



JOHANNES KEPLER  
UNIVERSITÄT LINZ  
Netzwerk für Forschung, Lehre und Praxis



# Analyse der Beziehungen von Requirements und Testcases anhand des JHotDraw-Frameworkes

BAKKALAUREATSARBEIT  
(Projektpraktikum)

zur Erlangung des akademischen Grades

BAKKALAUREUS DER TECHNISCHEN WISSENSCHAFTEN

in der Studienrichtung

INFORMATIK

Angefertigt am *Institut für Systems Engineering und Automation*

Betreuung:

*Univ.-Prof. Dr. Alexander Egyed, M.Sc*

*Dipl.-Ing. Benedikt Burgstaller*

Eingereicht von:

*Michael Herceg*

# **Eidesstattliche Erklärung**

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benutzt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 13. Juli 2010

Michael Herceg

# Kurzfassung

Ein hilfreiches Verfahren den Softwareengineering-Prozess zu optimieren, ist die Einführung von Requirementsmatrizen. Diese beinhalten beispielsweise wichtige Szenarien, die mit den dazugehörigen Klassen sowie Methoden verknüpft werden. Somit lassen sich für unterschiedliche Fälle klare Bedingungen sowie Strukturen und deren Dazugehörigkeiten identifizieren.

Ziel dieser Arbeit war es anhand des JHotDraw-Frameworkes diese Requirementsmatrix zur Laufzeit zu überprüfen. Es wurde dafür ein Analyse-Tool entwickelt, welches die Matrix sowie Testcases einliest, und daraus eine Zusammenfassung in verschiedenen Arten darstellt und darüberhinaus auch Operationen zur Kombinierung liefert. Die Testcases wurden für jedes einzelne Requirement mit dem Eclipse-Tracingtool TPTP erstellt.

Anhand der daraus resultierenden Ergebnisse wurde nach markanten Mustern gesucht, die sich durch starke oder sehr schwache Ähnlichkeiten, beschreiben lassen.

Zur besseren Übersicht wurden die Klassen in drei Kategorien eingeteilt: grün - Klassen werden von TPTP aufgezeichnet und werden in der Matrix richtig gelistet (werden unter anderem auch truePositives genannt); orange: Klassen werden in der Matrix gelistet, kommen aber in den TestCase-Fällen nicht vor (falseNegatives); rot: Klassen kommen in den TestCase-Fällen vor, allerdings werden sie zum jeweiligen Requirement in der Matrix nicht gelistet (falsePositives).

Als Ergebnis dieser Arbeit wurde identifiziert, dass keines der erstellten und optimierten TestCases den grünen Bereich zur Gänze sättigen. Desweiteren wurde festgestellt, dass vor allem im roten Sektor viele Klassen immer wieder vorkommen, und somit den Großteil der roten Menge bilden. Im Gegensatz dazu, lässt sich feststellen, dass im orangen Bereich viele unterschiedliche Klassen gelistet werden, und es untereinander nur eine kleine, gemeinsame Schnittmenge gibt.

# Abstract

A useful method to optimize the software engineering process, is the introduction of Requirementsmatrices. These include, for example, key scenarios, which are associated with the related classes and methods. Therefore clear conditions, structures and their corresponding identification can be described for different cases.

The aim of this study was to examine the requirementsmatrix of JHotDraw at runtime. An analysis tool was developed, which reads in the matrix as well as the test cases, and which gives the user the ability to create different combinations and views as well. The test cases were created for each requirement with the Eclipse TPTP Tracingtool.

Based on the results, the goal was to seek for distinctive patterns that can be described by strong or very weak similarities.

For a better overview,the classes were divided into three categories: green - classes are recorded from TPTP and are listed correctly in the matrix (they are also called truePositives); orange: Classes are listed in the matrix, but they are not listed in the TestCases (falseNegatives); red: Classes are listed in the TestCases, but not in the matrix (falsePositives).

As a result of this work it was identified that none of the created and optimized testcases satisfy the green zone in its entirety. Furthermore, it was found that many classes occur mainly in the red sector over again. In contrast, it can be concluded that in the orange area, many different classes are listed.

# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>3</b>
1.1	Einleitung und Motivation . . . . .	3
1.2	Gliederung der Bakkalaureatsarbeit . . . . .	3
<b>2</b>	<b>TPTP</b>	<b>4</b>
2.1	Was ist TPTP? . . . . .	4
2.2	Verwendung in dieser Arbeit . . . . .	5
<b>3</b>	<b>JHotDraw</b>	<b>7</b>
3.1	Was ist JHotDraw? . . . . .	7
3.2	Architektur . . . . .	8
<b>4</b>	<b>Trace</b>	<b>11</b>
4.1	Erste Schritte . . . . .	11
4.2	Testsznarien . . . . .	11
<b>5</b>	<b>Analyse-Tool</b>	<b>15</b>
5.1	Spezifikationen . . . . .	15
5.2	Modell . . . . .	16
5.3	Implementierung . . . . .	19
5.4	Parsererzeugung . . . . .	26
<b>6</b>	<b>Analyse von JHotDraw</b>	<b>28</b>
6.1	Erster Durchlauf . . . . .	28
6.2	Zweiter Durchlauf . . . . .	31
6.2.1	Testcases . . . . .	31
6.2.2	Unterschiede zum ersten Durchlauf . . . . .	36
6.3	Verfeinerung des Analyse-Tools . . . . .	37
6.3.1	Neue View: Beziehung zw. Requirement und Testcase . . . . .	37
6.3.2	Neue View: Prozentuelle Überlappung . . . . .	38
6.4	Verfeinerung der Ergebnisse mittels Kombinationen . . . . .	39
6.5	Automatisierung der Optimierung . . . . .	40
6.6	Gegenüberstellung aller erstellten Testsznarien . . . . .	42
6.7	Recall,Precision und weitere Merkmale . . . . .	45

<b>7 Zusammenfassung</b>	<b>48</b>
7.1 Zusammenfassung . . . . .	48

## Kapitel 1

# Einleitung

### 1.1 Einleitung und Motivation

Ein wichtiger Bestandteil des Softwareengineering-Prozesses ist die Erstellung einer sogenannten Requirementsmatrix. Diese beinhaltet wichtige Szenarien, die mit den dazugehörigen Klassen sowie Methoden verknüpft werden. Somit lassen sich für unterschiedliche Fälle klare Bedingungen sowie Strukturen und deren Dazugehörigkeiten identifizieren.

Ziel dieser Arbeit war es anhand des JHotDraw-Frameworkes diese Requirementsmatrix zur Laufzeit zu überprüfen. Es wurde dafür ein Analyse-Tool entwickelt, welches die Matrix sowie Testcases einliest, und daraus eine Zusammenfassung in verschiedenen Arten darstellt und darüberhinaus auch Operationen zur Kombinierung liefert. Die Testcases wurden für jedes einzelne Requirement mit dem Eclipse-Tracingtool TPTP erstellt.

Anhand der daraus resultierenden Ergebnisse wurde nach markanten Mustern gesucht, die sich durch starke oder sehr schwache Ähnlichkeiten, beschreiben lassen.

### 1.2 Gliederung der Bakkalaureatsarbeit

In Kapitel 2 und 3 werden TPTP und JHotDraw näher erklärt; unter anderem wird näher auf ihre Funktion eingegangen. Kapitel 4 geht auf die Requirementsmatrix von JHotDraw ein, und analysiert die nötigen Testcases, die dafür erstellt werden müssen. Kapitel 5 beschäftigt sich mit dem Analyse-Tool, das für diese Arbeit entworfen und implementiert wurde. In Kapitel 6 erfolgt die ausführliche Analyse von JHotDraw und stellt drei verschiedenen erzeugte Testszenarien gegenüber.

Kapitel 7 ist eine kurze Zusammenfassung der Bakkalaureatsarbeit.

## Kapitel 2

# TPTP

### 2.1 Was ist TPTP?

Das TPTP Projekt ist eine OpenSource-Plattform mit implementierten Werkzeugen für Testen, Tracen und Monitoring von Software Systemen. Hierbei wird versucht mit diesen Tools eine automatisierte Variante der Qualitätsüberprüfung anzubieten, die darüberhinaus auch einen hohen Grad an Flexibilität aufweist, was Erweiterung und Pluginentwicklung betrifft.

TPTP basiert auf dem Eclipse-Framework (eine freie Entwicklungsumgebung) und bietet somit die Möglichkeit zur Intergration anderer Tools unter der Verwendung der Eclipse-Umgebung. Hierbei werden Daten durch einen OMG-definierten Trace ausgetauscht.

Das Gesamtpaket, das TPTP ausmacht, besteht im Grunde aus vier unterschiedlichen Modulen, die miteinander kommunizieren können:

- Plattform
- Trace And Profiling
- Testing
- Monitoring

In dieser Arbeit wird hauptsächlich mit dem Unterprojekt von TPTP gearbeitet, welches für Tracing und Profiling zuständig ist.

Mit sogenannten Agent Controllern kann man das Trace-Verhalten beeinflussen und den eigenen Bedürfnissen anpassen. Beispielsweise ist es in dieser Arbeit sehr wichtig



bestimmen zu können, ab wann TPTP zum Tracen starten und wann dieser Vorgang beendet werden soll. Für diese Problemstellung gibt es bereits vorintegriert ein Play/Pause-System, das mit einfachen Mausklicks den Start- und Endzeitpunkt eines Trace-Vorganges definiert.

## 2.2 Verwendung in dieser Arbeit

Agent Controller sind Daemon-Prozesse, die aufseiten des Testsystems und der TPTP Workbench lokalisiert sind. Unter Linux ist es ratsam diese Controller manuell als root zu starten, da es sonst zu Problemen in der Ausführung kommen kann. Beispielsweise wird der von TPTP automatisch registrierte Agent Controller richtig erkannt, lässt aber kein Tracen zu, da keine Verbindung zum Testprogramm erstellt werden kann. Das manuelle Starten wird simpel mit der Ausführung der Datei ACStart.sh durchgeführt, welches sich im Unterverzeichnis *agent controller/bin* der TPTP-Installation befindet.

**Execution Statistics**

Filter: Default Filter. Click [here](#) to set filter

>Class	Package	Base Time (se)
▶ AbstractAttributedComp	# org.jhotdraw.draw	1,04 %
▶ AbstractAttributedFigure	# org.jhotdraw.draw	21,76 %
▶ AbstractBean	# org.jhotdraw.beans	1,95 %
▶ AbstractCompositeFigure	# org.jhotdraw.draw	0,15 %
▶ AbstractCompositeFigure	# org.jhotdraw.draw	0,04 %
▶ AbstractConnector	# org.jhotdraw.draw	0,28 %
▶ AbstractDrawing	# org.jhotdraw.draw	0,05 %
▶ AbstractFigure	# org.jhotdraw.draw	1,37 %
▶ AbstractHandle	# org.jhotdraw.draw	1,49 %
▶ AbstractLocator	# org.jhotdraw.draw	0,01 %
▶ AbstractSelectedAction	# org.jhotdraw.draw.action	1,44 %
▶ AbstractSelectedActions	# org.jhotdraw.draw.action	2,84 %
▶ AbstractTool	# org.jhotdraw.draw	2,49 %
▶ AlignAction	# org.jhotdraw.draw.action	0,10 %
▶ AttributeAction	# org.jhotdraw.draw.action	1,79 %
▶ AttributeKey	# org.jhotdraw.draw	18,04 %
▶ AttributeKeys	# org.jhotdraw.draw	7,00 %

Abbildung 2.1: TPTP-Klassenstatistik

Ist der Agent Controller nun gestartet und richtig registriert, kann das Tracen im Eclipse gestartet werden. Wie vorhin bereits erwähnt, kann der Startzeitpunkt definiert wer-

den, um unnötige Klassen und Methoden nicht mitzuschneiden und somit eine erste Filterung durchzuführen.

Ist der gewünschte Zeitabschnitt fertig aufgezeichnet, wird mittels Doppelklick auf den Agentcontroller die Statistikansicht aufgerufen, in der alle vorgekommenen Klassen und Methoden gelistet werden. Für Detailanalysen werden auch sämtliche Prozentangaben angezeigt, die sich hauptsächlich mit der prozentuellen Aufteilung der verschiedenen Klassen und Methoden auseinandersetzt.

Für diese Arbeit ist nur der Unterpunkt Klassen interessant, da die Tracematrix von JHotDraw nur Klassen und keine Methoden beinhaltet.

