Bachelorarbeit

Visualisierung von Testdaten in Software-Städten

Von Katharina Legde

Matrikelnummer: 2712205

Emailadresse: legdekat@tu-cottbus.de

Studiengang: Informations- und Medientechnik

Semester: WS 2010/2011

Lehrstuhl: Software-Systemtechnik, Brandenburgische Technische Universität Cottbus

Betreuer: Prof. Dr. Claus Lewerentz, Frank Steinbrückner

Abgabedatum: 15.02.2011

Inhaltsverzeichnis

Abbildungsverzeichnis

1 Einleitung

Abstrakte Daten korrekt und schnell zu interpretieren, stellt sich für den Menschen oft als sehr schwierig heraus. In diesem Zusammenhang hört man oft Sätze wie zum Beispiel: *„Darunter kann ich mir nichts vorstellen!“* oder *„Das versteh ich nicht!“*. In diesen beiden Ausrufen kommt den Wörtern „verstehen“ und „vorstellen“ eine große Bedeutung zu. Um etwas verstehen zu können, muss man es sich vorstellen können. Hat man eine Vorstellung, so besitzt man ein mentales Bild. Man hat es sich visualisiert.

„Visualisieren bezeichnet die Tätigkeit, einen bislang im Zeichensystem der Wortsprache ausgedrückten Inhalt entweder durch bildsprachliche Zeichen zu ergänzen, oder aber ihn ganz in die Bildsprache zu übersetzen.“ [1]

Die Visualisierung wird in Bereichen der Wirtschaft, Medizin oder der Wissenschaft oft genutzt, um komplexe Inhalte zu veranschaulichen, optisch ansprechend aufzubereiten, das Verständnis zu verbessern und vorher unbekannte Zusammenhänge herauszustellen. Ein hervorzuhebendes Gebiet der Visualisierung ist die Informationsvisualisierung, anzufinden in Wettervorhersagen, Diagrammen von Arbeitszeit-oder Bevölkerungsverteilungen. Sie ist dafür verantwortlich komplexe Inhalte für Laien zugänglich zu machen und ihnen so die Möglichkeit zu geben neue Erkenntnisse zu gewinnen. Bezieht man die Informationsvisualisierung nun auf die Bereich Informatik, findet man dort auch zahlreiche Einsatzgebiete. Ein Beispiel ist die Visualisierung von Software-Eigenschaften, auch genannt Metriken. Zu solchen Software-Metriken gehören die Komplexität eines Projekts, die Einhaltung von Standards, wie bspw. die Java Code Conventions, der Ressourcenaufwand, die Entwicklungszeit und der Schulungsaufwand. In dieser Bachelorarbeit wird die Metrik der Software Tests näher betrachtet. Dazu gehören Aussagen über den Abdeckungsgrad und den Testerfolg eines Projekts. Der Abdeckungsgrad zeigt dabei auf wie viel Prozent des Quelltextes durchs Tests ausgeführt werden. Der Testerfolg wiederum zeigt an wie viele dieser Tests korrekt ausgeführt werden können. Diese beiden Metriken sind wichtig für eine Bewertung der Software-Qualität. Die Informationen über Software-Metriken sind oft abstrakt und können schnell sehr umfangreich werden. Daher ist es wünschenswert ein Werkzeug zu benutzen, welche verschiedenste Software-Metriken, optisch ansprechend visualisiert und somit die Interpretationszeit verringert. Crococosmo verfügt über diese Fähigkeit. Es visualisiert verschiedenste Eigenschaften von Software anhand einer Stadtmetapher und ermöglicht es eine Auskunft über Struktur und Entwicklungsgrad von Projekten zu erlangen, vorher unbekannte Zusammenhänge zu entdecken, Sachverhalte besser verstehen zu können und die Qualität der Software zu verbessern. Im Rahmen dieser

Bachelorarbeit sollte Crococosmo nun um die Fähigkeit der Visualisierung der Testauswertungen erweitert werden. Um auf diese Weise eine schnelle, intuitive und revisionsübergreifende Auswertung eben dieser Daten vorzunehmen, Testfehler zu entdecken und beheben zu können und effizienter Personal und Ressourcen in dieser zeitaufwändigen Aufgabe der Software-Entwicklung einteilen zu können.

1.1 Aufbau der Arbeit

Die Bachelorarbeit ist im Wesentlichen in einen Theorie- und in einen Praxisteil gegliedert. Der Theorieteil befasst sich mit der Thematik Testen. Darin werden Ziele und Bedeutung des Vorgangs Testen herausgearbeitet, auf die unterschiedlichsten Arten des Testens eingegangen und die Automatisierung dieser zeitaufwendigen Tätigkeit angesprochen. Weiterer Bestandteil des Theorieteils wird die Thematik der Visualisierung sein. In diesem Kapitel soll auf die Bedeutung und Ziele der Informationsvisualisierung und besonders der Visualisierung der Testdaten eingegangen werden und die einzelnen Schritte zu einer angemessenen Visualisierung dieser Daten erkenntlich werden. Weiterhin werden bereits genutzte Visualisierungskonzepte in diesem Bereich erläutert und bewertet.
Der Praxisteil dieser Bachelorarbeit erläutert den Weg von rohen Komponententests zu den Testauswertungsdaten bis hin zur ihrer Visualisierung. Auf diesem Weg genutzte Werkzeuge und Implementierungen werden beschrieben, das entwickelte Visualisierungskonzept erklärt und evaluiert.

2 Stand der Technik

Da Software und auch Software-Tests zunehmend umfangreicher werden, wird es für den Entwickler oft zu einer mühsamen Arbeit die Tests auszuführen und den Testausgang zu protokollieren. Dies ist der Hauptgrund für die Verwendung von Testtools, die diese Aufgaben schnell und zuversichtlich übernehmen. Dabei gibt es ein umfangreiches Angebot von OpenSource-Software aber auch Programme kommerzieller Art. Im Folgenden sollen einige Programme, die Auskünfte über die Testauswertungen von Java-Projekten geben, genannt und ihre Vorgehensweise näher erläuter werden. Jedes, der im Folgenden genannten Software-Projekte, arbeitet nach dem dynamischen Test-Verfahren. Das bedeutet, dass zu testende Programm muss zur Auswertung der Testdaten ausgeführt werden.

2.1 Clover

Clover ist ein kommerzielles Testwerkzeug, welches Unterstützung für Ant, Maven, Eclipse und die Kommandozeile anbietet. Es liefert umfangreiche Informationen über die

Testabdeckung des gesamten Projekts oder wahlweise einzelner Klassen. Weiterhin liefert es eine Auswertung des Testerfolges. Clover nimmt die Visualisierung dieser Daten anhand von Statusanzeigen, Diagrammen und Treemaps[1] vor. Wie in Abbildung 1 zu erkennen, werden die Testabdeckung und der Testerfolg für das ganze Projekt übersichtlich anhand von Statusanzeigen dargestellt. Die Treemap stellt bei Clover das wichtigste Visualisierungskonzept dar. Sie ist interaktiv und gibt dem Benutzer die Möglichkeit in Interessenbereiche zu zoomen. Er ist in der Lage von der Paketansicht über die Klassenansicht bis hin in den Quelltext der Klasse zu navigieren, wo wiederum eine Auswertung der Testabdeckung anhand einer Editor-Ansicht vorgenommen wird. Durch die Färbung entsprechender Zeilen werden Informationen über die Ausführung dargelegt. Im Zusammenhang bedeutet eine Rotfärbung, dass Zeilen nicht durch Tests ausgeführt werden. Eine Grünfärbung besagt im Gegensatz dazu, dass diese Zeile getestet wurde. Die Information über die Testabdeckung wird anhand einer Instrumentierung[2] gesammelt. Diese Instrumentierung erfolgt hierbei anhand eines Quelltext-Duplikates. Clover ist in der Lage, die gesammelten Informationen in HTML-Dateien abzuspeichern.

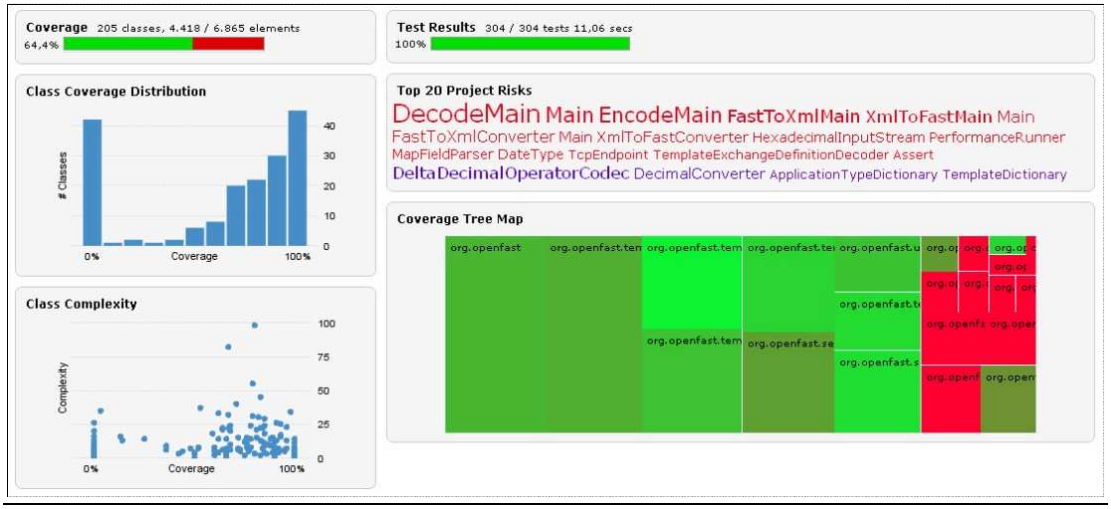

Abbildung 1: Erstellte Testauswertung (HTML-Seite) durch Clover

2.2 Cobertura

Cobertura ist ein OpenSource-Programm, welches Unterstützung für Ant, Maven und die Kommandozeile liefert. Dieses Testtool liefert umfangreiche Informationen zur Testabdeckung des gesamten Projektes, einzelner Klassen, Methoden und sogar einzelner Bedingungen. Mit Hilfe von Cobertura können neben der Zeilenabdeckung auch Informationen über die Zweigabdeckung gesammelt werden. Es visualisiert ausschließlich

[1] Visualisierung hierarchischer Strukturen anhand von verschalteten Rechtecken
[2] Anreicherung des Quelltextes mit Zusatzinformationen. In diesem Fall Anreicherung jeder Zeile des Quelltextes mit einem Zähler vgl. Kapitel Testautomatisierung

anhand einer Tabelle in der das Paket oder die Klasse mit Informationen über Zeilenabdeckung, Zweigabdeckung und Angaben über die zyklomatische Komplexität nach McCabe vermerkt ist. Diese Informationen werden anhand von Statusanzeigen mit einer Prozentzahl visualisiert. Cobertura ist in der Lage die Abdeckung einzelner Klassen, Pakete oder des gesamten Projekts anzuzeigen. Dabei ist es dem Anwender möglich bis in den Quelltext zu navigieren, um dort eine Auswertung der Testabdeckung direkt an den Quellcode vorzunehmen. Wird die Zeile rotgefärbt, bedeutet es, dass sie nicht durch Tests durchlaufen wurde. Ist eine Zeile wiederum grün gefärbt, bedeutet dies, dass sie durch Tests ausgeführt und dementsprechend auch abgedeckt ist. Cobertura berechnet die Testabdeckung anhand von einer Bytecodeinstrumentierung[3]. Die erlangten Informationen können letztendlich in XML-Dateien oder in HTML-Dateien langzeitig gespeichert werden.

Coverage Report - All Packages

Package	# Classes	Line Coverage	Branch Coverage	Complexity
All Packages	278	54% 3169/5868	54% 1246/2268	2,021
org.openfast	41	71% 628/884	64% 251/390	1,769
org.openfast.codec	3	71% 28/39	50% 5/10	1,9
org.openfast.debug	6	0% 0/83	0% 0/12	1,222
org.openfast.error	10	62% 41/66	7% 1/14	1,379
org.openfast.examples	9	0% 0/72	0% 0/12	2,105
org.openfast.examples.consumer	3	0% 0/80	0% 0/22	4,4
org.openfast.examples.decoder	2	0% 0/67	0% 0/14	3
org.openfast.examples.interpret	2	0% 0/119	0% 0/32	3,286
org.openfast.examples.performance	4	0% 0/185	0% 0/66	2,769
org.openfast.examples.producer	5	0% 0/239	0% 0/76	2,759
org.openfast.examples.scp10	2	0% 0/81	0% 0/14	2,714
org.openfast.examples.tmplexch	4	0% 0/156	0% 0/34	5
org.openfast.examples.util	2	0% 0/49	0% 0/14	2,125
org.openfast.examples.xml	4	0% 0/182	0% 0/46	3,636
org.openfast.extensions	2	0% 0/38	0% 0/12	2
org.openfast.impl	3	58% 18/31	0% 0/2	1,5
org.openfast.logging	5	12% 7/58	0% 0/18	2,091
org.openfast.session	39	40% 215/533	15% 15/96	1,484
org.openfast.session.multicast	6	23% 22/92	10% 1/10	1,75
org.openfast.session.tcp	2	0% 0/47	0% 0/8	2,364
org.openfast.session.template.exchange	10	92% 286/310	69% 79/114	2,455
org.openfast.template	17	70% 480/684	69% 234/336	1,772
org.openfast.template.loader	17	93% 366/392	92% 152/164	2,253
org.openfast.template.operator	18	84% 262/310	73% 186/254	2,83
org.openfast.template.serializer	15	87% 218/249	59% 62/104	2,156
org.openfast.template.type	13	63% 103/161	46% 29/62	2,1
org.openfast.template.type.codec	24	75% 327/431	64% 161/248	2,579

Abbildung 2: Testauswertung (HTML-Datei) von Cobertura

2.3 Emma

Emma ist ein OpenSource-Tool, welches eine Unterstützung für Ant, Maven und die Kommandozeile liefert. Diese Software gibt Aufschluss über die Testabdeckung des gesamten Projekts und einzelner Pakete und Klassen. Diese wird anhand einer Tabelle visualisiert. Der Anwender ist in der Lage diese Tabelle interaktiv wahrzunehmen und sich durch das Projekt zu navigieren. In der Editor-Ansicht kann eine Auswertung der Testabdeckung vorgenommen werden. Liegt eine Rotfärbung einer Zeile vor, ist dies ein Zeichen dafür, dass sie nicht durch Tests ausgeführt wird. Erscheint eine Zeile wiederum grün, betitelt das das Gegenteil. Emma ist in der Lage Auskunft über die Klassen-, Methoden-, Block-, und (partielle)

[3] Zeilenweise Anreicherung des Quelltextes mit Zähler während der Kompilierung vgl. Kapitel 3.5 Testautomatisierung

Zeilenabdeckung zu geben. Diese Informationen werden mittels einer Offline-Klasseninstrumentierung[4] oder wahlweise einer „On-the-fly“-Instrumentierung[5] erbracht. Emma ist in der Lage, die gesammelten Informationen in einer XML oder einer HTML-Datei auszuliefern.

EMMA Coverage Report (generated Sat Dec 18 13:22:15 CET 2010)

[all classes]

OVERALL COVERAGE SUMMARY

name	class, %	method, %	block, %	line, %
all classes	73% (175/241)	57% (892/1567)	56% (15315/27507)	53% (3098.4/5837)

OVERALL STATS SUMMARY

total packages:	28
total executable files:	202
total classes:	241
total methods:	1567
total executable lines:	5837

COVERAGE BREAKDOWN BY PACKAGE

name	class, %	method, %	block, %	line, %
org.openfast.debug	0% (0/4)	0% (0/24)	0% (0/373)	0% (0/83)
org.openfast.examples	0% (0/9)	0% (0/29)	0% (0/320)	0% (0/67)
org.openfast.examples.consumer	0% (0/3)	0% (0/8)	0% (0/391)	0% (0/79)
org.openfast.examples.decoder	0% (0/2)	0% (0/7)	0% (0/322)	0% (0/67)
org.openfast.examples.interpret	0% (0/2)	0% (0/9)	0% (0/625)	0% (0/119)
org.openfast.examples.performance	0% (0/4)	0% (0/29)	0% (0/762)	0% (0/185)
org.openfast.examples.producer	0% (0/5)	0% (0/33)	0% (0/1016)	0% (0/238)
org.openfast.examples.scp10	0% (0/2)	0% (0/9)	0% (0/379)	0% (0/81)
org.openfast.examples.tmplexch	0% (0/4)	0% (0/11)	0% (0/673)	0% (0/156)
org.openfast.examples.util	0% (0/1)	0% (0/6)	0% (0/167)	0% (0/49)
org.openfast.examples.xml	0% (0/4)	0% (0/13)	0% (0/854)	0% (0/182)
org.openfast.extensions	0% (0/2)	0% (0/11)	0% (0/163)	0% (0/38)
org.openfast.session.tcp	0% (0/2)	0% (0/11)	0% (0/141)	0% (0/47)
org.openfast.logging	50% (2/4)	27% (4/15)	15% (38/261)	12% (6.7/58)
org.openfast.session.multicast	17% (1/6)	19% (5/27)	24% (83/347)	24% (22/92)
org.openfast.error	88% (7/8)	59% (19/32)	52% (136/262)	61% (40/66)
org.openfast.session	68% (19/28)	28% (43/153)	62% (2064/3326)	40% (207.5/524)
org.openfast	97% (32/33)	69% (233/336)	69% (2741/3948)	69% (604.2/880)

Abbildung 3: Testauswertung (HTML-Datei) von Emma

2.4 Pin

Pin ist ein von zwei Studenten am Hasso-Plattner-Institut für Software-Entwicklung programmiertes Tool zur Visualisierung der Testabdeckung von C/C++ Programmen. Pin nutzt dabei das Visualisierungskonzept einer 3D-Treemap. Die rechteckige Grundfläche der Treemap stellt das gesamte Projekt als flache Ebene dar. Eine Hierarchieebene darüber werden die im Projekt befindlichen Pakete anhand von Rechtecken visualisiert. Wiederum eine Hierarchieebene höher, werden die zum Paket zugehörigen Klassen dargestellt. Pin visualisiert also nicht nur die Testabdeckung, sondern gibt auch Aufschluss über die Struktur des Projekts. Diese in der Treemap beinhalteten verschachtelten Rechtecke sind mit einer Färbung versehen. Sie gibt Auskunft über den Grad der Testabdeckung dieser Klasse oder des Pakets. Je grüner die Fläche eingefärbt ist, desto höher ist der Abdeckungsgrad und dementsprechend die Anzahl, der durch Tests ausgeführten Zeilen. Besteht wiederum eine Rot- oder Orangefärbung der Rechtecke, kann auf eine geringe Testabdeckung geschlossen werden. Die Treemap ist eine interaktiv zugängliche Visualisierung, das bedeutet der Anwender ist in der Lage bis auf den Quelltext hinein zu zoomen. Die daraufhin angezeigt Editor-Ansicht zeigt eine Auswertung der Testabdeckung direkt am Quelltext. Dabei wird ein ähnliches Farbkonzept wie das bereits Erläuterte genutzt. Denn grün gekennzeichnete Zeilen sind durch Tests abgedeckt und wiederum rot gekennzeichnete Zeilen besagen, dass keine Tests diese Zeile ausführen. Pin visualisiert die Information über die Testabdeckung einer

[4] Instrumentierung erfolgt vor dem Laden der Klassen
[5] Instrumentierung erfolgt während des Ladens der Klassen mittels eines spezifischen Classloaders

Klasse zusätzlich anhand einer SeeSoft-Darstellung[6] neben jeder Quelltext-Ansicht. Dort erlangt man einen direkten Überblick über die geöffnete Datei. Der Nutzer ist in der Lage den Quelltext aus der Vogelperspektive zu sehen und auf diese Weise die Testabdeckung anhand von Färbungen auszuwerten. In der SeeSoft-Darstellung sind die durch Tests abgedeckten Zeilen grün und die nicht abgedeckten Zeilen rot gekennzeichnet. Diese Informationen werden über eine von den Studenten eigens entwickelte Instrumentierungsart erlangt.

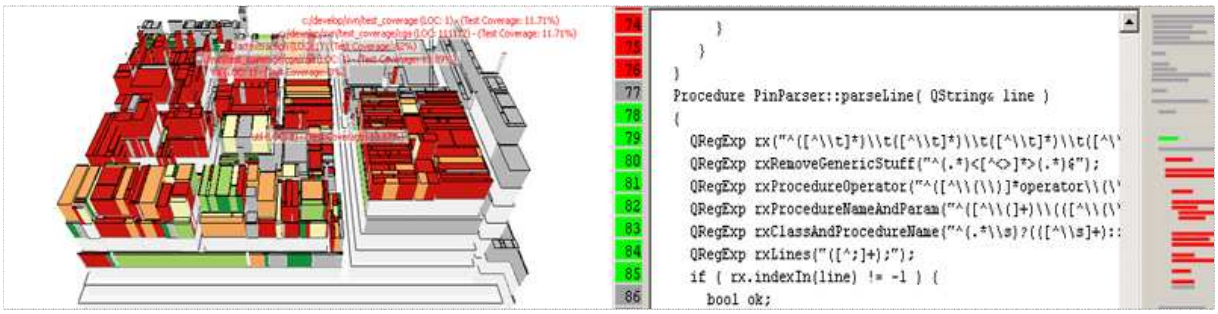

Abbildung 4: 3D-Treemapvisualisierung (links) Quelltext und SeeSoft-Darstellung (rechts) von Pin

2.5 Tarantula

Tarantula ist ein Standalone-Testwerkzeug[7], welches bisher nur für die Testauswertung von C-Programmen genutzt werden kann. Es macht sich das Prinzip der SeeSoft- Darstellung zu Nutze. Dabei wird der Quelltext aus der Vogelperspektive visualisiert. Jede Zeile wird anhand von aneinandergereihten Pixeln dargestellt. Diese können durch unterschiedliche Färbungen verschiedene Informationen ausdrücken. In diesem Zusammenhang werden bei Tarantula der Testerfolg und die Testabdeckung visualisiert. Die Testabdeckung ist allgemein an den farbigen Bereichen zu erkennen. Sind Pixel grau eingefärbt, werden sie nicht von Tests durchlaufen, sind sie wiederum rot, grün oder eine im Farbverlauf zwischen diesen beiden Farben befindliche eingefärbt, so werden sie von Tests durchlaufen. Die Färbung zeigt den Grad des Erfolgs der Tests an, die diese Zeile ausführen. Auf diese Weise werden Zeilen, die durch erfolgreich verlaufene Tests ausgeführt werden, grün eingefärbt. Eine Rotfärbung erfolgt im Gegensatz dazu, wenn Zeilen durch fehlgeschlagene Tests ausgeführt werden. Wird ein Bereich oder eine Zeile mit einer Farbe im Farbverlauf zwischen grün und rot eingefärbt, so ist schlussfolgernd festzustellen, dass diese Zeile durch sowohl fehlgeschlagene als auch erfolgreich verlaufene Tests ausgeführt wird. Tarantula gewinnt diese Informationen nicht etwa über eine Bytecode-Instrumentierung oder Quellcode-Instrumentierung, sondern über eine eigens entwickelte Art. Es überwacht dabei jeden Test und zeichnet das Testergebnis zusammen mit den ausgeführten Zeilen auf. Diese Vorgehensweise kommt der

[6] Visualisierung der Zeilen des Quelltextes anhand von aneinandergereihten Pixeln innerhalb einer Box, die die Klasse repräsentiert.
[7] Es arbeitet unabhängig von weiterer Software.

Instrumentierung zwar nah, aber kann nicht so benannt werden, da hier keine Zähler im Quellcode existieren, sondern die Zeilenzahlen aufgezeichnet werden. Eine Instrumentierung wäre in diesem Zusammenhang wenig effektiv, da letztendlich die Zeilennummer für die entsprechende Färbung entscheidend ist. Bei der Visualisierung erlaubt Tarantula verschiedene Einstellungen, die es ermöglichen, nur den Testerfolg oder die Testabdeckung ersichtlich zu machen. Zurzeit arbeitet das Entwicklerteam an einer weiteren Version von Tarantula, die es erlauben soll, Java-Testdaten auszuwerten. Tarantula bietet derzeit leider keinen Export der erlangten Informationen an.

Abbildung 5: Visualisierung von Testdaten mittels Tarantula

2.6 Zusammenfassung

Dieses Kapitel gibt einen kurzen Überblick über vorhandene Visualisierungsprogramme für Testdaten und auch deren Visualisierungskonzept. Dieser Überblick fasst nicht alle auf dem Markt befindlichen Programme ab, aber nennt, die zurzeit herausstechenden Konzepte der Testdaten-Visualisierung. Die folgende Tabelle ist eine Übersicht über die im Kapitel genannten Programme mit ihren wichtigsten Stammdaten in Bezug auf die Visualisierung der Testdaten.

Werkzeug	Testabdeckung	Testerfolg	Visualisierung	Instrumentierung	Dateien
Clover	Ja	Ja	-Statusanzeige -Treemap -Diagramme -Editoransicht	-Duplikatinstru-mentierung	HTML
Cobertura	Ja	Nein	-Tabellen mit Statusanzeigen	-Bytecodeinstru-mentierung	HTML,XML

			-Editoransicht		
Emma	Ja	Nein	-Tabelle -Editoransicht	-Offline Instru-metierung -On the fly Instru-metierung	HTML,XML
Pin	Ja	Nein	-3D-Treemap -Editoransicht -SeeSoft	-Binärdateienin-strumentierung	-
Tarantula	Ja	Ja	-SeeSoft	-	-

Tabelle 1: Übersicht über bestehende Software mit einigen wichtigen Stammdaten

3 Testen von Software

Heutzutage sind Rechnersysteme nicht mehr aus dem Alltag wegzudenken. Sie begegnen uns in der Luft- und Raumfahrt, im Krankenhaus, an der Börse usw. Da Rechnersystemen eine immer größere Bedeutung und Verantwortung in diesen Bereichen zukommt, wird der Ruf nach einer fehlerfrei funktionierenden Software immer lauter. Softwarefehler können Personen-, Sach- und Vermögensschäden verursachen. Beispielsweise mussten Sparkassen und Landesbanken einen Vermögensschaden von dreistelliger Millionenhöhe verbüßen, da EC-und Kreditkarten den Jahreswechsel von 2009 auf 2010 nicht verarbeiten konnten und schlichtweg nicht mehr lesbar waren [24]. Schwerwiegender war der Fehler eines Luftabwehrsystems, welches im Golfkrieg eingesetzt wurde. Hiermit sollten gegnerische Raketen erfasst und abgewehrt werden. Allerdings reduzierte sich die Präzision des Systems nach längerer Betriebszeit. Aus diesem Grund konnte bei einer ca. 100 Stunden andauernden Verteidigung einer amerikanischen Baracke eine feindliche Rakete nicht genau erkannt werden. Durch diesen Softwarefehler starben 28 Soldaten [25]. Das sind nur zwei Beispiele für die zahlreich auftretende Softwarefehler, die durch ausreichendes Testen hätten vermieden werden können.

