert werden.

DeleteUnusedVariable

DeleteUnusedVariable ist das vierte Refactoring. Es besteht aus dem *UnusedVariable*-BadSmell und dem *DeleteVariable*-Restructuring. Dieses Refactoring wird verwendet, um zu demonstrieren wie

die bestehende Architektur, um einen weiteren Refactoring erweitert werden kann. Deswegen muss dieses Refactoring keine besonderen Eigenschaften aufweisen, da dieser ein anderes Ziel erfüllt.
Der *UnusedVariable*-BadSmell sucht nach Variablen, die nirgendwo im Quelltext verwendet werden. Dabei mach es kein Unterschied ob diesen ein Wert zugewiesen wurde oder nicht.
Der *DeleteVariable*-Restructuring nimmt ungenutzte Variable entgegen und entfernt diese aus dem Quelltext.

2.3 TGraph

TGraphen ist eine Graphentechnologie, die an der Universität Koblenz von der Arbeitsgruppe Ebert entwickelt wurde [Bil06]. Dabei handelt es sich um eine spezielle Art von Graphen mit besonderen Eigenschaften [EBF+]. Die wichtigsten Eigenschaften, die für das Erstellen der Bachelor-Arbeit benötigt werden, werden in diesem Kapitel beschrieben.
Beim TGraphen handelt es sich um einen gerichteten Graphen. Das bedeutet, dass alle Kanten einen Start- und einen Zielknoten haben. In TGraphen wird, ausgehend von der Kante, der Startknoten als *Alpha* und der Zielknoten als *Omega* bezeichnet.
Die Kanten und Knoten in einem TGrapen haben unterschiedliche Typen. Diese Typen werden in einem Schema definiert. Bezogen auf die Quelltextrepräsentation mit TGrapen bedeutet es, dass Kanten mit bestimmten Typen für Klassen, Methoden, Variablen etc. und Kanten mit bestimmten Typen für Zuweisungen, Rechenoperationen etc. stehen.
Die Kanten und Knoten in einem TGraphen haben Attribute. Die Anzahl und die Art des Attributes ist von dem jeweiligen Objekt abhängig. Ein Knoten, der eine Variable repräsentiert, kann zum Beispiel einen Namen, einen Typ und die Zugehörigkeit zur einer Methode haben.
Die ausgehenden Kanten aus einem Knoten sind geordnet. Die Kanten haben eine bestimmte Reihenfolge in der sie einen Knoten verlassen und diese Reihenfolge wird beim Auswerten des TGraphen beachtet. Zum Beispiel kann das Erstellen von zwei Variablen in einer Klasse vertauscht werden, indem die Kanten, die das Erstellen der Variablen repräsentieren, in ihrer Reihenfolge vertauscht werden.
Zwei Knoten können durch mehr als eine Kante miteinander verbunden sein. Somit kann ein Knoten, ausgehen aus einem anderen Knoten, auf verschiedenen Pfaden erreicht werden.
Jeder Knoten und jede Kante haben eine eindeutige Identifikationsnummer, über die sie in dem TGraphen gefunden werden können.

2.3.1 JGraLab

Zu dem Erstellen und dem Verändern von TGraphen wird die Java-Bibliothek *JGraLab* angeboten [Bar09]. Die JGraLab bietet die Möglichkeiten ein eigenes TGraphen-Schema und damit einen neuen TGraphen zu erstellen. Johannes Meier bietet in seiner Arbeit [Mei13] ein überschaubares Beispiel, bei dem er zunächst ein Schema und mit diesem dann einen TGraphen mit JGraLab erstellt. Sein Beispiel bietet einen guten Einstieg in den Umgang mit TGraphen unter Verwendung der JGraLab-Bibliothek.
In dieser Bachelor-Arbeit muss keine neuer TGraphen erstellt werden. Es wird bereits ein fertiger TGraphen aus den übergebenen Quelltexten generiert. Dieser TGraphen muss mit Hilfe der JGraLab-Bibliothek verändert werden. Es müssen Knoten und Kanten erstellt, gelöscht oder ihre Attribute verändert werden. In der Abbildung 2.7 wird ein Ausschnitt aus dem *SplitMultipleDeclaration*-

Restructuring gezeigt, bei dem eine neue Variable erstellt und an die richtige Stellen im Quelltext eingehängt wird.
Zum erstellen einer Variable muss ein Knoten von dem Typ *DataObject* erstellt werden (Zeile 1 Abb. 2.7). Der Knoten wird über den Graphen erstellt. Dadurch bekommt er direkt eine eindeutige Identifikationsnummer. Nach dem Erstellen müssen die Attribute des Knotens gesetzt werden. In Zeile 2 und 3 wird dem Knoten ein Name gegeben. Dieser wird im Quelltext als Variablenname angezeigt. In Zeile 4 wird die Options gesetzt. Diese wird aus der Variable genommen, die zerteilt werden soll. Dadurch behält die neue Variable, die Eigenschaften von der alten Variable. Über die Methode *setAttribute* können Attribute auch nachträglich bearbeitet werden.
Nachdem die neue Variable vollständig erstellt wurde, wird diese als Knoten und nicht als Objekt benötigt. Der einfachste Weg dies zu erreichen, ist den Knoten über seine Identifikationsnummer aus dem TGraphen herauszubekommen (Zeile 6 Abb. 2.7). Als nächstes muss der neue Knoten über eine Kante an einen bereits bestimmten Knoten angehängt werden. Dazu wird über dem Graphen eine Kante erstellt. Als Parameter werden der Kanten-Typ, in diesem Fall ist es *DeclarationDeclares*, der Startknoten und der Zielknoten übergeben (Zeile 8). Dabei erstellt der TGraph automatisch die Kante und hängt diese mit der richtigen Ausrichtung zwischen die beiden Knoten.

```
1    DataObject dataObject = graph.createVertex(DataObject.class);
2    dataObject.setAttribute("fullyQualifiedName", newVariableName);
3    dataObject.setAttribute("name", newVariableName);
4    dataObject.setAttribute("options", variableVertex.getAttribute(
        "options"));
5
6    Vertex dataObjectVertex = graph.getVertex(dataObject.getId());
7
8    graph.createEdge(DeclarationDeclares.class, declarationVertex,
        dataObjectVertex);
```

Abbildung 2.7: Variablenerstellung in JGraLab

2.3.2 GreQL

GreQL ist eine Abfragensprache, die auf die TGraphen angewendet werden kann. Dadurch können bestimmte Knoten, Kanten oder ganze Pfade gefunden und ausgegeben werden. GreQL-Abfragen können mit der JGraLab auf den TGraphen ausgeführt werden. Da GreQL-Abfragen die Basis für die BadSmell-Erkennung darstellen, wird als Beispiel die Abfrage aus dem *UnusedVariable*-BadSmell genommen. Dabei werden Variablen gesucht, die im Quelltext nicht verwendet werden. Diese Abfrage ist nicht sehr groß bietet aber einen guten Überblick über die Möglichkeiten von GreQL. Die GreQL-Abfrage wird in der Abbildung 2.8 abgebildet.
Jede GreQl-Abfrage besteht aus vier Klauseln.
Die erste Klausel ist die *from*-Klausel (Zeile 1). In ihr kann ein Suchbereich aufgespannt werden. Bei dieser GreQl-Abfrage, werden Kanten von dem Typ *DeclarationDaclares* benötigt (Zeile 2). Diese werden in der Variable *i* zwischengespeichert.

Die zweite Klausel ist die *with*-Klausel (Zeile 3). In ihr kann der Suchbereich eingegrenzt werden. Die einzelnen Bedingungen können mit *and* oder *or* miteinander verbunden werden. Die erste Bedingung überprüft die Anzahl von eingehenden Kanten von dem Typ *HasDataObject*, die in den Knoten eingehen, auf den die i-Kanten zeigen. Die Anzahl der eingehenden Kanten muss null sein, da sie die Anzahl der Verwendungen der Variable symbolisiert. Die zweite Bedingung ist, dass der Knoten auf den die i-Kanten zeigen, genau eine Option haben darf (Zeile 5). Damit werden alle Variablen gefunden.
Die dritte Klausel ist die *report*-Klausel (Zeile 6). In ihr wird definiert, welche Werte ausgegeben werden sollen. In diesem Fall sollen alle Knoten ausgegeben werden, auf die die i-Kanten zeigen (Zeile 7). Da die Anzahl der i-Kanten durch die *with*-Klausel verkleinert wurde, werden nur Variablen ohne Verwendung zurückgegeben.
Die vierte Klausel ist die *end*-Klausel (Zeile 8). Diese symbolisiert das Ende der GreQl-Abfrage.

```
1   from
2       i:E{frontend.java.DeclarationDeclares}
3       with
4           inDegree{frontend.java.HasDataObject}(endVertex(i)) = 0
5       and count(endVertex(i).options) = 1
6   report
7       endVertex(i)
8   end
```

Abbildung 2.8: Variablensuche mit GreQL

3 Anforderungen

In diesem Kapitel werden die Anforderungen an das Rafactoring-Tool definiert. Diese umfassen die Architektur, die einzelnen Services und die verwendeten Techniken. Die Anforderungen geben die Rahmenbedingungen für die Entwicklung vor. Am Ende der Fertigstellung lässt sich dadurch das Ergebnis leichter auf fehlende Funktionalitäten überprüfen. Es werden zunächst die Technischen Anforderungen 3.1, danach die funktionalen Anforderungen3.2 und abschließend die nichtfunktionalen Anforderungen3.3 vorgestellt.

3.1 Technische Anforderungen

Die technischen Anforderungen legen fest, welche Software zur Erstellung des Prototypen verwendet werden muss. Sie werden im Folgenden mit „TA“ abgekürzt. Dabei werden auch Programme aufgelistet, die zur Realisierung verwendet werden können. Für sie sind Nutzungslizenzen zur Verfügung gestellt.
Das Ziel der Bachelor-Arbeit ist es zu Untersuchen, wie Refactorings Servicebasiert realisiert werden können. Daher muss die Architektur Service-Basiert entworfen werden.

TA01 *- SOA muss zur Erstellung der Architektur verwendet werden.*

Johannes Meier hat in seiner Bacher-Arbeit [Mei12] einige Implementierung von SOA getestet. Bei seiner Untersuchung, hat er festgestellt, dass SCA sich für Realisierung von große Projekten eignet. Darüber hinaus gibt es auch Frameworks, welche die Implementierung von SCA unterstützen. Eines von diesen Frameworks ist das IBM RSA Websphere, für welches auch eine Nutzungslizenz zur Verfügung gestellt wird.

TA02 *- SCA muss zur Service Implementierung verwendet werden.*

TA03 *- IBM RSA Websphere muss als Framework für SCA verwendet werden.*

Für die Realisierung des Refactoring-Tools wird von der Firma „pro et con“[Kai] ein Parser zur Verfügung gestellt. Dieser kann Java zu TGraphen und TGraphen zu Java parsen. Durch den Parser werden einige Techniken festgelegt und müssen daraufhin verwendet werden.

TA04 *- TGraphen müssen zur Quelltext-Repräsentation verwendet werden.*

TA05 *- SOAMIG-Schema muss als TGraph-Schema verwendet werden.*

TA06 *- JGraLab muss zur TGraph-Manipulation verwendet werden.*

TA07 *- GreQL muss als Abfrage-Sprache für die TGraphen verwendet werden.*

3.2 Funktionale Anforderungen

Die funktionalen Anforderungen legen fest, welche Funktionalität der zu entwickelnde Prototyp beinhalten muss. Sie werden im Folgenden mit „FA“ abgekürzt. Da es sich bei dem Prototypen um ein Refactoring-Tool handelt, muss dieser auch die Funktionalität von den wesentlichen Bestandteilen beinhalten.

FA01 *- Das Refactoring-Tool muss eine BadSmell-Erkennung durchführen können.*

FA02 *- Das Refactoring-Tool muss ein Restructuring durchführen können.*

Da der Fokus bei der Realisierung von dem Reafactoring-Tool auf der Erweiterbarkeit liegt, muss dieses so generische gebaut sein, dass er weitere BadSmells und Refactorings aufnehmen kann, um so seine Funktionalität zu erweitern. Die Architektur darf eine bestimmte Anzahl nicht vorschreiben.

FA03 *- Das Refactoring-Tool muss beliebig viele BadSmells beinhalten können.*

FA04 *- Das Refactoring-Tool muss beliebig viele Restructurings beinhalten können.*

Das Refactoring-Tool kann unbegrenzt viele BadSmells und Restructurings beinhalten. Daher soll eine Verknüpfung zwischen diesen hergestellt werden. Die Verknüpfung soll als Hilfestellung für die Orchestrierung und die Verwendung durch den Nutzer dienen.

FA05 *- Restructurings müssen mit BadSmells zu Refactorings verknüpft werden können.*

Da die BadSmells und Restructurings jederzeit hinzugefügt oder entfernt werden können, muss die Möglichkeit gegeben werden die Verknüpfungen zu aktualisieren.

FA06 *- Neue Refactoring-Verknüpfungen müssen hinzugefügt werden können.*

FA07 *- Alte Refactoring-Verknüpfungen müssen gelöscht werden können.*

Das Refactoring-Tool arbeitet mit Quelltext-Dateien. Aus diesem Grund benötigt er Quelltext-Dateien, um den Refactoring-Prozess zu starten. Am Ende des Prozesses werden ebenfalls Quelltext-Dateien zurückgeliefert.

FA08 *- Das Refactoring-Tool muss Quellcode-Dateien als Eingabe akzeptieren.*

FA09 *- Das Refactoring-Tool muss Quellcode-Dateien ausgeben können.*

Das Refactoring-Tool benötigt dynamische Ein- und Ausgaben, damit dieser von dem Nutzer bedient werden kann. Bei bestimmten Funktionen muss das Tool auch Informationen von dem Nutzer anfordern können. Da es sich bei dem Refactoring-Tool um einen Prototypen handelt, muss die Schnittstelle zum Nutzer nur die nötigste Funktionalität realisieren.

FA10 *- Das Refactoring-Tool muss Informationen zur Laufzeit anfordern können.*

FA11 *- Das Refactoring-Tool muss die Eingabe zusätzlicher Informationen zur Laufzeit ermöglichen.*

FA12 *- Das Refactoring-Tool muss Informationen zur Laufzeit ausgeben können.*

3.3 Nichtfunktionale Anforderungen

Die nichtfunktionalen Anforderungen legen fest, welche Eigenschaften der zu entwickelnde Prototyp beinhalten muss. Sie werden im Folgenden mit „NA“ abgekürzt. Da ein Refactoring-Tool von Menschen benutzt wird, müssen einige Anforderungen an die Benutzerfreundlichkeit gestellt werden. Diese sollen sicherstellen, dass im Prototypen Grundlagen für eine simple Benutzung angelegt werden.
Da das Refactoring-Tool erweiterbar entwickelt werden soll, können sich die einzelnen BadSmells und Restructurings ändern. Es können neue dazukommen oder ältere entfernt werden. Der Nutzer soll das Tools bedienen können, ohne zu wissen ob etwas an den Bestandteilen verändern wurde.

NA01 *- Das Refactoring-Tool muss ohne die Kenntnis der Bestandteile des Tools benutzt werden können.*

Das Refactoring-Tool soll im Nachhinein um weitere BadSmells und Restructurings erweitert werden können. Dabei soll das Hinzufügen von weiteren BadSmells und Restructurings ohne Änderungen an dem existierenden Quelltext passieren.

NA02 *- BadSmells müssen ohne Änderungen im Quelltext des Refactoring-Tools hinzugefügt werden können.*

NA03 *- Restructurings müssen ohne Änderungen im Quelltext des Refactoring-Tools hinzugefügt werden können.*

Die einzelnen Services sollen ebenfalls von anderen Tools genutzt werden können. Um dies zu ermöglichen, müssen sie einzeln weiterverwendet werden können. Dafür dürfen die Services nicht mit anderen Services in Abhängigkeit stehen. Andernfalls müssen diese mit übernommen werden, was die Wiederverwendbarkeit enorm erschwert.

NA04 *- Die Services dürfen untereinander keine Abhängigkeiten aufweisen.*

4 Architektur

In diesem Kapitel wird die Herleitung und der Aufbau der Architektur vorgestellt. Zunächst wird anhand eines Aktivitätsdiagramms Abb.4.1 der Ablauf eines Refactorings analysiert. Daraus können die ersten Services und deren Ablauf entnommen werden. Danach wird die Architektur in einer abstrahierten Form vorgestellt, um die wichtigsten Services einzuführen.
Bei allen Diagrammen in diesem Kapitel wird auf das Darstellen von Objekten zur Gunsten der Übersichtlichkeit verzichtet. Die Objekte tauchen aber bei den Beschreibungen einzelner Services auf.

4.1 Refactoring Analyse

Bevor mit der Erstellung der Architektur für das Refactoring-Tool begonnen werden kann, muss analysiert werden, was während eines Refactoring-Durchlaufs passiert. Das Aktivitätsdiagramm Abb.4.1 stellt den Prozess in einer vereinfachten Form da. Aus diesem Diagramm lassen sich die ersten Services und auch die ablaufbezogene Kombination von Services, genannt Orchestrierung, erkennen.

