Daniel Ramm

Untersuchungen und Evaluierung von Modellen, Methoden und Tools zur Bewertung des objektorientierten Entwurfs

DIPLOMARBEIT

HOCHSCHULE MITTWEIDA (FH)
UNIVERSITY OF APPLIED SCIENCES

Mathematik / Physik / Informatik

Mittweida, 2009

Daniel Ramm

Untersuchungen und Evaluierung von Modellen, Methoden und Tools zur Bewertung des objektorientierten Entwurfs

eingereicht als

DIPLOMARBEIT

an der

HOCHSCHULE MITTWEIDA (FH)
UNIVERSITY OF APPLIED SCIENCES

Mathematik / Physik / Informatik

Mittweida, 2009

Erstprüfer: Prof. Dr.-Ing. Wilfried Schubert
Zweitprüfer: Prof. Dr. rer. nat. Konrad Schulz

Vorgelegte Arbeit wurde verteidigt am:

Bibliographische Beschreibung

Ramm, Daniel:

Untersuchungen und Evaluierung von Modellen, Methoden und Tools zur Bewertung des objektorientierten Entwurfs.-2009.- 65 S. Mittweida, Hochschule Mittweida (FH), Fachbereich Mathematik - Physik - Informatik, Diplomarbeit, 2009

Referat:

Ziel der Diplomarbeit ist das Untersuchen aktuell angebotener Metriken vor allem im Bereich der objektorientierten Metriken. Damit sollen Kriterien gefunden werden, um einen Quellcode zu bewerten und diesen als „guten/ konformen Code“ bezeichnen zu können. Darauf aufbauend sollen Tools gefunden werden, die nach diesen Kriterien/ Metriken suchen und helfen auch größere Projekte so zu bewerten. Diese Funktionsfähigkeit soll in dem nächsten Schritt untersucht und bewiesen werden.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings-Verzeichnis

Abkürzungsverzeichnis

A	Abstractness
AC	Number of Abstract Classes/ Interfaces
C	Dependency Cycle
Ca	Afferent Couplings
CC	Concrete Class Count
Ce	Efferent Couplings
CLOC	Comment Lines of Code
CSV	Comma-Separated Values
CV	Class Variables
CYC	Cyclic Dependencies
D	Distance from the Main Sequence
DC	Density of Comments
DCYC	Direct Cyclic Dependencies
DIP	Dependency Inversion Principle
DIT	Depth in Tree
DOI	Depth of Inheritance
EP	Encapsulation Principle
EXEC	Executable Statements
HTML	Hypertext Markup Language
I	Instability
LCOM	Lack of Cohesion in Methods
LoC	Lines of Code

LSP	Limited Size Principle
MLOC	Method Lines of Code
MQ	Modularization Quality
NBD	Nested Block Depth
NCSS	Non Commented Source Statement
NLOC	Non-Comment Lines of Code
NOA	Number of Attributes
NOC(1)	Number of Children in Tree
NOC(2)	Number of Classes
NOD	Number of Descendants
NOF	Number of Fields
NOI	Number of Interfaces
NOM	Number of Methods
NOP	Number of Packages
NORM	Number of Redefined/ Overriden Methods
NOT	Number of Types
NOTa	Number of Abstract Types
NOTc	Number of Concrete Types
NOTe	Number of Exported Types
NP	Number of Parameters
NSC	Number of Children
NSF	Number of Static Attributes
NSM	Number of Static Methods
NT	Number of Tramps
OO	Object-Oriented
OV	Object Variables
PAR	Number of Parameters

RFC	Response for Class
SIX	Specialization Index
TLOC	Total Lices of Code
TXT	Textdatei
WAC	Weighted Attributes per Class
WMC	Weighted Methods per Class
XML	Extensible Markup Language
V(G)	Zyklomatische Komplexität

1. Einleitung

1.1. Motivation

Durch ständiges Weiterentwickeln vorherrschender Programmiersprachen und Architekturmuster, ist es möglich stabile, benutzerfreundliche Anwendungen zu schreiben. Die Anforderungen wie Funktionalität, Zuverlässigkeit, Benutzbarkeit, Wartbarkeit und vielen weiteren sind alle umsetzbar, nur unterschiedlich gut. Leider sind alle Anforderungen von einem Faktor abhängig, dem Programmierer. Da es, im Gegensatz zu einer Programmiersprache, für Programmierer keine festen, einheitlichen Vorschriften gibt, wie ein Programm geschrieben wird, kann es für eine Aufgabe verschiedene Programme geben. Obwohl alle Anforderungen erfüllt sein können, sind sie unterschiedlich gut umgesetzt. Darüber hinaus ist ein Programm nichts absolutes, weitere Funktionen können hinzugefügt oder verändert werden, Ein ständiger Blick auf die Anforderungen und das Wachsen der Fähigkeiten eines Programmierers ermöglicht das Verbessern des Quellcodes.

Leider steht die Komplextität des Quellcodes immer mit den Fähigkeiten des Programmierers in Konflikt. Sei es, weil sich ein anderer Programmierer um den Quellcode kümmert, ob mit der Zeit Zusammenhänge vergessen wurden, eine weitere Funktion dem Programm hinzugefügt werden mußte oder Fehler die man erst später entdeckt hat und jetzt korrigieren muß. Dies sind alles Beispiele dafür, wenn sich ein Programm verändert, meist jedoch zum schlechteren. Um den Faktor Mensch als einen der größten Fehlerquellen ausschließen zu können und dem Schreiben von Programmcode eine einheitliche Richtung zu geben, entstand ein weiteres Teilgebiet der Informatik, die „Software-Qualtität“.

Dies war jedoch zu Zeiten, als es das Konzept des objektorientierten Programmierens noch nicht gab.

1.2. Ziel dieser Arbeit

Im Auftrag der Hochschule Mittweida (FH) soll eine Untersuchung statt finden, die aktuelle Modelle, Methoden und Tools zur Bewertung des objektorientieren Entwurfs evaluiert.

Im Rahmen dieser Arbeit, sollen aktuell gültige Metriken dargestellt werden. Es geht hierbei um die statische Code-Analyse, die geschriebene Code untersucht, bevor er kompiliert und zur Ausführung kommt. Dabei wird untersucht, inwieweit diese mit einem objektorientieren Entwurf vereinbar sind. Während der Untersuchung sollen verfügbare Tools die auf diesen Metriken aufbauen dargestellt und auf deren Leistungsfähigkeit untersucht werden. Als Repräsentant für objektive Programmiersprachen wird für diese Untersuchung Java gewählt. Damit die untersuchten Tools später eine praktische Verwendung finden, werden gezielt Eclipse-Plugins untersucht. Da die Eclipse Entwicklungsumgebung für viele Betriebssysteme erhältlich ist, lassen sich die Plugins ebenso plattformübergreifend verwenden. Weiterhin zielt diese Untersuchung auf frei erhältliche Tools, bei denen keine kostenpflichtigen Lizenzen erworben werden müssen. Ziel soll es sein, einige der vielversprechensten Tools vorzustellen, zu untersuchen und abschließend ein Tool als Empfehlung anzugeben.

1.3. Vorschau

Der erste Abschnitt dieser Arbeit behandelt den theoretischen Aspekt. Darin werden Metriken vorgestellt die in der Fachliteratur bekannt und anerkannt sind. Es wird gezielt erklärt, wie die Daten erhoben und berechnet werden. Dabei wird eine gedankliche Trennung zwischen den prozeduralen und objektorientierten Metriken vorgenommen. Daraufhin werden die eigentlichen Tools vorgestellt. Dabei werden Fakten über das Tool, sowie die untersuchten Metriken erklärt. Abschließend wird eine erste Bewertung/ Einschätzung gegeben. Diese Einschätzung soll in dem nächsten Abschnitt verfeinert werden, indem die Tools zu einer praktischen Anwendung kommen. Dabei gliedert sich das Kapitel in mehrere Projekte welches jedes untersucht wird. Die erste Bewertung, sowie das Verhalten bei einer praktischen Anwendung, fließen abschließend in das nächste Kapitel ein. Dieses gibt einen vollständigen Blick auf die Tools, bei denen jeweils die Vorteile und Nachteile aufgezählt werden sollen. Abschließend folgt eine Zusammenfassung über die Ergebnisse der Untersuchung und dieser Arbeit.

2. Software-Metriken

Während der Entwicklung von Anwendungen, gibt es die Möglichkeit, den Quellcode durch Software-Tests auf korrekte Funktionsweise zu prüfen. Jedoch kommt man damit schnell an seine Grenzen, wenn der Quellcode zu komplex wird und man ungenügende Testfälle generiert oder sie gar vergisst. Ebenso durch Zeitmangel oder mangelnde Werkzeugunterstützung [Hof08] erschöpfen die Möglichkeiten die Qualität zu erhalten. Ebenso besteht das Problem, dass durch steigende Komplexität der Software-Systeme, sich die Abhängigkeiten der Komponenten untereinander erhöhen. Beginnt man Änderungen an einer Stelle, sind Änderungen bzw. sogar Erweiterungen an anderen Stellen nötig. Die Zahl möglicher Fehler steigt sehr schnell an. Dadurch sinkt wiederrum die Qualität der Software und dies hat folgende negative Eigenschaften als Konsequenz [Wul02]:

- Steifheit: Erweiterungen innerhalb des Software-Systems sind schwer umzusetzen. Wird der Quellcode an einer Stelle verändert, sind auch Änderungen in vielen weiteren Komponenten nötig.
- Zerbrechlichkeit: Diese steht oft in Verbindung mit Steifheit. Wird an einer Stelle etwas geändert zerbricht das System in viele kleine Teile, da in anderen Bereichen neue Fehler entstehen die nichts mit der aktuellen Änderung zu tun haben, so dass diese auch wieder geändert werden müssen. Dadurch entsteht ein lange Abfolge von neuen Problemen.
- Zähigkeit: Komplexe Software-Architekturen machen es schwer den Überblick zu behalten. Die schwierige Architektur und die unsauberen Änderungen am Quellcode, die nicht konform der Architektur gehen, machen weitere Änderungen immer schwieriger und zäher.
- Unbeweglichkeit: Durch eine Vielzahl von Verbindungen mit anderen Komponenten erschwert es das Wiederverwenden. Obwohl diese Komponenten durch ihr Design das System optimal ergänzen würden, ist der Programmieraufwand durch die zu vielen Abhängigkeiten einfach zu hoch. Daher scheint es einfacher, wenn man diese Komponente einfach neu schreibt.

Deshalb ist es wichtig, die nötige Qualität durch ein stabiles und flexibles Software-System zu erhalten oder gar wieder zu erlangen.

2.1. Kriterien an Software-Metriken

Die Qualität von Software ist, rein objektiv, schwer zu messen und zu bewerten, da in jeder Progammiersprache eine Funktion mit unterschiedlich großen Aufwand umgesetzt werden kann. Kann eine Schleife in Java schon mit nur ein paar Zeilen geschrieben werden, ist es in Assembler schon aufwendiger. Obwohl der Quellcode von heutigen Rechnern sehr schnell übersetzt und ausgeführt ist, ist der Aufwand für den Programmierer wesentlich höher.

Während kleine Programme oft schnell mit einem Blick gelesen und verstanden werden, sind größere Programme, die aus mehreren Paketen bestehen, die auch von mehreren Programmierern geschrieben wurden, sehr komplex und beinahe unmöglich komplett zu verstehen. Aus diesem Grund gibt es Software-Metriken, mit Hilfe dieser ist es möglich, feste Kenngrößen quantitativ zu erfassen [Hof08]. Um Software-Metriken erfolgreich und gewinnbringend einzusetzen, sollten diese einige Gütekriterien erfüllen:

- **Objektivität** Gemessene Werte müssen durch Regeln und Fakten gewonnen werden. Sie dürfen nicht durch subjektiven Einflüssen oder reiner Ermessenssache entstehen. Da viele Metriken mit Hilfe von Computer berechnet werden, ist dieses Kriterium beinahe immer erfüllt.
- **Robustheit** Wiederholtes Anwenden einer Metrik auf einen Quellcode muss immer das gleiche Ergebniss zur Folge habe. Es dürfen dabei nicht unterschiedliche Messwerte entstehen.
- **Vergleichbarkeit** Bei verschiedenen Messungen gleicher Kenngrößen sollten die Ergebnisse stets in Relation zu einander stehen.
- **Ökonomie** Metriken sollten stets in Relation zu Kosten-Nutzen stehen und nicht den Aufwand des eigentlichen Nutzen übersteigen.
- **Verwertbarkeit** Werden Software-Metriken eingesetzt, sollten diese auch ausgewertet werden und nicht nur für statistische Zwecke oder Übewachung verwendet werden.

2.2. Vorstellung wichtiger Metriken

2.2.1. LOC und NCSS Metrik

Eine der einfachsten Metriken ist die LoC-Metrik (**L**ines **o**f **C**ode). Der Einsatz dieser Metrik ist einfach und ohne große Hilfsmittel durchführbar. Es wird jede Code-Zeile aus sämtlichen Quelldateien aufaddiert. Das errechnete Ergebnis ist ein Maß für die Programmkomplexität und wird traditionell zur Produktivitätsmessung benutzt [Tha00]. Aus den erhaltenen Werten lässt sich die Produktivität pro Tag oder auch LOC pro Mannmonat ableiten.

Eine Weiterentwicklung der LOC-Metrik ist die NCSS-Metrik (NCSS = **N**on **C**ommented **S**ource **S**tatement). Hierbei werden ebenfalls alle Code-Zeilen aller Quelldateien aufaddiert, dabei jedoch, jedes Kommentar/ jede Kommentarzeile ignoriert und fließen nicht in die Berechnung mit ein. Wird dieser Wert durch den LOC-Kennwert geteilt, erhält man ein Maß für den Dokumentationsgrads des Software-Moduls.

Nachteile

Die Verwendung dieser beiden Metriken ist sehr einfach, haben jedoch den Nachteil, dass sie das Kriterium der Vergleichbarkeit nur unzureichend erfüllen können. Da die Werte sehr von der verwendeten Programmiersprache abhängig sind und eine einfache Umformatierung diese erheblich verändern können. Ebenso wird der Aufwand für das Schreiben des Quellcodes nicht beachtet, wenn mehrere Zeilen gelöscht und neugeschrieben bzw. wieder verändert wurden.

Darüber hinaus sind diese Metriken nur teilweise auf grafische Entwicklungstools wie z.b. bei Visual Studio anwendbar.

Weiterhin führen diese Metriken, in Verbindung als Produktivitätsmessung gebracht, zu einer negativen Motivation, da viele LOC's nicht immer notwendig sind [Dum93].Auch geben diese Metriken keine Aussage über guten/ effektiven Programmierstil. Ein Programm mit 10.000 LOC kann gleiches leisten wie ein Programm mit 100.000 LOC. Diese Aspekte haben jedoch keinen Einfluss auf die Leistungsfähigkeit dieser Metriken und sollten nur am Rande betrachtet werden.

Alternative Zählweisen

Da, wie oben schon erwähnt, das reine Zählen der Zeilen eine schlechte Aussagekraft besitzt, gibt es eine Reihe alternativer Zähltechniken, wie z.B.:

- Es wird nur die Zeile gezählt, die ausführbaren Code enthält.
- Es wird jede Zeile gezählt, die ausführbaren Code und Variablendeklarationen enthält.
- Es wird jede Zeile gezählt, die ausführbaren Code, Deklarationen und Kommentare enthält.
- Es wird jeder Separator einer Anweisung gezählt.

Wenn eine Volumenmetrik wie LOC verwendet wird, sollte vorher unbedingt angegeben werden, welche Zählweise verwendet wird und was alles zur Betrachtung gezogen wird. Da die Metrik für verschiedene Gesichtspunkte herangezogen werden kann, ist es nicht möglich einzelne Zählweisen zu präferieren.

Vorteile

Obwohl solche „Volumenmetriken zur Messung der Software-Komplexität ungeeignet und die ermittelten Größen allenfalls als grobe Eckwerte zu verstehen sind“ [Hof08], besitzen sie einige wichtige Vorteile. Die Messdaten sind sehr schnell erhoben ohne zusätzliche Software und auch mit geringen Aufwand. Sie vermitteln einen schnellen Eindruck über die Grösse des Projekts. Gerade dies wird in Firmen dazu benutzt Software-Entwicklern in einen gewissen Verantwortlichkeitsbereich einzuteilen [Hof08].

Des weiteren dienen die Werte als Ausgangspunkt für weitere Berechnungen komplexerer Metriken.

Um den Aspekt der Vergleichbarkeit zu verbessern, ist es möglich einen Sprachfaktor in die LOC-Metrik zu integrieren. Damit soll es möglich sein, Programmiersprachen übergreifend, erhaltene Werte zu vergleichen. Als Ausgangssprache wird dazu Assembler verwendet und dann mit dem jeweiligen Sprachfaktor multipliziert. Damit lässt sich die Güte der LOC- Und NCSS-Metriken zwar verbessern, prinzipielle Limitierungen aber leider nicht beseitigen [Hof08].

2.2.2. Halstead-Metrik

Die Halstead-Metrik gehört mit zu den bekanntesten und ältesten Metriken. Sie wurde 1977 von Maurice Howard Halstead vorgestellt. Dabei versucht er auf Basis algorithmischer Eigenschaften eines Programms vor allem den Programmieraufwand abzuschätzen [Dum93]. Das Grundprinzip baut sich auf dem Begriff *token* auf und bezeichnet eine grundlegende syntaktische Einheit, wie sie von einem Compiler erkannt werden kann [Tha00]. Diese Menge wird in *Operatoren* und *Operanden* unterteilt. Diese beiden Einheiten bilden das Grundgerüst der Metrik. Als *Operatoren* werden Datentypen, Schlüsselworte, vordefinierte Operatoren, Zuweisungen, Kontrollstrukturen und Präprozessoranweisungen gezählt. Die *Operanden* stellen Konstanten, Listen sowie Bezeichner von Methoden und Variablen dar. Die Halstead-Metrik verwendet folgende 4 Grundeinheiten für die weitere Berechnung:

$$\eta_1 = \text{Anzahl unterschiedlicher Operatoren} \tag{2.1}$$

$$\eta_2 = \text{Anzahl unterschiedlicher Operanden} \tag{2.2}$$

$$N_1 = \text{Gesamtzahl der Vorkommen aller Operatoren} \tag{2.3}$$

$$N_2 = \text{Gesamtzahl der Vorkommen aller Operanden} \tag{2.4}$$

Für ein besseres Verständnis wird folgende Abb 2.1 betrachtet:

```
1  int addieren1(int t)
2  {
3    int x=0;
4
5    for(int i=1; i<=10; i++)
6    {
7     x=t+i;
8    }
9
10   return x;
11 }
```

Operatoren		Operanden	
int	4x	**0**	1x
addieren1	1x	**1**	1x
(), {}	4x	**10**	1x
=	3x	**t**	2x
<=	1x	**x**	3x
+	1x	**i**	4x
++	1x		
for	1x		
return	1x		
;	5x		

Listing 2.1: Addieren von Zahlen

Aus diesen Daten werden die folgenden vier Basiswerte errechnet:

$$\eta_1 = 10 \tag{2.5}$$

$$\eta_2 = 6 \tag{2.6}$$

$$N_1 = 22 \tag{2.7}$$

$$N_2 = 12 \tag{2.8}$$

Als einen weiterführenden Gedanken kann das *Minimalvokabular* berechnet werden. Dazu wird die Größe η_2^* benötigt.

$$\eta_2^* = \text{Anzahl der Ein- und Ausgabeoperanden} \tag{2.9}$$

Damit wird die minimale Anzahl an Operanden beschrieben, die zur Implementierung eines bestimmten Algorithmus benötigt werden [Hof08]. Um ein Ergebnis für den Algorithmus zu erhalten, wird von einem idealen Lösungsweg ausgegangen, indem kein Ergebnis zwischen gespeichert werden muss. Das bedeutet, es gibt nur die Eingangsoperatoren und den Ausgabewert, der von der Methode zurückgegeben wird. Dabei entspricht der Wert von η_2^* genau der Anzahl von Parametern der Methode plus 1. Dem Beispiel von Abb. 2.1 folgend:

$$\eta_2^{*addieren1} = 2 \tag{2.10}$$

Mit den vier Basisgrößen aus (2.1), (2.2), (2.3), (2.4) sowie (2.9) lassen sich nun alle wichtigen Kenngrößen der Halstead-Metrik berechnen. Die erste Kenngröße η bezeichnet die Größe des *Vokabulars*, sie stellt die Summe der unterschiedlichen Operatoren und Operanden dar. *Vokabulargröße*:

$$\eta = \eta_1 + \eta_2 \tag{2.11}$$

Damit ergibt sich für unser Beispiel:

$$\eta^{*addieren1} = \eta_1^{addieren1} + \eta_2^{addieren1} = 10 + 6 = 16 \tag{2.12}$$

Als nächstes folgt η^*, was das *Minimalvokabular* entspricht. Wie in [Hof08] beschrieben stellt η^* alle Operanden und Operatoren dar, die mindestens nötig wären, um einen Algorithmus zu implementieren, der in einer frei wählbaren und fiktiven Programmiersprache

geschrieben ist. Halstead geht davon aus, dass es in dieser Programmiersprache einen Operator Δ gibt, der genau diesen Algorithmus umsetzt. Damit hätte das Programm in etwa folgende Form:

$$Operand_1 = \Delta(Operand_2, ..., Operand_{\eta_2^*}) \tag{2.13}$$

Damit gäbes es nur die 2 Operatoren Δ und $=$, sowie η_2^* Operanden.

$$\eta^* = \eta_2^* + 2 \tag{2.14}$$

Dadurch folgt:

$$\eta^{*addieren1} = \eta_2^{*addieren1} + 2 = 2 + 2 = 4 \tag{2.15}$$

Weitere Kennwerte die Halstead definiert:

Die *Programmlänge N*. Sie stellt die Gesamtanzahl aller Operatoren und Operanden dar.

$$N = N_1 + N_2 \tag{2.16}$$

Wird die Programmlänge mit dem Zweierlogarithmus der Vokabulargröße multipliziert erhält man das *Volumen des Programms*:

$$V = N * \log_2 \eta \tag{2.17}$$

Beide Werte stellen den Umfang des Programmes dar und beziehen sich auf alle Elemente die im Quellcode vorkommen. Ganz im Gegensatz wie zur Zeilenmetrik, bei der nur Zeilen gemessen werden und daher weniger von Form und Schreibstil des Progammierers abhängig ist. Bei dem Volumen wird die Einheit „Bits“ verwendet, mit der Vorraussetzung das Operanden und Operatoren in der gleichen Größe codiert sind. Mögliche Werte für das Volumen sollten, laut [KL07], zwischen 20 und maximal 1000 betragen. Dabei stellt 20 eine parameterlose Funktion mit einer nichtleeren Zeile dar. Bei einem Wert über 1000 ist es sehr wahrscheinlich, dass die Funktion zu viele Aufgaben übernimmt.