3.1 Ziele und Bedeutung von Software-Tests

Im Allgemeinen sind Software-Tests notwendig zur Überprüfung von Produkteigenschaften und eventuellen Produktrisiken.

„Das Ziel eines Software-Tests ist es, durch Ausführung des Testobjekts auf einem Rechner mit ausgewählten Testdaten in einer definierten Umgebung festzustellen, ob sich das Testobjekt in diesen Fällen so verhält, wie es eine definierte Testreferenz vorschreibt.“ [2]
Ein Testobjekt kann dabei sowohl ein Teil eines Softwaresystems widerspiegeln oder aber auch das gesamte System. Das angestrebte Verhalten eines Testobjekts ist in einer Testreferenz vermerkt, gegen welches es wiederum getestet wird. Die Testreferenz wird im Pflichtenheft, also im Anfangsstadium des Software - Entwicklungsprozesses erarbeitet. Verhält sich ein Testobjekt so wie in der Testreferenz erwartet, kann man von einer guten Software-Qualität sprechen.
Im Allgemeinen kann gesagt werden, dass Tests ausgeführt werden, um Fehler, also Abweichungen vom in der Testreferenz festgelegten Ausgang, aufzudecken. Testdaten sollten so gewählt werden, dass möglichst viele Fehler aufgedeckt werden. Da für die ausgewählten Testdaten festgestellt werden kann, ob ein Testobjekt funktioniert, bedeutet das allerdings noch nicht, dass das Testobjekt fehlerfrei ist. Dijkstra äußerte sich 1972 zu diesem Thema folgendermaßen:
“Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.” [3] (Durch das Testen kann nur die Anwesenheit, nie aber die Abwesenheit von Fehlern bewiesen werden.)
Man kann nie davon ausgehen ein Testobjekt vollständig getestet zu haben, denn dafür müsste man alle möglichen Eingaben und Wertekombinationen, die auftreten können, abfangen. Da die Anzahl dieser Möglichkeiten bei komplexen Programmen geradezu ins Unendliche steigt, muss versucht werden mit einer geringen Anzahl an Tests eine hohe Testabdeckung zu erreichen.

3.2 Arten von Software-Tests

Tests müssen sicherstellen, dass ein Software-System auf bestimmte Eingaben oder Wertekombinationen in vorgesehener Weise reagiert und somit für den Markt geeignet ist. Außerdem sollten sie in der Lage sein, Schäden und Einschränkungen bei der Anwendung von Software aufzuzeigen. Diese Prüfung kann auf verschiedenste Art und Weise passieren. Es gibt die dynamischen Verfahren, wofür eine Ausführung des Systems notwendig ist und die statischen Verfahren, bei der auf eine Ausführung der Software verzichtet werden kann.

3.2.1 Statische Verfahren

Beim Statischen Verfahren wird das Testobjekt einer Analyse unterzogen. Für diese Analyse wird bevorzugt mit Werkzeugen gearbeitet, sie kann allerdings auch durch eine oder mehrere

Personen erfolgen. Das Ziel der Untersuchung ist es Fehlerquellen und Verstöße gegen Spezifikationen bspw. dem Pflichtenheft oder Standards bspw. der Java Code Conventions aufzudecken. Bei dem statischen Verfahren muss in folgende Bereiche untergliedert werden:

- Reviews: Die Reviews bezeichnen neben einem bestimmten Vorgehen bei der Prüfung von Dokumenten auch das gesamte Prinzip der Prüfung der Testobjekte durch Personen. Reviews sollten sofort nach der Fertigstellung bestimmter Dokumenten erledigt werden, um so eventuelle Fehler schon am Anfang der Entwicklung zu eliminieren, so dass sie nicht erst nach einiger Entwicklungsarbeit zum Vorschein kommen. Reviews sind eine kostengünstige Maßnahme zur Fehlerbeseitigung, zusätzlich steigern sie die Produktivität, durch das frühzeitige Ausbessern von Fehlern und sorgen für ein einheitliches Verständnis über das Testobjekt im Team.
- Statische Analyse: Diese Analyse wird durch eigens dafür konzipierte Werkzeuge durchgeführt, bspw. eine Rechtschreibanalyse. Obwohl solch Analyse durch Hilfswerkzeuge erledigt wird, kommt es hier nicht zur Ausführung des Testobjekts. In der Praxis ist meist der Quellcode das einzige formale Dokument was einer Analyse unterzogen werden kann. Durch die Unterstützung eines Werkzeugs ist hier der Aufwand wesentlich geringer als bei einem Review. Analysewerkzeuge wären beispielsweise der Compiler oder ein Werkzeug zur Überprüfung der Einhaltung von Standards.

3.2.2 Dynamische Verfahren

Entgegengesetzt zum Statischen Verfahren wird beim dynamischen Verfahren eine Ausführbarkeit der Software und damit auch der Testobjekte vorausgesetzt. Das Testobjekt wird hierbei mit Eingabedaten versehen und ausgeführt. Um einen dynamischen Test durchzuführen, müssen zuerst Bedingung, Voraussetzungen und Ziele der Tests festgelegt werden. Danach werden Testfälle entworfen. Diese Aufgabe gehört zu den schwerwiegendsten, denn hiervon hängen die Qualität des Tests ab und damit auch die Qualität des gesamten Systems. Im weiteren Verlauf kommt es dann zur Ausführung der Tests. Es sollte eine Testsequenz, also eine Reihe von Testfällen geben, die nacheinander ausgeführt werden können.

Zur Erstellung der einzelnen Testfälle können folgende Verfahren verwendet werden:

- Blackbox – Verfahren: Bei diesem Verfahren wird das Testobjekt als schwarzer Kasten angesehen, sodass der innere Aufbau verborgen bleibt. Man beobachtet dementsprechend das Testobjekt von außen. So können die Interaktionen zwischen

den einzelnen Komponenten eines Systems getestet werden und dafür Testfälle entworfen werden. Die Testfälle werden hierbei aus der Spezifikation abgeleitet. Aufgrund vieler möglicher Eingaben und Kombinationen ist ein vollständiger Test relativ unmöglich, daher muss eine Auswahl aus repräsentativen Testfällen getroffen werden. Dies geschieht bspw. mittels Äquivalenzklassenbildung, einer Erstellung eines Zustandsautomaten oder einer Ursache-Wirkungs-Graph-Analyse.

- Whitebox – Verfahren: Dieses Verfahren wird oft auch als codebasiertes Verfahren bezeichnet, das heißt hierbei muss der Programmtext im Gegensatz zum Blackbox-Verfahren vorliegen. Das Konzept eines White-Box Tests ist es, jede einzelne Codezeile des zu testenden Objekts einmal zur Ausführung zu bringen. Das setzt eine Ausführung des Programmteils bzw. des gesamten Programms voraus. Bei der Erstellung der Testfälle sollte wieder auf die Spezifikation zurückgegangen werden, um festzustellen, ob ein fehlerhaftes Verhalten vorliegt. Dafür werden verschiedene Verfahren genutzt, bspw. die Anweisungsüberdeckung, Zweigüberdeckung oder Pfadüberdeckung.

3.3 Testautomatisierung

Aufgrund drastisch sinkender Entwicklungszeit wird der Ruf nach einer Automatisierung des Testprozesses immer lauter. In den letzten Phasen der Softwareentwicklung sind Entwickler meist in großer Zeitnot und müssen bedauerlicher Weise Abstriche beim Testen machen. Um das Testen so effektiv wie möglich zu gestalten und langfristig ökonomisch durchführen zu können, besteht die Möglichkeit den Testprozess zu automatisieren. Das hat den Vorteil, dass das Wiederholen von bestimmten Tätigkeiten erleichtert wird. Dazu gehören bspw. die Ausführung aller Testklassen und deren Ergebnisauswertung. Testtools sind in der Lage Testklasse anhand von Testskripten auszuführen, Informationen über den Verlauf des Tests zu sammeln und objektive Messungen, wie z. B. Testabdeckung auf einfache Weise durchzuführen.

Testmanagementwerkzeuge unterstützen den Testmanager in Aufgaben wie Teststeuerung oder Testüberwachung. Sie stellen meist die Beziehung zwischen den Testfällen, der Testbasis bzw. -spezifikation und den Testbedingungen her. Ein Testmanagementsystem ist allerdings auch in der Lage die Testergebnisse zu dokumentieren und Berichte über den Testfortschritt zu erstellen.

Neben Testmanagementwerkzeugen gibt es auch Prüfwerkzeuge, die den Testprozess effizienter gestalten. Solche Werkzeuge verwalten Reviewanmerkungen und entstehende

Dokumente wie bspw. Checklisten zur Durchführung der Prüfung und deren Ergebnisse, sie übernehmen auch wichtige Verwaltungsaufgaben, wie z. B. Aufwanderfassung. Weiterhin sind sie hilfreich bei Ermittlung von Produktmaßen, wie bspw. Einhaltung der Java Code Conventions, Angaben zur Verbesserung des Codes oder eine Ermittlung der Testabdeckung und visualisieren meist diese.
Des Weiteren gibt es noch Werkzeuge, die bei der Erstellung von Testfällen hilfreich sind. Dabei wird zunächst der Quellcode in ein Systemmodell, welches meist durch UML-Diagramme dargestellt wird, überführt. Diese Werkzeuge werden Testentwurfswerkzeuge genannt und ihr Vorgehen nennt man modellbasiertes Testen. Eine große Bedeutung bei der Testautomatisierung kommt den Testdatengeneratoren zu. Sie unterstützen den Tester bei der Entwicklung von Testdaten aus einer Datenbank oder einer Datei. Testdaten sind dabei meist eine anonymisierte Kopie von Daten aus gängigen Systemen.
Um sichergehen zu können, dass das erstellte Projekt einer bestimmten Beanspruchung standhalten kann, müssen Leistungstests erfolgen. Sie enthalten Lastprofile. Damit wird das Projekt unter einem bestimmten Gesichtspunkt belastet ausgeführt. Auf diese Weise können Speicherlecks und Adressierungsprobleme aufgedeckt werden.
Testdurchführungswerkzeuge führen anhand von Skripten Testfälle eigenständig aus. Dies erspart dem Entwicklerteam eine Menge Zeit. Solche Testwerkzeuge müssen über ein Testrahmen verfügen, der für die Ausführung der Systemkomponente und deren Analyse verantwortlich ist bspw. wäre JUnit zu nennen. Wichtige Bestandteile des Testrahmens sind Testtreiber und Platzhalter. Testdurchführungswerkzeuge verfügen über Überdeckungsanalysatoren. Sie überprüfen die Ausführung bestimmter Strukturelemente, anhand von Instrumentierungen des Quellcodes. Das bedeutet der Quellcode des zu testenden Objekts wird mit Zählern ergänzt. Kommt es nun zur Ausführung dieses Elements durch einen Tests wird der Zähler erhöht. Dies geschieht beliebig oft, je nach dem wie oft das Element durch einen Test ausgeführt wird. Die Instrumentierung kann auf drei Arten erfolgen[26]:

- Instrumentierung erfolgt direkt im Quellcode: Das führt zu einer Vermischung von Programmcode und Metriken und wird daher eher selten genutzt.
- Instrumentierung erfolgt anhand eines Duplikats des Quellcodes. Diese Variante liefert eine klare Trennung von Instrumentierung und Quellcode, hat allerdings im Gegensatz dazu einen höheren Speicherbedarf.

- Instrumentierung auf Bytecodeebene: Diese Instrumentierungsart ist nur bei Java-Anwendungen möglich. Solche Anwendungen werden nicht direkt in Maschinencode gewandelt, sondern durch den Compiler in einen Zwischencode transformiert, den Bytecode. An dieser Stelle erfolgt dann die Instrumentierung. Es wird dabei nur eine Datei erstellt, die alle Zählerwerte des kompletten Programms enthält. Das hat den Vorteil, dass keine Kopie der Quelltext-Datei erstellt werden muss und auch direkt keine Änderungen im Quelltext vorgenommen werden müssen.

4 Visualisierungen von Testdaten

Die Visualisierung im Allgemeinen ist eine kognitive Aktivität des Menschen. Dies bezeichnet die Erstellung eines mentalen Bildes oder die Verwendung eigener Erinnerungen, um das Verständnis zu einem bestimmten Gegenstand oder Sachverhalt sicherzustellen und zu verbessern. In der Computertechnologie beschreibt es das graphische Abbilden von abstrakten Daten, um dem Betrachter eine Auswertung eben dieser Daten zu erleichtern [12].

„Visualisierung ist der Prozess der Transformation von Informationen in eine visuelle Form, die es dem Benutzer auf visuelle Weise gestattet, verborgene Aspekte in den Daten zu entdecken, die für Exploration und Analyse wesentlich sind.“ [4]

Die Visualisierung lässt sich in viele verschiedene Untergruppen gliedern. Beispielsweise ist die Architekturvisualisierungen, die genutzt wird, um zu zeigen wie sich eine Bauwerk in seine Umgebung einfügt oder die medizinische Visualisierung, im Zusammenhang auf Diagnose und Auswertung eines C.T.‘s. Ein weiterer spezieller Zweig der Visualisierung ist die Informationsvisualisierung.

„Informationsvisualisierung ist die Nutzung computergenerierter, interaktiver, visueller Repräsentationen von abstrakten, nicht-physikalischen Daten zur Verstärkung des Erkenntnisgewinns.“ [4]

Die Informationsvisualisierung beschäftigt sich mit der Visualisierung von abstrakten Daten, die nicht mit physikalischen Zuständen assoziiert werden können [13]. Der Unterschied zur medizinischen oder architektonischen Visualisierung liegt hauptsächlich bei der Nutzergruppe. Im Gegensatz zur Visualisierung sind es bei der Informationsvisualisierung nicht Experten wie zum Beispiel Physiker oder Mediziner, die die erstellten Bilder auswerten, sondern die Nutzergruppe ohne mathematischen oder natur- und ingenieurwissenschaftlichen Hintergrund [4]. Die Hauptaufgabe ist es, die abstrakten Informationen auch dem

Alltagsnutzer zugänglich zu machen. Im Alltag begegnet uns die Informationsvisualisierung in Abbildungen von Nahverkehrsnetzen, Wettervorhersagen oder Auswertungen von Börsenergebnissen. Sie schafft dem Betrachter die Möglichkeit abstrakte Daten intuitiv auszuwerten und sofort Schlüsse aus der betrachteten Abbildung zu ziehen. So sind zum Bespiel bei Nahverkehrsnetzen die Routen der Züge farblich gekennzeichnet. Auf diese Weise wird dem Betrachter die Möglichkeit eröffnet, zügig seine Route zum gewünschten Ziel zu ermitteln und so eventuell noch pünktlich den richtigen Zug erreichen zu können.
Die Informationsvisualisierung wird auch im Gebiet der Informatik angewandt. Dort wird der Wunsch Software-Eigenschaften zu visualisieren immer größer, aufgrund der umfangreichen Auskünfte, die mittels einer Metrik über ein System gegeben werden können. Dies ermöglicht den Firmen eine vertrauensvolle Überwachung und Kontrolle des Systemzustands. Die Software-Metriken sind meist in textueller Form schwer zu überblicken und führen im schlimmsten Fall zu einer Falschinterpretation gegebener Informationen. Um die Möglichkeit einer Falschauswertung und den Interpretationsaufwand und –zeit einzugrenzen, werden diese Metriken oft visualisiert. Zu solchen Metriken gehören unter anderem auch die Testdaten.

4.1 Ziele und Bedeutung von Visualisierung von Testdaten

Visualisierungen sind für den Menschen von großer Wichtigkeit. Es ist wissenschaftlich nachgewiesen, dass der Mensch 75% seiner Eindrücke der realen Welt visuell aufnimmt und die restlichen 25% auditiv oder mit Hilfe andere Sinnesorgane [14]. Weiterhin wirken sich Visualisierungen positiv auf das Gedächtnis und die Aufnahmefähigkeit aus. Das bedeutet, visualisiert man Informationen kann man davon ausgehen, dass sie sich besser einprägen und schneller verstanden werden.
Typische Ziele der Visualisierung sind nach Preim und Dachselt [4]:

- Entdecken von neuen Zusammenhängen oder Besonderheiten (discovery)
- Zuverlässiges Treffen von Entscheidungen (decision making)
- Explorative Analyse von Informationsräumen (exploration)
- Finden von Erklärungen von Mustern, Gruppen von Datenobjekten oder einzelnen Eigenschaften (explanation)

Diese Ziele können auch auf die Visualisierung von Testdaten bezogen werden. Sie sollen schnell und aussagekräftig analysiert und interpretiert werden können, um auf diese Weise eventuell noch nicht bekannte Zusammenhänge in Erfahrung zu bringen. Der Betrachter soll in der Lage sein, aufgrund seiner Interpretation der Visualisierung zuverlässige Entscheidungen treffen zu können. Die Visualisierung von Testdaten sollte zu einer

entdeckenden Analyse des virtuellen Informationsraums beitragen und das Verständnis über Testfehler und Testabdeckung beim Betrachter erhöhen und somit neue Erkenntnisse über die Datenmenge und ihrer Eigenschaften hervorbringen.

Die Visualisierung von Testdaten wird außerdem nötig, da heutzutage Systeme immer komplexer und umfangreicher werden. Sie besitzen eine immer ansteigende Anzahl an Funktionen, zusätzlich soll dem Kunden eine stetig steigende Software-Qualität geboten werden. Eine Auswertung der Testdaten ist auch ein Indiz für die Software-Qualität und kann bei umfangreichen Systemen nur anhand von einer Visualisierung dargestellt und ausgewertet werden. Aufgrund des extremen Umfangs ist es möglich, dass der Entwickler schnell die Übersicht über die Testergebnisse und die Testabdeckung verliert. Eine Visualisierung erlaubt dem Entwickler oder dem Testmanager, eine intuitive Auswertung und Übersicht über den Testfortschritt, Testabdeckung und Testerfolg des Projekts und stellt somit auch eine hohe Software-Qualität sicher. Werden Tests vernachlässigt oder falsch interpretiert, kann das folgenschwere Nachteile haben. Bei der Visualisierung von Testdaten stehen vor allem die Testabdeckung und die Testergebnisse im Vordergrund. Visualisiert man diese, gibt man dem Testmanager die Möglichkeit, eine schnelle und konkrete Auswertung des Testzustands vorzunehmen. Er ist damit in der Lage einen besseren Überblick über den Testfortschritt des gesamten Projekts zubekommen. Der Testmanager sieht genau an welcher Stelle keine Tests existieren, also die Testabdeckung niedrig ist, und kann dort demnach mehr Fachpersonal zum Erstellen von Tests einsetzen. Das Gleiche gilt für die Testergebnisse. In Gebieten vieler fehlgeschlagener Tests, ist der Testmanager in der Lage mehr Fachkräfte zu engagieren, die die Fehler beheben.

Im Allgemeinen kann gesagt werden, dass durch Visualisierung der Testabdeckung und des Testerfolgs, eine Verbesserung der Wirtschaftlichkeit, im Hinblick auf die Einteilung der Fachkräfte vorgenommen wird. Zusätzlich wird durch die Informationsvisualisierung effizienter in der Softwareentwicklungsphase „Testen“ vorgegangen. Da meist in der Test-phase die Zeitnot sehr groß ist, wird dort eine Visualisierung dringend benötigt. Denn Visualisierung dieser Software-Metrik verkürzt die Analysezeit der Testauswertung. Daher könnte die Behauptung aufgestellt, dass eventuell mehr Fehlzustände der Software in kürzerer Zeit aufgedeckt und korrigiert werden und somit die Software-Qualität gesteigert wird. Es kann außerdem aber auch auf einen Blick festgestellt werden, ob die Testaktivität beendet werden kann, eben wenn bestimmte vom Auftraggeber gestellte Kriterien bspw. der Überdeckungsgrad erfüllt sind.

4.2 Grundlegendes Konzept der Visualisierung

Da aus abstrakten Daten Bilder entstehen sollen, müssen die Daten transformiert werden. Das kann im Allgemeinen an einer Visualisierungspipeline erklärt werden. Dort durchschreiten die Daten laut Schumann [15] folgende drei Schritte.

- Datenaufbereitung: Diese Phase wird auch als Filtering bezeichnet. Ziel ist es hierbei die Daten aufzubereiten, sie also in einer für die Visualisierung geeigneten Form bereitzustellen. Die Daten müssen gefiltert werden, da meist von einer umfangreichen Datenmenge nur eine bestimmte Teilmenge für eine Visualisierung benötigt wird.
- Geometrische Transformation: In dieser Phase der Visualisierungspipeline müssen die aufbereiteten Daten in Geometriedaten transformiert werden, wenn sie noch nicht in geometrischer Form vorliegen.
- Bildgenerierung: Aus den in der vorherigen Phasen erstellten Geometriedaten werden nun ein oder mehrere Bilder erstellt. Dabei können unterschiedliche Techniken angewandt werden. Dies ist abhängig von dem Ausgabemedium und dem Anwendungszweck.

Abbildung 6: Allgemeine Visualisierungspipeline

Diese Visualisierungspipeline wird im Grundprinzip von vielen anderen Modellen aufgegriffen. So geschieht das auch bei dem Data State Reference Model [16] [17], wie in Abbildung 1 zusehen. Die visuelle Darstellung von Testdaten lässt sich ebenfalls auf dieses Modell beziehen.

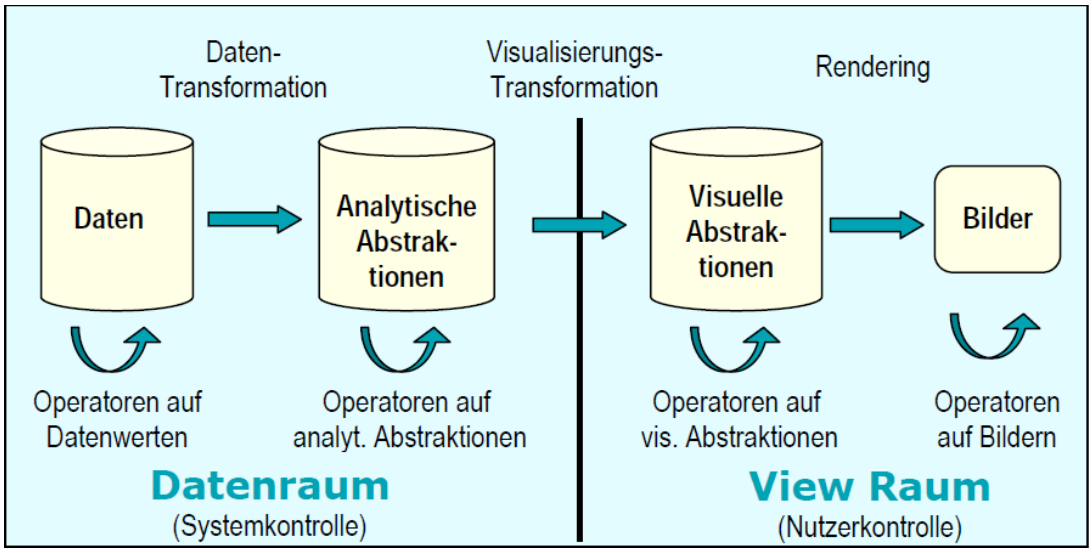

Abbildung 7: Data State Reference Model [16]

Beim Data State Reference Model lassen sich vier Stadien der Visualisierungspipeline erkennen. Der Begriff Visualisierungspipeline wird verwendet, da es nicht möglich ist eine dieser vier Stadien zu überspringen. Demzufolge müssen alle Stadien hintereinander ausgeführt werden, um das korrekte Ergebnis zu erhalten. Die vier Stadien der Visualisierung setzen sich dabei zusammen aus Daten, analytische Abstraktion der Daten, visuelle Abstraktion der Daten und letztendlich der Veranschaulichung der Daten. Zwischen den einzelnen Stadien werden die Daten transformiert [17]. Das erste Stadium, in der Abbildung „Daten“ betitelt, spiegelt das Vorhanden sein der Rohdaten dar. Diese Daten können bspw. Koordinaten eines 3D Modells sein oder aber auch Aktienwerte pro Monat. Das zweite Stadium der Visualisierungspipeline ist die Analytische Abstraktion, dort werden die Daten über Daten ermittelt, es kann hier auch von Meta-Daten gesprochen werden. Zur Ermittlung dieser Daten werden analytische Operatoren verwendet. Um die Beispiele der Aktien und das 3D- Modell wieder aufzugreifen, werden hier bspw. die Abstände der Aktienwerte zwischen den Monaten berechnet oder eine Normalisierung der Daten des 3D-Modells vorgenommen. Diese Daten müssen nun separiert werden. Man muss sich die Frage stellen: „Welche Daten lassen sich visualisieren?“. Ist eine Aufteilung der Daten erfolgt, befinden sich die Daten in der visuellen Abstraktionsphase des Data Reference Models. Von dort aus werden die visualisierbaren Daten in die letzte Phase weitergereicht, in der sie letztendlich mit Hilfe von grafischen Elementen dargestellt werden und zur Auswertung bereitstehen.

Bezieht man nun das Data Reference Model auf die Visualisierung von Testdaten sehen die Stadien folgendermaßen aus.

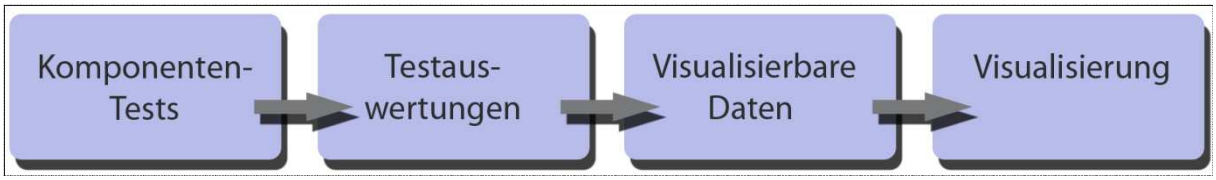

Abbildung 8: Visualisierungspipeline von Testdaten

Am Anfang der Visualisierungspipeline stehen die Tests selbst. Sie spiegeln die Rohdaten dar. In der zweiten Phase werden die Informationen über die Daten ermittelt. Das bedeutet dort werden anhand von Werkzeugen die Testabdeckung und der Testerfolg berechnet. Diese Daten enthalten noch einige Informationen, die nicht visualisierbar sind oder nicht zum Verständnis der Testauswertung beitragen. Daher werden hier wichtige Daten, wie bspw. Zeilenanzahl, Anzahl der abgedeckten Zeilen und Anzahl der Fehler gesammelt und weiter an die Visualisierung gereicht. Dort werden diese Daten dann anhand bestimmter Elemente und Farben dargestellt. Sie stehen so dem Testmanager oder dem Entwickler zur Auswertung zur Verfügung.

