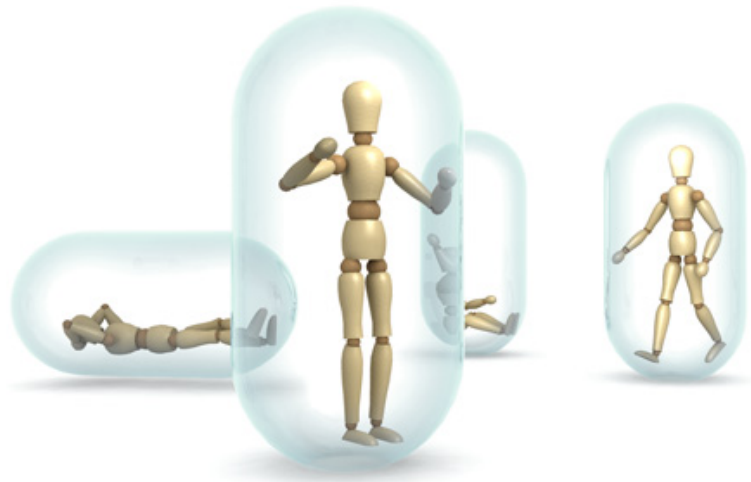


### Hinweis:

Diese Druckversion der Lerneinheit stellt aufgrund der Beschaffenheit des Mediums eine im Funktionsumfang stark eingeschränkte Variante des Lernmaterials dar. Um alle Funktionen, insbesondere Animationen und Interaktionen, nutzen zu können, benötigen Sie die On- oder Offlineversion. Die Inhalte sind urheberrechtlich geschützt.  
©2018 Beuth Hochschule für Technik Berlin

## MET - Software- und Architekturmetriken



## Lernziele und Überblick

Die Lerneinheit „Softwaremetriken“ ist eine Ergänzung zum Thema Softwarequalität, zu der auch die Lerneinheit „Testen“ gehört. Es geht darum, fehlende Softwarequalität oder Softwarekomplexität sichtbar zu machen. Verletzungen von Regeln oder Probleme im Code werden gelistet und optimal visualisiert. Mit dieser Hilfe können dann Entwickler, Architekten oder Projektleiter die Weichen für neue Qualität besser stellen.



### Lernziele

- Sie lernen die Bedeutung der Metriken als Qualitätsmaß kennen sowie diese praktisch zu beurteilen.
- Sie lernen Basismetriken kennen und diese zu berechnen.
- Sie lernen Codemetriken kennen und die Werkzeuge zu gebrauchen.
- Sie lernen Architekturmetriken kennen und wenden Werkzeuge an, um diese zu visualisieren.



### Gliederung der Lerneinheit

Nach einer kurzen Einführung in das Thema Metriken werden zunächst einfache Metriken wie die McCabe-Metrik behandelt. Nach der Halstead-Metrik und dem Code Coverage wird auf die unterschiedlichen Arten von Regelverletzungen eingegangen, die von vielen Werkzeugen als Metriken aggregiert angezeigt werden.

Im zweiten Teil folgen JDepend, Kopplungsmetriken und die Schuldfrage - der Technical Debt.

Abschließend werden aktuelle verfügbare Werkzeuge für Java aufgeführt und verglichen.



### Zeitbedarf und Umfang

Zum Durcharbeiten dieser Lerneinheit benötigen Sie ca. 120 Minuten.

Die Anwendung eines Werkzeuges wie XRadar, Sonar oder SonarJ auf ein bestehendes Projekt dauert je drei Stunden.



Film



© Beuth Hochschule Berlin - Dauer: 13:18 Min. - Streaming Media 25.3 MB

Die Hinweise auf klausurrelevante Teile beziehen sich möglicherweise nicht auf Ihren Kurs.

Stimmen Sie sich darüber bitte mit ihrer Kursbetreuung ab.

## 1 Motivation

Als Metrik bezeichnet man alles Messbare. Gemeint sind **Maß oder Kennzahlen**, die in allen Gebieten wie Musik, Physik, Mathematik, u.a. vorkommen.



### Definition

#### Softwaremetrik

Eine Softwaremetrik ist eine Maßzahl, die etwas über die Software aussagt.

*„Eine Softwaremetrik ist eine Funktion, die eine Software-Einheit in einen Zahlenwert abbildet. Dieser berechnete Wert ist interpretierbar als der Erfüllungsgrad einer Qualitätseigenschaft der Software-Einheit.“* (IEEE Standard 1061, 1992)



### Beispiel

Wartbarkeit,  
Veränderbarkeit,  
Testbarkeit

Eine einfache Metrik wäre damit beispielsweise die Anzahl der Codezeilen oder die Anzahl der Klassen in einem OO-Projekt. Aber natürlich geht die Thematik Metriken viel weiter - bis hin zu Architekturmetriken, die etwas über den Architekturzustand des Softwaresystems mitteilen.

Da eine gute Softwarearchitektur auch wieder Wartbarkeit, Veränderbarkeit und Testbarkeit bedeutet, wird diese Thematik auch unter dem Begriff **Software-Stability** geführt. Es beschreibt, wie stabil das Softwaresystem in Bezug auf Veränderungen, also auf seine Weiterentwicklung reagiert.

Grundsätzlich gibt es in Bezug auf Softwareentwicklung viele Arten von Metriken. Ein Softwareprojekt besteht ja durchaus aus vielen Elementen, beispielsweise:

- Sourcecode
- Binärcode
- externen Bibliotheken
- Buildinformation
- Versionskontrollinformation
- Formen von Meta- und Zusatzinformationen (z. B. Annotations, JavaDoc, etc.)

Dies bedeutet auch, dass sich viele Metriken messen lassen. In dieser Lerneinheit werden wir einige davon betrachten, wie z. B.:

- Komplexitätsmetriken
- Testmetriken
- Architekturmetriken
- Kopplungsmetriken
- Stylemetriken

In der Informatik gibt es mehr Arten von Metriken als nur Software- bzw. Codemetriken. Besonders aus dem Projektmanagement sind viele Metriken bekannt, wie z. B. die Function-Point Metrik oder COCOMO zur Aufwandsabschätzung.

Wir werden uns zu Beginn auf die einfachen Softwaremetriken konzentrieren und dann die Architekturmetriken kennenlernen.



### Hinweis

Gute Metriken sind nicht gleichbedeutend mit guter Qualität.  
Gute Qualität bedeutet nicht unbedingt gute Metriken!

Die Erfahrung aus vielen Projekten zeigt aber auch, dass die Metrik-Größe und die Qualitätseigenschaft durchaus häufig miteinander korrelieren!

Da heutzutage immer größere Projekte entstehen und die Qualität geprüft werden muss, kommt auch der Visualisierung von Qualitätsmetriken eine immer größere Bedeutung zu. Damit werden wir uns gegen Ende des Moduls beschäftigen.

Weitere  
Informationen

Interview mit Jonathan Aldrich zum Thema Static Analysis (45 Minuten):

 <http://www.se-radio.net/>

## 2 Einfache Metriken

Es ist oftmals hilfreich einen Überblick über ganz einfache Codemetriken zu haben. Dazu gehören die folgenden Metriken:

### Codezeilen

Die Anzahl der Lines of Code – auch LOC genannt - ist ein bekanntes Maß wenn es darum geht Projektgrößen zu vergleichen. So hat checkstyle beispielsweise 22.369 Codezeilen und Apache Tomcat aber schon 159.364. Letzteres gehört damit sicher zu den größeren Projekten. Dabei ist wichtig zu wissen, ob alle Codezeilen mitgerechnet werden. Werden alle Dateizeilen mitgerechnet so kommt Tomcat auf 314.461 Zeilen was z. B. an Dokumentations- oder Konfigurationsdateien liegen kann.

Codezeilen können in verschiedenen Sprachen eine vollkommen unterschiedliche Expressiveness haben. Darunter versteht man die Ausdrucksstärke einer Sprache. Mit 1 wird oft die Sprache C bezeichnet, moderne dynamische Sprachen haben oft eine Expressiveness von 7 und mehr. Dies bedeutet, dass unter Umständen im Mittel siebenmal so effizient codiert werden kann.

Vergleichen sie beispielsweise die Codezeile in Clojure mit einer möglichen Implementierung in einem alten Basic Dialekt:

```
(filter (fn [x] (= 1 (rem x 2))) (map (fn [x] (* x x)) (range 10)))
```

Die Funktion liefert alle ungeraden Quadratzahlen zwischen 0 und 100.

Bei Codezeilen sollte also immer auch die Expressiveness der Sprache berücksichtigt werden.



Hinweis

### Kommentarzeilen

Anzahl der Zeilen die das Programm beschreiben. Als Daumenregel sollte diese Zahl zwischen 30 % und 60 % der Codezeilen liegen.

### Methoden / Funktionen

Die Anzahl der Methoden oder Funktionen eines Programmes bzw. Projektes.

### Klassen

Die Anzahl der verwendeten Klassen ist natürlich auch ein Komplexitätsmaß. So hatte um das Jahr 2000 das größte deutsche Java-Softwareprojekt (Cheops) etwas über 30.000 Klassen, was schon ganz ordentlich viel ist.

### Packages

Hier werden einfach die Anzahl der Pakete oder Namensräume berechnet.

### Dateien

Dateien sind zwar selbsterklärend, jedoch liefern diese leider kaum ein gutes Maß für die Komplexität einer Software. So gibt es beispielsweise Ruby Programme, die in eine Datei gepackt sind und dennoch sehr komplex sind (z. B. Kirbybase).

### Duplikationen

Einige gute Werkzeuge durchsuchen den Code nach ähnlichen Bausteinen. Das kann auf verschiedenen Ebenen geschehen die nach, identischen Codezeilen, ähnlichen Blöcken oder sogar ähnlichen Dateien suchen. Zudem liefert dies gute Hinweise, ob ein Refactoring durchgeführt werden könnte oder nicht. Häufig wurde dann das DRY-Prinzip (Don't Repeat Yourself) verletzt.

Beispielsweise können gleiche Blöcke in einer Methode zusammengefasst werden, was meistens die Wartbarkeit des Codes deutlich erhöht. Hier ein Beispiel einer automatischen

## Analyse von Basismetriken:

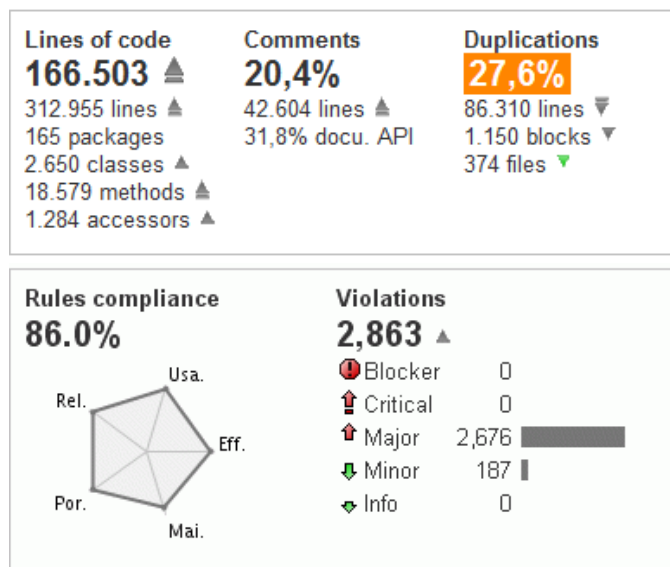
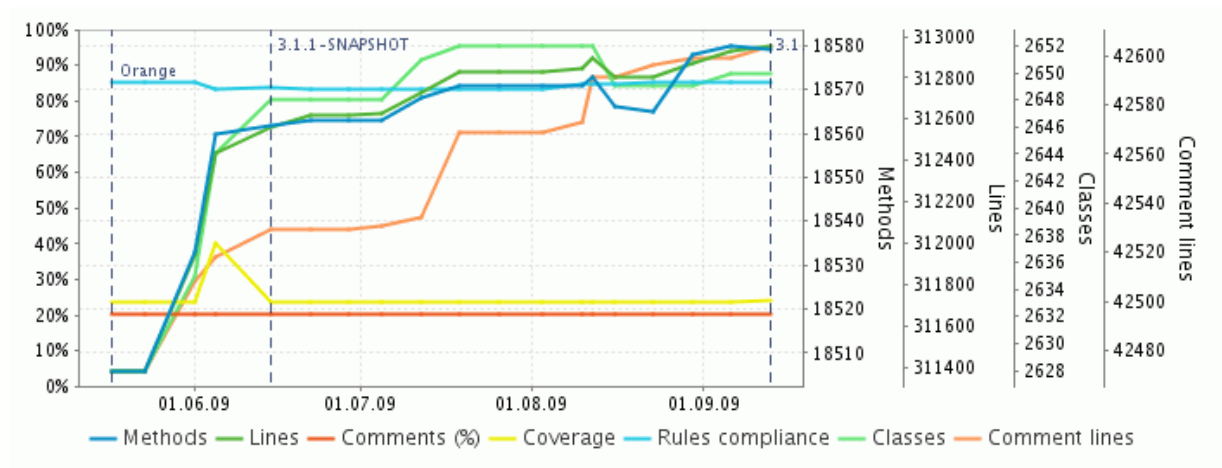


Abb.: Analyse von Basismetriken

Es ist hilfreich, wenn alle Metriken in der Historie verfolgt werden können. Bei guten Werkzeugen ist dies einfach möglich und man erhält graphisch aufbereitete Ausgaben wie die der folgenden Abbildung.