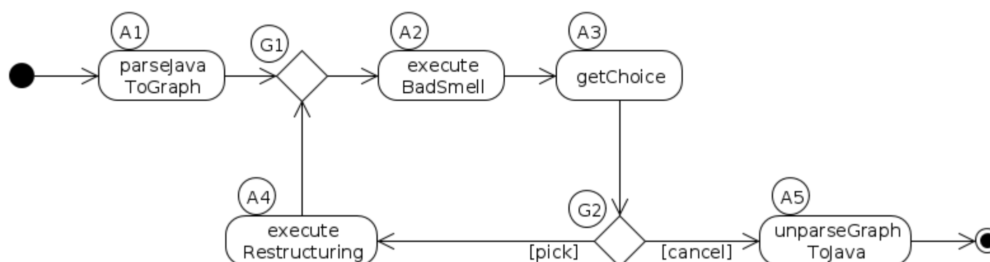

Abbildung 4.1: Refactoring Orchestrierung

Zu Beginn benötigt das Refactoring-Tool Java-Code, welcher zu einem Graphen geparsed wird. Somit steht der erste Service „parseJavaToGraph“ (A1 Abb.4.1) fest. Der geparste Graph wird an den zweiten Service „executeBadSmells“ (A2) übergeben. Dieser führt die BadSmell-Erkennung durch und liefert alle gefundenen BadSmells. Die gefundenen BadSmells werden an den dritten Service „getChoice“ (A3) weitergeleitet. Dieser ermöglicht das Auswählen eines BadSmell, um diesen zu beseitigen oder den Vorgang abzubrechen (G2). Wenn ein BadSmell gewählt wurde, so wird dieser an den vierten Service „executeRestructurings“ (A4) weitergegeben. Dieser beseitigt den BadSmell und übergibt den veränderten Graphen an den „executeBadSmells“-Service (A2) um zu überprüfen in welchem Zustand sich der neue Graph befinden und welche BadSmells sich darin befinden.
Der Ablauf, beginnend mit dem Finden der BadSmells, über die Auswahl eines BadSmells, bis hin zu der Beseitigung des BadSmells kann beliebig oft wiederholt werden. Er wird erst beendet, wenn der Benutzer sich bei der Auswahl entscheidet den Prozess abzubrechen (G2) oder keine BadSmells mehr existieren. In beiden Fällen wird der aktuelle Graph an den fünften Service „unparseGraphToJava“ (A5) übergeben. Dieser unparsed den Graphen und liefert den veränderten Java-Code.
Mit Hilfe dieser Analyse sind fünf Services mit der dazugehörigen Orchestrierung gefunden worden. Die Orchestrierung wird an dieser Stelle nicht weiter behandelt oder aufgelistet. Dies geschieht im folgendem Kapitel 4.2. Die fünf Services werden zur Übersicht mit allen Details aufgelistet:

- parseJavaToGraph

Beschreibung: Parsed den übergebenen Java-Code und liefert einen Graphen mit der Code-Repräsentation.
Eingabe: Java-Code
Ausgabe: Graph
Eigenschaft: -

- executeBadSmells

Beschreibung: Führt die BadSmell-Erkennung auf den übergebenen Graphen aus und liefert alle gefundenen BadSmells.
Eingabe: Graph
Ausgabe: BadSmells[]
Eigenschaft: -

- getChoice

Beschreibung: Ermöglicht eine Auswahl aus den übergebenen BadSmells.
Eingabe: BadSmells[]
Ausgabe: Integer
Eigenschaft: -

- executeRestructurings

Beschreibung: Beseitigt den übergebenen BadSmell durch Graph-Manipulation und liefert den überarbeiteten Graphen.
Eingabe: Graph, BadSmell
Ausgabe: Graph
Eigenschaft: -

- unparseGraphToJava

Beschreibung: Unparsed den übergebenen Graphen und liefert den daraus resultierenden Java-Code.
Eingabe: Graph
Ausgabe: Java-Code
Eigenschaft: -

Diese fünf Services werden als Grundlage für die Erstellung der Architektur im folgendem Kapitel 4.2 verwendet.

4.2 Vereinfachte Architektur

Die gefundenen Services und ihre Orchestrierung aus dem Kapitel 4.1 werden an dieser Stelle zu der ersten, vereinfachten Version der Architektur überführt. Das Ergebnis wird mit Hilfe eines Diagramms 4.2 abgebildet. Zur Darstellung wird das SAM-Diagramm aus dem Kapitel 2.1.1 verwendet. Zusätzlich werden in Abb.4.2 Services mit Orchestrierung grau hinterlegt, um diese schneller zu erkennen.

Zur Erstellung der Architektur muss zunächst die Orchestrierung, welche den gesamten Refactoring-

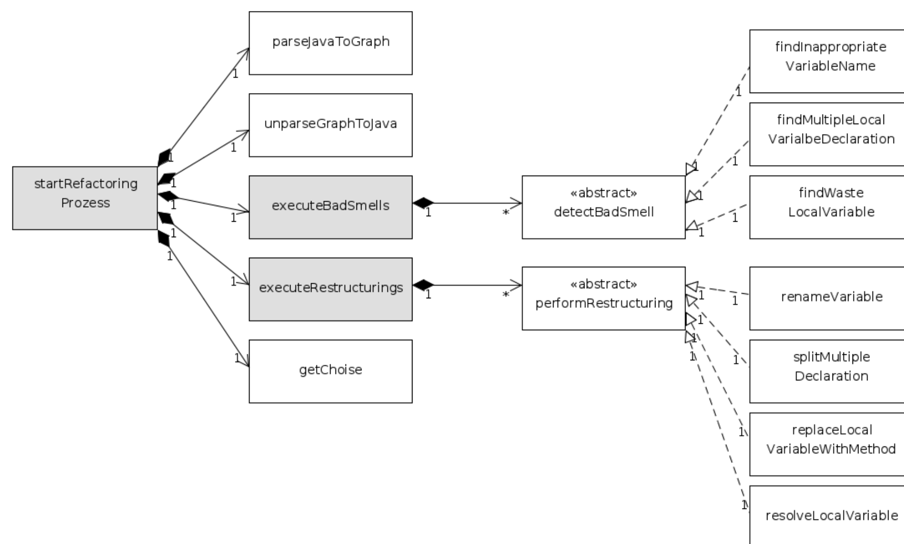

Abbildung 4.2: Vereinfachte Architektur

Ablauf steuert, in einen eigenen Service gekapselt werden. Da dieser Service den gesamten Refactoring-Prozess steuert, wird er „startRefactoringProzess“ (Abb.4.2 links) genannt. Als Eingabe benötigt er Java-Code und als Ausgabe gibt er den veränderten Java-Code wieder. Da dieser Service nur den Ablauf steuert, benötigt er Referenzen auf die fünf Services aus dem Kapitel 4.1, um die nötigen Aufgaben in der richtigen Reihenfolge zu erfüllen.
Als nächstes werden die Services „executeBadSmells“ und „executeRestructurings“ (Abb.4.2) genauer analysiert und weiter verfeinert.
Der Service „executeBadSmells“ liefert als Ergebnis alle gefundene BadSmells aus dem übergebenen Graphen. Somit besteht dieser aus mehreren BadSmell-Einheiten, welche nacheinander ausgeführt werden. Die Ergebnisse werden daraufhin gebündelt und ausgegeben.
Der Service „executeRestructurings“ führt ein Restructuring durch, um den übergebenen BadSmell zu beseitigen. Somit besteht dieser ebenfalls aus mehreren Restructuring-Einheiten, aus denen der richtige gewählt und ausgeführt wird.
Somit beinhalten beide Services ebenfalls eine Orchestrierung, welche die angebundenen Einheiten ausführt und verwaltet. Ausgehen von dieser Erkenntnis gibt es mehrere „detectBadSmell“-Services, welche ein bestimmtes BadSmell erkennen und mehrere „performRestructuring“-Services, welche ein bestimmtes BadSmell beseitigen.
Wie in Abb.4.2 zu sehen, sind die Services „findInappropriateVariableName“, „findMultipleLocalVarialbeDeclaration“ und „findWasteLocalVariable“ an den „detectBadSmell“-Service gebunden. Dadurch hat der Service „executeBadSmells“ den Zugriff auf alle an ihn angebundenen Services und kann diese über den abstrakten Service ausführen. Jetzt stellt sich aber die Frage nach der Unterscheidung der „detectBadSmell“-Services durch den „executeBadSmells“ -Service, da alle drei „detectBadSmell“-Services über den selben abstrakten Service angesprochen werden. Bisher hatten alle Services im Kapitel 4 immer eine gleichnamig Methode, wodurch der Service mit der Methode gleichzusetzen war. Bei diesem Vorgehen können die drei „detectBadSmell“-Services ausgeführt, aber nicht voneinander unterschieden werden. Deswegen muss der „detectBadSmell“-Service um eine zweite Methode (z.B. „getName“) erweitert werden, mit der sich die Services identifi-

zieren können. Mit dieser Erweiterung kann das Problem behoben werden, aber nur wenn jeder „detectBadSmell“-Service einen einzigartigen Namen liefert. Ansonsten bleibt das Identifikationsproblem weiter bestehen. Das beschriebene Problem und die dazugehörige Lösung betrifft äquivalent auch den „executeRestructurings“-Service.
Um die Übersicht in dem Diagramm 4.2 zu verbessern, werden die Services ihrer Funktion entsprechend angeordnet. Die Anordnung der Services mit ihren Referenzen erfolgt von links nach rechts. Dadurch bilden sich Ebenen, welche die einzelnen Verschachtelungen deutlich abbilden. Da die Services „parseJavaToGraph“ und „unparseGraphToJava“ sich mit dem Parsen beschäftigen und nur Daten für den eigentlich Refactoring-Prozess aufbereiten, werden beide in dem Diagramm 4.2 oben positioniert. Die Services „executeBadSmells“ und „executeRestructurings“ realisieren die Kernfunktionalität des Refactoring-Tools und werden daher mittig positioniert. Der Service „getChoice“ wird an die verbleibende, freie Stelle unten positioniert.
Mit Hilfe dieser Analyse ist ein weiterer Services entstanden. Darüber hinaus werden im Laufe dieser Bachelor-Arbeit drei „detectBadSmell“-Service und vier„performRestructuring“-Services realisiert. Insgesamt sind also acht weitere Services dazugekommen:

- startRefactoringProzess

Beschreibung:	Startet den Refactoring-Prozess und steuert den dafür nötigen Ablauf. Benötigt Java-Code und liefert am Ende den veränderten Java-Code.
Eingabe:	Java-Code
Ausgabe:	Java-Code
Eigenschaft:	-

- findInappropriateVariableName

Beschreibung:	Findet und liefert alle Variablen, dessen Namen nicht den definierten Anforderungen entsprechen.
Eingabe:	Graph
Ausgabe:	BadSmell
Eigenschaft:	Selbstauskunft

- findMultipleLocalVarialbeDeclaration

Beschreibung:	Findet und liefert alle lokale Variablen, denen mehrfach neue Werte zugewiesen wurden.
Eingabe:	Graph
Ausgabe:	BadSmell
Eigenschaft:	Selbstauskunft

- findWasteLocalVariable

Beschreibung:	Findet und liefert alle lokale Variablen, die eingespart und deren Inhalt besser integriert oder ausgelagert werden kann.
Eingabe:	Graph
Ausgabe:	BadSmell
Eigenschaft:	Selbstauskunft

- renameVariable

Beschreibung:	Benennt die übergebene Variable neu und liefert den veränderten Graph

	zurück.
Eingabe:	Graph, BadSmell
Ausgabe:	Graph
Eigenschaft:	Selbstauskunft

- splitMultipleDeclaration

Beschreibung:	Zerteilt die Mehrfachzuweisungen der übergebenen Variable und liefert den veränderten Graph zurück.
Eingabe:	Graph, BadSmell
Ausgabe:	Graph
Eigenschaft:	Selbstauskunft

- replaceLocalVariableWithMethod

Beschreibung:	Ersetzt die übergebene Variable mit einem äquivalenten Methodenaufruf und liefert den veränderten Graph zurück.
Eingabe:	Graph, BadSmell
Ausgabe:	Graph
Eigenschaft:	Selbstauskunft

- resolveLocalVariable

Beschreibung:	Löst die übergebene Variable auf und integriert ihren Inhalt. Liefert den veränderten Graph zurück.
Eingabe:	Graph, BadSmell
Ausgabe:	Graph
Eigenschaft:	Selbstauskunft

Die in diesem Kapitel 4.2 definierte Architektur bildet das Grundgerüst der vollständigen Architektur. In dem folgenden Kapitel 5 wird mit Hilfe der definierten Anforderungen im Kapitel 3 das Grundgerüst weiter verfeinert.

5 Erweiterte Architektur mit Anforderungen

Bisher basierte die Erstellung der Architektur auf der Analyse des Refactoring-Prozesses 4.1. An dieser Stelle werden darüber hinaus die definierten Anforderungen (Kapitel 3) hinzugefügt. Da die Architektur servicebasiert erstellt wird, ist die erste Anforderung bereits erfüllt, die TA01 3.1 „SOA muss zur Erstellung der Architektur verwendet werden.“ Durch die Anforderungen werden Techniken vorgeschrieben: somit ist die Architektur nicht mehr abstrakt und es können die Ein- und Ausgänge der Services genau definiert werden.
Um beim Vorstellen der erweiterten Architektur die Übersicht zu wahren, wird diese ausschnittsweise abgebildet. Als Ursprung jeden Ausschnitts wird einer der fünf Services aus dem Abschnitt 4.1 genommen, auf den „startRefactoringProzess“-Service referenziert. Diese werden grau hinterlegt, da sie ab jetzt überwiegend für die Orchestrierung verantwortlich sind und die Funktionen aus den eigenen Referenzen holen. Da in diesem Kapitel sehr viele neue Services definiert werden, werden diese nicht am Ende des Kapitels aufgelistet. Dafür wird ein eigenes Kapitel 6 erstellt, in dem alle Services alphabetisch geordnet aufgelistet werden.

5.1 parseJavaToGraph

Da für die Realisierung des Prototypen von der Firma „pro et con“[Kai] bereits ein Parser zur Verfügung gestellt wird, muss der „praseJavaToGraph“-Service erweitert werden, damit die TA04 Kapitel 3.1: „TGraphen müssen zur Quelltext-Repräsentation verwendet werden.“ erfüllt wird. Die Abbildung 5.1 bietet dazu eine grafische Übersicht.

Abbildung 5.1: parseJavaToGraph

Der Parser gibt nicht den TGraphen, sondern eine XML-Datei mit dem TGraphen aus. Daher muss der TGraph aus der XML-Datei zusätzlich eingelesen werden. Aus diesen beiden Schritten ergeben sich zwei Services: der „parseJavaToXML“ und der „importGraph“. Die beiden Services sind in der Mitte der Abbildung 5.1 zu finden. Da der Parser bekannt ist, können die Eingabeparameter und der Rückgabetyp von dem „parseJavaToGraph“-Service angepasst werden.
Als Eingabe benötigt der Parser zwei String-Arrays mit den Pfaden zu den Java-Quelltextdateien und den dazugehörigen Classdateien. Da diese Dateien für den Parser lokal vorliegen müssen, aber der Benutzer diese von seinem eigenen System zur Verfügung stellt, muss der Benutzer die Dateien hochladen und der Parser diese anschließend herunterladen. Daraus ergibt sich ein weiterer Service, der „downloadData“.
Dieser Service nimmt die Pfade zu den Dateien als String-Array entgegen und liefert die lokalen Pfade ebenfalls als String-Array zurück. Er befindet sich in der Abbildung 5.1 rechts. Damit ist auch gleichzeitig die FA08 Kapitel 3.2 „Das Refactoring-Tool muss Quellcode-Dateien als Eingabe akzep-

tieren.“ erfüllt.
Als Ausgabe wird der importierte TGraph zurückgegeben. Jeder TGraph braucht ein Schema, um die Typen der Kanten und Ecken zu bestimmen. Der Parser nutzt das SOAMIG-Schema, wodurch die TA05 als dem Kapitel 3.1 „SOAMIG-Schema muss als TGraph-Schema verwendet werden.“ erfüllt ist.

5.2 unparseGraphToJava

Für die Realisierung des Unparsers wurde von der Firma „pro et con“[Kai] bereits ein Unparser zur Verfügung gestellt. Daher wird der „unparseGraphToJava“-Service ebenfalls um weitere Services erweitert und stellt das Gegenstück zu dem „praseJavaToGraph“-Service. Die Abbildung 5.2 bietet dazu eine grafische Übersicht.

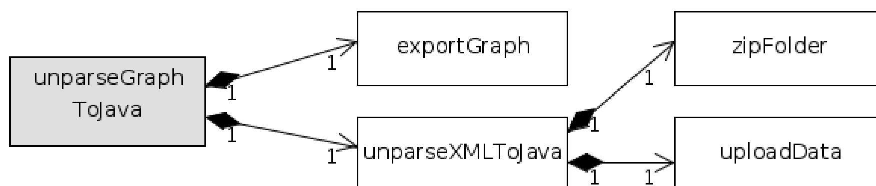

Abbildung 5.2: unparseGraphToJava

Der Unparser nimmt nicht den TGraphen, sondern eine XML-Datei mit dem TGraphen an. Daher muss der TGraph in die XML-Datei exportiert werden. Aus diesen beiden Schritten ergeben sich zwei Services: der „unparseXMLToJava“ und der „exportGraph“. Die beiden Services sind in der Mitte der Abbildung 5.2 zu finden. Da der Unparser bekannt ist, können die Eingabeparameter und der Rückgabetyp von dem „unparseGraphToJava“-Service angepasst werden.
Als Eingabe wird der TGraph genommen. Dieser wird zunächst in eine XML-Datei exportiert, damit der Unparser ihn weiter verarbeiten kann. Der Unparser generiert aus der XML-Datei die Java-Dateien mit der dazugehörigen Ordnerstruktur. Darüber hinaus liefert er noch eine Error- und eine Debug-Datei, aus denen bei Problemen die Fehlermeldungen entnommen werden können. Um diese ganzen Dateien bequem dem Nutzer zur Verfügung stellen zu können, werden sie zunächst zu einem ZIP-Archiv verpackt und anschließend hochgeladen. Daraus ergeben sich zwei weitere Services, der „zipFolder“ und der „uploadData“.
Beide befinden sich in der Abbildung 5.2 rechts. Damit ist auch gleichzeitig die FA09 3.2 Kapitel „Das Refactoring-Tool muss Quellcode-Dateien ausgeben können.“ erfüllt.
Als Ausgabe wird der Link zu dem ZIP-Archiv als String zurückgegeben. Über diesen kann das Archiv heruntergeladen werden.

5.3 executeBadSmells

Der „executeBadSmells“-Service startet die erste Hälfte des Refactoring-Prozesses. Er bietet die Möglichkeit die angebundenen „detectBadSmell“-Services einzeln auszuführen. Somit werden durch diesen Service die Anforderungen FA01 „Das Refactoring-Tool muss eine BadSmell-Erkennung durch-

führen können.“ und FA03 „Das Refactoring-Tool muss beliebig viele BadSmells beinhalten können.“ aus dem Kapitel 3.2 erfüllt. Um herauszufinden, welche Services zur Zeit zur Verfügung stehen, wird ein weiterer Service gebraucht, der „getBadSmellNames“. Dieser liefert die Namen aller angebundenen „detectBadSmell“-Services.
Als Eingabe benötigt der „executeBadSmells“-Service den aktuellen TGraphen und den Namen eines „detectBadSmell“ -Services. Es wird bewusst darauf verzichtet alle BadSmells gleichzeitig auszugeben. Dafür gibt es mehrere Gründe. Dadurch, dass die BadSmells einzeln abgerufen werden können, wird zusätzliche Funktionalität angeboten, was mehr Flexibilität bedeutet. Die angeknüpften „detectBadSmell“-Services können abgefragt werden, ohne diese zwangsweise ausführen zu müssen. Diese Eigenschaft wirkt sich positiv auf die Performance aus. Durch das ausführen einzelner „detectBadSmell“-Services ist der Rückgabetyp ein Eindimensionales-Array, was die Komplexität des Rückgabewertes minimiert. Das hat den positiven Effekt, dass der Rückgabewert anschließend leicht weiterverwendet werden kann, ohne diesen auseinander nehmen zu müssen. Der letzte Grund bezieht sich auf die Restructurings. Diese müssen einzeln ausgeführt werden. Durch das einzelne Ausführen von den BadSmells können beide Servicetypen aneinander angeglichen werden und somit gleich angesprochen werden. Das vereinfacht die Implementierung und die Wartung, da das verhalten von beiden Services sich stark ähnelt.
Als nächstes werden die einzelnen „detectBadSmell“-Services und ihre Referenzen vorgestellt. Die Abbildung 5.3 bietet dazu eine grafische Übersicht.