4.4 Anforderungen an eine Visualisierung

Visualisierungen sollten nach Schumann und Müller [6] folgende wichtige Anforderungen erfüllen:

- Expressivität: Eine Visualisierung sollte eine Datenmenge möglichst korrekt und genau wiedergeben, das bedeutet die Daten sollten vor der Visualisierung nicht verfälscht werden.
- Effektivität: Da es für eine Datenmenge viele Visualisierungsarten gibt, gilt es die Effektivste zu verwenden. Diese Visualisierungsart nutzt die Fähigkeit des Betrachters und des Ausgabegeräts optimal aus.
- Angemessenheit: Die Erstellung einer Visualisierung ist stets mit Aufwand und Kosten verbunden. Dabei gilt es abzuschätzen, ob sich der Zeit- und auch Kostenaufwand im Gegensatz zum Nutzen der Visualisierung auch auszahlt.

Zusätzlich sollte aber auch noch folgende Anforderung erfüllt sein:

- Interaktivität: Der Betrachter einer Visualisierung muss in der Lage sein, einen Überblick über das Gesamte zu bekommen und auch in Interessengebieten hinein zoomen zu können. Dabei ändert sich nur das Erscheinungsbild der Visualisierung nicht in etwa die Datenbasis. An dieser Stelle werden nur die Parameter der Visualisierung geändert, dies geschieht in diesem Fall interaktiv und durch eine Neugenerierung des Bildes. Der Benutzer ändert dabei einen Parameter und der Computer antwortet darauf mit einem neuen Bild. Somit ist

der Benutzer in der Lage die Visualisierung auf seine Bedürfnisse anzupassen[18].

4.3 Bewertung genutzter Konzepte für die Visualisierung von Testdaten

In dem ersten Kapitel wurde bereits beschrieben, welche Programme sich zur Visualisierung von Testdaten eignen und zurzeit auf dem Markt befinden. Weiterhin wurde auf die genutzten Visualisierungskonzepte eingegangen. In diesem Abschnitt sollen die Visualisierungskonzepte näher erläutert und Vor- und Nachteile selbiger genannt werden.
Hierfür wird noch einmal eine abgewandelte Form der Übersichtstabelle aus dem Kapitel Stand der Technik aufgegriffen.

Werkzeug	**Testabdeckung**	**Testerfolg**	**Visualisierung**
Clover	Ja	Ja	-Statusanzeige -Treemap -Diagramme -Editoransicht
Cobertura	Ja	Nein	-Tabellen mit Statusanzeigen -Editoransicht
Emma	Ja	Nein	-Tabelle -Editoransicht
Pin	Ja	Nein	-3D-Treemap -Editoransicht -SeeSoft
Tarantula	Ja	Ja	-SeeSoft

Tabelle 1: Übersicht bestehender Programme und ihrer Visualisierungskonzepte

Wie in der vorherigen Tabelle zu erkennen, gibt es verschiedene Arten eine Testauswertung darzustellen. Die gängigsten Arten sind dabei Tabellen, Statusanzeigen, Diagramme, Editoransichten, Treemaps und SeeSoft-Darstellungen. Im Folgenden werden Vor-und Nachteile dieser Visualisierungskonzepte genannt und sie anhand des Information Seeking Mantra [5] von B.Shneiderman bewertet.

4.3.1 Information Seeking Mantra

Eine effiziente und benutzerfreundliche Weise der Informationsvisualisierung liefert das Visual Information Seeking Mantra (Mantra visueller Informationssuche) von B.Shneiderman [5]. Diese Richtlinie beschäftigt sich mit dem Visualisierungsgrundsatz: „Overview first,

zoom and filter, then details-on-demand“. Zu erst sollte der Anwender in der Lage sein, sich einen Überblick (Overview), über das gesamte Projekt zu verschaffen. Der Gesamteindruck dient der Orientierung, z. B. eine Landkarte beim Internetdienst GoogleMaps. Dem Benutzer wird es ermöglicht einen Ausschnitt der Landkarte im Verhältnis zur Gesamtkarte zu sehen. Die Anforderung Zoom besagt, dass eine Visualisierung in der Lage sein muss, einen bestimmten Bereich, für den sich der Anwender interessiert, vergrößern zu können. Dabei sollte es dem Nutzer ermöglicht werden bestimmte Informationen zu filtern und für ihn unwichtige Informationen von dieser Sicht zu entfernen. Diese Anforderung wird durch das im „Information Seeking Mantra“ erwähnten Filter beschrieben. Bezieht man sich nun erneut auf das Beispiel einer Landkarte, erkennt man, dass dort an Interessengebiete herangezoomt werden kann und die restlichen Bereiche von der Visualisierung entfernt werden können. Auf diese Weise werden dem Anwender wirklich nur die benötigten und für ihn interessanten Daten gezeigt. Möchte man nun allerdings bestimmte Details zur angezeigten Information in Erfahrung bringen, wie zum Beispiel Straßennamen oder Stadtnamen, können sie nachträglich aktiviert werden. Dies wird in dem Visualisierungsgrundsatz mit details-on-demand (Details auf Wunsch)beschrieben. Dieser Grundsatz nennt die 4 Hauptaufgaben einer Visualisierung, die sich ein Nutzer vorstellt, um schnell und effizient eine Auswertung vornehmen zu können. B.Shneidermann nennt in Taxonomy of Information Visualizations [5] (Taxonomie von Informationslösungen) weitere grundlegende Aufgaben einer Visualisierung. Sie sind Relate, History, Extract. Besteht der Wunsch die Beziehung zwischen einzelnen Objekten zu erkennen, muss dem Nutzer diese Möglichkeit offenstehen. Durch die Relate(beziehen)-Aufgabe lässt sich ein Zusammenhang zwischen angezeigten Objekten leichter und besser erfassen und gliedert das Objekt dementsprechend ins gesamte Bild ein. Zum Beispiel ist die Information über die Zugehörigkeit einer Stadt zu einem Bundesland zu nennen. Ebenfalls muss dem Nutzer die Möglichkeit eröffnet werden, seine durchgeführten Aktionen rückgängig zu machen oder wiederherzustellen, daher sollte eine History(Geschichte)-Funktion in einem Programm bestehen. Zu guter Letzt sollte der Nutzer mit Hilfe der Extract(gewinnen, entnehmen)-Funktion in der Lage sein, die Ergebnisse der visuellen Suche oder die durchgeführten Aktionen zu speichern und auf andere Projekte anwenden zu können. Die Nutzer der Visualisierung sollen in der Lage sein, das Ergebnis der Manipulation der Daten immer wieder aufrufen zu können.

4.3.2 Tabellen

Tabellen sind geordnete Darstellungen von Werten oder Texten. Dabei wird eine Unterteilung der Daten in Zeilen und Spalten vorgenommen. Tabellen geben in fast allen

Anwendungsgebieten der Informationsvisualisierung einen sehr guten Gesamtüberblick. Der Betrachter kann dabei selbst entscheiden welche Bereiche ihn interessieren und sich dabei einfach an Spalten und Zeilen orientieren. Bei der Visualisierung der Testdaten ist die tabellarische Form effektiver und platzsparender, so auch bei den Programmen Cobertura und Emma. Dabei können manche Anforderungen des „Information Seeking Mantra“ von B.Shneidermann angewandt werden. So liefert eine Tabelle eine Übersicht über die Testauswertungen des ganzen Projekts. Der Nutzer kann die Information filtern, indem er sich am Tabellenkopf orientiert und nur in der Spalte bzw. Zeile sucht, die die Informationen enthält, die für ihn von Interesse sind. Sind diese Einträge nun noch interaktiv zugänglich, ist auch die letzte Anforderung vom „Information Seeking Mantra“ erfüllt. Denn der Nutzer kann nun auch Details von bspw. einer Klasse ansehen. Tabellenauswertungen sind bei immer umfangreicher werdenden Projekten zeitaufwändig, es kann zum „Verrutschen“ innerhalb der Tabelle kommen und so wiederum zu Fehlinterpretationen der Testauswertungen führen. Emma stellt die Testdaten nur durch Zahlen dar. Das trägt zur Erhöhung der Fehlinterpretation bei, denn anhand von Zahlen lassen sich Testdaten nur schwer auswerten. Bspw. könnte man denken, dass eine 31% Testabdeckung sehr gut wäre, es aber leider nicht ist und sich noch viele Fehler im Programmtext befinden könnten. Die Tabellen-Darstellung ist für eine effektive Übersicht zu empfehlen, schafft es allerdings nicht eine intuitive Testauswertung beim Betrachter zu veranlassen und trägt ein großes Risiko der Fehlinterpretation mit sich.

4.3.3 Statusanzeigen

Statusanzeigen oder auch Prozessfortschrittsanzeigen werden in vielen Bereichen der Visualisierung genutzt. Sie geben dem Betrachter die Möglichkeit schnell, intuitiv und effektiv Daten auszuwerten. Da meist bei Statusanzeigen eine Grünfärbung für den Fortschritt verwendet wird und eine Rotfärbung für die ausstehende Tätigkeit, geht man direkt auf das unterbewusste Empfinden des Menschen ein. So wird rot als Signalfarbe erkannt und warnt meist vor etwas. Das zeigt dem Anwender sofort, dass dort noch Tests fehlschlagen oder entwickelt werden müssen, um den Software-Fehler einzugrenzen. Clover verwendet diese Darstellung, um eine Auswertung der Testabdeckung und des Testerfolgs vornehmen zu können. Das „Information Seeking Mantra“ ist bei einzelnen Statusanzeigen leider nicht anwendbar, da sich nur ein Fortschritt für ein bestimmtes Objekt anzeigen lässt. Das bedeutet, man besitzt eine Statusanzeige für bspw. eine Klasse oder für das gesamte Projekt. Außerdem ist es nicht möglich in Interessenbereiche zu zoomen oder Details zum Objekt zu erhalten. Mischt man nun die Tabellendarstellung mit dem Konzept der Statusanzeigen wie es

Cobertura anstellt, lässt sich das „Mantra“ wieder anwenden. Denn man erhält durch die Tabellenform eine Komplettansicht der Testabdeckung oder des Testerfolgs anhand von Statusanzeigen. Ist diese Tabelle nun interaktiv zugänglich, kann eine nähere Betrachtung der Testdaten vorgenommen, sowie Details zum untersuchten Objekt erhalten werden. Alles in allem stellt dieses Konzept der Statusanzeigen eine intuitive Möglichkeit dar, um eine Auswertung vorzunehmen. Diese Darstellung allein liefert allerdings nicht die Möglichkeit ein Objekt mit einem anderen zu vergleichen. Dazu müsste eine Kombination aus Statusanzeigen und der Tabellenansicht verwendet werden.

4.3.4 Diagramme

Diagramme sind vielseitig verwendbare Darstellungen von Daten oder Sachverhalten. Sie eignen sich besonders, um einen Vergleich zwischen Daten zu visualisieren. Diagramme ermöglichen dem Nutzer eine schnelle Erkennung von Zusammenhängen und eine intuitive Auswertung. Es gibt viele verschiedene Arten von Diagrammen. Kreisdiagramme bspw. eignen sich gut, um prozentuale Anteile darzustellen. Balkendiagramme erleichtern den Vergleich zwischen vielen Objekten mit gleichen Attributen. Liniendiagramme eignen sich, um Anstiege oder Absenkungen von Attributsausprägungen zu erkennen. Clover nutzt bspw. Balkendiagramme. Die Darstellung gibt Informationen über das Attribut der Testabdeckung und lässt einen Vergleich der Attributsausprägungen zu. Auf so eine Art und Weise lässt sich ebenfalls der Testerfolg darstellen. Bei dem Konzept der Diagramme kann das „Information Seeking Mantra“ angewandt werden. Denn der Anwender ist in der Lage sich einen Überblick zu verschaffen und anhand der x- und y-Achse die Informationen zu filtern, die für ihn interessant sind. Sollte das Diagramm interaktiv zugänglich sein, kann der Benutzer Details zu einem bestimmten Objekt erhalten. Diagramme eignen sich sehr gut, um abstrakte Daten für Betrachter leicht zugänglich darzustellen, zusätzlich sind sie optisch ansprechender als bspw. Tabellen. Sie erlauben eine schnelle und intuitive Auswertung der Daten und ermöglichen das Erkennen von Zusammenhängen. Das Visualisierungskonzept der Diagramme bietet eine gute Möglichkeit wenige Attribute mit ihren Ausprägungen miteinander zu vergleichen. Zieht man nun bspw. ein Kreisdiagramm zur Arbeitszeitauswertung in Betracht, wird es problematisch dort ein weiteres Attribut wie bspw. Pausenzeiten zu vermerken. Diagramme werden bei viel darzustellenden Attributen schnell unübersichtlich und eignen sich sehr schlecht zum Vergleich mehrere Attribute. Weiterhin stellt sich die Darstellung der Diagramme als problematisch dar, wenn eine genaue Wertebetrachtung notwendig wird. Dies ist darauf zurückzuführen, dass meist bei der Einteilung der x- und y-Achse Wertebereiche abgetragen werden, um das Diagramm zu

skalieren. Alles in allem sind Diagramme optisch ansprechend und eignen sich gut zum Vergleich weniger Attribute und ihrer Ausprägungen. Sie können aber schnell unübersichtlich werden.

4.3.5 Editoransicht

Die Editoransicht bereichert die Testauswertung. Dabei wird der Quelltext zeilenweise mit Farben unterlegt, die Auskunft über Testerfolg oder Testabdeckung geben können. So sind durch fehlgeschlagene Tests ausgeführte Zeilen oder nicht ausgeführte Zeilen rot unterlegt. Grün gekennzeichnete Zeilen besagen, dass sie durch (erfolgreiche) Tests ausgeführt werden. Bei den im Stand der Technik bereits erwähnten Programmen wird bisher nur eine Auswertung der Testabdeckung am Quelltext vorgenommen. Man erhält leider keine Auskunft über die Testergebnisse bei den Programmen Clover, Cobertura und Pin. Tarantula ist dort alleinig in der Lage den Testerfolg neben der Testabdeckung zu visualisieren. Alles in allem eignet sich eine Editordarstellung zur Auswertung von Testabdeckung und Testerfolg sehr gut, da sofort ersichtlich wird in welchen Bereichen noch Tests entworfen werden müssen, bzw. noch Fehler im Quelltext vorhanden sind. Bezieht man sich allerdings nun auf das „Information Seeking Mantra“ fällt auf, dass der Betrachter keinen Überblick über das Projekt in einer Editoransicht erhält. Es würde sich sehr gut eignen, die Editoransicht mit der Tabellenansicht zu kombinieren, wie es bei Cobertura, Clover und Emma der Fall ist. Denn so kann ein Überblick über das gesamte Programm gewonnen werden und der Nutzer ist in der Lage in Interessengebiete zu zoomen, um dort einige Details zu erfahren.

4.3.6 Treemaps

Treemaps sind ein sehr starkes und fähiges Visualisierungskonzept für Testdaten. Treemaps zeigen hierarchische Strukturen innerhalb eines Projektes anhand von verschachtelten Rechtecken an. Die Größe der Rechtecke wird dabei durch bspw. die Anzahl der Zeilen bestimmt. Diese Rechtecke stellen dann Pakete oder Klassen dar. Sie können verschieden gefärbt sein, dass wiederum stellt verschiedene Attribute dar. In den Programmen Clover und Pin wird die Färbung genutzt, um den Abdeckungsgrad anzuzeigen. So bedeutet ein grün gefärbtes Rechteck eine hohe Testabdeckung und ein rot gefärbtes eine geringe Testabdeckung. Die Treemaps sowohl in der 2D als auch in der 3D Varianten sind sehr gute Darstellungsmittel, da sie auf kleinstem Raum viele Informationen anzeigen können und eine intuitive Auswertung erlauben. Bei dieser Darstellung wird auf das Unterbewusstsein eingegangen und dementsprechend die Farbe Rot für eventuelle Fehlerbereiche verwendet und die Farbe Grün für Erfolgsmitteilungen. Effektiv sind diese Darstellungen, da sie die

Hierarchie im Projekt anzeigen, dazu den Umfang oder den Anteil des Einzelobjektes am Ganzen und die Testabdeckung der Objekte. Ist die Treemap nun auch noch interaktiv zugänglich, kann der Benutzer durch das Projekt navigieren und Hierarchiestufen auf- oder absteigen. So erfüllen die Treemaps alle Anforderungen die im „Information Seeking Mantra“ genannt wurden. Nachteil ist wiederum, dass bei den Programmen Pin und Clover, leider kein Testerfolg mit der Treemap visualisiert wird. Dazu müsste eventuell eine Kombination mit der Statusanzeige erfolgen.

4.3.7 SeeSoft-Darstellung

Eine SeeSoft-Darstellung ermöglicht eine Auswertung der Testdaten direkt am Quelltext. Dabei wird der Quelltext aus der Vogelperspektive gezeigt, so wird dem Anwender eine Übersicht über die Quelltext-Datei erlaubt. Die Zeilen der Quelltext-Datei sind farbig eingefärbte, aneinandergereihte Pixel. Anhand dieser farbigen Pixel kann nun eine Auswertung über ein Attribut erfolgen. Bei dem Programm Pin entspricht das Attribut der Testabdeckung. Dabei bekommt man pro Datei eine Übersicht über nicht abgedeckte Zeilen, die rot eingefärbt werden und abgedeckte Zeilen, die grün gekennzeichnet werden. Bei Tarantula werden sogar zwei Attribute gleichzeitig dargestellt. Dabei wird die Testabdeckung durch farbig gekennzeichnete Zeilen dargestellt. Die Färbung dieser Zeilen ist wiederum abhängig von dem Erfolg des Tests, der diese Zeile abdeckt. Die SeeSoft-Darstellung erlaubt eine intuitive Auswertung der Testdaten, man benötigt kein Expertenwissen, um festzustellen, welche Bereiche nicht durch (erfolgreiche) Tests abgedeckt sind. Sie ermöglicht eine sehr schnelle Auswertung aufgrund der Vogelperspektiven. Der Anwender ist in der Lage die gepixelten Reihen zu vergrößern und so eine direkte Ansicht des Quelltextes zu bekommen. So ist also auch eine Auswertung der Testabdeckung und des Testerfolgs direkt am Quelltext möglich. Diese Vorteile spiegeln alle Anforderungen des „Information Seeking Mantra“ wieder. Nachteile dieser Darstellungen sind allerdings, dass maximal zwei Attribute gleichzeitig dargestellt und ausgewertet werden können. Diese Darstellung schlägt fehl, wenn nun auch noch Informationen zum Objekt wie z. B. Klassenname oder Zugehörigkeit zu einem Paket dargestellt werden sollen.

4.3.8 Zusammenfassung

Jedes dieser Visualisierungskonzepte hat Vor- und Nachteile. Sie eignen sich gut zur Auswertung der Testdaten und visualisieren auch im entsprechenden Maße. Allerdings stechen zwei Visualisierungskonzepte heraus. Die Treemap-Darstellung und die SeeSoft-Darstellung in Kombination wären ein optimales Visualisierungstool für Testdaten. Da sie

sowohl für Experten als auch Laien einen sehr guten Zugang zu abstrakten Daten liefern und umfangreiche Informationen über diese Daten grafisch darstellen können. Bei all diesen Visualisierungskonzepten fällt allerdings auf, dass sie nie revisionsübergreifend visualisieren. Das heißt man erkennt leider nicht, ob Fehler bspw. in der aktuellen Version erst entstanden sind oder in der vorherigen schon vorhanden waren. Der einzige Weg wäre es hier eine Darstellung einer vorherigen Version neben die Aktuelle zu stellen, um einen revisionsübergreifenden Vergleich führen zu können.

5 Implementierung

In diesem Kapitel sollen die Arbeitsabläufe der Umwandlung von den Komponententests in eine angemessene Visualisierung erläutert werden. Dabei wird eine Reihe von Werkzeugen verwendet auf die im Folgenden näher eingegangen werden soll.

5.1 Genutzte Visualisierungspipeline

Um eine Transformation der Komponententests in eine ansprechende, auswertbare Visualisierung durchzuführen, sind einige Zwischenschritte notwendig. In der Abbildung 1 soll der Prozess der Datenumwandlung verdeutlicht werden.

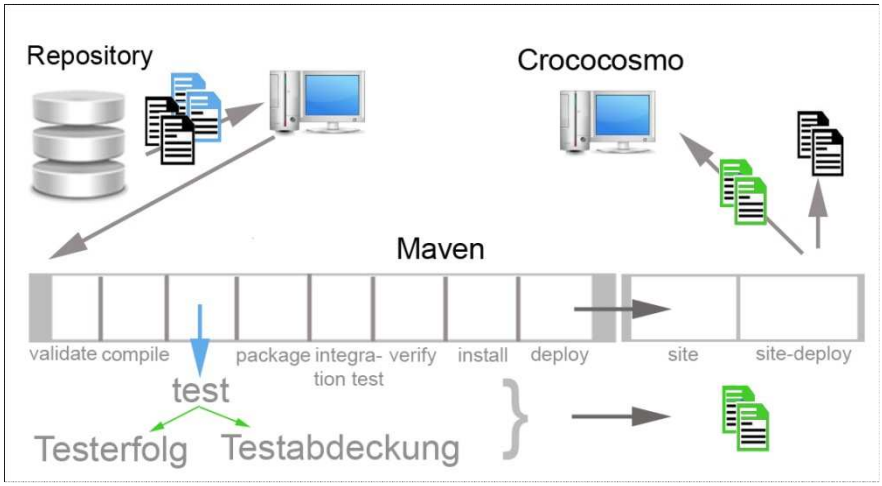

Abbildung 9: Visualisierungspipeline

Wie in der vorhergehenden Abbildung zu erkennen, werden drei Hauptverarbeitungsschritte durchlaufen. Der erste Schritt ist es ein Projekt aus einem Repository zu laden. Dabei müssen folgende Voraussetzungen erfüllt werden:

- Das Projekt muss ein Java-Programm sein.
- Das Projekt muss JUnit-Tests enthalten.
- Das Projekt muss eine POM.xml enthalten.

Ist das Projekt auf dem Computer geladen, wird es von Maven, einem Buildmanagement-Tool, bearbeitet. Es durchläuft dabei zwei Phasen, in Maven Lebenszyklen genannt. Der erste Lebenszyklus, der Build - Lifecycle, besteht dabei aus acht Phasen, wobei nur drei zur Auswertung der Software-Tests benötigt werden. In der ersten Phase validate wird dabei die Struktur der POM.xml, der Steuerdatei für Maven, überprüft. In der zweiten Phase wird eine Kompilierung des Quelltextes durchgeführt. Hierbei wird der Quelltext in Maschinencode umgewandelt, um ihn für den Computer ausführbar zu machen. In der dritten Phase des Build - Lifecycles kommt es zur Ausführung der JUnit-Tests. Dabei wird durch das Surefire-Plug-In die Ausführung der Tests angestoßen und Informationen über den Testerfolg gesammelt. In diesem Zusammenhang wird ein weiteres Plug-In genutzt, das Cobertura Plug-In. Es nimmt eine Instrumentierung des Quelltextes auf Bytecode-Ebene[8] vor, woraus dann Informationen über die Testabdeckung gesammelt werden können. Das SureFire-Plug-In liefert dabei eine Auskunft über die Testfehler. Diese Informationen müssen nun noch in Dateien gespeichert werden. Dafür wird ein zweiter von Maven zur Verfügung gestellter Lebenszyklus, der Site - Lifecycle genutzt. Er ist dafür verantwortlich, die in der test-Phase gespeicherten Informationen in gewünschte Dateiformate zu schreiben. Beispielweise werden in dem Site - Lifecycle die Informationen über Testabdeckung und Testerfolg in xml-Dateien gespeichert. Diese Dateien werden im letzten Schritt der Visualisierungspipeline in Crococosmo eingelesen. Dort werden die Daten von Sensoren analysiert und gefiltert. Sie werden in visualisierbare Daten transformiert, ehe sie letztendlich anhand eines Towers für den Betrachter grafisch zugänglich gemacht werden.

5.2 Werkzeuge der Visualisierungspipeline

In der vorher beschriebenen Visualisierungspipeline werden drei Werkzeuge erwähnt, die im Folgenden näher erläutert werden sollen. Es handelt sich dabei um:

- JUnit, das Werkzeug zur Testentwicklung
- Maven, das Werkzeug zur Testauswertung
- Crococosmo, das Werkzeug zur Testvisualisierung.

5.2.1 JUnit

JUnit ist ein von Kent Beck und Erich Gamma entwickeltes Framework zum Testen von Java Programmen. JUnit arbeitet auf Grundlage von Unit-Tests mit deren Hilfe Klassen getestet werden können. Im Allgemeinen werden Programme in einzelne Komponenten (engl. Units)

[8] Vgl. Kapitel 3.3 Testautomatisierung

unterteilt. Jede Komponente besitzt eine oder mehrere Funktionen. Mittels eines Unit-Tests kann nun eine dieser Komponenten getestet werden. In JUnit bezieht sich der Begriff Komponente auf eine Java-Klasse. Solche Unit-Tests kennen nur 2 Ergebnisse, entweder sie fallen positiv aus oder negativ.

JUnit soll im Allgemeinen das Verfahren des „Test Driven Development“ [8] zu Deutsch „Testgetriebene Entwicklung“ fördern, d.h. am Anfang produziert der Programmierer gerade so viel Code, dass dazu ein Test existieren kann. Dieser Code wird dann geprüft. Sind alle Tests positiv verlaufen, kann der Programmierer weiterhin Code implementieren. Danach wird erneut getestet. Sollten dann Fehler auftreten, weiß der Programmierer sofort wo der Fehler liegt, d.h. kurz gesagt fehlerarmen Code zu implementieren, indem jeder Schritt getestet wird. Am besten ist es während der Programmierung Tests zu schreiben, sonst besteht Gefahr Testfälle zu übersehen. In der folgenden Darstellung wird der innere Aufbau des JUnit-Werkzeugs ersichtlich.