Einige Werkzeuge bieten zudem eine graphische Übersicht an, in der die Metriken als gewichtete Flächen dargestellt werden. Mit einem Klick auf die Fläche (ähnlich der Tools zum Festplattenverbrauch) kann man sich dann in die Region bzw. den Fehler hinein „drillen“.

Die hier aufgeführte Liste wird nie vollständig sein. Im Code wird es immer weitere Merkmale geben, die man untersuchen oder in Metriken verwandeln kann (Anzahl der ToDo-Markierungen, Annotations, etc.).

Abb.: Basis Timeline

## 2.1 McCabe Metrik

Eine der bekanntesten Metriken ist die nach **THOMAS J. MCCABE** benannte McCabe-Metrik. Es handelt sich hierbei um eine reine Code-Metrik, die relativ einfach zu ermitteln ist. Die Idee dabei ist automatisiert zu ermitteln, wie komplex der Code einer Software-Einheit (z. B. Methode oder Klasse) ist, und dies über ggf. viele 10.000 Klassen.



Die Grundannahme besagt, dass sich Komplexität dadurch berechnen lässt, wie viele potenzielle Wege in einem Programm genommen werden können. In diesem Falle bedeutet das, wie viele Kontrollflüsse im Programmgraphen möglich sind.



Definition

### McCabe Cyclomatic Complexity

Jedes konditionelle Konstrukt (sei es eine Schleife oder ein if/else/switch) erhöht die Komplexität um 1. (Daher sollte es wirklich besser Conditional Complexity heißen).

Die Formel lautet:



Formel

$$M = E - N + P$$

Legende:

- M = die Cyclomatic Complexity,
- E = die Anzahl der Kanten des Graphs,
- N = die Anzahl der Knoten im Graph und
- P = die Anzahl der Ausstiegspunkte (return, last command, exit, etc.)

In der Regel wird der Ausstiegspunkt P wieder mit dem Einstiegspunkt verbunden und als Kante mitgezählt.



Beispiel

```
if(a1) code1 else code2 end if(b2) code3 else code4 end
```



Zeichnen / Entwerfen


### Übung MET-01

#### Cyclomatic Complexity

Wie hoch ist die Cyclomatic Complexity im oben gezeigten Beispiel?

Zeichnen Sie den Graphen!

Versuchen Sie die Aufgabe selber zu lösen bevor Sie sich die Musterlösung ansehen.

 [Musterlösung \(Siehe Anhang\)](#)

Bearbeitungszeit: 10 Minuten

### 3 Halstead Metrik

Eine der weniger bekannten Metriken ist die Halstead-Metrik. Dennoch wird sie in vielen Metrik-Werkzeugen implementiert, da sie maschinell einfach zu berechnen ist. Auch sie ist ein statisches Verfahren, bei dem der Code nicht ausgeführt, sondern der Quellcode analysiert wird.

Berechnet werden kann diese Metrik folgendermaßen:

- $n_1$  => die Anzahl der verschiedenen verwendeten **Operatoren** im Code. Dies sind also Befehle wie

```
! != % %= & && || &= ( )
* *= + ++ +=
, - -- -= ->
. ... / /= : ::
< << <= <= = == > >= >> >>=
? [ ] ^ ^= { } | |= ~ (für C++)
```

und **break**, **case**, **return**, **try**, aber auch **private**, **friend**, **static**. Ansonsten aber keine Typen.

- $n_2$  => die Anzahl der verschiedenen verwendeten **Operanden** im Code.  
Also Typdeklarationen (**int**, **bool**, **void**) und alle Konstanten, Typennamen und Identifier, die keine reservierten Namen sind.
- $N_1$  => die Gesamtzahl der Operatoren im Code.
- $N_2$  => die Gesamtzahl der Operanden im Code.

Aus diesen vier Werten lassen sich dann folgende Maßzahlen berechnen:

<b>Programmlänge:</b>	$N = N_1 + N_2$
Größe des verwendeten <b>Vokabulars:</b>	$n = n_1 + n_2$
<b>Halstaed-Volumen:</b>	$V = N * \log_2 n$
<b>Halstaed-Länge:</b>	$L = n_1 * \log_2 n_1 + n_2 * \log_2 n_2$
<b>Schwierigkeit:</b>	$D = (n_1 / 2) * (N_2 / n_2)$
<b>Aufwand:</b>	$E = D * V$

Tab.: Maßzahlen

Die Quellen listen sogar ein Schätzmaß für die Anzahl der zu erwartenden Bugs.

Diese Metrik liefert nur Maßzahlen auf lexikalischer Ebene. Die tatsächliche Komplexität wird daher nur schätzungsweise erfasst. Insbesondere dynamische Programmiersprachen, die mit Konstrukten wie **currying**, **fibers**, **yield** und **closures** eine extreme Komplexität einführen oder vermindern, entziehen sich sicher immer weiter dem Versuch, die Komplexität akkurat zu messen.

Dennoch ist nicht unbedingt die absolute Zahl relevant. Ob ein Modul ein D von 65,25 und ein anderes 67,78 hat ist nicht so bedeutend. Wichtiger ist, ein Gefühl für diese Zahlen zu bekommen und das Überschreiten von Größenordnungen als Alarmsignal richtig werten zu können.

Weitere Quellen zur Halstead-Metrik

<http://www.virtualmachinery.com/sidebar2.htm>

Artikel: Komplexität und Qualität von Software  
VON XAVIER-NOËL CULLMANN und KLAUS LAMBERTZ

[verifysoft.com](http://verifysoft.com) [mscoder.pdf](http://mscoder.pdf) (737 KB)

## 4 Code Coverage / Test Coverage

Bei dem Thema Coverage werden zwei verschiedene Bereiche unterschieden:

1. Die Abdeckung, in der der Code bei dem Ablauf durchlaufen wird.
2. Die Testabdeckung: die kontrolliert/untersucht, wie gut die Tests bspw. die angewendeten Methoden erfassen können.

Wird von Coverage gesprochen, so ist unter Softwaretechnikern meistens die Test-Coverage gemeint. Dennoch sollte man genau spezifizieren, was gemeint ist.

### Code Coverage

Wie oft eine Methode oder ein Codestück durchlaufen wird, wird ebenfalls mittels einer Metrik untersucht. Diese kann sehr nützlich sein, um beispielsweise Abläufe zu prüfen oder um zu debuggen.



Beispiel

#### Beispiel aus der Dokumentation der Programmiersprache D

Hier ein Beispiel aus der Dokumentation der Programmiersprache D, die solch ein Feature direkt in den Compiler eingebaut hat. Nach der Ausführung des Programms werden definierte Codestellen ausgegeben mit der Angabe, wie oft die Zeile durchlaufen wurde:

```

/* Eratosthenes Sieve prime number calculation. */
|
|import std.stdio;
|
|bit flags[8191];
|
|int main()
5|{   int    i, prime, k, count, iter;
|
|   writefln("10 iterations");
22|   for (iter = 1; iter <= 10; iter++)
10|   {   count = 0;
10|       flags[] = true;
163840|       for (i = 0; i < flags.length; i++)
81910|       {   if (flags[i])
18990|           {   prime = i + i + 3;
18990|               k = i + prime;
168980|               while (k < flags.length)
|               {
149990|                   flags[k] = false;
149990|                   k += prime;
|               }
18990|               count += 1;
|           }
|       }
|   }
1|   writefln("%d primes", count);
1|   return 0;
|}
sieve.d is 100% covered

```

(Quelle: [http://www.digitalmars.com/d/2.0/code\\_coverage.html](http://www.digitalmars.com/d/2.0/code_coverage.html))

### Test-Coverage

Test Coverage wurde bereits in der Lerneinheit TST Testen angesprochen. Es ist eine wichtige Metrik, die sich häufig auf Unit-Tests bezieht - aber natürlich können sich Tests auch auf andere Einheiten beziehen:

- Codeblöcke - sind beispielsweise durch ifs, switches oder Schleifen abgedeckt, bzw. getestet worden?
- Sind größere Einheiten wie Klassen oder Packages ausreichend getestet worden? Wenn ja, zu wie viel Prozent?



Derartige Werkzeuge können ideal in ein Buildmanagementwerkzeug (Ant, Maven) eingebaut werden und dann aussagekräftige Reports liefern:

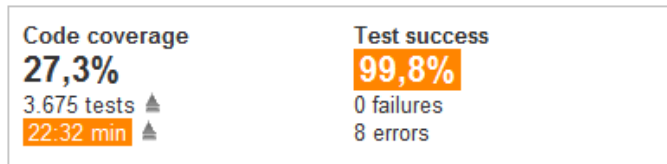


Abb.: Code-Coverage Report

Während der Arbeit am Code kann eine derartige Metrik sehr hilfreich sein. In jeder bekannten Entwicklungsumgebung gibt es daher Plug-Ins, die einem diese Ergebnisse liefern.