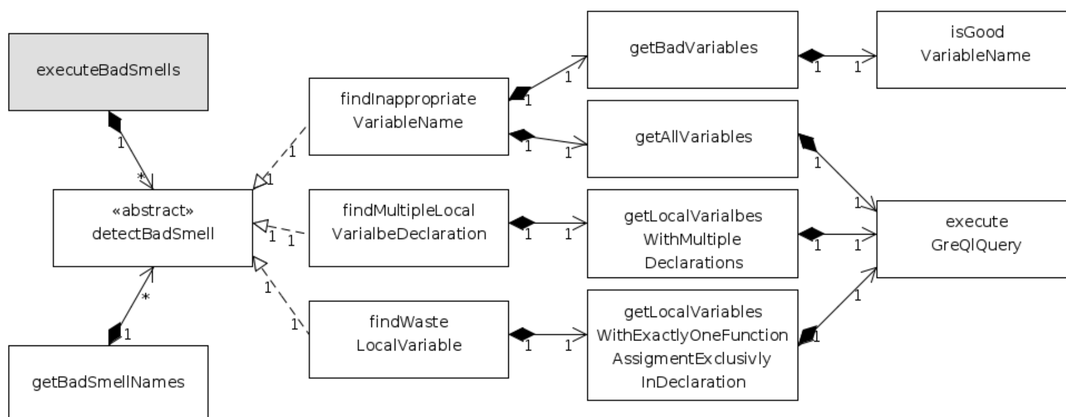

Abbildung 5.3: executeBadSmells

Die Grundlage der Realisierung in jedem „detectBadSmell“-Services bildet eine GreQl-Abfrage. Die eigentliche Abfrage wird in ein Service gekapselt und kann mit Hilfe des „executeGreQlGruery“-Service ausgeführt werden. Da bei TGraphen alle Objekte im Quelltext als Knoten repräsentiert werden und ihre Verbindungen als Kanten, können die BadSmells mit Knoten dargestellt werden. Dies ist möglich, weil die BadSmells sich immer auf ein Objekt beziehen, beispielsweise auf Klassen, Methoden oder Variablen. Dadurch ist der Rückgabetyp aller „detectBadSmell“-Services ein Array aus Knoten.
Der „findInappropriateVariableName“-Service benötigt zwei Referenzen für die Umsetzung (Abb. 5.3). Die erste Referenz benötigt den „getAllVariables“-Service. Dieser Service kapselt eine GreQl-Abfrage, mit der alle Variablen aus dem Graphen ausgelesen werden. Als nächstes werden die gefundenen Variablen an die zweite Referenz übergeben, den „getBadVariables“-Service. Hier wird der

Name jeder Variable ermittelt und einzeln an den „isGoodVariableName“-Service übergeben. Dieser überprüft den Namen auf die dort definierten Regeln und liefert ein TRUE wenn der Name den Regeln entspricht. Andernfalls wird ein FALSE zurückgegeben. Variablen mit schlechten Namen werden in dem „getBadVariables“-Service gesammelt und anschließend zurückgegeben. Somit liefert der „findInappropriateVariableName“-Service alle Variablen, deren Namen nicht den definierten Regeln entsprechen.
Der „findMultipleLocalVarialbeDeclaration“-Service benötigt eine Referenz für die Umsetzung (Abb. 5.3). Er benötigt den „getLocalVarialbesWithMultipleDeclarations“-Service. Dieser Service kapselt eine GreQl-Abfrage, welche die nötigen Variablen identifizieren kann, ohne sich weiterer Services bedienen zu müssen. Mit dieser GreQl-Abfrage können alle lokale Variablen gefunden werden, welchen mehrfach ein Wert zugewiesen wird.
Der „findWasteLocalVariable“-Service benötigt eine Referenz für die Umsetzung (Abb. 5.3). Er benötigt den „getLocalVariablesWithExactlyOneFunctionAssigmentExclusivlyInDeclaration“-Service. Dieser Service kapselt eine GreQl-Abfrage, welche die nötigen Variablen identifizieren kann, ohne sich weiterer Services bedienen zu müssen. Mit dieser GreQl-Abfrage können alle lokale Variablen gefunden werden, welche genau ausschließlich in der Deklaration eine funktionbasierte Zuweisung haben.

5.4 getChoice

Der „getChoice“-Service wird an dieser Stelle vorgezogen, da er für die Umsetzung des „execute-Restructurings“ -Services benötigt wird. Darüber hinaus wird er nicht als ein Service angesprochen, sondern in mehrere kleinere Services zerteilt. Dadurch wir der „getChoice“-Service in seiner abstrakten Form komplett entfernt. Eine Übersicht über die neuen Services bietet die Abbildung 5.4.

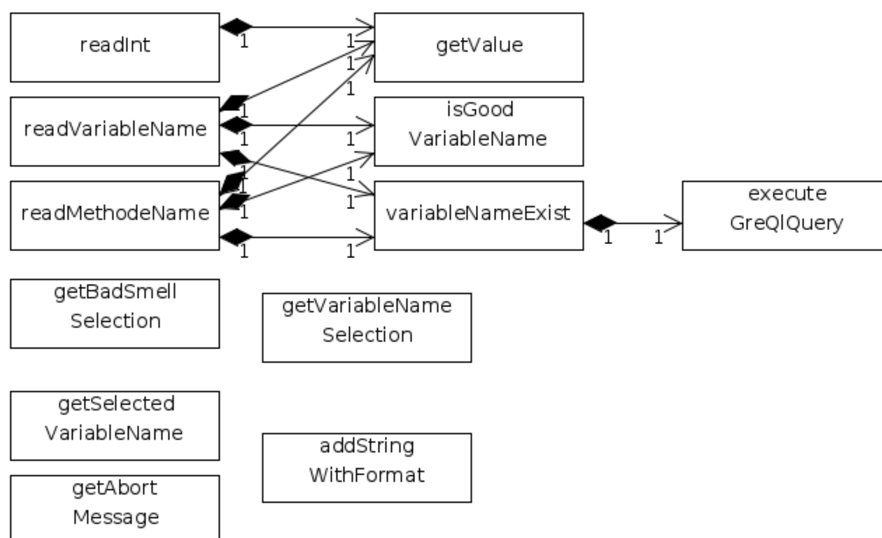

Abbildung 5.4: getChoise

Die neuen Services lassen sich in drei Kategorien einteilen. Die erste Kategorie besteht aus drei Services (Abb. 5.4 oben): „readInt“, „readVariableName“ und „readMethodeName“. Alle drei dieser Services erfüllen die FA10 Kapitel 3.2: „Das Refactoring-Tool muss Informationen zur Laufzeit anfordern können.“ Zur Realisierung der Nutzereingaben verwenden sie den „getValue“ -Service. Dieser Service zeigt die für den Nutzer übergebenen Nachrichten an und fordert diesen auf einen Wert einzugeben. Somit erfüllt der „getValue“ -Service die Anforderungen FA11 Kapitel 3.2 „Das Refactoring-Tool muss die Eingabe zusätzlicher Informationen zur Laufzeit ermöglichen.“ und FA12 Kapitel 3.2 „Das Refactoring-Tool muss Informationen zur Laufzeit ausgeben können.“ Hierbei werden die Nachrichten immer zusammen mit einer Eingabeaufforderung ausgegeben.
Nachrichten ohne Eingabeaufforderung werden gesammelt und mit der nächsten Eingabeaufforderung angezeigt. Dieser Umstand beruht auf der Implementierung und wird demzufolge im Kapitel 7 genauer beschrieben. Das beschriebene Verhalten beruht, auf der Tatsache, dass bei jedem Aufruf ein neuer Popup erstellt wird, welcher durch die Nutzereingabe geschlossen wird. Der „getValue“ -Service liefert als Rückgabewert den übergebenen Text des Nutzers als String zurück ohne zu Prüfen oder zu einem anderen Datentypen zu überführen. Die Überprüfung und die Überführung machen dieselben Services, die die Anforderung gestartet haben.
Der „readInt“-Service ermöglicht die Auswahl aus einer Menge. Als Eingabe nimmt er einen String, von dem auszugebenden Text und die Anzahl von Elementen als Integer, aus denen gewählt werden soll. Die Nutzereingabe wird zuerst zu einem Integer überführt und anschließend darauf geprüft, ob sie eines der übergebenen Elemente repräsentiert. Bei korrekter Eingabe wird die Zahl als Integer ausgegeben. Dabei wird noch ein zusätzliches Element für das Abbrechen hinzugefügt. Dabei wird der Integer „-1“ zurückgegeben. Wenn die Nutzereingabe nicht zu einem Integer überführt werden kann oder diese zu keinem Element passt, wird der Nutzer aufgefordert einen neuen Wert einzugeben.
Der „readVariableName“-Service ermöglicht die Eingabe eines neuen Variablennamen. Als Eingabe nimmt er ein String von dem auszugebenden Text und den aktuellen Graphen. Die Nutzereingabe wird zuerst mit dem „isGoodVariableName“-Service geprüft, um sicherzugehen, dass kein neuer BadSmell entsteht. Danach wird die Benutzereingabe mit dem „variableNameExist“-Service geprüft um sicher zu gehen, dass der neue Name nicht bereits vergeben ist. Dazu wird der aktuelle Graph nach der Nutzereingabe durchsucht. Wenn die Nutzereingabe den definierten Regeln nicht entspricht oder bereits vergeben ist, wird der Nutzer aufgefordert einen neuen Variablennamen einzugeben.
Der „readMethodeName“-Service ist äquivalent zum „readVariableName“-Service aufgebaut. Der einzige Unterschied ist, dass der Nutzer jedes Mal aufgefordert wird einen neuen Methodennamen anstatt eines Variablennamen einzugeben.
Es werden zwei weitere Services hinzugefügt, um die Auflistung einer Auswahl zu vereinfachen und die Darstellung einheitlich zu machen. Der „getBadSmellSelection“-Service nimmt ein Array aus „badSmellNamen“ und ein Array aus „badSmellCounts“ entgegen. Aus diesen beiden Array baut er eine Übersicht für die Ausgabe und liefert diese als String zurück. Der „getVariableNameSelection“-Service nimmt die gefundenen „badSmells“ als Vertex-Array entgegen und baut aus diesen eine Übersicht für die Ausgabe und liefert diese als String zurück. Dadurch, dass er ein Array aus Knoten nimmt, kann die Ausgabe sehr gut erweitert werden. Denn aus dem Knoten kann nicht nur der Name, sondern auch der Type, die Sichtbarkeit, die dazugehörige Methode oder die Klasse ermittelt werden.
Da alle Ausgaben auf Strings basieren, werden noch drei weitere Services hinzugefügt, um die Formatierung einheitlich zu machen und diese bei Bedarf schnell ändern zu können. Der „getSelectedVariableName“ -Service nimmt eine Vertex der eine Variable repräsentiert und liefert eine Übersicht über die gewählte Variable als String zurück. Der „getAbortMessage“-Service liefert eine Abbruchnachricht als String zurück. Der „addStringWithFormat“-Service nimmt die bisherige Nachricht als

String und eine neue Nachricht als String an. Diese beiden Nachrichten werden nach dem definierten Format verknüpft und als String zurückgegeben.
Die vorgestellten Services bietet die Grundlage um die NA01 aus dem Kapitel 3.3 „Das Refactoring-Tool muss ohne die Kenntnis der Bestandteile des Tools benutzt werden können“ zu erfüllen. Die Services ermöglichen es die wichtigsten Informationen dem Nutzer anzuzeigen und ihn somit durch den Refactoring-Prozess zu führen. Wie das Informieren des Nutzers genau umgesetzt werden kann, wird in Kapitel 7 beschrieben.

5.5 executeRestructurings

Der „executeRestructurings“-Service startet die zweite Hälfte des Refactoring-Prozesses. Er bietet die Möglichkeit die angebundenen „performRestructuring“-Services einzeln auszuführen. Somit werden durch diesen Service die Anforderungen FA02 „Das Refactoring-Tool muss ein Restructuring durchführen können.“ und FA04 „Das Refactoring-Tool muss beliebig viele Restructurings beinhalten können.“ aus dem Kapitel 3.2 erfüllt. Um herauszufinden, welche Services zur Zeit zur Verfügung stehen, wird ein weiterer Service gebraucht, der „getRestructuringNames“. Dieser liefert die Namen aller angebundenen „performRestructuring“-Services.
Als Eingabe benötigt der „executeRestructurings“-Service den aktuellen TGraphen, die gefundenen BadSmells als Array und den Namen eines „performRestructuring“-Services. Bevor es mit dem Restructurieren begonnen wird, fordert der „executeRestructurings“-Service den Nutzer auf einen BadSmell aus dem übergebenen Array auszuwählen. Dazu benötigt er den Service aus dem bereits vorgestellten Abschnitt „getChoise“. Zunächst den „getVarialbeNameSelection“-Service, um eine Übersicht über die BadSmells zu erstellen, welche dem Nutzer angezeigt werden soll. Danach den „readInt“-Service, um mit Hilfe der erstellten Übersicht ein BadSmell auszuwählen oder den Prozess abzubrechen. Anschließend geht der gewählte BadSmell und der aktuelle TGraph an den gewählten „performRestructuring“-Service, wo das Restructuring durchgeführt wird. Alle „performRestructuring“ -Services liefern den veränderten TGraphen als Ausgabe.
Als nächstes werden die einzelnen „performRestructuring“-Services und ihre Referenzen vorgestellt. Die Abbildung 5.5 bietet dazu eine grafische Übersicht. Die Services aus dem Abschnitt „getChoise“ und der Service „executeGreQlQuery“ werden aus Übersichtsgründen nicht in der Abbildung erscheinen, aber im weiteren Textverlauf erläutert.
Jeder „performRestructuring“-Services lässt sich in der Realisierung in drei Bestandteile aufteilen. Zuerst müssen alle nötigen Informationen, die für das Umsetzung nötig sind, gesammelt werden. Diese Informationen können von dem Nutzer angefordert oder aus dem Graphen ausgelesen werden. Das Auslesen von Informationen geschieht mit Hilfe der Abfragesprache GreQl. GreQl ist ein Bestandteil der Bibliothek JGraLab, mit der auch der TGraph manipuliert werden kann. Somit werden an dieser Stelle zwei Anforderungen aus dem Kapitel 3.1 erfüllt: die TA06 „JGraLab muss zur TGraph-Manipulation verwendet werden.“ und die TA07 „GreQL muss als Abfrage-Sprache für die TGraphen verwendet werden.“
Als nächstes muss der Graph um weitere Knoten und Kanten erweitert werden, welche für die Umsetzung nötig sind. Im letzten Schritt wird der bestehende Graph mit den gesammelten Informationen manipuliert und es werden die neu erstellten Knoten und Kanten eingebunden oder bestehenden umgeleitet. Dabei hängt es von der Komplexität des Restructurings ab, aus wie vielen Bestandteilen er besteht und wie oft diese durchlaufen werden müssen.

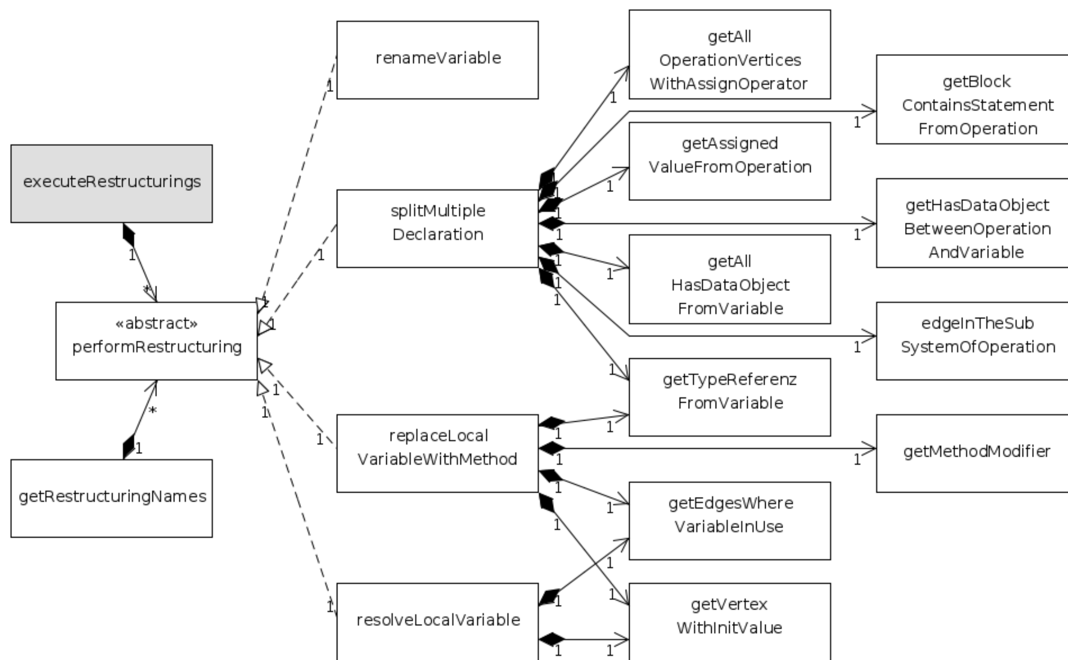

Abbildung 5.5: executeRestructurings

Der „renameVariable“-Service benötigt zwei Referenzen für die Umsetzung. Die erste Referenz benötigt den „getSelectedVariableName“-Service, um die gewählte Variable dem Nutzer anzuzeigen. Die zweite Referenz benötigt den „readVariableName“-Service, um einen neuen Variablennamen zu erhalten. Anschließend wir die Variable umbenannt.

Der „splitMultipleDeclaration“-Service benötigt zehn Referenzen für die Umsetzung. Zunächst werden drei Referenzen für die Interaktion mit dem Nutzer benötigt. Die erste Referenz benötigt den „getSelectedVariableName“-Service, um die gewählte Variable dem Nutzer anzuzeigen. Die zweite Referenz benötigt den „readVariableName“-Service, um neue Variablennamen zu erhalten. Die dritte Referenz benötigt den „addStringWithFormat“-Service, um die erstellten Nachrichten zusammenzufügen. Die weiteren sieben Referenzen kapseln GreQl-Abfragen um bestimmte Informationen aus dem TGraphen auszulesen. Die Übersicht darüber ist in der Abbildung 5.5 zu sehen.
Der „getAllOperationVerticesWithAssignOperator“-Service liefert alle Knoten, die eine Zuweisung auf die übergebene Variable durchführen. Als Eingabe benötigt er den TGraphen und die ID von der Variable.
Der „getBlockContainsStatementFromOperation“-Service liefert die Kante, die die Stelle im Code repräsentiert, an der die übergebene Zuweisung stattgefunden hat. Als Eingabe benötigt er den TGraphen und die ID von der Zuweisung.
Der „getAssignedValueFromOperation“-Service liefert den Knoten, mit dem zugewiesenen Wert von der Zuweisung. Als Eingabe benötigt er den TGraphen und die ID von der Zuweisung.
Der „getHasDataObjectBetweenOperationAndVariable“-Service liefert die Kante zwischen der Zuweisung und der Variable, welche den Start der Umleitung repräsentiert. Als Eingabe benötigt er den TGraphen, die ID der Zuweisung und die ID der Variable.
Der „getAllHasDataObjectFromVariable“-Service liefert alle Kanten, die die Verwendung der Varia-

ble repräsentieren. Als Eingabe benötigt er den TGraphen und die ID der Variable.
Der „getTypeReferenzFromVariable"-Service liefert den Knoten, der den Typen der Variable repräsentiert. Als Eingabe benötigt er den TGraphen und die ID der Variable.
Der „edgeInTheSubSystemOfOperation" -Service liefert ein Boolean in Abhängigkeit davon, ob die Kante ein Kind des Knotens ist. Dabei ist es egal, ob sie ein direktes oder ein indirektes Kind ist. Als Eingabe benötigt er den TGraphen, die Kante und den Knoten.
Nachdem alle nötigen Informationen gesammelt wurden, können die neuen Variablen mit den neuen Namen erstellt werden und anschließend alle Verweise in der richtigen Reihenfolge auf sie umgeleitet werden.