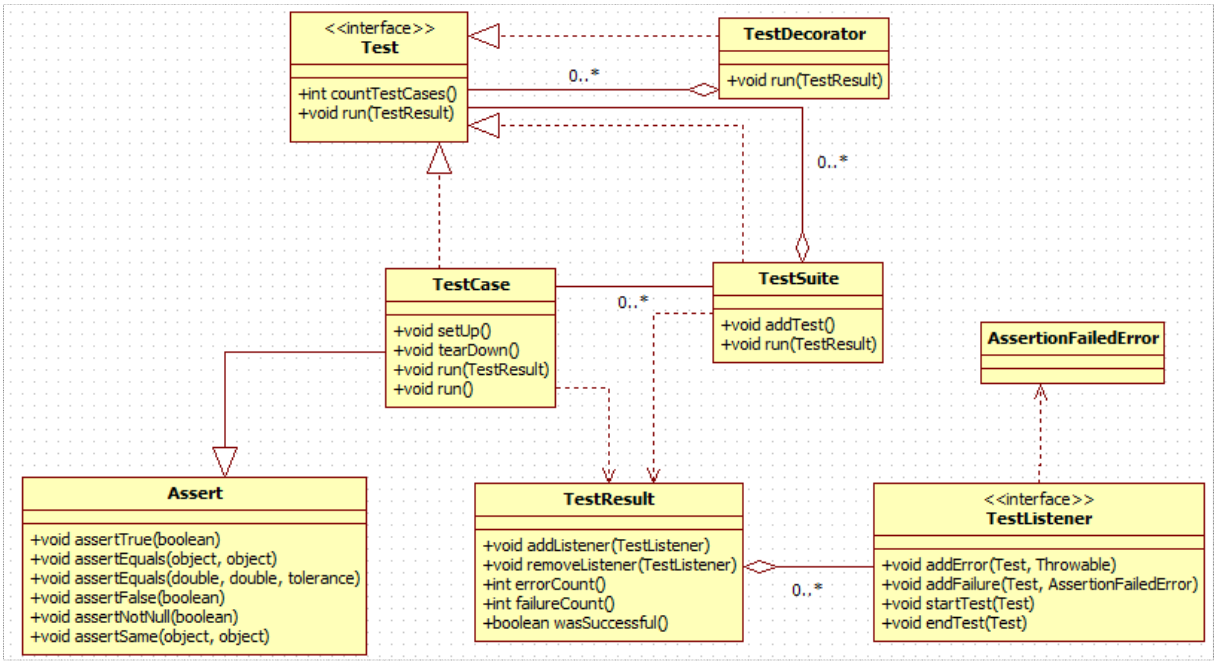

Abb. 1: Klassendiagramm des Tools [9]

Die Schnittstelle Test wird von TestCase oder TestSuite implementiert und bezeichnet damit unseren erstellten Test für die Klasse. Eine TestSuite kann aus mehreren TestCases bestehen. Ein TestCase setzt sich zusammen aus den Methoden setUp(), run() und tearDown(). SetUp() wird vor dem eigentlich Test durchgeführt und wird meist zur Wertzuweisung von Variablen genutzt. Danach folgt der eigentliche Test, der hier mit run() beschrieben ist. TearDown() wiederrum wird von JUnit zum „Aufräumen“ genutzt. Werte von Arrays oder Variablen können hier wieder zurückgesetzt werden. Ein TestCase erbt von der Assert-Klasse, die es

ermöglicht bestimmte Behauptungen aufzustellen und somit tatsächliche Werte mit erwarteten zu vergleichen. Sehr oft werden dabei AssertTrue() und AssertFalse() verwendet. Mit Hilfe der Funktion AssertTrue() wird angenommen, dass die aufgestellte Behauptung einen positiven Ausgang erhält. Im Gegensatz dazu wird mit AssertFalse() der aufgestellten Behauptung ein negativer Ausgang zugewiesen. Ein TestCase oder eine TestSuite liefern ein TestResult. Diese Klasse registriert wiederum wie viele Fehler bei Ausführung der Test-Klasse aufgetreten sind. Das ist auf die AssertionFailedError-Klasse zurückzuführen, sie wird nur aktiv, wenn eine Behauptung fehlschlägt. Die Klasse AssertionFailedError gehört zum TestListener. Dieser ist dafür verantwortlich, dass ein Test gestartet und auch wieder beendet wird**.** Ein Testdecorator wiederum ist für das Verhalten vor und nach einem gesamten Test zuständig. Das bedeutet in einem TestDecorator kann ein einmaliges Verhalten vor der Ausführung eines Tests und nach der Ausführung eines Tests festgelegt werden.

5.2.1.1 Typischer Aufbau einer JUnit-Klasse

```
import junit.framework.TestSuite;
import org.junit.*;

public class test extends TestSuite{
          @BeforeClass
          public static void setUpBeforeClass() throws Exception {
          }
          @Before
          public void setUp() throws Exception {
          }
          @Test
          public void testX () {
          }
          @After
          public void tearDown() throws Exception {
          }
          @AfterClass
          public static void tearDownAfterClass() throws Exception {
          }
}
```

Methoden, die vor dem Test ausgeführt werden.

Der eigentliche Test

Methoden, die nach dem Test ausgeführt werden.

5.2.1.2 Annotationen

Annotation bezeichnen in JUnit Anmerkungen vor den Methoden, somit wird vermieden, dass Testklassen von Oberklassen abgeleitet werden müssen und Namenskonventionen in Bezug auf Methodennamen eingehalten werden müssen. Diese Annotationen gibt es erst seit JUnit 4.0 und wurden aufgrund von immer mehr genutzten Metaprogrammierung in Java geschaffen. Im Allgemeinen gibt es sechs unterschiedliche Annotationen `@BeforeClass`, `@Before`, `@Test`, `@After`, `@AfterClass`, `@Ignore`. Die Annotation

`@BeforeClass` wird dabei meist zuerst in der Testklasse implementiert. Sie läuft nur einmal zu Beginn der Testausführung und eignet sich somit bestens für bspw. einen Log-In an einer Datenbank oder einer Initialisierung eines Parameters, der für alle Testfälle der Klasse benötigt wird. Folgt man der Reihenfolge wird nun meist die `@Before` Methode implementiert. Diese wird vor jedem Testfall der TestSuite ausgeführt. Meist wird sie zur Initialisierung von Variablen benutzt, die für den folgenden Test benötigt werden. Hat man mehrere mit `@Test` gekennzeichnete Methoden, so wird `@Before` für jede Test-Methode ausgeführt. Die eigentlichen Tests werden mit `@Test` für JUnit erkenntlich gemacht. Danach kann man von einer mit `@After` gekennzeichneten Methode alle Variablen falls nötig wieder auf einen bestimmten Wert zurücksetzen. `@After` Methoden werden auch ausgeführt, wenn `@Before` Methoden fehlgeschlagen sind. Alle mit der Annotation `@After` versehenden Methoden werden pro Testmethode ausgeführt. JUnit erlaubt es hier praktisch nach dem Test „aufzuräumen“, um noch andere Tests mit den gleichen Variablen durchführen zu können. Die Annotation `@AfterClass` ermöglicht ein gründlicheres „aufräumen“. An dieser Stelle kann bspw. ein Log-Out an einer Datenbank erfolgen. Die mit `@AfterClass` gekennzeichneten Methoden werden erst ausgeführt, wenn alle mit der Annotation `@Test` versehenden Tests ausgeführt wurden. Wirft die `@BeforeClass` Methode einen Exception, wird die `@AfterClass` Methode trotzdem ausgeführt. Die Annotation `@Ignore` besagt, dass diese Methode für die Ausführung uninteressant ist und ignoriert werden kann.

5.2.1.3 Assertions

Mit Hilfe von JUnit lassen sich verschiedene Behauptungen(engl. Assertions) aufstellen. Sie sind der wichtigste Bestandteil eines Unit-Tests und geben dem Programmierer Auskunft über den Testverlauf und zeigen ihm somit, an welcher Stelle das Programm noch einmal überabreitet werden muss. JUnit stellt dort schon verschiedene Methoden der Klasse Assert zur Verfügung. Die wichtigsten und meist genutzten Methoden sind dabei AssertTrue() und AssertFalse(). Mit Hilfe der beiden Funktionen lässt sich die Korrektheit der Behauptung überprüfen.

Bsp.:

```
int num = 0;
public void test1() {
     assertTrue(num==6);
};
public void test2() {
     assertFalse(num==6);
};
```

In diesem Beispiel wird anfangs die Integer-Variable `num` mit dem Wert 0 initialisiert. Die beiden Testmethoden überprüfen die Behauptung `num==6` auf Korrektheit. `Test1()` schlägt hier fehl, da angenommen wird, dass die Behauptung `num==6` wahr ist. An dieser Stelle werden nun der `TestListener` und damit die `AssertionFailedError`-Klasse aktiv. Sie liefert einen Failure zurück, verkünden also, dass ein Test fehlgeschlagen ist und beenden diese Methode prompt. `Test2()` liefert im Gegensatz dazu ein positives Testergebnis, denn an dieser Stelle ist man davon ausgegangen, dass die Behauptung `num==6` falsch ist. Zusätzlich zu diesen beiden Methoden liefert JUnit noch eine hilfreiche Funktion, `AssertEquals()`. Im Allgemeinen lässt sich mit `AssertEquals()` der Inhalt zweier Objekte auf Gleichheit überprüfen.

Bsp.:

```
int[] vec = new int[3];
vec[0] = 1;
vec[1] = 2;
vec[2] = 3;

String s1 = new String("Hallo");
String s2 = "Hallo";

public void test3(){
      assertEquals(3, vec.length);    (1)
      assertEquals(s1, s2);      (2)
};
```

In diesem Beispiel wird ein Integer Array mit drei freien Stellen erzeugt. Diese werden mit den Werten 1, 2 und 3 aufgefüllt. In der Methode `test3()` wird nun überprüft, ob das Array auch tatsächlich 3 Stellen hat, denn mit `assertEquals()` wird der Inhalt zweier Objekte auf Gleichheit überprüft. Das erkennt man auch an der 2.ten `assertEquals()`-Methode, dort hat man zwei unterschiedliche Objekte mit gleichem Inhalt zur Verfügung. Dieser Testfall wird positiv ausgehen, da der Inhalt beider Objekte gleich ist. An erster Stelle in dem Aufruf der `assertEquals()` ist das erwartete Objekt zu nennen und an zweiter das Eigentliche. An dieser Stelle wird JUnit keinen Failure zurückgeben, denn das Feld ist wie erwartet drei Stellen lang. `AssertEquals()` lässt sich auch mit vielen anderen Datentypen durchführen, wie z.B. String, Integer, Long oder Double. Bei numerischen Werten liefert JUnit noch eine interessante Funktion. Es lässt hier nun auch ein Toleranz-Wert eintragen.

Bsp.:

```
int min= 1;
int max= 5;
public void test4(){
      assertEquals(min, max, 4);
}
```

In diesem Beispiel werden zwei Integer-Zahlen mit Werten initialisiert und auf ihre Differenz zueinander geprüft. In der Methode `test4()` ist zu erkennen, dass als erstes die beiden zu vergleichenden Werte der `assertEquals()`-Funktion übergeben werden und dann wiederrum der Toleranzbereich. Diese Methode liefert keinen Fehler zurück, denn die Differenz von 5 und 1 ist 4. Man hat hier ebenfalls die Möglichkeit eine Zahl größer als 4 als Toleranzbereich zu nutzen. Der Test würde dann immer noch keine Fehler zurück liefern. In diesem Beispiel sind nur Integer-Zahlen erläutert, `assertEquals()` funktioniert allerdings auch mit Gleitkomma-Zahlen. Es besteht die Möglichkeit `assertEquals()` auch auf Arrays anzuwenden, dafür liefert JUnit eine eigene Methode, `AssertArrayEquals()`. Sie besitzt prinzipiell die gleichen Eigenschaften und Funktionen wie die bereits beschriebene `AssertEquals()`-Methode, der einzige Unterschied ist, dass `AssertArrayEquals()` auf Basis von Arrays arbeitet. Im Gegensatz zu `AssertEquals()` steht die `AssertSame()`-Methode, sie gelingt, wenn ihre Parameter identisch sind. Die `AssertSame()`-Methode ähnelt dem Operator ==.

Bsp.:

```
String s1 = new String("Hallo");
String s2 = new String("Hallo");
int i1 = 17;
int i2 = 17;
int i3 = 18;
public void test5(){
     assertSame(s1, s1);   (1)
//   assertSame(s1, s2);   (2)
     assertSame(i1, i2);   (3)
//   assertSame(i2, i3);   (4)
}
```

In diesem Beispiel werden mittels `s1` und `s2` String Objekte erzeugt, die die Zeichenkette „Hallo“ enthalten. Man hat also hier zweimal die gleiche Zeichenkette, aber zwei unterschiedliche Objekte. Zusätzlich werden `i1` und `i2` jeweils mit dem Wert 17 initialisiert. In der Methode `test5()` werden vier unterschiedliche `assertSame()`-Methoden untersucht. In der ersten Methode wird `s1` mit `s1` verglichen. Dieser Test gelingt. In der zweiten Testmethode werden `s1` und `s2` miteinander verglichen. Dieser Test würde fehlschlagen, daher ist er hier auskommentiert, um ein Fortlaufen der `test5()`-Methode zu gewährleisten. Er schlägt aus dem Grund fehl, da `s1` und `s2` unterschiedliche Objekte ansprechen, die allerdings denselben Inhalt haben, also gleich sind, aber durch den unterschiedlichen Speicherort nicht identisch. In der dritten `assertSame()`-Methode lässt

sich erkennen, dass `i1` mit `i2` verglichen wird. Dieser Test wird gelingen, da `i1` und `i2` dasselbe Objekt ansprechen und den gleichen Inhalt haben. Sollte nun allerdings ein Fall wie in der vierten `assertSame()`-Methode vorkommen, wird dieser Testfall fehlschlagen. Die beiden Variablen sprechen zwar dasselbe Objekt an, besitzen aber eine unterschiedlichen Inhalt.

5.2.2 Maven

Maven ist ein Projektmanagement-Tool für die Entwicklung von Softwareprojekten. Es ist in der Lage dem Software-Entwickler das Kompilieren, Testen, Packen und Verteilen auf andere Rechner insofern zu erleichtern, dass möglichst viele Schritte davon automatisiert werden. Darüber hinaus verfügt Maven über mehrere definierte Lebenszyklen, die aus Phasen bestehen. Diese einzelnen Phasen spiegeln die verschiedenen Stadien eines Projektes dar. Der Benutzer kann aktiv in die Lebenszyklen eingreifen und sie nach eigenen Vorstellungen verändern. Dies geschieht im Project Object Model - der POM.xml – der für Maven wichtigsten Konfigurationsdatei eines Projektes. Dort werden Plug-Ins vermerkt, die den Buildzyklus erweitern oder zur Auswertung des Projektmanagements wichtig sind. Diese werden durch Maven von dem entfernten globalen Repository in das auf dem Rechner befindliche lokale Repository geladen und stehen dort allen Maven-Projekten zur Verfügung.

5.2.2.1 Maven's Grundsätze

Maven beruht auf vier Grundsätzen laut Bachmann [10]:

- Konvention vor Konfiguration,
- Wiederverwendbarkeit,
- Deklarative Ausführung und
- einheitliche Organisation der Abhängigkeiten

Diese Grundsätze sollen Zeit sparen und das Verständnis für die zu entwickelnde Software erleichtern.

Konvention vor Konfiguration

Maven stellt bestimmte Standard-Verhalten für Projekte bereit, welche im Superpom[9] vermerkt sind. Diese Standard-Verhalten sollen für alle Maven-Projekte gelten, sie können allerdings auch nach eigenem Belieben angepasst werden. So wird dem Softwareentwickler bspw. die grundlegende Entscheidung über den Aufbau der Verzeichnisstruktur abgenommen.

[9] Elterliche POM.xml von dem alle POM.xml - Dateien erben

Dabei sorgt Maven ohne weitere Anpassungen dafür, dass Projekte folgendermaßen aufgebaut sind.

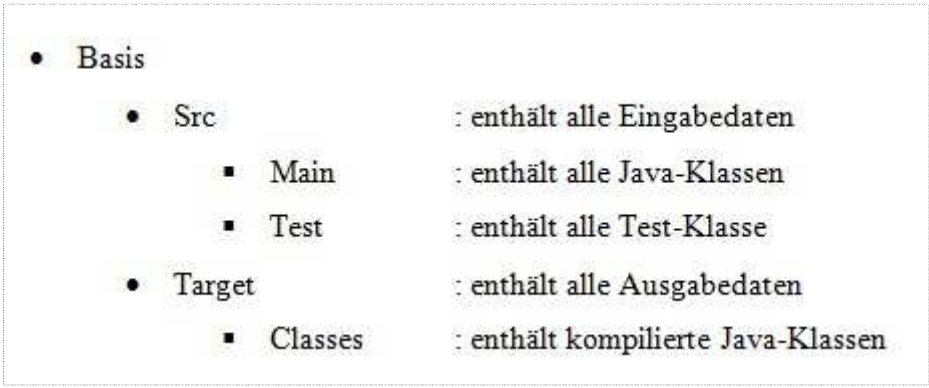

Abbildung 10: Maven's Standardverzeichnisstruktur

Maven erlaubt es ebenfalls eine eigene Verzeichnisstruktur zu entwickeln, allerdings müssen dann die Pfadnamen in der POM.xml beschrieben sein.

Ein weiteres standardisiertes Verhalten, welches durch Maven gepflegt wird, ist der Lebenszyklus. Maven stellt hier sicher, dass vordefinierte Phasen eines Builds in einer vorgegeben Reihenfolge ablaufen und bestimmte Plugin-Goals ausgeführt werden. Maven stellt drei Lebenszyklen zur Verfügung, die unterschiedliche Aufgaben erfüllen.

Zu den drei Lebenszyklen „Lifecycle“ gehören [22]:

- Build – Lifecycle
- Site – Lifecycle
- Clean - Lifecycle

Der Build – Lifecycle

Dieser Lebenszyklus ermöglicht es ein Projekt zu kompilieren, es zu testen und es zu verteilen. Der Buildzyklus stellt die Hauptfunktion von Maven dar und besteht aus folgenden acht Phasen [21].

1. Phase: Validate: Maven stellt hier die Struktur der POM.xml sicher. Es wird überprüft, ob jeder Starttag auch einen Endtag besitzt.

2. Phase: Compile: An dieser Stelle wird der Quellcode kompiliert. Meistens wird der javac-Compiler für Java-Quellcode verwendet.

3. Phase: Test: In dieser Phase wird der kompilierte Code getestet. Dies kann mit JUnit-Tests geschehen.

4. Phase: Package: Der kompilierte Code wird zur Weitergabe verpackt. Das Format dieses Packages wird in der POM.xml festgelegt. Meistens handelt es sich um .jar Dateien. Es besteht aber die Möglich auch .ear, .war oder maven-plugins auszuliefern.

5. Phase: Integration-Test: Das Softwarepaket wird in eine andere Umgebung geladen und dort auf Funktionsfähigkeit überprüft.
6. Phase: Verify: Es wird überprüft, ob das Softwarepaket eine gültige Struktur hat und Qualitätskriterien entspricht.
7. Phase: Install: Installiert das Softwarepaket auf dem lokalen Repository und steht somit anderen Projekten zur Weiterverwendung zur Verfügung.
8. Phase: Deploy: Eine Version der Software wird auf entferntem Maven Repository geladen und steht dort einer Community von Programmierern zur Verfügung.

Der Site-Lifecycle

Dieser Lebenszyklus wird genutzt, um verschiedene Projektdokumentationen zu erzeugen. Der Site – Lifecycle wird bspw. zur Erzeugung einer Projekt-Website genutzt oder zur langzeit Speicherung von Auswertungen von Softwaremetriken. Er besteht aus zwei Lebensphasen.

1. Phase: site: Erzeugt eine Projektdokumentation durch die Ausführung der unter dem Reporting-Abschnitt der POM.xml notierten Plug-Ins und deren Konfiguration.
2. Phase: site – deploy: Ist in der Lage die erstellte Dokumentation auf dem gewünschten Server zu laden und sie so anderen Entwicklern zur Verfügung zu stellen.

Der Clean – Lifecycle

Dieser Lebenszyklus wird zum Aufräumen nach dem Erstellen von Dokumentationen oder nach einem Build genutzt und besteht aus einer Phase.

1. Phase: clean: Räumt das Projekt auf und löscht dabei im Build erzeugte Dateien oder erstellte Dokumentationen.

Bei der Ausführung von Maven ist es zwingend notwendig eine dieser Phasen der drei Lebenszyklen anzugeben. Jede Phase eines Lebenszyklus impliziert das Ausführen der vorherigen Phasen.

Wiederverwendbarkeit:

Es gibt Plug-Ins die verschiedene Aufgaben übernehmen, bspw. das Kompilieren von Quellcode oder die Ausführung von Tests. In jeder Phase des Lebenszyklus werden bestimmte Plug-Ins benötigt, um spezielle Build-Aufgaben zu übernehmen. Diese Plug-Ins werden in der POM.xml vermerkt und durch selbige gesteuert. Beim Ausführen von Maven werden dann alle Plug-Ins, die in der POM.xml vermerkt sind, vom globalen Maven-Repository heruntergeladen und in dem auf dem Rechner befindlichen lokalen Maven-Repository gespeichert. Ist dort bereits ein Plug-In vorhanden was nun in einem neuen Projekt verwendet werden soll, muss es nicht erneut heruntergeladen werden, sondern kann gleich aus

dem lokalen Repository genutzt werden. Das liefert im Hinblick auf die Plug-Ins eine hohe Wiederverwendbarkeit.

Deklarative Ausführung

Maven's wichtigstes Element ist die POM.xml. Sie ist die Steuer- und Konfigurationsdatei in der genau beschrieben ist wann welches Plug-In auf welche Art und Weise aktiv werden soll. Kurz gesagt, „Ohne das POM wüsste Maven nicht, was es zu tun hat" [10] Deklarative Ausführung bedeutet in dem Fall, dass Plug-Ins in der POM genau mit Herkunft, Abhängigkeiten und Ausführungszeitpunkt erfasst sein müssen, um ausgeführt zu werden.

Einheitliche Organisation von Abhängigkeiten

Die Organisation der Abhängigkeiten ist eine der „größten Stärken" [10] von Maven, denn Maven übernimmt das Auflösen von transitiven Abhängigkeiten ganz von allein. So müssen nicht mehr die Abhängigkeiten der eigentlichen Abhängigkeit aufgelistet werden. Dies erleichtert dem Anwender die Benutzung und mindert den Aufwand.

5.2.2.2 Maven's wichtigster Bestandteil – POM.xml

POM ist eine Abkürzung für Project Object Model und liefert Maven alle wichtigen Steuerdaten und Beschreibungen des Projekts. Hier können bspw. zu benutzende Plug-Ins eingetragen werden, Abhängigkeiten vermerkt werden oder Profile aktiviert werden. Jede POM.xml ist von Maven's Superpom abgeleitet, welches wichtige Voreinstellung enthält, die bei jedem Projekt getroffen werden müssen bspw. Name der Repositories oder Aufbau der Verzeichnisstruktur. Die wichtigsten Bereiche einer POM.xml sind die folgenden: Koordinaten, Profile, Dependencies, Buildzyklus und Reporting. Diese Bereiche sind in folgender Beispiel-POM zu erkennen.

```
<project>
        <modelVersion>4.0.0</modelVersion>
        <groupId></groupId>
        <artifactId></artifactId>
        <packaging></packaging>
        <version></version>
        <name></name>
        <url></url>
        <profiles>
                <profile>
                </profile>
        </profiles>
        <dependencies>
                 <dependency>
                        <groupId></groupId>
                        <artifactId></artifactId>
                        <version></version>
                        <scope></scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId></groupId>
                                <artifactId></artifactId>
                                <version></version>
                        </plugin>
                </plugins>
        </build>
        <reporting>
                <plugins>
                        <plugin>
                                <groupId></groupId>
                                <artifactId></artifactId>
                                <version></version>
                        </plugin>
                </plugins>
        </reporting>
</project>
```

Koordinaten

Profil

Dependecies - Abhängigkeiten

Buildzyklus

Berichte

Die Koordinaten

Der erste Bereich einer POM.xml sind die Koordinaten, sie bezeichnen fünf identifizierende Informationen zum Artefakt, wie GroupId, ArtifactId, Version, Packaging und Classifier. Die GroupId beschreibt dabei eine Gruppierungsbezeichnung des jeweiligen Projektes, ähnlich einem Package-Namen. Meistens wird hier der umgekehrte Domainname der Firma verwendet bspw. org.firma.projekt. Innerhalb des mit der GroupId bezeichneten Verzeichnisses sind Artefakte zu finden. Sie sind die eigentlichen Quellcodedateien und

werden in Maven mittels einer ArtifactId identifiziert. Die einzelnen Artefakte können in unterschiedlichen Versionen vorliegen, daher muss zu jedem Artefakt eine Versionsnummer angegeben werden. Sie wird meist in diesem Format angegeben <Major>.<Minor>.<Bugfix>-<Qualifier>-<Buildnumber> bspw. 3.5.2-SNAPSHOT. Der Qualifier SNAPSHOT wird benutzt, um anzuzeigen, dass sich das Projekt noch in der Entwicklungsphase befindet. Eine weitere allerdings optionale Koordinate ist Classifier. Sie erlaubt es Artefakte zu unterscheiden, die aus dem gleichen POM gebaut wurden, aber einen unterschiedlichen Inhalt besitzen. Zu jedem Artefakt muss zusätzlich das Packaging angegeben werden. Es bezieht sich auf das letztendliche Format der Anwendung. Mit Maven können folgende Dateien erzeugt werden: .jar, .war, .ear , .pom oder maven-plugin.

Die Profile

Manchmal ist es notwendig einen Teil der Einstellung des Builds konfigurierbar zu halten. Dort kommen Profile ins Spiel. Sie ermöglichen es Einstellungen an eine bestimmte Bedingung zu knüpfen, das können beispielsweise bestimmte Systemvoraussetzungen sein. Somit wird einem Programm eine Plattformunabhängigkeit und Portabilität gewährleistet. Um diese Profile zu aktivieren gibt es verschiedene Möglichkeiten. Die erste Möglichkeit ist der explizite Aufruf über die Command-Line.

Bsp. :
```
mvn groupId:artifactId:goal -P profile-1.
```

Ein Eintrag in die Maven settings.xml ermöglicht ebenfalls eine Aktivierung von Profilen.

Bsp. :
```
<settings>
        <activeProfiles>
                <activeProfile>profile-1</activeProfile>
        </activeProfiles>
</settings>
```

Eine weitere Möglichkeit Profile zu aktivieren kann direkt in der POM.xml vorgenommen werden. Hier kann sogar ein Trigger - ein Auslöser - festgelegt werden.