Um diese Zusammenfassung in eine XML-Datei abspeichern zu können, muss ein Report erstellt werden. Die abgespeicherte Datei hat eine eindeutig definierte Struktur, die für eine Beispielklasse folgendermaßen aussieht:

```
<data Class="AbstractBean" Package="org.jhotdraw.beans"
Base_Time_.seconds.="1,95 %" Delta:_Base_Time_.seconds.="0,000000"
Average_Base_Time_.seconds.="0,01 %" Cumulative_Time_.seconds.="2,94 %"
Delta:_Cumulative_Time_.seconds.="0,000000" Calls="0,15 %"
Delta:_Calls="0"/>
```

Für weitere Analysen könnte man auch beispielsweise die Dauer der Ausführung miteinander beziehen, aber in diesem Fall ist nur der Name der mitgezeichneten Klasse relevant.

## Kapitel 3

# JHotDraw

### 3.1 Was ist JHotDraw?

JHotDraw ist ein Framework, das sich auf die visuelle Ausgabe von technischen Zeichnungen spezialisiert hat. Einige Funktionen, die angeboten werden, sind unter anderem die Möglichkeit zwischen verschiedenen Views auszuwählen, eine eigene Tool-Palette, benutzerdefinierte Figuren, Speicher- und Ladefunktion für erstellte Grafiken und eine Druckfunktion. Selber basiert JHotDraw auf dem JFC Swing Framework, das um einige Erweiterungen erweitert wurde.

Das Ziel der Entwickler war es eine strukturell einfach gehaltene Basis zu schreiben, die es anderen Software-Entwicklern ermöglicht schnell und effizient eigene Zeichenprogramme zu entwickeln - basierend auf dem JHotDraw-Framework. Mit ein wenig Erfahrung ist es auch möglich die vorhandenen Funktionen zu erweitern, oder gänzlich umzubauen. Das Hinzufügen von gänzlich neuen Funktionen ist ebenfalls möglich.

Ursprünglich wurde JHotDraw von Kent Beck und Ward Cunningham in Smalltalk geschrieben, und später auf Java umgestellt. Das Projekt war eines der ersten, welches explizit als Framework definiert wurde und von Beginn an für die Wiederverwendung durch andere Software-Entwickler gedacht war. Desweiteren war JHotDraw auch eines der ersten Projekte, das sich ausführlich mit Design Pattern auseinandergesetzt hat und somit auch eines der Grundpfeiler in der Design Pattern-Gemeinschaft wurde.

Einige Beispielprojekte, die JHotDraw verwenden, sind:

- JARP - Visualisiert Petrinetze und bietet die Möglichkeit zur interaktiven Echtzeitsimulation.
- Joone - Neutrales Netz-Framework, das mit eigenen Algorithmen erweitert werden kann.

- JStock - Ein real-time Aktienüberwachungstool.

## 3.2 Architektur

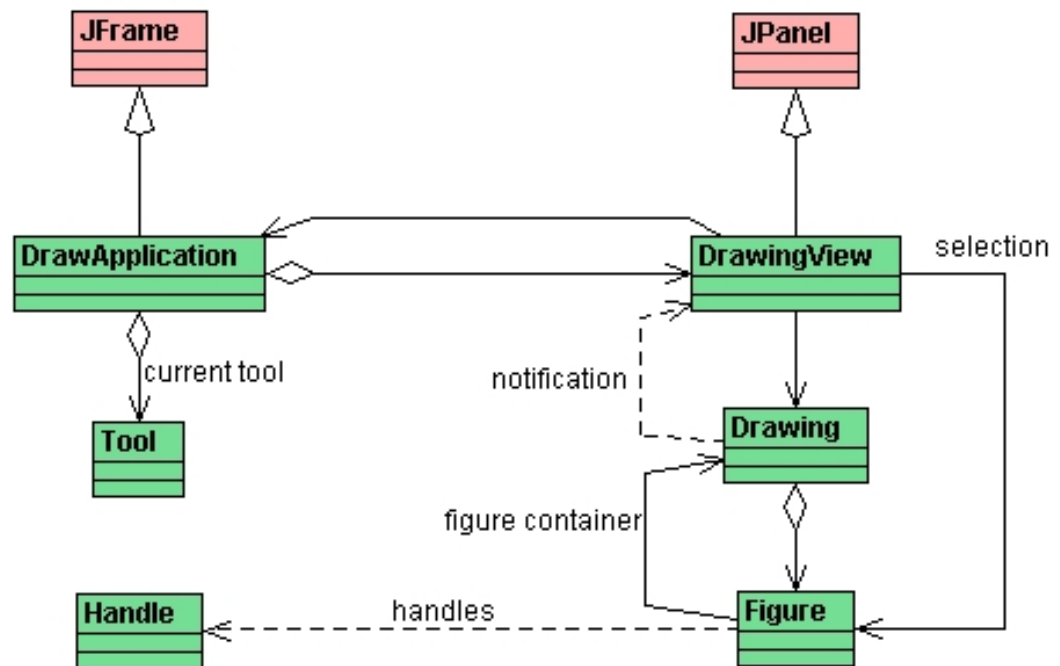


Abbildung 3.1: Architektur von JHotDraw

Jede auf JHotDraw basierende Software besitzt eine DrawWindow-Klasse, die eine Unterklasse von `javax.swing.JFrame` ist. Je nachdem wieviele verschiedene Views in Verwendung sind bzw. vorgesehen sind, kann die Anzahl der internen Frames von Projekt zu Projekt unterschiedlich sein. In `DrawingView` werden die Grafiken angezeigt und parallel dazu neuer Input durch den Benutzer generiert, der zur Erzeugung neuer Grafiken führen kann. Solche Veränderungen in `Drawing` werden von `DrawingView` registriert und verwaltet.

Ein sogenanntes `Drawing` besteht aus mehreren `Figure`s, die mit `Handles` untereinander kommunizieren können. Zum Beispiel welche `Figure`s miteinander in direkter Beziehung stehen oder zur selben Gruppe gehören.

Üblicherweise besitzt `DrawingWindow` genau eine Toolpalette, die das Manipulieren, Erzeugen und Löschen der unterschiedlichen `Figure`s und Grafiken ermöglicht.

Dadurch, dass JHotDraw auf dem MVC-Prinzip (Model View Controller) basiert, ist

die Programmlogik strikt von der visuellen Ausgabe getrennt und lässt für Neuentwicklungen viele Freiräume, um aufbauend auf dem eigenen Modell einen Editor erstellen zu können.

Solch ein Modell lässt sich simpel in einer einzigen Klasse (JModellerClass) realisieren. Für jedes Modell wird ein GraphicalCompositeFigure gebildet, das aus mehreren Figures bestehen kann.

Mittels der CompositeFigure-Klasse lassen sich mehrere Figuren in eine einzelne zusammenfassen. Die Idee hinter diesem Composit-Prinzip ist, dass ein Container aus mehreren Komponenten des selben Types besteht, allerdings wie eine einzelne Komponente behandelt wird. Das heißt eine abgeleitete Komponente verwendet eine Menge von Elementen anstatt einer einzigen Komponente.

Diese hierarchische Strukturierung wird unter anderem in CompositeFigure definiert. Hierbei ist zu berücksichtigen, dass CompositeFigure keine eigene visuelle Darstellung besitzt, sondern lediglich die Kindfiguren, die vom Benutzer im Zeichenfenster definiert wurden, darstellt.

Wenn ebenfalls mit AttributeFigure gearbeitet wird, können in der GraphicalCompositeFigure-Klasse verschiedene Attribute der Figuren und Texte beeinflusst werden, beispielsweise Farbe, Schriftart usw.

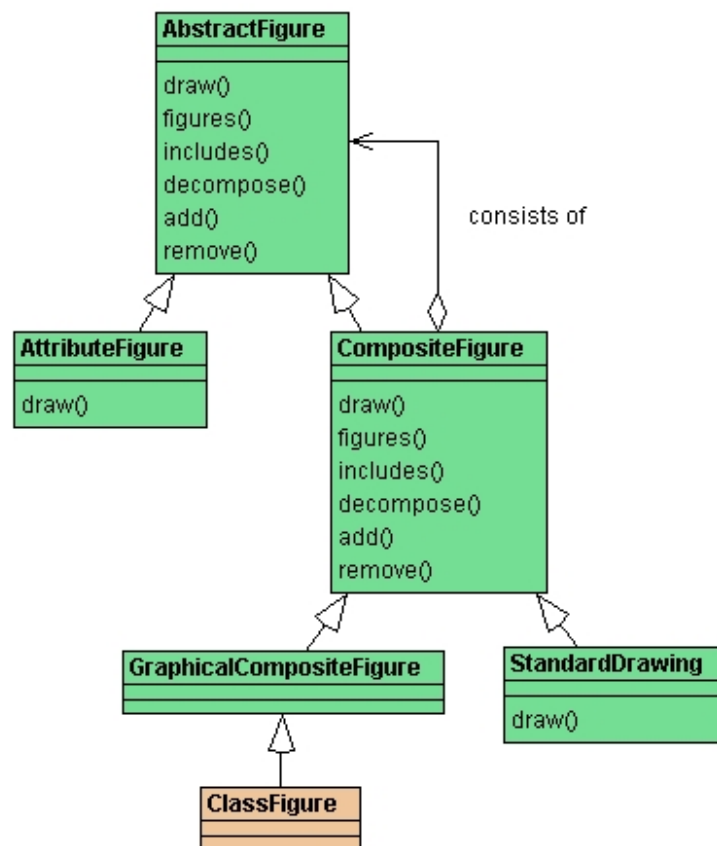


Abbildung 3.2: Figure-Architektur

## Kapitel 4

# Trace

### 4.1 Erste Schritte

Nachdem JHotDraw ein Framework und keine fertige Applikation ist, muss zuerst ein geeignetes Programm basierend auf diesem Framework realisiert werden. Für diese Zwecke gibt es allerdings bereits eine fertig-implementierte Test-Applikation, die von JHotDraw zur Verfügung gestellt wird. Sämtliche relevanten Methoden und Funktionen werden in diesem Testprogramm eingebunden und dementsprechend in der Toolbox zur Verfügung gestellt.

Die für diese Arbeit verwendete Tracematrix besteht aus 21 unterschiedlichen Test-szenarien. Hauptaugenmerkmal wurde hierbei auf die Funktionalität gelegt (beispielweise Figuren zeichnen, löschen und bearbeiten). Allerdings werden auch Usability und Performance berücksichtigt, die in dieser Arbeit aber nicht näher analysiert wurden.

Für diese einzelnen Szenarien muss jeweils eine Tracedatei mittels TPTP generiert werden. Zu beachten ist, dass versucht werden sollte tatsächlich nur diesen Testbereich mitzuschneiden, was sich bei einigen wenigen Szenarien als nicht ganz trivial herausstellt.

Neben den 21 Testszenarien sind in der Tracematrix insgesamt 546 Klassen gelistet, die allesamt beim Tracen vorkommen sollten. Die Zugehörigkeit einer Klasse zu einem Szenario wird simpel mit einem  $x$  in der jeweiligen Spalte definiert.

### 4.2 Testszenarien

Für diese Arbeit wurde die Standard-Testapplikation verwendet, die bereits fertig-implementiert von JHotDraw zur Verfügung gestellt wird (siehe Abbildung 3.1).