Dies bedeutet für die Beispielanwendung:

$$V^{addieren1} = (22 + 12)\ \log_2(10 + 6) = 34\ \log_2 16 = 34 * 4 = 136\,[\text{Bit}] \tag{2.18}$$

Das *Minimalvolumen* (potential volume) V^* stellt die kleinste mögliche Implementierung dar, die in der anfangs genannten idealen Programmiersprache möglich wäre.

$$V^* = \eta^* * \log_2 \eta^* = (2 + \eta_2^*) * \log_2(2 + \eta_2^*) \tag{2.19}$$

In Bezug auf die Gleichung (2.15) folgt:

$$V^{*addieren1} = 4 * \log_2 4 = 8\,[\text{Bit}] \tag{2.20}$$

Der nächste Kennwert ist das *Level L* (program level). Es stellt das Minimalvolumen im Verhältnis zum realen Programmvolumen dar und gibt darüber Auskunft, wie weit der geschriebene Quelltext von der kleinstmöglichen Implementierung abweicht. Im Idealfall würde der Wert 1 erreicht, genau dann, wenn für jedes Problem und Aufgabe eine Funktion der Programmiersprache bereit gestellt wird, siehe (2.13). In der Praxis können solche Level 1 Programme leider nicht erreicht werden. Die Formel zu Berechnung stellt sich folgendermaßen dar:

$$L = \frac{V^*}{V} \tag{2.21}$$

Damit ergibt sich als Beispiel:

$$L^{addieren1} = \frac{8}{136} = 0,0588 \tag{2.22}$$

Da sich der Kennwert L vor allem auf das Minimalvolumen bezieht und dieser ohne den Funktionsaufruf nicht aus dem Quelltext abgelesen werden kann, leitete Halstead einen weiteren Kennwert $\hat{\text{L}}$ ab, Wenn man nicht in der Lage ist V^* zu berechnen, ist es möglich das Level des Programmes anhand des Quellcode selber herzuleiten. Dabei ging Halstead von mehreren Überlegungen aus. Zum einen, dass durch die steigende Anzahl an Operatoren das Level im umgekehrten proportionalen Verhältnis sinkt. Auf der anderen Seite ging er davon aus, dass jede Funktion mindestens 2 Operatoren benötigt und stellte dies durch:

$$\hat{L} \sim \frac{2}{\eta_1} \tag{2.23}$$

in Verhältnis.

Ähnlich wie bei den Operatoren, führt auch das häufigere Verwenden von Operanden zu einem Sinken des Levels. Die durchschnittliche Wiederholungsrate von Operanden wird mittels P berechnet:

$$P = \frac{N_2}{\eta_2} \tag{2.24}$$

Ist jeder Operand nur einmal vertreten, beträgt P = 1 und steigt mit jeder weiteren Verwendung. Dadurch ergibt sich aus (2.23) folgende Beziehung:

$$\hat{L} \sim \frac{1}{P} \tag{2.25}$$

Somit kann aus den Gleichungen (2.23) bis (2.25) folgende alternative Definition für L hergeleitet werden:

$$\hat{L} = \frac{2\eta_2}{\eta_1 N_2} \tag{2.26}$$

Der Unterschied zu (2.21) besteht darin, dass sich $\hat{\text{L}}$ aus den Basisgrössen der jeweiligen Implementierung ergibt und als einen Näherungswert betrachtet werden kann. Für den Beispielcode aus Abb (2.1) ergibt sich nun folgender Wert:

$$\hat{L}^{addieren1} = \frac{2 * 6}{10 * 12} = \frac{12}{120} = 0,1 \tag{2.27}$$

Die nächste Kenngröße stellt den *Schwierigkeitsgrad* (difficulty) D eines Programms dar. Dabei geht der Grundgedanke davon aus, je näher ein Programm einer Level 1 (L=1) Bewertung kommt, umso einfacher ist es aufgebaut. Wird der Kehrwert aus diesem Wert gebildet, erhält man eine Maßzahl über die Schwierigkeit einer Implementierung.

$$D = \frac{1}{\hat{L}} \tag{2.28}$$

Aus (2.22) ergibt sich für das Beispiel folgender Wert:

$$D^{addieren1} = \frac{1}{0,0588} = 17 \tag{2.29}$$

Ein weiterer Wert stellt der *Implementieraufwand* E (programming effort) dar. Er beschreibt den Aufwand der nötig ist, das Programm zu erstellen bzw. zu verstehen.

$$E = \frac{V}{L} \tag{2.30}$$

Für das Beispiel ergibt sich dadurch folgender Aufwand:

$$E^{addieren1} = \frac{V^{addieren1}}{L^{addieren1}} = \frac{136}{0,0588} = 2312 \tag{2.31}$$

Als alternative Berechnung lässt sich die Formel mit Hilfe von (2.21) substituieren, damit erhält man für E:

$$E = \frac{V}{L} = \frac{V}{\frac{V^*}{V}} = \frac{V^2}{V^*} \tag{2.32}$$

Damit zeigt sich, dass der Aufwand, um eine Funktion zu implementieren, mit steigenden Anforderungen (Minimalvolumen), das Code-Volumen im Quadrat ansteigen lässt.

Als Abschluß folgen noch 2 Kenngrössen. Diese ist zum einen die *Implementierzeit* T, welche sich proportional zum Implementieraufwand verhält. Sie berechnet sich aus:

$$T = \frac{E}{S} \text{ [sec]} \tag{2.33}$$

Mit S, als *Stroud Number*, welche Halstead von John Stroud (Psychologe) übernimmt. Dieser definierte „Momente“ als Zeit in der das menschliche Gehirn grundlegende, elementare Entscheidungen trifft. Diese quantifizierte er als 5 bis 20 Entscheidungen pro Sekunde. Das bedeutet je nach Zustand der Person (z.b. die Fähigkeit sich konzentrieren zu können) kann S Werte zwischen $5 \leq S \leq 20$ annehmen. Empirische Untersuchungen haben gezeigt, dass $S = 18$ ein guter Wert für die nötige Zeit in Sekunden ergibt. Damit würde die Funktion aus dem Beispiel folgende Zeit zur Implementierung benötigen:

$$T = \frac{2312}{18} = 128,44 \tag{2.34}$$

Die letzte Kenngrösse ist die *Fehlerabschätzung* $\hat{B}$ (total number of delivered bugs) und gibt eine Schätzung über die Zahl der Fehler in der Implementierung [KL07]. Wie bei (2.33) gezeigt wird, muss ein Mensch pro Sekunde circa 18 Entscheidungen treffen um ein Programm zu schreiben. Als Konsequenz zeigt sich, dass jede Entscheidung die Gefahr einer Fehlentscheidung birgt. Die Fehlerabschätzung versucht eine Abschätzung potenzieller Fehler eines fertigen Programms zu geben und nicht der tatsächlich enthaltenen Fehler.

$$\hat{B} = \frac{V}{3000} \tag{2.35}$$

Damit würde die Funktion aus dem Beispiel zu folgender Anzahl an Fehlern neigen:

$$\hat{B} = \frac{136}{3000} = 0,0453 \tag{2.36}$$

Die Metriken von Halstead gehören mit zu den ältesten die auch heute noch Verwendung finden, auch wenn immer wieder Kritik über die Aussagekraft des Aufwandes oder des Schwierigkeitsgrads geäußert wird. Ein weitere Kritikpunkt der an der Metrik von Halstead geäußert wird ist, dass er alle seine Kenngrössen nur von den Operatoren und Operanden ableitet und daraus die Schwierigkeit oder den Aufwand ableitet [Dum93]. In einigen Programmiersprachen kann es immer zu Ungenauigkeiten bei der Zuordnung von Operatoren und Operanden geben, jedoch hatte Halstead selber keine Regeln zur Einteilung vorgegeben, so dass Ungenauigkeiten in Bezug auf die Metrik einkalkuliert werden müssen. Besonders an dieser Metrik ist die Einfachheit, sobald die Basisgrössen festgelegt sind, lassen sich die Kennwerte durch die Formeln schnell berechnen.

Zusammenfassung

- **Vokabular**

$$\eta = \eta_1 + \eta_2 \tag{2.37}$$

- **Programmlänge**

$$N = N_1 + N_2 \tag{2.38}$$

- **Minimalvokabular**

$$\eta^* = 2 + \eta_2^* \tag{2.39}$$

- **Volumen**

$$V = N \log_2 \eta \tag{2.40}$$

- **Minimalvolumen**

$$V^* = \eta^* \log_2 \eta^* \tag{2.41}$$

- **Level**

$$L = \frac{V^*}{V} \tag{2.42}$$

$$\hat{L} = \frac{2\,\eta_2}{\eta_1\,N_2} \tag{2.43}$$

- **Schwierigkeit**

$$D = \frac{1}{L} \tag{2.44}$$

- **Aufwand**

$$E = \frac{V}{L} = \frac{V^2}{V^*} \tag{2.45}$$

- **Zeit**

$$T = \frac{E}{S} = \frac{E}{18} \tag{2.46}$$

- **Fehler**

$$B = \frac{V}{3000} \tag{2.47}$$

Abbildung 2.1.: Zusammenfassung der wichtigsten Halstead-Metriken

2.2.3. McCabe-Metrik

Die folgende Metrik wurde von Thomas J. McCabe 1976 veröffentlicht. Die zentrale Kenngrösse dabei ist die *zyklomatische Zahl* $V(G)$, die sich aus einem Programm abgeleiteten Kontrollflussgraphen G herleiten lässt. Die zyklomatische Zahl gibt Auskunft über die Komplexität eines Progamms und gibt damit Aufschluss ab wann ein Programm zu unübersichtlich wird bzw. wann eine Methode zu viele Funktionen gleichzeitig übernimmt. Die hohe Bedeutung dieser Metrik zeigt sich vor allem im Bereich der Anwendungsgebiete, neben der reinen Bewertung über die Programmkomplexität bietet die Metrik ebenfalls die Möglichkeit zur Verwendung bei Softwaretestverfahren.

Vorraussetzung für die Berechnung ist ein Programmflussgraph. Aus diesem bestimmt man die Knotenanzahl und Verzweigungen innerhalb des Graphen. Dabei wird die Anzahl der *Kanten* $|E|$, die Anzahl der *Knoten* $|N|$ und die Anzahl der zusammenhängenden *Komponenten* p berechnet. Ergebnis dieser Berechnung sind die linear unabhängigen Programmpfade.

$$V(G) = |E| - |N| + 2\,p \tag{2.48}$$

Da die Maßzahl eine Bewertung darstellt, sind einige Eigenschaften dieser Zahl zu beachten:

1. Die zyklomatische Zahl v(G) ist mindestens 1; $v(G) \geq 1$.
2. v(G) stellt die Anzahl aller linear unabhängigen Programmpfade in G dar.
3. Befehle können, innerhalb eines Knoten E, beliebig verändert oder gelöscht werden, diese haben keinen Einfluss auf den Wert von v(G).
4. Durch Einfügen einer Kante N in den Graphen G erhöht sich auch die zyklomatische Zahl v(G) um Eins. Durch Entfernung einer Kante verringert sich der Wert um Eins.
5. v(G) ist nur von der Struktur von G abhängig.

Wird die McCabe-Metrik während der Entwicklung eingesetzt, hilft sie rechzeitig kritische Programmabschnitte zu erkennen. Laut [Hof08] wird sogar in Firmen oft eine obere Grenze festgelegt, die Programmierer zu beachten haben und später sogar ein Abnahmekriterium darstellen. Eine weitere Bedeutung erhält die zyklomatischen Zahl, laut McCabe, ab einem Wert von 10. Dieser Wert, welche nur eine Empfehlung von McCabe ist, stellt einen Schwellenwert dar, ab dem der Quellcode dazu neigt unübersichtlich zu

werden. Es werden zu viele Aufgaben übernommen und die Wahrscheinlichkeit von Fehlern steigt. Aktuell wird ein Wert von 15 empfohlen [KL07]. Werte darüber hinaus legen nahe an eine Reimplementierung der Funktion zu denken und einige Aufgaben auszulagen. McCabe betont jedoch, dass in seinen Untersuchungen bei Programmierern große Differenzen zwischen Programmierstil und der zyklomatischen Zahl auftraten. Während unerfahrene Programmierer in ihren geschriebenen Quelltexten teilweise Werte zwischen 3-7 erreichten, traten bei erfahrenen Programmierern, die sich auch in effizientem Programmieren auskannten, Komplextitäten zwischen 40 und 50 auf.

Einige Beispielgraphen sollen dem besseren Verständniss helfen.

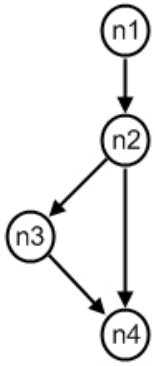


Abbildung 2.2.: *If-Anweisung.*

$|E| = 4, |N| = 4$

$v(C) = 4 - 4 + (2 * 1) = 2$

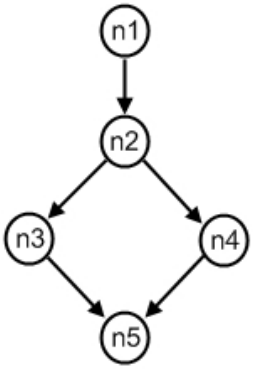


Abbildung 2.3.: *If-Else-Anweisung.*

$|E| = 5, |N| = 5$

$v(C) = 5 - 5 + (2 * 1) = 2$

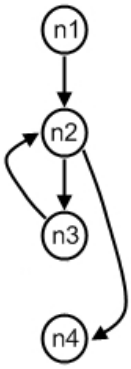


Abbildung 2.4.: *Do-Anweisung.*

$|E| = 4, |N| = 4$

$v(C) = 4 - 4 + (2 * 1) = 2$

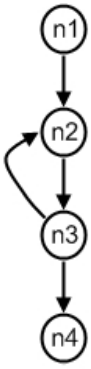


Abbildung 2.5.: *Do-While-Anweisung.*

$|E| = 4, |N| = 4$

$v(C) = 4 - 4 + (2 * 1) = 2$

Eine weitere Aussage von McCabe bezieht sich auf die **Dekomposition**. Während in der ursprünglichen Form $p = 1$ für ein einzelnes Programm gesetzt wird. Erhöht sich dieser Wert wenn der Programmentwurf *dekomponiert* vorliegt. Dabei besteht das komplette Programm aus weiteren Unterkomponenten, d.h. jede Funktion stellt eine Unterkomponente dar. Das bedeutet auch, dass der Kontrollflussgraph G aus einer Menge von Unterpraphen G1,...,Gn besteht. Diese besitzen untereinander keinen Bezug auf andere Graphen. In der Graphentheorie spricht man von einem *Wald*.

Die vollständige Formel lautet:

$$v(G) = \sum_{i=1}^{k} e_i - \sum_{i=1}^{k} n_i + 2k = \sum_{i=1}^{k} v(G_i) \tag{2.49}$$

Dabei stellt G_i die jeweiligen Komponenten k der gesamten Komposition dar. Zusammenfassend bedeutet es, dass für jede Funktion die Einzelkomplexität berechnet und dann mit allen anderen Funktionen aufaddiert wird. Gerade diese Eigenschaft der Kompostion findet Verwendung in der objektorientieren Entwicklung.

Wie anfangs erwähnt findet diese Metrik auch Verwendung bei Testfällen. Dabei ist offensichtlich, bei Funktionen mit einer hohen zyklomatischen Komplexität werden mehr Testfälle benötigt, als bei Funktionen mit niedrigen Werten. Als Orientierung kann man davon ausgehen, es werden mindestens soviele Testfälle benötigt wie die Zahl v(G) angibt.

2.2.4. weitere Bewertungsverfahren

Neben den bis jetzt genannten Metriken gibt es noch eine Vielzahl weiterer Bewertungsverfahren. Diese stammen jedoch auch aus der Zeit in der objektorientierte Programmierung noch kein Begriff war und konnten sich auf lange Sicht gesehen einfach nicht durchsetzen, so dass man zwar in [Dum93] und [Dum92] weitere Metriken finden kann, die z.b. die Metrik von McCabe versuchen zu erweitern, besitzen aber in aktuellen Tools und Publikationen keine hohe Bedeutung mehr.

2.3. Objektorientierte Metriken

Die bisher genannten Metriken zählen zu den klassischen Bewertungsverfahren und haben ihren Stellenplatz schon viele Jahre als traditionelle Metriken inne. Die Umfangsmetrik wie *Lines of Code* oder die Komplexitätsmetrik nach *McCabe* und *Halstead* waren

ursprünglich nach den grundlegenden Prinzipien der imperativen Programmierung entwickelt worden. Diese setzen sich aus einer Abfolge von Anweisungen zusammen die rein sequentiell abgearbeietet werden. Der Vorteil dieser Bewertungsverahren ist, dass sie in ihrer Gesamtheit weiter Verwendung finden und korrekt arbeiten. Jedoch nicht ein Gesamtbild über den kompletten objektorientierten Aufbau abgeben können, da ihnen Teile wie Vererbung fehlen und diese nicht bewerten können. Jedoch verwenden heutige Objektorientierte Programmiersprachen gerade in Klassen- und Objektmethoden weiterhin das imperative Programmierparadigma und gerade dies lässt die klassischen Metriken weiterhin Verwendung finden. Um aber die Aspekte eines Objektorientierten Entwurfs beachten und bewerten zu können, müssen weitere Metriken herangezogen werden und genau darum soll es in diesem Kapitel gehen. Um verwertbare Ergebnisse liefern zu können, ist es wichtig die spezifischen Eigenschaften objektorientierter Programmiersprachen abzubilden. Diese werden durch die Bereiche der Komponenten- und Strukturmetriken eingegliedert [Hof08].

2.3.1. Komponentenmetriken

Die Funktionsweise dieser Metrik ist relativ simpel. Eine *Komponentenmetrik* zerlegt ein Software-System in mehrere Teilkomponente und bewertet diese unabhängig vom gesamten Systen. Dabei ist eine mehrstufige Genauigkeit möglich, ganz abhängig von den angegeben Parametern. Überlichweise stellt eine Klasse eine Teilkomponente dar, in vereinzelten Fällen kann auch ein gesamter Klassenverbund (Paket, Bibliothek etc.) gleichgesetzt werden. Hintergrund dieser Arbeitsweise ist die Möglichkeit diese Teilkomponenten vergleichbar gegenüber zu stellen. Es folgt eine Übersicht gebräuchlicher Komponentenmetriken laut [Hof08].