Der „replaceLocalVariableWithMethod"-Service benötigt sechs Referenzen für die Umsetzung. Zunächst werden zwei Referenzen für die Interaktion mit dem Nutzer benötigt.
Die erste Referenz benötigt den „getSelectedVariableName"-Service, um die gewählte Variable dem Nutzer anzuzeigen.
Die zweite Referenz benötigt den „readMethodeName"-Service, um einen neuen Methodennamen zu erhalten.
Die weiteren vier Referenzen kapseln GreQl-Abfragen, um bestimmte Informationen aus dem TGraphen auszulesen. Die Übersicht darüber ist in der Abbildung 5.5 zu sehen.
Der „getTypeReferenzFromVariable"-Service, der für den „splitMultipleDeclaration"-Service erstellt wurde, kann an dieser Stelle wiederverwendet werden.
Der „getMethodModifier"-Service liefert alle Attribute aus einer Klasse, die für die Erstellung einer Methode nötig sind. Als Eingabe benötigt er den TGraphen und den Klassennamen.
Der „getEdgesWhereVariableInUse"-Service liefert alle Kanten, die die Verwendung der Variable repräsentieren. Als Eingabe benötigt er den TGraphen und die ID der Variable.
Der „getVertexWithInitValue"-Service liefert den Knoten mit dem zugewiesenen Wert von der Variable. Als Eingabe benötigt er den TGraphen und die ID der Variable. Nachdem alle nötigen Informationen gesammelt wurden, kann die Methode mit dem neuen Namen und dem alten Wert der Variable erstellt werden. Anschließend werden alle Verwendungen der Variable durch den Methodenaufruf ersetzt und die Variable gelöscht.

Der „resolveLocalVariable"-Service benötigt zwei Referenzen für die Umsetzung. Die zwei Referenzen kapseln GreQl-Abfragen, um bestimmte Informationen aus dem TGraphen auszulesen. Die Übersicht darüber ist in der Abbildung 5.5 zu sehen.
Beide Services, der „getEdgesWhereVariableInUse"-Service und der „getVertexWithInitValue"-Service, die für den „replaceLocalVariableWithMethod"-Service erstellt wurden, können an dieser Stelle wiederverwendet werden. Somit kann der „resolveLocalVariable" -Service komplett aus bereits vorhandenen Services erstellt werden. Nachdem alle nötigen Informationen gesammelt wurden, werden die Verwendungen der Variable durch ihren Wert ersetzt und die Variable gelöscht.

5.6 combineRefactorings

Die letzte fehlende Funktionalität ist das Kombinieren von BadSmells und Restructurings zu Refactorings. Diese Funktionalität ist auch in den drei Anforderungen aus dem Kapitel 3.2 beschrieben. Die FA05 „Restructurings müssen mit BadSmells zu Refactorings verknüpft werden können.", die FA06 „Neue Refactoring-Verknüpfungen müssen hinzugefügt werden können." und die FA07 „Alte Refactoring-Verknüpfungen müssen gelöscht werden können." Um diese Anforderungen zu erfüllen, werden zwei weitere Services hinzugefügt.

Der „combineRefactorings“-Service ermöglicht das Verknüpfen von Restructurings mit den BadSmells, wodurch Refactorings entstehen. Mit dem „getRefactoringNames“-Service können die vorhandenen Refactoring-Kombinationen abgerufen werden. Für die Realisierung der Verknüpfungen braucht der „combineRefactorings“-Service alle Namen an ihn angehängter BadSmells und Restructurings. Diese können über die beiden Services „getBadSmellNames“ und „getRestructuringNames“ erreicht werden. Darüber hinaus benötigt er drei Services aus dem Abschnitt „getChoise“ für die Interaktion mit dem Nutzer. Zur Erstellung der Ausgabe wird der „addStringWithFormat“-Service und der „getAbortMessage“-Service benötigt. Für die Ausgabe und die Nutzereingabe wird der „readInt“-Service benötigt. Zuletzt wird noch ein „readRefactoringName“ -Service benötigt, welcher den Nutzer auffordert einen neuen Refactoringnamen einzugeben. Die Abbildung 5.6 bietet dazu eine grafische Übersicht.

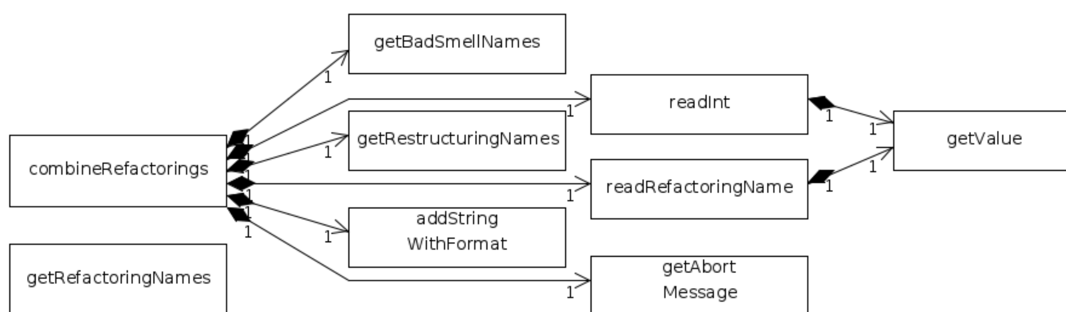

Abbildung 5.6: combineRefactorings

Die Refactoringnamen erleichtern das Ausführen von den „detectBadSmell“-Services und den „performRestructuring“ -Services. Dazu wird die Eingabe von dem „executeBadSmells“-Service von dem BadSmellName auf den RefactoringName geändert. Das selbe wird auch bei dem „execute Restructuring“-Service gemacht. Diese nehmen den RefactoringName und ermittelt mit diesem den nötigen BadSmellName um den „detectBadSmell“-Services oder den dazugehörigen RestructuringName um den „performRestructuring“-Services zu starten. Dies erleichtert die Orchestrierung. Denn nach dem Finden der BadSmells über den RefactoringNamen, kann die dazugehörige Restructurierung über den selben RefactoringName durchgeführt werden.

6 Service Katalog

Der Service Katalog bietet eine Übersicht über alle Services, die im Kapitel 5 definiert wurde. Die Services werden alphabetisch sortiert aufgelistet.

S01 - addStringWithFormat

Beschreibung: Fügt zwei Strings mit der vordefinierten Formatierung aneinander.
Eingabe: String message, String addMessage
Ausgabe: String message
Eigenschaft: -

S02 - combineRefactorings

Beschreibung: Führt den Kombinations-Prozess durch, wodurch Restructurings an BadSmells geknüpft werden.
Eingabe: -
Ausgabe: -
Eigenschaft: -

S03 - downloadData

Beschreibung: Lädt die übergebenen Dateien lokal herunter und liefert die Pfade zurück.
Eingabe: String[] dataPath
Ausgabe: String[] localDataPath
Eigenschaft: -

S04 - edgeInTheSubSystemOfOperation

Beschreibung: Überprüft ob die Edge ein (indirektes) Kind von dem Vertex ist.
Eingabe: Edge edge, Vertex operation, TGraph graph
Ausgabe: boolean child
Eigenschaft: -

S05 - executeBadSmells

Beschreibung: Führt den gewählten BadSmell-Service auf den übergebenen Graphen und liefert alle gefundenen BadSmells.
Eingabe: TGraph graph, String refactoringName
Ausgabe: Vertex[] badSmells
Eigenschaft: -

S06 - executeGreQlQuery

Beschreibung: Führt eine GreQl-Abfrage an dem TGraph durch und liefert das Ergebnis zurück.
Eingabe: TGraph graph, String greQlQuery
Ausgabe: JValue result
Eigenschaft: -

S07 - executeRestructurings

Beschreibung: Führt den gewählten Restructuring-Service an dem übergebenen Graphen aus und liefert den manipulierten Graphen zurück.
Eingabe: TGraph graph, Vertex[] badSmells, String refactoringName
Ausgabe: TGraph graph
Eigenschaft: -

S08 - exportGraph

Beschreibung: Schreibt den TGraphen in eine XML-Datei hin und liefert den Pfad zur dieser.
Eingabe: TGraph graph
Ausgabe: String xmlPath
Eigenschaft: -

S09 - findInappropriateVariableName

Beschreibung: Findet und liefert alle Variablen, deren Namen nicht den definierten Anforderungen entsprechen.
Eingabe: TGraph graph
Ausgabe: Vertex[] badSmells
Eigenschaft: Selbstauskunft

S10 - findMultipleLocalVarialbeDeclaration

Beschreibung: Findet und liefert alle lokale Variablen, denen mehrfach neue Werte zugewiesen wurden.
Eingabe: TGraph graph
Ausgabe: Vertex[] badSmells
Eigenschaft: Selbstauskunft

S11 - findWasteLocalVariable

Beschreibung: Findet und liefert alle lokale Variablen, die eingespart und deren Inhalt besser integriert oder ausgelagert werden kann.
Eingabe: TGraph graph
Ausgabe: Vertex[] badSmells
Eigenschaft: Selbstauskunft

S12 - getAbortMessage

Beschreibung: Liefert eine einheitliche Abbruchnachricht.
Eingabe: -
Ausgabe: String message
Eigenschaft: -

S13 - getAllHasDataObjectFromVariable

Beschreibung: Liefert alle HasDataObjects von der gewählten Variable.
Eingabe: TGraph graph, int variableId
Ausgabe: Edge[] hastDataObjects
Eigenschaft: -

S14 - getAllOperationVerticesWithAssignOperator

Beschreibung: Liefert alle Knoten vom Typ Operation, mit Operation = ASSIGN von der übergebenen Variable.
Eingabe: TGraph graph, int vertexId
Ausgabe: Vertex[] operations
Eigenschaft: -

S15 - getAllVariables

Beschreibung: Liefert alle Variablen des Graphen.
Eingabe: TGraph graph
Ausgabe: Vertex[] variables
Eigenschaft: -

S16 - getAssignedValueFromOperation

Beschreibung: Liefert den Knoten mit den zugewiesenen Werten der Operation.
Eingabe: TGraph graph, int operationId
Ausgabe: Vertex value
Eigenschaft: -

S17 - getBadSmellNames

Beschreibung: Liefert die Namen aller BadSmell-Services, die im Moment angehängt sind.
Eingabe: -
Ausgabe: String[] badSmellNames
Eigenschaft: -

S18 - getBadSmellSelection

Beschreibung: Baut aus den übergebenen BadSmellNamen und der dazugehörigen BadSmellAnzahl eine Übersicht für die Ausgabe und liefert diese als String zurück.
Eingabe: String[] badSmellNames, int[] badSmellCounts
Ausgabe: String message
Eigenschaft: -

S19 - getBadVariables

Beschreibung: Filtert die guten Variablen heraus und liefert die schlechten zurück.
Eingabe: Vertex[] variables
Ausgabe: Vertex[] badVariables
Eigenschaft: -

S20 - getBlockContainsStatementFromOperation

Beschreibung: Liefert die BlockContainsStatement-Kante von der übergebenen Operation.
Eingabe: TGraph graph, int operationId
Ausgabe: Edge blockContainsStatement
Eigenschaft: -

S21 - getEdgesWhereVariableInUse

Beschreibung: Liefert alle Kanten in denen die Variable verwendet wurde.
Eingabe: TGraph graph, int variableId
Ausgabe: Edge[] variableAssignments
Eigenschaft: -

S22 - getHasDataObjectBetweenOperationAndVariable

Beschreibung: Liefert den HasDataObject zwischen der Operation und der Variable.
Eingabe: TGraph graph, int operationId, int variableId
Ausgabe: Edge hasDataObject
Eigenschaft: -

S23 - getLocalVariablesWithExactlyOneFunctionAssigmentExclusivlyInDeclaration

Beschreibung: Liefert alle lokale Variablen, welche ausschließlich in der Deklaration genau eine funktionsbasierte Zuweisung haben.
Eingabe: TGraph graph
Ausgabe: Vertex[] badSmells
Eigenschaft: -

S24 - getLocalVarialbesWithMultipleDeclarations

Beschreibung: Liefert alle lokalen Variablen denen mehrfach ein Wert zugewiesen wurde.
Eingabe: TGraph graph
Ausgabe: Vertex[] badSmells
Eigenschaft: -

S25 - getMethodModifier

Beschreibung: Liefert alle nötigen Modifier aus der übergebenen Klasse zur Erstellung einer Methode.
Eingabe: TGraph graph, String className
Ausgabe: Vertex[] modifier
Eigenschaft: -

S26 - getRefactoringNames

Beschreibung: Liefert die Namen aller Refactoring-Kombinationen, die im Moment verfügbar sind.
Eingabe: -
Ausgabe: String[] refactoringNames
Eigenschaft: -

S27 - getRestructuringNames

Beschreibung: Liefert die Namen aller Restructuring-Services, die im Moment angehängt sind.
Eingabe: -
Ausgabe: String[] restructuringNames
Eigenschaft: -

S28 - getSelectedVariableName

Beschreibung: Liefert eine einheitliche Nachricht für den übergebenen Namen.
Eingabe: Vertex variable
Ausgabe: String message
Eigenschaft: -

S29 - getTypeReferenzFromVariable

Beschreibung: Liefert die TypeReferenz von der gewählten Variable.
Eingabe: TGraph graph, int variableId
Ausgabe: Vertex typeReferenz
Eigenschaft: -

S30 - getValue

Beschreibung: Zeigt die übergebene Nachricht an und fordert den Nutzer auf einen Wert einzugeben.
Eingabe: String message
Ausgabe: String value
Eigenschaft: -

S31 - getVarialbeNameSelection

Beschreibung: Baut aus den übergebenen BadSmells eine Übersicht für die Ausgabe und liefert diese als String zurück.
Eingabe: Vertex[] badSmells
Ausgabe: String message
Eigenschaft: -

S32 - getVertexWithInitValue

Beschreibung: Liefert den Knoten mit dem initialisierten Wert.
Eingabe: TGraph graph, int variableId
Ausgabe: Vertex value
Eigenschaft: -

S33 - importGraph

Beschreibung: Liest den TGraphen aus der übergebenen XML-Datei aus und gibt diesen anschließend aus.
Eingabe: String xmlPath
Ausgabe: TGraph graph
Eigenschaft: -

S34 - isGoodVariableName

Beschreibung: Überprüft den übergebenen Namen nach den definierten Regeln und entscheidet ob er diesen entspricht.
Eingabe: String name
Ausgabe: boolean goodName
Eigenschaft: -

S35 - parseJavaToGraph

Beschreibung: Parsed den übergebenen Java-Code und liefert einen Graphen mit der Code-Repräsentation.
Eingabe: String[] javaPath, String[] classPath
Ausgabe: TGraph graph
Eigenschaft: -

S36 - parseJavaToXML

Beschreibung: Parsed den übergebenen Java-Code und liefert eine XML-Datei, die den TGraphen beinhaltet.
Eingabe: String[] javaPath, String[] classPath
Ausgabe: String xmlPath
Eigenschaft: -

S37 - readInt

Beschreibung: Ermöglicht die Auswahl aus der übergebenen Menge. Es wird -1 zum Abbrechen zurück gegeben.
Eingabe: String message, int count
Ausgabe: int result
Eigenschaft: -

S38 - readMethodeName

Beschreibung: Ermöglicht die Eingabe eines neuen Methodennamens.
Eingabe: String message, TGraph graph
Ausgabe: String methodeName
Eigenschaft: -

S39 - readRefactoringName

Beschreibung: Ermöglicht die Eingabe eines neuen Refactoringnamens.
Eingabe: String message
Ausgabe: String refactoringName
Eigenschaft: -

S40 - readVariableName

Beschreibung: Ermöglicht die Eingabe eines neuen Variablennamen.
Eingabe: String message, TGraph graph
Ausgabe: String variableName
Eigenschaft: -

S41 - renameVariable

Beschreibung: Benennt die übergebene Variable neu und liefert den veränderten Graph zurück.
Eingabe: TGraph graph, Vertex[] badSmells
Ausgabe: TGraph graph
Eigenschaft: Selbstauskunft

S42 - replaceLocalVariableWithMethod

Beschreibung: Ersetzt die übergebene Variable mit einem äquivalenten Methodenaufruf und liefert den veränderten Graph zurück.
Eingabe: TGraph graph, Vertex[] badSmells
Ausgabe: TGraph graph
Eigenschaft: Selbstauskunft

S43 - resolveLocalVariable

Beschreibung: Löst die übergebene Variable auf und integriert ihren Inhalt. Liefert den veränderten Graph zurück.
Eingabe: TGraph graph, Vertex[] badSmells
Ausgabe: TGraph graph
Eigenschaft: Selbstauskunft

S44 - splitMultipleDeclaration

Beschreibung: Zerteilt die Mehrfachzuweisungen der übergebenen Variable und liefert den veränderten Graph zurück.
Eingabe: TGraph graph, Vertex[] badSmells
Ausgabe: TGraph graph
Eigenschaft: Selbstauskunft

s45 - startRefactoringProzess

Beschreibung: Startet den Refactoring-Prozess und steuert den dafür nötigen Ablauf. Benötigt Java-Code und liefert am Ende den veränderten Java-Code.
Eingabe: String[] javaPath, String[] classPath
Ausgabe: String javaURL
Eigenschaft: -

S46 - unparseGraphToJava

Beschreibung: Unparsed den übergebenen Graphen und liefert den daraus resultierenden Java-Code.
Eingabe: TGraph graph
Ausgabe: String javaURL
Eigenschaft: -

S47 - unparseXMLToJava

Beschreibung: Unparsed die übergebene XML-Datei und liefert den daraus resultierenden Java-Code.
Eingabe: String xmlPath
Ausgabe: String javaURL
Eigenschaft: -

S48 - uploadData

Beschreibung: Lädt die übergebenen Dateien hoch und liefert die URL zurück.
Eingabe: String dataPath
Ausgabe: String dataURL
Eigenschaft: -

S49 - variableNameExist

Beschreibung: Überprüft ob der übergebene Namen bereits im TGraphen vorhanden ist.
Eingabe: TGraph graph, String variableName
Ausgabe: boolean variableNameExist
Eigenschaft: -

S50 - zipFolder

Beschreibung: Verpackt den übergebenen Ordner in ein ZIP-Archiv.
Eingabe: String sourceFolder
Ausgabe: String zipFolder
Eigenschaft: -

7 Implementierung

In diesem Kapitel wird die Implementierung von einigen Services beschrieben. Es ist unnötig alle Services genau zu beschreiben, da diese Services andere Services in der richtigen Reihenfolge ausführen und somit nur aus ganz wenig Orchestrierung bestehen. Zur Serviceimplementierung wird SCA aus dem Kapitel 2.1.2 und als Framework für SCA wird IBM RSA Websphere verwendet.
Somit sind gleich zwei weitere Anforderungen die TA02 „SCA muss zur Service Implementierung verwendet werden.“ und die TA03 „IBM RSA Websphere muss als Framework für SCA verwendet werden.“ aus dem Kapitel 3.1 erfüllt. Zunächst wird in Kapitel 7.1 beschrieben, wie die Architektur in den Prototypen umgesetzt wird. Danach werden in Kapitel 7.2 die einzelnen Composites beschrieben. Abschließend wird in Kapitel 7.3 die Implementierung der „detectBadSmell“-Services und der „performRestructuring“-Services erläutert.