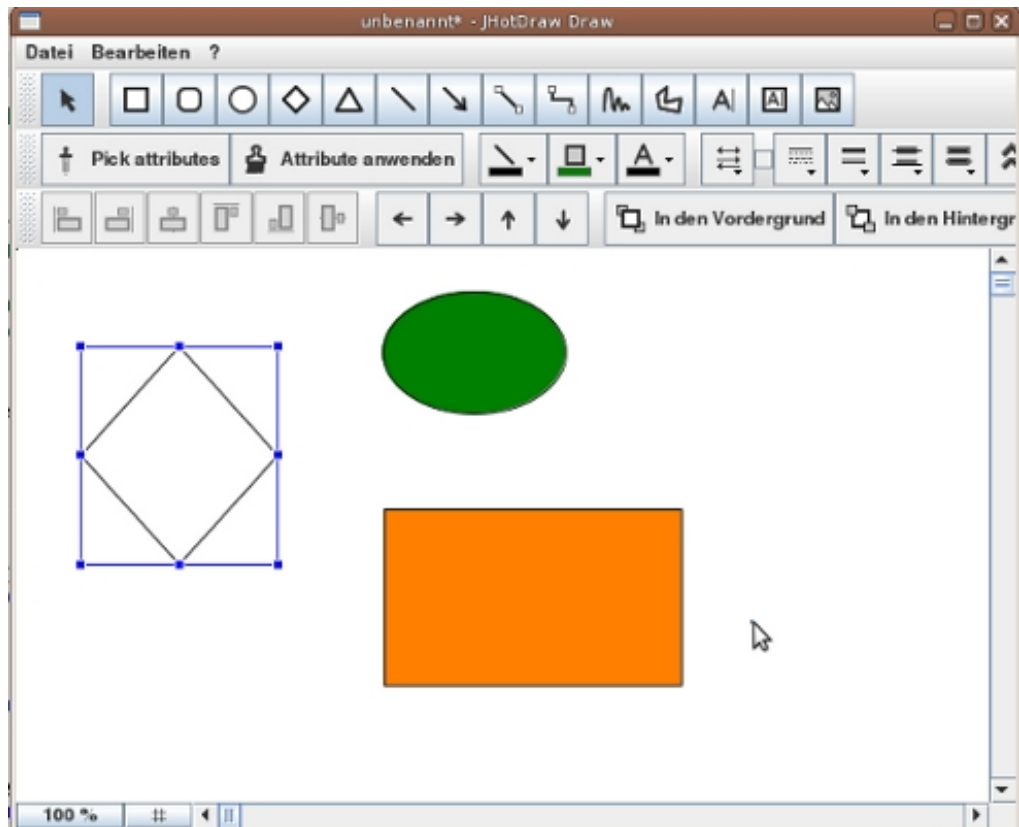


Abbildung 4.1: Testapplikation

Die folgenden Punkte beschäftigen sich mit den einzelnen Testszenarien und wie diese mittels TPTP getracet wurden. Eines der Hauptziele hierbei war es auch nur den tatsächlichen Testbereich aufzuzeichnen.

- **R01-Paint figures and connections**

Verschiedene vordefinierte und benutzerdefinierte Figuren zeichnen. Danach miteinander mittels einer Connection verbinden.

- **R02-Create figures and connections**

Benutzerdefinierte Figuren zeichnen. Danach miteinander mittels einer Connection verbinden.

- **R03-Delete figures**

Bereits erstellte und gezeichnete Figuren und Grafiken wurden der Reihe nach gelöscht.

- **R04-Delete connections**

Sämtliche vorher definierten Connections (zu unterschiedlichen Figurtypen) wurden der Reihe nach gelöscht.



- **R05-Select and deselect figures**

Verschiedene Figuren wurden angeklickt, um sie somit zu selektieren, und wieder deselektiert.
- **R06-Move, scale, or rotate figures**

Alle gezeichneten Figuren wurden mit der Maus verschoben, skaliert und rotiert, sodass jedes der Elemente zumindest einmal alle drei Operationen aufzeichnen konnte.
- **R07-Change properties of figures**

Die Eigenschaften aller Figuren wurden verändert, beispielsweise die Dicke des Randes.
- **R08-Edit text in figures**

Vorher eingefügter Text wurde nun editiert, indem der Inhalt, die Schriftart und die Farbe verändert wurden.
- **R09-Align figures**

Alle Figuren wurden einzeln, als auch in Gruppen unterschiedlich positioniert (links, zentriert, rechts, etc.)
- **R10-Change z-ordering**

Das sogenannte *z-ordering*, das für die Darstellung bei überlappenden Figuren zuständig ist, wurde für alle Figuren mehrmals von Vorder- zu Hintergrund umgestellt.
- **R11-Grouping and ungrouping figures**

Alle vorhandenen Figuren wurden willkürlich miteinander gruppiert, degruppiert und wieder zu neuen Gruppen hinzugefügt.
- **R12-Launch the application**

Das Testprogramm wurde gestartet. Hierbei ist zu beachten, dass TPTP bereits von Start an zum Aufzeichnen anfangen muss und schließlich manuell beendet wurde, sobald das Testprogramm vollständig geladen war (+ zusätzlich dazu noch wenige Sekunden Toleranzzeit, um sicher zu gehen, dass auch alle relevanten Klassen aufgezeichnet werden).
- **R13-Open an existing drawing**

Ein vorher abgespeichertes Bild wurde geöffnet. Für dieses Szenario wurde das Testprogramm nach dem Speichern beendet und neu gestartet, um ganz sicher zu gehen, dass keine unerwünschten Klassen mit in den Trace aufgenommen werden.

- **R14-Save a drawing**

Ein Bild, das aus mehreren unterschiedlichen Figuren bestand (mit unterschiedlichen Farben, Linienarten und Textfeldern), wurde in eine neue Datei auf der Festplatte abgespeichert.

Alle weiteren Szenarien bzw. Requirements sind fokussiert auf Performance und Usability, weshalb sie in dieser Arbeit und Analyse nicht näher eingebunden wurde, da das Hauptaugenmerk auf Funktionalität gelegt wird.

Zu jedem dieser getesteten Testszenarios wurde in Eclipse in der sogenannten *Profiling and Logging-View* ein Report erstellt, der als XML-Datei abgespeichert wurde.

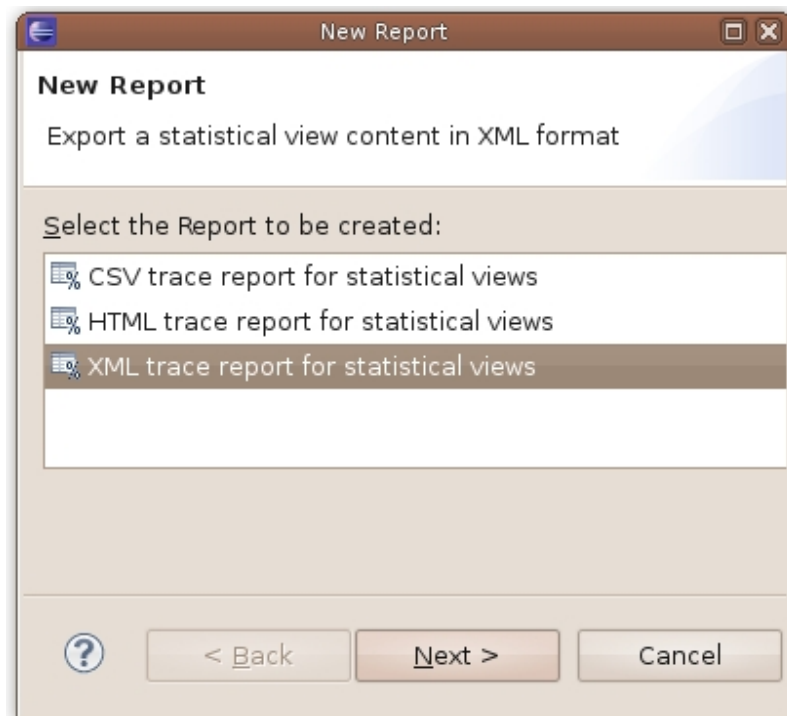


Abbildung 4.2: TPTP-Report

## Kapitel 5

# Analyse-Tool

### 5.1 Spezifikationen

Das Analyse-Tool, das für diese Arbeit entwickelt wurde, muss verschiedene Anforderungen erfüllen, die vor allem im Bereich der Testszenarienanalyse fokussiert sind. Darüber hinaus, müssen folgende Eigenschaften in die Planung und Entwicklung eingebunden werden:

#### Laden von Daten

Es muss möglich sein die Tracematrix (die in Form einer Excel-Datei vorliegt), sowie die verschiedenen Tracereports (in Form von XML-Dateien) zu laden. Hierbei sollte eine Funktion angeboten werden, um einen ganzen Ordner auswählen zu können, um sich mühsames Laden von vielen, einzelnen Tracereports zu ersparen.

#### Speichern von Daten

Neu erstellte Szenarien sollen abgespeichert werden können, um einerseits den Grad der Usability zu erhöhen und andererseits die Möglichkeit anzubieten, später mit den erstellten Szenarien weiterarbeiten zu können.

#### Verknüpfungen

Die einzelnen Requirements, die in der Tracematrix definiert sind, müssen mit den von TPTP aufgezeichneten Daten verknüpft werden können. Beispielsweise sollen neue Testszenarien in Form von  $R01, R02+R03 \rightarrow T01+(T02+T03)$  erzeugt werden können. Demzufolge müssen auch unterschiedliche Operationen zur Verfügung gestellt werden, die selektierte Sets miteinander verknüpfen, subtrahieren und je nach Wünschen zusammenschneiden.

### Visuelle Ausgabe

Die neu generierte Testszenarien müssen in einer Form visuell dargestellt werden, um bei der Analyse einen ersten Eindruck zu erhalten, wie sich die definierten Szenarien mit den tatsächlichen Requirements abdecken.

### Statistik

Darüberhinaus sollte eine simple Statistik erstellt werden, die dem Benutzer in einer simplen Art darstellt, welche Klassen am häufigsten im richtigen oder falschen Bereich aufzufinden sind.

## 5.2 Modell

Die Struktur des Tools ist nach dem MVC-Prinzip in unterschiedliche Teile aufgesplittet. (siehe Abbildung 4.1)

### Model

Die Klasse Model ist für die logische Ausarbeitung und Verwaltung der Daten zuständig. Hier werden vor allem die vom Benutzer erstellten Testszenarien gespeichert und entsprechende Operationen angeboten, die dem Benutzer erlauben verschiedene Sets miteinander zu verknüpfen.

Zwei wichtige Elemente in der Klasse *Model* sind:

- Ein Set zur Speicherung von allen geladenen Requirements (von der Tracematrix) und ein weiteres für Testcases.
- Eine Liste von allen neu erstellten Testszenarien, die vom Benutzer generiert werden. Beispielsweise:  $R01, R02+R03 \rightarrow T01+(T02+T03)$ , wobei  $R$  für Requirements und  $T$  für Testcases stehen.

Eine weitere Klasse im Model-Bereich ist *Scenario*, welche im Prinzip aus einem Set von Strings besteht (zur Speicherung der Klassennamen) und einigen Methoden zur Sortierung, Verwaltung, etc. besteht.

*TestCase* ist verantwortlich für die Speicherung und Verwaltung der erzeugten Testszenarien. Dabei werden zwei verschiedene Sets verwendet:

- *actual*: Speichert die Klassen, die in diesem Szenario vorkommen sollten. Definiert nach der Tracematrix, die von JHotDraw vorgegeben ist.

- traced: Hier befinden sich die Klassen, die von TPTP für dieses Szenario aufgezeichnet wurden und somit tatsächlich im finalen Programm auch vorkommen.

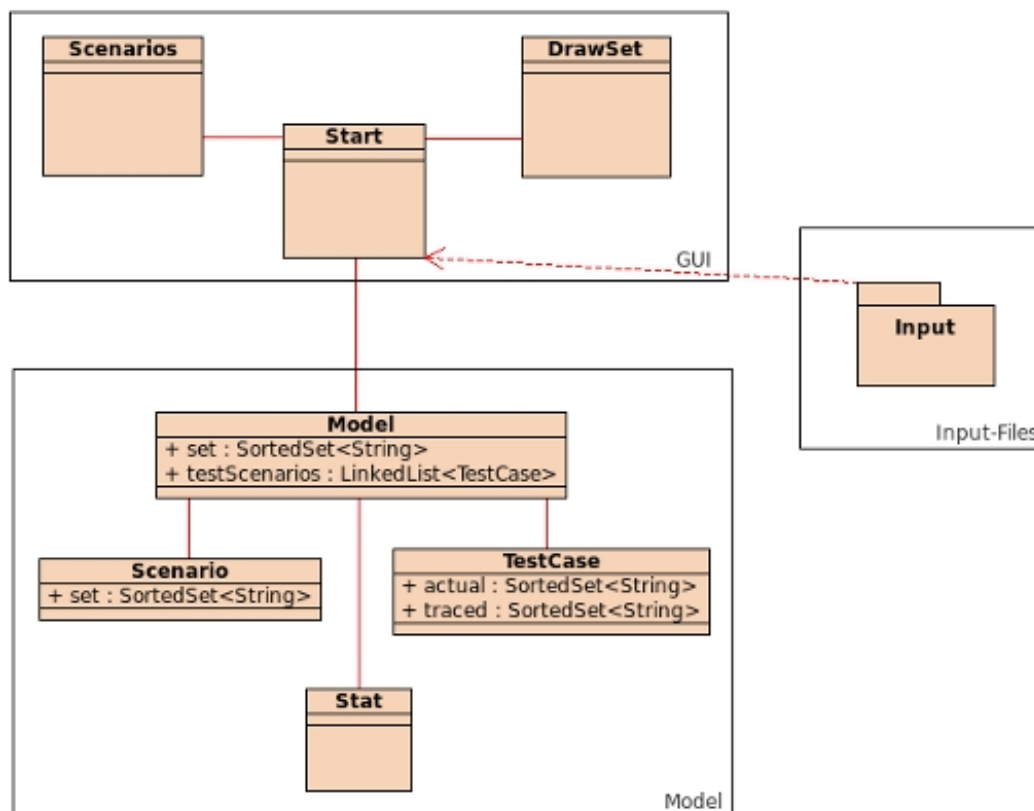


Abbildung 5.1: Basis-Struktur

## GUI

Der GUI-Bereich ist zuständig für die visuelle Ausgabe der Analyse sowie die eigentliche grafische Benutzeroberfläche. Dabei gibt es wieder unterschiedliche Bereiche, die unterschieden werden müssen:

- Die eigentliche Benutzeroberfläche (Klasse *Start*): Soll minimalistisch und einfach gehalten werden, um Verwirrungen von Beginn an zu reduzieren.
- Tabellen: Es werden zwei verschiedene Sorten von Tabellen zur Verfügung gestellt. Die erste Tabelle beinhaltet sämtliche vorkommenden Klassen und klassifiziert diese in drei Bereiche:
  - grün: Die Klasse kommt sowohl im (von TPTP) aufgezeichneten Testcase vor, als auch in der Tracematrix.