Abkürzung	**Metrik**	**Typ**	**Signifikanz**
OV	Object Variables	Umfangsmetrik	Hoch
CV	Class Variables	Umfangsmetrik	Hoch
NOA	Number of Attributes	Umfangsmetrik	Hoch
WAC	Weighted Attributes per Class	Umfangsmetrik	Hoch
WMC	Weighted Methods per Class	Umfangsmetrik	Hoch
DOI	Depth of Inheritance	Vererbungsmetrik	Hoch
NOD	Number of Descendants	Vererbungsmetrik	Hoch
NORM	Number of Redefined Methods	Vererbungsmetrik	Hoch
LCOM	Lack of Cohesion in Methods	Kohäsionsmetrik	Fraglich

Tabelle 2.1.: Komponentenmetriken für objektorientierte Architekturen

Wie aus der Tabelle zu erkennen ist, zählen die Metriken OV, CV, NOA, WAC und WMC alle zu der Gruppe der *Umfangsmetriken*. Diese Maßzahlen helfen zur Bestimmung der Klassen, die durch ihren Platzverbrauch (*Datenkomplexität*) oder ihrer Implementierungsläng (*Methodenlänge*) aus der Teilkomponente herausstechen [Hof08]. Dadurch lassen sich Komponente herausfiltern, die einen größeren Aufbau besitzen und vermutlich dadurch auch größere funktionale Bedeutung besitzen. Weiterhin ist es durch diese Metriken mögliche monolithische Klassen aufzudecken, die dann durch ein mögliches Refactoring in mehrere Unterklassen aufgeteilt werden könnten. Bei der Bewertung der Datenkomplexität einer Klasse, gibt die Maßzahl von OV die Anzahl der Objektvariablen, CV die Anzahl der Klassenvariablen und NOA die Summe aus Objekt- und Klassenvariable an. Die anderen beiden Umfangsmetriken geben bei der Aufsummierung jedem Attribut (WAC) oder jeder Methode (WMC) eine weitere Gewichtung. Die gewichtete Attributskomplexität WAC bzw. die gewichtete Methodenkomplexität WMC betrachten jede Klasse einzeln und gehen dabei nicht auf geerbte Attribute oder Methoden ein. Die nächsten beiden Metriken DOI und NOD zählen zu der Kategorie *Vererbungsmetriken* und messen die Eingliederungstiefe einer Klasse in Bezug zum gesamten Software-System. Das bedeutet die Maßzahl DOI gibt die Anzahl der Oberklassen an. Analog dazu gibt NOD die Anzahl der Unterklassen an. Zu tiefe Vererbungen lassen die Struktur sehr komplex werden und erschweren dadurch die Wartbarkeit. Aus diesem Grund dienen beide Maßzahlen als zuverlässiges Warnsignal [Hof08]. Eine weiteres, wichtiges Kriterium bei der Komplexitätsbewertung wird durch NORM gemessen und gibt die Anzahl der reimplementierten Methoden wieder. Bei der Kohäsionsmetrik wird der Verknüpfungsgrad zwischen den jeweiligen Teilkomponenten gemessen. Die Messung wird dabei in mehreren Schritten durchgeführt. Zu Beginn ermittelt die LCOM-Metrik alle Methoden die auf die gleichen Variablen zugreifen und führt sie zu Methodenpaare zusammen. Im nächsten Schritt werden die Methoden herausgesucht die nur eigene Variablen verwenden. Aus diesen beiden Werten wird eine Differenz gebildet und das Ergebnis stellt dann den Kohäsionsgrad dar. Leider wurde laut [Hof08] noch nicht nachgewiesen, dass dieser Wert eine praktische Signifkanz für die Qualitätsbewertung besitzt und ist dadurch in ihrer Verwendung als fraglich einzustufen.

2.3.2. Strukturmetriken

Eine weitere Möglichkeit die Qualität des Quellcodes zu messen besteht in den Strukturmetriken, die vom Prinzip her eine völlig gegenteilige herangehensweise besitzen. Dabei wird nicht jede einzelne Klasse für sich analysiert, sondern es werden alle Klassen in Verbindung gesetzt. Eine wichtige Aufgabe übernehmen dabei *Kopplungsmetriken*, welche die Klassenverbände auf Interaktions- und Vererbungsmuster analysieren [Hof08]. Vor allem treten dabei die Begriffe *Fan-In* und *Fan-Out* in den Vordergrund.

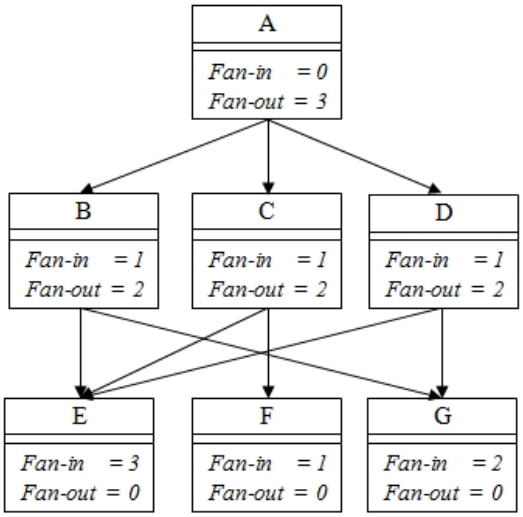


Abbildung 2.6.: Beispiel für Fan-In und Fan-Out

Wie in Abb: 2.6 zu erkennen ist, ergibt sich der *Fan-In* aus der Anzahl der Klassen die direkt auf die aktuelle Klasse zugreifen. Üblicherweise besitzen Klassen am unteren Ende einer Software-Architektur einen höheren Fan-In als Klassen an der Spitze. Der zweite Wert, gemessen an dem *Fan-Out* einer Klasse, gibt die Anzahl der Klassen an, auf die zugegriffen wird, von der ausgehenden Klasse. Diese beiden Maßzahlen legen den Grundstein für viele Metriken. Eine davon ist das *Henry/ Kafura-Maß*. Diese Metrik berechnet die Komplexiät c einer Klasse bzw. einer Komponente. Die Formel lautet:

$$c = \text{modullänge} * (F_{in} * F_{out})^2 \tag{2.50}$$

Wie anfangs erwähnt bezeichnet F_{in} und F_{out} alle möglichen Klassenkopplungen. Der Wert für die Modullänge kann durch eine Komponentenmetrik erfolgen, ohne dabei eine spezifische zu definieren. Die Auswertung der Ergebnisse soll, ähnlich wie bei McCabe, eine Aussage darüber treffen, wie schwierig es ist, die Klassen zu warten, verstehen bzw. zu

ersetzen. Darüber hinaus hilft es bei der Einschätzung welche Rolle diese Klasse einnehmen wird, ob sie eher einer Serverrolle (hoher Fan-out) zugeordnet werden kann oder als Clientrolle (hoher Fan-In). Davon ausgehend entstanden weitere Derivate die eine bessere Korrelation zwischen ihrer Formel und dem Software-System postulierten. Zu erwähnen wäre die Untersuchung von M.J. Shepperd der u.a. die Modulgröße als einen unabhänigen Wert betrachtet. Das *Henry/Kafura-Maß nach Sheppard* einer Klasse M:

$$C(M) = (fan - in(M) * fan - out(M))^2 \tag{2.51}$$

Eine weitere Metrik wird durch Card und Glass beschrieben. Ihre Formel setzt sich aus der Berechnung von Daten- sowie Strukturkomplexität zusammen:

$$v_{CG} = s_{CG} + d_{CG} \tag{2.52}$$

Dabei berechnet s_{CG} die Strukturkomplexität eines Systems mit n Klassen $C_1, ..., C_n$ [Hof08]:

$$s_{CG} = \sum_{i=1}^{n} F_{out}(C_i)^2 \tag{2.53}$$

Wie man aus der Formel erkennen kann, wird für die Berechnung der Strukturkomplexität nur der Fan-Out Wert genommen. Nach Card und Glass wird dem Fan-In Wert keine Bedeutung zugewiesen. Der zweite Teil der Formel berechnet die Datenkomplexität folgendermaßen:

$$d_{CG} = \sum_{i=1}^{n} \frac{IO(C_i)}{F_{out}(C_i) + 1} \tag{2.54}$$

Dabei bezeichnet C_i wieder eine Klasse bzw. eine Teilkomponente und dessen Fan-Out. $I/O(C_i)$ zählt alle IO-Variablen dieser Klasse. Davon ausgehend lässt sich die *relative Systemkomplexität* $\overline{v_{CG}}$ folgendermaßen berechnen, indem man v_{CG} durch die Anzahl der Klassen/ Teilkomponente dividiert. Es ergibt sich:

$$\overline{v_{CG}} = \frac{v_{CG}}{n} = \frac{1}{n} \left(\sum_{i=1}^{n} F_{out}(C_i)^2 + \sum_{i=1}^{n} \frac{IO(C_i)}{F_{out}(C_i) + 1} \right) \tag{2.55}$$

Diese Metrik ermöglicht dadurch vergleichbare Werte, auch bei sich stark in der Größe unterscheidenden Software-Systemen [Hof08], zu erhalten.

2.3.3. Softwarequalitätsmetriken nach Robert C. Martin

Unter dem Namen „Object-Oriented (OO) Design Quality Metrics“ veröffentlichte Robert Martin 1994 eine Abhandlung zur Bewertung von Objektorientierten Software-Systemen. Ziel dieser Abhandlung sollte es sein, mit Hilfe der gegebenen Metriken die Qualität eines Softwareentwurf zu messen und Auskunft darüber zu geben wie flexibel oder starr das System gegenüber Erweiterungen bzw. Veränderungen ist.

Dazu führt er den Begriff der *Stabilität* ein. Stabile Klassen sind gut wartbar und wiederverwendbar. Für Robert Martin gibt es daher 3 Aspekte die die *Stabilität* eines Systems beeinflussen.

- Zum einen gibt es die *unabhängigen Klassen*. die nichts dazu bewegt sich zu verändern, wenn andere Teile des Systems geändert werden. Sie sind auch von keiner anderen Klassen abhängig.
- Zum anderen die *verantwortlichen Klassen*, diese besitzen eine hohe Anzahl von Verbindungen, da viele Klassen von dieser abhängig sind. Wollte man diese verantwortliche Klasse ändern, hätte dies einen enorm hohen Aufwand an Änderungen in allen weiteren Klassen zu Folge.
- Die letzte ist eine Kombination aus beiden. Eine Klasse die von keiner abhängig ist, aber viele Klassen sind abhängig von ihr.

Weitere Begriffe die er definiert, sind *gute und schlechte Abhängigkeiten*. Wie eben erwähnt besitzt eine stabile Klasse die Eigenschaft vor Veränderung geschützt zu sein. Das bedeutet, wenn eine andere Klasse umgeschrieben wird, hat diese keine Konsequenzen für sie. Diese Eigenschaft besitzen die Klassen zu einem gewissen Grad. Wenn eine Abhängigkeit zu einer Klasse mit höherer Stabilität besteht, nennt man dies eine *gute Abhängigkeit*, wenn sie zu einer geringeren Stabilität führt eine *schlechte Abhängigkeit*.

Während bisher bei den Metriken von Komponenten die Rede war, verwendet Robert Martin den Begriff Kategorie. Thematisch bezeichnet er aber die gleiche Definition und zwar die Ebene auf welcher die Metrik verwendet wird. Eine Kategorie kann eine Klasse sein, oder mehrere Klassen z.B. in Form eines Package in Java, teilweise ist auch das gesamte Projekt möglich. Dies ist oft abhängig der benutzten Programmiersprache und Architektur.

Den Grad der Stabilität, die Anzahl der verantwortlichen Klassen, sowie die Abhängikeit der Klassen kann man mit folgenden Metriken messen:

- C_a: Afferente Kopplungen: Die Anzahl an Klassen außerhalb der Kategorie, welche abhängig von Klassen innerhalb der Kategorie sind.
- C_e: Efferente Kopplungen: Die Anzahl an Klassen innerhalb der Kategorie, welche abhängig von Klassen außerhalb der Kategorie sind.
- I: Instabilität:

$$I = \frac{C_e}{C_a + C_e} \tag{2.56}$$

  Der Wertebereich dieser Metrik liegt zwischen $[\,0, 1\,]$. Während I=0 eine vollständig stabile Kategorie bezeichnet, ist I=1 eine maximal instabile Kategorie.

Mit Hilfe dieser Metriken ist man in der Lage, stabile und instabile Kategorien zu bestimmen. Dies ist jedoch nur die eine Seite der Metrik, denn komplett stabile Klassen, hätten ein unveränderliches System zur Folge. Ein Software-System sollte aber veränderlich sein. Aus diesem Grund werden abstrakte Klassen in die Betrachtung der Metrik eingeführt. Robert Martin ist der Meinung, wenn eine Kategorie stabil sein soll, muss sie abstrakte Klassen besitzen. Diese ermöglichen Verbindungen zu anderen Klassen, sind aber stabil genug um Veränderungen zu überstehen. Diese Bedingung hat nun zur Folge, das ein System nicht nur aus abstrakten und stabilen Kategorien bestehen kann. Es sind instabile, aber dafür konkrete Kategorien notwendig. Daher ist es wichtig auch die *Abstraktheit* zu messen:

- N_a: Anzahl abstrakter Klassen innerhalb der Kategorie.
- N_c: Anzahl aller Klassen innerhalb der Kategorie.
- A: Abstraktheit:

$$A = \frac{N_a}{N_c} \tag{2.57}$$

Im nächsten Schritt werden die beiden Kennwerte *Stabilität I* und *Abstraktheit A* in Verbindung gesetzt.

Die beiden äußersten Punkte des Graphen. stellen die besten Konstellationen dar, die in einem Software-System herrschen können. Wie jedoch anfangs erwähnt, sind Kategorien oft nur in einem gewissen Grad stabil oder abstrakt und werden daher als partiell stabil bzw. abstrakt betrachtet, was auf der Hauptsequenz eine ausgewogene Verteilung darstellt. Je weiter man von dieser Hauptsequenz abweicht umso schlechter kann der Zustand des Systems betrachtet werden. Die Fälle A=0, I=0 stellen stabile, aber auch konkrete Kategorien dar, die sehr Fehler anfällig sind. Der gegenteilige Fall A=1, I=1 würde eine komplett abstrakte Kategorie mit keinerlei Verbindungen bedeuten.

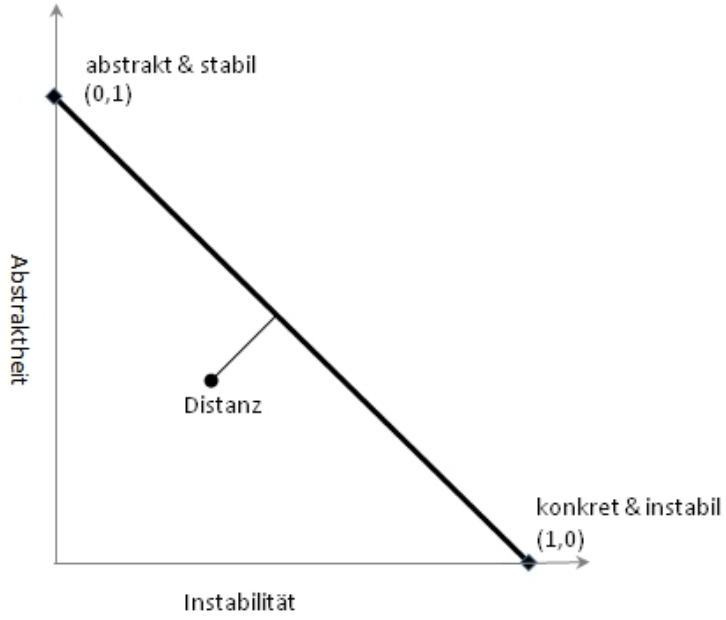


Abbildung 2.7.: IA-Diagramm

Im letzten Schritt beschreibt R.Martin eine Metrik um den *Abstand D* von dieser Ideallinie zu messen, d.h. wie weit die aktuelle Kategorie abweicht.

- A: Abstraktheit
- I: Instabilität
- D: Abstand

$$D = \left| \frac{A + I - 1}{2} \right| \tag{2.58}$$

- D_n: Normalisierter Abstand (vereinfachte Form)

$$D_n = |A + I - 1| \tag{2.59}$$

  Der Wertebereich liegt hierbei zwischen $[\,0, 1\,]$.

Diese Metriken sollen bei dem objektorientierten Design bzw. bei dessen Analyse helfen und Anhaltspunkte liefern das Software-System zu verbessern und sicher auszubauen.

2.4. Weitere Möglichkeiten der Software Analyse

Wie man sicherlich erkennen konnte, gibt es eine Vielzahl von Metriken. Neben den hier vorgestellten Statischen Code-Analyse gibt es Vielzahl weitere Metriken, die begleitend bei der Entwicklung einer Anwendung helfen können. Beginnend bei Prozessmetriken für Zeit- und Aufwandsschätzungen bis hin zu Syntax- und Semantikanalysen. Da diese lediglich Werkzeuge bzw. Hilfsmittel darstellen, ist es wichtig bereits im Vorfeld zu wissen, was man wissen möchte und mit welcher Metrik man dies erreicht.

3. Tools zur automatisierten Analyse

In diesem Kapitel soll es um die Darstellung der zu untersuchenden Tools gehen. In dem folgenden Abschnitt wird der Hintergrund zur automatisierten Analyse eingegangen. In dem darauf folgenden Abschnitt werden die jeweiligen Tools vorgestellt.

3.1. Hintergrund und Möglichkeiten

Wenn man sich mit dem Thema Metriken auseinander setzt, kommt man sicherlich zu dem Schluß, was für ein mächtiges Potenzial die Metriken besitzen. Anhand von Formeln kann man die Schwierigkeit berechnen, die ein Projekt erzeugen kann. Ebenso ist man in der Lage viele weitere Berechnungen anzustellen, sei es die mögliche Fehlerquote abzuschätzen oder ab wann es sinnvoll wäre eine Anwendung komplett neu zu schreiben, anstatt neue Anpassungen weiterhin durch zu führen. Die Ergebnisse sind nicht nur für Programmierer wichtig, Auftraggeber erhalten Kriterien mit denen sie arbeiten können. Wenn man Grenzen und Richtwerte definieren kann, dann kann man immer optimieren und die Effektivität erhöhen. Bei Metriken ist dies leider nur unter gewissen Bedingungen realisierbar. Wenn die Rede von Projekten ist, sind sicherlich nicht 1-2 Klassen gemeint. Beginnt man bei einer der einfachsten Metriken z.b. bei LoC, ist es noch mit Hilfe eines Editors möglich die Anzahl an Zeilen pro Klasse zu zählen. Wenn aber nur noch Kommentare, bzw. leere Zeilen gezählt werden sollen, vor allem bei einer Anzahl von über 50 Dateien, werden einem schnell die menschlichen Grenzen bewußt. Alleine um die benötigten Daten auszulesen, wären einige Stunden an Arbeit nötig, auch mit der Gefahr, übersehener Werte. Es sollte nun klar sein, dass diese Aufgaben niemals sinnvoll und effizient durch einen Menschen übernommen werden können. An diesem Punkt setzen die Anaylse-Tools an und ermöglichen schnelle sowie sichere Auswertungen, selbst größere Projekte. Im Sinne dieser Evaluation müssen einige Kriterien gesetzt werden, die diese Tools erfüllen müssen. Eines der wichtigsten Kriterien ist die kostenlose und freie Verfügbarkeit. Ein anderes Kriterium ist der objektorientierte Ansatz. Er ist sehr wichtig. Daher wurden beispielhaft Tools heraus gesucht und geprüpft die diese Anforderungen erfüllen. Da die Programmiersprache ebenfalls freiverwendbar ist und als Vorbild

für Objektorientierte Programmierung dient, sind die Tools vor allem in der Lage Java-Quellcode zu untersuchen. In dem nächsten Abschnitt geht es darum, die jeweiligen Tools zu benennen und ihr Funktionen vorzustellen, die Metriken aufzuzählen, welche von dem Tool untersucht werden könnnen. Darüber hinaus wird versucht auf Besonderheiten hin zuweisen.

3.2. Nützliche Metrik-Tools

3.2.1. Code Analysis Plugin - CAP

Dieses Tool liegt in Form eines Plugin für Eclipse vor. Es wurde im Jahre 2004 von Johannes Schneider und Matthias Mergenthaler entwickelt und steht bei souceforge zur freien Verwendung bereit. Die letzte Version 1.2.0 wurde in November 2006 veröffentlicht. Das Ziel dieses Tools ist es:

- bei der Bewertung und Verbesserung des Designs zu helfen,
- die „typischen“ Schächen und Fehler einer Architektur zu erkennen, sowie
- die Wartbarkeit, Benutzerfreundlichkeit und die Wiederverwendbarkeit zu verbessern.

Die Installation erfolgt über die Eclipse-interne Updatefunktion oder per Download von Sourceforge. Nach erfolgreicher Installation kann das Tool über das Menü in der Paketübersicht „Show CA“ gestartet werden.

Die ermittelten Werte

Das Tool stützt sich vor allem auf die Metriken von Robert Martin. Dazu misst es die *Anzahl der konkreten Klassen* (CC), *Anzahl der abstrakten Klassen und Interfaces* (AC), die *Anzahl der afferenten Kopplungen* (Ca) sowie der *Anzahl an efferenten Kopplungen* (Ce).

Berechnet werden: Abstraktheit (A), Instabilität (I) und Abstand von der Hauptsequenz (D). Darüber hinaus werden alle Packages nach zyklischen Abhängigkeiten (C) untersucht.