7.1 Architekturumsetzung

Für die eins zu eins Umsetzung der Architektur müsste für jeden Service eine Composite, eine Component und ein Interface erstellt werden. Dadurch wären die einzelnen Services sehr modular und ihre Implementierung könnte bei jedem Service einzeln ersetzt werden. Das Problem dabei wäre, dass auf diese Weise bei 50 Services 50 Composites, 50 Components und 50 Interfaces erstellt werden müssten. Das wären insgesamt 150 Objekte. Zusätzlich kämen die Verbindungen zwischen den einzelnen Services hinzu. Bei so vielen Objekten und Verbindungen würde die Modularität steigen, die Übersicht aber gleichzeitig sinken.
Bei dem Refactoring-Tool handelt es sich um einen Prototypen. Um die Zerlegung zu zeigen, reicht es diese exemplarisch an einigen Service zu demonstrieren. Die restlichen Services können grob gruppiert werden. Die grobe Gruppierung wird dadurch erreicht, dass mehrere Services auf einem Interface abgebildet werden. Dadurch verringert sich gleichzeitig die Anzahl der Verbindungen. Dies ist möglich, weil die Verbindungen in SCA über die Interfaces geknüpft werden. Über diesen können alle Service angesprochen werden, welche an dem Interface angehängt sind.
An jede Component muss mindestens ein Interface und genau eine Implementierung angehängt werden. Die Components haben ebenfalls die Referenzen auf die Services, die sie benötigen. Zuletzt können mehrere Components in einer Composite gesammelt werden. Wenn sich mehrere Components in einer Composite befinden und untereinander Referenzen haben, können diese Verbindungen bequem im IMB RSA Websphere erstellt werden. Denn der Websphere bietet eine graphische Darstellung der Composite in der per *Drag and Drop* die Verbindung zwischen einer Referenz und einem Service gezogen wird.
Diese spezielle Möglichkeit wird in dem Prototypen bevorzugt genutzt. Wenn sich der Service für die Referenz in einer anderen Composite befindet, muss die Verbindung in der XML-Darstellung der Composite manuell eingegeben werden. Da die Einträge in der XML alle auf Strings basieren, ist es sehr anfällig für Tippfehler. Um die Zerlegung in dem Prototypen zu zeigen, wird es daher nur wenige Composites geben.
Die erste Composite wird den „startRefactoringProzess“-Service beinhalten. Die zwei Composite wird alle Services, die dem Parsen angehören beinhalten. Die dritte Composite wird alle Services, die dem Unparsen angehören beinhalten. Die letzte Composite wird die verbliebenen Services von dem Refactoring-Prozess beinhalten.

7.2 Compositezerlegung

In diesem Kapitel werden die vier definierten Composites beschrieben. Gleichzeitig wird die Orchestrierung der Services in der Composite erläutert. Dieses Vorgehen ermöglicht die Fokussierung auf die Realisierung von den Services mit der meisten Funktionalität in Kapitel 7.3. Der Quelltext der Implementierung wird an dieser Stelle nicht gezeigt, da dieser auf der beigefügten CD eingesehen werden kann. An dieser Stelle werden lediglich die wichtigsten Bestandteile erläutert.

7.2.1 StartRefactoringProzessComposite

Die StartRefactoringProzessComposite beinhaltet nur den „startRefactoringProzess“-Service. Dieser hat die Aufgabe den gesamten Refactoring-Prozess zu leiten und die Services in der richtigen Reihenfolge zu starten. Zusätzlich informiert er den Nutzer über die Zwischenergebnisse.
Zu Beginn startet der „startRefactoringProzess“-Service den „parseJavaToGraph“-Service, wodurch er den TGraphen bekommt. Danach werden die Namen alle Refactoring-Verknüpfungen mit dem „getRefactoringNames“-Service abgerufen. Mit Hilfe des TGraphen und der Namen, können jetzt durch den „executeBadSmells“-Service die einzelnen „detectBadSmell“-Services gestartet werden, um die BadSmells zu erhalten.
Als nächstes werden die BadSmells für die Ausgabe vorbereitet. Dazu wird mit Hilfe des „getBadSmellSelection“-Service ein String für die Ausgabe generiert. Mit dem String und der Anzahl der verschiedenen BadSmells kann über den „readInt“-Service ein BadSmells-Typ durch den Nutzer gewählt werden. Mit den gewählten BadSmells und dem RefactoringName kann der „executeRestructurings“-Service gestartet werden, welcher den Graphen manipuliert. Wenn dies geschehen ist, wird wieder der „executeBadSmells“-Service ausgeführt, die Ausgabe erstellt und der Nutzer aufgefordert eine Auswahl zu treffen. Der Nutzer hat an dieser Stelle auch die Möglichkeit durch die Eingabe der Zahl 0 den Prozess abzubrechen. Dann wird der TGraph an den „unparseGraphToJava“-Service übergeben, welcher daraus Java-Quelltext generiert.

7.2.2 ParseComposite

Die ParseComposite beinhaltet den „parseJavaToGraph“-Service und alle Services die für seine Umsetzung nötig sind. Für die Realisierung des „parseJavaToXML“-Services wird ein „parseJava“-Web-Service zur Verfügung gestellt, welcher den „pro et con“[Kai] Parser ausführt. Dieser Service legt fest, dass die Java- und Class-Dateien auf einem FTP-Server sein müssen. Die nach dem Parsen erstellte XML-Datei wird auf dem selben FTP-Server abgelegt. Somit muss der Nutzer die Java- und Class-Dateien zunächst auf den FTP-Server hochladen, damit die URL's an den Parser übergeben werden können. Danach muss der „downloadData“-Service mit Hilfe der Java-Bibliothek „org.apache.commons.net.ftp“ die generierte XML-Datei lokal herunterladen, damit diese an den „importGraph“-Service übergeben werden kann. Dieser Ablauf demonstriert die Orchestrierung und die Realisierung in der ParseComposite.
Das besondere an dem „parse“-Service ist, dass dieser verschlüsselt ist. Zum Verwenden muss er entschlüsselt werden. Durch diese Tatsache kann der Prototyp demonstrieren, wie verschlüsselte Services angesprochen werden können. Für das Verschlüssen von Services und das Entschlüsseln von Referenzen bietet der IBM RSA Websphere eine GUI. Diese ist in die Administrative Konsole inte-

griert. Weil diese nicht intuitiv zu finden ist, wird ihr Auffinden im Anhang anhand von zwölf Bildern erklärt.

7.2.3 UnparseComposite

Die UnparseComposite beinhaltet den „unparseGraphToJava“-Service und alle Services die für seine Umsentzung nötig sind. Bis auf den „uploadData“-Service, da dieser sich bereits in der ParseComposite befindet. Der „uploadData“-Service lädt die Dateien auf den FTP-Server, damit der Nutzer diese von demselben Server herunterladen kann, wo er die Java- und Class-Dateien am Anfang hochgeladen hat. Damit wird der „uploadData“-Service zusammen mit dem „downloadData“-Service an die FTPManager-Component gehängt, da beide den selben FTP-Server ansprechen müssen. Anschließend werden beide in die ParseComposite aus dem Kapitel 7.2.2 positioniert.
Wenn der „unparseGraphToJava“-Service gestartet wird, übergibt dieser den TGraph an den „exportGraph“ -Service, welcher diesen in eine XML-Datei schreibt. Die XML-Datei wird daraufhin an den „unparseXMLToJava“-Service weitergeleitet. Dieser beinhaltet den „pro et con“[Kai] Unparser. Der Unparser besteht aus einer ausführbaren Datei, der „jgen.exe“. Diese benötigt die XML-Datei mit dem Graphen, welche mit dem Parameter „-s“ markiert wird und einen Ordner in dem die generierten Java-Dateien am Ende abgelegt werden, welcher mit dem Parameter „-o“ markiert wird. Zusätzlich werden zwei Parameter „-e“ und „-d“ mit den dazugehörigen Ordnern für den Error- und Debug-Output gesetzt. Als letztes wird der Parameter „-p“ gesetzt, damit die ausgegebenen Java-Dateien in der dazugehörigen Ordnerhierarchie ausgegeben werden. Nach dem Unparsen wird der Ordner mit den generierten Java-Dateien, dem Error-Log und dem Debug-Log an den „zipFolder“ übergeben, damit dieser den Ordner zu einem Zip-Archiv verpackt. Das Zip-Archiv wird an den bereits beschriebenen „uploadData“-Service übergeben, um das Archiv hochzuladen und somit für den Nutzer erreichbar zu machen. Dieser Ablauf demonstriert die Orchestrierung und die Realisierung in der UnparseComposite.

7.2.4 RefactoringComposite

Die RefactoringComposite beinhaltet die verbliebenen Services. Den Startpunkt stellt die RefactoringManagerComponent da. Diese beinhaltet zunächst den „executeBadSmells“-Service und den „getBadSmellNames“ -Service, da beide Services eine Referenz auf alle angebundenen „detectBadSmell“-Services haben. Zusätzlich beinhaltet die RefactoringManagerComponent den „executeRestructurings“ -Service und den „getRestructuringNames“-Service, da beide Services eine Referenz auf alle angebundenen „performRestructuring“-Services haben. Zuletzt beinhaltet die RefactoringManagerComponent noch den „getRefactoringNames“-Service, da dieser benötigt wird, um den „executeBadSmells“- und den „executeRestructurings“-Service aufzurufen.
Eigentlich müsste die Composite noch den „combineRefactorings“-Service beinhalte, damit die Kombinationen dynamisch erstellt werden können. In dem Prototyp wird dieser ausgelassen und die nötigen Kombinationen werden festgeschrieben. Der „combineRefactorings“-Service besteht fast ausschließlich aus der Interaktion mit dem Nutzer. Die vorhandenen BadSmells und Restructurings werden dem Nutzer demonstriert und er kann sich aus dieser Auswahl jeweils eins von jeder Seite aussuchen, zusammenfügen und diese mit einem Namen versehen. Die genannten Schritte sind bereits in anderen Services vorgekommen und bieten somit keine neuen Erkenntnisse. Daher wird der „com-

bineRefactorings“-Service ausgelassen, um den Fokus auf die „detectBadSmell“-Services und die „performRestructuring“-Services zu legen.

In der RefactoringManagerComponent werden drei HashMaps erstellt. Die erste Map beinhaltet die „getBadSmellNames“-Services. Diese werden zur Laufzeit aus der Referenz ausgelesen und in die Map eingefügt. Als Key wird immer der Name und als Value der Service gewählt. Nach dem selben Prinzip wird die zwei Map erstellt. Diese beinhaltet die „performRestructuring“-Services. Die dritte Map beinhaltet die Refactoring-Kombinationen. Als Key wird der Name und als Value die Kombination gewählt. Um die Kombination mit einer Klasse repräsentieren zu können, wird eine Refactoring Klasse hinzugefügt. Diese Klasse beinhaltet ein „detectBadSmell“- und ein „performRestructuring“-Service. Dazu ermöglicht sie den Zugriff auf beide Services.
Die Kombinationen für die dritte Map müssen extern gespeichert werden, da Services kein „Gedächtnis“ haben und somit Informationen nur für die Zeit des Zugriffs speichern können. Die Kombinationen können in einer Datei oder in einer Datenbank gespeichert werden. Zu diesem Zweck wird ein „read“-Service erstellt. Dieser hat die Aufgabe, die gespeicherte Kombinationen auszulesen. Intern werden die Kombinationen in einer Datei als Array gespeichert und beim Aufruf mithilfe von GSON [Goo] aus der Datei ausgelesen. Das Array wird zurückgegeben und in der RefactoringMangerComponent in die dritte Map überführt.
Zur Reduzierung der Anzahl von Verbindungen zwischen Referenzen und den dazugehörigen Services, werden in der RefactoringComposite zwei Interfaces erstellt. Diese bündeln mehrere Services. Das erste, ist das „GreQlQueryManager“-Interface. Dieses Interface bündelt alle Services, die eine GreQl-Abfrage beinhalten und eine Referenz auf den „greQlQuery“-Service haben. Dadurch können alle Service mit GreQl-Abfragen über eine Referenz erreicht werden. Das zweite, ist das „ShowAndGetUserInfos“-Interface. Dieses Interface bündelt alle Services aus dem Kapitel 5.4 „getChoice“. Dadurch können alle Stringerstellungs-, Ausgabe- und Eingabe-Services über eine Referenz erreicht werden. Beide Interfaces bündeln Services, die ein ähnliches Tätigkeitsfeld abdecken und sich in der Implementierung ähneln.

7.3 Service Implementierung

In diesem Kapitel wird die Implementierung der „detectBadSmell“-Services und der „performRestructuring“ -Services beschrieben. Alle diese Services befinden sich in der RefactoringComposite (Kapitel 7.2.4). Als erstes werden die drei „detectBadSmell“-Services, die bereits in Kapitel 5.3 erwähnt wurden, beschrieben. Danach werden die vier „performRestructuring“-Services, die bereits in Kapitel 5.5 erwähnt wurden, beschrieben.

7.3.1 findInappropriateVariableName

Der „findInappropriateVariableName“-Service besteht aus zwei Schritten. Darüber hinaus kann über die Methode „getName“ der eigene Name abgefragt werden.
Im ersten Schritt werden alle Variablen aus dem Graphen ausgelesen. Dazu wird der „getAllVariables“-Service verwendet.
Im zweiten Schritt werden die gefundenen Variablen entsprechend den definierten Regeln abgeglichen. Dafür wird der „getBadVariables“ -Service verwendet. Dieser Service nimmt die Variablen entgegen und liest zunächst über das Attribute „name“ die Namen der Variablen aus. Anschließend

gibt er die Namen an den „isGoodVariableName“-Service weiter. In diesem Service sind die Regeln definiert.
Die erste Regel lautet: „Der Variablenname darf kein Java-Schlüsselwort sein“. Die zweite Regel lautet: „Der Variablenname muss mit einem kleinen Buchstaben anfangen“. Die dritte Regel lautet: „Der Variablenname muss länger als drei Zeichen lang sein“. Die vierte und die letzte Regel lautet: „Der Variablenname muss ab einer Länge von zehn Zeichen Großbuchstaben im Namen beinhalten“. Der übergebene Variablenname wird mit Hilfe der Regeln überprüft. Falls er allen Regeln entspricht, wird ein TRUE zurückgegeben. Wenn er eine der Regel nicht erfüllt, wird ein FALSE zurückgegeben. Variablen, die ein FALSE bekommen haben, werden in dem „getBadVariables“-Service gesammelt und anschließend zurückgegeben.
Für die Realisierung der einzelnen Regeln eignen sich schwarze Listen. In diesen sind alle Namen aufgelistet, welche nicht benutzt werden dürfen. Ebenfalls eignen sich reguläre Ausdrücke. Diese können den Variablennamen auf die Struktur hin überprüfen und erkennen ob dieser der gewünschten Struktur entspricht.

7.3.2 findMultipleLocalVarialbeDeclaration

Der „findMultipleLocalVarialbeDeclaration“-Service besteht aus einer einzigen GreQl-Abfrage. Darüber hinaus kann über die Methode „getName“ der eigene Name abgefragt werden.
Diese Abfrage ist in dem „getLocalVarialbesWithMultipleDeclarations“-Service gekapselt. Mit ihr können alle lokalen Variablen gefunden werden, welchen mehrfach ein Wert zugewiesen wird.

7.3.3 findWasteLocalVariable

Der „findWasteLocalVariable“-Service besteht aus einer einzigen GreQl-Abfrage. Darüber hinaus kann über die Methode „getName“ der eigene Name abgefragt werden.
Diese Abfrage ist in dem „getLocalVariablesWithExactlyOneFunctionAssigmentExclusivlyInDeclaration“ -Service gekapselt. Mit ihr können alle lokale Variablen gefunden werden, welche ausschließlich in der Deklaration eine funktionbasierte Zuweisung haben.

7.3.4 renameVariable

Der „renameVariable“-Service besteht aus drei Schritten.
Im ersten Schritt wird der Nutzer aufgefordert aus den übergebenen BadSmells einen BadSmell auszuwählen. Dazu wird zunächst mit dem „getVarialbeNameSelection“-Service eine Übersicht für die Ausgabe erstellt. Danach wird mit Hilfe der erstellten Ausgabe und dem „readInt“-Service der Nutzer aufgefordert einen BadSmell auszuwählen.
Im zweiten Schritt wird der Benutzer aufgefordert einen neuen Variablennamen einzugeben. Dazu wird zunächst mit dem „getSelectedVariableName“-Service eine Übersicht für die Ausgabe erstellt. Danach wird mit Hilfe der erstellten Ausgabe und dem „readVariableName“-Service der Nutzer aufgefordert einen neuen Variablennamen einzugeben.
Im dritten Schritt wird der neue Variablenname in das Attribute „name“ der gewählten Variable eingetragen. Dadurch ist der Variablenname geändert.