- orange: Die Klasse kommt nicht im aufgezeichneten Testcase vor, allerdings in der Tracematrix.
  
- rot: Die Klasse kommt im aufgezeichneten Testcase vor, allerdings nicht in der Tracematrix.

Die zweite Tabelle beinhaltet eine statistische Auswertung der Testszenarien. Diese besteht aus drei Spalten:

- grün: Alle Klassen, die im grünen Bereich vorkommen, mit der Anzahl der Häufigkeit.
  - orange: Alle Klassen, die im orangen Bereich vorkommen, mit der Anzahl der Häufigkeit.
  - rot: Alle Klassen, die im roten Bereich vorkommen, mit der Anzahl der Häufigkeit.
- DrawSets: Die unterschiedlichen Testszenarien werden der Reihe nach grafisch in Form von Balken dargestellt, wobei die unterschiedlichen Bereiche (grün, orange, rot) in den entsprechenden Farben eingefärbt werden.

### Input

Es gibt zwei verschiedene Arten von möglichen Input-Files:

- Tracematrix: Ist die Matrix mit den vorgegebenen Requirements und den dazugehörigen Klassen.
- Testcases: XML-Files, die mittels TPTP generiert werden und alle Klassen beinhalten, die zum jeweiligen Testcase gehören.

## 5.3 Implementierung

Entwickelt wurde das Analyse-Tool mit Java (*java-1.5.0-gcj-4.4*, Linux) und unter der Verwendung der Eclipse- Entwicklungsumgebung. Hauptfokus wurde auf eine einfache Bedienung sowie ein simples Interface gelegt, das im Prinzip nur aus einer Tool-Palette und drei Unterfenstern besteht, die die Analysen in unterschiedlichen Arten darstellen. Zu den Unterpunkten der Implementierung:

### Grundgerüst

Die wichtigste Aufgabe des Grundgerüsts ist es die unterschiedlichen Testcases (von TPTP-gelieferte XML-Reports), Requirements (von der Tracematrix gelieferte Informationen zu den verschiedenen Zugehörigkeiten zwischen Klassen und einem einzelnen Requirement) und die vom Benutzer erstellten Testszenarien zu verwalten. Hierbei wird

grundsätzlich mit Sets gearbeitet, da diese bereits mit allen nötigen Funktionen ausgestattet sind.



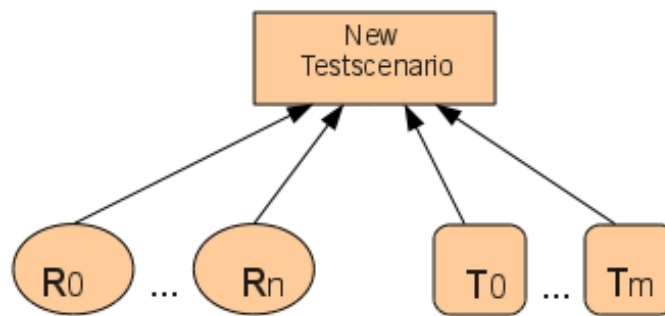


Abbildung 5.2: Neues Testscenario generieren

Demzufolge besteht ein Testscenario aus zwei verschiedenen Elementen (siehe Abbildung 4.2):

- Requirements: Sind Sets von Klassen, die laut der Tracematrix einem bestimmten Requirement zugeordnet sind.
- Testcases: Von TPTP-erzeugte Aufzeichnungen, die dem vorgegebenen Szenario entsprechen.

Wichtig ist hierbei, dass die Requirements und Testcases untereinander verschieden verknüpft werden können: Addition, Differenz und Schnitt. Desweiteren ist es auch möglich mittels Klammerungen unterschiedliche Kombinationen zu erreichen. Beispielsweise werden bei einer Eingabe von  $((R01-R03)+R04)$  alle Klassen von R03 dem Set R01 abgezogen, und danach wird R04 dazuaddiert. Näheres dazu in Kapitel 4.4.

## GUI

Die Toolpalette der grafischen Benutzeroberfläche besteht aus sechs Auswahlmöglichkeiten (siehe Abbildung 4.3.):

- **1:** Tracematrix in einem Ordner auswählen und in das Modell laden.
- **2:** Testcases im XML-Format laden. Entweder alle einzeln, oder einen ganzen Ordner laden.
- **3:** Das Menü zum Erstellen von Testscenarios aufrufen.
- **4:** Alle erstellten Testscenarios in eine XML-Datei speichern.
- **5:** Gespeicherte Testscenarios laden. Ein bereits existierendes Set aus Testscenarios wird hierbei mit dem geladenen Set ergänzt.

- **6:** Das Analyse-Tool beenden.

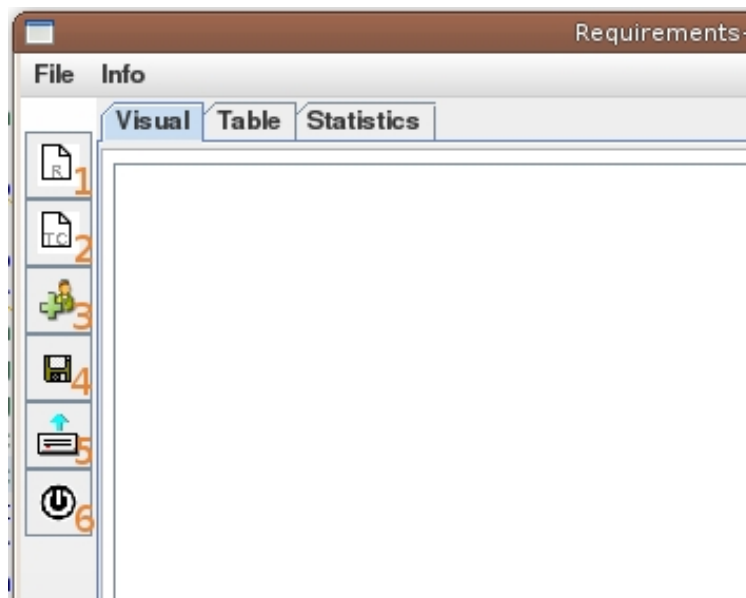


Abbildung 5.3: Toolpalette

Zum Erstellen von neuen Testszenarien gibt es ein eigenes Fenster, das verschiedene Funktionen bereitstellt (Siehe Abbildung 4.4).

In der linken Tabelle können mehrere Requirements selektiert werden und mit einem Klick auf dem Button rechts daneben mit einem oder mehreren selektierten Testcases kombiniert werden. Die neu erschaffenen Testszenarien erscheinen dann in der rechten Tabelle. Zu berücksichtigen ist hierbei, dass bei Mehrfachselektion von Requirements bzw. Testcases der Additionsoperator als Verküpfungsbasis ausgewählt wird. Beispielsweise generiert das Tool bei selektierten Testcases  $T01$ ,  $T03$ ,  $T04$  folgenden neuen Testcase:  $T01+T03+T04$ .

Will man komplexere Kombinationen durchführen, kann man dies in den darunterliegenden Feldern eingeben. Ein eigens dafür erzeugter Parser (siehe Kapitel 4.4) ist in der Lage Verschachtelungen aufzulösen und die Operationen in richtiger Reihenfolge durchzuführen.

Die Reihenfolge der eingegebenen Elemente spielt eine wichtige Rolle. Beispielsweise führen  $R01-R02$  und  $R02-R01$  zu völlig unterschiedlichen Ergebnissen:

R01	R02	R01-R02	R02-R01
Class 1	Class 3	Class 1	Class -
Class 3	Class 2	Class -	Class 2
Class 4	Class 5	Class 4	Class 5

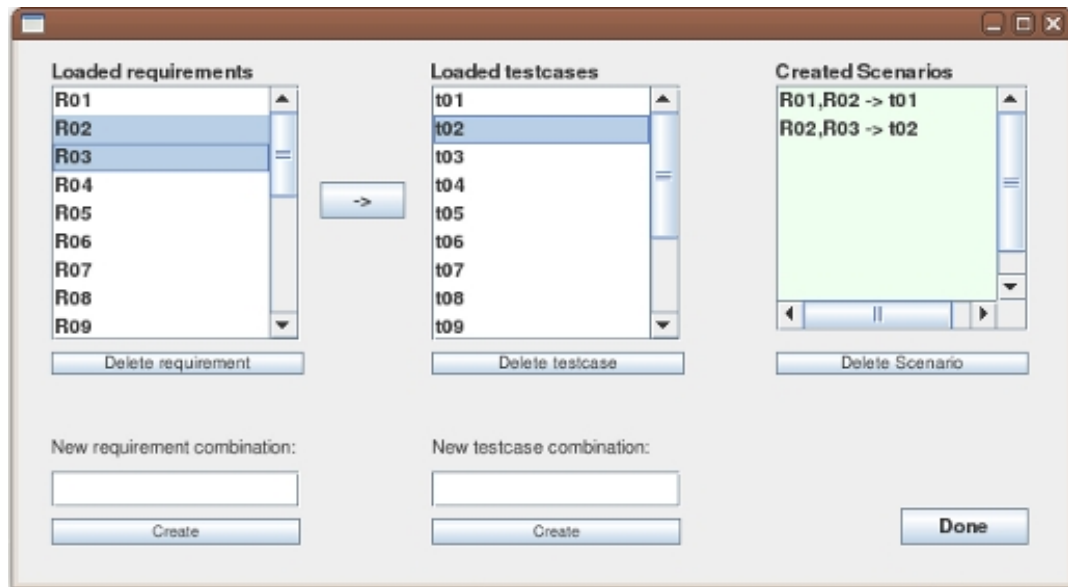


Abbildung 5.4: Generierung von neuen Testscenarien

### Grafische Ausgabe der Analyse

Es gibt zwei verschiedene Arten, wie die Testscenarien dargestellt werden:

- Als Grafik in Form von Balken (siehe Abbildung 4.5). Für jedes Testscenario wird ein eigenes Rechteck gezeichnet, das je nach prozentueller Gewichtung der verschiedenen Bereiche (grün=OK, orange und rot = nicht OK) unterschiedlich eingefärbt werden.
- Anhand einer eingefärbten Tabelle (siehe Abbildung 4.6). Es werden alle auftretenden Klassen aufgelistet, die je nach ihrer Zugehörigkeit in den Spalten rechts daneben farbig dargestellt werden, wobei *weiß* gar keine Zugehörigkeit zu diesem Testscenario symbolisiert.
- Eine dritte Darstellungsform ist ebenfalls eine Tabelle, die allerdings zusätzlich zur Zugehörigkeit zu einem Szenario auch die Anzahl des Vorkommens einer Klasse auflistet, und somit als statistische Hilfe betrachtet werden kann.