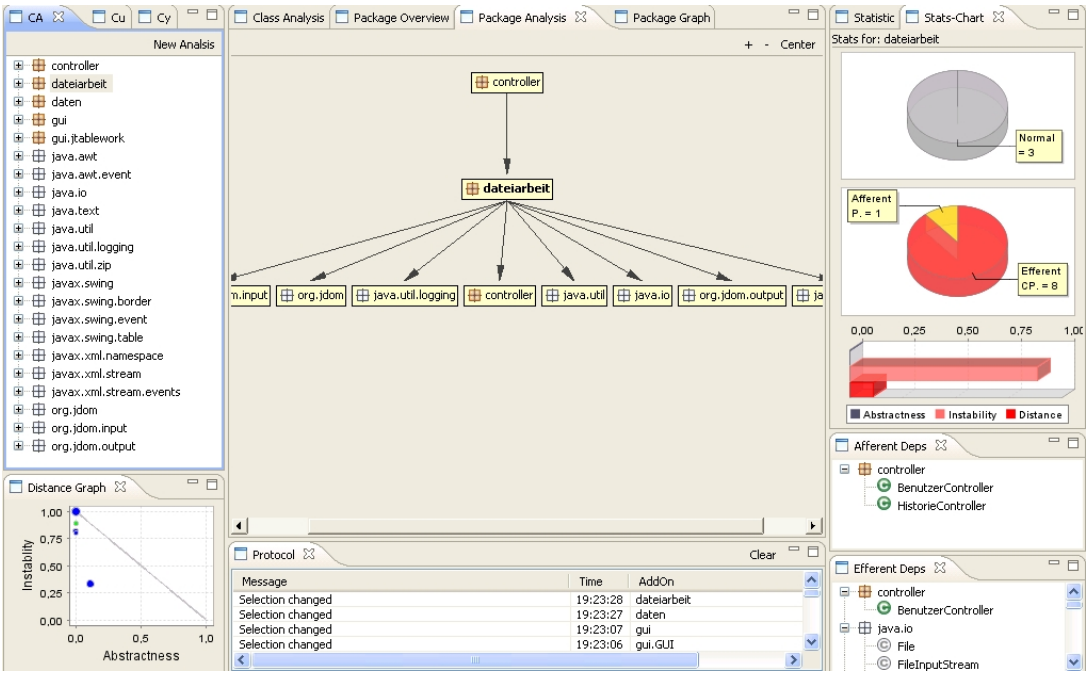


Abbildung 3.1.: Screenshot der CAP-Ansicht

Ausgabe und Präsentation der Werte

Für die Ausgabe der Messwerte wird eine eigene Perspektive generiert, in der beinahe gänzlich auf Tabellen verzichtet wird. Was die Lesbarkeit und Verständlichkeit enorm erhöht. Abbildung 3.1 zeigt eine beispielhafte Darstellung.

Die ermittelten Werte werden auf verschiedene Weise präsentiert. Alle Pakete werden bezüglich ihrer Instabilität und Abstrakheit in einem IA-Diagramm dargestellt. Ein Baumdiagramm in der Mitte stellt die efferenten und afferenten Pakete. bzw. Klassen dar. Auf der rechten Seite werden kategorisch afferente und efferente Klassen gegliedert und aufgelistet. Außerdem werden die ermittelten Daten in einer Statistik gezeigt. Dabei besteht die Wahl zwischen der Darstellung in Zahlen oder in Form von Balken und Tortendiagramm. Ebenso bietet es die Möglichkeit, über die linke Seite, alle zyklischen Verbindungen zu zeigen.

Erste Bewertung

Besonders positiv fällt die grafische Aufarbeitung auf. Man ist sofort in der Lage, sich eine Gesamtübersicht über das komplette Projekt zu verschaffen. Davon ausgehend ist man in der Lage, sich immer tiefer in das Projekt vorzuwagen, bis man auf Klassenebene alle Daten und Kopplungen erkennen kann. Dabei kann man informationsbezogen

suchend vorgehen, das bedeutet, man kann über das IA-Diagramm die Pakete auswählen welche besonders aufällig sind, ebenso aus der Ansicht der efferenten Kopplungen eine Auswahl treffen, beinahe jedes Fenster bietet die Möglichkeit Daten auszuwählen. Es erhöht dadurch sehr die Benutzerfreundlichkeit und Verständlichkeit der berechneten Werte.

Negative Dinge gibt es kaum zu finden. Sehr bedauerlich ist es, dass dieses Tool nur auf die Metriken von Robert Martin bezogen ist und nicht weitere Informationen liefern kann. Es dient ausschließlich zum Messen und Auswerten und bietet keine Möglichkeit um Daten direkt zu verändern. Weiterhin ist der Menüpunkt „Show CA“ schlecht gewählt, er kann schnell übersehen werden und lässt nicht sofort auf CAP schließen, gerade wenn in diesem Menü weitere Funktionen hinterlegt sind.

3.2.2. RefactorIT

Das Tool wurde von der Firma Aqris entwickelt und 2002 in der Version 1.0 veröffentlicht. Es ist mittlerweile in der Version 2.7 (beta) vom Stand 2008 auf sourceforge als Plugin oder Standalone erhältlich. Wie der Name schon andeutet, ist die eigentliche Aufgabe dieses Tools den Prozess der Strukturverbesserung zu automatisieren (Refactoring). Um dies überhaupt möglich zu machen, muss der Quelltext durchsucht und korrekt analysiert werden. Aus diesem Grund greift es auf eine größere Menge von Metriken zurück, weshalb dieses Tool für diese Evaluation sehr interssant wird.

Verwendet man das Plugin für Eclipse, muss die jar-Datei unter „Eclipse/plugins/“ abgelegt werden. Darauf hin, wird Eclipse in der Menüleiste um den Punkt „RefactorIT“ erweitert und in der Paketübersicht um die gleiche Menüfunktion.

Die ermittelten Werte

RefactorIt stellt zum einen eine Vielzahl von Such- und Browsing Funktionen bereit, die in Bezug auf die Neustrukturierung des Quelltextes zielen. Diese besitzen jedoch keinerlei Bedeutung wenn es um Metriken geht und werden daher nicht näher betrachtet. Des weiteren wird die Möglichkeit geboten den Quelltext durch gegebene Metriken zu untersuchen. In dem man über das Untermenü von „RefactorIt“ den Menüpunkt Metrics auswählt, wird einem die Möglichkeit gegeben, Metriken auszuwählen, die man auf den Quellcode anwenden möchte. Wobei diese nach verschiedenen Kategorien sortiert sind, Die in Tabelle (3.2.2) aufgelisteten Metriken wurden vollständig übernommen. In der Kategorie „simple“ sind vor allem die Metriken aus LOC vertreten, ebenso wie die McCabe

Metrik. Bei den „Object Oriented“ Metriken, stehen die Metriken von Robert Martin im Mittelpunkt, Metriken wie „Number of Abstract Types“ oder „Number of Concrete Types“ stellen die Basiswerte für weitere Berechnungen dar. Die letzte Kategorie ist „Quality Metric“ und besteht aus Metriken verschiedenster Art. Es werden dabei nach Zyklischen Verbindungen innerhalb eines Paketes gesucht, wie zyklische Verbindungen zwischen den Paketen, oder ungenutze Parameter einer Methode. Zu vielen Metriken kann man obere und untere Grenzen manuell editieren. Somit lassen sich die Bedingungen zu jedem Projekt besser anpassen. Diese Profile sind abspeicherbar und später für erneuten Gebrauch auswählbar.

simple	**Object Oriented**
CLOC- Comment Lines of Code	A - Abstractness
V(G) - Cyclomatic Complexity	Ca - Afferent Coupling(slow)
DC = CLOC / LOC - Density of Comments	DIT - Depth in Tree
EXEC - Executeable Statements	Ce - Efferent Couling
NLOC - None-Comment Lines of Code	I - Instability(slow)
NP - Number of Parameters	NOTa - Number of Abstract Types
LoC - Total Lines of Code	NOC(1) - Number of Children in Tree
	NOTc - Number of Concrete Types
Quality Metrics	NOTe - Number of Exported Types
CYC - Cyclic Dependencies	NOF - Number of Fields
DIP - Dependency Inversion Principle	NOT - Number of Types
DCYC - Direct Cyclic Dependencies(very slow)	RFC - Response for Class
D - Distance from the Main Squence(very slow)	WMC - Weighted Methods per Class
EP - Encapsulation Principle(slow)	NOA - Number of Attributes
LCOM - Lack of Cohesion of Methods	
LSP - Limited Size Principle	
MQ - Modularization Quality	
NT - Number of Tramps	

Tabelle 3.1.: Auflistung aller Metriken von RefactorIT

Ausgabe und Präsentation der Werte

Innerhalb der standardmäßigen Java Perspektive wird neben der Konsolen Ansicht, eine weitere Ansicht erzeugt, „Metrics“. Geordnet nach Paketen, folgen Spaltenweise die ausgwählten Metriken mit ihren Werten. Werden Grenzewerte überschritten, werden diese Felder rot hervorgehoben. Jedes Paket lässt sich aufklappen und von Klassenebene bis auf Methodenebene alle Messwerte aufschlüsseln, siehe Abb. 3.2.

Neben der Ausgabe in einem Ansichtsfenster, besteht die Möglichkeit gemessene Daten zu exportieren. Diese können dann in verschiedenen Formaten (XML, CSV, TXT, HTML)

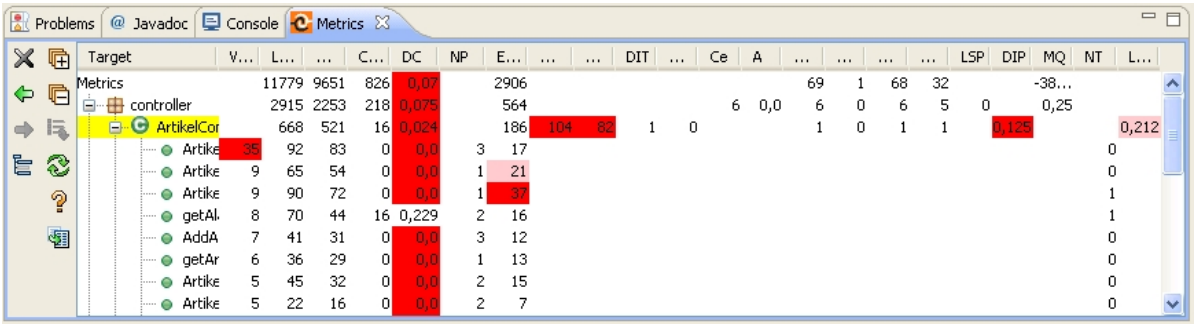


Abbildung 3.2.: Ausgabe der Messwerte durch RefactorIT

abgespeichert werden. Um Daten übersichtlicher auswerten zu können, ist der Export via HTML zu empfehlen, da verwendete Abkürzungen im Kopf der Auswertung in Form einer Legende nochmals aufgeschlüsselt werden, mit den dazu gesetzten Grenzen jeder Metrik. Ergänzend sollte noch die Funktion „Draw Dependencies“ erwähnt werden. Eine Ansicht, in der dynamisch alle Verbindungen von Paket zu Paket eingezeichnet werden und dem Benutzer dadurch den Aufbau des Projektes darstellt. Durch hinzu schalten weiterer Details werden alle Klassen eingeblendet.

Erste Bewertung

Besonders erfreulich ist die hohe Anzahl angebotener Metriken und die Möglichkeit diese selber frei zu wählen. Ebenso Grenzen selber neu zu definieren und diese als Profil zu speichern. Als weitere gute Eigenschaft muss man die Exportmöglichkeit aufzählen, das HTML-Format hilft für besseres Verständnis.

Nachteilig wirkt sich jedoch die Ausgabe der Messwerte aus. Ein kleines Fenster, das bereits wenig Platz bietet, werden die Daten spaltenweise in Zahlen dargestellt. Bei bereits wenigen Metriken ist die Spaltenüberschrift nicht mehr zu erkennen, darüber hinaus enthalten diese auch nur Abkürzungen. Die weitere Ansicht der Abhängigkeiten, ist theoretisch gut. Wenn ein Projekt übersichtlich und durchdacht programmiert ist, bietet diese Funktion sicherlich nützliche Informationen. Bei zu vielen Klassen oder zu vielen Abhängigkeiten der Pakete bzw. Klassen untereinander entsteht nur ein Wust von Pfeilen und Namen.

3.2.3. Swat4j

Swat4j wird von der Firma CodeSWAT als proprietäre Software angeboten. Es wird jedoch eine Evaluatation-Version zum Herunterladen angeboten, die auf maximal 50 Dateien pro Analyse beschränkt ist. Neben Swat4j existiert eine kleinere Version, die auf jegliche Graphen und Diagramme verzichtet und rein auf die Daten ausgerichtet ist. Die aktuelle Version 1.4.0 von Swat4j wurde April 2007 herausgegeben.

Die Installation erfolgt wie alle anderen Tools über 2 Möglichkeiten. Automatisiert kann die Software Update Funktion von Eclipse genommen werden, oder man lädt die Datei über die Webseite herunter und fügt sie dem plugin Ordner von Eclipse hinzu. Danach wird, ähnlich wie bei RefactorIT, ein Menüpunkt in der Menüleiste erstellt und darüber gesteuert.

Die ermittelten Werte

Über das Menü von Swat4j wird die Analyse gestartet. In einem Fenster wählt man entweder das komplette Projekt aus oder die einzelnen Pakete. Durch die Beschränkung auf 50 Dateien, ist dies evtl. nötig. Darauf hin beginnt die Analyse des Projekts in 8 Phasen völlig automatisch. Nach Abschluss, erhält man eine Gesamtübersicht mit den wichtigsten Daten, darauf hin wechselt das Tool in seine eigene Perspektive, die enorme Auswertungsmöglichkeiten bietet. Prinzipiell werden LOC in verschiedenen Formen angewendet. Ebenso die Metriken von Robert Martin, sowie Struktur- und Komponentenmetriken werden von Swat4j untersucht. Dabei geht das Tool Ebenenweise vor, Methoden-, Klassen-, Paket- und Projektebene. Tabelle A.1 im Anhang enthält eine Auflistung aller berechneten Metriken.

Ausgabe und Präsentation der Werte

Obwohl nicht mehr Kennwerte ermittelt werden, als in anderen hier vorgestellten Tools, liegt die Besonderheit in der Aufarbeitung und Verwendung der erhobenen Messwerte. Die Perspektive gliedert sich in 4 große Fenster. Während im oberen Bereich auf der linken Seite die jeweiligen Metriken ausgewählt werden können, wird auf der rechten die grafische Auswertung in Form verschiedener Diagramme dargstellt. Die Gruppierung der Metriken erfolgt ähnlich der Messwerte nach Methoden, Klassen, Pakete oder Dateien. Besonders interessant ist jedoch die Möglichkeit mehrere Metrik gegenüber zu stellen z.B. NLoC im Verhältnis zur zyklomatische Komplexität. Im unteren Bereich der Perspektive

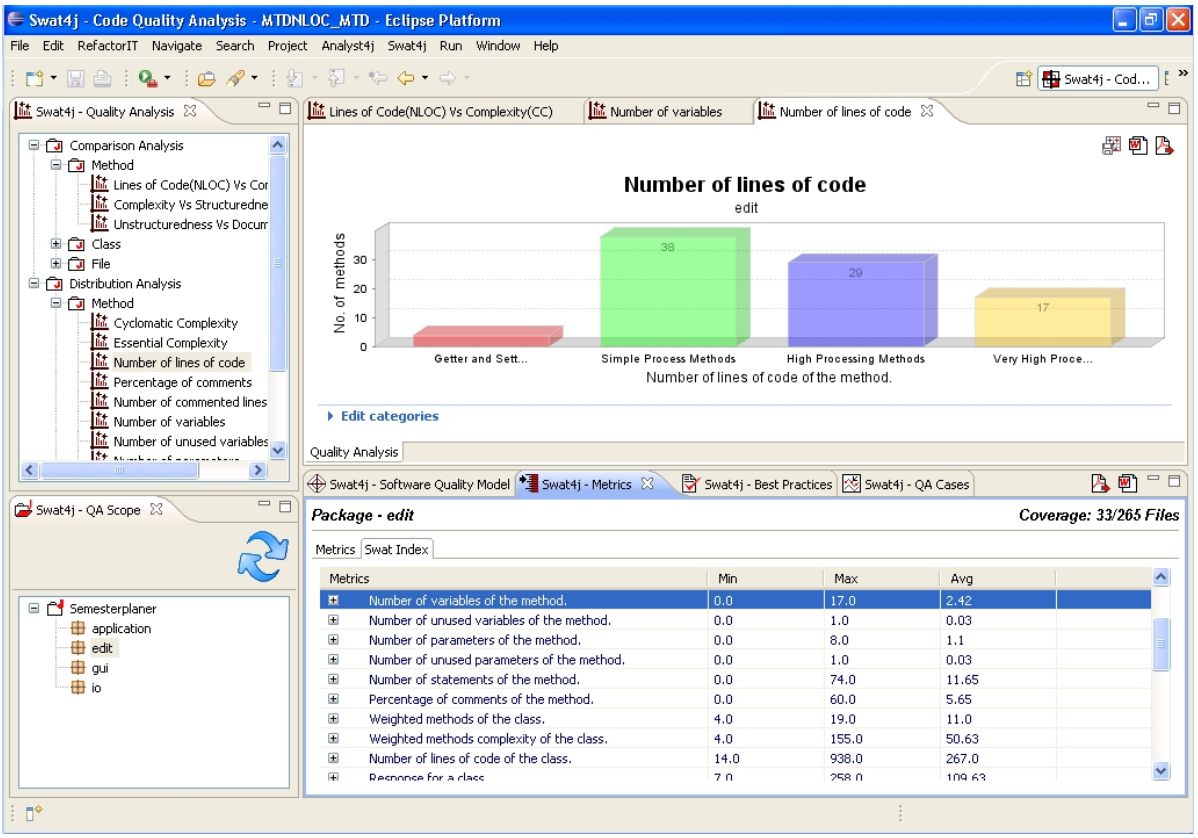


Abbildung 3.3.: Swat4j-Perspektive

ist auf der linken Seite der „Scope“, der das analysierte Projekt darstellt und darüber das Paket ausgewählt werden kann, welches nähere Betrachtung erhalten soll. Auf der rechten Seite sind alle Daten des Tools, auf mehren Tabulatoren verteilt, untergebracht. Eines der Tabulatoren enthält die Messwerte. Präsentiert werden diese in Tabellenform mit Minimum, Maximum und Durchschnittswert. Letztendlich liefern diese aber nur eine Übersicht über das Paket, weniger über die Klassen selber. Auf weiteren Tabulatoren versucht das Tool Bewertungen über Eigenschaften der analysierten Pakete zu geben. Eine Tabulatorseite enthält Bewertungen nach Testbarkeit, Entwurfsqualität, Leistung, Verständlichkeit, Wartbarkeit und Wiedervendbarkeit. Jeder dieser Punkte wieder aufgegliedert und mit Hilfe der Metriken gezeigt, wo die Werte eingehalten wurden und wo überschritten. Ebenso gibt es eine Tabulatorseite bezüglich Dokumentation, toter Code, potenzielle Bugs, Sicherheit, Namenskonventionen und Leistung. Jeder dieser Punkte wird ebenfalls bewertet und der Grad der Erfüllung, grafisch dargestellt. Innerhalb dieses Bereiches werden die Klassen mit den gefundenen Mängeln aufgelistet und sind direkt zugänglich.

Jede der Tabulatorseiten, sowie grafischen Diagramme, sind als PDF- oder RTF-Datei exportierbar.

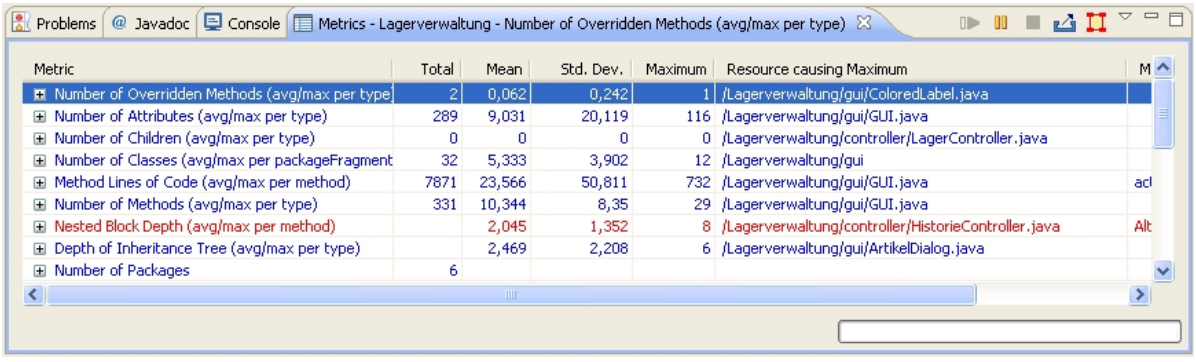
Erste Bewertung

Auf den ersten Blick wirkt das Tools sehr aufgeräumt. 4 Fenster teilen sich die komplette Perpektive. Bei ersten Arbeiten erhält man sehr schnell Diagramme und durch Bewertungen wird man auf Fehler hingewiesen. Das ist jedoch fast nur auf Paketebene betrachtet. Möchte man genau wissen, welche Klasse oder Methode die meisten Zeilen enthält oder zu komplex ist, erhält man darüber leider keine Informationen. Damit lässt sich das Tool beinahe nur für statistische Zwecke verwenden. Es ist ideal um den allgemeinen Zustand eines Projektes zu sehen, ob es einen kritischen Zustand erreicht hat oder noch einzelne Änderungen verantwortbar sind. Es zeigt zwar Fehler bzw. schlechten Progammierstil im Quellcode den man direkt bei der jeweiligen Klasse ändern kann, aber steht dies mehr im Hintergrund.

3.2.4. Metrics v.1.3.6

Dieses Tool gehört zu den etwas kleineren Varianten zur Analyse. Angeboten wird es ebenfalls für Eclipse als Update oder über Sourceforge zum direkten herunterladen. Es liegt aktuell in der Version 1.3.6 vor und wurde Juli 2005 veröffentlicht.

Nach erfolgreicher Installation gestaltet sich das Tool unauffällig. Über die Eigenschaften des jeweiligen Projektes findet man den Punkt „Metrics“. Darin lässt sich das Tool aktivieren. Um eine Veränderung innerhalb des Projektes zu erkennen ist es nötig eine Ansicht für dieses Tool hinzuzufügen. Erreicht wird dies über die Menüleiste „Fenster → Zeige Ansicht → weitere“ dort kann man unter „Metrics“ die Tool eigene Ansicht auswählen. Darauf hin ist das Tool einsatzbereit, siehe Abb 3.4.