7.3.5 splitMultipleDeclaration

Der „splitMultipleDeclaration“-Service besteht aus sieben Schritten.
Im ersten Schritt wird der Nutzer aufgefordert aus den übergebenen BadSmells einen BadSmell auszuwählen. Dazu wird zunächst mit dem „getVarialbeNameSelection“-Service eine Übersicht für die Ausgabe erstellt. Danach wird mit Hilfe der erstellten Ausgabe und dem „readInt“-Service der Nutzer aufgefordert ein BadSmell auszuwählen.
Im zweiten Schritt muss die Anzahl der Zuweisungen ermittelt werden. Dazu wird der „getAllOperationVerticesWithAssignOperator“-Service verwendet. Dieser liefert alle Knoten, die eine Zuweisung auf die übergebene Variable durchführen. Daraus lässt sich die Anzahl der Zuweisungen ermitteln.
Im dritten Schritt wird für jede Zuweisung ein neuer Variablenname benötigt. Dazu wird zunächst mit dem „getSelectedVariableName“-Service eine Übersicht für die Ausgabe erstellt. Danach wird mit Hilfe der erstellten Ausgabe und dem „readVariableName“-Service der Nutzer aufgefordert einen neuen Variablennamen einzugeben. Dabei wird der Nutzer so oft aufgefordert einen neuen Variablennamen einzugeben, bis die Anzahl der Variablennamen der Anzahl der Zuweisungen entspricht.
Im vierten Schritt werden alle benötigten Informationen aus dem Graphen ausgelesen. Als erstes wird die erste Zuweisung gewählt. Als zweites wird mit dem „getBlockContainsStatementFromOperation“-Service die Kante ermittelt, an der die übergebene Zuweisung stattgefunden hat. Als drittes wird mit dem „getAssignedValueFromOperation“-Service der Knoten mit dem zugewiesenen Wert der Zuweisung ermittelt. Als viertes wird mit dem „getHasDataObjectBetweenOperationAndVariable“-Service die Kante zwischen der Zuweisung und der Variable ermittelt, welche den Start der Umleitung repräsentiert. Als fünftes werden mit dem „getAllHasDataObjectFromVariable“-Service alle Kanten ermittelt, die die Verwendung der Variable repräsentieren. Als sechstes wird mit dem „getTypeReferenzFromVariable“-Service der Knoten ermittelt, der den Typen der Variable repräsentiert.
Im fünften Schritt wird eine neue Variable an der ermittelten Stelle erstellt. Dafür wird der erste-, neue Variablenname und der ermittelten Type verwendet.
Im sechsten Schritt müssen alle Verwendungen der alten Variable, die nach der ermittelten Stelle geschehen, auf die neue Variable umgeleitet werden. Als erstes werden alle Verwendungen, die vor dem ermittelten Start der Umleitung geschehen sind aussortiert. Der Start der Umleitung wird ebenfalls aussortiert. Als zweites müssen die Eigenschaften der Graphen und die Eigenschaften von Java im Hinblick auf Zuweisungen betrachtet werden. Denn in Java wird zuerst die rechte Seite der Zuweisung betrachtet und danach die linke. Bei den Graphen ist es umgekehrt. Es wird zunächst die linke Seite der Zuweisung betrachtet und danach die rechte. Um diesen Unterschied auszugleichen, muss die Stelle dort überprüft werden, wo die neue Variable erstellt wurde. Wenn sich dort eine Verwendung der alten Variable befindet, muss diese ebenfalls aussortiert werden. Dazu wird der „edgeInTheSubSystemOfOperation“-Service verwendet. Als drittes können alle verbliebenen Verwendungen auf die neue Variable umgeleitet werden.
Im siebten Schritt muss noch der Start der Umleitung gelöscht werden. Dazu wird der Alpha-Knoten der Umleitung gelöscht. Dieser befindet sich zwischen der alten Variable und der Zuweisung der Werte auf die neue Variable. Diese Verbindung hat keine Auswirkungen für den Unparser, sodass der Quelltext richtig generiert wird. Denn die Zuweisung kann nicht mehr von oben erreicht werden. Die Kante, die auf die Zuweisung zeigte, wurde für die Erstellung der neuen Variable verwendet. Die Verbindung für den Start der Umleitung hat aber Auswirkungen auf die GreQl-Abfragen der „detectBadSmell“-Services. Da die Verbindung immer noch in dem Graphen existiert und von der alten Variable aus erreichbar ist, werden mehr BadSmells gefunden, als tatsächlich im Graphen exis-

tieren.
Nach dem Löschen wird die neue Variable als alte Variable markiert. Danach werden die Schritte drei bis sieben mit der nächsten Zuweisung und dem nächsten Variablennamen solange wiederholt, bis alle Zuweisungen behandelt worden sind.

7.3.6 replaceLocalVariableWithMethod

Der „replaceLocalVariableWithMethod“-Service besteht aus sechs Schritten.
Im ersten Schritt wird der Nutzer aufgefordert aus den übergebenen BadSmells einen BadSmell auszuwählen. Dafür wird zunächst mit dem „getVarialbeNameSelection“-Service eine Übersicht für die Ausgabe erstellt. Danach wird mit Hilfe der erstellten Ausgabe und dem „readInt“-Service der Nutzer aufgefordert einen BadSmell auszuwählen.
Im zweiten Schritt wird der Benutzer aufgefordert einen Methodennamen einzugeben. Dazu wird zunächst mit dem „getSelectedVariableName“-Service eine Übersicht für die Ausgabe erstellt. Danach wird mit Hilfe der erstellten Ausgabe und des „readMethodeName“-Service der Nutzer aufgefordert einen Methodennamen einzugeben.
Im dritten Schritt werden alle benötigten Informationen aus dem Graphen ausgelesen. Als erstes wird mit dem „getTypeReferenzFromVariable“-Service der Knoten ermittelt, der den Typen der Variable repräsentiert. Als zweites wird mit dem „getVertexWithInitValue“-Service der Knoten mit dem von der Variable zugewiesenem Wert ermittelt. Als drittes werden mit dem „getEdgesWhereVariableInUse“-Service alle Kanten ermittelt, die die Verwendung der Variable repräsentieren. Als viertes werden mit „getMethodModifier“-Service alle Attribute ermittelt, die für die Erstellung einer Methode nötig sind.
Im vierten Schritt wird eine neue Methode erstellt. Dafür werden der ermittelte Type, der ermittelte Wert und die ermittelten Attribute verwendet.
Im fünften Schritt werden alle ermittelten Verweise von der alten Variable auf die neue Methode umgeleitet.
Im sechsten Schritt wird die alte Variable gelöscht.

7.3.7 resolveLocalVariable

Der „resolveLocalVariable“-Service besteht aus vier Schritten.
Im ersten Schritt wird der Nutzer aufgefordert aus den übergebenen BadSmells einen BadSmell auszuwählen. Dazu wird zunächst mit dem „getVarialbeNameSelection“-Service eine Übersicht für die Ausgabe erstellt. Danach wird mit Hilfe der erstellten Ausgabe und des „readInt“-Services der Nutzer aufgefordert einen BadSmell auszuwählen.
Im zweiten Schritt werden alle benötigten Informationen aus dem Graph ausgelesen. Als erstes wird mit dem „getVertexWithInitValue“-Service der Knoten mit von der Variable zugewiesenem Wert ermittelt. Als zweites werden mit dem „getEdgesWhereVariableInUse“-Service alle Kanten ermittelt, die die Verwendung der Variable repräsentieren.
Im dritten Schritt werden alle ermittelten Verweise von der alten Variable auf den ermittelten Wert umgeleitet.
Im vierten Schritt wird die alte Variable gelöscht.

8 Ansätze zu Erweiterbarkeit

In diesem Kapitel wird beschrieben wie die bestehende Architektur und der Prototyp um einen weiteren Refactoring erweitert werden können. Als Refactoring wird *DeleteUnusedVariable*-Refactoring gewählt, welches bereits in Kapitel 2.2.1 beschrieben wurde. Es wird die Realisierung des Refactorings vorgestellt und demonstriert, wie die Erweiterung auf der bestehenden Infrastruktur abläuft.

8.1 Refactoringerstellung

DeleteUnusedVariable-Refactoring besteht aus dem *UnusedVariable*-BadSmell und dem *DeleteVariable*-Restructuring. Aus dem *UnusedVariable*-BadSmell wird der „unusedVariable“-Service und aus dem *DeleteVariable*-Restructuring der „deleteVariable“-Service erstellt.

Der „unusedVariable“-Service besteht aus einer einzigen GreQl-Abfrage. Darüber hinaus kann über die Methode „getName“ der eigene Name abgefragt werden.
Diese Abfrage ist in dem „getUnusedVariables“-Service gekapselt. Mit ihr können alle ungenutzten Variablen gefunden werden.

unusedVariable

Beschreibung:	Findet und liefert alle ungenutzten Variablen aus dem Graph.
Eingabe:	TGraph graph
Ausgabe:	Vertex[] badSmells
Eigenschaft:	Selbstauskunft

Der „deleteVariable“-Service besteht aus zwei Schritten.
Im ersten Schritt wird der Nutzer aufgefordert aus den übergebenen BadSmells einen BadSmell auszuwählen. Dazu wird zunächst mit dem „getVarialbeNameSelection“-Service eine Übersicht für die Ausgabe erstellt. Danach wird mit Hilfe der erstellten Ausgabe und dem „readInt“-Service der Nutzer aufgefordert einen BadSmell auszuwählen.
Im zweiten Schritt wird die gewählte Variable aus dem TGraphen gelöscht.

deleteVariable

Beschreibung:	Entfernt die übergebene Variable aus dem Graphen und liefert den veränderten Graph zurück.
Eingabe:	TGraph graph, Vertex[] badSmells
Ausgabe:	TGraph graph
Eigenschaft:	Selbstauskunft

8.2 Refactoringerweiterung

Damit das Refactoring in die existierende Infrastruktur aufgenommen wird, müssen die beiden erstellten Services an die dazugehörigen übergeordneten Services angehängt werden. Nach dem Anhängen

der Services müssen diese zu einem Refactoring verknüpft werden, damit diese darüber angesprochen werden können. Wenn diese beiden Schritten abgeschlossen sind, kann das Refactoring im Programm verwendet werden. Die Orchestrierung muss nicht verändert werden, da diese auf keine feste Refactoringmenge beschränkt ist und sich dynamisch an die Anzahl von Refactorings anpasst.
Es wird gezeigt wie der „unusedVariable“-Service und der „deleteVariable“-Service angehängt und anschließend zu einem Refactoring kombiniert werden müssen.

Der „unusedVariable“-Service erbt die Eigenschaften des abstrakten „detectBadSmell“-Services und muss an die Referenz des „executeBadSmells“-Services angebunden werden. Auf der Implementiere-bene bedeutet es, dass eine Wire zwischen der Referenz des „executeBadSmells“-Services und dem SCA-Service des „unusedVariable“-Service erstellt werden muss. Damit auf den „unusedVariable“-Service referenziert werden kann muss dieser mit seiner Component und ihren Bestandteilen in die Composite aufgenommen werden.
Da das selbe auch für den „deleteVariable“-Service in ähnlicher Form zutrifft, werden beide Einträge der XML gleichzeitig in der Abbildung 8.1 aufgelistet. Wie sich der Aufbau der Einträge zusammensetzt wurde bereits in Kapitel 2.1.3 beschrieben und wird an dieser Stelle nicht erläutert.
Es ist nur wichtig, dass die *UnusedVariableComponent* in Zeile 4 das Interface *BadSmell* beinhaltet, da das Interface des abstrakten „detectBadSmell“-Services ist und dass die *DeleteVariableComponent* in Zeile 14 das Interface *Restructuring* beinhaltet, das das Interface des „performRestructuring“-Services ist. Die beiden Interfaces müssen von den beiden Services implementiert werden, damit diese im nächsten Schritt referenziert werden können.

Wenn der „unusedVariable“-Service und der „deleteVariable“-Service in die XML eingetragen worden sind, kann auf diese referenziert werden. Die Referenzen müssen in die bereits aus dem Kapitel 7.2.4 bekannte *RefactoringManagerComponent* eingetragen werden. Der veränderte Eintrag der *RefactoringManagerComponent* wird in der Abbildung 8.2 demonstriert. Die Referenz auf den „unusedVariable“-Service wird am Ende der Zeile 9 und die Referenz auf den „deleteVariable“-Service wird am Ende der Zeile 12 eingetragen.

Nachdem die beiden Services „unusedVariable“ und „deleteVariable“ erstellt und in die XML eingetragen sind, müssen diese noch zu dem *DeleteUnusenVariable*-Refactoring kombiniert werden. Die Kombinationen werden als GSON-Datei gespeichert, wie in Kapitel 7.2.4 beschrieben. Daher kann die Kombination direkt in der Datei ergänzt werden. Die zu ergänzende Zeile sieht folgendermaßen aus: **["DeleteUnusedVariable",“UnusedVariable","DeleteVariable"]**. Sie besteht aus dem Refactoringnamen, dem BadSmellnamen und dem Restructuringnamen. Nach dem Eintragen wird die Kombination automatisch aus der Datei ausgelesen. Somit ist das Refactoring *DeleteUnusedVariable* vollständig in den Prototypen integriert worden.

```
1  <component name="UnusedVariableComponent">
2     <implementation.java class="refactoring.badsmell.
         UnusedVariableImpl"/>
3     <service name="BadSmell">
4        <interface.java interface="refactoring.badsmell.BadSmell"/>
5     </service>
6     <reference name="greQlManager" target="GreQlManagerComponent/
         GreQlManager">
7        <interface.java interface="refactoring.greql.GreQlManager"/>
8     </reference>
9  </component>
10
11 <component name="DeleteVariableComponent">
12    <implementation.java class="refactoring.restructuring.
         DeleteVariableImpl"/>
13    <service name="Restructuring">
14       <interface.java interface="refactoring.restructuring.
            Restructuring"/>
15    </service>
16    <reference name="showAndGetUserInfos" target="
         ShowAndGetUserInfosComponent/ShowAndGetUserInfos">
17       <interface.java interface="userinteraction.
            ShowAndGetUserInfos"/>
18    </reference>
19 </component>
```

Abbildung 8.1: Service XML-Einträge

```
1 <component name="RefactoringManagerComponent">
2     <implementation.java class="refactoring.RefactoringManagerImpl"
        />
3     <service name="RefactoringManager">
4       <interface.java interface="refactoring.RefactoringManager"/>
5     </service>
6     <reference name="database" target="DatabaseComponent/Database">
7       <interface.java interface="refactoring.simpleservice.Database
          "/>
8     </reference>
9     <reference multiplicity="1..n" name="badSmells" target="
        InappropriateVariableNameComponent/BadSmell
        MultipleLocalVarialbeDeclarationComponent/BadSmell
        WasteLocalVariableComponent/BadSmell UnusedVariableComponent
        /BadSmell">
10      <interface.java interface="refactoring.badsmell.BadSmell"/>
11    </reference>
12    <reference multiplicity="1..n" name="restructurings" target="
        RenameVariableComponent/Restructuring
        SplitMultipleDeclarationComponent/Restructuring
        ReplaceLocalVariableWithMethodComponent/Restructuring
        ResolveLocalVariableImpl/Restructuring
        DeleteVariableComponent/Restructuring">
13      <interface.java interface="refactoring.restructuring.
          Restructuring"/>
14    </reference>
15  </component>
```

Abbildung 8.2: RefactoringManagerComponent XML-Eintrag

9 Validierung

In diesem Kapitel werden zwei Bestandteile der Bachelor-Arbeit getestet. Als erstes wird im Kapitel 9.1 die Wiederverwendbarkeit des erstellten Services an dem Beispiel eines Metrik-Tools getestet. Als zweites wird im Kapitel 9.2 das Verknüpfen von Refactorings genauer analysiert und eine verbesserte Variante davon vorgeschlagen.

9.1 Servicewiederverwendbarkeit als Metrik

Mit dem Begriff Metrik werden in der Informatik Verfahren zur Messung von zählbaren Größen bezeichnet [Ebe96]. Das Ergebnis einer Metrik ist immer eine Zahl. Metrik kann ein Prozess sein, der die Anzahl aller Variablen oder die Anzahl aller Methoden liefert. Mit diesem Verhalten ähnelt die Metrik der BadSmell-Erkennung. Sie unterscheiden sich darin, dass die BadSmell-Erkennung die einzelnen BadSmells liefert und die Metrik einfach die Anzahl gefundener BadSmells.
Da die BadSmell-Erkennung bereits realisiert ist und die Metrik dieser sehr stark ähnelt, kann die Metrik mit den bestehenden Services nachgebaut werden. Ausgehen von dem Refactoring-Prozess, welcher in Kapitel 4.1 in vereinfachter Form vorgestellt wurde, wird die dort verwendete Abbildung 4.1 für den Metrik-Prozess angepasst. Daher werden die Unterschiede und die Ähnlichkeiten zwischen den beiden Prozessen anhand der Abbildung 9.1 verdeutlicht.

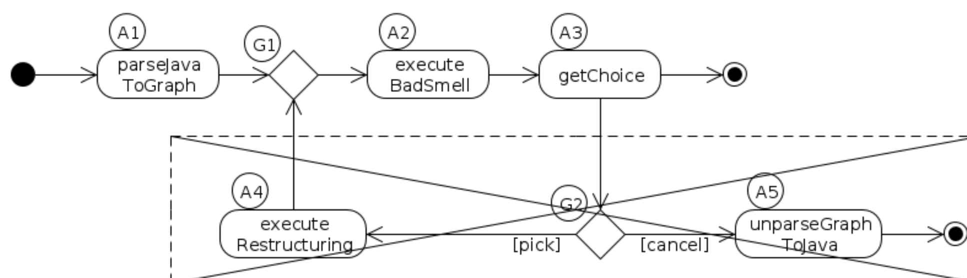

Abbildung 9.1: Metrik Orchestrierung

Die Abbildung 9.1 zeigt die Bereiche an, die für die Metrik benötigt werden. Da das Diagramm auf dem *Refactoring Orchestrierung*-Diagramm aufbaut, werden die Unterschiede direkt verdeutlicht. Es ist zu sehen, dass der Metrik-Prozess nur die drei oberen Services benötigt (A1 bis A3 in Abb. 9.1). Die unteren drei Services werden nicht benötigt (A4 und A5). Diese sind auch in der Abbildung in einem gestrichelten, durchgestrichenem Rechteck abgebildet.
Es gibt auch einen Unterschied in der Orchestrierung. Denn durch die Metrik wird der Quelltext nicht verändert. Es ist wird eine Analyse durchgeführt, daher reicht es diese einmal durchzuführen. Wenn der Quelltext sich nicht ändert, dann ändern sich die Ergebnisse der Analyse auch nicht. Dadurch, dass die Orchestrierung keine Wiederhollungen enthält, wird das Ende des Prozesses bereits nach der Benutzerausgabe erreicht.
Zusammengefasst werden folgende Services für die Realisierung des Metrik-Prozesses benötigt: Es wird eine Parser benötigt, damit der Quelltext zu einem Graphen überführt wird. Es wird ein Metrik-Service benötigt, um die definierten Metriken auf den Graphen durchführen zu können. Zuletzt wird

noch ein Visualisierung-Service benötigt, um die gefundenen Ergebnisse aus dem Graphen dem Nutzer zu präsentieren.
Der Parser ist bereits als „*S35 - parseJavaToGraph*“-Service vorhanden und kann wiederverwendet werden.
Als Metrik-Service kann der „*S05 - executeBadSmells*“-Service verwendet werden. Es muss nur die Ausgabe angepasst wird, damit nicht die BadSmells sondern nur ihre Anzahl ausgegeben werden.
Die Visualisierung mit der Anzahl der BadSmells kann von dem „*S18 - getBadSmellSelection*“-Service erstellt werden.
Um die Visualisierung dem Nutzer anzuzeigen, kann der „*S37 - readInt*“-Service verwendet werden. Die eingegebene Zahl kann zu der Beendigung des Programms genutzt werden. Dann kann der Nutzer sich die Ergebnisse anschauen und selber entscheiden, wann er das Programm beenden möchte.
Das einzige was neu erstellt werden muss, ist die Orchestrierung, die diese Services miteinander verbindet. Der „*S45 - startRefactoringProzess*“-Service kann an dieser Stelle nicht wiederverwendet werden, da wie bereits an der Abbildung 9.1 zu sehen ist, dieser mehr erfüllt als benötigt wird.
Der neue Service für die Orchestrierung wird „*startMetrikProzess*“ genannt und führt die benötigten Services in der beschriebenen Reihenfolge aus.