Der geneigte Student sei beispielsweise einmal dazu angehalten sich Emma bzw. das eclemma (<http://www.eclemma.org/>) Plug-In zu installieren. Dies geht in wenigen Minuten und man erhält sofort die ersten Anzeigen wie die Abbildung zeigt:

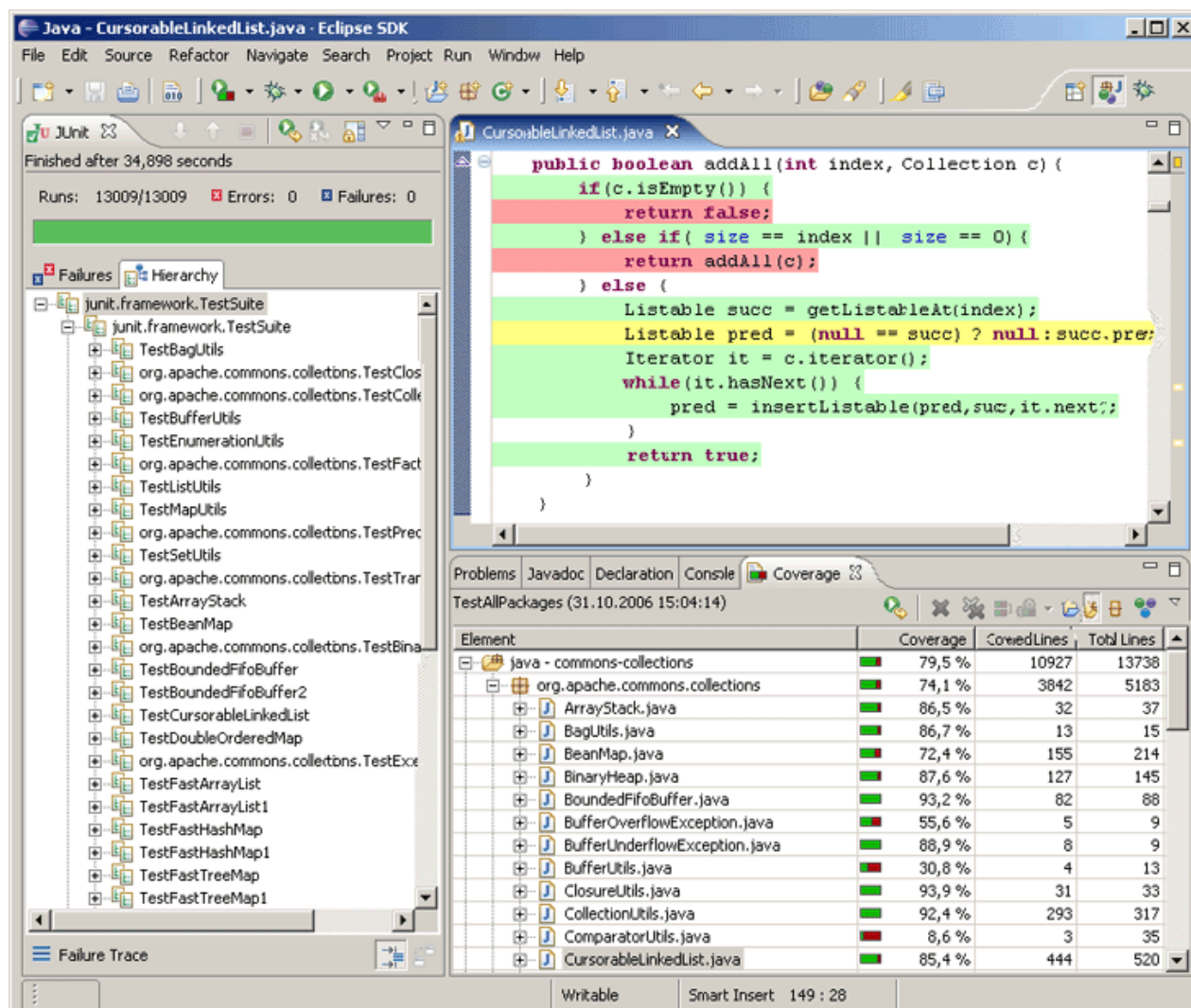


Abb.: Coverage-Metrik in Eclipse

## 5 Regelverletzungen

Für die Prüfung von Code- und Programmierstil gibt es seit vielen Jahren hervorragende Werkzeuge. Verletzungen der Regeln werden meist als Metriken aggregiert angezeigt. Der Entwickler kann sich diese einzeln auflisten lassen und danach die Probleme bearbeiten.

Es gibt viele Arten von Regelverletzungen wie beispielsweise der Verstoß gegen Rahmenbedingungen eines Styleguides oder die Verletzung von Regeln, welche die Community oder man selbst als gültig definiert hat.



Für Java sind die bekanntesten Werkzeuge:

- Checkstyle
- PMD und
- Findbugs

Quellen

Eine Übersicht weiterer Werkzeuge finden Sie im Internet unter:

 <http://java-source.net/open-source/code-analyzers>

Hier nun einige Beispiele, was von den Werkzeugen üblicherweise geprüft werden kann:

Dokumentation in **JavaDoc**:

- Gibt es ein `package.html`?
- Ist der JavaDoc Code gut formatiert?
- Werden Methoden / Variablen auch kommentiert?

Java **Namenskonventionen**:

- Stimmen Sie mit der regular Expression „`^[a-z]+(\.[a-z][a-z0-9]*)*$`“ überein?

Ist der **Header** und sind die **Imports** in Ordnung?

- Vermeidung von Imports mit Stern („`de.vfh.swt.*`“)
- Keine Angabe von falschen Imports
- Keine redundanten Imports
- Keine nicht verwendeten Imports

Angabe von **Größenbeschränkungen**:

- Die Anzahl der ausführbaren Codestücken kann limitiert werden (z. B. die Methodenanzahl)
- Die Dateigröße und Zeilenanzahlgröße kann beschränkt werden
- Die Länge der Methode kann beschränkt werden. Hinweis: Diese Prüfung forciert das Refactoring **Extract Method**.
- Ein Limit für die Anzahl der Parameter ist einstellbar.

Prüfen auf WhiteSpaces oder Tabulatoren,

Prüfen der **Reihenfolge** der **Modifier** in Klassen: Für Java gilt beispielsweise z. B.

1. `public`
2. `protected`
3. `private`
4. `abstract`
5. `static`
6. `final`
7. `transient`

8. **volatile**
9. **synchronized**
10. **native**
11. **strictfp**

Es dürfen keine **leeren Blöcke** vorkommen.

Es werden viele weitere **Coding Standards** geprüft:

- Vermeidung von leeren Statements.
- Lokale Variablen, die ihren Wert nicht ändern sollten **final** sein.
- Gleiche Namensgebung innerhalb von Blöcken (**shadowing**) ist zu vermeiden.
- Verwende keine **Magic-Numbers**. Vermeide nach Möglichkeit den Einsatz überflüssiger und somit ineffektiver Zahlen (außer -1,0,2), und nutze vorrangig Deklaration in **final** Variablen die durch Erklärungen ergänzt werden können.
- Fehlendes **default** im **switch** Konstrukt.
- Kontrollvariablen sollten nicht modifiziert werden:

```
for (int x = 0; x < 1; x++) {x++; }
```

- Unnötige **throws**-Deklarationen
- Findet unnötigen Code:

```
if (b == true), b || true, !false, etc.
```

- Korrekte Namensdefinition für JUnit 3 Tests.
- Gibt es gleiche Strings im Code?

Auch ein gutes Klassendesign kann überprüft werden:

- Sichtbarkeit der Klassenvariablen
- Verwendung von **final**
- Ist die Klasse auf Vererbung ausgerichtet?

Es sind nicht nur **beliebige Regular Expressions** für die Analyse einer Zeile einstellbar. Es sind beliebige Regeln implementierbar und erweiterbar.



Beispiel

#### Beispiel aus Checkstyle für ineffizienten Code

Wie man hier sieht, finden heutige Werkzeuge auch ineffizienten Code. Dazu noch ein Beispiel aus Checkstyle:

```
if (valid())
    return false;
else
    return true;
```

Dieses Beispiel sollte ersetzt werden mit:

```
return !valid();
```

Einige Werkzeuge wie Checkstyle enthalten die folgenden Metriken:

1. Anzahl der **boolschen Ausdrücke** wie **&&**, **||**, **&**, **|** und **^** in einer Anweisung
2. **ClassDataAbstractionCoupling**: Hinweis: Dies entspricht dem **Ce** aus JDepend!
3. **ClassFanOutComplexity**: Hinweis: Dies entspricht dem **Ca** aus JDepend!
4. **CyclomaticComplexity**: Nun die kennen wir bereits.
5. **NPathComplexity**: **Die Anzahl der möglichen Abläufe**. Dies entspricht quasi der **McCabe** Metrik.
6. JavaNCSS: **Anzahl der Codezeilen**. Mit Beschränkungen auf Methode=50, Klasse=1500, FileMax=2000

Frage: Welche Werte halten Sie bei Punkt 6 und den anderen Metriken für sinnvoll?

### Zusammenfassung:

Die Kraft und Nützlichkeit von Styleanalysetools ist für die Code- und Systemqualität nicht zu unterschätzen. Als erfahrener Developer ist man in der Regel erst einmal vor den Kopf gestoßen, wenn zu Beginn viele tausend Fehlermeldungen von diesen Werkzeugen angezeigt werden. Häufig taucht dann der Wunsch auf, in der Konfigurationsdatei einige Tests „auszukommentieren“, wodurch diese ausgeschaltet werden. Besser wäre es hier innezuhalten und sich dies gut zu überlegen. In der Regel ist es wirklich besser, die Ursache im Code zu bearbeiten und nicht den Fehlercheck zu verkleinern.

Werkzeuge wie Checkstyle sollten immer initial im Build mit eingebunden sein oder zur Not auch als PlugIn auf Tastendruck ausgeführt werden. Dies kann zu Beginn zwar mit ein paar Schwierigkeiten verbunden sein, auf die Dauer zahlt sich diese Vorgehensweise, jedoch besonders bei großen Projekten, aus. Insbesondere auch deshalb, weil Styleanalysetools auch Dinge erkennen, die eine gute Architektur forcieren.



Beispiel

### Praxisbeispiel zu Checkstyle

Im Folgenden werden die Konfiguration und die Ergebnisse von Checkstyle gezeigt. Dabei sollte zu Beginn überlegt werden, ob dies unter ANT (oder einem Buildmanagementsystem) oder als Plug-In in der von Ihnen präferierten IDE gemacht werden soll.

- Download von **checkstyle-all-\*.jar**
- Bekanntmachen des Tasks:

```
<taskdef resource="checkstyletask.properties"
        classpath="lib/checkstyle-all-5.0-beta01.jar" />
```

- Definition des Targets::

```
<target name="CHECKSTYLE">
  <checkstyle config="docs/sun_checks.xml">
    <fileset dir="src" includes="**/*.java" />
    <formatter type="plain" />
    <formatter type="xml" toFile="checkstyle_errors.xml" />
  </checkstyle>
</target>
```

- Einbinden der Analysedefinition. Z. B. sun\_checks.xml (ggf. rausnehmen der Dinge die nicht gehen...). Die Datei hat so ungefähr den folgenden Inhalt:

```
<module name="Checker">
...
  <module name="AvoidInlineConditionals"/>
  <module name="DoubleCheckedLocking"/>
  <module name="EmptyStatement"/>
  <module name="EqualsHashCode"/>
  <module name="HiddenField"/>
  <module name="IllegalInstantiation"/>
  <module name="InnerAssignment"/>
  <module name="MagicNumber"/>
  <module name="MissingSwitchDefault"/>
  <module name="RedundantThrows"/>
  <module name="SimplifyBooleanExpression"/>
  <module name="SimplifyBooleanReturn"/>
...
```

Wie man hier sieht, werden für Testoberbegriffe konkrete Tests definiert. Alle Tests sind üblicherweise gut dokumentiert und begründet

(<http://checkstyle.sourceforge.net/availablechecks.html>).

Analyse des Ergebnisfiles **checkstyle\_errors.xml** oder des Outputs auf der Konsole.

Das XML-File sieht dann ungefähr so aus:

```
<?xml version="1.0" encoding="UTF-8"?>
<checkstyle version="5.0-beta01">
  <file name="C:\EDLCHESS\EDLENGINE\src\de\edlchess\cmds\PositionCommand.java">
  </file>
  <file name="C:\EDLCHESS\EDLENGINE\src\de\edlchess\consts\Info.java">
  </file>
  ... noch mehr files...
  <error line="44" column="35" severity="error"
message="&apos;71776119061217280L&apos; sollte durch eine Konstante definiert sein."
source="com.puppycrawl.tools.checkstyle.checks.coding.MagicNumberCheck"/>

  <error line="10" column="9" severity="error"
message="Javadoc-Kommentar fehlt."
source="com.puppycrawl.tools.checkstyle.checks.javadoc.JavadocMethodCheck"/>

  <error line="10" column="29" severity="error"
message="Der Parameter arenaCmd sollte als &apos;final&apos; deklariert sein."
source="com.puppycrawl.tools.checkstyle.checks.FinalParametersCheck"/>

  <error line="33" column="9" severity="error"
message="Die Methode &apos;identify&apos; ist nicht für Vererbung entworfen - muss
abstract, final oder leer sein."
source="com.puppycrawl.tools.checkstyle.checks.design.DesignForExtensionCheck"/>
```

## Links:

[Sun Code Conventions](#)

[Checkstyle](#)

[PDM](#)

[Findbugs](#)

## 6 JDepend

Eine der ältesten und bekanntesten Metriken wurde noch in den 90er Jahren von **MIKE CLARK** vorgestellt. Er lieferte dazu gleich ein Werkzeug zur Codeanalyse, welches sich heute in vielen weiteren Frameworks, IDEs und Tools in vielen Sprachen wiederfindet.