Bsp. :
```
<profiles>
        <profile>
                <activation>
                        <jdk>1.4</jdk>
                </activation>
        </profile>
</profiles>
```

In dem obigen Beispiel lässt sich erkennen, dass das Profil nur aktiviert wird, wenn beispielsweise ein JDK mit der Versionsnummer 1.4 beginnt. Wie man erkennen kann sind Profile durchaus nützlich und geben dem Programmierer die Möglichkeit ihr System unter

unterschiedlichen Bedingungen und in unterschiedlichsten Umgebungen zum Laufen zu bringen.

Die Dependencies

Die Dependencies beschreiben die Abhängigkeiten, die von einem Projekt zu anderen Projekten bestehen. Ein Projekt ist somit in der Lage Libraries oder Artefakte anderer Projekte zu benutzen. Werden beispielsweise zum Testen des Projekts JUnit Tests verwendet, so muss als Dependency JUnit vermerkt werden.

Bsp.:

```
<dependencies>
    <dependency>
        <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.8.1</version>
            <scope>test</scope>
    </dependency>
</dependencies>
```

In einer Abhängigkeit müssen auf dieselbe Art und Weise Koordinaten angegeben werden, wie am Anfang eines jeden POMs. So wird also eine GroupId, eine ArtifactId und eine Versionsnummer benötigt, um die Abhängigkeit genau identifizieren und auflösen zu können. Maven lokalisiert anhand der Koordinaten das benötigte Projekt und lädt es in das lokale Repository. Hierbei kommt die positive Eigenschaft von Maven zum Vorschein. Maven ist in der Lage transitive Abhängigkeiten aufzulösen. Das ist nur möglich, da zu jedem Projekt eine POM.xml existiert und diese auch von Maven heruntergeladen wird. So können Schritt für Schritt alle Abhängigkeiten aufgelöst werden. Maven erlaubt es zusätzlich noch anzugeben, wann eine Dependency benutzt werden soll. Dies wird mit dem Tag `<scope>` bewerkstelligt. Maven unterscheidet hier zwischen `compile`, `provided`, `runtime`, `test` und `system`. `Compile` ist dabei als Default-Einstellung vermerkt und besagt, dass die Abhängigkeiten in allen Schritten zur Verfügung stehen. `Provided` ähnelt dem `Compile` sehr, geht aber davon aus, dass die Library, der Container oder das JDK zur Laufzeit zur Verfügung gestellt wird. `Runtime` ist das genau Gegenteil zu `Provided`, es wird angenommen, dass die Abhängigkeit nicht etwa zur Kompilierung gebraucht wird, sondern bei der Ausführung. Test bedeutet hingegen, dass die Abhängigkeit nur in der Testphase des Projekts benötigt wird. `System` setzt voraus, dass sich die ausführbare JAR-Datei der Dependency auf dem Rechner befindet und nicht erst heruntergeladen werden muss. Weiterhin können bestimmte Abhängigkeiten des Artefakts mittels `<exclusions>`

ausgeschlossen werden, um einem durch inkompatible Versionen verursachtem Problem vorzubeugen.

Der Buildzyklus

Der Buildzyklus der POM.xml bezieht sich auf die verschiedenen Lifecycle-Phasen eines Maven-Projekts. Hier kann bestimmt werden, welche Plug-Ins unter welcher Konfiguration während des Builds genutzt werden sollen. Zusätzlich ist es noch möglich anzugeben in welcher Lifecycle-Phase das Plug-In aktiv werden soll. Außerdem kann hier die Verzeichnisstruktur angepasst werden, bestimmte Build-Schritte geändert werden und Ressourcen hinzugefügt werden.

Bsp.:
```
<build>
        <sourceDirectory>src/main/java </sourceDirectory>
        <testSourceDirectory>src/test/java </testSourceDirectory>
        <plugins>
              <plugin>
                    <groupId>org.apache.maven.plugins</groupId>
                    <artifactId>maven-surefire-plugin</artifactId>
                    <version>2.4.3</version>
                    <configuration>
                    ...
                    </configuration>
              </plugin>
        </plugins>
  </build>
```

Eigenschaften von Plug-Ins, die im Buildzyklus angegeben werden können, wären bspw. `phase`, `goals`, `dependencies` und `extensions`. Soll das Plug-In nur in einer bestimmten Phase des Lifecycle ausgeführt werden, so wird ein Eintrag in dem Tag `<phase>` nötig. Typische Phasen wären beispielsweise `install` oder `test`. Diese Angabe muss nicht unbedingt im POM vorhanden sein, sie kann auch direkt in den Java-Klassen als Annotation vermerkt werden. Entscheidet man sich gegen `<phase>` so ist das Goal über Voreinstellungen des Plug-Ins gebunden.

Das Reporting

Das Reporting eignet sich um Dokumentationen zu dem Projekt zu erzeugen und benutzt den Site - Lifecyle. Im Allgemeinen fertigt der Befehl mvn site eine Übersicht in HTML-Form übers Projekt an. Dort enthalten sind dann die Namen der Entwickler, die Dependencies des Projekts, eine Übersicht über gebrauchte Plug-Ins und deren Versionen und benötigte Lizenzen. Diese ganzen Merkmale des Projekts werden in übersichtlichen html -wahlweise auch in xml- Dateien von Maven ausgeliefert. Benutzt man Reporting Plugins, erhält man

einen noch besseren Überblick über das Projekt. Zum Beispiel können hier mit Hilfe des Surefire-report-plugins die Erfolgsrate von Unit-Tests berechnet werden oder mit Hilfe des Cobertura Plug-Ins Test-Coverage Berichte angefertigt werden.

Bsp.:

```
<reporting>
      <plugins>
            <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-surefire-plugin</artifactId>
            <version>2.5</version>
            </plugin>
            ...
      </plugins>
</reporting>
```

5.2.2.3 Maven-Plug-Ins

Maven ist in der Lage, durch bestimmte Funktionalitäten erweitert zu werden. An dieser Stelle kommen Plug-Ins zum Einsatz, die in der POM.xml vermerkt werden. Plug-Ins können beim Buildzyklus von Nutzen sein, aber auch beim Reporting eingesetzt werden.

Build-Plug-Ins

Build-Plug-Ins werden während des Buildzyklus gebraucht. Die meisten Plug-Ins sind schon an eine Lebenszyklusphase gebunden und im Superpom vermerkt, benötigen daher meist keinen direkten Aufruf des Goals. Es reicht also vollkommen nur die Lebenszyklusphase anzugeben. Beispiele dieser Plug-Ins sind maven-clean-plugin, maven-compiler-plugin, maven-install-plugin.

Reporting-Plugins

Maven kann Berichte zu einem Projekt erzeugen, damit kann bspw. die Qualität von auf Java basierendem Quellcode überwacht werden, Dokumentationen erzeugt werden, Tests ausgewertet und Testabdeckungen berechnet werden. Hierfür sind bestimmte Plug-Ins von Nöten. Beispiele dieser Plug-Ins sind das maven-pmd-plugin, cobertura-maven-plugin und das surefire-report-plugin. Das Cobertura und das Surefire-Plug-In werden näher beschrieben, da sie in dieser Bachelorarbeit zur Testauswertung verwendet werden.

Cobertura-maven-Plugin

Hauptaufgabe von Cobertura ist es zu bestimmen, wie viele Zeilen des Codes durch Unit-Tests abgedeckt sind. Daraus wird dann ein Prozentwert ermittelt, die sogenannt Line Coverage. Cobertura ist auch in der Lage, die Zeilen anzuzeigen, die nicht von dem Unit Test durchlaufen werden (rot dargestellt und mit einer 0 gekennzeichnet). Im Allgemeinen arbeitet

Cobertura sowie viele andere Coverage Tools mit Instrumentierungen, d.h. der Quellcode wird auf Bytecodeebene mit Zusatzinformationen angereichert, um das Verhalten des Programms zu untersuchen bspw. wird das Betreten einer Methode oder einer if Anweisung dokumentiert und somit wiederum ein Zähler hochgezählt. Cobertura führt diese Instrumentierung zeilenweise durch und kann somit ohne Probleme feststellen, welche Zeilen vom Test nicht durchlaufen werden. Cobertura liefert seine Ergebnisse wahlweise als html oder als xml Datei aus, dies wird in der POM.xml des jeweiligen Projektes festgelegt.

Surefire-report-Plugin:

Während das Surefire-Plug-In für die Ausführung der Tests verantwortlich ist, gibt es ein ebenfalls zur Surefire-Gruppe gehörendes Plug-In, welche das Erstellen von Berichten über die gesammelten Informationen des Surefire-Plug-Ins als Aufgabe besitzt. Dies ist das Surefire-report-plugin. Es hilft dabei die Erfolgsrate der Tests zu bestimmen, die sogenannte Successrate. Diese wird anhand der erfolgreich ausgewerteten Unit-Tests bestimmt und ein prozentualer Wert ermittelt.

5.2.3 Crococosmo

Bei Crococosmo handelt es sich um ein Werkzeug zur Visualisierung von Softwarestrukturen und -metriken. Crococosmo ist in der Lage diese in 2D oder 3D Darstellungen zu visualisieren. Dadurch bietet es umfangreiche Möglichkeiten sich ausgiebig über ein Projekt zu informieren, es zu verwalten und auch pflegen zu können. Crococosmo ist mit Hilfe des Software-Cockpits in der Lage komplex anmutende Softwareprojekte in leichter verständliche Stadtstrukturen zu wandeln und darzustellen. Diese Darstellung wird durch einen attribuierten Hierarchie-Graphen ermöglicht. So ist es möglich zu jedem Knoten mehrere Attribute zu speichern und diese wiederum zu visualisieren. Die Visualisierung wird dabei anhand eines Stadtbildes vorgenommen. Crococosmo stellt Packages als Straßen und zugehörige Klassen als Hochhäuser dar. Auf diese Hochhäuser können nun die Attribute wie bspw. Grad der Verknüpfungen von Klassen und Methoden, Anzahl der Komponenten innerhalb einer Klasse und Alter der Komponente in Form von Farbe, Höhe oder Form abgebildet [23]. Dem Benutzer stehen dabei zahlreiche Einstellungsmöglichkeiten zur Verfügung, um die Visualisierung nach seinen Interessen zu richten. Inhalt dieser Arbeit wiederum war es, Crococosmo um eine Funktionalität zu erweitern. Es sollte in der Lage sein, auch Testdaten verwalten zu können und somit die aufwändige Analyse der Testauswertungen zu verringern. Dafür musste es um zwei Sensoren und einen Tower erweitert werden. Diese werden in folgenden Kapiteln näher erklärt.

5.3 Sammlung der Testdaten

Bevor die Testdaten in Crococosmo eingelesen werden können, müssen sie gesammelt werden. Denn eine Hauptaufgabe dieser Bachelorarbeit ist es, die Entwicklung der Testdaten während der Entstehung eines Programmes verfolgen zu können und somit die testgetriebene Entwicklung[10] zu fördern. Das bedeutet es müssen die Testdaten zu verschiedenen Zeitpunkten erfasst werden. Daher wird ein Eintrag in den bereits vorhandenen Parse-Zyklus notwendig. Dabei wird das Projekt mittels einer for-Schleife zu verschiedenen Zeitpunkten ausgecheckt und mit Hilfe des Software-Cockpits geparst. An dieser Stelle wird eine Datei namens test.bat aufgerufen. Sie ist für die Erzeugung der Testauswertungsdaten zuständig. Diese Daten werden von Maven pro Revision erzeugt. Da sie aufgrund von Maven's Einstellungen immer gleich benannt werden, muss einer Überspeicherung vorgebeugt werden. Die test.bat-Datei ist dementsprechend auch für die Umbenennung nach und die Sammlung der Daten in einem Ordner. Sowohl die Datei als auch der Ordner werden nach der Revisionsnummer benannt. Bei diesem Zyklus ist zu beachten, dass pro ausgecheckter Revision auch die passende POM.xml vorhanden sein muss. Da sonst Fehler bei der Ausführung von Maven auftreten und so das Projekt nicht korrekt gebaut werden kann. Aus diesem Grund wird eine Änderung der POM.xml pro Revision notwendig. Der Benutzer wird im Parse-Zyklus aufgefordert dies zu tun.

5.3.1 Änderung der POM

Der bereits vorhandene Parse-Zyklus wurde durch eine Funktionalität erweitert. Es besteht nun die Möglichkeit der Testauswertung. Bevor das geschieht, müssen allerdings im Vorfeld einige Voraussetzungen erfüllt werden. In dem Parse-Zyklus wird der Benutzer nun aufgefordert die POM.xml des ausgecheckten Projekts zu ändern. Die POM.xml ist die Steuerdatei für Maven, das Werkzeug welches zur Testauswertung genutzt wird. Um sicherzustellen, dass auch genau die Dateien erstellt werden, die später mittels Crococosmo eingelesen werden können, muss die POM.xml um bestimmte Plug-Ins und somit auch Funktionalitäten erweitert werden. Liegt ein einfaches Projekt ohne erbende Unterprojekte vor, genügt es folgende Konfiguration in der POM.xml zu vermerken.

[10] Der Programmierer erstellt Komponenten und testet diese gleich im Nachhinein.

```
<project>
      <build>
                  ...
                  ...
      </build>
      <reporting>
        <plugins>
            <plugin>
                  <!-- Testabdeckung coverage.xml-->
                  <groupId>org.codehaus.mojo</groupId>
                  <artifactId>cobertura-maven-plugin</artifactId>
                  <version>2.4</version>
                  <configuration>
                        <format>xml</format>
                  </configuration>
            </plugin>
            <plugin>
                  <!-- Testerfolg TEST-*.xml -->
                  <groupId>org.apache.maven.plugins</groupId>
                  <artifactId>maven-surefire-report-plugin
                  </artifactId>
                  <version>2.5</version>
            </plugin>
        </plugins>
      </reporting>
</project>
```

Abbildung 11: Erweiterung der POM.xml

Wie in der vorherigen Abbildung zu erkennen, muss die POM.xml in dem Reporting-Bereich um zwei Plug-Ins erweitert werden. Ein Plug-In ist dabei das Cobertura-Plug-In, welches für die Sammlung der Informationen über die Testabdeckung zuständig ist und diese Daten anhand von einer Coverage.xml abspeichert. Bei diesem Plug-In ist eine Konfiguration vorzunehmen, da Cobertura in den Standard-Einstellungen auf die Erstellung einer HTML-Seite konfiguriert ist. Das XML-Format wird hierbei benötigt, um die Informationen der Testabdeckung im Endeffekt durch Crococosmo ermitteln zu können. Das zweite Plug-In, welches dringend in der POM.xml benötigt wird, um eine Auswertung der Testdaten vorzunehmen, ist das SureFire-Report-Plug-In. Es sammelt Informationen über den Testausgang von JUnit-Tests und erstellt eine TEST-*.xml.

Liegt das Projekt nun mit Unterprojekten, die von der POM.xml des Oberprojekts erben, vor, müssen die Plug-Ins auf eine andere Art in der .xml-Datei vermerkt werden.

```
ParentPOM.xml
<project>
....
<build>
      .....
      <pluginManagement>
            <plugins>
              <plugin>
                  <!-- Testabdeckung coverage.xml-->
                  <groupId>org.codehaus.mojo</groupId>
                  <artifactId>cobertura-maven-plugin</artifactId>
                  <version>2.4</version>
              </plugin>
            </plugins>
      </pluginManagement>
</build>
.....
```

Abbildung 12: Ausschnitt einer elterlichen POM.xml eines Multimodul-Projektes

In der vorherigen Abbildung ist ein Ausschnitt einer elterlichen POM.xml zu erkennen. Da von diesem Project Object Model alle Unterprojekte erben, muss die Konfiguration auf eine andere Weise geschehen. Das Cobertura Plug-In ist dafür verantwortlich den Code während der Kompilierung zu instrumentieren. Da dies allerdings nicht nur bei dem Oberprojekt geschehen soll, ist es von großer Wichtigkeit das Plug-In in die Sektion pluginManagement der POM.xml – Datei einzutragen. In dieser Sektion werden *„analog zum dependencyManagement, Plugins für die Verwendung in erbenden Projekten konfiguriert“*[11] Ist das Cobertura Plug-In also in der pluginManagement-Sektion vermerkt, kann davon ausgegangen werden, dass Unterprojekte auch durch eben dieses Plug-In instrumentiert werden. Da allerdings nun nur die Instrumentierung des elterlichen Projekts und der Kindprojekte abgedeckt sind, müssen diese Informationen noch anhand von XML - Dateien gespeichert werden. Dies sollte erneut sowohl in der elterlichen POM als auch in allen erbenden POMs geschehen, die gemeinsam mit dem Quellcode in einem Verzeichnis liegen. Dort ist dieselbe Konfiguration in der Reporting-Sektion der POM.xmls vorzunehmen, wie bei Projekten ohne Kindprojekte (vgl. Abbildung 11). So kann garantiert werden, dass die Testdaten korrekt ausgewertet werden und diese Informationen in dem richtigen Format abgespeichert werden.

5.4 Entwicklung der Sensoren

Um die gesammelten Daten über die Testauswertungen nun zu visualisieren, ist es notwendig sie in Crococosmo einzulesen. Das wird mittels Sensoren garantiert. Sie sind in Crococosmo

unter dem Menüpunkt File -> Import ->Analysis Data zu finden. Dort besteht dann die Möglichkeit verschiedene Sensoren zu starten und somit Dateien für bestimmte Softwaremetriken visuell auszuwerten. Die Sensoren, die sich zum Einlesen der Testdaten eignen, sind hierbei der TestDataSensor und der SureFireSensor. Sie erben von der Klasse AbstractSensor, die sich im Paket Sensors im IO-Paket von Crococosmo befindet und werden durch den SensorRunner gestartet.

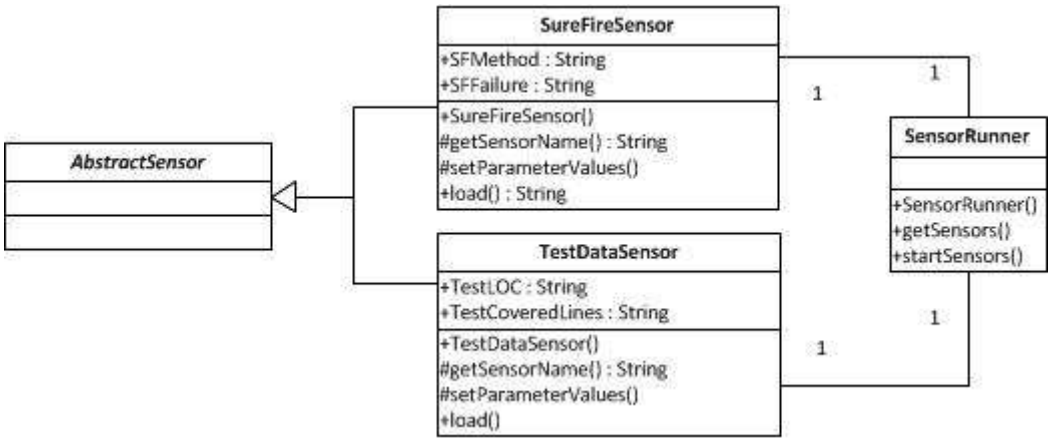

Abbildung 13: Klassendiagramm der Sensoren

Der TestDataSensor eignet sich dabei zum Einlesen der durch Cobertura erstellten Datei, die bspw. den Namen coverage_11.xml oder coveragenetbeans_11.xml und beinhaltet damit Informationen über die Testabdeckung. Der SurefireSensor wiederum eignet sich zum Einlesen der durch das SureFire-Plug-In erstellen Dateien, die bspw. den Namen org.openfast.ApplicationTypeDictionaryTest_11.xml tragen und sind damit verantwortlich für die Informationen über den Testerfolg.

5.4.1 Der TestDataSensor

Der TestDataSensor analysiert, die von Cobertura erstellten Dateien und filtert die Informationen, die letztendlich für das Visualisieren der Testabdeckung benötigt werden. Diese Informationen sind:

- die Anzahl der Zeilen der zu testenden Klassen und
- die Anzahl der abgedeckten Zeilen.

Eine von Cobertura erstellte Datei enthält umfangreiche Informationen über Struktur und die Testabdeckung des gesamten Projekts, der einzelnen Packages, jeder Klasse, jeder Methode und jeder Zeile.

```
<packages>
	<package name="org.openfast" line-rate="0.7829232995658466" branch-rate="0.7101910828025477" complexity="1.7964285714285715">
		<classes>
			<class name="org.openfast.ApplicationTypeDictionary" filename="org/openfast/ApplicationTypeDictionary.java"
				line-rate="1.0" branch-rate="0.6666666666666666" complexity="2.3333333333333335">
				<methods>
					<method name="&lt;init&gt;" signature="()V" line-rate="1.0" branch-rate="1.0">
						<lines>
							<line number="8" hits="136" branch="false"/>
							<line number="10" hits="136" branch="false"/>
						</lines>
					</method>
```

Abbildung 14: Auszug einer durch Cobertura erstellten coverage.xml-Datei

Aufgrund des gewählten Visualisierungskonzeptes werden nur die Informationen über den Abdeckungsgrad und die Anzahl der Zeilen der Klasse benötigt. In Abbildung 23 sind die Information über den Abdeckungsgrad im Tag `<class>` unter dem Attribut `line-rate` vermerkt. Weiterhin sind in Abbildung 23 noch die Tags `<line>` zu erkennen, deren Anzahl entspricht der gesamten Zeilenanzahl einer Klasse. Diese beiden Informationen werden durch den TestDataSensor pro Klasse gefiltert. Das geschieht mittels der bereits in Crococosmo implementierten Klasse `SensorXMLTools`, die es ermöglicht eine XML-Datei einzulesen und daraus bestimmte Inhalte zu erfassen. Diese Informationen müssen nun pro Klasse bzw. Knoten am Graphen vermerkt werden. Die Information über die Anzahl der Zeilen einer Klasse wird im Attribut `TESTLOC` vermerkt. Die Information über den Abdeckungsgrad liegt allerdings zurzeit noch in prozentualer Form vor. Mittels folgender Formel wird aus dem prozentualen Wert ein konkreter Wert für die Anzahl der abdeckten Zeilen ermittelt.

$$Abgedeckte\ Zeilen = Zeilenanzahl \cdot Abdeckungsgrad$$

Dieser Wert wird nun im Attribut `TestCoveredLines` pro Klasse gespeichert.

5.4.2 Der SureFireSensor

Die Klasse `SureFireSensor.java` analysiert, die vom SureFire-Plug-In erstellen XML-Dateien. Sie enthalten Informationen über den Testerfolg. Das Plug-In erstellt dabei pro Testklasse eine Datei mit umfangreichen Systeminformationen, wie zum Beispiel den Namen des Compiler oder des Betriebssystems. Viele Informationen in diesen Dateien eignen sich nicht zur Visualisierung und müssen daher gefiltert werden. Aufgrund des gewählten Visualisierungskonzeptes werden folgende zwei Informationen benötigt:

- Anzahl der Testmethoden
- Anzahl der fehlgeschlagenen Testmethoden

Absichtlich wurden die fehlgeschlagenen Testmethoden zur Visualisierung gewählt, da das Hauptaugenmerk beim Testen auf die Fehlerfindung und –analyse gelegt wird.

```
<?xml version="1.0" encoding="UTF-8" ?>
<testsuite failures="0" time="0.01" errors="0" skipped="0"
tests="1" name="org.openfast.ApplicationTypeDictionaryTest">
  <properties>
    <property name="java.runtime.name" value="Java(TM) SE
Runtime Environment"/>
    <property name="sun.boot.library.path" value="C:\Program
Files\Java\jdk1.6.0_22\jre\bin"/>
    <property name="java.vm.version" value="17.1-b03"/>
    <property name="java.vm.vendor" value="Sun Microsystems
Inc."/>
    <property name="path.separator" value=";"/>
    <property name="java.vm.name" value="Java HotSpot(TM) 64-
Bit Server VM"/>
    <property name="os.name" value="Windows 7"/>
      ...
 </properties>
  <testcase time="0.008"
classname="org.openfast.ApplicationTypeDictionaryTest"
name="testLookup"/>
</testsuite>
```

Abbildung 15: Auszug einer vom SureFire-Plug-In erstellten XML-Datei

In der obigen Abbildung ist ein Auszug aus einer XML-Datei zu erkennen, die durch das SureFire-Plug-In erstellt wurde. Als Attribute des Tags `<testsuite>` sind dabei alle wichtigen Daten zu der Testklasse enthalten. So lässt sich der Name der Testklasse ablesen, wie viele Testmethoden fehlgeschlagen sind (`failures`) oder nicht ausgeführt werden konnten (`errors`) und wie viele Testmethoden überhaupt vorliegen (`tests`). Unter dem Tag `<properties>` werden nun alle wichtigen Systemdaten aufgelistet, wie zum Beispiel der Name des Betriebssystems oder Informationen über das genutzte JDK und die Virtual Machine. In dieser Abbildung ist nur ein kleiner Teil dieser umfangreichen Daten zu erkennen. Unter dem Tag `<testcase>` sind nun alle Testmethoden noch einmal einzeln aufgeführt mit ihren wichtigsten Merkmalen.

Der SureFireSensor ist nun dafür zuständig die Fehleranzahl und die Anzahl der Testmethoden zu filtern. Dies geschieht erneut mit Hilfe der Klasse `SensorXMLTools`. Der Wert der Anzahl der Testmethoden wird im vorher bereits registrierten Attribut `SFMethod` am Graph gespeichert. Die Anzahl der fehlgeschlagenen Testmethoden beinhaltet nicht nur den Wert der Testfehler sondern auch die Tests, die nicht ausgeführt werden konnten. Diese Information wird im dem Attribut `SFFailure` gespeichert.

5.5 Visualisierungskonzept

Da nun die Daten mit Hilfe der Klasse `SensorsXMLTools` von Crococosmo eingelesen sind, erfolgt eine Auswertung der Metriken des Testerfolgs und der Testabdeckung. Um diese abstrakten Daten erfolgreich zu visualisieren und optisch zugänglich zu gestalten, wird ein folgendes Visualisierungskonzept genutzt. Im Allgemeinen ist die Visualisierung der Testabdeckung von der Visualisierung des Testerfolgs getrennt. Beide Metriken werden

anhand von unterschiedlichen Turmformen in Crococosmo visualisiert, damit sie unterscheidbar bleiben. Innerhalb dieser verschiedenen Formen wird anhand eines Füllstandes gezeigt zu viel Prozent die Klasse durch Tests abdeckt ist und der prozentuale Anteile der fehlschlagenden Testmethoden einer Testklasse aufgezeigt. Es wird also bei beiden Metriken ein Vergleich zwischen Soll und Ist-Wert angestellt. So werden bei Türmen der Testabdeckung die Anzahl der Zeilen der zu testenden Klasse als 100% angesehen und bis zu dem Stand der zur Zeit aktuell abgedeckten Zeilen mit einer Farbe gefüllt. Das heißt der Soll-Wert wäre hier eine komplette Einfärbung der Klasse, da so die Testabdeckung 100% betragen würde, und somit jede Zeile durch Tests ausgeführt werden würde. Bei dem Testerfolg wird ebenfalls das Prinzip des Füllstandes genutzt. Hierbei ist allerdings ein hoher Füllstand nicht wünschenswert, denn an dieser Stelle der Testfehler visualisiert wird. Die Höhe eines Testerfolgs bzw. Testfehler Towers ist auf die Anzahl der Testmethoden in der Testklasse zurückzuführen. Dieser Tower wird bis zu dem Stand gefüllt, der der Anzahl der fehlgeschlagenen Testmethoden einer Testklasse entspricht. So wäre hierbei also ein niedriger Füllstand anzustreben. Dieses Visualisierungskonzept soll im Folgenden an einem Beispiel der Java-Klasse mit dem Namen `X.java` erklärt werden.