- startMetrikProzess

Beschreibung:	Startet den Metrik-Prozess und steuert den dafür nötigen Ablauf. Benötigt Java-Code und liefert am Ende das Ergebnis als String.
Eingabe:	String[] javaPath, String[] classPath
Ausgabe:	String result
Eigenschaft:	-

An dem Metrik-Beispiel ist gut zu sehen, dass die Services eine lockere Bindung haben und daher leicht Wiederverwendung finden. Die grobe Funktionalität konnte vollständig aus dem Refactoring-Tool entnommen und musste nicht erneut entwickelt werden. Durch die Wiederverwendung der Services im Metrik-Tool erleichtert sich die Wartung. Da die Services nur an einer Stelle verändert werden müssen, um für beide Tools wirksam zu sein.

9.2 Refactoring Verknüpfung

In diesem Kapitel wird eine effizientere Möglichkeit der Verknüpfung von BadSmells und Restructurings vorgestellt.
In Kapitel 7.2.4 wurde vorgestellt, dass jede Refactoring-Kombination aus einem BadSmell und einem Restructuring besteht. Diese Art zu Verknüpfen hat einen negativen Einfluss auf die Performance. Um das Problem an einem Beispiel zu demonstrieren, wird das *ResolveWasteLocalVariable* -Refactoring genauer untersucht.
Das *ResolveWasteLocalVariable* -Refactoring wurde bereits in Kapitel 2.2.1 vorgestellt. Es besteht aus dem *WasteLocalVariable*-BadSmell, dem *ResolveLocalVariable*-Restructuring und dem *ReplaceLocalVariableWithMethod*-Restructuring. Es sind ein BadSmell und zwei Restructurings. Die alte Verknüpfungsart kann immer nur ein BadSmell mit einem Restructuring verknüpfen. Daher muss das *WasteLocalVariable*-BadSmell zwei Mal verwendet werden.
Bei dem Ausführen der BadSmell-Erkennung werden alle Refactorings-Kombinationen durchlaufen

und es wird jedes BadSmell ausgeführt. Da das *WasteLocalVariable*-BadSmell zweimal unter den Kombinationen erscheinen, wird es auch zweimal ausgeführt. Während einer BadSmell-Erkennung ändert sich der Graph nicht, er darf sich auch nicht ändern, bevor die Ergebnisse dem Nutzer vorgestellt werden. Andernfalls würde der Nutzer Restructurings auf veralteten BadSmells ausführen, was zu Konflikten führen könnte wenn diese nicht mehr vorhanden sind. Daher wird bei mehrfachem Ausführen eines BadSmells immer dasselbe Ergebnis geliefert und somit Rechenzeit vergeudet.
Die alte Art zu Verknüpfen hat zwei Probleme: Erstens kann der User nicht auf einen Blick erkennen welche Restructurings ausgeführt werden können um den gefundenen BadSmell zu beseitigen. Zweitens werden BadSmells mehrfach ausgeführt, was Rechenzeit vergeudet.

Um beide Probleme zu beheben löst sich die neue Art zu Verknüpfen von dem Refactoring-Gedanken, dass jedes Refactoring aus einem BadSmell und einem Restructuring besteht. Es werden die BadSmells in den Fokus gestellt. Die Verknüpfung geht von dem BadSmell aus, damit dieser nur einmal existiert und nur einmal ausgeführt werden muss. An jeden BadSmell werden passende Restructurings angehängt. Damit kann der Nutzer sofort erkennen, welche Möglichkeiten ihm zur Verfügung stehen, um den BadSmell zu beseitigen.
Wenn jeder BadSmell genau ein Mal vorkommt, dann müssen die Restructurings mehrfach vorkommen. Das mehrfache Vorkommen von Restructurings ist unkritisch und sogar typisch für Refactorings. Martin Fowler gibt in manchen Beispielen [Fow00] bis zu vier Restructurings zur Beseitigung eines BadSmells an. Einige Restructurings kommen auch mehrfach bei den verschiedenen BadSmells vor.
Es ist unkritisch wie viele Restructurings an ein BadSmell oder ob diese an verschiedene BadSmells angehängt sind. Denn ein Restructuring wird nur ausgeführt, wenn der Nutzer diesen explizit Ausführt. Einen Vergleich bitet der Refactoring-Prozess. Hier werden immer alle BadSmells und daraufhin ein Restructuring durchgeführt. Wenn der Nutzer in einem Durchlauf vier Restructurings ausführt, wurden alle BadSmells fünf mal ausgeführt. Wenn der Nutzer in einem Durchlauf keinen Restructuring ausführt und den Durchlauf direkt abbricht, wurden alle BadSmells einmal ausgeführt. Daher muss der Fokus, vor allem im Hinblick auf die Performance, auf den BadSmells liegen.

10 Zusammenfassung

Im Rahmen dieser Bachelor-Arbeit wurden Refactorings servicebasiert betrachtet. Es hat sich gezeigt, dass Services gut für die Realisierung von Refactorings geeignet sind. Der gesamte Refactoring-Prozess ließ sich sehr leicht grob Zerteilen, sodass der Parser und der Unparser abgekapselt werden konnten. Dadurch konnte der Fokus der Betrachtung auf die einzelnen Refactorings gelegt werden.
Die Refactorings ließen sich in BadSmells und Restructurings zerlegen. Im Rahmen der Beispiele, von vier BadSmells und fünf Restructurings, konnten für beide einheitliche Schnittstellen definiert werden. Dadurch konnten alle BadSmells und alle Restructurings auf die gleich Weise angesprochen werden. Die Erweiterung, um einen weiteren Refactoring konnte durch die einheitlichen Schnittstellen mit wenig Aufwand durchgeführt werden. Durch das Verknüpfen der Services in der XML-Datei, musste sogar der Java-Quelltext nicht verändert werden. Das sind bemerkenswerte Eigenschaften, die das Erweitern und das Warten der Software erleichtern.
In der weiterführenden Servicezerteilung der einzelnen BadSmells und Restructurings haben sich mehrere Services herauskristallisiert, die von mehreren Services genutzt wurden. Besonders interessant waren dabei die Services mit GreQL-Abfragen. Denn das Entwickeln eines GreQL-Services nimmt viel Zeit in Anspruch. Die Abfrage muss entworfen, auf Randfälle getestet und am Ende optimiert werden. Es hat sich bereits bei fünf Restructurings gelohnt die GreQL-Abfragen in eigene Services zu verpacken. Da einige Abfragen bei mehreren Restructuring-Services gebraucht wurden.Das beste Beispiel dafür ist das *ResolveLocalVariable*-Restructuring, denn es konnte komplett aus den GreQl-Services des *ReplaceLocalVariableWithMethod*-Restructurings zusammengestellt werden.
Da die Vorteile der Servicezerlegung bereits bei vier Refactorings bemerkbar sind, werden diese mit steigender Refactoringzahl weiter zunehmen. Dabei muss bedacht werden, dass bei der Wiederverwendung von Services diese nicht dupliziert werden. Somit müssen Änderungen nur an einem Service durchgeführt werden und alle Services, die den veränderten Service benötigen, werden mit der neuen Version weiterarbeiten.

Services bringen allerdings nicht nur Vorteile mit sich, sondern auch einige Probleme. Ein besonders großes Problem bei Services ist die enorme Lernkurve am Anfang des Lernprozesses. Die Motivation von Services klingt sehr einfach und verlockend. Sie ähnelt in der Komponentenzerlegung stark der Objekt-Orientierung. Damit kann die Lernhürde bereits am Anfang unterschätzt werden. Es benötigt aber Zeit, um ein Gefühl für die richtige Granularität von Services zu entwickeln. Noch schwerer ist es die Möglichkeiten zu erkennen einige von diesen Services durch universelle Services zu ersetzen, um die Vorteile von Services besser nutzen zu können.
Die Services sind sehr gut Theoretisch aufgearbeitet und es gibt auch Implementierungen, welche die theoretischen Ideen umsetzen. Es fehlen aber praktische Beispiele an denen die theoretischen Ideen demonstriert werden. In den meisten Fällen geht es nicht weit über ein „Hallo Welt“-Beispiel hinaus. Ein „Hallo Welt“-Beispiel reicht aber nicht aus um damit einen Prototypen für ein Refactoring-Tool zu schreiben, geschweige denn eine professionelle Software mit Services zu entwickeln. Oft werden Eigenschaften von Services erwähnt ohne dabei auf die Realisierung oder die Verwendung dieser Eigenschaft auf der Quelltextebene zu beschreiben. Damit wird die bereits hohe Lernkurve noch weiter Angehoben.

Insgesamt betrachtet sind Services sehr gut für große Projekte geeignet, da diese mehr Möglichkeiten zur Wiederverwendbarkeit einzelner Services bieten. Für die agile Entwicklung eignen sich Services durch ihre lose Bindung zureinander. Dadurch können einzelne Services leicht verändert oder ersetzt

werden.
Nach dem Überwinden der anfänglichen Lernkurve bieten Services keine „großen Überraschungen“ mehr und können effizient zur Entwicklung von Software verwendet werden. Durch fortschreitende Serviceentwicklung wird sich die Situation der praktischen Beispielen verbessern, wodurch die Lernkurve abflachen wird.

CD-Verzeichnis

Im diesem Kapitel werden die Inhalte auf der beigefügten CD aufgelistet und kurz beschrieben.

Servicebasierte Refactorings - Sergej Tihonov - Nov 2013.pdf
Bei der PDF-Datei handelt es sich um die digitale Version der Bachelor-Arbeit.

Servicebasierte Refactorings Prototyp.zip
Der Zip-Archive beinhaltet den Prototypen als IBS RSA Websphere Projekt. Er enthält alle Quelltexte und Bibliotheken, die benötigt werden.

JsonDatabase.json
Diese JSON-Datei beinhaltet die Refactoring-Kombinationen und kann mit GSON [Goo] ausgelesen werden.

Abkürzungen

Abb.	Abbildung
AM	Assembly Model
FA	Funktionale Anforderungen
FTP	File Transfer Protocol
GreGL	Graph Repository Query Language
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines
JGraLab	Java Graphenlabor
NA	Nichtfunktionale Anforderungen
RSA	Rational Software Architect
SAM	Service Assembly Model
SCA	Service Component Architecture
SOA	Service-Oriented Architecture
TA	Technische Anforderungen
XML	Extensible Markup Language
ZIP	ZIP-Dateiformat (englisch: zipper)

Abbildungen

2.1 SAM-Diagramm Beispiel 1 4
2.2 SAM-Diagramm Beispiel 2 4
2.3 AM-Diagramm Übersicht 6
2.4 ParseJavaToGraph Interface 7
2.5 ParseJavaToGraphImpl Klasse 8
2.6 ParseServices Composite 9
2.7 Variablenerstellung in JGraLab 13
2.8 Variablensuche mit GreQL 14

4.1 Refactoring Orchestrierung 19
4.2 Vereinfachte Architektur 21

5.1 parseJavaToGraph 25
5.2 unparseGraphToJava 26
5.3 executeBadSmells 27
5.4 getChoise 28
5.5 executeRestructurings 31
5.6 combineRefactorings 33

8.1 Service XML-Einträge 53
8.2 RefactoringManagerComponent XML-Eintrag 54

9.1 Metrik Orchestrierung 55

10.1 Admin Console 1/12 71
10.2 Admin Console 2/12 71
10.3 Admin Console 3/12 72
10.4 Admin Console 4/12 72
10.5 Admin Console 5/12 73
10.6 Admin Console 6/12 73
10.7 Admin Console 7/12 74
10.8 Admin Console 8/12 74
10.9 Admin Console 9/12 74
10.10 Admin Console 10/12 75
10.11 Admin Console 11/12 75
10.12 Admin Console 12/12 76

Literatur

[ABB+09] ARSANJANI, Ali ; BOOCH, Grady ; BOUBEZ, Toufic ; BROWN, Paul C. ; CHAPPELL, David ; DEVADOSS, John ; ERL, Thomas ; JOSUTTIS, Nicolai ; KRAFZIG, Dirk ; LITTLE, Mark ; LOESGEN, Brian ; MANES, Anne T. ; MCKENDRICK, Joe ; ROSS-TALBOT, Steve ; TILKOV, Stefan ; UTSCHIG-UTSCHIG, Clemens ; WILHELMSEN, Herbjörn: *SOA Manifesto*. http://www.soa-manifesto.org/. Version: 2009, Abruf: 10.07.2013

[Bar09] BARRETO, Jos Angel M.: *Transaktionskonzept fur die TGraphenbibliothek JGraLab*, Universität Koblenz · Landau, Diploma Thesis, mar 2009

[Bil06] BILDHAUER, Daniel: *Ein Interpreter für GReQL 2*, Universität Koblenz · Landau, Diploma Thesis, aug 2006

[Cha07] CHAPPELL, David: *Introducing SCA*. http://www.davidchappell.com/articles/introducing_sca.pdf. Version: jul 2007, Abruf: 10.07.2013

[Ebe96] EBERT, Christof: *Software-Metriken in der Praxis: Einführung und Anwendung von Software-Metriken in der industriellen Praxis*. Berlin Heidelberg, Germany : Springer, 1996. – ISBN 3–6428–8195–5

[EBF+] EBERT, Jürgen ; BILDHAUER, Daniel ; FALKOWSKI, Kerstin ; RIEDIGER, Volker ; WINTER, Andreas ; SCHWARZ, Hannes: *TGraphen — Universität Koblenz · Landau*. http://www.uni-koblenz-landau.de/koblenz/fb4/institute/IST/AGEbert/MainResearch/Graphentechnologie/TGraphen, Abruf: 9.11.2013

[Fow00] FOWLER, Martin: *Refactoring: Wie Sie das Design vorhandener Software verbessern*. München, Germany : Addison-Wesley, 2000. – ISBN 3–8273–1630–8

[Goo] GOOGLE: *google-gson - A Java library to convert JSON to Java objects and vice-versa*. http://code.google.com/p/google-gson/, Abruf: 15.09.2013

[HBBK07] HURWITZ, J. ; BLOOR, R. ; BAROUDI, C. ; KAUFMAN, M.: *Service Oriented Architecture For Dummies*. Wiley, 2007. – ISBN 9780470116784

[Jel13] JELSCHEN, Jan: *Software Evolution Services: Vision Document for a PhD Thesis*. jan 2013

[Kai] KAISER, Uwe: *pro et con : Analyse - Reengineering - Migration*. http://www.proetcon.de/de/migCOBOL_d.html, Abruf: 25.09.2013

[LCF+11] LAWSON, S.D. ; COMBELLACK, M. ; FENG, R. ; MAHBOD, H. ; NASH, S.: *Tuscany SCA in Action*. Manning Publications Company, 2011. – ISBN 9781933988894

[Mei12] MEIER, Johannes: *Eine Fallstudie zur Interoperabilität von Software-Evolutions-Werkzeugen in SCA*, Carl von Ossietzky Universität Oldenburg, Bachelor Thesis, nov 2012

[Mei13] MEIER, Johannes: *TGraphen mit JGraLab*. jan 2013

[Obj] OBJECT MANAGEMENT GROUP: *OMG Unified Modeling Language(TM) (OMG UML), Superstructure*. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/, Abruf: 9.11.2013

Index

Ansätze zu Erweiterbarkeit, 51
  Refactoringerstellung, 51
  Refactoringerweiterung, 51
Architektur, 19
  Refactoring Analyse, 19
  Vereinfachte Architektur, 20

Erweiterte Architektur mit Anforderungen, 25
  combineRefactorings, 32
  executeBadSmells, 26
  executeRestructurings, 30
  getChoice, 28
  parseJavaToGraph, 25
  unparseGraphToJava, 26

Implementierung, 43
  Architekturumsetzung, 43
  Compositezerlegung, 44
    ParseComposite, 44
    RefactoringComposite, 45
    StartRefactoringProzessComposite, 44
    UnparseComposite, 45
  Service Implementierung, 46
    findInappropriateVariableName, 46
    findMultipleLocalVarialbeDeclaration, 47
    findWasteLocalVariable, 47
    renameVariable, 47
    replaceLocalVariableWithMethod, 49
    resolveLocalVariable, 49
    splitMultipleDeclaration, 48

Metrik, 55

Refactoring, 10
  DeleteUnusedVariable, 11
    DeleteVariable, 11
    UnusedVariable, 11
  RenameInappropriateVariable, 10
    InappropriateVariableName, 10
    RenameVariable, 10
  ResolveWasteLocalVariable, 11
    ReplaceLocalVariableWithMethod, 11
    ResolveLocalVariable, 11
    WasteLocalVariable, 11
  SplitMultipleLocalVariableDeclaration, 11
    MultipleLocalVarialbeDeclaration, 11
    SplitMultipleDeclaration, 11
Refactoring Verknüpfung, 45

Service, 3
  Service Component Architecture, 5
  Service-Oriented Architecture, 3
ServiceKatalog, 35

TGraph, 12
  GreQL, 13
  JGraLab, 12

Validierung, 55
  Refactoring Verknüpfung, 56

Anhang

Für das Verschlüssen von Services und das Entschlüsseln von Referenzen bietet der IBM RSA Websphere eine GUI. Diese ist in die Administrative Konsole integriert. Weil diese nicht intuitiv zu finden ist, wird ihr Auffinden im Anhang anhand von zwölf Bildern erklärt. Die wichtigen Stellen werden in den Abbildungen Rot umrandet.
Als erstes muss der Server gestartet und die Administrative Konsole geöffnet werden (Abb. 10.1).

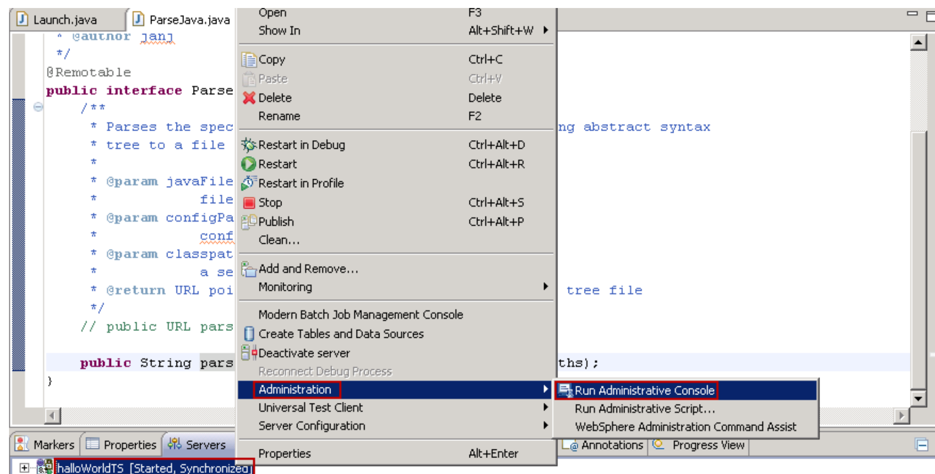

Abbildung 10.1: Admin Console 1/12

Als zweites muss der Nutzer sich in die Admin Console anmelden (Abb. 10.2).