Bereits im Begriff JDepend ist das Wort Depend, was mit Abhängigkeit übersetzt wird, enthalten. Es geht also primär um eine Abhängigkeitsanalyse der Komponenten eines meist größeren Softwaresystems.

Die Aufgabe von JDepend besteht nach den Worten von **MIKE CLARK** darin, folgendes zu tun:

*„JDepend traverses a set of Java class and source file directories and generates design quality metrics for each Java package. JDepend allows you to automatically measure the quality of a design in terms of its extensibility, reusability, and maintainability to effectively manage and control package dependencies. Package dependency cycles are reported along with the hierarchical paths of packages participating in package dependency cycles.“*

Erweiterbarkeit,  
Wiederverwendbarkeit,  
Wartbarkeit

Auch hier muss wieder der Sourcecode geparkt und aus jeder Klasse Metriken gewonnen werden, die im Folgenden vorgestellt werden. Diese Metriken erlauben es die Erweiterbarkeit, Wiederverwendbarkeit und Wartbarkeit des Codes zu messen und zu beurteilen. Besser gesagt geben die Metriken unter Umständen Hinweise darauf, dass der Code u.U. nicht besonders gut erweiterbar, wiederverwendbar oder wartbar ist.

Im Forschungsgebiet der Softwaretechnik wird in diesem Zusammenhang auch von „**Software Stability**“ gesprochen, also davon wie stabil die Software insgesamt ist.

Welches Ziel verfolgen die JDepend Metriken?

1. JDepend fördert **Design by Contract**. Es motiviert, stabile Packages – in Form von Interfaces oder abstrakten Klassen – zu entwickeln. Ziel ist es, dass die instabilen (sich häufig ändernden) Klassen ausschließlich von stabilen Klassen abhängig sind und nicht von anderen konkreten Implementierungen. In diesem Fall können auch verschiedene Personen parallel / gleichzeitig gegen stabile Interfaces programmieren.
2. Die **Unabhängigkeit der Packages** wird dabei gefördert. Dies gilt sowohl für eigene Packages, als auch für externe Packages. Die Abhängigkeiten zu externen Packages (z. B. commons.jar) kann sichtbar gemacht und ggf. isoliert werden. D. h., dass die externe Funktionalität z. B. mit stabilen Komponenten gekapselt werden kann (z. B. Interfaces oder abstrakte Klassen für Facade auf die Bibliothek). Unabhängige, interne Packages können ebenfalls viel besser parallel – u. U. sogar in einem eigenen Releasezyklus – entwickelt werden.
3. JDepend macht **Zyklen sichtbar**. Darauf wird später noch genauer eingegangen. Zyklen sind Komponenten die voneinander zyklisch abhängen. Dies können z. B. Klassen oder Packages sein. Für die Änderbarkeit und Wartbarkeit von Code zeigt die Praxiserfahrung, dass Zyklen „tödlich“ sind.

Schauen wir uns einmal die von JDepend generierten Metriken an:

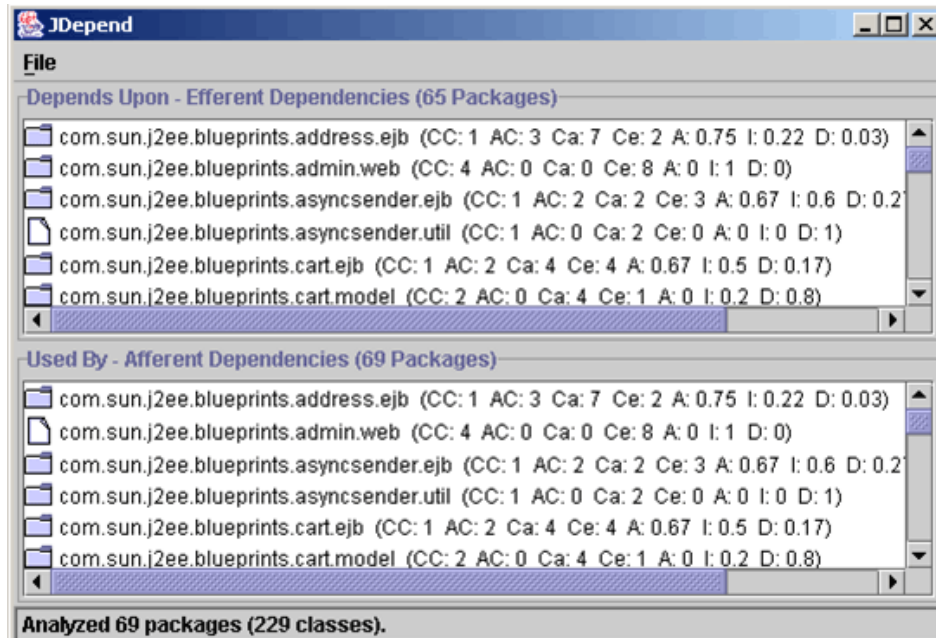


Abb.: Ausgabe der JDepend-Metriken

Wie man sieht, bezieht sich JDepend initial auf Packages, d. h. auf die Abhängigkeitsanalyse von Packages. Jedoch lässt sich der View vergrößern und verkleinern, d. h. man kann durchaus auch auf Klassenebene analysieren. In der Regel wird aber in großen Projekten auf Package-Ebene angesetzt.

## 6.1 JDepend-Metriken

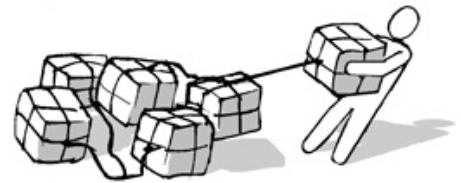
Betrachten wir einige der Metriken von JDepend.

**Die gesamte Anzahl der konkreten Klassen, abstrakten Klassen und Interfaces:** Schon diese Zahl ist ein Indikator dafür, ob das Package beispielsweise bereits zu groß ist.

### Afferent Couplings (Ca):

Afferent Couplings, die auf Deutsch mit „hinbringend“ übersetzt werden können, geben Aufschluss darüber, wie viele andere Packages von dem gerade betrachteten Package abhängen.

Salopp gesagt: bei wie viel anderen Packages wird es „knallen“, wenn ich in diesem Package etwas ändere. Natürlich ist es das Ziel, diese Zahl möglichst gering zu halten. J. Clark schreibt hier sogar von **Verantwortlichkeit** dieses Packages in Bezug auf seine Dienste für andere.



### Efferent Couplings (Ce):

Zu Deutsch „hinausführend / wegführend“ (z. B. aus der Biologie für Nervenzellen). Hier wird angegeben, wie viele andere Packages benötigt meine Klassen von den betrachteten Packages. Salopp gesagt: wie sensibel bin ich wenn sich andere Klassen ändern. Wie stark wird es bei mir „knallen“ wenn sich andere Klassen ändern. Dies scheint eine Maßzahl für die Unabhängigkeit des Packages zu sein.

### Abstractness (A):

Diese Zahl spiegelt ein Verhältnis wieder und die Werte liegen zwischen Null und Eins (einschließlich). Definiert ist A als  $AC/(CC+AC)$ . Dies bedeutet also: Wie hoch ist das Verhältnis an abstrakten Klassen zu allen Klassen. Daher ist ein Paket mit der Metrik 1 ein komplett abstraktes Paket, welches nur definiert aber nichts wirklich tut. Ist die Metrik Null (was leider zu oft der Fall ist), gibt es nur konkrete Klassen. Im Zusammenhang mit der Instability auf Achsen aufgetragen würden die Punkte der Packages auf den Achsen „kleben“.

**Instability (I):**

Die Instabilität ist etwas interessanter. Sie berechnet sich nach der Formel  $I = Ce / (Ce + Ca)$ . Aber was bedeutet dies nun? Im Nenner steht die Gesamtzahl aller Abhängigkeiten alias  $Ce + Ca$ . Also sowohl wie empfindlich andere Pakete auf Änderungen bei mir sind als auch wie empfindlich ich reagiere, wenn andere Pakete sich ändern (wobei ich hier das betrachtete Paket ist). Hier wäre eine 0 ein stabiles Paket und eine 1 ein komplett instabiles Paket.

**Distance from the Main Sequence (D):**

Dies bedeutet übersetzt Abstand zur Hauptlinie. Die Hauptlinie ist eine idealisiert gedachte Linie, für die  $A+I=1$  gilt (siehe auch Bilder auf der nächsten Seite). Diesem Wert wird die Eigenschaft zugeschrieben, zu beschreiben, wie sehr das Paket zwischen Abstraktheit A und Stabilität ausgewogen ist. Optimal ausgewogen wäre ein Paket, das genau auf dieser Linie liegt, d. h. den Abstand d minimiert. Das komplett abstrakte Pakete  $A=1$  ist auch komplett stabil ( $I=0$ ). Auf der anderen Seite ist ein Paket, welches ausschließlich konkret ist sehr instabil, da es konkret gesprochen von keinen Interfaces oder abstrakten Klassen abhängt.

**Package Dependency Cycles:**

Sehr interessant und auch wichtig ist auch die Ausgabe von Zyklen. Wenn Pakete viele Zyklen beinhalten, dann bewirken Klassenänderungen Kettenreaktionen an weiteren Änderungen. Die Entwickler sind dann nur noch am Debuggen und kommen inhaltlich nicht voran. Ein oft zu beobachtendes Phänomen in vielen Unternehmen.

Zyklen

Dabei gibt es bei Zyklen viele Fälle:

- Zyklen können zwischen Klassen sein. In der Regel werden hier Klassen verschiedener Pakete betrachtet. Jedoch sind auch Zyklen zwischen Klassen innerhalb eines Paketes gefährlich. Beispielsweise eine Klasse X aus Paket A benötigt eine Klasse Y aus Paket B. Dies ist ein **direkter Zyklus**.
- Zyklen können auch zwischen **mehr als zwei** verschiedenen Paketen vorkommen. Z. B. **A** -> **B** -> **C** -> **A**. Somit sind alle drei Klassen in einen Zyklus verwickelt.
- Schließlich gibt es indirekte Zyklen. Gibt es im obigen Beispiel noch eine Klasse P, die das Paket A benötigt, so ist auch sie indirekt von diesem Zyklus abhängig und wird entsprechend von JDepend- / Analyse-Tools markiert.