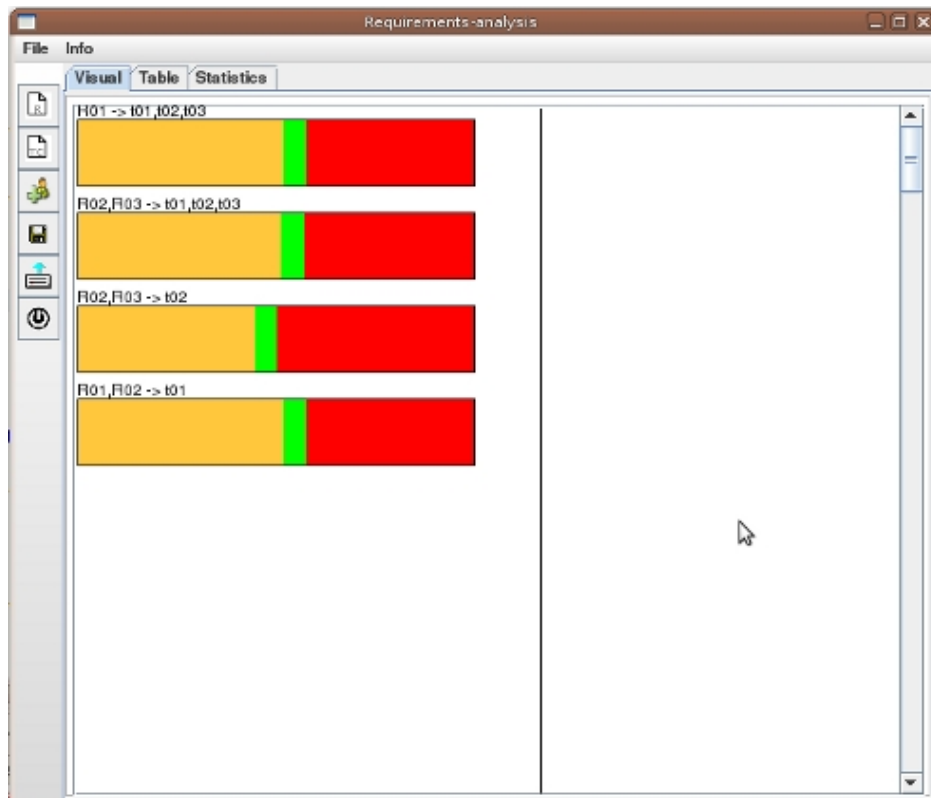


Abbildung 5.5: Sets in gezeichneter Form

Classes	R01,R02 -> t01	R02,R03 -> t02	R02,R03 -> t01,t02,t03	R01 -> t01,t02,t03
AbstractAttributedCo...	Green	Yellow	Yellow	Yellow
AbstractAttributedDe...	Green	Yellow	Yellow	Yellow
AbstractAttributedF...	Green	Yellow	Yellow	Yellow
AbstractBean	Red	Red	Red	Red
AbstractCompositeF...	Green	Yellow	Yellow	Yellow
AbstractConnection...	Red	Red	Red	Red
AbstractConnector	Red	Yellow	Green	Red
AbstractDrawing	Red	Red	Red	Red
AbstractFigure	Red	Green	Red	Red
AbstractHandle	Red	Red	Red	Red
AbstractLineDecorat...	Yellow	Red	Red	Yellow
AbstractLocator	Red	Red	Red	Red
AbstractSelectedActi...	Red	Red	Red	Red
AbstractTool	Red	Red	Red	Red
AlignAction	Red	Red	Red	Red
ArrowTip	Yellow	Red	Red	Yellow
AttributeAction	Red	Red	Red	Red
AttributeKey	Red	Red	Red	Red
AttributeKeys	Red	Red	Red	Red
BezierFigure	Green	Red	Red	Red
BezierNodeHandle	Red	Red	Red	Red
BezierOutlineHandle	Red	Red	Red	Red
BezierPath	Red	Red	Red	Red
BezierPathIterator	Green	Red	Red	Green
BezierTool	Red	Yellow	Green	Red
BidirectionalConnec...	Red	Red	Red	Red
BorderRectangleFig...	Yellow	Red	Yellow	Yellow
BoundsOutlineHandle	Red	Red	Red	Red
ChopBezier Connect...	Red	Red	Red	Red
ChopDiamondConn...	Red	Red	Red	Red

Abbildung 5.6: Tabelle mit farbigen Kennzeichnungen

## 5.4 Parsererzeugung

### Was ist JavaCC?

Mittels JavaCC (Java Compiler Compiler) kann man schnell und einfach eigene Parser für Java generieren - dabei wird ein LL-Parser erzeugt. Dementsprechend wird die Eingabe von links nach rechts abgearbeitet.

Die Grammatik wird in EBNF definiert und benötigt keine weiteren Modifikationen. Um die Entscheidung, wie Verschachtelungen expandiert werden sollen, beeinflussen zu können, wird bei JavaCC die Lookahead-Methode verwendet. Man kann diesen Lookahead als optionale Einstellung mit angeben. Diese Einstellungen sowie die gesamte Grammatik werden in eine .jj-Datei geschrieben, welche dann mit dem JavaCC zu einem Parser generiert wird. Hierbei werden mehrere .java-Files erzeugt, die man in das eigene Projekt einbinden muss.

### Generierter Parser für das Tool

```
1 SKIP : { " " | "\t" }
2
3 TOKEN : { < NAME : ([ "a"-"z", "A"-"Z", "0"-"9" ])+ > }
4 TOKEN : { < EOL : "." > }
5
6 Scenario parse() : {
7     Scenario value;
8 }
9 {
10     value=expr()
11     (<EOF> | <EOL>) { return value; }
12 }
13
14 Scenario expr() : {
15     Scenario x;
16     Scenario y;
17 }
18 {
19     x=term()
20     (
21         "+" y=expr() { x.plus(y); }
22     )*
23     { return x; }
24 }
25
26 Scenario term() : {
27     Scenario x;
28     Scenario y;
```

```
29 }
30 {
31   x=name()
32   (
33     "-" y=term() { x.minus(y); }
34     |
35     "/" y=term() { x.intersect(y); }
36   )*
37   { return x; }
38 }
39
40 Scenario name() : {
41   Token t;
42   Scenario value;
43 }
44 {
45   t=<NAME>    { return model.getScenario(t.image); }
46   |
47   "(" value=expr() ")" { return value; }
48 }
```

Die einzelnen Kombinationsmöglichkeiten, beispielsweise der Plusoperator, werden, wie man am Code sehen kann, mit Setoperationen ausgeführt. In diesem Fall  $x.plus(y)$ , wobei  $x$  und  $y$  verschiedene Mengen sind. Desweiteren sind auch tiefe Verschachtelungen mittels Klammern möglich.

## Kapitel 6

# Analyse von JHotDraw

### 6.1 Erster Durchlauf

Mit den in Kapitel 3 beschriebenen Testcases wurden die ersten Testszenarien erstellt. Im ersten Durchlauf wurden zunächst die einzelnen Requirements mit den entsprechenden getrackten Testcases verknüpft, ohne zusätzliche Kombinationen zu erstellen.

Zwei markante Eigenschaften, die man alleine durch das Betrachten der visuellen Ausgabe der Mengen erkennen kann, sind:

- Der grüne Bereich ist bei allen Testszenarien relativ gering. Das liegt vor allem daran, dass der rote Bereich der Mengen bei allen Fällen sehr groß ausfällt und somit den grünen Sektor kleiner aussehen lässt.
- Oranger und roter Bereich sind viel zu groß. Im orangen Sektor befinden sich jene Klassen, die in der Requirementsmatrix aufgelistet wurden und somit auf jeden Fall im getrackten Testcase vorkommen müssen. Im Gegensatz dazu befinden sich im roten Bereich jene Klassen, die von TPTP aufgezeichnet wurden, allerdings nicht in der Requirements Matrix gelistet werden. Das sind zusätzliche Klassen, die vor allem dadurch zustande kommen, dass die Testcases nicht präzise genug angesetzt wurden, bzw. die Beschreibung der Requirements viel zu lapidar beschrieben ist um genau entscheiden zu können, welcher Bereich nun genau zu tracken ist.

In Abbildung 5.1 kann man diese grafische Aufteilung gut für die Testcases 3-6 erkennen.





Abbildung 6.1: Erste Testszenarien

Versucht man den roten Bereich durch Kombinationen mit anderen Testcases zu reduzieren, gelingt dies bei den meisten Fällen. Beispielsweise lässt sich für Requirement 3 folgende Kombination realisieren:  $R03 \rightarrow t03-t02$ . Nachdem im Gegensatz zu  $t03$  der rote Bereich bei  $t02$  relativ gering ausgefallen ist, könnte man mit dieser Tatsache experimentieren und versuchen durch eine Subtraktion diesen Sektor zu reduzieren. Das Resultat ist zufriedenstellend, da fast der gesamte rote Bereich durch diese Kombination eliminiert wurden. Großer Nachteil dieser Methode ist allerdings die Möglichkeit auch für diesen Testcase relevante Klassen so mit zu löschen, was in diesem Fall für zwei Klassen auch geschehen ist.

Was allerdings viel wichtiger ist, als der rote Bereich, ist der relativ große orange Sektor. Das sind schließlich jene Klassen, die auf jeden Fall vorhanden sein sollten und aus gewissen Gründen doch nicht von TPTP aufgezeichnet wurden.

Gründe, die hierbei in Betracht gezogen werden müssen:

- Das Requirement ist in der Requirements Matrix zu ungenau beschrieben. Beispielsweise lässt sich aus  $R01 - Paint\ figures\ and\ connections$  nicht herauslesen, ob es reicht nur wenige Rechtecke und Ellipsen zu zeichnen, oder ob alle vom Framework zur Verfügung gestellten Figuren gemeint sind. Desweiteren ist nicht eindeutig, ob es unterschiedliche Formen der connections gibt. Somit bleibt nur noch die Möglichkeit alle möglichen Figuren beim Aufzeichnen mit TPTP zu zeichnen und erhöht damit auch die Wahrscheinlichkeit für dieses Requirement irrelevante Klassen mitzuzeichnen, was den roten Bereich um Einiges vergrößert.

- Nachdem JHotDraw ein Framework und keine fertige Applikation ist, lässt sich schwer sagen, welche Funktionen für ein gewisses Requirement mit einbezogen werden müssen. In dieser Arbeit wurde mit der Demoapplikation gearbeitet, die von JHotDraw bereits zur Verfügung gestellt wird.

Betrachtet man neben der visuellen Ausgabe der Testszenarienmengen auch die Klassensicht, erkennt man für gewisse Fälle immer wieder-auftretende Muster. Einzelne Klassen, die bei vielen Testfällen nicht aufgezeichnet wurden, aber laut Requirementsmatrix vorhanden sein müssten (siehe Abbildung 5.2.).

Auffallend oft werden Klassen mit dem Präfix ODG oder SVG aufgelistet. Wie es scheint müssen bei den Testcases die unterschiedlichen Dateiformate ebenfalls berücksichtigt werden. JHotDraw bietet nämlich die Möglichkeit die erstellten Grafiken als XML und JPG abzuspeichern. Zusätzlich dazu werden ebenfalls Vektorgrafiken unterstützt (SVG) und das OpenDocument-Format (ODG).

Diese zwei Formate wurden in den ersten Testcase-Durchläufen nicht berücksichtigt und müssen alle neue aufgezeichnet werden. Nachdem dadurch auch andere Panels und Views benötigt werden, ist es von Vorteil eigene Applikationen für jedes dieser Formate zu erstellen. Von JHotDraw werden im Paket *samples* solche Demoapplikationen bereits zur Verfügung gestellt. Um die Testergebnisse zu verfeinern, werden deshalb beim nächsten Trace-Durchlauf drei verschiedene Demoanwendungen aufgezeichnet:

- Hauptanwendung, die Grafiken im XML und JPG-Format handhabt.
- ODG-Format
- SVG-Format

SVGAttributedFigure			ODGAttributedFigure		
SVGBezierFigure			ODGBezierFigure		
SVGCreateFromFileTool			ODGDrawing		
SVGEllipseFigure			ODGDrawingPanel		
SVGFigure			ODGEllipseFigure		
SVGGroupFigure			ODGGroupFigure		
SVGImageFigure			ODGPathFigure		
SVGPathFigure			ODGPathOutlineHandle		
SVGRectFigure			ODGPropertiesPanel		
SVGRectRadiusHandle			ODGRectFigure		
SVGTextAreaFigure			ODGRectRadiusHandle		
SVGTextFigure			Options		

Abbildung 6.2: Ausschnitt aus Klassenansicht

## 6.2 Zweiter Durchlauf

Aus dem ersten Durchlauf wurde ersichtlich, dass bei einigen Testszenarien das ODG- und SVG-Format miteinbezogen werden, jedoch nicht für alle. Um deshalb die Menge an Klassen zu reduzieren, die nicht enthalten sein müssen, werden die Testcases je nach Requirement angepasst und es wird versucht alle möglichen Kriterien zu erfüllen.