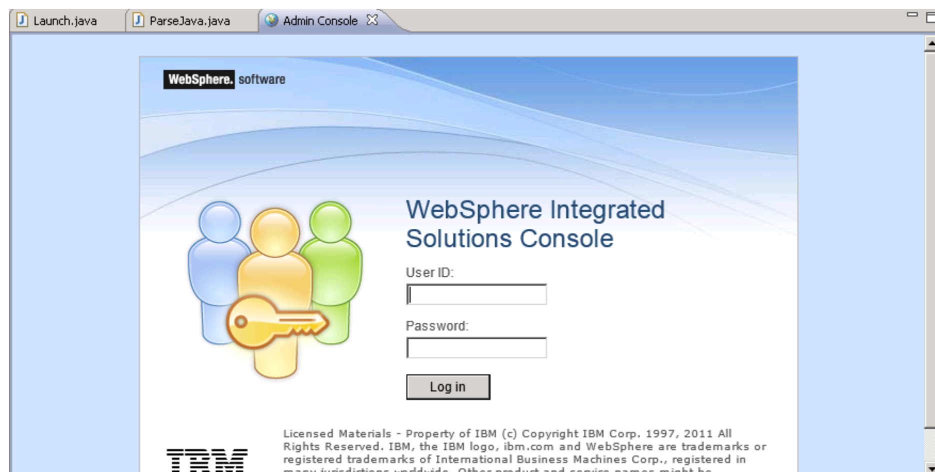

Abbildung 10.2: Admin Console 2/12

Als drittes muss unter „Applications“ und „Application Types“ der Punkt „Business-level applications“ gewählt werden (Abb. 10.3).

Als viertes muss das Projekt ausgewählt werden, in dem sich die verschlüsselte Referenz befindet (Abb. 10.4).

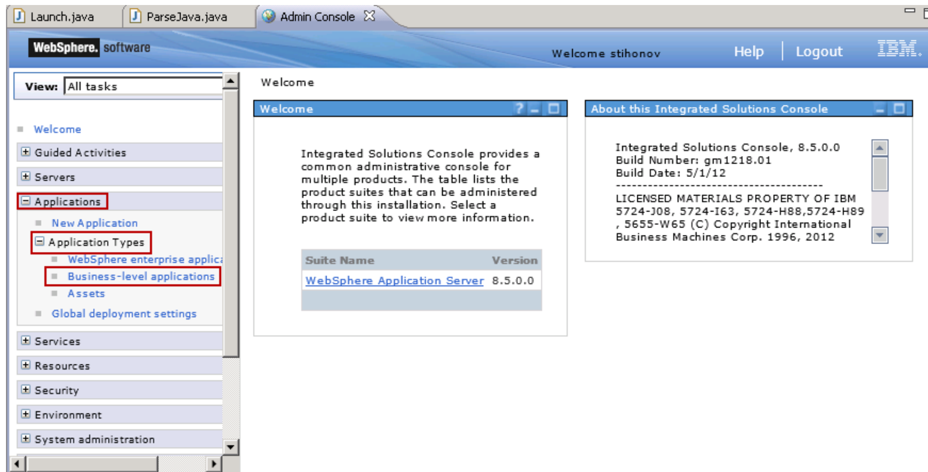

Abbildung 10.3: Admin Console 3/12

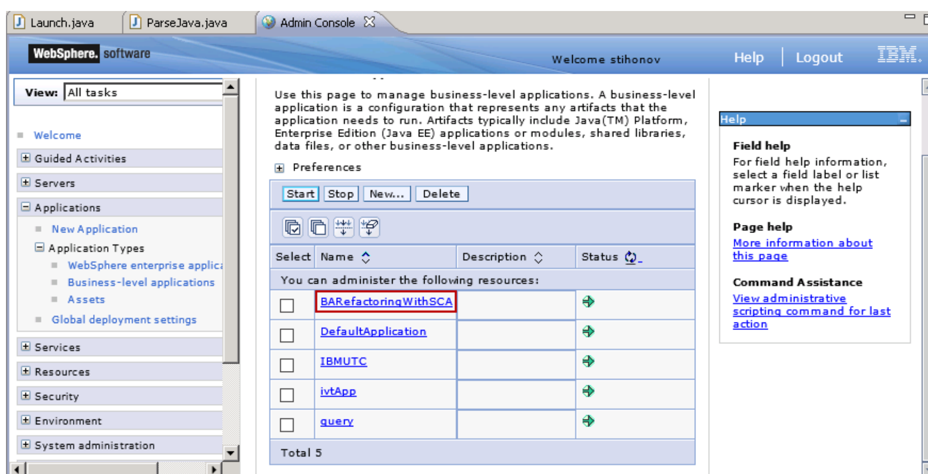

Abbildung 10.4: Admin Console 4/12

Als fünftes muss die Composite ausgewählt werden, in der sich die verschlüsselte Referenz befindet (Abb. 10.5).

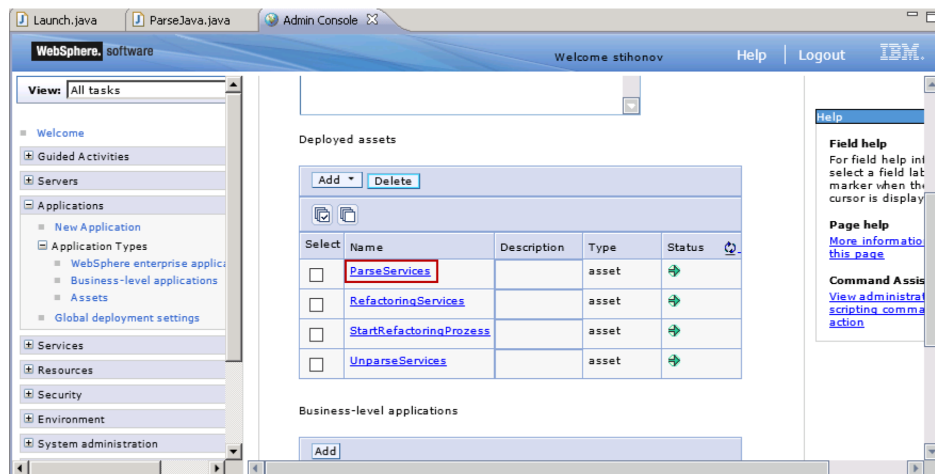

Abbildung 10.5: Admin Console 5/12

Als sechstes muss „References policy sets and bindings“ zum entschlüsseln ausgewählt werden (Abb. 10.6). Zum verschlüsseln wird ein WebService benötigt. Dann erscheint über dem Punkt „References policy sets and bindings“ der Punkt „Services policy sets and bindings“. Dort kann der Service verschlüsselt werden.

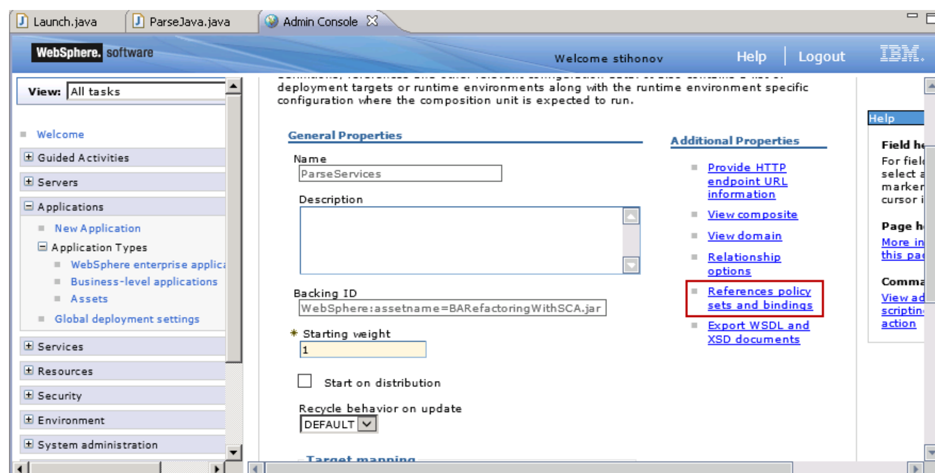

Abbildung 10.6: Admin Console 6/12

Als siebtes muss der Port von der Referenz gewählt werden. An diesen Port muss ein „Policy Set“ angehängt werden (Abb. 10.7). Das „Policy Set “ legt fest, nach welchem Protokoll der Informationsaustausch stattfinden soll.

Als achtes muss ein „Policy Set“ erstellt werden. Dieser benötigt nur den „HTTP transport“, da der Service nur mit „Basic authentication“ verschlüsselt wurde (Abb. 10.8).

Als neuntes müssen die Einstellungen von dem „HTTP transport“überprüft werden. In der Abbildung 10.9 sind die Standardeinstellungen zu sehen. Diese reichen für das entschlüsseln aus.

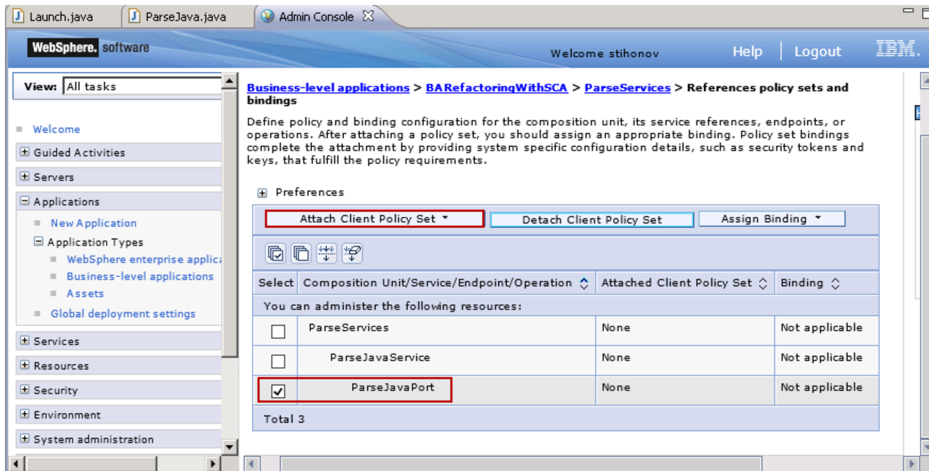

Abbildung 10.7: Admin Console 7/12

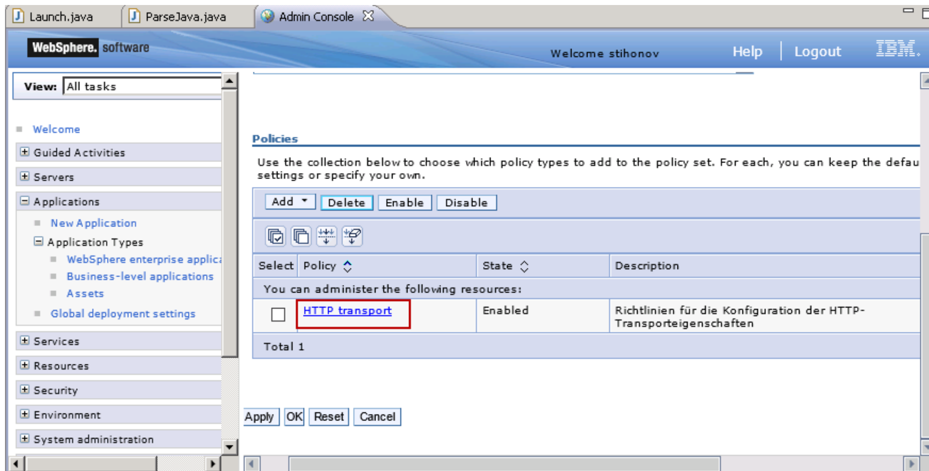

Abbildung 10.8: Admin Console 8/12

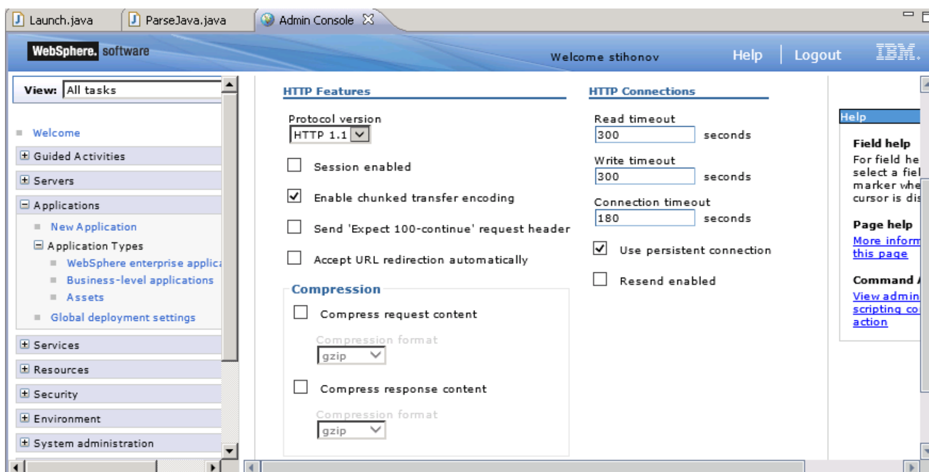

Abbildung 10.9: Admin Console 9/12

Als zehntes muss noch das „Binding“ hinzugefügt werden (Abb. 10.10). Das „Binding“ beinhaltet den Benutzernamen und das Passwort. Es kann ein neues „Bindung“ erstellt werden oder ein bereits vorhandenes, wie zum Beispiel „Client sample“, verwendet werden.

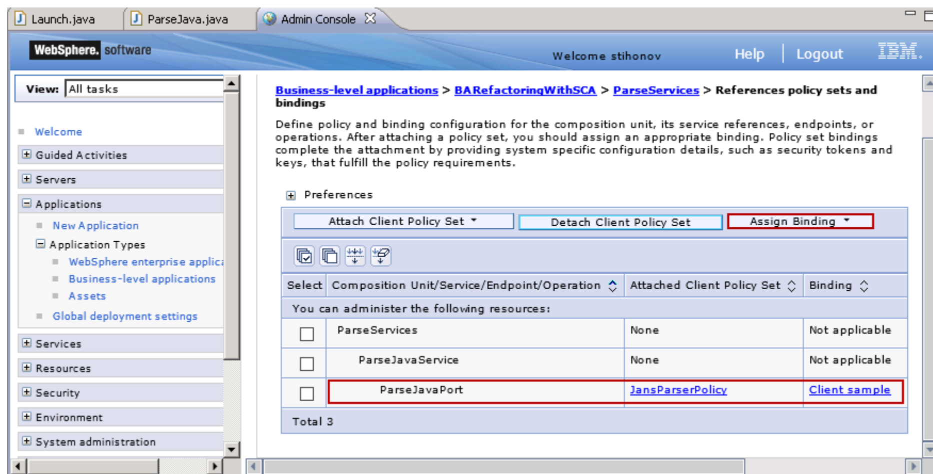

Abbildung 10.10: Admin Console 10/12

Als elftes muss überprüft werden, dass das „Binding“ ebenfalls einen „HTTP transport“ beinhaltet (Abb. 10.11).

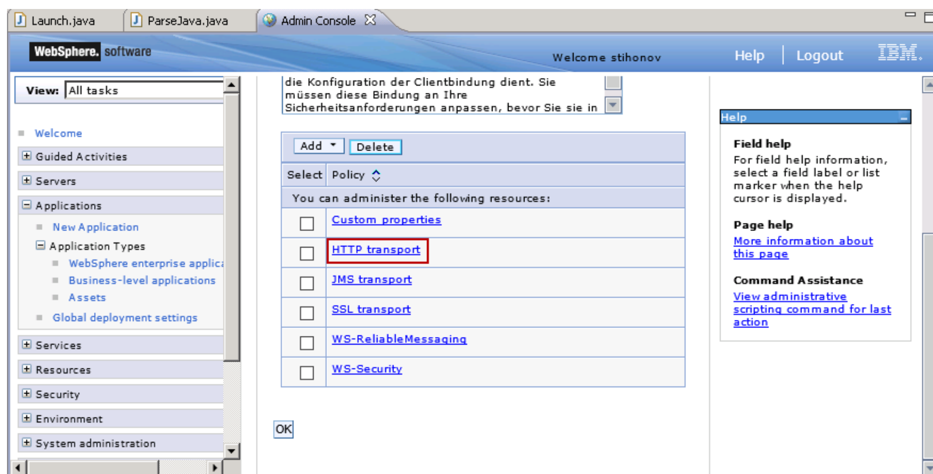

Abbildung 10.11: Admin Console 11/12

Als zwölftes muss der Benutzername und das Passwort in das Feld „Basic autentication for outbound service requests“ eingetragen werden (Abb. 10.12). Wenn dies erledigt ist, müssen alle Einstellungen gespeichert werden. Danach ist die Referenz entschlüsselt und kann verwendet werden.

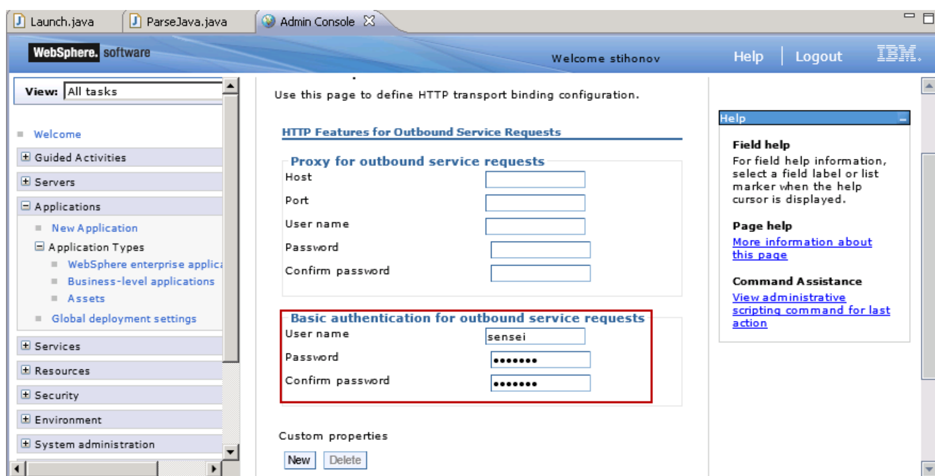

Abbildung 10.12: Admin Console 12/12

Versicherung

Hiermit versichere ich, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Außerdem versichere ich, dass ich die allgemeinen Prinzipien wissenschaftlicher Arbeit und Veröffentlichung, wie sie in den Leitlinien guter wissenschaftlicher Praxis der Carl von Ossietzky Universität Oldenburg festgelegt sind, befolgt habe.

Oldenburg, den 14. November 2013 ______________________________

Sergej Tihonov