**6.2 Zyklenanalyse als Teil des Buildzyklus**

Werkzeuge die derartige Metriken ermitteln, sollten selbstverständlich in den Build-Zyklus mit eingebunden werden um im Rahmen des Continuous Integration Prozesses den Entwicklern, Projektleitern oder der Qualitätssicherung zur Verfügung zu stehen. Das bedeutet, es kann der gesamte JDepend-Prozess (oder der anderer Werkzeuge, die gleiche oder ähnliche Metriken ermitteln) gestartet und beispielsweise ein \*.txt-Report zum Versand generiert werden. Der Aufruf von JDepend aus Buildmanagementwerkzeugen wie ANT ist daher auch relativ einfach:

```
<target name="jdepend">
  <jdepend outputfile="docs/jdepend-report.txt">
    <exclude name="java.*"/>
    <exclude name="javax.*"/>
    <classpath>
      <pathelement location="build" />
    </classpath>
    <classpath location="build" />
  </jdepend>
</target>
```

Quelle <http://clarkware.com/software/JDepend.html>





Beispiel

Einige andere Werkzeuge ermöglichen es, die Zyklensuche als JUnit Tests einzufügen. Dazu ein Beispiel:

```
import java.io.*;
import java.util.*;
import junit.framework.*;

public class DistanceTest extends TestCase {

    private JDepend jdepend;

    public DistanceTest(String name) {
        super(name);
    }

    protected void setUp() throws IOException {
        jdepend = new JDepend();
        jdepend.addDirectory("/path/to/project/ejb/classes");
        jdepend.addDirectory("/path/to/project/web/classes");
        jdepend.addDirectory("/path/to/project/thirdpartyjars");
    }

    /**
     * Tests the conformance of a single package to a
     * distance from the main sequence (D) within a
     * tolerance.
     */

    public void testOnePackage() {

        double ideal = 0.0;
        double tolerance = 0.125; //project-dependent

        jdepend.analyze();

        JavaPackage p = jdepend.getPackage("com.xyz.ejb");
        assertEquals("Distance exceeded: " + p.getName(),
            ideal, p.distance(), tolerance);
    }

    /**
     * Tests the conformance of all analyzed packages to a
     * distance from the main sequence (D) within a tolerance.
     */

    public void testAllPackages() {

        double ideal = 0.0;
        double tolerance = 0.5; //project-dependent

        Collection packages = jdepend.analyze();

        Iterator iter = packages.iterator();
        while (iter.hasNext()) {
            JavaPackage p = (JavaPackage)iter.next();
            assertEquals("Distance exceeded: " + p.getName(),
                ideal, p.distance(), tolerance);
        }
    }

    public static void main(String[] args) {
        junit.textui.TestRunner.run(DistanceTest.class);
    }
}
```

Quelle <http://clarkware.com/software/JDepend.html>

In dem Beispiel werden im oberen Bereich die Quellen eingelesen, dann JDepend komplett aufgerufen und nachgesehen, ob Zyklen enthalten sind. Diese können dann über die Datenstruktur **p** komplett ausgegeben werden. Auf der Webseite von JDepend sind noch mehr Beispiele aufgeführt, wie zum Beispiel auf **d** getestet werden kann (wird im späteren Verlauf noch erklärt). Auf dem Markt sind Werkzeuge für viele Sprachen erhältlich, die etwas Derartiges leisten (wie z. B. auch **Macker** für Java).

Die folgende Abbildung zeigt, wie schon vor vielen Jahren in Community-Werkzeugen mittels [javaforge.com](http://javaforge.com) die Instability gemessen wurde.

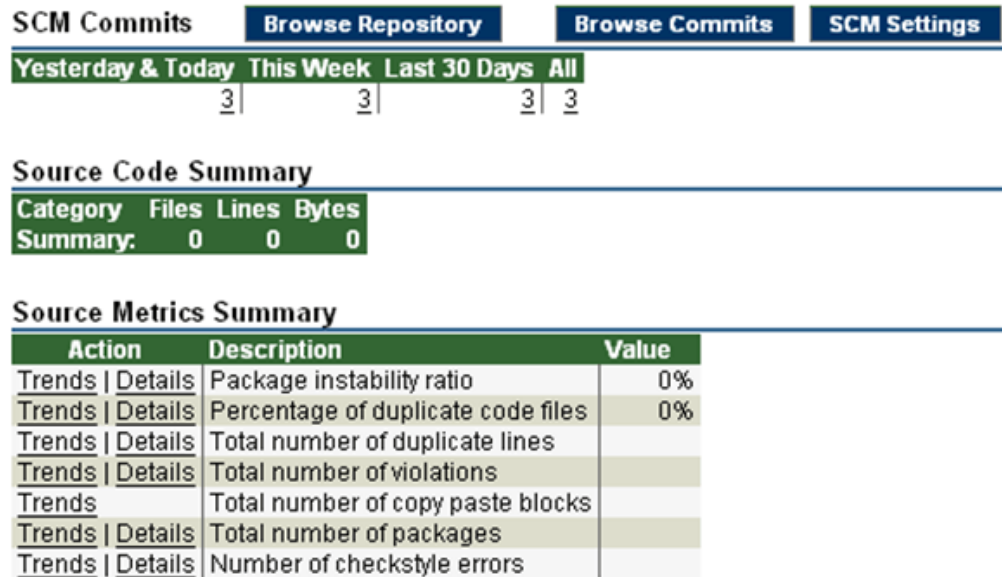


Abb.: Messung der Instability

## Analyse der Grafiken

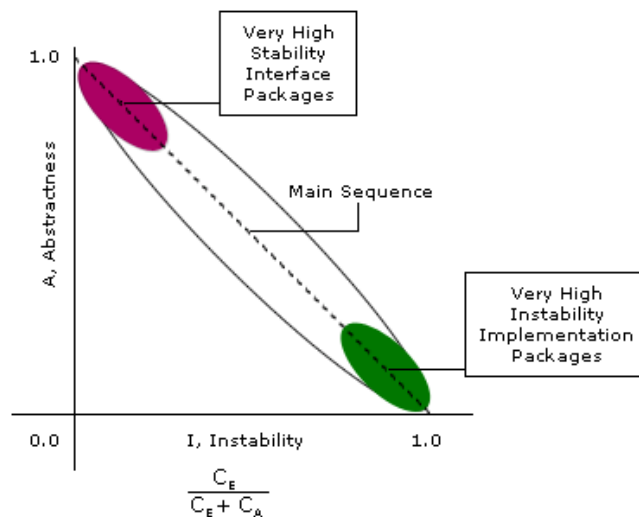


Abb.: Main Sequence Darstellung

In der ersten Abbildung ist die Main Sequence zu sehen, die die Linie **A+I** nachzeichnet. Dabei wird ersichtlich, dass Pakete nicht gleichzeitig komplett abstrakt (**A=1**) und komplett stabil (**I=1**) sein können. Es sollte immer ein ausgewogenes Maß an konkreten Klassen auf der einen Seite und an abstrakten Klassen / Interfaces auf der anderen Seite vorhanden sein.

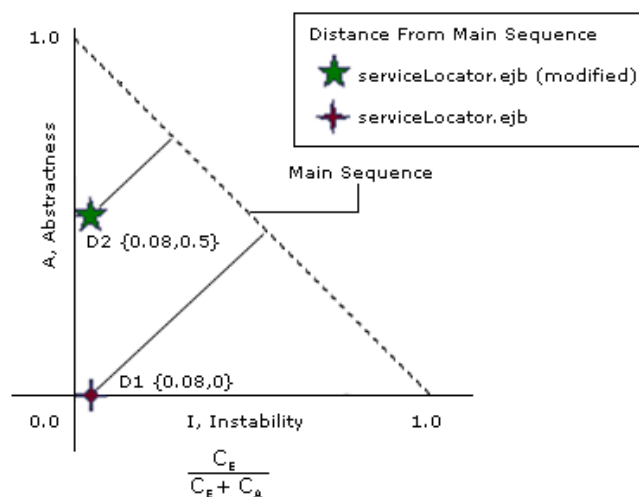


Abb.: Distance from Main Sequence

(Stern: Klasse A) `de.vfh.swt.metriken.cpool`; Kreuz: B) Klasse `de.vfh.swt.metriken.media`)

In der zweiten Abbildung sehen wir ein Paket A, welches recht weit links oben liegt. Daraus lässt sich schlussfolgern, dass offensichtlich viele abstrakte Klassen sowie eine hohe Stabilität vorliegen (I ist klein).

Der zweite Punkt hat wenig abstrakte und zu viele konkrete Klassen. Hier liegt eine hohe Instabilität vor (I ist groß). Von allen Punkten kann jetzt ein  $d$ , also der Abstand zur „Main Sequence“, berechnet werden. Anschließend sollte dieser dann einzeln betrachtet und gegebenenfalls aufsummiert werden.

Pakete mit geringer Abstraktion sollten von Paketen mit hoher Abstraktion abhängen!

In einer idealen Welt würde es nur Implementierungsklassen geben, die nur von abstrakten Klassen abhängen. In der realen Welt muss man aber Kompromisse eingehen...

JDepend misst den Abstand des Packages zur „Main Sequence“  $D$ .



Hinweis

Je größer der Abstand  $D$  des Paketes zu der Main-Sequence ist, desto höher ist die Wahrscheinlichkeit, dass das Package einen Review oder ein Refactoring vertragen kann.

Dies sind natürlich nur Hinweise die konkret geprüft werden müssen und in Sonderfällen auch jeder Grundlage entbehren können. Dennoch zeigt die Erfahrung, dass Pakete mit großem  $D$  durchaus ein Refactoring vertragen können. Ähnlich dem Hinweis einer guten Testabdeckung.

## 7 Kopplungsmetriken

Ein weiterer sehr interessanter Metrik-Bereich ist die Kopplung. Darunter versteht man, wie stark Komponenten miteinander verknüpft sind.

Unterschieden werden zwei Bereiche:

- Cohesion: übersetzt etwa Zusammenhang / Bindekraft
- Coupling: übersetzt etwa Kopplung / Verbindung. Oft auch im Zusammenhang mit Dependency genannt.

Cohesion

Cohesion untersucht beispielsweise die Methoden einer Klasse / Komponente und wie diese in Beziehung zueinander stehen. Dabei ist das Ziel, dass eine Klasse / Komponente eine Aufgabe erbringt und nicht mehrere. Ist dies der Fall, liegt eine hohe Kohäsion vor. Diese Klassen sind wartbarer, änderbarer und natürlich auch einfacher zu verstehen. Auf der anderen Seite liegt dann meistens eine geringe Kopplung vor.



Das gegenteilige Beispiel wäre eine Klasse mit geringer Cohesion. Hier würden die Klasse und deren Methoden viele Aufgaben erfüllen. Meistens liegt dann eine große Kopplung vor, da noch mehr externe Komponenten diese Klasse benötigen. Die vorliegende Komponente ist daher viel schwerer zu ändern und zu verstehen. Ein Refactoring sollte die Klasse so umgestalten, dass diese nur noch eine Aufgabe erfüllt.

Ein Kernelement der Cohesion ist, ob die Komponente sinnvoll in unabhängige Teile zerschnitten werden kann und dann die Menge beider Teile den gleichen Dienst erbringt wie vorher.

Weiterführendes Material ist zu dieser Thematik:

<http://www.aivosto.com/project/help/pm-oo-cohesion.html>

## 7.1 Coupling / Kopplung

Bei der Kopplung gibt es verschiedene Arten der Messung da man verschiedene Kopplungsvarianten betrachten kann und diese zusätzlich unterschiedlich gewichtet werden können. Beispielsweise kann die Kommunikation der Module wie z. B. der Datenaustausch, die Parameterübergabe, die Verwendung weiterer Daten, etc. betrachtet werden.

Hierzu einige Beispiele.



Hinweis

Von **PRESSMANN** stammt ein Ansatz [ Pr09 ] , der wie folgt arbeitet:

- $d_i$ : Anzahl der Eingabeparameter Daten
- $c_i$ : Anzahl der Eingabeparameter Kontrolle
- $d_o$ : Anzahl der Ausgabeparameter Daten
- $d_o$ : Anzahl der Ausgabeparameter Kontrolle
- $g_d$ : Anzahl der global verwendeten Daten
- $g_c$ : Anzahl der global verwendeten Kontrolldaten
- $w$ : Anzahl der aufgerufenen Module (Das entspricht unserem  $C_e$  aus JDepend)
- $r$ : Anzahl der Module, die das betrachtete Modul aufrufen (entspricht dem  $C_a$  aus JDepend)

Bei dieser Metrik werden auch Kontrolldaten betrachtet das heißt, wenn ein Modul einem anderen Modul Steuerungsinformation übergibt, so wird dies stärker gewichtet, als wenn nur einfache Verarbeitungsdaten betrachtet werden.

**PRESSMANN** definiert die Kopplung so:

$$C = 1 - \frac{1}{(d_i + 2 \cdot c_i + d_o + 2 \cdot c_o + g_d + 2 \cdot g_c + w + r)}$$

Eine seiteneffektfreie Methode wäre eine Methode mit einem Eingabeparameter ( $d_i=1$ ), und einem Ausgabeparameter ( $d_o=1$ ) wie z. B. die Berechnung eines Zinswertes. Da diese ja von einem Modul aufgerufen wird, muss ( $r=1$ ) der Nenner in obiger Formel drei sein. Folglich ergibt sich in der Regel ein minimales  $C$  von 0,67.

Maximal strebt dieser Wert natürlich gegen eins.