Metric	Total	Mean	Std. Dev.	Maximum	Resource causing Maximum	M
Number of Overridden Methods (avg/max per type)	2	0,062	0,242	1	/Lagerverwaltung/gui/ColoredLabel.java	
Number of Attributes (avg/max per type)	289	9,031	20,119	116	/Lagerverwaltung/gui/GUI.java	
Number of Children (avg/max per type)	0	0	0	0	/Lagerverwaltung/controller/LagerController.java	
Number of Classes (avg/max per packageFragment	32	5,333	3,902	12	/Lagerverwaltung/gui	
Method Lines of Code (avg/max per method)	7871	23,566	50,811	732	/Lagerverwaltung/gui/GUI.java	acl
Number of Methods (avg/max per type)	331	10,344	8,35	29	/Lagerverwaltung/gui/GUI.java	
Nested Block Depth (avg/max per method)		2,045	1,352	8	/Lagerverwaltung/controller/HistorieController.java	Alt
Depth of Inheritance Tree (avg/max per type)		2,469	2,208	6	/Lagerverwaltung/gui/ArtikelDialog.java	
Number of Packages	6					

Abbildung 3.4.: Messwertausgabe von Metrics

Die ermittelten Werte

Analysiert werden die meisten Metriken. Der Schwerpunkt liegt sehr bei objekt-orientierten Metriken von u.a. Robert Martin. Ebenso kommt die grundlegende Metrik LOC zur Verwendung. Abbildung 3.2.4 stellt eine Auflistung verwendeter Abkürzungen und Begriffe aller berechneten Metriken dieses Tools dar.

NORM - Number of Overriden Methods	NOF - Number of Attributes
NSC - Number of Children	NOC(2) - Number of Classes
MLOC - Method Lines of Code	NOM - Number of Methods
NBD - Nested Block Depth	DIT - Depth of Inheritance Tree
NOP- Number of Packages	Ca - Afferent Coupling
NOI - Number of Interfaces	V(G) - McCabe Cyclomatic Complexity
TLOC - Total Lines of Code	RMI - Instability
PAR - Number of Paramteres	LCOM - Lack of Cohesion of Methods
Ce - Efferent Coupling	NSM - Number of Static Methods
RMD - Normalized Distance	RMA - Abstractness
SIX - Specialization Index	WMC - Weighted Methods per Class
NSF - Number of Static Attributes	

Tabelle 3.2.: Auflistung berechneter Werte von Metrics

Unter den Fenstereigenschaften, lässt sich dieses Projekt weiter definieren. Neben Einstellungen bezüglich Farbe und Reihenfolge der präsentierten Metriken, lassen sich hier auch die Grenzwerte ändern und den eigenen Bedürfnissen anpassen.

Ausgabe und Präsentation der Werte

Die Daten werden nach jeder Kompilierung mit berechnet und in der Metrics-Ansicht ausgegeben. Dargestellt werden die Ergebnisse in einer Tabelle, jedoch nicht geordnet nach Paketen, sondern nach der berechneten Metrik in Bezug auf das komplette Projekt. In den darauf folgenden Spalten folgen der absolute Wert, Durschnitt, Standardabweichung, Maximum, sowie zugehörig die Datei und Methode welche das Maximum erzeugt haben. Wurden die Grenzwerte der jeweiligen Metrik überschritten, wird diese rot markiert. Jede der aufgelistetes Metriken lassen sich aufklappen und Ebene für Ebene die Ergebnisse aufschlüsseln. Abhängig von der jeweiligen Metrik bis auf Methodenebene.

Neben der Ausgabe der berechneten Messwerte, besteht die Möglichkeit einer grafischen Ausgabe aller Abhängigkeiten dieses Projektes. Die Implementierung dieser Funktion ist in diesem Tool sehr gelungen, sodass es dadurch möglich ist, zyklischen Abhängigkeiten

und detaillierte Informationen über Verbindungen zwischen einzelnen Objekten herauszulesen. Farbliche Trennung und weitere Filterfunktionen ermöglichen die Ergebnisse zu bekommen, die man benötigt. Abbildung 3.5 zeigt eine Beispielhafte Darstellung des Graphen.

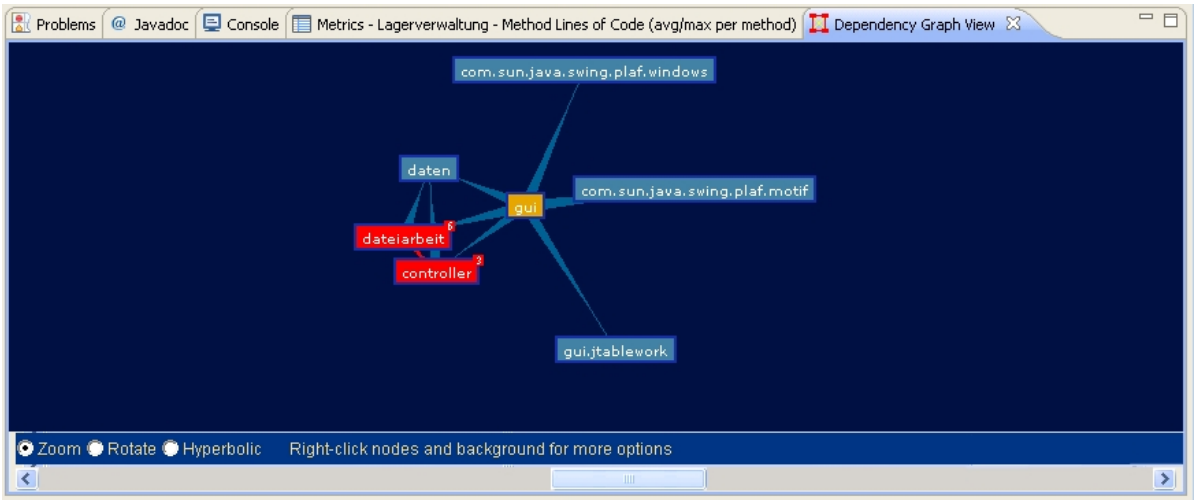


Abbildung 3.5.: Abhängigkeitsgraph von Metrics

Als letzte Möglichkeit lassen sich die Ergebnisse in Form eine XML exportieren.

Erste Bewertung

Die umständliche Installation des Tools und die einfache Darstellung der Daten, lässt ein schlechtes Tool vermuten. Doch bei genauerer Betrachtung zeigen sich durchdachte Funktionen. Die dargestellten Ergebnisse lassen sich bis zu der Klasse/ Methode zurück verfolgen und bearbeiten. Sie stehen an richtiger Stelle, um auch die Informationen zu bekommen die jeweils wichtig sind. Wenige Mausklicks erlauben tiefer in das jeweilige Projekt vor zu dringen und konkreter die Probleme zu erfassen. Obwohl keine Diagramme die Daten aufwerten und darstellen, ist die zusätzliche Funktion des Abhängigkeitsgraphen wirklich hilfreich.

3.2.5. State of Flow - Eclipse Metrics

Das Tool „Eclipse Metrics“ ist ein weiteres Plugin für Eclipse. Erhältlich über die Update-Funktion oder über sourceforge, gehört es zu den neueren Tools. Aktuell liegt es in der Version 3.12. vor und wurde Juni 2009 veröffentlicht. Entwickelt wurde es von Lance

Walton für State of Flow. Einzige Aufgabe dieses Tools ist es, bei jeder Kompilierung des Quellcodes diesen nach definierten Metriken zu analysieren.

Die Installation erfolgt sehr ähnlich dem bereits vorgestellten Tool Metrics, auf welches der Autor auf seiner Webseite auch verweist. Nach erfolgreicher Integration in Eclipse, wird EclipseMetrics über die Projekteigenschaften und dem Punkt „Metrics“ mit „Enable Metrics Gathering“ aktiviert. Daraufhin starten ein Kompilierungsvorgang, während Eclipse Metrics jede der Javadateien analysiert.

Die ermittelten Werte

Jeweils ein Set von Metriken wird auf Methoden und Klassen angewendet. Eine Auflistung dieser Metriken enthält Tabelle 3.2.5.

Methoden:	**Klassen**
Feature Envy	Efferent Couplings
Lines of Code	Lack of Cohesion
McCabe Cyclomatic Complexity	Number of Fields
Number of Levels	Weighted Methods per Class
Number of Locales in Scope	
Number of Parameters	
Number of Statements	

Tabelle 3.3.: Aufflistung berechneter Metrik von EclipseMetrics

Wie zu erkennen ist, wird nur eine geringe Anzahl an Metriken verwendet, im Vergleich zu den bereits vorgestellten Analyse Tools.

Ausgabe und Präsentation der Werte

Die Präsentation der Ergebnisse gestaltet sich noch einfacher. Ein roter Marker ergänzt die Symbole von Eclipse in der Quellcodeansicht, siehe Abb. 3.6

Wird dieser von dem Mauszeiger berührt gibt er Informationen zu der betreffenden Methode oder Klasse wieder. Darüber hinaus, werden überschrittene Grenzwerte als eine Warnung von Eclipse in der „Probleme“ Ansicht ausgewiesen.

Um die Möglichkeit zu besitzen eine Gesamtübersicht über das komplette Projekt zu erhalten, bietet das Tool eine Export Funktion. In Form von CSV oder HTML generiert es die Daten mit Balkendiagramm und allen Metriken.

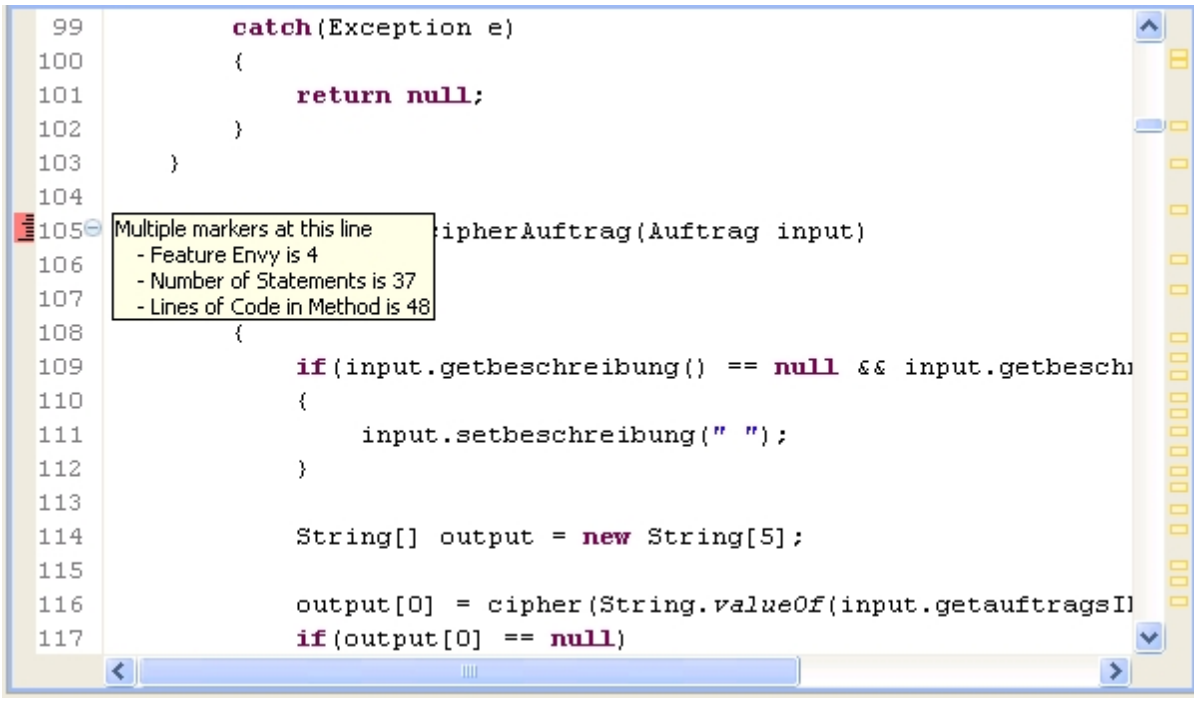


Abbildung 3.6.: Ausgabe der Messergebnisse von EclipseMetrics

Erste Bewertung

Ein kleines Tool, mit eingeschränkten Möglichkeiten. Es bietet wie die meisten Tools die Möglichkeit obere Grenzwerte zu definieren, aber mehr auch nicht. Als hilfreich kann das Tool direkt während der Entwicklung betrachtet werden, da nach jeder Kompilierung die Informationen direkt in Nähe der Klasse/ Methode abrufbar sind und man erkennt sofort, wenn während der Entwicklung Grenzwerte überschritten werden. Um sich im Nachhinein einen Gesamtüberblick des Projektes zu verschaffen ist es sicherlich nicht geeignet. Die Export Funktion zeigt Balkendiagramme, die jedoch keine Rückschluß liefern können, da keine offensichtlicher Bezug zwischen den beiden Achsen zu erkennen ist. Eine Tabelle am Fuße der generierten Webseite gliedert Metrik zu Paket mit Zahlen auf, die informativ aber nicht hilfreich sind.

3.2.6. JDepend - JDepend4Eclipse

Abschließend ist JDepend und JDepend4Eclipse zu erwähnen. JDepend ist ein sebstständige Java-Anwendung, welche von ClarkWare in der Version 2.9 angeboten wird. Von den Signaturen der angebotenen Dateien ausgehend, wurden diese Dezember 2004 erstellt. Um JDepend starten zu können, ist es nötig, den CLASSPATH zu dem Ordner zu definieren, in dem alle Datei extrahiert wurden. Darauf hin kann man per Konsole

die Javaanwendung starten. Dabei besteht die Möglichkeit, eine grafische-, textuelle- oder XML-Oberfläche zu Nutzen. Im Falle dieser Untersuchung wurde das JDepend Projekt in Eclipse geladen. Bei dem Aufruf der Anwendung wird der Pfad zu dem zu untersuchenden Projekt als Argument übergeben.

JDepend4Eclipse bietet gleiche Funktionalität wie JDepend, ist jedoch ein Plugin für Eclipse und bietet eine eigene Perpektive die eine bessere grafische Ausgabe und einfacherere Handhabung ermöglicht. Die Installation erfolgt über das Eclipse eigene Software-Update oder zum herunterladen auf der Webseite von Andrei Loskutov. Die aktuelle Version 1.2.1 wurde Sptember 2007 veröffentlicht.

Die ermittelten Werte

Beide Tools richten sich ausschließlich nach den Metriken von Robert Martin. Tabelle 3.2.6 stellt eine Auflistung der berechneten Werte dar.

CC	- Concrete Class Count	A	- Abstractness
AC	- Abstract Class/ Interfaces Count	I	- Instability
Ca	- Afferent Couplings	D	- Distance from the Main Sequence
Ce	- Efferent Couplings	V	- Volatilty
		C	- Dependency Cycle

Tabelle 3.4.: Auflistung berechneter Metriken von JDepend/ JDepend4Eclipse

Wie zu erkennen ist, werden die konkreten Klassen und abstrakten Klassen gezählt, die afferenten und efferenten Kopplungen und davon ausgehend die Metriken berechnet. Zusätzlich versucht das Tool zyklische Abhängigkeiten zu erkennen. Weitere Berechnungen werden nicht angeboten.

Ausgabe und Präsentation der Werte

Obwohl man in JDepend von 3 verschiedenen Oberflächen wählen kann, ist die Formatierung annähernd identisch. Beispielhaft dient Abb. 3.7. In der grafischen Oberlfäche steht ein einzelnes Fenster zur Verfügung, die obere Hälfte listet efferente Kopplungen und die untere Hälfte afferente Kopplungen auf. Geordnet nach jedem Paket werden die Daten, in Klammern und durch Semikolon getrennt, dahinter aufgelistet (siehe Abb. 3.7).

JDepend4Eclipse erzeugt für die Ausgabe der Daten eine eigene Perpektive in denen die gleichen Daten jedoch übersichtlicher sortiert sind. Diese lassen sich sogar, als XML- oder TXT-Datei exportieren.

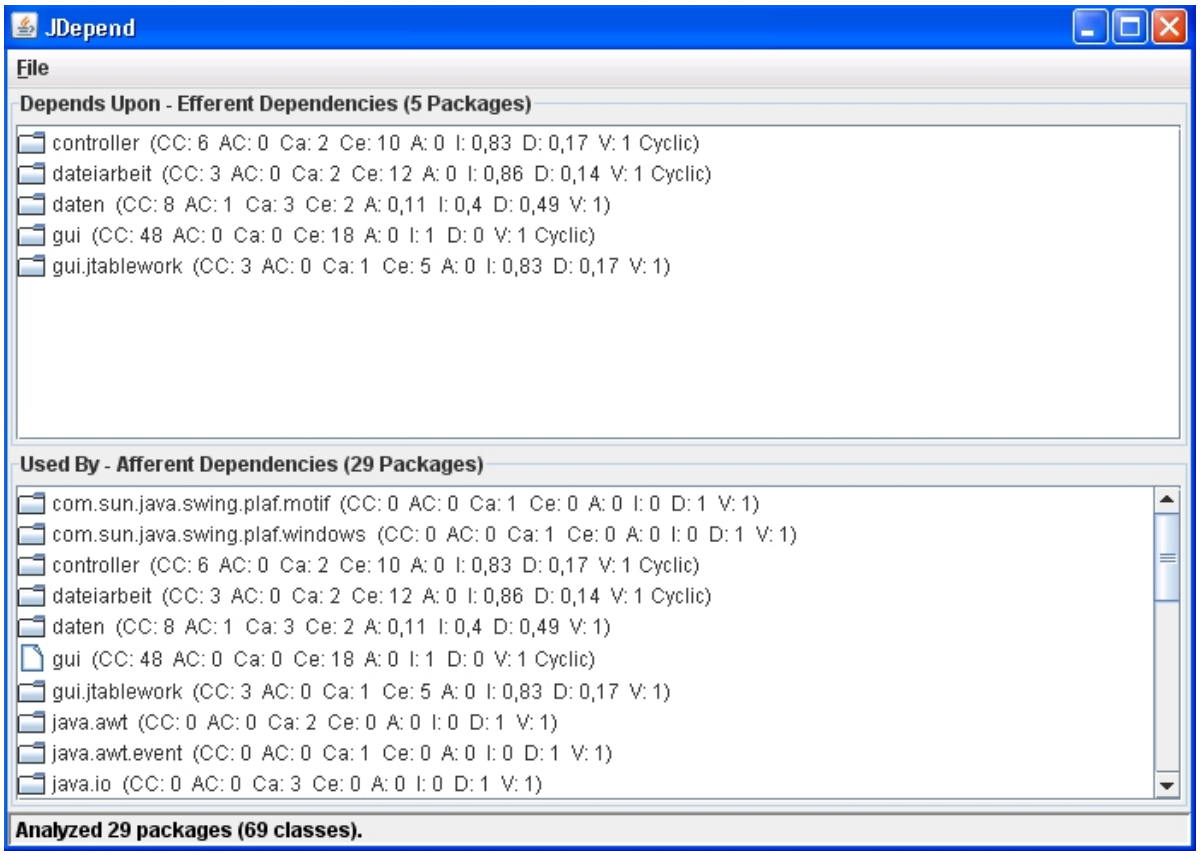


Abbildung 3.7.: JDepend grafische Oberfläche

Erste Bewertung

Das eigentliche *JDepend* ist umständlich einzurichten und zu starten. Der Einfluss eines Linux/ Unix Programmierers ist eindeutig zu erkennen. Selbst wenn JDepend einsatzbereit ist muss als Argument der Pfad zu dem zu untersuchenden Projekt jedes Mal angegeben werden. Wenn dieses Projekt mit Hilfe einer grafischen Entwicklungsumgebung geschrieben wurde, ist es nicht einfach den korrekten und oft enorm langen Pfad zu finden. Die verwendeten Metriken sind unzureichend und betrachten das Projekt sehr einseitig. Die Darstellung der Daten, ist eine reine Auflistung, die für rein statistische Zwecke ausreichend sein mögen, aber für effektives Arbeiten keinesfalls.