Desweiteren gab es auch bei einigen Szenarien gewisse Klassen, die anhand der Beschreibung des Requirements, nicht in den ersten Durchlauf miteinbezogen wurden, die weiter unten näher erklärt werden.

Um optimale Ergebnisse zu erzielen, wurden deshalb alle Requirements einzeln analysiert und folgendermaßen wurde versucht alle relevanten Klassen bei den neuen Traces zu reproduzieren.

### 6.2.1 Testcases

- **R01 - paint figures and connections**

Wie beim ersten Durchlauf wurden sämtliche Zeichenfunktionen ausgewählt. Zusätzlich dazu kommen noch die Vektorgrafik- und OpenDocumentformate hinzu. Was aus der Requirementsbeschreibung nicht hervorgeht, aber in der Requirementsmatrix gelistet ist, sind sämtliche Gruppen-Zeichenfunktionen. Beispielsweise *SVGGroupFigure*. Im Gegensatz dazu, darf die Klasse *GroupAction* nicht vorhanden sein, was das Aufzeichnen der tatsächlich benötigten Klassen erschwert.

Die gezeichneten Figuren wurden mit unterschiedlichen Connections miteinander verbunden.

- **R02 - create figures and connections**

Betrachtet man die Klassenaufflistungen von *R01 - paint figures and connections* und *R02 - create figures and connections*, sind sehr wenige Unterschiede ersichtlich. Die meisten Klassenanforderungen sind zwischen beiden Requirements ident, allerdings gibt es dennoch Unterschiede. *BezierTool* ist beispielsweise bei *R01* nicht gelistet, bei *R02* allerdings schon. Der Name der Klasse lässt darauf schließen, dass *BezierTool* eine der Klassen ist, die für die Darstellung bzw. Verwaltung von Figuren mittels Bezierkurven verantwortlich ist. Inwiefern es aber einen Unterschied zwischen *paint figure* und *create figure* in diesem Zusammenhang gibt, ist nicht ersichtlich und bedarf einer sehr genauen Klassenanalyse. *BezierTool* ist hierbei nicht die einzige Klasse mit diesen Eigenschaften, es gibt noch wenige andere, die in *R01* nicht benötigt werden, in *R02* allerdings schon

und der direkte Zusammenhang bzw. Unterschied nicht klar ist.

- **R03 - delete figures**

Alle erstellten Figures und Zeichnungen wurden auf zwei Arten gelöscht: Durch Selektieren und Drücken der Entferntaste auf der Tastatur und durch Auswählen des Menüpunktes *Löschen*. Eine Tatsache, die nur durch eine Analyse der Klassenzugehörigkeit klar wird, ist, dass auch die Cut-Funktion in *R03* miteinbezogen werden muss. Wie auch in R01 und R02 gibt es in diesem Fall einige Klassen, deren Zusammenhang nicht eindeutig ist. Warum beispielsweise die Klasse *FileIconsSample* nötig ist, um Figures zu löschen, ist nicht klar.

Desweiteren werden sämtliche Selektierfunktionen in der Requirementsmatrix ignoriert, die allerdings wichtig zum Löschen von Figures sind, da man ohne eine Selektion dieser Zeichnungen nichts entfernen bzw. löschen kann.

- **R04 - delete connections**

Ähnliche Gestaltung, wie bei *R03*, nur mit dem Unterschied, dass keine Figures, sondern *Connections* gelöscht werden. Auch in diesem Fall werden sämtliche Selektionsfunktionen ignoriert und nicht in der Requirements-Matrix gelistet.

Ebenfalls wie in R03, werden hier die verschiedenen Formate nicht miteinbezogen. ODG und SVG werden nämlich in der Requirements-Matrix nicht gelistet und müssen somit für das Löschen von Figures oder Connections die selben Klassen aufweisen.

- **R05 - select and deselect figures**

Die Figures werden entweder einzeln oder in Gruppen mittels der Shift-Taste selektiert und anschließend wieder deselektiert. Hierbei gibt es beim Tracen keine Unterschiede zum ersten Durchlauf, da keine Unterscheidung zwischen normalen Grafiken und Vektor- bzw OpenDocument-Grafiken vorhanden ist.

- **R06 - move, scale or rotate figures**

Wichtigster Unterschied zum ersten Durchlauf der Testcases ist, dass mit move nicht nur die Bewegung von Figures mittels Maus gemeint ist, sondern auch explizit die move-Buttons auf der Toolpalette verwendet werden müssen, da diese ebenfalls bei den Requirements vorhanden sein müssen.

- **R07 - change properties of figures**

Change properties ist eine Beschreibung die man weitläufig interpretieren kann. Beispielsweise kann man die Strichstärke, Farbe, Form, und Linienart auf verschiedenste Arten umstellen. Interessant ist hierbei zu beobachten, dass nur eine ganz bestimmte Palette an Eigenschaften unter diesem Requirement zusammengefasst werden und nicht alle. Warum beispielsweise die Klasse `BoundsOutlineHandle`, die schließlich für die Linieneigenschaften der Figuren zuständig ist, nicht gelistet wird, ist aus der Beschreibung nicht erkenntlich. Im Gegensatz dazu scheint die Klasse `CanvasToolBar` auf jeden Fall Bestandteil dieses Requirements zu sein, obwohl man auch ohne dieser `ToolBar` Figuren bzw. die Eigenschaften der Figuren ändern kann.

Um sicher zu gehen, wurden deshalb alle möglichen Szenarien durchgespielt, sowohl bei der Hauptapplikation, als auch bei den kleineren Anwendungen, die ODG und SVG unterstützen.

- **R08 - edit text in figures**

Das Interessante bei diesem Requirement ist die Tatsache, dass alle relevanten Klassen zum Editieren von Texten in der Matrix zusammengefasst werden, ausser die des ODG-Formates. In der Testapplikation von ODG lassen sich Texte in Figuren sehr wohl editieren, löschen und bearbeitet, aber gelistet werden diese Klassen trotzdem nicht.

Um sicher zu gehen, wurde trotzdem der ODG-Fall dieses Testcases berücksichtigt und von TPTP mitaufgezeichnet. Allerdings separat, um später den Unterschied analysieren zu können.

- **R09 - align figures**

Nur fünf Klassen sind laut der Requirements-Matrix für die Anpassung von Figuren notwendig. Die einzige Klasse, die im ersten Durchlauf auch tatsächlich aufgezeichnet wurde, war `AlignAction`. Alle anderen Klassen sind nicht direkt für diese Aufgaben zuständig, sondern hauptsächlich für die Verwaltung von Toolbars, die allerdings zu diesem Zeitpunkt bereits alle geladen sind, und somit direkt im Testcase nicht mehr vorkommen.

Der erste Durchlauf hat bei *R09* einen sehr großen roten Bereich aufweisen können, was sich auch hier nicht verhindern lassen wird können, da für die Anpassung von Figuren noch unzählige andere Klassen benötigt werden, beispielsweise jene die für Selektierungen zuständig sind.

- **R10 - change z-grouping**

Bei diesem Testcase gibt es keine Unterschiede zum ersten Durchlauf, nachdem ODG und SVG nicht einbezogen wurden.

- **R11 - group and ungroup figures**

Unterschiedliche Figuren wurden gruppiert und anschließend wieder direkt degruppiert. Aus der Beschreibung des Requirements ist nicht direkt ersichtlich, ob nur gewisse Figuren gemeint sind, oder alle zusammen, von daher wurden sämtliche möglichen Figurenarten getracet.

Interessant ist hierbei die Klasse *SplitAction*, die für die Spaltung von Gruppierungen und Connections zuständig ist, aber beim ersten Durchlauf nicht von TPTP aufgezeichnet wurde, obwohl mehrere Figuren und Figurengruppen degruppiert wurden.

- **R12 - launch application**

Bei diesem Testcase wurden drei verschiedene Applikationen von Beginn an aufgezeichnet und zwar bis zu dem Zeitpunkt, bis alle Toolbars und Paletten geladen waren und zusätzlich dazu wurden in etwa 5 Sekunden Toleranzzeit eingebaut. Demoapplikationen mit denen gearbeitet wurde:

- draw.Main: Hauptanwendung, die Grafiken in XML und JPG abspeichert.
- odg.Main: Programm, welches das ODG-Format verwendet.
- svg.Main: Programm, welches Vektorgrafiken verwendet und diese als SVG-Dateien abspeichert.

- **R13 - open existing drawing**

Um sicher zu gehen, dass alle notwendigen Klassen aufgezeichnet werden, wurden bei *R14* unterschiedliche und alle möglichen Figuren erstellt und diese als XML, JPG, ODG, und SVG-Dateien abgespeichert.

- **R14 - save drawing**

Die Beschreibung dieses Requirements lässt sich wieder weitläufig interpretieren. Es wird nicht genau definiert, ob es Einschränkungen bezüglich der Figurenarten gibt. Dieses Kriterium ist für das Aufzeichnen mittels TPTP sehr wichtig, da im Prinzip jede einzelne Unterart an Figuren von anderen Klassen verwaltet werden.

Nachdem die Priorität beim zweiten Durchlauf besonders darin lag, den orangen Bereich so klein wie möglich zu halten, spielt der rote Bereich eine untergeordnete Rolle und deshalb wurden im zweiten Durchlauf alle möglichen Figuren aufgezeichnet.

## 6.2.2 Unterschiede zum ersten Durchlauf

Mit diesen neu erstellten und optimierten Testcases lassen sich auf den ersten Blick bereits eindeutige Unterschiede erkennen. Der grüne Bereich ist bei allen Testcases gewachsen, was darauf schließen lässt, dass mehr Klassen mit TPTP aufgezeichnet wurden als beim ersten Durchlauf und somit präzisere Analysen erlauben.

Auf Abbildung 5.3 sieht man beispielsweise die Unterschiede der ersten drei Testcases. Auffallend ist hierbei die Tatsache, dass der orange Bereich noch immer relativ groß ist. Trotz der Berücksichtigung aller Klassen beim zweiten Durchlauf und allen möglichen bzw. unterschiedlichen Szenarien (SVG, ODG) konnten wieder nicht alle Klassen aufgezeichnet werden, die laut Requirements-Matrix zum jeweiligen Requirement dazugehört.

Auffallend oft werden dabei einzelne Klassen, wie zum Beispiel *FileIconsSample*, gelistet, die laut der Beschreibung der einzelnen Requirements im Prinzip nicht nötig sind, um die jeweilige gewünschte Operation durchzuführen, aber anscheinend dennoch benötigt werden.

Der rote Bereich ist nach wie vor der Dominierende und lässt sich allein durch optimiertes Tracen nicht wesentlich reduzieren. Allerdings gibt es die Option mittels Kombination mit anderen Testcases diesen Sektor zu reduzieren (siehe dazu Kapitel 5.2.4).