Beispiel

### Kopplungsansatz von **PRESSMANN**

Nehmen wir noch ein anderes Beispiel:

Es liegen drei Eingabeparameter und ein Ausgabeparameter vor, dazu ein Steuerungsparameter, globale Variablen werden keine verwendet. Das Modul verwendet zwei andere Module und wird selbst an weiteren vier Stellen im Code aufgerufen.

Es ergibt sich:

$$C = 1 - \frac{1}{(3 + 2 \cdot 1 + 1 + 2 \cdot 0 + 0 + 0 + 2 + 4)} = 0,917$$

Sicherlich kann man darüber streiten, ob die angegebene Gewichtung sinnvoll ist. Andere Metriken würden vielleicht  $C_a$  und  $C_e$  stärker gewichten als einfache Eingabeparameter (Daten).

## 7.2 Die ACD Metrik

Eine weitere - noch etwas genauere - Metrik wird von SonarJ gemessen und stammt von **JOHN LAKOS** [lak96]. Dazu holen wir etwas weiter aus.

Bereits in der Lerneinheit „Design“ wurde ein Architekturansatz vorgestellt, bei dem Komponenten in ein Raster eingetragen und logisch gruppiert wurden. Dabei kann sich beispielsweise eine Achse auf das Schichtenmodell beziehen: oben den View zeigen und unten die Datenhaltungsschicht. Auf einer weiteren Achse können weitere Schichten oder logische Domain Objekte stehen. Betrachten wir dazu die aus der Lerneinheit „Design“ bekannte Abbildung.

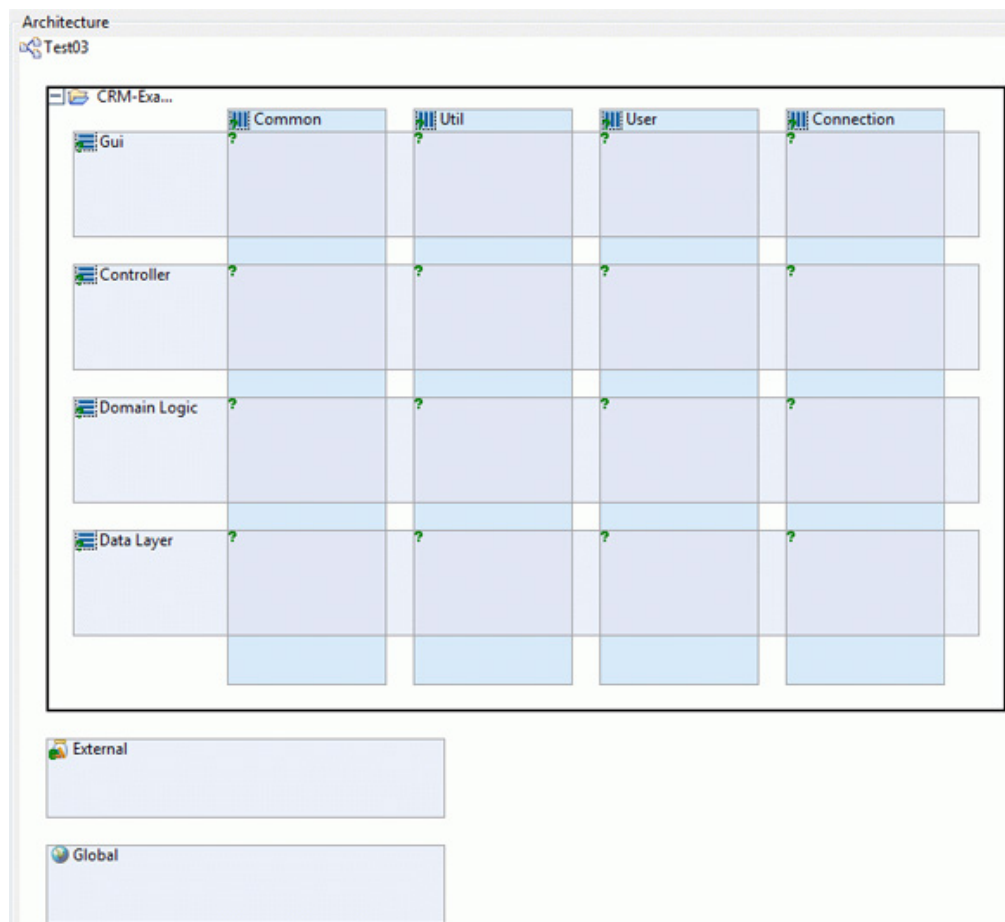


Abb.: Definition einer Architektur

Für Architekten ist es an dieser Stelle wichtig zu wissen, wie der „Aufruf-Fluss“ festgelegt wird. Kann jede Komponente jede andere nach „Spaghetti-Manier“ aufrufen? Oder ist ein Top-Down mit Links-nach-Rechts-erlaubt viel sinnvoller?! Natürlich ist letzteres sinnvoller, weswegen es viele Werkzeuge gibt - wie SonarJ oder Macker – die so etwas testen.

Nimmt man einmal eine Komponente heraus und betrachtet welche anderen Komponenten dadurch mit aufgerufen werden, so lässt sich ein Graph erstellen. Dieser Graph kann sich sowohl auf Packages in großen Systemen als auch auf Klassen in kleinen Systemen beziehen.



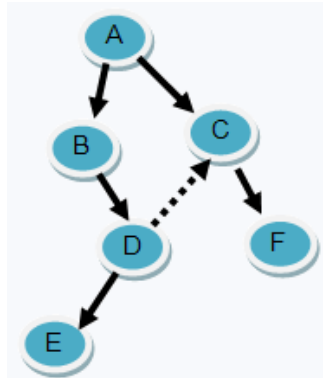
Hinweis

Wie schon in der Lerneinheit Design erläutert, zeigt sich auch hier im Folgenden, dass Packages besser nicht unstrukturiert aufeinander zugreifen sollten. Der Zugriff von Komponente A (oder beispielsweise Paket `de.vfh.swt.A`) auf Package B sollte besser über eine oder wenige Schnittstellen (quasi Facade Interfaces) geschehen, die die Funktionalität gebündelt bereitstellen.

der aufgezeichnete Graph ist seinerseits bereits ein Maß für den Kopplungsgrad. Ein großer Graph mit vielen Zyklen ist logischerweise viel schwerer zu warten und zu ändern, da viel mehr Codestellen betroffen sind. Sinnvoll ist daher immer ein kleiner und zyklischer Graph.

### Berechnung der ACD Metrik

Wie berechnet sich nun die ACD Metrik? Schauen wir uns ein Beispiel an:

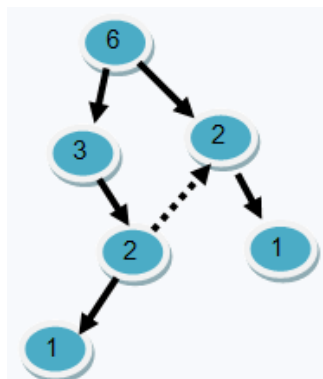


Die Abbildung zeigt einige Komponenten von A bis E, die sowohl Klassen als auch Packages darstellen können. A verwendet B und C. B verwendet D und indirekt auch E. C verwendet F. In der ersten Variante gibt es keine Zyklen. In der zweiten Variante verwendet D auch wieder C, wodurch ein Zyklus erzeugt wird.

ACD steht für Average Component Dependency und beschreibt nach LAKOS sozusagen die mittlere Komponentenabhängigkeit. Dabei wird jedoch nicht zwischen Steuerung und Datenfluss unterschieden.

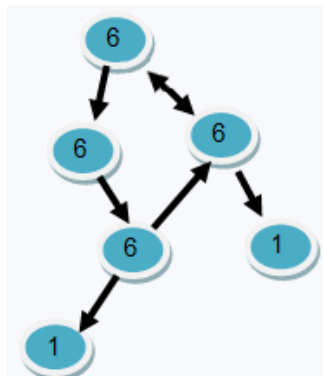
Nach der ACD Methode bekommt initial jede Komponente eine Eins, da sie implizit auch von sich selbst abhängt. Malen Sie sich dazu den obigen Graphen auf Papier und tragen die Zahlen ein. Danach beginnt man von unten und addiert in jeder Komponente die Summe der unterliegenden / abhängigen Zahlen dazu.

Es ergibt sich das folgende Ergebnis für die erste zyklensfreie Variante:



Die ACD-Metrik wird berechnet, indem die Summe aller Werte durch die Anzahl der Knoten geteilt wird! In unserem Beispiel ist **ACD = 15 / 6 = 2,5**. Dies bedeutet, dass im Durchschnitt, jeder Knoten von sich selbst (=1) und weiteren 1,5 Knoten abhängt.

Fügen wir jetzt einen kritischen Zyklus von D nach C und von C nach A hinzu, so ändern sich die Zahlen bedeutend. Wir erhalten den folgenden Graphen:



Wir sehen hier, dass im Zyklus eigentlich jeder Knoten auch von jedem Anderen abhängt. Der betrachtete Kreis erhält daher auf jedem Knoten eine 6 als Wert. Wird nun die neue mittlere



Beispiel

Abb.: Komponentengraph



Formulieren

Abhängigkeit berechnet, so ergibt sich ein ganz anderer Eindruck:

**ACD =  $26 / 6 = 4,33$ .** Ein viel höherer Wert, der sicherlich auf ein Refactoring hinweist.

Was passiert nun, wenn man zwei Graphen vergleicht? Nehmen wir unseren obigen Graphen mit Zyklus und einem Wert von 4,33. Vergleichen wollen wir mit einem Graphen der ähnlich aussieht, über 100 Knoten verfügt aber keinen Zyklus hat.

In diesem Fall hätte der 100-knotige Graph evtl. einen weitaus höheren ACD Wert, einfach wegen der absoluten Anzahl der Knoten. Dennoch könnte dieser Graph „sauberer“ sein als der niederwertigere kleine Zyklengraph. Um vergleichbare Werte zu erhalten, müssen wir nochmals durch die Anzahl der Knoten dividieren. Nur dann erhalten wir einen relativen Wert, der von der Anzahl der Knoten unabhängig ist. Wir nennen ihn daher **rACD** und definieren:

**ACD** = Anzahl der Abhängigkeiten / Anzahl der Knoten

**rACD** = ACD / Anzahl der Knoten

Wir erhalten nun:

Graph 1:  $rACD1 = 2,5 / 6 = 0,4167$

Graph 2:  $rACD2 = 4,33 / 6 = 0,7267$

Es zeigt sich, dass für zyklensfreie Graphen ein maximaler Wert von 0,5 also 50% erreicht werden kann. Besser wäre allerdings ein geringerer Wert.

## Übung MET-02

### Zyklenfreie Graphen

Veranschaulichen Sie sich anhand von Beispielen, warum in zyklensfreien Graphen kaum ein größerer Wert als 50% erreicht werden kann, wenn die Anzahl der Knoten wächst!

Bearbeitungszeit: 15 Minuten

Die Erfahrung zeigt, dass ein rACD-Wert bei größeren Graphen ein sehr wahrscheinlicher Hinweis auf Zyklen in den betrachteten Graphen ist.

Das Sonar-Team (nach ZITZEWITZ, Java-Magazin 2.2007, S.30) kommt zu der Erkenntnis, dass aus der Erfahrung weitere Formeln abgeleitet werden können, die gute Werte für rACDs – in Abhängigkeit von der Knotengröße – liefern:

- Bei 200 Knoten sollte der rACD unter 15% liegen
- Bei 1000 Knoten sollte der rACD unter 7,5% liegen

Es kann folgender nützlicher Zusammenhang hergestellt werden:

```
<FORMEL>
  rACDmax=1,5*(1/(2^log5(n)))   wobei n > 200
</FORMEL>
```

## 7.3 Zusammenfassend

Mit den hier vorgestellten Kopplungsmetriken wird automatisiert ein guter Überblick über die „Verzahnung“ von Komponenten gegeben. Hilfreich sind zyklensfreie Komponenten mit einem möglichst geringen rACD.

In modernen Werkzeugen werden diese Metriken schnell ermittelt und können geeignet farblich dargestellt werden. Dabei sehen Entwickler und Architekten sofort, welche Pakete oder Komponenten kritisch sind, so dass ein Refactoring vorgenommen werden kann.