JDepend4Eclipse besitzt, Dank der eigenen Perspektive in Eclipse, eine wesentlich übersichtlichere Darstellung. Außerdem sind die Dateien direkt zugänglich aus dieser Ansicht und können sofort verändert werden. Dieses Plugin dient wesentlich besser die Struktur eines Projektes, mit Hilfe der Metriken von Robert Martin, zu verbessern. Ein offensichtliches Manko besitzt dieses Plugin jedoch. Es ist nicht möglich, ein komplettes Projekt zu analysieren. Jedes Paket muss einzeln untersucht werden. Um dies zu umgehen, sollten

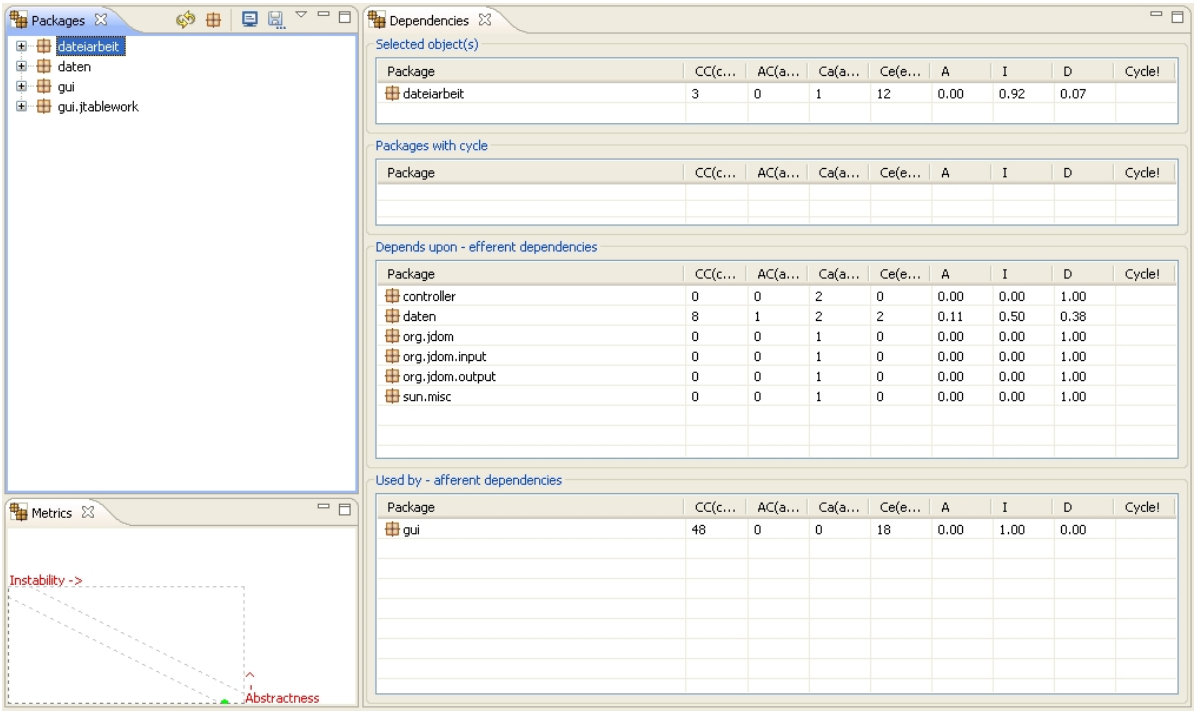


Abbildung 3.8.: Oberfläche von JDepend4Eclipse

alle Pakete mit Hilfe der Strg-Taste ausgewählt werden, erst dann werden mehrere Pakete untersucht und in die Perspektive mit aufgenommen.

3.3. weitere Tools für Code-Qualität

Ergänzend seien noch 2 weitere Tools erwähnt. Zum einen *Eclipse Checkstyle Plug-in* und zum anderen *PMD*. Diese beiden Tools sind recht häufig im Internet zu finden, obwohl sie keine Metriken berechnen und den Aufbau von Projekten analysieren, stellen sie eine große Hilfe in Bezug auf Quellcodequalität dar. Sie erweitern die Regeln von Eclipse und die gefunden Warnungen und Fehler und listen diese zusammen in der Fehleransicht auf. Während Eclipse nur Syntaktische Fehler erkennt, um eine korrektes kompillieren zu gewährleisten. Gehen diese beiden Tools einen Schritt weiter und versuchen auf semantische Fehler hinzuweisen. Als Kernpunkte zählen:

- potenzielle Fehler (z.B. leere try, catch oder switch Anweisungen)
- toter Code (z.B. ungenutzte Variablen, Parameter und Methoden)
- unvorteilhafter Code (z.B. verschwenderische Verwendung von String-Konstrukten)
- komplizierte Ausdrücke (z.B. unnötige If-Anweisungen um Schleifen zu simulieren)

- doppelter Code (vor allem durch Copy-and-Paste entstandene Fehler)

Checkstyle gewann 2007 die Eclipse Community Auszeichnung in der Kategorie „Bestes Open Source Eclipse-basierendes Entwicklungstool“. Da häufig wiederkehrende Fehler, gerade von unerfahrenen Programmierern, abgefangen werden können und so teilweise langes suchen von Fehlern unterbinden können.

4. Fallbeispiele und Untersuchungen

Dieses Kapitel soll die Leistungsfähigkeit ausgewählter Anaylse-Tools überprüfen. Dazu werden einige Tools ausgewählt, die in dem vorherigen Kapitel vorgestellt worden und, sofern es möglich ist, miteinander verglichen. Dabei soll überprüft werden inwiefern die erhobenen Daten verlässlich sind oder es Abweichungen gibt und wie hoch der Anaylseumfang auf das gesamte Projekt ist. Ebenso wichtig ist die Dauer der Analyse und die Fehlertoleranz. Dazu werden 3 Anwendungen verwendet, die sich in ihrer Architektur und ihrem Programmierstil komplett unterscheiden.

- **Anwendung 1**: *Filmdatenbank*: stellt ein kleines Projekt dar, das nur aus 1 Datei besteht und darin alle Funktionalitäten vereint [Appa].
- **Anwendung 2**: *Lagerverwaltung*: stellt ein durchschnittliches Projekt dar, es gliedert sich in mehrere Pakete mit jeweils ein paar Klassen [Appb].
- **Anwendung 3**: *Semesterplaner*: stellt ein überdurchschnittlich großes Projekt dar, mit mehreren hundert Java Dateien [Appc].

Die Untersuchung wird Projektweise vorgenommen und gliedert sich in mehrere Phasen. Während der *Planung* werden relevante Kriterien für das jeweilige Projekt bestimmt und Überlegungen angestellt, welche Informationen wichtig sein könnten, Innerhalb der *Messung* werden Daten gesammelt und diese dann während der *Auswertung* verglichen und bewertet.

Anmerkung: Die verwendeten Untersuchungsobjekte [Appa], [Appb], [Appc] sind nicht vom Autor erstellt worden, sondern sie wurden dem Autor mit der Aufgabenstellung zur Verfügung gestellt. Die namentlichen Angaben zu den Erstellern wurden aus datenschutzrechtlichen Gründen anonymisiert.

4.1. Filmdatenbank

Die erste Anwendung ist ein sehr einfaches Projekt. Die Funktion bezieht sich auf das Auslesen einer XML-Datei und diese Informationen zu persistieren in Form einer PDF

Datei oder eines HTML-Dokuments. Neben einigen externen Bibliotheken die benötigt werden, besteht das Projekt aus einem Standard Paket, dass 2 Java-Klassen (GUI, SAX) enthält. Die GUI.java besteht, laut Eclipse, aus 621 Zeilen. Alle weiteren Informationen sollen mit den Analyse-Tools herausgesucht und miteinander verglichen werden.

4.1.1. Planung

Auffällig sind die hohe Anzahl an Importen, per Eclipse 46 Stück. Davon werden 4 als ungenutzte Importe ausgewiesen. Um die zyklomatische Zahl korrekt auswerten zu können, wird eine Methode als Repräsentant ausgewählt. Folgende Fragen sollen untersucht und wenn möglich gemessen werden.

- Wieviele Abhängigkeiten besitzt diese einzelne Klasse?
- Wie lässt sich das zeitliches Verhalten nach auslösen der metrischen Analyse beurteilen?
- Statistische Werte: Anzahl Zeilen, kommentierte Zeilen.
- Exemplarische Berechnung der zyklomatischen Komplexität bei getJButton3() (siehe Anhang A
- Inwiefern werden die Kriterien von Robert Martin innerhalb des Projekts erfüllt?
- Können zyklische Abhängigkeiten gefunden werden?

4.1.2. Messergebnisse

	LoC	**CLoC**	**Ca**	**Ce**	**I**	**A**	**V(G)**
CAP	x	x	0	13 Pakete/ 43 Klassen	1	0	x
RefactorIT	620 (399)	122	0	11 Pakete	1	0	5
SWAT4j	398	122	x	x	0	0	(1,76)
Metrics	398	x	0	1 Paket	1	0	2 (1,76)
EclipseMetrics	x	x	x	40 Klassen	x	x	x
JDepend4Eclipse	x	x	0	14 Pakete	1	0	(11)

Tabelle 4.1.: ausgewählte Messwerte

RefactorIT:

EXEC (executable-statements) des Projekt: 105.

EXEC (executable-statements) von getJButton3(): 5.

NOT (number-of-types): 11 (davon 0 abstrakt).

SWAT4J:

Number of anonymous classes of the project: 9 interfaces.

Number of functions of the project: 37.

Metrics:

Number of Attributes: 18.

Number of Methods: 27.

Weighted Methods per Class: 50 (GUI: 39, SAX: 11).

Während der Untersuchung, war es nicht möglich detaillierte Informationen durch Swat4j zu erhalten, einzig eine grobe Übersicht mit Durschschnittswerten steht zur Verfügung, bevor das Tool in eine leere Übersicht wechselt. Da Durchschnittswerte nicht auf konkrete Klassen anwendbar sind, können nur beschränkt Informationen von diesem Tool bezogen werden.

Die Tools CAP und JDepend bieten keine Möglichkeit, Zeilen zu zählen. EclipseMetrics gibt sie nur für vereinzelte Methoden an. RefactorIT liefert zum einen die gesamten Lines of Code der Datei und zum anderen auch Non-Commented Lines of Code. Metrics hingegen liefert neben den LoC noch LoC pro Methode. Zu jedem Tool wurden die wichtigsten Information zusätzlich aufgelistet. Die Analysedauer ist bei allen sehr kurz, da Swat4j jedoch immer 8 Phasen der Analyse durchläuft, braucht es von allen am längsten.

4.1.3. Auswertung und Interpretation

Ein großes Problem ist die Vergleichbarkeit der Messwerte, da durch die Tests zwei Dinge ersichtlich wurden. Zum einen verwenden die Tools zwar die gleiche Bezeichnung, besitzen aber unterschiedliche Definition was gemessen wird z.B. bei den Lines of Code. Des Weiteren werden die Messungen auf unterschiedlichen Ebenen angewendet. Während RefactorIT die zyklomatische Zahl nur für Methoden berechnet, geben andere Tools Durchschnittswerte für die jeweilige Klasse oder das jeweilige Paket an. Dadurch wird eine ausführliche und verständliche Auswertung enorm erschwert.

Am ausführlichsten bei den *LoC* ist RefactorIT, da es neben den beiden wichtigen Kategorien LoC und CLoC, weitere statistische Daten ausgibt. So unterscheidet es zwischen

reinen geschrieben Code und nur erzeugten Leerzeilen. Dadurch ist ersichtlich, dass alleine 99 von 620, leere Zeilen darstellen. Das bedeutet, das 16% der gezählten Zeilen nur für optische Zwecke enthalten sind und dadurch den Zeilenumfang enorm vergrößern. Ebenso stellt es die Anzahl der Kommentare dem geschrieben Quellcode gegenüber, worüber sich der Grad der Kommentierung innerhalb der Quellcode gut abbilden lässt. Sollte dieser Wert zu klein sein, wird schnell ersichtlich, dass nötige Erklärungen zu Methoden dringend fehlen. Die beiden Tools JDepend und CAP sind vor allem auf die Analyse derArchitektur ausgelegt und ermöglichen keine Auswertung von LoC. EclipseMetrics erschwert die Auswertung, da die Informationen nur vereinzelt an den Methoden stehen und diese nicht durchgängig sind. Die letzten beiden Tools Swat4j und Metrics lesen nur die wichtigsten Daten aus, dabei zählen sie nicht den reinen LoC sondern nur die Zeilen in denen tatsächlich etwas geschrieben wurde.

Betreffend der *afferenten und efferenten Kopplungen* gibt es ebenfalls unterschiedliche Messungen, wie anfangs erwähnt, können über Eclipse 46 Importe gezählt werden, davon werden 4 als ungenutzt deklariert. Bei näherer Betrachtung sind doppelte Importe zu erkennen, in dem anfangs alle Klassen eines Paket per Wildcard importiert werden und später Klassen aus diesem Paket einzeln importiert werden. So dass insgesamt 13 Pakete importiert werden. Mit diesem Wissen vorausgehend, lassen sich die Messwerte aus Tabelle 4.1.2 erklären.

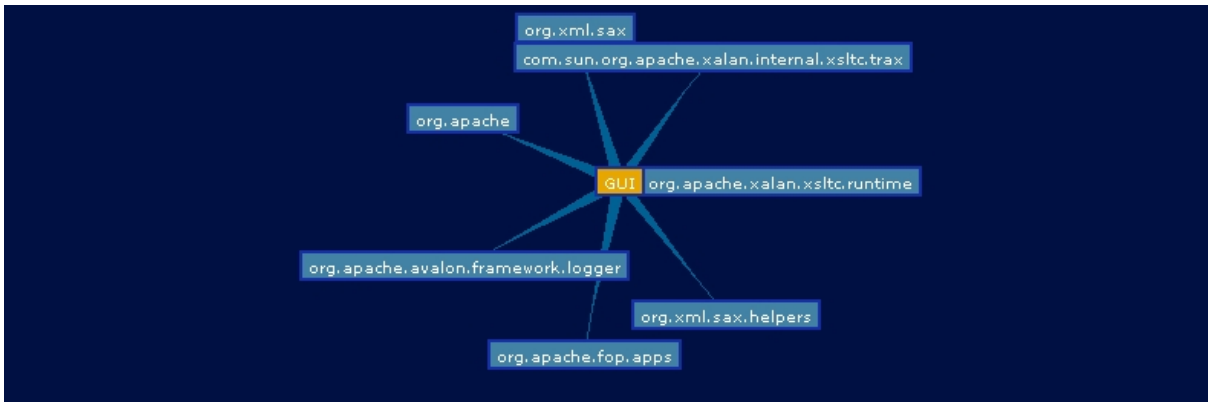


Abbildung 4.1.: Externe Kopplungen (Metrics)

Da die Filmdatenbank nur aus einer Java-Datei besteht, besitzt sie keinerlei efferente Verbindungen dies haben alle Tool erfolgreich erkannt. Afferente Kopplungen werden durch die Importe erzeugt, die von jedem Tool mitberechnet werden. Das Code Analysis Plugin zählt 43 Pakete, 3 Importe wurden mit Wildcards erzeugt in denen komplette Pakete importiert wurden, als einzelne Klassen, diese scheint CAP nicht mit zuzählen, dadurch ergeben sich auch nur 43 Klassenimporte. RefactorIT zählte 11 efferente Kopplungen

was sehr gut ist, da es die tatsächlich benutzten Pakete zählte und nicht alle und somit überflüssigen Importe. Während Swat4j diese Metriken nicht anbietet, zählt Metrics genau 1 Kopplung und zwar das „default package“ somit lässt sich vermuten, das Metrics System-Pakete/Importe ignoriert und nur die Verbindungen bzgl. des eigentlichen Projektes betrachtet, dies müsste in einem der anderen beiden Projekte untersucht werden. Über den Abhängigkeiten-Graph lassen sich zumindest alle externen Bibliotheken darstellen. siehe Abb. 4.1. Bleiben die letzten beiden Tools übrig. EclipseMetrics zählt 40 Klassen ohne weitere Angaben des Tools, lässt sich nur vermuten, dass es ungenutzte Importe und Importe mit Wildcards nicht mitzählt. JDepend4Eclipse lässst eine Analyse nur dann zu, wenn das Projekt aus Paketen besteht, das von Eclipse erzeugte Standard Paket zählt nicht dazu, so dass eine kleine Änderung im Quellcode nötig war, um den Test durchführen zu können. Danach zählte das Tool 14 efferente Kopplungen, da es jedoch nur 4 Pakete in der Ansicht ausgab, liegt die Vermutung nahe, dass JDepend alle Importe zählt und das zusätzlich erzeugte Paket, damit der Test durchgeführt werden konnte. Somit ist dieses Tool sehr unhandlich, wenn bei kleineren Projekten auf Pakete verzichtet wird. Darüber hinaus misst es ungenau, da es nicht alle gezählten Pakete bei efferenten Kopplungen auflistete.

Bezüglich der *Instabilität* und *Abstraktheit* muss gesagt werden, dass es durch die fehlende Auswertung von Swat4j keine korrekte Angabe der Werte gab. Eclipse Metrics bietet diese Metriken nicht an, während alle anderen Tools korrekt und zuverlässig die Werte errechneten.

Weit interessanter stellen sich die Werte bei der *zyklomatischen Komplexität* dar. Die Tools CAP und EclipseMetrics bieten diese Berechnung nicht an und JDepend gibt den Wert 11 an, ohne weitere Details, deshalb gibt es keine Möglichkeit zu erfahren wie es auf diesen Wert kommt. Swat4j gibt den Durchschnittswert von 1,76 zu dem Projekt an, dies deckt sich mit dem Wert von Metrics. In Bezug auf die ausgewählte Methode getJButton3() (Anhang A.1) konnten nur RefactorIT und Metrics Auskunft geben. Während Metrics dieser Methode eine Komplexität von 2 ausgibt, errechnet RefactorIt einen Wert von 5. Bei näherer Betrachtung der Methode zeigt sich ein try/catch-Block, dem RefactorIt scheinbar eine höhere Wertigkeit zuweist, genauer sollte das in einem der anderen Projekte beobachtet werden, da die Werte in den anderen Methoden fast identisch sind und wenig Möglichkeiten bieten, die Ursache zu erkennen.

Abschließend sollen noch die *zusätzlichen Informationen* erwähnt werden. RefactorIT schlüsselt mit EXEC die Anzahl der Anweisungen auf. Dabei wird jedes Semikolon gezählt, jedoch werden verschachtelte Anweisungen, z.B. wie bei getJButton3() indem ein

komplettes Interface innerhalb einer Anweisung definiert wird, nicht beachtet. Solch eine Programmierweise verfälscht damit die Ergebnisse. Die 11 Datentypen stellen die globalen Variablen der Klasse GUI dar. Wichtig dabei ist, dass nicht die Variablen gezählt werden sondern die Typen. Über Swat4j erfährt man u.a. dass das Projekt 9 Interfaces implementiert und aus 37 Methoden besteht. Daten die eher für statistische Auswertungen dienen als für konkrete Probleme. Vergleicht man die Werte direkt mit dem Projekt, erkennt man, dass das Tool jede Methode zählt auch Interfaces die innerhalb einer Methode definiert wurden und auf verschachtelte Konstrukte zurück zuführen sind. Während Metrics hingegen nur die Methoden zählt die direkt unterhalb der Klasse definiert werden. Viel interessanter stellt sich der WMC Wert dar. Indem alle zyklomatischen Komplexitäten einer Klassen zusammen gezählt werden, zeigt sich eine Gewichtung über den Funktionsumfang der jeweiligen Klassen, wie z.B. das die Klasse SAX nur 22% der Funktionen beinhaltet gegenüber der GUI Klasse. Im Vergleich zu RefactorIT zählt Metrics alle globalen Variablen.

4.2. Lagerverwaltungsanwendung

Die nächste Anwendung die von den Analyse-Tools untersucht werden soll, ist eine Lagerverwaltungssoftware. Sie besitzt eine Grafische Oberfläche, Benutzerverwaltung sowie diverse Funktionen zur Lagerverwaltung. An dieser Software haben meherere Programmierer gearbeitet und unterschiedliche Klassen implementiert. Innerhalb Eclipse gliedert sich das Projekt in 5 große Pakete und zusätzlich 2 Ordner die das Handbuch und Bilder beinhalten:

- controller: 5 Java-Dateien,
- dateiarbeit: 3 Java-Dateien,
- daten: 9 Java-Dateien, 4 XML-Dateien, 1 User-Datei
- gui: 12 Java-Dateien,
- gui.jtablework: 3 Java-Dateien.
- *Ordner*: handbuch,
- *Ordner*: icons.

Zusätzlich bindet das Projekt eine externe Bibliothek jdom.jar ein. Das Projekt in Eclipse beinhaltet 206 Warnungen.

4.2.1. Planung

Wie es sich in dem vorangehenden Kapitel gezeigt hat, wird die Auswertung aller Tools über das komplette Projekt sehr aufwendig. Aus diesem Grund wird versucht auffallende Merkmale hervorzuheben, ohne dabei bereits in vorangegangenen untersuchten Aspekte erneut einzugehen.

Interressant wäre zu sehen, ob unterschiedliche Programmierstile innerhalb des Projektes zu erkennen sind. Da sich bereits gezeigt hat, dass es erhebliche Unterschiede bei dem Messen der efferenten und afferenten Kopplungen des jeweilen Tools gibt, sollte untersucht werden, inwieweit sich dieses Verhalten durch mehrere Pakete und Klassen ausprägt. Durch die erhöhte Anzahl an Warnungen lässt sich eine unsaubere Programmierung vermuten. Daher sollte geklärt werden, in wie weit sich dieses auf die Metriken auswirkt. Weiterhin könnten Komplikationen bei den Tools entstehen, falls diese eine geringe Fehlertoleranz besitzen.