Revision	Testabdeckung	Testfehler
3	0%	0 von 0
10	50%	4 von 5
100	75%	2 von 5
120	60%	4 von 5

Die Klasse `X.java` liegt in 4 verschiedenen Versionen vor. Das bedeutet, sie wurde zu vier verschiedenen Zeitpunkten durch das SoftwareCockpit geparst und von Maven einer Testdatenauswertung unterzogen. Somit liegen Testdaten für vier Versionen vor, die durch Crococosmo visualisiert werden können.

Bezieht man sich nun auf die erste durch das SoftwareCockpit geparste und durch Maven analysierte Version, ist zu erkennen, dass noch keine Testdaten für diese Klasse vorliegen. Die Testabdeckung liegt bei 0% .Sie würde wie folgt durch Crococosmo visualisiert.

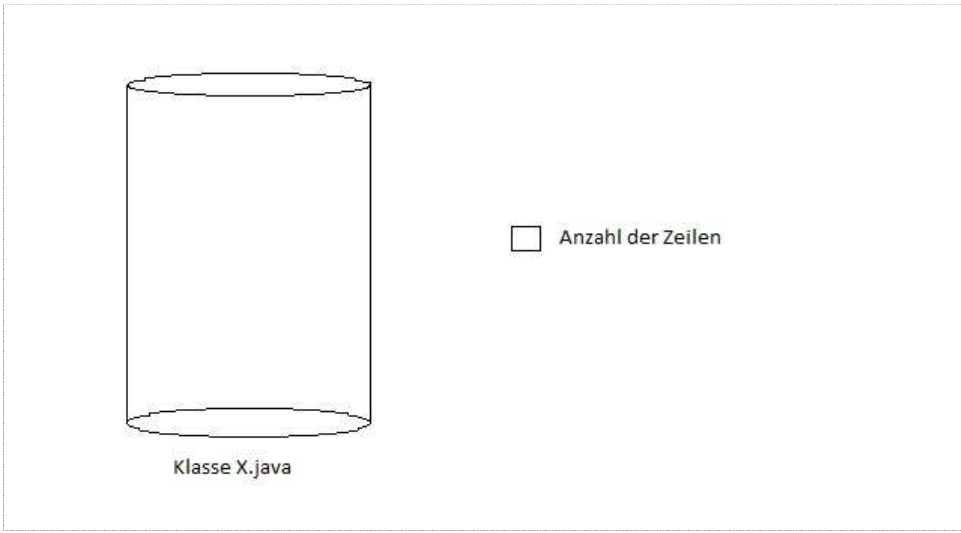

Abbildung 16: Visualisierung einer 0%-igen Testabdeckung

In dieser Abbildung erkennt man, dass der zylinderartige Turm für die zu testende Klasse X durchsichtig ist. Es wurden keine Tests für diese Klasse implementiert. Dementsprechend wäre auch eine Füllung des Testfehler-Towers der sich auf die Testklasse der Klasse X bezieht nicht möglich. Er würde folgendermaßen dargestellt werden.

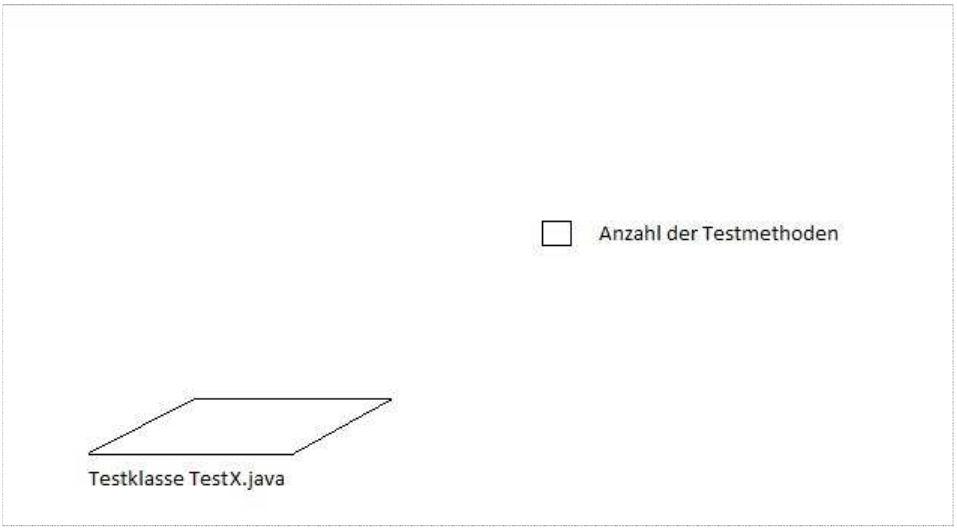

Abbildung 17: Visualisierung eines 0%-igen Testfehlers

Bei dieser Abbildung der Testklasse TestX.java erkennt man, dass sich die Form gegenüber der Testabdeckung zu einem Rechteck geändert hat und sich keine Höhe ausbildet. Das ist auf das Fehlen der Tests zurückzuführen. Sind keine Tests vorhanden, stehen dementsprechend keine Testmethoden zur Ausbildung der Höhe zur Verfügung. Somit bildet sich nur eine Grundfläche.

In Revision 10 beträgt die Testabdeckung der Klasse X 50%. Da im Vergleich zur Revision 3, also der vorhergehenden eingelesenen Version in Crococosmo, sich die Testabdeckung um 50% verbessert hat, kommt es zur Ausbildung des Füllstandes wie im Folgenden zu erkennen.

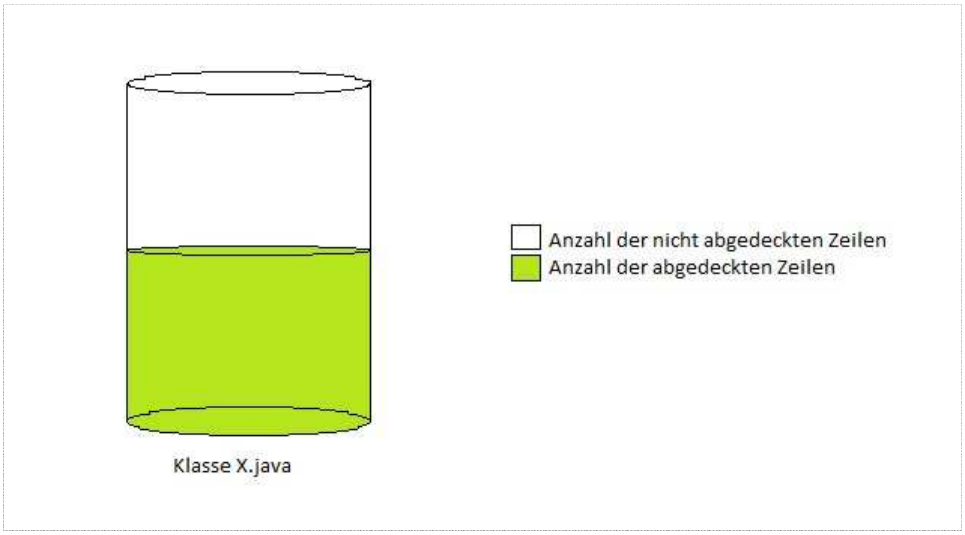

Abbildung 18: Visualisierung einer 50%-igen Testabdeckung

Die hellgrüne Färbung ist darauf zurückzuführen, dass die Testdaten im Vergleich zur vorherigen Version neu entstanden sind. So wurden also Tests neu entwickelt, die die Klasse `X` zu 50% ausführen. Die anderen 50% bedeuten in diesem Zusammenhang, dass die Klasse `X` weiterhin getestet werden müsste, um die Softwarequalität zu erhöhen und den eventuell bestehenden Testfehler einzugrenzen. Sieht man sich nun die Testklasse der Klasse `X` mit dem Namen `TestX.java` an, so erkennt man ebenfalls das Testmethoden entstanden sind. In der folgenden Darstellung ist dies am Füllstand, also an der Ausbildung der Höhe zu erkennen.

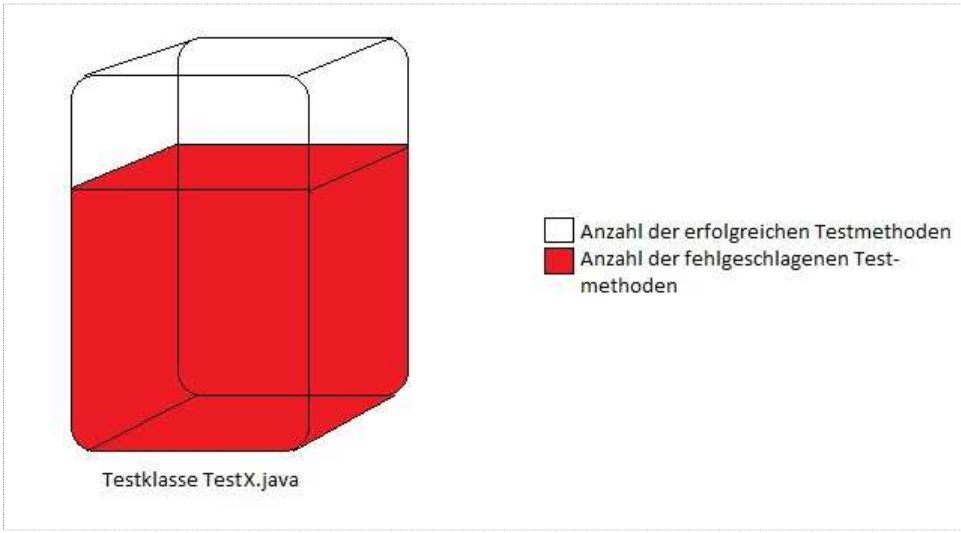

Abbildung 19:Visualisierung eines 80%-igen Testfehlers

In der Testklasse `TestX.java` sind fünf Testmethoden vorhanden. Vier von diesen Methoden schlagen fehl. Das heißt, dass der Füllstand bis zu 80% gefüllt sein muss. Die hellrote Färbung ist darauf zurückzuführen, dass in der vorherigen Revision noch keine Testfehler vorhanden waren. Neue Testfehler bekommen als Erkennungsmerkmal eine hellrote Farbe, um Aufmerksamkeit auf sich zu lenken.

In Revision 100 beträgt die Testabdeckung 75%. Sie ist also im Vergleich zur vorherigen Version um 25% gestiegen. Dies wird folgendermaßen visualisiert.

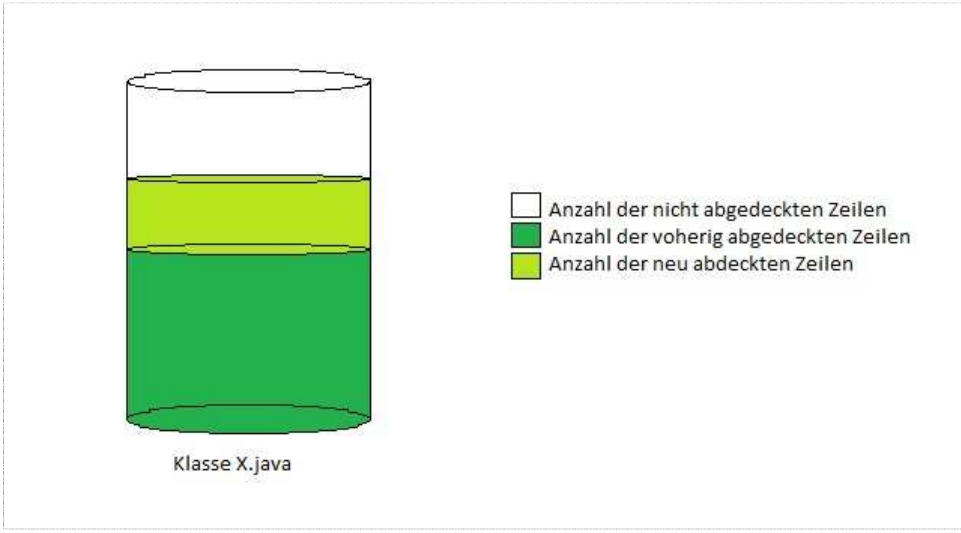

Abbildung 20:Visualisierung einer verbesserten Testabdeckung

Bei dieser Visualisierung erkennt man, dass der zylinderartige Tower mit zwei Farben gefüllt ist. Der dunkelgrün gefärbte Bereich zeigt an, zu wie viel Prozent die Klasse in der vorherigen Revision abgedeckt war. Der nun darauf befindliche hellgrüne Bereich zeigt an, dass sich die Testabdeckung im Vergleich zur Revision 10 verbessert hat und um wie viel Prozent der Abdeckungsgrad gestiegen ist. In diesem Fall sind es 25%. Der darauf gestapelte durchsichtige Bereich zeigt nun, dass die Testabdeckung der Klasse `X` noch nicht zu 100% erfolgt. Somit müssten mehr Tests für diese Klasse entwickelt werden. Sieht man sich nun die Testfehler der zugehörigen Testklasse `TestX.java` an, erkennt man eine unterschiedliche Färbung der Blöcke. In diesem Fall wurden Testfehler im Vergleich zur vorhergehenden Version beseitigt.

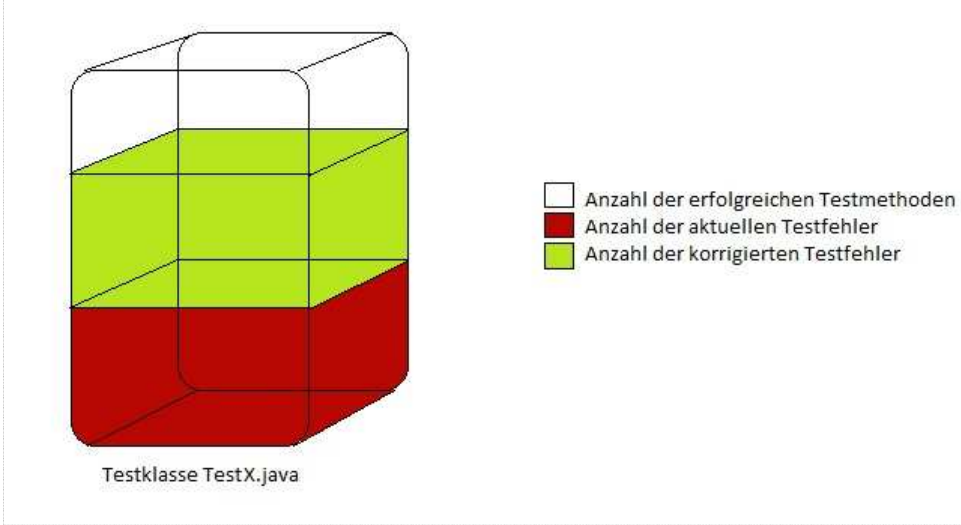

Abbildung 21: Visualisierung korrigierter Testfehler

Hierbei ist ersichtlich, dass der aktuelle Testfehler niedriger ist als der Testfehler der Revision 10. Das heißt, die zu testende Klasse `X.java` wurde an den Stellen korrigiert, an denen vorherige Testfehler bestanden. In dieser Revision schlagen nun zwei Testmethoden weniger fehl als in der vorhergehenden. Diese werden hellgrün dargestellt. Da nun aber immer noch Fehler bestehen, die allerdings in der vorherigen Version schon bestanden haben, wird dieser Bereich dunkelrot gefärbt. Der durchsichtige Bereich beschreibt dabei die Testmethoden, die weder in der vorherigen noch in der aktuellen Version fehlschlagen.
In Revision 120 sinkt der Abdeckungsgrad der Klasse `X` auf 60%. Das entspricht einer Verminderung um 15%. Diese wird wie folgt dargestellt.

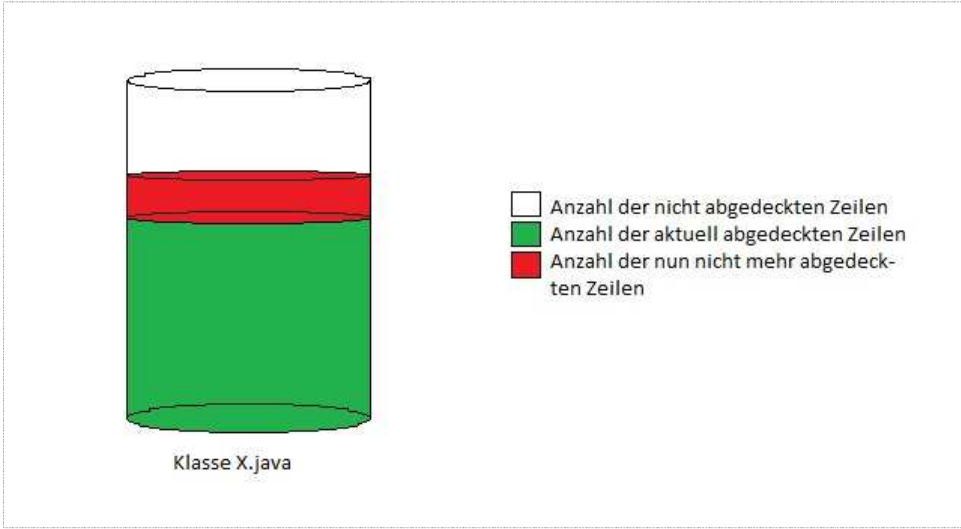

Abbildung 22: Visualisierung einer verminderten Testabdeckung

Bei dieser Darstellung ist eine Verminderung der abgedeckten Zeilen zu erkennen. Diese Verminderung kann entstehen, wenn Testmethoden entfernt werden, oder bestimmte vorher getestete Zeilen nun nicht mehr von Tests ausgeführt werden. Dieser Bereich wird hellrot dargestellt, um die Aufmerksamkeit des Betrachters auf diesen Bereich zu lenken. Der dunkelgrüne Bereich liefert Information über den aktuellen Abdeckungsgrad, der in dieser Darstellung 60% beträgt. Der durchsichtige Bereich gibt Auskunft über die Zeilen, die weder in der vorherigen Version noch in der aktuellen abgedeckt wurden. Sieht man sich nun den Testfehler-Tower dieser Version an, sehe die Visualisierung wie folgt aus.

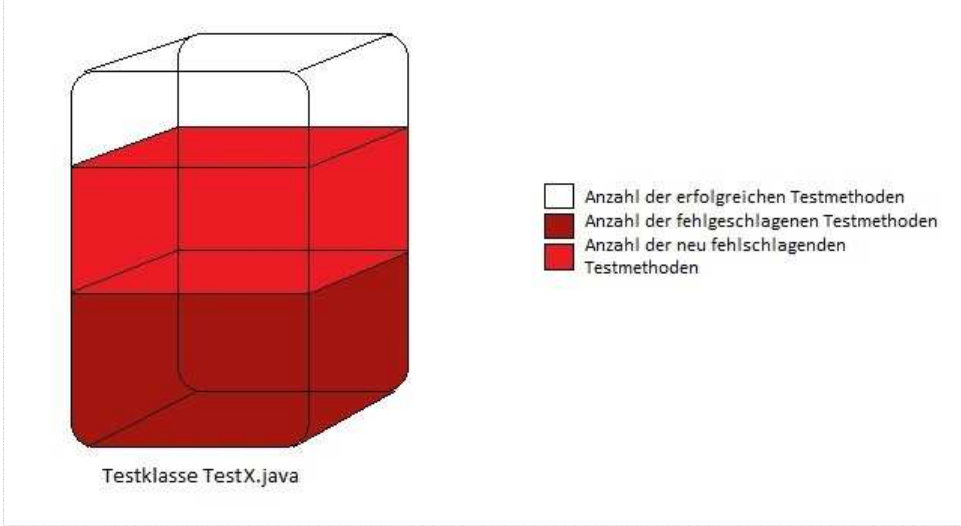

Abbildung 23:Visualisierung neuer Testfehler

In der vorhergehenden Revision 100 beschränkte sich der Testfehler auf zwei Testmethoden. Die Anzahl der fehlschlagenden Testmethoden erhöhte sich in der Revision 120 auf vier. Diese Erhöhung um zwei Testmethoden ist an dem hellrot gefärbten Bereich zu erkennen. Er zeigt an, wie viele Methoden im Vergleich zur vorherigen Version neu fehlschlagen. Der dunkelrote Bereich gibt Auskunft über den Testmethoden, die nicht nur in der aktuellen Version fehlschlagen sondern auch schon in der vorherigen. Der durchsichtige Bereich enthält die Information über die Anzahl der erfolgreich verlaufenen Testmethoden der `TestX.java`.

5.5 Einstellung in Crococosmo

Der Benutzer kann, die vorher eingelesenen Dateien, die von Maven erstellt worden sind, in angemessener Form visualisieren. Passend zu dem Repräsentationstyp `TestDataTower` steht eine Konfigurierung namens `TestData-TowerConfig` zur Verfügung. Dabei können der Testerfolg und die Testabdeckung sowohl getrennt von einander als auch gemeinsam in einer Darstellung ausgewertet werden. Hierfür stehen dem Benutzer vier Comboboxen zur Verfügung. Zwei von diesen sind für die Darstellung der Testabdeckung bzw. TestCoverage verantwortlich während die anderen zwei Comboboxen für die Darstellung des Testerfolgs bzw. TestSuccess zuständig sind.

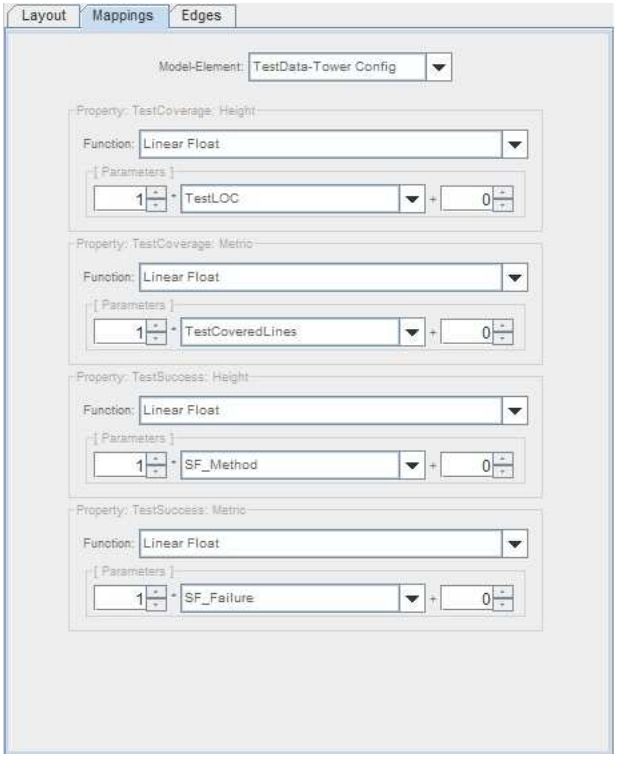

Abbildung 24: Konfiguration des Repräsentationstyps TestDataTower

Für die Darstellung des Testerfolgs und der Testabdeckung müssen jeweils zwei Attribute auf die Höhe des Tower und die Metrik des Towers abgebildet werden. Dies geschieht, um dem Benutzer einen direkten Vergleich zwischen Soll- und Ist-Werten ermöglichen zu können. Die Höhenmetrik der Testabdeckung (TestCoverage), wurde durch den `TestDataSensor` ermittelt und in das Attribut `TestLOC`[11] gespeichert. Auf diese Höhenmetrik wird nun die Metrik der abgedeckten Zeilen abgebildet. Dieses Attribut heißt `TestCoveredLines`[12] und wurde ebenfalls aus den im Vorhinein eingelesenen Dateien mittels der Klasse `TestDataSensor` ermittelt.

Werden nun die Metriken des TestSuccesses näher betrachtet, erkennt man, dass dort eine Visualisierung nach dem gleichen Schema ablaufen muss. So besteht auch dort die Möglichkeit die Höhenmetrik mit einer darauf projizierten Metrik zu vergleichen. Die Höhenmetrik wird dabei durch das Attribut `SF_Method` dargestellt. Es enthält den Wert für die Anzahl der Testmethoden in der jeweiligen Testklasse. Der Wert des Attributs wurde mit Hilfe des `SureFireSensors` ermittelt. Im Vergleich zur Anzahl der Methoden steht nun die Anzahl der fehlgeschlagenen Testmethoden. Diesen Wert liefert wiederum ein anderes Attribut welches mit Hilfe der Combobox eingestellt werden kann. Es trägt den Namen `SF_Failure`. Dieses Attribut beinhaltet den Wert der fehlgeschlagenen Testmethoden der

[11] Test Lines Of Code entspricht der Zeilenanzahl der zu testenden Klassen.
[12] TestCoveredLines entspricht den abgedeckten Zeilen der Testklasse.

Testklassen und wurde durch den `SureFireSensor` ermittelt. Um eine Darstellung dieser Metriken zu ermöglichen, müsste Crococosmo um einen Repräsentationstower `TestDataTower` und dessen Konfiguration namens `DefaultTestData-TowerProperty` erweitert werden.

5.6 Implementierung des Towers und dessen Konfiguration

Die Klasse `TestDataTower` beschreibt einen Repräsentationstyp und erbt von der Klasse `MetricTower`.