Ideal ist, wenn derartige Metriken von Beginn an greifen und mit den Werkzeugen sofort Alarm schlagen, da dies das natürliche Wachstum einer guten Architektur fördert.



## 8 Technical Debt

Schon im Jahre 1992 kam **WARD CUNNINGHAM** auf die Idee, die Menge an Fehlern, an Regelverletzungen sowie die Komplexität als „**Technical Debt**“ zu bezeichnen. Frei übersetzt bedeutet dies, dass man eine Schuld eingeht, weil auf der technischen Seite etwas verbessert werden kann.

Diese Schuld kann verschiedene Ursachen haben. Beispielsweise:

- schlechte Architektur / Design
- viele Codemetrikverletzungen
- zu hohe Komplexität
- zu große Kopplung
- Zyklen

Einige Fehler sind möglicherweise leichter zu beheben. Jedoch gibt es auch eine intrinsische Komplexität, die meistens schwerer zu auflösen bzw. zu verringern ist. Wieder andere Problembereiche sind von langfristiger (Architektur) und manche sind von kurzfristiger Natur (fehlendes JavaDoc), da dies leichter zu beheben werden kann.

Die gesamte Problematik spricht einen Themenbereich an, der in der Industrie oft zu finden ist. Es besteht der Druck mit einem Produkt schnell auf den Markt zu kommen. Daher wird eine Lösung „*Quick-and-Dirty*“ implementiert. Das Produkt wird mitsamt seiner Schuld ausgeliefert, also auch dem Dirty-Teil der in dem Code steckt. Analog zum Finanzwesen wurde ein Kredit aufgenommen (Debt) und nun sind Zinsen zu zahlen. Die mehr oder weniger hohen Zinsen werden gezahlt, weil das Produkt - in der Regel - weiterentwickelt werden muss. Je höher jedoch die Anzahl der Verletzungen ist, desto schwieriger wird es das Programm weiterzuentwickeln und zu ändern. Für das Debugging müssen die Entwickler nun viel mehr Zeit investieren.

Das Ziel für den Verkäufer einer Software ist idealerweise, das Produkt mit einer Schuld (Debt) von Null auszuliefern. Die Erwartungshaltung des Käufers ist, ein Produkt mit möglichst geringem Technical Debt zu kaufen.

Über diese Thematik wird an vielen Stellen u.a. in Foren gerne diskutiert. Interessant ist auch, dass moderne Werkzeuge versuchen, die Schuld als Geldwert auszudrücken. Das heißt, dass alle genannten Metriken in Manntage umgerechnet werden. Wie viele Manntage werden benötigt, um die Vielzahl der Code- und Architekturmetrikfehler zu beseitigen? Manntage wiederum lassen sich wiederum recht einfach - je nach Lohnniveau - als Geldwert umrechnen.



Hinweis

Ausschlaggebend ist, dass sich Softwarekosten nicht nur aus Entwicklungskosten zusammensetzen, sondern diese im Einzelfall noch viel höhere Wartungskosten, Einarbeitungskosten, Änderungskosten, etc. mit sich bringen, die es zu berücksichtigen gilt!

Gezielt geschieht dies mit dem Werkzeug Sonar folgendermaßen:

**Debt(in man days) =**

```
cost_to_fix_duplications +
cost_to_fix_violations +
cost_to_comment_public_API + cost_to_fix_uncovered_complexity +
cost_to_bring_complexity_below_threshold
```

Wobei gilt:

<b>Duplications</b> =	$\text{cost\_to\_fix\_one\_block} * \text{duplicated\_blocks}$
<b>Violations</b> =	$\text{cost\_to\_fix\_one\_violation} * \text{mandatory\_violations}$
<b>Comments</b> =	$\text{cost\_to\_comment\_one\_API} * \text{public\_undocumented\_api}$
<b>Coverage</b> =	$\text{cost\_to\_cover\_one\_of\_complexity} * \text{uncovered\_complexity\_by\_tests}$ (80% of coverage is the objective)
<b>Complexity</b> =	$\text{cost\_to\_split\_a\_method} * (\text{function\_complexity\_distribution} \geq 8) + \text{cost\_to\_split\_a\_class} * (\text{class\_complexity\_distribution} \geq 60)$

Tab.: Sonar – Kostenberechnung

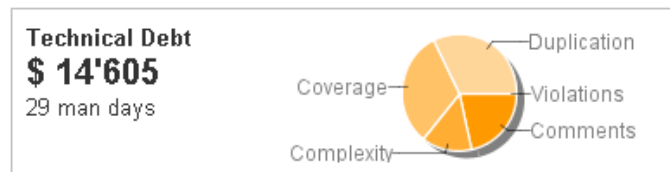


Abb.: Sonar - Berechnung der Manntage

Die Zahl soll die Verantwortlichen oder Entwickler dazu motivieren, in Qualität zu investieren. Dies ist besonders sinnvoll, wenn aus dem Technical Debt nicht unbedeutende „Zinszahlungen“ (Aufwand) resultieren.

Natürlich ist klar, dass diese absolute Zahl sehr wahrscheinlich nichts mit der Realität zu tun hat. Es mag sogar Fälle geben, wo ein hoher Technical Debt nicht weiter schlimm ist. Und auch hier ist es wie mit den JDepend-Metriken (z. B. dem d). Entscheidend ist nicht die absolute Zahl, sondern die Erfahrung, die mit derartigen Grafiken und Zahlen gewonnen wird (welche sich jederzeit mit einem einfachen Klick weiter detaillieren lassen).

Derartige Metriken und Werkzeuge sensibilisieren für den Qualitätsfaktor. Dieser ist in der Regel genauso wichtig, wie als Erster mit einem Produkt auf dem Markt zu sein. Qualität steht mit an oberster Stelle, wenn es darum geht, den Markt längerfristig zu beherrschen und sich gegenüber der Konkurrenz zu behaupten.

#### Links:

[!\[\]\(c50c8b7b2cc2cf9ff925edec0ee94c0d\_img.jpg\) Matrin Fowlers Kommentar](#)

[!\[\]\(6a9b39b98eb945faa14c645ec99e4eaa\_img.jpg\) Announcement von Sonar](#)

## 9 Werkzeuge

Die bisher erwähnten Werkzeuge boten zumeist reine Codemetriken an. Für die Verbesserung der Architektur gibt es Programme, die die low-level Analysen von Checkstyle oder PMD gleich mit integrieren. Auf dieser Seite werden beispielhaft solche Werkzeuge für die Sprache Java vorgestellt. Für .NET und auch andere Sprachen wie Ruby oder Python gibt es ganz ähnliche Werkzeuge.

Lizenztypen

Die Werkzeuge teilen sich in zwei Lizenztypen: kommerzielle und freie Programme. Die Freien unterteilen sich wiederum in drei Kategorien anhand ihrer Fähigkeiten:

1. Allzweckwerkzeuge,
2. rein physische Abhängigkeitenanalyse und
3. Abhängigkeitenanalyse gegen eine logische Struktur.

Werkzeuge mit grafischer Oberfläche

Die hier aufgeführten kommerziellen Werkzeuge haben alle eine grafische Oberfläche.






- **SonarJ:** Das teuerste und zugleich mächtigste Werkzeug stammt von der deutschen Firma Hello2morrow ([www.hello2morrow.com](http://www.hello2morrow.com)). Die Architektur wird horizontal und vertikal eingeteilt, um in den Ebenen Subsysteme herauszubilden. Damit werden Java-Artefakte (Klassen und Pakete) verknüpft. Für Entwickler gibt es ein Eclipse-Plugin, das bereits zur Entwicklungszeit Architekturverletzungen als Problem anzeigt. Klassenbeziehungen lassen sich virtuell wegnehmen, so dass Auswirkungen von Refaktorisierungen gleich angezeigt werden. SonarJ arbeitet auf Klassendateien. Der Quellcode wird eingebunden, um daran Abhängigkeiten zu erläutern. Ein Ant Task erzeugt HTML-Berichte und stoppt den Build bei Architekturverletzungen. Einige Metriken werden in der Oberfläche und in den Berichten aufgeführt.
- **Lattix LDM:** Die US-amerikanische Firma Lattix stellt LDM her. Diese organisiert die Struktur in Matrizen nach dem DSM12 Ansatz ([http://en.wikipedia.org/wiki/Dependency\\_Structure\\_Matrix](http://en.wikipedia.org/wiki/Dependency_Structure_Matrix)). Da sich keine logische Struktur definieren lässt, findet LDM demnach keine Architekturverletzungen. Folglich ist eine Automatisierung nicht möglich. An Paketen lassen sich Regeln aufstellen, auf welche zugegriffen werden kann. Es sollen sich UML-Diagramme im Austauschformat XML einlesen lassen. Im Zusammenhang mit MDA-Projekten soll die Struktur analysierbar sein. Jedoch scheint schon der Import einfacher Diagramme aus MagicDraw im XML-Format problematisch zu sein.
- **Structure101:** Das Programm stammt von der irischen Firma Headway Software. Wie Lattix folgt Structure101 dem DSM-Ansatz, und bietet darüber hinaus mehr Visualisierungen der Pakete an. Ein logisches Modell baut auf der Java Pakethierarchie auf. Structure101 geht davon aus, dass Pakete auf gleicher Ebene in der Pakethierarchie die gleiche logische Bedeutung haben. Aufgrund des starken Bezugs der logischen zur physischen Struktur konzentrieren sich die meisten Ansichten nur auf Klassen und Pakete. Auf jeder Paketebene identifiziert es Bereiche, die keinen Zusammenhang zu anderen Paketen haben. Es wendet nur wenige Metriken an. Ein Einbau in ein Build-System ist über die Kommandozeile möglich. Ein Eclipse-Plugin weist zur Entwicklungszeit auf Architekturverletzungen hin.
- **STAN:** Der Hersteller Odyssees Software [Ody] stellt STAN (Structure Analysis for Java) her. Es ist als Eclipse-Plugin und als separate Anwendung erhältlich. Die Definition logischer Strukturen wird durch STAN nicht gestattet, weshalb alle Analysen auf Paketebene geschehen, auf die diverse Metriken angewendet werden. Dabei lassen sich umfangreiche Berichte mit Diagrammen generieren. Mit der Testversion lässt sich nur ein beigelegtes Testprojekt begutachten.

Bis auf JDepend hat keines der nachfolgenden, freien Werkzeuge eine grafische Oberfläche. Sie sind dafür konzipiert, beim Build mitzulaufen und einen Bericht, meist im HTML-Format, zu generieren. Die freien Allzweckwerkzeuge untersuchen die Architektur auf verschiedene Weise. Beide nachfolgenden Werkzeuge binden eine Vielzahl anderer freier Werkzeuge ein.