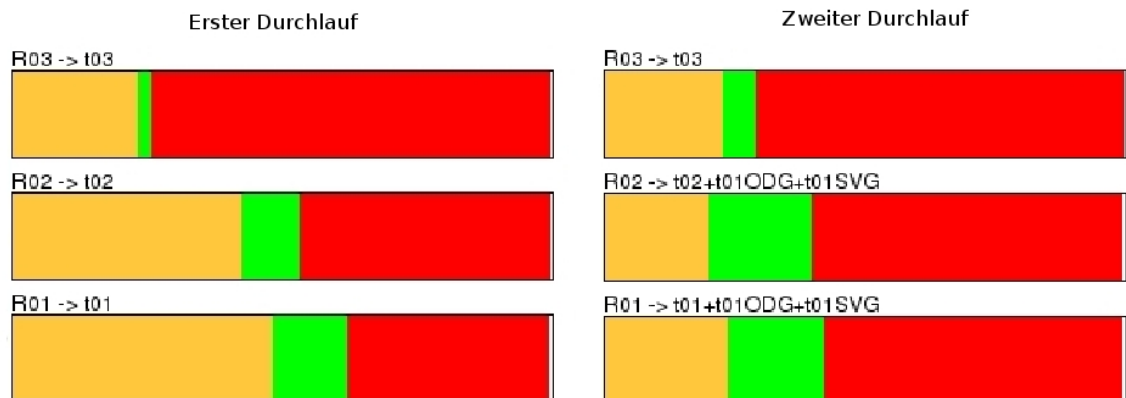


Abbildung 6.3: Unterschiede zwischen den beiden Testszenarien



## 6.3 Verfeinerung des Analyse-Tools

Nachdem der zweite Durchgang der Testcases ebenfalls einen relativ hohen, orangen Anteil (Klassen, die in der Requirements-Matrix gelistet wurden, aber von TPTP nicht aufgezeichnet wurden) besitzt, muss man versuchen mittels Kombinationen mit anderen Testcases diese so gut wie möglich zu reduzieren. Hierbei muss berücksichtigt, dass man durch diese Kombinationen den grünen Sektor allerdings auch vermindern kann.

Deshalb sind zusätzliche Hilfsmittel nötig, um sich einen besseren Überblick über die Beziehungen zwischen den einzelnen Testcases zu verschaffen.

### 6.3.1 Neue View: Beziehung zw. Requirement und Testcase

In dieser neuen Ansicht werden sämtliche vorkommenden Klassen aufgelistet, inklusive ihrer Beziehungen zu den einzelnen Requirements und Testcases: In der mittleren Spalte werden die Requirements aufgelistet, in welchen die einzelnen Klassen vorkommen und in der rechten Spalte die dazugehörigen Testcases die diese Klassen aufgezeichnet haben. Somit ist auf einen Blick ersichtlich, welche Klassen mit größerer Sorgfalt behandelt werden müssen, da sie in den Testszenarien nicht vorkommen.

Classes	Requirements	Testcases
AbstractAttributedCompositeFigure	R01 , R02	t01, t02, t03
AbstractAttributedDecoratedFigure	R01 , R02	t02
AbstractAttributedFigure	R01 , R02	t01, t02, t03
AbstractBean		t01, t02
AbstractCompositeFigure	R01 , R02 , R03	t01, t02, t03
AbstractConnectionHandle		t02
AbstractConnector	R02	t01
AbstractDrawing		t01
AbstractFigure	R02	t01, t02
AbstractHandle		t01, t02, t03
AbstractLineDecoration	R01	t02
AbstractLocator		t02
AbstractRotateHandle		t02
AbstractSelectedAction		t01, t02, t03
AbstractTool		t01, t02, t03
AlignAction		t01, t02, t03
ApplyAttributesAction		t01
ArrowTip	R01	
AttributeAction		t01, t02, t03
AttributeKey		t01, t02, t03

Abbildung 6.4: Beziehung zw. Requirement und Testcase

### 6.3.2 Neue View: Prozentuelle Überlappung

Nachdem sich manche Requirements in der Struktur sehr ähnlich sind, ist die Wahrscheinlichkeit nicht gering, dass man mithilfe von Kombinationen aus diesen verschiedenen Fällen eine bessere Trefferquote erreicht und somit den orangen Bereich verkleinert und den grünen Sektor vergrößert.

Aus diesem Grund ist es hilfreich sich eine Übersicht über die prozentuelle Überlappung der einzelnen Testcases und der Requirements zu erschaffen.

Hierbei werden in verschachtelten Schleifen alle geladenen Testcases mit den Requirements verglichen (bei jedem Schritt werden jeweils zwei Mengen, die getrackten Klassen und die Klassen der Requirements-Matrix bezüglich eines bestimmten Requirements, miteinander verglichen und daraus die prozentuelle Überlappung jedes einzelnen Szenarios ausgerechnet.

Daraus lassen sich bestimmte Muster herauslesen, mit denen man weiterarbeiten kann. Beispielsweise wird aus den zuvor erstellten Testcases ersichtlich, dass t01ODG (Figuren im ODG-Format zeichnen) mit 42,86 Prozent eine höhere Überlappung für R08 (editiere Texte in Figuren) hat, als die tatsächlich dafür erstellten Testcases.

Vergleicht man diese Werte mit anderen Cases lassen sich schnell optimale Kombinationen herauslesen, die in Kapitel 5.4 näher beschrieben werden.


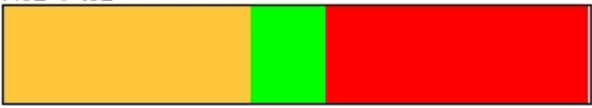



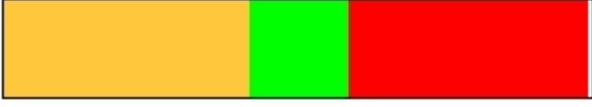
	R01	R02	R03	R04	R05
main	2,70	2,67	5,88	0,00	25,00
mainALL	31,08	36,00	35,29	42,86	37,50
odg	8,11	9,33	5,88	0,00	6,25
odgALL	24,32	21,33	23,53	28,57	25,00
svg	14,86	18,67	11,76	14,29	12,50
svgALL	0,00	0,00	0,00	0,00	0,00
t01	25,68	28,00	17,65	42,86	25,00
t01ODG	16,22	18,67	11,76	14,29	25,00
t01SVG	27,03	30,67	11,76	14,29	25,00
t02	27,03	24,00	17,65	28,57	25,00
t03	21,62	16,00	23,53	0,00	25,00
t04	14,86	12,00	5,88	0,00	18,75
t05	9,46	5,33	5,88	0,00	18,75
t06	17,57	12,00	11,76	28,57	25,00

Abbildung 6.5: Ausschnitt: Prozentuelle Überlappung

## 6.4 Verfeinerung der Ergebnisse mittels Kombinationen

Unter Verwendung der neuen View, welche die prozentuelle Überlappung anzeigt, wurden nun passend zu den Requirements unterschiedliche TestCase-Kombinationen erstellt, mit dem Ziel den grünen Bereich zu maximieren, indem der orange Sektor minimiert wird.

In allen Szenarien kam es entweder zu gleichen oder besseren Ergebnissen, allerdings führte die Neukombination der Testcases in keinem Fall zu einer Verschlechterung (siehe Tabelle).

Visuell	Name
<div style="border: 1px solid black; padding: 2px;"> <p>R02 -&gt; t02+t01SVG+t01</p>  </div> <div style="border: 1px solid black; padding: 2px;"> <p>R02 -&gt; t02</p>  </div>	<p><b>R02 - create figures and connections:</b></p>
<div style="border: 1px solid black; padding: 2px;"> <p>R04 -&gt; t04+t01+odgALL+t02+t06</p>  </div> <div style="border: 1px solid black; padding: 2px;"> <p>R04 -&gt; t04</p>  </div>	<p><b>R04 - delete connections:</b></p>
<div style="border: 1px solid black; padding: 2px;"> <p>R07 -&gt; t07+t07ODG+t07SVG+t01SVG+t12ODG</p>  </div> <div style="border: 1px solid black; padding: 2px;"> <p>R07 -&gt; t07+t07ODG+t07SVG</p>  </div>	<p><b>R07 - change properties of figures:</b></p>

In der Tabelle ist ersichtlich, dass bei der Optimierung zwar der rote Bereich angewachsen ist, jedoch auch gleichzeitig mehrere Klassen im grünen Bereich aufgezeichnet wurden.

## 6.5 Automatisierung der Optimierung

Die neu erstellte Ansicht *prozentuelle Überlappung* (siehe Kap. 5.3.2) bietet eine Gesamtübersicht darüber, wie gut Testcases zu den jeweiligen Requirements passen. Mit diesem Wissen kann man ein Verfahren zur automatischen Optimierung entwickeln, welches folgendermaßen aufgebaut ist:

- Einen sogenannten Threshold definieren.
- Alle Klassen der verschiedenen Testcases, die zu einem bestimmten Requirement eine höhere Überlappung haben als den definierten Threshold, werden zu einer großen Gesamtmenge verschmolzen.
- Aus dieser großen Menge werden alle Testcases, bzw. die dazugehörigen Klassen, die eine kleinere oder gleich große prozentuelle Überlappung haben als den Threshold, subtrahiert.

**Beispiel** - Optimierung der Testcases für R01 mit Threshold 0:

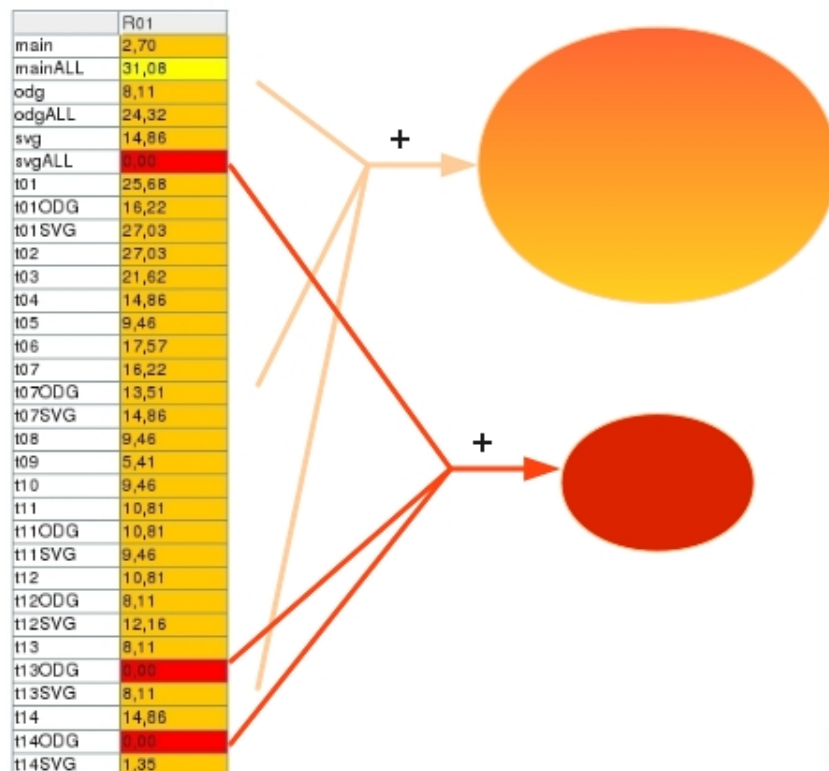


Abbildung 6.6: Zweiter Schritt

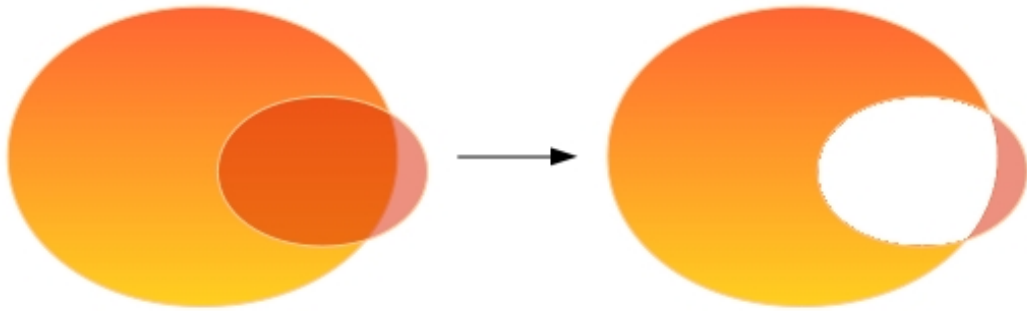


Abbildung 6.7: Dritter Schritt

Man erhält mit diesem Verfahren ein neues Testcase, das aus Additionen und Subtraktionen von allen vorhandenen Testcases entstanden ist.

Diese automatische Optimierung wird für jedes einzelne Requirement wiederholt, sodass am Ende der Berechnungen gänzliche neue und angepasste Testcases zur Verfügung stehen.

Je nachdem wie hoch man den Threshold einstellt, kann man das Ergebnis stark beeinflussen. Ein höherer Wert bewirkt, dass mehr Testcase-Mengen von der Gesamtmenge subtrahiert werden und umgekehrt bewirkt ein kleinerer Wert, dass mehr Testcase-Mengen miteinander vereint und somit weniger weggezogen werden.