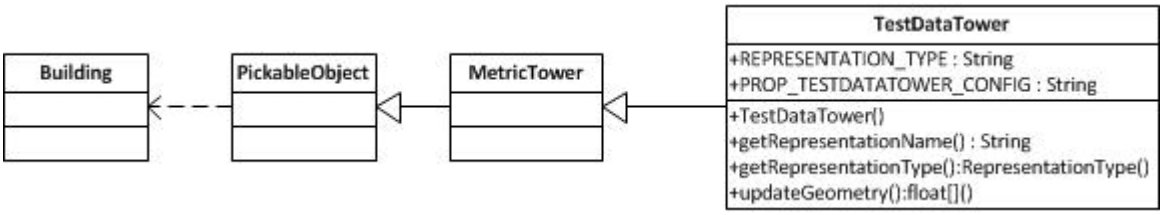

Abbildung 25: Eingliederung des TestDataTowers in Crococosmo

Der Repräsentationstyp `MetricTower` ist darauf ausgerichtet eine Klasse als Zylinder zu visualisieren und eine Stapelung dieser zu erlauben, um zwei Attributausprägungen miteinander vergleichen zu können, bspw. der Vergleich zwischen abgedeckten Zeilen und Gesamtanzahl der Zeilen. Der `TestDataTower` stellt Eigenschaften zur Verfügung, die mit Werten belegt werden können. Diese Eigenschaften sind die Werte der x-,y- und z-Koordinaten, der Farbe und Angabe zur Form des Towers. Eine Angabe zur Form ist nötig, da mit Hilfe des `TestDataTowers` sowohl zu testende Klassen als auch Testklassen visualisiert werden. In der Methode `updateGeometry()` wird mit Hilfe zweier in einander geschachtelter `ArrayLists` die Reihenfolge gewährleistet in welcher die Eigenschaften des Towers in der späteren Konfiguration hinzugefügt werden müssen und in welcher Reihenfolge die verschiedenen Blöcke gestapelt werden sollen.
Diese Konfigurationsklasse trägt den Namen `DefaultTestDataTowerProperty` und ist in dem `PropertyMapping`-Package des `EvoViewers` zu finden.

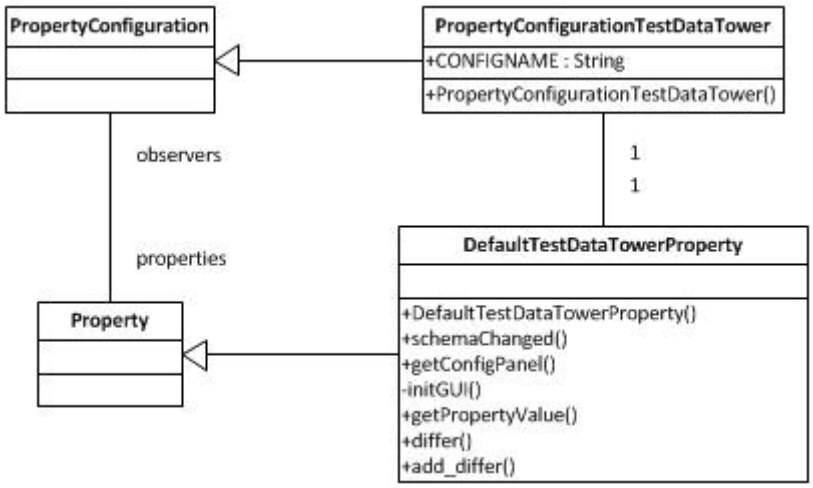

Abbildung 26: Eingliederung der DefaultTestDataProperty und der PropertyConfiguration des TestDataTowers

Sie ermöglicht eine revisionsübergreifende Visualisierung und ist somit für Visualisierungen verschiedenster Szenarien zuständig. Um dies zu ermöglichen, ist es notwendig, die im `TestDataTower` erwähnten Eigenschaften in gegebener Reihenfolge mit Werten zu belegen. Diese Eigenschaften sind die x-,y- und z-Ausdehnung und Farbe und Form des Towers. Die x- und z-Ausdehnung entsprechen der Anzahl aller Beziehungen der Methoden und Attribute einer Klasse sowohl untereinander also auch zu Methoden und Attributen anderer Klassen. Wie aus dem Visualisierungskonzept ersichtlich können keine konkreten Werte für die y-Ausdehnung, Farbe und Form der Tower festgelegt werden, da sie in unterschiedlichen Szenarien verschiedene Werte annehmen. Daher ist es nötig jeden eintretbaren Fall gesondert zu betrachten und die Visualisierung dementsprechend anzupassen. Auftretende Szenarien sind:

- Die Visualisierung der Testklassen bzw. der Testfehler
- Die Visualisierung der zu testenden Klassen bzw. des Abdeckungsgrads
- Die Visualisierung, der zuerst geparsten Version.
- Die Visualisierung der Verminderung der Testfehler oder des Abdeckungsgrads.
- Die Visualisierung der Erhöhung der Testfehler oder des Abdeckungsgrads.

Aufgrund eines ähnlichen Visualisierungskonzeptes müssen sich die Darstellung von Testfehlern und die Darstellung des Abdeckungsgrads in der Form von einander unterscheiden. Daher ist es nötig herauszufinden welche Klassen Testklassen entsprechen und welche wiederum zu testende Klassen wiederspiegeln. Wird eine Klasse als Testklasse identifiziert, wird ihr eine rechteckige Form zugewiesen. Wird eine Klasse wiederum als zu

testende Klasse erkannt, so wird sie als zylinderartige Säule dargestellt. Die `DefaultTestDataTowerProperty` ermöglicht es die aktuell angezeigte Version in Crococosmo mit der vorhergehenden zu vergleichen. Das Szenario der ersten geparsten Version bezeichnet daher einen Sonderfall. Denn dabei ist keine vorhergehende Vergleichsversion vorhanden. Daher werden in diesem Fall die Testfehler und der Abdeckungsgrad nur für die aktuelle Version dargestellt. Wird hingegen eine andere Version zur Darstellung ausgewählt, so ist es möglich diese mit ihrer Vorgängerversion zu vergleichen. Dabei können zwei Fälle auftreten. Entweder haben sich der Testfehler und der Abdeckungsgrad im Vergleich zur vorherigen Version vergrößert oder sie haben sich vermindert. Dabei werden unterschiedliche Farben und y-Ausdehnung genutzt.

6 Evaluation der Visualisierung

Das folgende Kapitel soll das Visualisierungskonzept und die Implementierung in der Praxis zeigen. Es soll das Konzept auf die vorher bereits erwähnten Anforderungen und Ziele der Visualisierung geprüft werden. Weiterhin wird das Konzept mit bereits genutzten Visualisierungskonzepten verglichen und anhand eines Beispiels in der Praxis bewertet.

6.1 Evaluation des Konzeptes

Eine Visualisierung sollte, wie vorher schon erwähnt, folgende Ziele erreichen und ermöglichen [4]:

- Explorative Analyse von Informationsräumen
- Entdecken von neuen Zusammenhängen oder Besonderheiten
- Zuverlässiges Treffen von Entscheidungen
- Finden von Erklärungen von Mustern, Gruppen von Datenobjekten oder einzelnen Eigenschaften

Weiterhin werden folgende Anforderungen bei der Visualisierung erfüllt werden[6]:

- Expressivität
- Effektivität
- Angemessenheit
- Interaktivität

Im Folgenden soll gezeigt werden, dass die in dieser Bachelorarbeit erarbeitete Visualisierung von Testdaten diese Kriterien erfüllt. Dafür wird folgende Abbildung genutzt, um die Kriterien nachzuweisen.

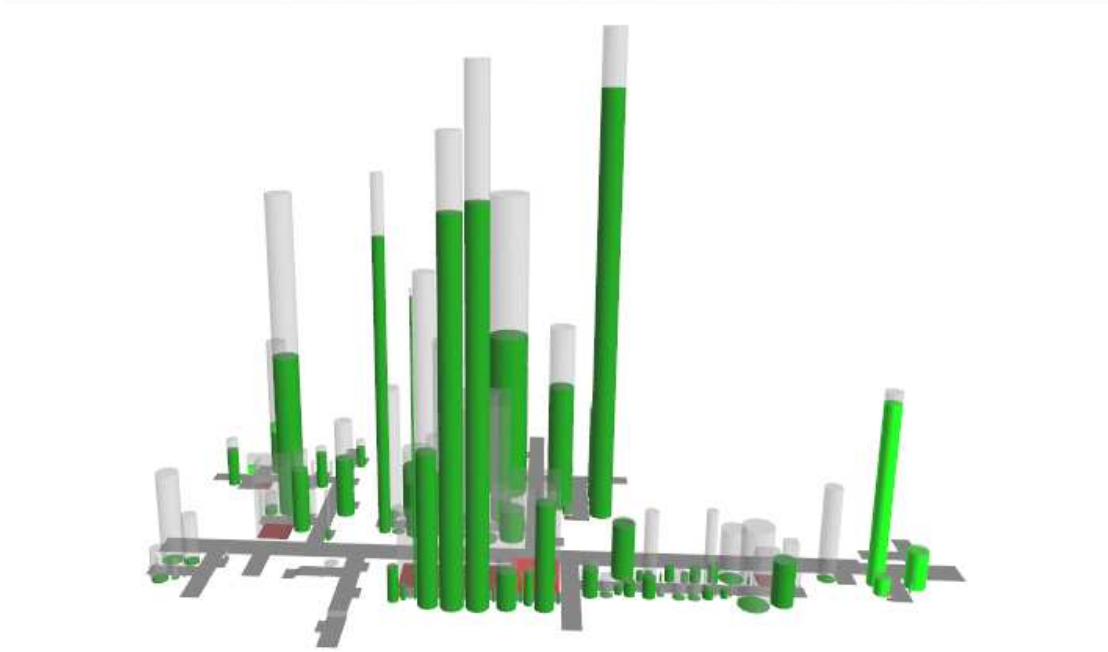

Abbildung 27: Ziele und Anforderungen einer Visualisierung am Beispiel

Eine erfolgreiche Visualisierung sollte eine explorative Analyse der Datenmenge ermöglichen. Durch Crococosmos 3D-Visualisierungsstil wird es dem Benutzer ermöglicht die Datenmenge aus verschiedenen Blickwinkeln zu betrachten und sie so entdeckend wahrnehmen zu können. Auf diese Weise können mehr Informationen über die abstrakten Daten gewonnen werden, als würden sie bspw. in textueller Form vorliegen. Visuelle Darstellungen prägen sich langzeitiger und besser ein und eignen sich besser zur Interpretation als Zahlen. Weiterhin sollte eine Visualisierung in der Lage sein neue Zusammenhänge und Besonderheiten aufzudecken. Dieses Ziel wird in der erarbeitet Visualisierung erreicht, indem revisionsübergreifende Änderungen der Testdaten dargestellt werden. Auf diese Wiese können Zusammenhänge zwischen den Revisionen angestellt werden. Zum Beispiel können Tests einer neuen Version fehlschlagen, die vorher erfolgreich verliefen. Dies könnte für den Testmanager oder Entwickler bedeuten, dass neu erstellte Programmzeilen in den zu testenden Klassen fehlerhaft sind. Schlagen wiederum neue Tests fehl, könnte dies bedeuten, dass neue Fehler entdeckt wurden oder die Behauptungen der Tests fehlerhaft sind. Mit Hilfe dieser Visualisierung kann sich ein Überblick über den Testzustand des Projekts innerhalb kürzester Zeit verschafft werden. So können sofort

Besonderheiten wie eine niedrige Testabdeckung (in der Abbildung als zylinderförmige Säulen ohne Füllstand) oder eine hohe Fehlerquote (in der Abbildung als rechteckige Säulen mit rotem Füllstand) in bestimmten Bereichen des Projekts aufgedeckt werden. In solchen Fällen müssen effiziente Entscheidungen getroffen werden, da in der letzten Software-Entwicklungsphase „Testen“ meist große Zeitnot herrscht. So gilt es nun die Ressourcen und das Personal auf die identifizierten Bereiche zu verteilen, um Abgabefristen und Anforderungen des Auftraggebers einzuhalten.

Die Anforderung der Expressivität wird gewährleistet, da die Daten weder während der Aufbereitung noch der geometrischen Transformation zur Visualisierung verfälscht werden. „Das heißt, nur die in der Datenmenge enthaltene Informationen und nur diese sollen durch die Visualisierung dargestellt werden“[6] Das bedeutet der Anwender bekommt nur eine Auswertung der Testdaten und keiner anderen Daten. Das Kriterium der Effektivität ist erfüllt, da die Visualisierung „die visuelle Fähigkeit des Betrachters und die charakteristischen Eigenschaften des Ausgabegeräts unter Berücksichtigung der Zielsetzung und des Anwenderkontextes optimal ausnutzt.“[6] In den folgenden Darstellung wird erklärt wie die visuelle Fähigkeit des Betrachter ausgenutzt wird, um die möglichen Fälle der Testfehler- und der Testabdeckungsvisualisierung effektiv zu gestalten.

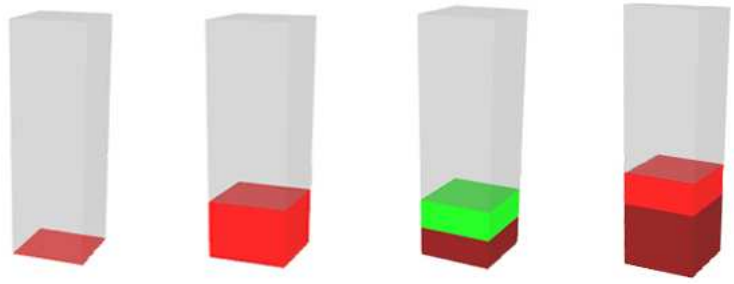

Abbildung 28: Visualisierung der Testfehler in Crococosmo

Die Darstellung der Testfehler soll auffällig sein, da ihnen besondere Aufmerksamkeit zu gewiesen werden muss. Aus diesem Grund wird eine hellrote Färbung für neue Fehler genutzt und eine dunkelrote Färbung für Fehler vorhergehender Revisionen. Werden Fehler im Vergleich zur Vorgängerversion beseitigt wird dies hellgrün dargestellt, und bewirkt damit beim Betrachter ein positives Gefühl. Die Farbwahl ist auf die Assoziationen des Betrachters ausgerichtet. So wird rot als signalisierend erkannt und als Warnfarbe wahrgenommen und grün als angenehm und beruhigend. Nach dem gleichen Prinzip sind die Testabdeckungs-Tower modelliert.

Abbildung 29: Visualisierung der Testabdeckung in Crococosmo

Die Testabdeckung soll genau wie die Testfehler-Visualisierung auffällig sein. Daher werden Bereiche mit verbesserter Testabdeckung hellgrün dargestellt und Bereiche gesunkener Testabdeckung rot. Durch die Farbgebung wird erneut durch hellgrün etwas Positives ausgedrückt und Rot als Signal und Warnfarbe genutzt, da Bereichen gesunkener Testabdeckungen genauso viel Aufmerksamkeit zu kommen muss, wie Testfehlern. Bei beiden Darstellungen wird ein Füllstandprinzip genutzt, da es die Auswertung dieser abstrakten Metriken auf einen Blick ermöglicht. Der Benutzer kann sofort sehen, dass Testmethoden vorhanden sind, die erfolgreich verlaufen oder Zeilen in zu testenden Klassen vorhanden sind, die nicht durch Test abgedeckt werden. Diese Bereiche werden grau dargestellt. Weiterhin kann gesagt werden, dass der Betrachter bei dieser Visualisierung nicht überfordert wird. Da sowohl für die Visualisierung der Testabdeckung und der Testfehler das gleiche Farbkonzept genutzt wird. Um sie voneinander zu unterscheiden wird für beide eine andere Form genutzt. Durch dieses Farb- und Formkonzept wird eine intuitive Auswertung der abstrakten Testdaten ermöglicht und senkt damit auch die Interpretationszeit beim Betrachter. Die Anforderung der Angemessenheit ist damit definitiv erfüllt. „Die Angemessenheit einer Visualisierung beschreibt (…) Aufwand und Kosten zur Durchführung des Visualisierungsprozesses.“[6] Um ein Projekt durch die verwendete Visualisierungspipeline[13] analysieren und visualisieren zu lassen, wird im Durchschnitt ein 10-20 minütiger Zeitaufwand notwendig. Dieser Aufwand ist nicht zu vergleichen mit dem der Auswertung von den Testauswertungsdaten in textueller Form. Im Fall von Cobertura sind diese Dateien über 20000 Zeilen lang und lassen sich nur schlecht analysieren und interpretieren. Eine Visualisierung von Testdaten ist immer zu empfehlen und auch zu bevorzugen. Denn wie schon erwähnt wird jede Minute in der letzten Software-Entwicklungsphase vor dem Ausliefern des Produkts benötigt. Daher ist eine visuelle Darstellung der Testdaten wünschenswert. Die Interaktivität ist bei der Visualisierung

[13] Vgl. Kapitel 5.1 Genutzte Visualiserungspipeline

gegeben. „Der große Nutzen der Interaktion mit der Visualisierung liegt darin, dass der Benutzer damit die Möglichkeit hat, die Visualisierung auf seine Bedürfnisse anzupassen.“[18] Der Benutzer hat innerhalb von Crococosmo die Möglichkeit sich entweder die Testfehler-Visualisierung und die Testabdeckungsvisualisierung einzeln darstellen zu lassen, oder aber auch gemeinsam in einer Darstellung. Er kann die Visualisierung hierbei auf sein Interessengebiet einschränken. Auf diese Weise wird er nicht mit zusätzlichen Informationen überhäuft, deren Auswertung für ihn an dieser Stelle überflüssig wäre. Sind Bereiche der Visualisierung anhand der Übersicht schwer zu erkennen, durch bspw. Überdeckungen, ist der Anwender in der Lage, den Blickwinkel zu ändern und bestimmte Bereiche heran zu zoomen. Auf diese Weise ist die Interaktion mit der Visualisierung ebenfalls gegeben.

6.2 Vergleich zwischen erarbeitetem und bereits genutzten Visualisierungskonzepten

In diesen Abschnitt soll die erarbeitete Visualisierung mit den Visualisierungskonzepten Statusanzeigen, Tabellen, Diagrammen und Treemaps verglichen werden. Für diesen Vergleich werden die Testdaten eines Projekts durch Clover, Cobertura und durch die erarbeitete Visualisierung dargestellt.

Als erstes wird die erarbeitete Visualisierung mit dem Konzept der Statusanzeige und Tabellen verglichen.

Package	Classes		Line Coverage
org.openfast.examples	9	0%	0/72
org.openfast.examples.consumer	3	0%	0/80
org.openfast.examples.decoder	2	0%	0/67
org.openfast.examples.interpret	2	0%	0/119
org.openfast.examples.performance	4	0%	0/185
org.openfast.examples.producer	5	0%	0/239
org.openfast.examples.scp10	2	0%	0/81
org.openfast.examples.tmplexch	4	0%	0/156
org.openfast.examples.util	2	0%	0/49
org.openfast.examples.xml	4	0%	0/182
org.openfast.extensions	2	0%	0/38

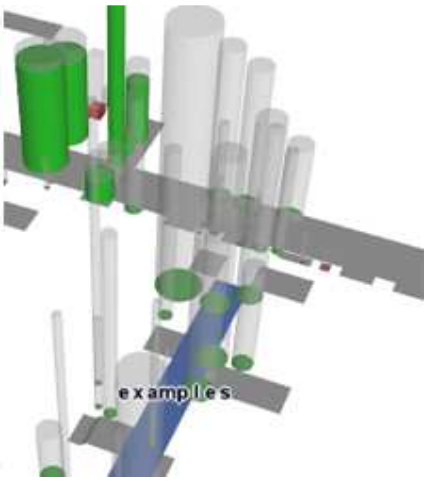

Abbildung 30: Vergleich Tabellen/Statusanzeigen und erarbeiteter Visualisierung

In der vorherigen Abbildung wird das erarbeitete Visualisierungskonzept dem Konzept der Statusanzeigen und Tabellen gegenübergestellt. Beide Konzepte stellen in dieser Abbildung den selben Inhalt dar, nämlich das keine Testabdeckung in dem Paket Examples erfolgt. Es fällt auf, dass das erarbeitete Visualisierungskonzept optisch ansprechender ist. Der Anwender wird nicht sofort mit allen zur Verfügung stehenden Informationen überfordert. Im Tabellen-Konzept erregt die rote Färbung der Statusanzeigen die Aufmerksamkeit des

Anwenders, er bemerkt, dass dort die Testabdeckung niedrig bzw. nicht vorhanden ist. Danach sucht er in der gleichen Zeilen nach dem Namen des Pakets. Dies muss er für alle Unterpakete des Pakets Examples tun, um herauszufinden, dass keine Testabdeckung im gesamten Paket vorliegt. Der Nutzer muss dabei also mehrere Interpretationsschritte durchführen. Im erarbeiteten Visualisierungskonzept hingegen kann er auf einen Blick erkennen, dass im Paket Examples keine Testabdeckung vorliegt. Das Prinzip der Tabelle bietet eine effiziente Übersicht über alle wichtigen Informationen gleichzeitig und nutzt dabei den zur Verfügung stehenden Platz gut aus. Es fördert allerdings dadurch auch ein Verrutschen innerhalb der Zeilen auf Grund ihrer Anordnung. Dadurch kann es schnell zu Fehlinterpretationen kommen und so den Arbeitsaufwand erhöhen. Die Wahrscheinlichkeit dafür ist beim erarbeiteten Prinzip trotz gegebener Übersichtlichkeit stark eingegrenzt. Denn will der Nutzer den zur Testauswertung gehörigen Namen der Klassen wissen, braucht er nur die Maus über den Tower bewegen und der Name wird angezeigt.
Nun soll das Konzept der Diagramme mit dem erarbeiteten Konzept verglichen werden.

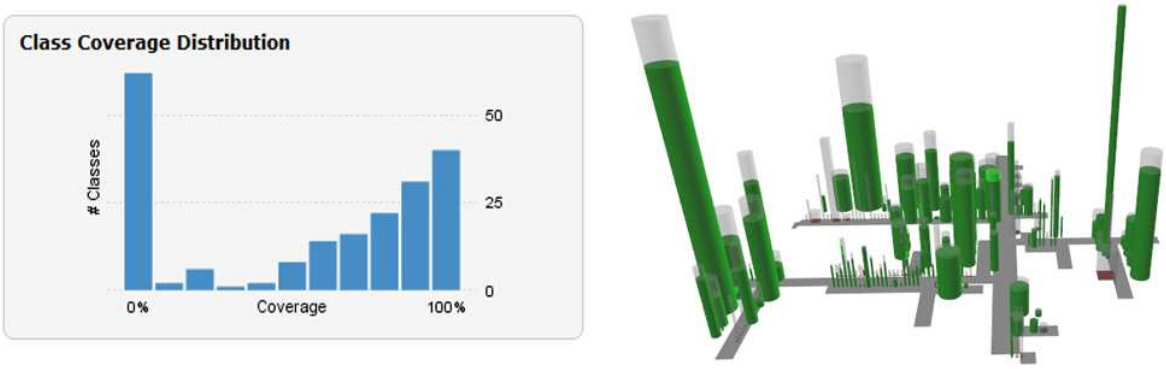

Abbildung 31: Vergleich Diagramm und erarbeiteter Visualisierung

Beide Darstellungsarten zeigen den gleichen Inhalt auf. Sie stellen die Testabdeckung des gesamten Projekts in einer Darstellung dar. Aus dem Diagramm wird ersichtlich, dass mehr als 50 Klassen eine 0%ige Testabdeckung haben und mehr als 40 Klassen eine 100%ige. Dies kann im erarbeiteten Prinzip ebenfalls festgestellt werden, in dem der Blickwinkel geändert und die Zoom-Funktion genutzt wird. Auffällig hierbei ist, dass beide Prinzipien optisch ansprechend sind und dabei allerdings den Benutzer nicht überfordern. Beim erarbeiteten Visualisierungskonzept werden Klassen mit einer hohen Zeilenanzahl in den Vordergrund gehoben. Eine hohe Zeilenanzahl bedeutet hohe Fehleranfälligkeit. Solchen Bereiche sollten ersichtlich sein, da dort eine hohe Testabdeckung erwünscht ist. Im Diagramm sind leider keine Klassennamen erkenntlich. Dies kann im erarbeiten Konzept wiederum in Erfahrung gebracht werden, wenn mit der Maus über den Klassentower gefahren wird. Im Allgemeinen kann gesagt werden, dass sich dieses Diagramm gut eignet, um sich einen Überblick zu

verschaffen. Allerdings nicht genügend Information bietet zur Behebung des Testabdeckungsproblems. Das ist darauf zurückzuführen, dass Diagramme zu den 2D-Ansichten gehören und sich somit im Vergleich zum erarbeiteten 3D-Konzept weniger Informationen anzeigen lassen.

Der letzte Vergleich soll den Unterschied zwischen einer Treemap-Darstellung und dem arbeiteten Visualisierungskonzepts herausgearbeitet werden.

Abbildung 32: Vergleich Treemap und erarbeiteter Visualisierung

Beide Visualisierungskonzepte zeigen erneut den gleichen Inhalt auf. Sie zeigen eine Übersicht über die Testabdeckung der Unterpakete des Projekts Openfast. Bei der Treemap zeigt die Stärke der Färbung den Abdeckungsgrad an, d.h. je grüner eine Färbung ist desto besser ist die Testabdeckung. Auffällig bei dieser Gegenüberstellung ist, dass beide Visualisierungskonzepte die Hierarchie innerhalb des Projekts mit der Visualisierung der Testabdeckung vereinen. Sie nutzen somit effizient den Platz aus und bieten auf eine Blick eine gute Interpretationsgrundlage. So ist bei der Treemap und bei der erarbeiteten Visualisierung sofort ersichtlich in welchen Bereichen mehr Test-Aufwand, mehr Personal und mehr Ressourcen eingesetzt werden müssen, um ein nahezu fehlerfreies und somit qualitativ hochwertiges Projekt gewährleisten zu können. Im Unterschied zur Treemap schafft es die erarbeitete Visualisierung die Testabdeckung der einzelnen Klassen in der Übersicht darzustellen. Bei der Treemap wär dazu ein Navigieren durch die Hierarchiestruktur des Projekts notwendig und würde einen höheren Zeitaufwand bedeuten.

Der stärkste Unterschied zur Treemap ist allerdings die Darstellung der Testfehler. Die Treemap-Darstellung stößt dabei an ihre Grenzen. Da sie nur entweder in der Lage ist die Testabdeckung zu zeigen oder den Testerfolg bzw. Testfehler. Für eine Visualisierung beider Metriken in einer Darstellung stehen die nötigen Mittel nicht mehr zur Verfügung, Farbe, Form und Text wurden schon im anderen Zusammenhang genutzt. Das erarbeitete Visualisierungskonzept nutzt dafür eine andere Form der Tower. Sie sind nun rechteckige

Säulen, die sich je nach Anzahl der Testmethoden in der Höhe ausbilden und sich nach der Anzahl der Testfehler füllen. Die Methoden, die zur Zeit zur Testfehler-Visualisierung genutzt werden sind die SeeSoft-Darstellung, die Editoransicht und die Statusanzeigen. Editor-und SeeSoft-Darstellungen können nicht mit der erarbeiteten Visualisierung verglichen werden, da sie Text farblich unterlegen. Auf Text wurde bei der erarbeiteten Visualisierung bewusst verzichtet, denn sie befasst sich mit der Darstellung abstrakter in Textform vorliegenden Daten, um sie verständlich zu gestalten. Eine textuelle Darstellung würde diesem Prinzip wiedersprechen.
Das Prinzip der Statusanzeigen im Hinblick auf die Testfehler kann allerdings mit der erarbeiteten Visualisierung verglichen werden.