- **Sonar:** Sonar (<http://www.sonar.codehaus.org/>) bezeichnet sich als „Code quality management platform“. Ein Werkzeug für die Analyse von Abhängigkeiten fehlt noch, steht aber auf der Anforderungsliste. Viele Metriken werden angewendet, die zum Teil grafisch angezeigt werden. Eine Besonderheit ist, dass jedes Analyseergebnis persistiert wird, um es später auf Zeitleisten anzuzeigen. Sonar läuft als eigenständiger Server und bringt als Benutzerschnittstelle eine Webanwendung mit. Das zu untersuchende Projekt muss leider für Maven konfiguriert sein. Die grafische Aufbereitung von Sonar ist sehr informativ und ansprechend. Sonar ist die einzige Plattform die den Technical Debt berechnet und in Form einer Geldschuld anzeigt.
- **XRadar:** XRadar (<http://xradar.sourceforge.net/>) benutzt mit JDepend und DependencyFinder zwei Werkzeuge, um Abhängigkeiten zu untersuchen. Eine logische Struktur ist in Schichten und deren Subsysteme definierbar. Im Rahmen des Builds mit Maven oder Ant erzeugt XRadar umfangreiche Berichte (Spinnendiagramm) und prüft auf Architekturverletzungen. Die logische Struktur sowie andere Diagramme werden dabei grafisch bereitgestellt. Es bietet diverse Metriken, für die sich Grenzen konfigurieren lassen. Die Konfiguration für ein anderes als das mitgelieferte Testprojekt ist nicht einfach, da Fehlermeldungen im langen Protokoll untergehen. Die Berichte von mehreren Ausführungen wertet XRadar aus, um einen historischen Überblick zu geben. Darin enthalten sind diverse Diagramme über die Entwicklung von Metrikergebnissen.
- **JarAnalyzer:** JarAnalyzer betrachtet eine Jar-Datei als Komponente und analysiert die Abhängigkeiten zu anderen Jar-Dateien (<http://www.kirkk.com/main/Main/JarAnalyzer>). Als Ergebnis erhält man einen Bericht und eine dot-Datei. Die dot-Datei lässt sich in eine Grafik umwandeln, die Jar-Dateien mit ihren Abhängigkeiten darstellt.
- **JDepend:** Es analysiert die Klassendateien und erstellt einen Bericht mit Abhängigkeitsmetriken (<http://clarkware.com/software/JDepend.html>). Alle Angaben beziehen sich auf die Paketebene, so auch die genutzten und abhängigen Pakete. Zusätzlich lässt sich über den Umweg einer dot-Datei ein Diagramm erzeugen. Es zeigt jedes Paket und deren Beziehungen an.
- **DependencyFinder:** Es (<http://depfind.sourceforge.net/>) bringt einen Satz an Untersuchungswerkzeugen mit. Auf Metriken werden nur wenige angewendet. Eine komplette Untersuchung dauert verhältnismäßig lange und ergibt eine XML-Datei von 100 MB. Es werden daraus HTML-Berichte erzeugt.

Die freien Abhängigkeitswerkzeuge logischer Strukturprüfung liefern ausschließlich Ergebnisse in Textform:

- **dependometer:** (<http://dependometer.sourceforge.net/>) Eine logische Struktur lässt sich in Schichten und fachlichen Schnitten vornehmen. Ein Durchlauf bricht wiederholt bei maximaler Speicherzuweisung ab und hat bis dahin 4 GB Daten auf die Festplatte geschrieben.
- **Architecture Rules:** (<http://dependometer.sourceforge.net/>) Die Konfiguration der logischen Struktur ist nicht gut dokumentiert. Folglich wird auch kein Ergebnisbericht erzeugt. Die Ausgaben von Architecture Rules lassen verwertbare Details vermissen.
- **SA4J:** SA4J (Structural Analysis for Java) stammt von AlphaWorks / IBM (<http://www.alphaworks.ibm.com/tech/sa4j>), allerdings ist die Entwicklung im Betastadium 2004 eingestellt worden. Die Installation des Programms ist kompliziert.

- **Macker:** ( <http://innig.net/macker>) Macker prüft beim Build lediglich auf Verletzungen und fertigt darüber einen HTML-Bericht an. Eine logische Struktur lässt sich in Komponenten erstellen. Die Konfiguration von Paketmustern ist mächtig, für Komponentenstrukturen ist sie allerdings nicht gut dokumentiert. Der Bericht und eine Prüfung gegen die logische Struktur laufen schnell. Bei Verletzungen erhält das Build-System eine Fehlermeldung mit den nicht erlaubten Abhängigkeiten.
- **japan:** ( <http://japan.sourceforge.net/>) Die Definition der logischen Struktur lässt sich auf Paketebene ohne Angabe von Mustern erledigen. Jede Abhängigkeit muss einzeln beschrieben werden. Neben Ant wird IntelliJ unterstützt. Einen Bericht gibt es nicht. Die Ausgaben sind nicht verwertbar.
- **Classycle:** ( <http://classycle.sourceforge.net/>) Ein HTML-Bericht gibt die Abhängigkeiten detailliert wieder. Die logische Struktur lässt sich in Schichten und Komponenten definieren. Der Bericht und eine Prüfung gegen die logische Struktur laufen schnell. Bei Verletzungen erhält das Build-System eine Fehlermeldung mit den nicht erlaubten Abhängigkeiten.

Ohne Berücksichtigung der Kosten wäre wahrscheinlich SonarJ das beste Werkzeug. Bei den kostenfreien Programmen empfehlen sich Sonar, XRadar, Classycle und JarAnalyzer für eine weitere Prüfung.

Die folgenden beiden Tabellen fassen die Ergebnisse zusammen. Dabei steht Kom. für kommerzielle Software und FOSS für Free Open Source Software. Die Zeile Versuch gibt einen Eindruck des Aufwandes, das Werkzeug erfolgreich laufen zu lassen (++ = hoher Erfolg).

Kriterien/ Produkt	SonarJ	Lattix LDM	Structure 101	STAN	Sonar	XRadar	JarAnalyzer
Version	4.0	5.0	3.2	1.0.3	1.9	1.0	1.2
Ausgabejahr	2009	2009	2009	2009	2009	2008	2007
Lizenz	Kom.	Kom.	Kom.	Kom.	FOSS	FOSS	FOSS
Definition logischer Struktur	++	--	O	--	--	+	--
Vergleich logisch / physisch	++	--	+	--	--	++	--
Visualisierung der Abhängigkeit	++	+	+	+	--	O	+
Zyklen finden	++	+	--	O	--	+	O
Metriken	+	O	O	+	++	++	+
Berichte	O	+	O	+	++	+	+
Ergebnis im Quellcode	+	-	+	O	++	--	--
Einbau Build-System	++	--	O	+	O	++	++
Unterstützung OSGi	--	--	--	--	--	--	--
Kosten	--	-	-	-	++	++	++
Versuch	++	O	O	-	O	O	++

Tab.: Software-Werkzeuge I

Kriterien/ Produkt	JDepend	Dependency Finder	Architecture Rules	Macker	japan	Classycle
Version	2.9.1	1.2.1	2.1.1	12	0.2.4	1.3.3
Ausgabejahr	2004	2008	2008	2003	2005	2008
Lizenz	FOSS	FOSS	FOSS	FOSS	FOSS	FOSS
Definition logischer Struktur	--	--	O	O	-	+
Vergleich logisch / physisch	--	--	--	+	-	+
Visualisierung der Abhängigkeit	O	--	--	--	--	--
Zyklen finden	+	O	--	--	--	+
Metriken	+	O	--	--	--	-
Berichte	+	-	--	-	--	+
Ergebnis im Quellcode	--	--	--	--	--	--
Einbau Build-System	++	+	+	++	++	++
Unterstützung OSGi	--	--	--	--	--	--
Kosten	++	++	++	++	++	++
Versuch	+	O	-	-	-	++

Tab.: Software-Werkzeuge II

## Zusammenfassung

- Metriken sind ein wichtiges Hilfsmittel der Qualitätssicherung. Interessant ist meist nicht die absolute Metrik, sondern der Hinweis auf ein Problem und die Relationen der Metriken zueinander.
- Schon mit einfachen Metriken wie McCabe oder Halstaed lassen sich interessante Komplexitätsmetriken in Bezug auf den Code berechnen.

Metriken für Regelverletzungen im Code bilden einen weiteren großen wichtigen Bereich. Hier gibt es eine Vielzahl von Werkzeugen, die einem „über die Schulter“ schauen können.

- Geht es um Zyklen, abstraktes Programmieren und die Abhängigkeitsanalyse von Paketen ist JDepend eines der wichtigsten grundlegenden Werkzeuge. Viele der Prinzipien wurden in andere Werkzeuge integriert.
- In der Architekturanalyse gibt es Werkzeuge, die es ermöglichen Zugriffe zu prüfen und so ein besseres Design zu forcieren (z. B. nur Top-Down Zugriffe erlauben).

Mit weiteren Metriken wie dem rACD-Maß kann die intrinsische Modulkomplexität gemessen werden. Diese Metriken können Architekten wertvolle Hinweise auf mitunter nachteilige Designentscheidungen geben.

- Mit dem Technical Debt stehen mittlerweile neue interessante Metriken zur Verfügung, um die Gesamtheit aller Verletzungen – im Bereich Codemetriken und Architekturmetriken – zu messen und als Aufwandsschuld (in \$ oder €) anzugeben.

---

Sie sind am Ende dieser Lerneinheit angelangt. Auf den folgenden Seiten finden Sie noch die Übungen zur Wissensüberprüfung.

## Wissensüberprüfung



Zuordnung

## Übung MET-03

## McCabe Metrik

Die Komplexität eines Codes lässt sich nach McCabe über die Formel:

$M = E - N + P$  berechnen.

Wofür stehen die einzelnen Parameter. Ordnen Sie zu!

A

M =

B

E =

C

N =

D

P =

☐

Anzahl der Knoten  
im Graph

☐

Anzahl der Kanten  
des Graphs

☐

Anzahl der  
Ausstiegspunkte  
(return, last  
command, exit,  
etc.)

☐

Cyclomatic  
Complexity

? Test wiederholen Test auswerten



Zuordnung

## Übung MET-04

## JDepend - Metriken

Welche Aufgaben erfüllen die aufgeführten JDepend-Metriken? Ordnen Sie zu!

A

Diese Maßzahl sagt, wie viele andere Packages  
von dem gerade betrachteten Package  
abhängen.

☐

Distance from the  
Main Sequence  
(D)

B

Hiermit wird angegeben, wie viele andere  
Packages benötigen alle meine Klassen in dem  
betrachteten Package.

☐

Afferent Couplings  
(Ca)

C

Diese Zahl spiegelt ein Verhältnis wieder und  
die Werte liegen zwischen Null und Eins.  
Definiert ist sie als  $AC / (CC + AC)$ .

☐

Instability (I)

D

Diese Metrik berechnet sich nach der Formel  $I = Ce / (Ce + Ca)$ .

☐

Package  
Dependency  
Cycles

E

Diesem Wert wird die Eigenschaft  
zugeschrieben, zu beschreiben, wie sehr das  
Paket zwischen Abstraktheit A und Stabilität  
ausgewogen ist.

☐

Efferent Couplings  
(Ce)

F

Ausgabe von Zyklen.

☐

Abstractness (A)

? Test wiederholen Test auswerten



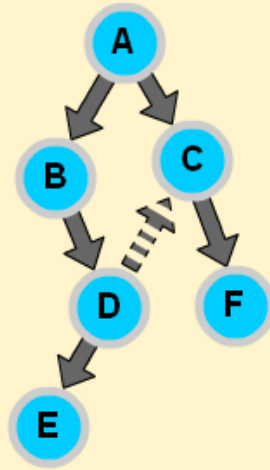
Zuordnung

## Übung MET-05

## ACD Metrik

Auf der rechten Seite sehen Sie einen Komponentengraphen mit Zyklus. Ordnen Sie die Zahlen den Komponenten entsprechend der ACD Methode so zu, dass ein zyklensfreier Komponentengraph entsteht.

Komponentengraph 1



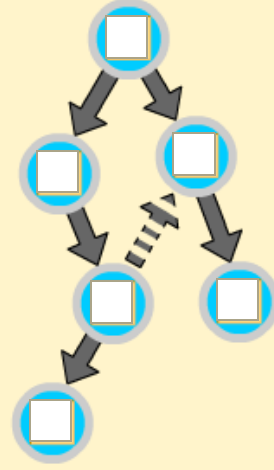
6

2

1

3

Komponentengraph 2



2

1

[? Test wiederholen](#) [Test auswerten](#)



## Appendix

---

### Lösung Übung 12-01

$$M = 9 - 7 + 1 = 3$$

Üblicherweise hat Code mit einer höheren McCabe Complexity auch eine niedrige Cohesion (siehe nachfolgende Kapitel).

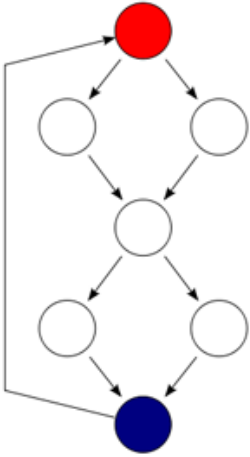


Abb.: Graph Übung 1201