In Abbildung 5.7 ist ersichtlich, dass nicht immer der gesamte zu subtrahierende Bereich auch tatsächlich abgezogen werden kann von der ersten Menge. Der Grund dafür: Nachdem es sich um zwei Klassenmengen handelt mit meist unterschiedlichen Belegungen, ist nicht immer garantiert dass eine Klasse in beiden Mengen vorhanden ist. Somit kommt es auch bei einigen Szenarien zu diesen Gaps.

## 6.6 Gegenüberstellung aller erstellten Testsznarien

Im zweiten Durchlauf wurden drei unterschiedliche Szenarien erstellt:

- **Szenario 1**

Ausgehend von der Information, die die Tracematrix alleine anhand der Namen der Requirements und der Klassenzusammengehörigkeit bietet, wurden zu jedem Requirement einzelne Testcases erstellt. (siehe Kapitel 5.2)

- **Szenario 2**

Aus den Testcases aus Szenario 1 wurden unter Verwendung der in Kapitel 5.3.2 entwickelten Ansicht (prozentuelle Überlappung) neue kombinierte Testcases erstellt. Hierbei wurden immer die vier am besten geeigneten Testcases herangezogen - diese vier Cases hatten jeweils die höchsten prozentuelle Überlappung. (siehe Kapitel 5.4)

- **Szenario 3**

Ein automatisierendes Verfahren wurde entwickelt, welches ebenfalls wie in Szenario 2 mit dem Wissen über die Gesamtüberlappung arbeitet. (siehe Kapitel 5.5)

Von Schritt zu Schritt bzw. von einem Szenario zum anderen, ist es zu vermehrten Verbesserungen gekommen. Der wichtigsten Sektor (der grüne Bereich) ist gewachsen, was daraus schließen lässt, dass die neuen Testcases vermehrt mit dem Requirement übereinstimmen.

Allerdings konnte selbst im letzten, optimierten Szenario der orange Sektor nicht auf null reduziert werden. Das liegt vor allem an der Tatsache, dass gewisse Klassen, die von der Requirementsmatrix vorgegeben wurden, von TPTP zu keiner Zeit aufgezeichnet wurden.

Der rote Bereich ist, wie erwartet, im zweiten Szenario um Einiges gewachsen, weil mehrere Testcases miteinander kombiniert wurden und es somit zu einer Aufblähung der Gesamtmenge kam. Dieser Sektor konnte jedoch im dritten Szenario dank der Automatisierung in den meisten Fällen reduziert werden.

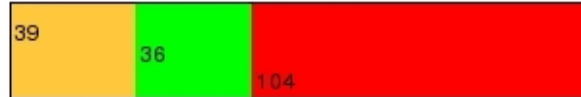
Zur besseren Veranschaulichung, folgt eine nähere Analyse am Beispiel von vier markanten Requirements, wobei Spalte 1 für Szenario 1 steht usw., und desweiteren wurde die Anzahl der Klassen hinzugefügt:

## R02 - create figures and connections

R02 -&gt; t02. 135 classes



R02 -&gt; t02+t01SVG+t01. 179 classes



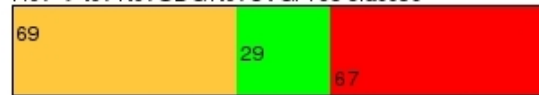
R02 -&gt; t2 (autom.). 262 classes



Ein markanter Zuwachs ist im grünen Sektor zu erkennen, vor allem im automatisierten Verfahren konnte dieser Bereich vergrößert werden. Nachteil dieser Methode ist aber die Einstellung eines Thresholds von 0, sodass nur wenige Testcases untereinander abgezogen wurden und somit der rote Bereich im Vergleich zu Szenario 1 stark angewachsen ist.

## R07 - change properties of figures

R07 -&gt; t07+t07ODG+t07SVG. 165 classes



R07 -&gt; t07+t07ODG+t07SVG+t01SVG+t12ODG. 218 classes



R07 -&gt; t7 (autom.). 281 classes



Zwischen Szenario 1 und 2 gibt es nur einen geringen Zuwachs an grünen Klassen, obwohl es im zweiten Fall zu einer Kombination aller vorhandenen Testcases kam.

## R09 - align figures

R09 -&gt; t09. 22 classes



R09 -&gt; t09. 22 classes



R09 -&gt; t9 (autom.). 70 classes



Die absolute Aufteilung ist bei allen Szenarien konstant gleich geblieben. Im letzten Fall ist der rote Sektor sogar noch dominierender. Daraus lässt sich schließen, dass in keinem vorkommenden Testcase die für R09 benötigten Klassen von TPTP aufgezeichnet wurden und somit entweder die Testcases zu ungenau angesetzt wurden, oder die Belegung in der Requirementsmatrix nicht aktuell oder falsch ist.

## R12 - launch application

R12 -&gt; t12+t12ODG+t12SVG. 129 classes



R12 -&gt; t12+t12ODG+t12SVG. 129 classes



R12 -&gt; t12 (autom.). 40 classes



Bei R12 handelt es sich um einen ähnlichen Fall wie bei R09, allerdings mit dem Unterschied, dass beim automatisierten Verfahren der rote Sektor um ein Vielfaches gesenkt werden konnte. Das liegt vor allem daran, dass die prozentuelle Überlappung bei R09 mit allen Testcases in vielen Fällen bei 0 lag, und somit viele dieser Klassen von der Gesamtmenge abgezogen wurden.

Man kann diese Ergebnisse weiter verfeinern, indem man zu Vergleichszwecken sämtliche Klassen aus der Analyse entfernt, die nur im orangenen Sektor vorkommen. Das sind nämlich jene Klassen, die wegen Messungsungenauigkeiten bei den Traces nicht vorkommen und somit zur Bestimmung weiterer Muster nicht relevant sind. In Kapitel 6.7. werden deshalb die drei erzeugten Szenarien von Kapitel 6.6 nochmals unterteilt in zwei Untergruppen (mit reinen, orangenen Klassen und ohne), um nach weiteren Mustern zu suchen.



## 6.7 Recall, Precision und weitere Merkmale

Die Trefferquote (Recall) und Genauigkeit (Precision) sind Werkzeuge zur Beurteilung der Güte einer Recherche. Diese Werte können dabei Werte zwischen 0 und 100 einnehmen und lassen sich folgendermaßen ausrechnen:

$$\text{Recall} = \text{truePositives} \frac{\text{truePositives}}{\text{truePositives} \cup \text{falseNegatives}}$$

$$\text{Precision} = \text{truePositives} \frac{\text{truePositives}}{\text{truePositives} \cup \text{falsePositives}}$$

Die Trefferquote ist hierbei ein Maß des Verhältnisses zwischen allen richtig positiven und allen relevanten Mengen, wobei die Genauigkeit das Verhältnis der richtig positiven und allen gefundenen Mengen widerspiegelt.

Wenn beispielsweise die Recherche alle Mengen zurückliefert, ist die Trefferquote maximal. Die Genauigkeit hängt allerdings von der Gesamtheit aller relevanter und nicht-relevanter Mengen ab.

In Tabelle 6.7.1 sind die Berechnungen des Recalls sowie Precision für die Szenarien 1 und 3 ersichtlich, wobei es zwei Unterscheidungen gibt: Das jeweilige, komplette Szenario und eines mit gelöschten orangen Sektoren (= WO, without orange):

	scen1	scen1 WO	scen3	scen3 WO
recall	25	61	44	99
precision	15	15	11	11

Tab. 6.7.1: Recall und Precision

Durch das Entfernen der orangen Sektoren lässt sich eine Steigerung der Trefferquote beobachten, wobei die Genauigkeit konstant gleich bleibt. Dies lässt sich dadurch erklären, dass nur die Menge der falseNegatives modifiziert wurde und somit kein direkter Einfluss auf die weitere Genauigkeit besteht.

Im dritten Szenario (WO) kommt es sogar zu einer 99-prozentigen Trefferquote, was bedeutet, dass 99 Prozent aller geforderten Klassen bzw. Mengen gefunden wurden.

Die Genauigkeit verweilt allerdings weiterhin bei 11 Prozent, da der Bereich der getrackten Klassen auch weiterhin einen großen Sektor abdeckt, welcher nicht in der Requirementsmatrix vorkommt, und somit der Grad der Genauigkeit sinkt, bzw. konstant gleich bleibt.

	scen1	scen1 WO	scen3	scen3 WO
recall	84	84	88	88
precision	64	89	86	99

Tab.6.7.2 Recall und Precision

Bei der Berechnungen in Tab. 6.7.1 wurden für die truePositives grüne, falsePositives rote und für die falseNegatives orange Klassen herangezogen, um die Trefferquote und Genauigkeit zu ermitteln. Damit wurde jener Fall abgedeckt, der sich mit der korrekten Güte der Rechercheergebnisse auseinandersetzt.

Es lassen sich allerdings auch konträre Verhältnisse in diese Berechnung miteinbeziehen, indem man beispielsweise die Mengen untereinander vertauscht, sodass die Basismenge der Berechnung nicht die Menge aller grünen Klassen ist, sondern jene mit den roten Klassen. Dadurch lässt sich eine Vergleichsbasis erstellen, inwieweit sich der falsche und negative Sektor auf die Trefferquote und Genauigkeit auswirkt.

Wie in Tab. 1 ist auch in Tab. 2 ein Muster zwischen Trefferquote und Genauigkeit zu beobachten, allerdings in umgekehrter Form: Die Trefferquote bleibt innerhalb der Szenarien konstant gleich, die Genauigkeit steigt allerdings, nachdem zur Berechnung nun auch der orange Sektor miteinbezogen wird, und dieser sich durch die Modifikation bzw. Weglassen ändert.

Aus diesen zwei Berechnungen wird ersichtlich, dass die Trefferquote einer Klassenmenge erhöht wird, indem der falseNegative-Sektor reduziert wird, allerdings es zu keinen Verbesserungen der Genauigkeit kommt.

### Prozentuelle Aufteilung

Ein weiterer interessanter Aspekt ist die prozentuelle Aufteilung der unterschiedlichen Bereiche; truePositives, trueNegatives, falsePositives und falseNegatives. Speziell nach dem Optimierungsschritt aus Seite 44 lassen sich gleichbleibende Verschiebungen beobachten.

	false	true
pos	10	1
neg	5	81

Tab.6.7.3 Scenario 1

	false	true
pos	19	3
neg	2	74

Tab.6.7.4 Scenario 1 WO

Die falsePositives und truePositives, also der rote und grüne Bereich, sind nach der Optimierung angestiegen. Im Gegensatz dazu haben sich die falseNegatives und trueNegatives reduziert.

Das selbe Verhalten lässt sich auch für das dritte Szenario beobachten:

	false	true
pos	27	3
neg	4	64

Tab.6.7.5 Szenario 3

	false	true
pos	48	6
neg	0.002	45

Tab.6.7.6 Szenario 3 WO

Am stärksten fallen die Veränderungen bei falsePositives und trueNegatives aus. Dieser Zusammenhang lässt sich dadurch erklären, weil die falseNegatives auf ein Minimum reduziert wurden, und zwischen den orangen und roten bzw. weißen Klassen eine engere Beziehung besteht. Daraus resultiert, dass bei der Eliminierung von den falseNegatives die Werte der truePositives nur minimal beeinflusst werden.

Es besteht somit eine klare Beziehung zwischen den falseNegatives und trueNegatives bzw. falseNegatives und falsePositives.

## Kapitel 7

# Zusammenfassung

### 7.1 Zusammenfassung

Im Rahmen dieser Arbeit wurde ein Analyse-Tool entwickelt, das Requirements und TestCases gegenüberstellt. Die wesentliche Aufgabenstellung war dabei, dass die Möglichkeit gegeben wird diese Requirements und TestCases untereinander verschieden kombinieren zu können. Desweiteren wurde ein weiterer Fokus auf die Darstellung der Ergebnisse gelegt, welche im Tool anhand von unterschiedlichen Views erfolgt.

Anhand des JHotDraw-Frameworkes wurde das Tool getestet und versucht markante Muster in der Zusammengehörigkeit der Klassen zu identifizieren, was auch entsprechend gelungen ist. Es wurde festgestellt, dass in gewissen Sektoren Klassen vermehrt vorkommen, als in anderen und es in keinem der TestCases zu völliger Sättigung der Bedingungen kam. Durch Optimierungen mittels verschiedener Verfahren, konnten die Ergebnisse verbessert werden. Unter anderem wurde der Grad der Trefferquote erhöht, allerdings blieb die Genauigkeit der überlappenden Mengen gleich groß.