4.2.2. Messergebnisse

CAP:	**Ca (Packages)**	**Ce (Packages)**	**I**	**A**	**D**
controller	2	9	82%	0	18%
dateiarbeit	1	8	89%	0	11%
daten	2	1	33%	11%	56%
gui	0	12	100%	0	0
gui.jtablework	1	4	80%	0	20%

RefactorIT:	**LoC**	**NCLoC**	**CLoC**	**DC**
controller	2915	2253	218	0,075
dateiarbeit	882	761	27	0,031
daten	1016	843	42	0,041
gui	6695	5587	508	0,076
gui.jtablework	271	207	31	0,114

Metrics:	**NBD**	**V(G)**	**PAR**	**WMC**
controller	8	35	6	470
dateiarbeit	2	14	2	131
daten	4	33	4	159
gui	7	140	11	676
gui.jtablework	4	11	6	29

Tabelle 4.2.: Auszug Messergebnisse

4.2.3. Auswertung und Interpretation.

Während der Untersuchung traten einige Fehler auf. Das Tool Swat4j konnte die Dateien DataManager.java, GUI.java und MACH.java nicht untersuchen, ebenso wurden keine weiteren Daten in der neuen Perspektive angezeigt, aus diesen Gründen wird das Tool innerhalb dieses Projekts nicht weiter betrachtet. Ähnliche Probleme traten auch in anderen Tools auf. Das Code Analysis Plugin untersuchte die BenutzerTest.java nicht und RefactorIT erzeugte einen Fehler während der Analyse. Daraus lässt sich erkennen, dass Warnungen, die duch Eclipse angezeigt werden, behoben werden sollten, um korrektes Arbeiten der Tools zu gewährleisten und fehlerhafte Messergebnisse zu vermeiden, wie sie sich bei RefactorIT zeigten.

Die Messergebnisse aus Tabelle 4.2.2 von CAP sind den Werten von JDepend annähernd identisch. Da jedes Tool die Importe unterschiedlich verarbeitet, entstehen abweichende Messwerte. Durch ungenutzte Importe und paketweises Importieren, welche später jedoch durch einzelne Klassen dieses Paketes erneut definiert werden, erhöhen sich die Abweichungen. Hinzu kommen ebenfalls die unterschiedlichen Definitionen innerhalb der Tools bezüglich dem Zählen externer Bibliotheken und Pakete, die in bereits importierten Paketen enthalten sind („java.util.*“/ „java.util.logging.*". Diese Schwankungen wirken sich auf die Ergebnisse der efferenten und afferenten Kopplungen aus und somit auf die Ergbnisse von Abstraktheit, Instabilität und Distanz. Als sehr genau erweist sich das Tool CAP gegenüber allen anderen. Durch ein IA-Diagramm lässt sich schnell erkennen, dass ein Grossteil der Pakete sich im konkreten und instabilen Bereich befindet, jedoch mit unterschiedlichem Abstand von der Hauptsequenz. Einzig das Paket „daten“ setzt sich von allen Paketen ab. Durch die Implementierung von Interfaces und der geringen Anzahl an efferenten Kopplungen entfernt sich dieses sehr stark von der Hauptsequenz und stellt dadurch ein stabiles, jedoch kaum abstraktes Paket dar. Was zur Folge eines starren und schwer veränderbaren Paketes hat. Dies sind Indikatoren, dass die Struktur dieses Projektes nicht optimal ist und verbessert werden könnte. Viel wichtiger ist, in denen sich alle Analyse-Tools einig sind, dass eine zyklische Abhängigkeit entdeckt wurde, die fatale Folgen haben kann, sollte etwas innerhalb dieser Klassen verändert werden. RefactorIT hat eindeutig Probleme die richtigen Kopplungen zu messen, sodass es in Bezug auf das Paket „gui“ 48 efferente Verbindung misst, ähnliche verfälschte Werte liefern die anderen Klassen. Als weiterhin korrekt kann das Tool Metrics betrachtet werden, da es Kopplungen nur innerhalb der eigenen definierten Pakete zählt und systemweite Pakete auslässt.

Ein weiterer Mangel lässt sich durch die Tools bzgl. des Aufbaus des Quellcodes erkennen. Da RefactorIT diese Werte innerhalb der ersten Spalte darstellt, sind diese sehr leicht ab zu lesen, Metrics bietet diese Informationen ebenfalls an, jedoch nicht als direkten Messwert. Bei einem Paket, das aus 761 Zeilen reinen geschriebenen Quellcode besteht, stellen lediglich 3% Kommentare dar. Von 761 Zeilen sind 27 Zeilen kommentiert. Durch diese Fakten aufmerksam geworden, lässt sich dieses Paket näher betrachten und es wird ersichtlich, dass in einer Klasse 4 einzelne Zeilen auskommentierten Quelltext darstellen und 12 Zeilen Javadoc Code darstellen, in dem der Name des Autors eingtragen und 11 Zeilen stellen eine Art Beschreibung und TODO Liste gleichzeit dar (Abb 4.2 stellt diesen Ausschnitt der Werte dar).

Target	V(G)	LOC	NCLOC	CLOC	DC
Metrics		11779	9651	826	0,07
controller		2915	2253	218	0,075
dateiarbeit		882	761	27	0,031
Cipher		721	637	0	0,0
DataManag		40	36	0	0,0
XMLDataMa		57	50	4	0,07
daten		1016	843	42	0,041
gui		6695	5587	508	0,076
gui.jtablework		271	207	31	0,114

Abbildung 4.2.: Mangelnde Kommentare durch RefactorIT dargestellt

Eine besonders nützliche Funktion wird durch Metrics gebracht, ähnlich wie RefactorIT weißt es durch farbliche Unterscheidung auf Überschreiten eines Grenzwertes hin. Da Metrics die Sortierung umgekehrt vornimmt, im Gegensatz zu den anderen Tools, wird sofort ersichtlich, welche Metriken nicht eingehalten wurden. Innerhalb dieser Metrik kann auf Projektebene nach den fehlerhaften Paketen gesucht werden. Die Tabelle 4.2.2 stellt 3 der Metriken dar, die durch die Lagerverwaltungssoftware überschritten wurden. Die Nested Block Depth (NBD) gibt die verschachtelte Tiefe innerhalb einer Methode an. Wie zu erkennen ist, sind 2 Pakete stark erhöht, *controller* besitzt mindestens 1 Methode die eine Verschachtelungstiefe von 8 besitzt, gleiches gilt für das Paket *gui* mit einer Tiefe von 7. Bei näherer Betrachtung des Paket *gui* zeigen sich 6 Methoden mit einer Tiefe von 6. Akkumulierend mit dem Wissen, dass das Paket die höchste Zahl an LoC besitzt und die höchste Gewichtung bei den Methoden erhalten hat, zeigt sich eine hohe Wichtigkeit dieses Paketes gegenüber dem restlichen Projekt. Durch den sehr hohen Wert von 140 bei

der zyklomatischen Zahl, lässt sich eine schlechte Strukturierung des Projektes erkennen und gibt dadurch Anlass über eine Restrukturierung des Quellcodes nach zu denken.

Bei dem Vergleich der Messwerte der Tools sind untereinander leichte Differenzen vorhanden. sodass leider nicht eindeutige Werte entstehen, jedoch werden stets die fehlerhaften und auffälligen Pakete indentifiziert und korrekt in den Kontext gebracht. Der Programmierstil ließ sich leider nicht erkennen, vermutlich auch, da die Aufgaben Funktionsorientiert verteilt wurden und nicht Paketweise, wonach die Analysetools leider stehts sortieren.

Durch die schlechten Importe und dem fehlerhaften Einlesen des Projekts in den einzelnen Tools, entstanden zum Teil große Differenzen bei den efferenten und afferenten Kopplungen, wie sie bei RefactorIT auftraten oder bei Swat4j das gar keine Analyse zu ließ. Die Vermutung die in der Planung statt fand, kann hiermit bestätigt werden, dass sich die Werte nur in dem Kontext des jeweiligen Tools auswerten lassen aber nicht mit denen der anderen Tools.

Was die Fehlertoleranz betrifft, scheinen die Tools die gleichen Probleme zu besitzen, aber unterschiedlich darauf zu reagieren. Swat4j gibt eine Liste nicht untersuchter Dateien an, CAP lässt nicht analysierbare Dateien einfach weg, ohne darauf hinzuweisen und RefactorIT berechnet falsche Werte bei fehlerhaften Dateien. Bei der Dauer der Analyse brauchte Swat4j am längsten gefolgt von RefactorIT, die den Quellcode sehr genau analysieren. Alle anderen Tool berechnen die Werte beinahe sofort.

4.3. Semesterplaner

Die letzte Anwendung ist der Semesterplaner. Diese Anwendung stellt einen Terminkalender für Studenten dar. Mit Hilfe dieser Anwendung kann man sich alle Daten über Vorlesungen, Übungen und Termine seines Semesters und Fachrichtung von der Hochschule herunterladen und in seine persönliche Planung aufnehmen. Ebenso ist es möglich, manuell Eintragungen zu erstellen oder zu editieren.

Während des Importierens des kompletten Projektes in Eclipse zeigt sich dessen Größe. Neben Quelltexten, nimmt Eclipse die Informationen zu Javadoc, einem html generierten Handbuch und vielen Grafiken auf. Eclipse weißt auf 2736 Warnungen hin, deren Großteil ungenutzte Variablen und Importe, innerhalb der ersten 100 Warnungen, darstellt. Bei einer ersten Einsichtnahme stellt Tabelle 4.3 eine Gliederung des Projekts dar.

System/Quelltexte:		**System/Quelltexte/transform:**
-application	-gui	-transform
-application.html.picneu	*-gui.pics*	
-application.html		
-edit	-io	
-helper	-test	
-jUnit		

Damit enthalten 8 Pakete Quelltexte, während 3 Pakete zusätzliche Dateien bereithalten.

4.3.1. Planung

Da dies ein großes Projekt darstellt, sollte zuerst herausgefunden werden, ob es möglich ist, die Verteilung der Dateien innerhalb der Pakete zu bestimmen und evtl. Pakete mit hoher Funktionalität zu lokalisieren. Obwohl in vorherigen Untersuchungen gezeigt wurde, dass die Bestimmung der Abstraktheit und Instabilität nie eindeutig verläuft, ist eine Tendenz stets heraus zulesen. Deshalb wäre ein Überlick der aktuelle Architektur dieses Projektes ineressant. Während das Projekt in Eclipse eingebunden wurde, zeigte sich während des Kompiliervorganges bereits ein hoher zeitlicher Aufwand von einigen Minuten. Wie verhalten sich die Tools auf diesen Aspekt. Als abschließende Untersuchung soll herausgefunden werden, wieviele Grenzwertüberschreitungen pro Tool gefunden werden und ob typische Programmierstilfehler entdeckt werden können, wie z.B. mangelnde Dokumentation des Quellcodes.

4.3.2. Messergebnisse

Ca/ Ce:	CAP	RefactorIT	Metrics	JDepend
application	2/ 14	208/ 19	212/ 12	3/ 15
edit	1/ 6	2/ 8	2/ 7	2/ 9
gui	2/ 12	14/ 25	13/ 9	2/ 12
helper	5/ 7	238/ 15	232/ 1	6/ 8
io	2/ 5	3/ 2	4/ 2	2/ 6
jUnit	0/ 0	0/ 203	0/ 202	0/ 7
test	0/ 10	0/ 9	0/ 9	0/ 11
transform	1/ 6	8/ 3	8/ 3	1/ 8

Tabelle 4.3.: Messergebnisse Ca/ Ce aller Tools

RefactorIT:	LoC	NCLoC	CLoC	DC
application	12656	9186	1930	**0,152**
edit	5800	3866	1518	0,262
gui	9392	6120	1960	0,209
helper	4416	2472	1422	0,322
io	578	292	206	0,356
jUnit	446490	298426	108252	0,242
test	1504	1154	106	**0,07**
transform	2256	1323	723	0,32

Tabelle 4.4.: Daten aus RefactorIT

Metrics:	NBD	V(G)	PAR	WMC
application	**9**	**116**	4	856
edit	**8**	**25**	**8**	350
gui	**6**	**14**	4	294
helper	4	**57**	5	113
io	3	5	1	22
jUnit	**9**	**140**	3	35389
test	**6**	**40**	1	56
transform	5	**11**	3	80

Tabelle 4.5.: ausgewählte Messergebnisse

4.3.3. Auswertung und Interpretation

Die Analyse konnte dieses Mal nicht flüssig durchgeführt werden. Das Tool RefactorIT brauchte für die 268 Klassen, 5 Minuten. Die Angabe von Swat4j für 84s ist ebenfalls nicht ganz korrekt, da Eclipse über 2 Minuten benötigte, bis die Übersicht präsentiert wurde. Während alle anderen Tools relativ zügig arbeiteten und eine durschschnittliche Wartezeit von 1 Minute zu messen war. Da das Betriebssystem auf einem PC mit 1,4 GHz und 620 MB Ram läuft, lassen sich die Analysezeiten durch aktuellere PC Systeme sicherlich verkürzen. Weitere Probleme traten nur bei RefactorIT auf, welches doppelte Java-Dateien entdeckte, was aber auf einen internen Fehler zurückzuführen sein muss, da keine Fehler durch die Projektverwaltung von Eclipse ausgegeben wurden.

Betrachtet man die Messwerte aus Tabelle 4.3.2, können zwei unterschiedliche Messgruppen festgestellt werden, da in vorangehenden Untersuchungen die differentielle Erhebung der Messwerte festgestellt wurde, treten diesmal einige Besonderheiten zu Tage. Zum einen hat CAP nur 1 Datei aus dem Paket gelesen und alle anderen 202 ausgelassen, damit lassen sich die abweichenden Messwerte erklären und zum anderen beziehen sich viele Kopplungen auf interne Pakete, gerade die durch jUnit erzeugten Verbindungen. Dadurch

kommt es zu ähnlichen Ergebnissen zwischen RefactorIT und Metrics. Besonders gut lässt sich diesmal die Korrelation zwischen den efferenten Kopplungen des „jUnit“ Paketes und den afferenten Kopplungen bei den Paketen „application“ und „helper“ erkennen, auf welche sich diese beziehen. Während die anderen Werte als normaler Durchschnitt betrachtet werden können, sind die über 200 Verbindungen sehr auffällig. Mit Hilfe der Werte aus Tabelle 4.3.2 wird sehr schnell eine Gewichtung ersichtlich. Mit 446490 Zeilen stellt das Paket 92% des gesamten Quelltextes dieses Projektes dar.

Durchschnittlich besitzt jedes Paket mehr efferente, als afferente Verbindungen, die Pakete sind mehr abhängig von anderen Paketen, als sie selbst von anderen. Dadurch neigen die Klassen zu einer hohen Instabilität. Ergänzend durch die mangelnde Implementierung von Interfaces und abstrakten Klassen, erfüllen beinahe alle Klassen relativ gut die Metriken von Robert Martin. Einzig das Paket „helper“ enthält Interfaces und entfernt sich deshalb etwas von der Hauptsequenz als andere Pakete. Damit bietet der Aufbau dieses Projektes sehr wenig Ansatzmöglichkeiten um überhaupt eine Restrukturierung in Betracht ziehen zu wollen.

Etwas auffälliger zeigt sich eher die Verteilung der Funktionalitäten. Durch das Tool Metrics werden 3 Metriken angezeigt die überschritten wurden. Durch eine zu hohe Verschachtelungstiefe und einer viel zu hohen zyklomatischen Komplexität innerhalb der Methoden zeichnet sich beinahe jedes Paket aus. Damit sollten viele Funktionen in eigene Methoden ausgelagert werden um die Möglichkeit potenzielle Fehler zu verringern und die Verständlichkeit des Quelltextes zu verbessern. Obwohl man, innerhalb dieses Projektes, auf eine gute durchschnittliche Kommentierung zurückgreifen kann.

Jedoch verweist RefactorIT auf 2 Pakete, die eine sehr schlechte Kommentardichte besitzen (siehe Tabelle 4.3.2).

Wird das Paket „test“ näher betrachtet, sollte eine Umstrukturierung in Betracht gezogen werden. Nach den statistischen Werten der Metriken, sind 106 von 1154 Zeilen auf Kommentare ausgerichtet. aber eine zyklomatische Zahl von 40 mit wenigstens einer Methode die eine Verschachtelungstiefe von 6 besitzt, scheinen sehr komplexe Aufgaben umgesetzt zu werden. Eine sehr schlechte Kombination.

Da Swat4j diesmal volle Unterstützung gibt, kann dieses ebenfalls herangezogen werden. Obwohl es keine relevanten Fakten liefern kann, die nicht durch andere Metriken ebenfalls erhoben wurden, kann es unterstützend verwendet werden. Die Abbildung 4.3 zeigt den geringen Grad an Dokumentation. Obwohl das Paket ungenutze Variablen und Methoden besitzt (toter Code) und nicht jede Variable als „private“ deklariert wurde (Sicherheit),

besitzt dieses Paket eine sehr gute Abdeckung möglicher Fehler, Leistungsfähigkeit, Einhaltung der Namenskonventionen und der Architektur.

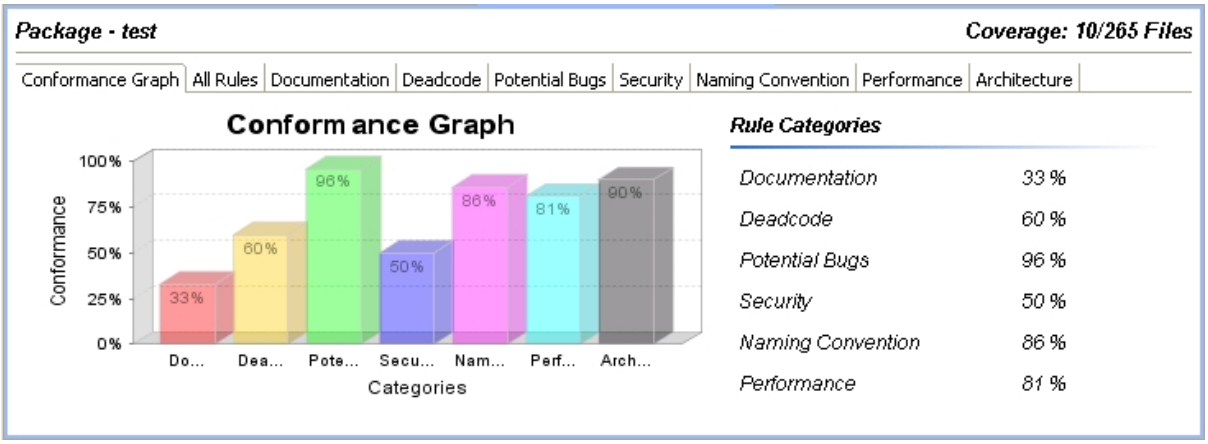


Abbildung 4.3.: Swat4j - Paket test

Ergänzend sollte erwähnt werden, die Beschränkung auf 50 Dateien pro Analyse bei der Testversion von Swat4j ist sehr hinderlich, da es dadurch nicht möglich ist, einen kompletten Eindruck über das Projekt zu erhalten.

Auf der Suche nach charakteristischen Merkmalen konnte wiederrum nichts konkretes herausgefunden werden, zwar sind innerhalb eines Paketes Dateien zu finden, die besonders stark kommentiert wurden und andere bei denen es komplett fehlt. Ebenso verhält es sich bei der Komplexität einzelner Methoden, jedoch korrellieren diese Werte mit der Wichtigkeit und dem Aufgabenbereich der einzelnen Klassen. In der paketübergreifenden Ansicht ließ sich jedoch das hohe Maß an Dokumentation finden, sowie eine hohe Komplexität der einzelnen Methoden, welches diesem Projekt einem besser Grad der Programmierung ausweist, als der Anwendung in Kapitel 4.2

5. Auswertung der Ergebnissse

Die aus Kapitel 3 und 4 durchgeführten Untersuchungen, fließen hier zusammen und sollen eine abschließende Bewertung der Tools geben. Dabei werden in den folgenden Abschnitten bei jedem der Tools die Stärken und Schwächen, dich sich während der Untersuchung gezeigt, haben mit den ersten Bewertungen zusammengeführt und bilden somit ein abschließendes Resümee. Dabei soll die Tabelle 5 zu Beginn als eine Schnellübersicht dienen, in der die wichtigsten Eigenschaften zusammengefasst dargestellt und bewertet werden.

Übersicht	CAP	RefactorIT	Swat4j	Metrics	EclipseMetrics	JDepend
Darstellung	++	-	++	+	–	–
Bedienbarkeit	++	-	+	+	–	+
Funktionsumfang	+	+	++	+	–	-
Anzahl Metriken	+	++	(++)	++	–	+
Zuverlässigkeit:						
der Ergebnisse	++	+	++	++	+	++
der Anwendung	++	+	–	++	+	++
Schnelligkeit	++	(+)	-	++	++	+
Dokumentation	+	++	++	-	+	+
Empfehlung	(++)	++	+	+++	-	–

Tabelle 5.1.: Zusammenfassung

Legende:

+++ : sehr zu empfehlen

++ : besonders gut erfüllt

\+ : gut erfüllt

\- : tendenziell eher schlecht

– : völlig unzureichend

() : unter Vorbehalt, nur bedingt zu bewerten

5.1. Code Analysis Plugin - CAP

5.1.1. Positives

Das Plugin arbeitet zuverlässig. Die Werte sind alle nachvollziehbar, vor allem durch die grafische Darstellung der Verbindungen. Zyklische Abhängigkeiten werden schnell gefunden. Die Oberfläche ist aufgeräumt und intuitiv bedienbar. Es erleichtert das Verständnis der kompletten Architektur eines Projekts und es enhält keine Abkürzung der verwendeten Metriken.

5.1.2. Negatives

Leider ist es nur für sehr wenige Metriken ausgelegt und hilft nicht für das tiefere Verständnis. Wenn etwas nicht eingelesen werden kann, wird nicht darauf hingewiesen, sodass man stets darauf achten sollte, ob alle Klassen aufgelistet werden. Was teilweise sehr fatal sein kann und Teile eines Paketes vollkommen verfälschen kann. Damit hilft es nicht den Quellcode zu verbessern, es zeigt einfach nur den jeweiligen Zustand an.