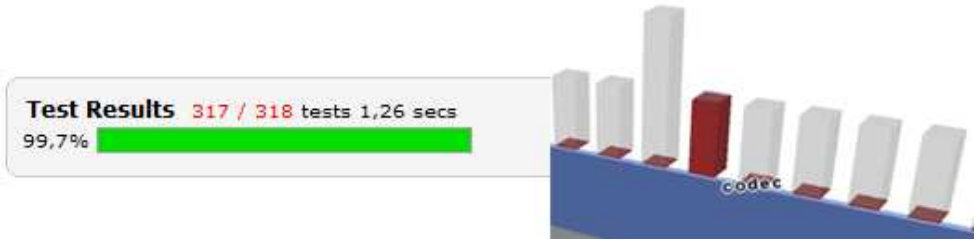

Abbildung 33: Vergleich der Testfehlerdarstellung bei Statusanzeige und erarbeiteter Visualisierung

Beide Visualisierungen weisen auf den gleichen Fehler hin. Bei der Statusanzeige könnte der Fehler schnell übersehen werden, da er keinen wahrnehmbaren Ausschlag auf der Anzeige hervorruft. Verlässt man sich also nur auf die Anzeige und liest nicht die dazugehörige Prozentanzeige, würde dieser Fehler nicht auffallen. Sieht man sich allerdings die Darstellung des Fehlers in Crococosmo an, ist er aufgrund der roten Farbe nicht zu übersehen. Weiterhin kann mit Hilfe von der erarbeiteten Visualisierung der genaue Ort der fehlschlagenden Testmethode auswendig gemacht werden. Somit sinkt der Aufwand bei der Fehlerlokalisierung im Vergleich zur Statusanzeige, denn dort ist nicht vermerkt an welcher Stelle ein Test fehlschlägt. SeeSoft-Darstellungen und Editoransichten wären nun noch in der Lage die fehlerbehafte Zeile festzustellen, auf Grund der direkten Auswertung der Testfehler am Text.
Der größte Vorteil der Testdaten-Visualisierung in Crococosmo ist die revisionsübergreifende Darstellung. Alle im Kapitel Stand der Technik genannten Programme visualisieren diese Daten nur für eine Version und zeigen somit leider nicht den Unterschied zur Vorgängerversion. Das bedeutet wichtige Informationen über die Testdaten können verloren gehen. Zum Beispiel das Löschen einer Testmethode oder Testklasse und damit sinkende Testabdeckung. Sie wäre nur im Vergleich zwischen zwei Testauswertungsdateien für zwei Revisionen zu erkennen.

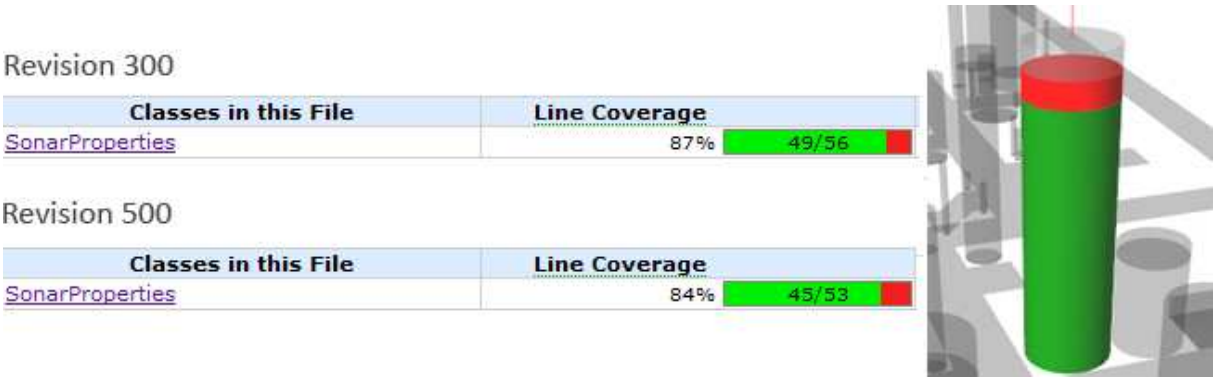

Abbildung 34: Besonderheit revisionsübergreifende Visualisierung

In der vorherigen Abbildung ist zu sehen, dass pro Revision ein Wert des Abdeckungsgrades für die Klasse SonarProperties vorliegt. Dieser Wert hat sich zwischen beiden Revisionen verändert. Um diese Veränderung zu bemerken, benötigt man beide Testauswertungsdateien der Revisionen, während das erarbeitete Visualisierungskonzept dies in einer Darstellung wiedergibt. Rechts in der Abbildung ist die Klasse SonarProperties in Revision 500 zu sehen. Die Grünfärbung besagt dabei die Testabdeckung der Revision 500. Der rote Bereich zeigt wiederum an das die Testabdeckung gesunken ist. Die Visualisierung in Crococosmo liefert dem Anwender also auch die Informationen zur Vorgängerversion auf einen Blick. Das gleiche würde für das Szenario der gestiegen Testabdeckung, gesunkenen Fehler oder gestiegenen Fehleranzahlen gelten. Weiterhin ist die erarbeitete Darstellung in der Lage, eine im folgenden Szenario bessere Informationen über die Testfehler zu geben.

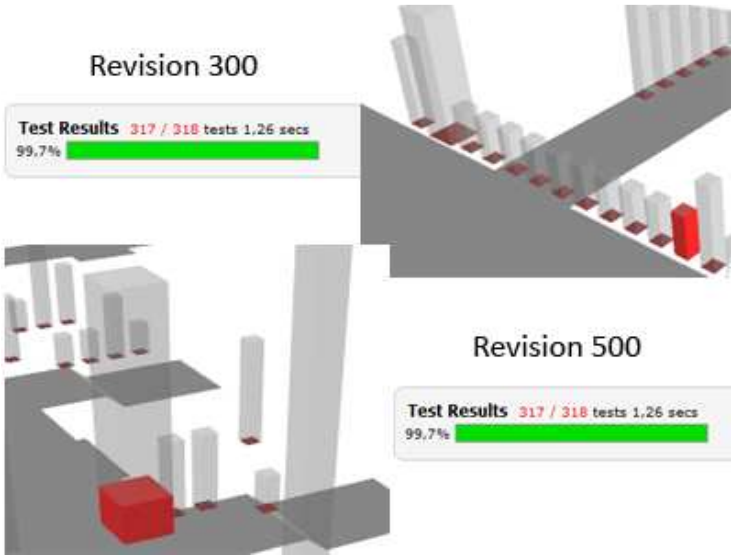

Abbildung 35: Besonderheit revisionsübergreifende Visualisierung

An dieser Abbildung sind zwei unterschiedliche Revisionen eines Projekts zu erkennen, die jeweils einen Fehler enthalten. Die Statusanzeige zeigt zwar an, dass sich ein Fehler im Projekt befindet, aber nicht an welcher Stelle. Der Programmierer könnte nun davon ausgehen, dass der Fehler in Revision 300 und in Revision 500 der gleiche ist und auch an der

gleichen Stelle suchen. So würde der Aufwand für die Fehlerlokalisierung unnütz ansteigen. Die Visualisierung zeigt dabei klar auf, dass sich der Fehler nicht an der gleichen Stelle befindet, wie in der Vorgängerversion. Eine genaue Lokalisierung des Fehlers kann erreicht werden, wenn mit der Maus über den angezeigten Testfehler gefahren wird. Mit Hilfe der erarbeiteten Visualisierung fällt die Auswertung dieser abstrakten Daten leichter, es können schneller Zusammenhänge gefunden und korrigiert werden und Schwachstellen auswendig gemacht werden.

6.3 Evaluation der Testauswertung am Beispiel Sonar-IDE

Das Projekt Sonar-IDE liegt in drei unterschiedlichen Revisionen vor, an denen die Entwicklung der Testdaten gut zu erkennen ist. In der folgenden Abbildung ist Sonar-IDE in der Revision 200 zu sehen.

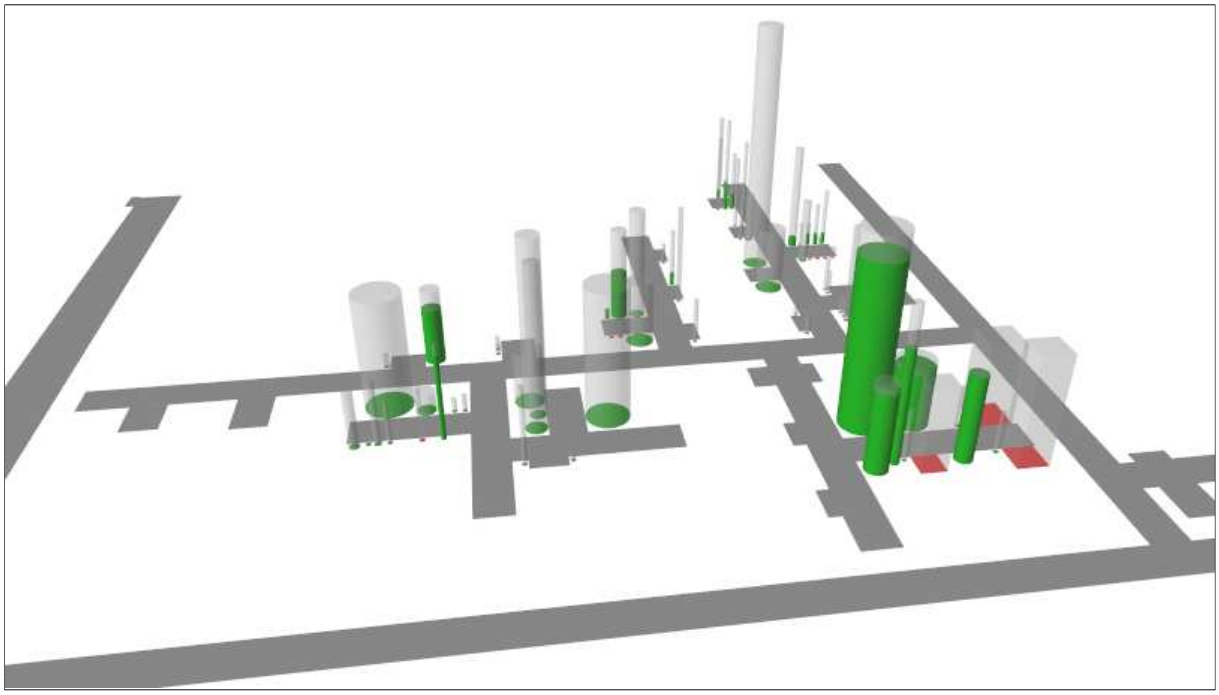

Abbildung 36: Sonar-IDE Revision 200

Auf den ersten Blick stellt man fest, dass die Testabdeckung, hier anhand von zylinderartigen Säulen zu erkennen, sich nicht über das gesamte Projekt ausdehnt. Es ist im vorderen Bereich zu sehen, dass dort der Füllstand der Klassen hoch ist, also die Testabdeckung im Durchschnitt bei 80-90% liegt. Vergleicht man diesen Bereich nun mit dem restlichen Projekt, wird ersichtlich, dass sich bei vielen Klassen kein Füllstand ausbildet. Also die Klassen nicht durch Tests abgedeckt sind. Die Visualisierung zeigt in diesem Fall auf, dass wenige Komponenten des Projekts getestet wurden. Das könnte bedeuten, dass sich in diesen Bereichen nichtentdeckte Software-Fehler befinden. Diese würden eine niedrigere Software-

Qualität hervorrufen, die sowohl beim Entwickler als auch beim Auftraggeber und natürlich auch beim Kunden unerwünscht ist. Die Visualisierung hilft in diesem Fall dem Entwickler oder dem Testmanager einen Eindruck vom Testzustand des Projektes zu bekommen. Es könnten zur Verbesserung der Testabdeckung nun Ressourcen und Personal effizienter eingesetzt werden, nämlich in den Bereichen der niedrigen Testabdeckung. Weiterhin ist zu erkennen, dass keine Fehler in den Testklassen auftreten, hier anhand von rechteckigen Säulen dargestellt. Es prägt sich kein Füllstand aus und dementsprechend schlagen keine Testmethoden fehl.

In folgender Abbildung ist Sonar-IDE in der Revision 300 zu sehen. Man erkennt eine Entwicklung der Testdaten.

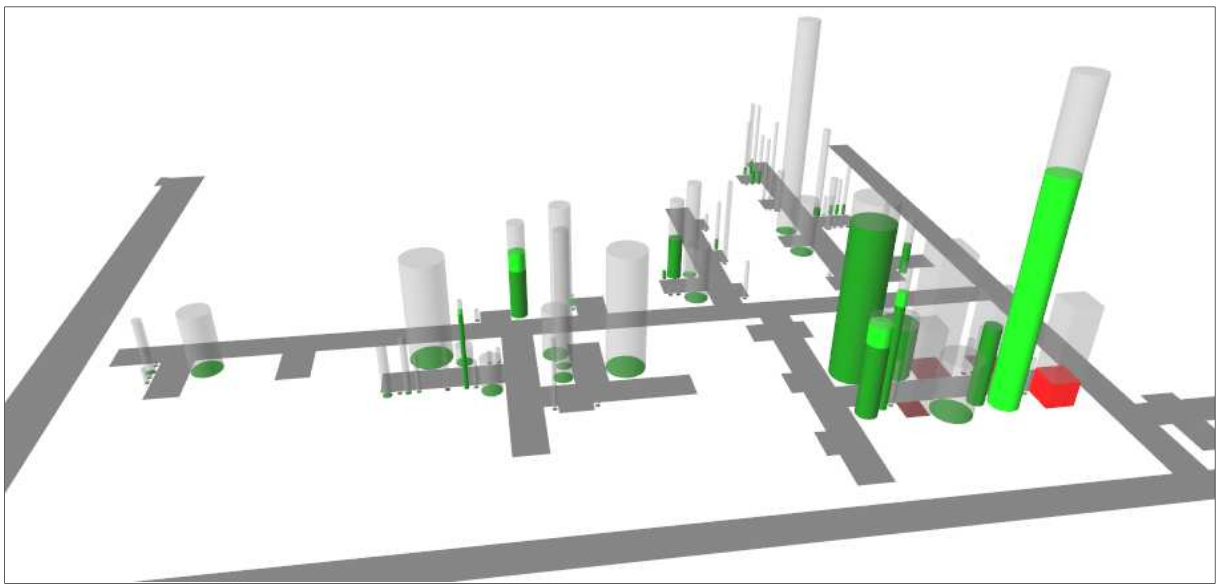

Abbildung 37:Sonar-IDE Revision 300

Die Entwicklung und Verbesserung der Testabdeckung ist anhand der hellgrünen Färbung zu erkennen. Die dunkelgrüne Färbung zeigt auf welchen Abdeckungsgrad die vorherige Version besaß. Die Aufmerksamkeit des Betrachters wird sofort auf die hellen farblich gekennzeichneten Bereiche gelenkt. So erkennt er schnell, dass die Testabdeckung in manchen Bereichen gestiegen ist, das ist an der hellgrünen Farbe zu erkennen. Er erkennt aber auch, dass Fehler in den Testmethoden vorhanden sind, dies ist anhand der hellroten Färbung der rechteckigen Säulen zu erkennen. Dieser Bereich benötigt besonderer Aufmerksamkeit. Testfehler bedeuten, dass Behauptungen, die in den Testmethoden angestellt wurden, nicht zu treffen oder nicht eintreten. Das kann zum einen an der Behauptung liegen, oder aber auch daran, dass der gewollte Zustand oder Wert in den zu testenden Klassen nicht auftritt. Das bedeutet, dass an dieser Stelle keine nahezu fehlerfreie Software gewährleistet werden kann. Treten diese Fälle in der Visualisierung auf, muss der Testmanager oder der Entwickler selbst

sofort handeln und diesen Fehler näher untersuchen und korrigieren. Er kann den Fehler genau lokalisieren und braucht nicht lange zu suchen. Weiterhin zeigt die Visualisierung erneut auf, dass der Abdeckungsgrad gering ist. Somit können sich noch viele nichtentdeckte Testfehler in der Software befinden, die wiederum Zuverlässigkeit und die Qualität der Software senken. Das bedeutet für den Testmanager oder den Entwickler, dass mehr Tests geschrieben werden müssen, um den eventuellen Softwarefehler weiterhin einzugrenzen.
Die folgende Abbildung zeigt Sonar-IDE in der Version 500. Dabei ist zuerkennen, dass sich das Projekt räumlich ausgedehnt hat, aber die Testabdeckung nicht sonderlich gestiegen ist.

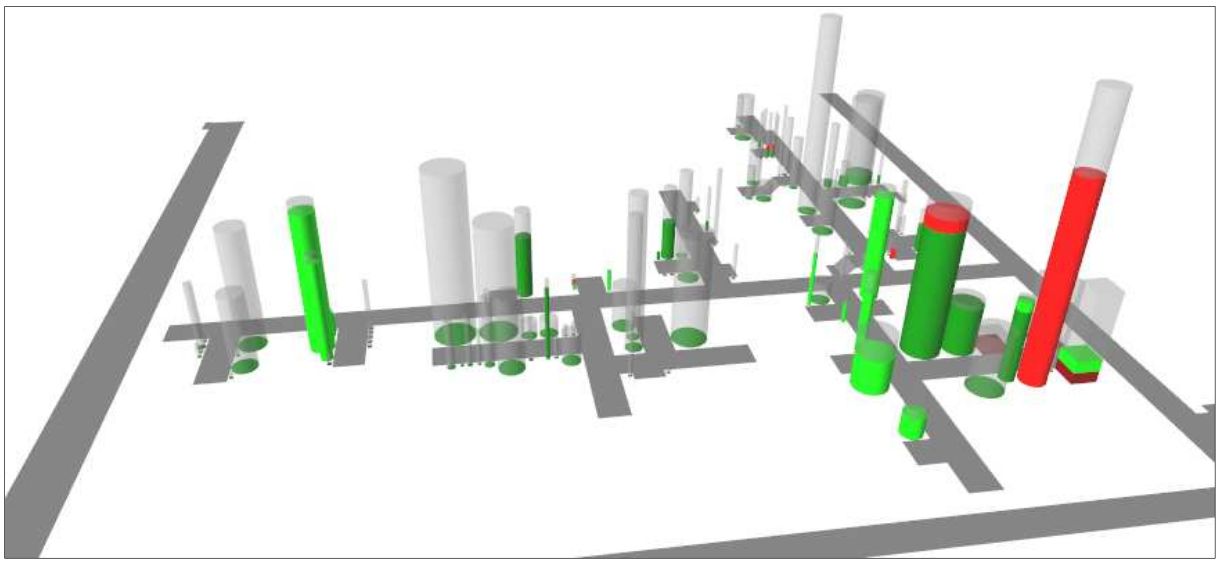

Abbildung 38: Sonar-IDE Revision 500

Aufgrund der revisionsübergreifenden Visualisierung lassen sich sofort Änderungen zur Vorgängerversion feststellen. So wird dem Anwender auf den ersten Blick mitgeteilt, dass sich die Testabdeckung in manchen Bereichen verbessert hat, hier hellgrün gekennzeichnet, und dass sich aber auch die Testabdeckung verringert hat, hier anhand einer hellroten Färbung der zylinderförmigen Säulen zu erkennen. In solchen gilt es herauszufinden, warum die Klassen nicht mehr durch Testmethoden ausgeführt werden. Das kann bspw. durch Löschen einer Testklasse geschehen oder durch Veränderungen der zu testenden Klasse. Es muss behoben werden, da sich an dieser Stelle erneut die Wahrscheinlichkeit erhöht Software-Fehler nicht zu entdecken. Sieht der Anwender sich nun die Testfehler an, erkennt er, dass Fehler behoben worden sind, allerdings noch Testmethoden fehlschlagen. Diese Bereiche erfordern erneut höhere Aufmerksamkeit, um eine zuverlässige Software zu gewährleisten.

Zusammenfassung

Die Visualisierung hilft im Allgemeinen die umfangreiche Anzahl an Dokumenten schnell und intuitiv auswerten zu können. Um an solche Informationen, wie in den vorherigen

Abbildung dargestellt, zu gelangen, müssten beim Sonar-IDE Projekt 67 von Maven erstellte Testauswertungsdateien gelesen und interpretiert werden. Der Arbeitsaufwand dafür wäre immens. Zeitaufwändige Auswertungen können in der letzten Software-Entwicklungsphase „Testen“ nicht benötigt werden, da dort akute Zeitknappheit herrscht und auf diese Weise viele eventuelle Testfehler übersehen werden könnten. Diese Visualisierung eignet sich sehr gut, um abstrakte Daten optisch auswerten zu können. Während Zahlen und Prozentwerte über Testabdeckung und Testfehler großen Spielraum für Fehlinterpretationen lassen, grenzt die Visualisierung dies stark ein. Aufgrund der Farbwahl, wird die Aufmerksamkeit des Anwenders sofort auf wichtige Bereiche gelenkt, die entweder eine Verbesserung zur Vorgängerversion aufzeigen oder aber eine Verschlechterung, und somit signalisieren, dass an diesen Stellen mehr Tests produziert werden müssen oder aber Testfehler korrigiert werden müssen. Visualisiert man Testdaten nicht, erkennt man nur mühselig und langsam die Verteilung der Testabdeckung über das gesamte System. Anhand der Visualisierung können revisionsübergreifende Zusammenhänge aufgedeckt werden, die vorher eventuell nicht aufgefallen sind. So kann eine Beziehung zwischen neuen Zeilen und Tests analysiert werden, bspw. werden neu eingefügte Zeilen auch durch bereits bestehende Tests abgedeckt. Man kann herausfinden, ob neue Tests fehlschlagen, oder fehlschlagende Tests nach Änderung der zu testenden Klasse erfolgreich verlaufen. Eine Visualisierung dieser abstrakten Daten schränkt den Arbeitsaufwand stark ein. Eine revisionsübergreifende Visualisierung schafft es sogar mehr Zusammenhänge zu entdecken und somit effektiver testen zu können.

Literaturverzeichnis

[1] Joachim Stary, Visualisieren. Berlin 1997, S. 12

[2] Klaus Grimm, Systematisches Testen von Software. Eine neue Methode und eine effektive Teststrategie. München Wien 1995, S.13

[3] Edsgar Wybe Dijkstra, The Humble Programmer, ACM Turing Lecture, 1972, S.6

[4] Bernhard Preim, Raimund Dachselt, Interaktive Systeme. Band 1: Grundlagen, Graphical User Interfaces, Informationsvisualisierung. Berlin Heidelberg 2010², S. 440-441

[5] Ben Shneiderman, The Eyes have it. A Task by Data Type Taxonomy for Information Visualizations. Maryland 1996, S. Online verfügbar: http://ebookbrowse.com/the-eyes-have-it-a-task-by-data-type-ta-43453-pdf-d43843777 Abrufdatum: 01.02.2011

[6] Heidrun Schumann, Wolfgang Müller, Visualisierung. Grundlagen und allgemeine Methoden. Berlin Heidelberg 2000, S. 9-13

[7] Heidrun Schumann, Wolfgang Müller, Visualisierung. Grundlagen und allgemeine Methoden. Berlin Heidelberg 2000, S.84

[8] Homepage http://www.junit.org, Abrufdatum: 08.10.2010

[9] Johannes Link, Softwaretests mit JUnit. Heidelberg 2005², S. 90

[10] Kai Uwe Bachmann, Maven 2.Eine Einführung, aktuell zur Version 2.0.9. München 2009, S.16 - 21

[11] Martin Spiller, Maven 2. Konfigurationsmanagement mit Java. Heidelberg, München, Frechen, Landsberg, Hamburg 2009, S.80 – 81

[12] Robert Spence, Information Visualization. Barcelona 2001, S. 1-2

[13] Stuart Card, Jock Mackinley, Readings in Information Visualization. Using Vision to Think. San Francisco 1999, S.6

[14] Rolf Däßler, Informationsvisualisierung. Stand, Kritik und Perspektiven. 1999 S. 2 Online verfügbar: http://fabdaz.fh-potsdam.de/daesslerwp/wp-content /uploads /2010/ 04/ informationsvisualisierung.pdf Abrufdatum: 01.02.2011

[15] Heidrun Schumann, Wolfgang Müller, Visualisierung. Grundlagen und allgemeine Methoden. Berlin Heidelberg 2000, S. 16 -20

[16] Heidrun Schumann, Konzepte und Methoden der wissenschaftlichen Visualisierung. 2004, S. 1 Online verfügbar: http://www.informatik.uni-rostock.de/~schumann/papers /2004+/Sommerschule_schumann.pdf Abrufdatum: 01.02.2011

[17] Ed H. Chi, A Taxonomy of Visualization Techniques using the Data State Reference Model. 2000, S. 2 Online verfügbar: http://www-users.cs.umn.edu/~echi/papers /infovis00/Chi-TaxonomyVisualization.pdf Abrufdatum: 01.02.2011

[18] Daniel Murrmann, Datenvisualisierung. Saarbrücken 2008, S. 22

[19] Bernhard Preim, Raimund Dachselt, Interaktive Systeme. Band 1: Grundlagen, Graphical User Interfaces, Informationsvisualisierung. Berlin Heidelberg 2010², S. 452- 477

[20] Ralf Turtschi, Die Gestaltgesetze. 2008. S. 2-3 Online verfügbar: http://www.agenturtschi.ch/t3/fileadmin/download/fachartikel/gestaltgesetze.pdf Abrufdatum: 01.02.2011

[21] Homepage: http://maven.apache.org/guides/introduction/introduction-to-the-lifecycle.html Abrufdatum: 01.02.2011

[22] Martin Spiller, Maven 2. Konfigurationsmanagement mit Java. Heidelberg, München, Frechen, Landsberg, Hamburg 2009, S.67-72

[23] Homepage: http://software-cities.org/gallery/crococosmo/ Abrufdatum: 01.02.2011

[24] Homepage: http://www.spiegel.de/netzwelt/netzpolitik/0,1518,670062,00.html Abrufdatum: 3.02.2011

[25] Homepage: http://www.uni-koblenz.de/~beckert/Lehre/Seminar-Softwarefehler/ Abrufdatum: 03.02.2011

[26] Torsten Cleff, Basiswissen. Testen von Software. Witten 2010, S. 270-272