5.2. RefactorIT

5.2.1. Positives

Da der Schwerpunkt dieses Tools vor allem auf genauer Analyse des Quellcodes liegt, um ein automatisiertes Restrukturieren zu ermöglichen, ermittelt es sehr zuverlässig alle Daten. Dazu muß der Quellcode fehlerlos vorliegen. Unter dieser Vorraussetzung liefert es sehr viele Informationen. Das man selber die Metriken bestimmen und die Grenzwerte definieren kann, nach denen man suchen will macht es sehr benutzerfreundlich. Es enthält alle wichtigen Metriken die man benötigt und es arbeitet zuverlässig. Ein Hilfe-Funktion wird in Eclipse integriert die das einarbeiten und verstehen hilft. Über CSV oder XML lassen sich die Informationen weiterverarbeiten und evtl. besser darstellen.

5.2.2. Negatives

Leider ist die Darstellung sehr unvorteilhaft. Da es Metriken für Pakete, Klassen und Methoden gibt, entstehen zu viele Spalten. Die Daten die berechnet werden sind nicht immer

nachvollziehbar, gerade bei den Metriken von Robert Martin zeigte es öfters Schwächen, daher sollte man diese Metriken mit diesem Tool nicht verwenden. Außerdem verlangsamen diese Metriken die Analyse. Die Exportfunktion ist auch zwiespältig. Zwar lassen sich Informationen übersichtlicher darstellen und HTML ist ein Format das man beinahe überall öffnen kann, doch bei zu großen Projekten muss man enorm scrollen und es fehlen einem dann die Spaltenüberschriften. Die Funktion, Abhängigkeiten zu zeichnen wirkt unausgereift und dadurch wird es überflüssig. Darüber hinaus verwendet es sehr viele Abkürzungen, damit stellt dieses Tool etwas für erfahrene Entwickler dar.

5.3. Swat4j

5.3.1. Positives

Ähnliches wie CAP benutzt es eine eigene Perspektive, dadurch wirkt alles sehr aufgeräumt und erhöht die Benutzerfreundlichkeit. Der Schwerpunkt dieses Tools liegt mehr auf der Präsentation bzw. Auswertung der Daten. Es berechnet die Werte und bewertet diese bereits. Für jede Metrik gibt es eine Darstellung. Für Anfänger ist diese eigene Bewertung sehr hilfreich, da es einem sagt, was zu verbessern ist. Durch Trennung in verschiedene Bereiche wie Wartbarkeit oder Leistung bekommt man ein Feedback darüber was, wodurch verbessert werden kann.

5.3.2. Negatives

In Bezug auf die 3 Projekte arbeitete es sehr unzuverlässig. Es konnte nur den Semesterplaner vollständig untersuchen und bewerten. Gerade bei dem Projekt, welches es untersuchen konnte, kam die Begrenzung auf 50 Dateien pro Analyse zum Vorschein. Sehr störend und nicht produktiv. Die Grafiken helfen um ein Projekt vorzustellen bzw. zu präsentieren, möchte man jedoch konkrete Metriken mit den dazugehörigen Ergebnissen, wird man diese nur vereinzelt finden und nur pro Paket. Eine praktische Verwendung findet nur die zusätzliche Funktion „Best Practices“ welches auf Verbesserung der Codequalität zielt, ähnlich wie PMD. Dem ersten Eindruck wurde es keinesweg gerecht, vor allem im Vergleich der Leistungsfähigkeit anderer Tools. Daher ist es nur für Anfänger oder zu Präsentationzwecken zu empfehlen.

5.4. Metrics

5.4.1. Positives

Es ist eines der zuverlässigsten und produktivsten Metriken. Die Darstellung der Ergebnisse ist durchdacht, da es nach Metriken sortiert und die Ansicht auf jeder Ebene den nötigen Daten anpasst. Dadurch wirkt es sehr aufgeräumt und übersichtlich. Werden Grenzwerte überschritten färbt es diese rot ein. Die wichtigsten Daten sind stets vorhanden. Es erzeugte nie Fehler und hatte nie Probleme die korrekten Werte zu ermitteln. Die zusätzliche Funktion des Abhänigkeitsgraphen überzeugt durch Darstellung und den nützlichen Funktionen. Die Metriken werden ausgeschrieben und erleichtern das Verständnis.

5.4.2. Negatives

Das Einrichten war anfangs unverständlich und problematisch. Die Dokumentation ist grundlegend aber nicht ausreichend. Die Analyse der efferenten und afferenten Kopplungen ist etwas eigenwillig, aber letztendlich verständlich. Die einzige Exportmöglichkeit ist jene nach XML, das ein direktes Verwenden der Daten nicht möglich macht. Das Aktivieren des Tools und das Einstellen erfolgt über die Projekteigenschaften da es keine eigene Menüstruktur besitzt. Die Berechnung wird durch den Kompiliervorgang von Eclipse gestartet und läuft dann jedes Mal mit.

5.5. Eclipse Metrics

5.5.1. Positives

Die Umsetzung lehnt an dem Tool von Metrics an. Es wurde versucht, die Daten innerhalb der Quellcode übersicht unterzubringen. Man sieht Veränderungen dadurch sofort und hilft direkt während man den Quellcode schreibt.

5.5.2. Negatives

Jedoch überwiegen die negativen Aspekte denen der Vorteile, die es mit sich bringt. Die umgesetzten Metriken besitzen in der Literatur große Zweifel. Man muß jedes Mal mit

dem Mauszeiger über das kleine Icon fahren, um die Informationen zu erhalten. Dieses Icon ist nicht an jeder Methode enthalten. Teilweise sind nur Informationen über Anzahl der Zeilen enthalten was den Sinn dieses Tools fragwürdig erscheinen lässt. Einzig die zyklomatische Zahl, die teilweise berechnet wird, ist wichtig. Alle gesammelten Berechnungen werden in der Fehlerübersicht bei Eclipse als Warnung ergänzend hinzugefügt, sodass aus 2300 Warnungen, wie es bei dem Semesterplaner war, über 6000 Warnungen entstehen und so wichtige Informationen unnötig verdeckt. Die Ansätze sind gut, leider ist die Umsetzung absolut unzureichend.

5.6. JDepend/ JDepend4Eclipse

5.6.1. Positives

Für JDepend gibt es verschiedene Ausführungen. Importiert man dieses Projekt in Eclipse, kann man JDepend4Eclipse als Ergänzung installieren, welches die Ausgabe der Daten enorm aufwertet und dadurch leserlich gestaltet.

5.6.2. Negatives

Sollte man das Code Analysis Plugin nicht installieren können, ist JDepend eine Alternative da es die gleichen Werte berechnet. Die berechneten Daten sind einzig auf die Metriken von Robert Martin begrenzt. Es kann nur Pakete analysieren, sollte ein standard-Paket von Eclipse erzeugt werden, wird dieses nicht untersucht. Damit man JDepend4Eclipse verwenden kann muß die eigentliche Anwendung JDepend als Projekt importiert und geöffnet vorliegen. Die Darstellung von JDepend ist sehr rudimentär. Die Verwendung dieses Tools bietet sich nur an, wenn man keines der oben genannten Tools verwenden kann.

5.7. Ergänzung

Der Vollständigkeit halber, werden hier weitere Tools namentlich erwähnt, die erhältlich sind und als metrische Analyseprogramme beworben werden. Da sie entweder nicht funktionsfähig waren, ungenügend Metriken anboten oder veraltet waren, wurden sie nicht vorgestellt.

- Lachesis Analysis: http://lachesis.sourceforge.net/
- Enerjy: http://enerjy.com/download.html
- X-Ray: http://atelier.inf.unisi.ch/ malnatij/xray.php
- JavaNCSS: http://www.kclee.de/clemens/java/javancss/

6. Zusammenfassung

Die Aufgabe war es, zu untersuchen, welche Metriken aktuell Güligkeit besitzen und welche Tools man dafür einsetzen kann. Dabei wurden die Metriken LoC, von Halstead und von McCabe vorgestellt. Bei den objektorientierten Metriken die Struktur- und Komponentenmetriken sowie die sehr wichtigen Metriken von Robert Martin.

Während der Suche nach Tools zeigten sich sehr viele mögliche Probanten, doch bei ersten Untersuchungen fielen große Mängel auf, sodass viele Tools unbrauchbar waren. Jedoch wurden während dieser Evaluation einige Tools, wie RefactorIT, Swat4j oder CAP gefunden und vorgestellt. Dabei wurde im ersten Schritt die einzelnen Fakten der Tools dargestellt und nach einer Installation und erstem Eindruck bewertet. Dabei zeigte sich bereits die Spezialisierung einzelner Tools, die ihr Einsatzgebiet als metrische Analyseprogramme teilweise stark einschränken.

In dem darauf folgenden Kapitel folgten Untersuchungen auf ihre tatsächliche Einsatzfähigkeit und Leistungsfähigkeit. Dabei zeigte sich die tatsächliche Funktionstüchtigkeit. Während auf den Webseiten der Anbieter, die vielfältigen Darstellungs- und Auswertungsmöglichkeiten beworben werden, ist deren praktischer Nutzen als Entwickler sehr unterschiedlich. Verwendet man die Tools das erste mal, wirken Balken-, Streu- und Kuchendiagramme sehr ansprechend. Beginnt man jedoch damit zu arbeiten und möchte konkret Daten erfahren, sind diese nicht immer existent und so verändert sich der Eindruck. Gerade bei dem Tool Swat4j war es sehr bedauerlich, dass nur 1 von 3 Projekten untersucht werden konnte. Selbst als es weitere Informationen präsentierte, waren Daten wie „Anzahl an Dateien des Paketes“, bei denen nur Minimum und Maximum als Kriterium vorhanden waren, extrem verwirrend. Die ausgewiesenen Durchschnittswerte über die Pakete sind einzig für statistische Angaben nützlich. Folglich traten andere Tools in den Vordergrund, welche anfangs schlechter bewertet wurden. Gerade wenn Fakten nötig sind, tritt grafische Darstellung verstärkt in den Hintergrund.

Ebenfalls auffällig war der Unterschied zwischen Theorie und Praxis, während in Kapitel 2 anerkannte und bewährte Metriken vorgestellt und ebenso auf zweifelhafte Metriken hingewiesen wurden. Fanden sich in der Praxis ganz andere Metriken wieder, sodass z.B. in vielen Tools „Lack of Cohesion of Methods“ und verwandte Metriken untersucht

werden, deren Messergebnisse jedoch keine Korrelation mit den untersuchten Objekten zugeschrieben werden konnte. Anderseits wurde die vorgestellte Halstead Metrik als einzige nur rudimentär in Swat4j aufgenommen. Obwohl ihr in der Literatur ein hoher Stellenwert zugeschrieben wird, scheint sie in der Praxis nur wenig Verwendung finden.

Darüber hinaus zeigten sich die Schwierigkeiten der eindeutigen Umsetzung in der Praxis. Obwohl die Metriken durch eindeutige Formeln und Berechnungsvorschriften definiert sind, lassen sich die Operanden nie eindeutig bestimmen, wodurch unterschiedliche Ergebnisse entstehen. Aus diesem Grund erschwerte es die Messergebnisse, der afferenten und efferenten Kopplungen, untereinander zu vergleichen.

Wenn man mit verschiedenen Analysetools arbeitet, sollte man stets beachten, welche der gesammelten Ergebnisse mit anderen Projekten verglichen werden kann und wie.

Je mehr man sich mit einem Tool und einem Projekt auseinander setzt, desto mehr Informationen lassen sich gewinnen. Was an sich kein Problem darstellt, sondern sehr dienlich ist. Die Darstellung wird jedoch immer komplizierter. Der starke Zuwachs an Informationen, den man auf jeder Ebene erhält (Paket, Klassen und Methodenebene) ist enorm. Wollte man über alle Metriken sprechen, die man mit den Tools verwenden kann, würde dies den Rahmen sprengen. Deshalb wurden einzelne wichtige Aspekte herausgesucht und vorgestellt. Dadurch zeigten sich innerhalb jedes Projektes einige wichtige Merkmale. In der Filmdatenbank konnte man anhand der statistischen Daten erkennen, wieviel leere Zeilen für einen übersichtlichen Code ausmachen. Während bei der Lagerverwaltung durch Weighted Methods per Class die Wichtigkeit einzelner Pakete in ihrer Funktion erkennen kann. Oder wie gut eine Dokumenation sein kann. Sicherlich lassen die Metriken eine wesentlich tiefere Analyse der Projekte zu, dies wäre jedoch eine Aufgabe für eine weitere Untersuchung. Abschließend wurden die gesammelten Daten aus der anfänglichen Bewertung, sowie den Informationen aus den analysierten Projekten verbunden. Eine erste Übersichtstabelle soll dabei helfen, die Ergebnisse grafisch darzustellen. Die einzelnen Vorteile und Nachteile wurden daraufhin dargelegt.

Als möglichen Ausblick würde es sich anbieten, eines der vorgestellten Projekte mit einem der vorgestellten Tools zu verwenden und mit dessen Hilfe den Quellcode zu überarbeiten, um ein konformes Projekt zu entwickeln. Dieses könnte man mit dem alten Projekt gegenüberstellen und sehen ob sich tatsächlich eine Verbesserung eingestellt hat.

Da einige der verwendeten Metriken sich in der Fachliteratur noch nicht durchgesetzt haben, würde sich ein Untersuchung und anbieten, die korrekte Funktionsweise dieser Metriken darzulegen.

Ebenso ist eine eigene Entwicklung eines solchen Metrik-Tools möglich. Da keines der untersuchten Tools vollständig überzeugen konnte, würde sich ein solches Projekt anbieten. Mit den in dieser Untersuchung gesammelten Daten, lassen sich Kriterien festlegen, um wichtige Metriken zu integrieren und unnötige zu entfernen. Da gerade die Darstellung der Messwerte große Diskrepanzen aufwies, ist eine eigene Entwicklung zu empfehlen. Viele der vorgestellten Tools wurde seit Jahren nicht mehr aktualisiert.

Abschließend muss noch die Empfehlung für das Tool Metrics ausgesprochen werden, die durchgängige Zuverlässigkeit und sehr intelligente Umsetzung lies es in vielen Kategorien in den Vordergrund rücken. Ergänzend kann RefactorIT herangezogen werden, vor allem wenn es um statistische Werte wie LoC geht, lassen sich die Informationen sehr schnell ablesen. Wie bereits erwähnt, vollkommen konnte kein Tool überzeugen, da jedes Tool seine Stärken und Schwächen hat.

A. Anhang

A.1. Auflistung aller untersuchten Metriken von Swat4j

Methodenebene
Cyclomatic complexity of the method.
Number of lines of code of the method.
Number of commented lines of the method.
Number of anonymous inner classes of the method.
Number of conditional statements of the method.
Number of variables of the method.
Number of unused variables of the method.
Number of parameters of the method.
Number of unused parameters of the method.
Number of statements of the method.
Percentage of comments of the method.

Klassenebene
Weighted methods of the class.
Weighted methods complexity of the class.
Number of lines of code of the class.
Response for a class.
Lack of cohesion of methods of the class.
Coupling between objects of the class.
Depth of Inheritance tree of the class.
Number of children of the class.
Percentage of non private fields of the class.
Percentage of non private methods of the class.
Number of inner classes of the class.
Number of fields of the class.
Number of parents of the class.

Paketebene
Number of lines of code of the package.
Number of functions of the package.
Number of classes of the package.
Number of anonymous inner classes of the package.
Number of interfaces of the package.
Number of enumerations of the package.
Number of files of the package.

Projektebene
Number of lines of code of the project.
Number of functions of the project.
Number of classes of the project.
Number of anonymous classes of the project.
Number of interfaces of the project.
Number of enumerations of the project.
Number of files of the project.
Number of packages of the project.

weitere Metriken
Number of lines of code of the file.
Halstead effort of the file.
Maintainability index of the file.
Halstead volume of the file.

Tabelle A.1.: Auflistung der Metriken von Swat4j

A.2. Die Methode getJButton3() aus der Filmdatenbank

```
1  /**
2   * This method initializes jButton3
3   *
4   * @return javax.swing.JButton
5   */
6  private JButton getJButton3() {
7    if (jButton3 == null) {
8      jButton3 = new JButton();
9      jButton3.setBounds(new Rectangle(500, 83, 212, 39));
10     jButton3.setText("HTML-Datei öffnen");
```

```
11        jButton3.setVisible( false );
12        jButton3.addActionListener(new java.awt.event.ActionListener() {
13          public void actionPerformed(java.awt.event.ActionEvent e) {
14                    int option= JOptionPane.showConfirmDialog(null, "Internet
                          Browser oeffnen?",null, JOptionPane.YES_NO_OPTION);
15                  Desktop d = Desktop.getDesktop();
16                  switch (option){
17                  case JOptionPane.YES_OPTION:
18                    try {
19                    d.browse(new URI("FilmDB.html"));
20                  } catch (IOException e1) {
21                    JOptionPane.showMessageDialog(null, "Fehler!\n\nProgramm
                          wird beendet...",null, JOptionPane.ERROR_MESSAGE);
22                    System.exit(0);
23                  } catch (URISyntaxException e1) {
24                    JOptionPane.showMessageDialog(null, "Fehler!\n\nProgramm
                          wird beendet... ",null, JOptionPane.ERROR_MESSAGE);
25                    System.exit(0);
26                  }
27                  }
28                }
29              });
30
31            }
32            return jButton3;
33          }
```

Listing A.1: Auszug der Methode getJButton3()

Literaturverzeichnis

[Appa] Quellcode zum Beleg Datenrepräsentation "Filmdatenbank", Interne Projektunterlagen, Hochschule Mittweida (FH), Fakultät MPI, 2008.

[Appb] Quellcode zum Softwaretechnik-Projekt „Lagerverwaltung“, Interne Projektunterlagen, Hochschule Mittweida (FH), Fakultät MPI, 2008.

[Appc] Quellcode zum Softwaretechnik-Projekt „Semesterplaner“, Interne Projektunterlagen, Hochschule Mittweida (FH), Fakultät MPI, 2008.

[Boi] Boissier, Guillaume: *Metrics 1.3.6*. http://sourceforge.net/projects/metrics/, http://metrics.sourceforge.net/ Stand: 10.08.2009.

[Cla] Clarkware: *JDepend*. http://www.clarkware.com/software/JDepend.html, Stand:10.08.2009.

[Cod] Codeswat: *Swat4j*. http://www.codeswat.com/cswat/index.php?option=com_content&task=view&id=20&Itemid=38, Stand: 10.08.2009.

[Dum92] Dumke, Reiner: *Softwareentwicklung nach Maß*. Vieweg, 1992.

[Dum93] Dumke, Reiner: *Modernes Software Engineering*. Vieweg, 1993.

[Hal77] Halstead, Maurice H.: *Elements of Software Science*. Elsevier, 1977.

[Hof08] Hoffmann, Dirk W.: *Software-Qualität*. Springer-Verlag Berlin Heidelberg, 2008.

[KL07] Klaus Lambertz, Xavier-Noel Cullmann: *Komplexität und Qualität von Software*. MSCoder, 1/2007:7, 2007. http://www.verifysoft.com/de_cmtpp_mscoder.pdf, Stand: 10.08.2009.

[Los09] Loskutov, Andrei: *JDepend plugin for Eclipse: JDepend4Eclipse*. http://andrei.gmxhome.de/jdepend4eclipse/, Stand: 10.08.2009, 10 2009.

[Mar94] Martin, Robert: *An Analysis of Dependencies*. OO Design Quality Metrics:8, 1994. http://www.objectmentor.com/resources/articles/oodmetrc.pdf, Stand: 10.08.2009.

[McC76] McCabe, Thomas J.: *A Complexity Measure*. IEEE Transactions on Software Engineering, SE-2, No. 4:12, December 1976.

[RD96] Reiner Dumke, Erik Foltin, Reinhard Koeppe: *Softwarequalität durch Meßtools*. Vieweg, 1996.

[Sch] Schneider, Johannes: *CAP - Code Analysis Plugin*. http://sourceforge.net/projects/cap4e/, http://cap.xore.de/, Stand: 10.08.2009.

[Tha00] Thaller, Georg Erwin: *Software-Metriken (einsetzen - bewerten - messe)*. Verlag Technik Berlin, 2000.

[Tho04] Thomas, Michael: *Software-Metriken*. Software Engineering, 2004. http://www.se.uni-hannover.de/documents/kurz-und-gut/ws2004-seminar-entwurf/software-metriken_mthomas.pdf, Stand: 10.08.2009.

[Wal] Walton, Lance: *Eclipse Metrics Plugin*. http://sourceforge.net/projects/eclipse-metrics/, http://www.stateofflow.com/projects/16/eclipsemetrics,http://eclipse-metrics.sourceforge.net/ , Stand: 10.08.2009.

[Wih] Wihler, Oliver: *RefactorIT*. http://sourceforge.net/projects/refactorit/, http://www.aqris.com/display/A/Refactorit?id=1349,http://www.peakengineering.de/RefactorIT/refactorit.html , Stand: 10.08.2009.

[Wul02] Wulff, Nikolaus: *Metriken zur Steigerung der Softwarequalität*. Javamagazin, 4.2002:7, 2002.

Selbständigkeitserklärung

Ich erkläre, dass ich die vorliegende Arbeit selbstständig und nur unter Verwendung der angegebenen Literatur und Hilfsmittel angefertigt habe.

Daniel Ramm Mittweida, den 02.11.2009