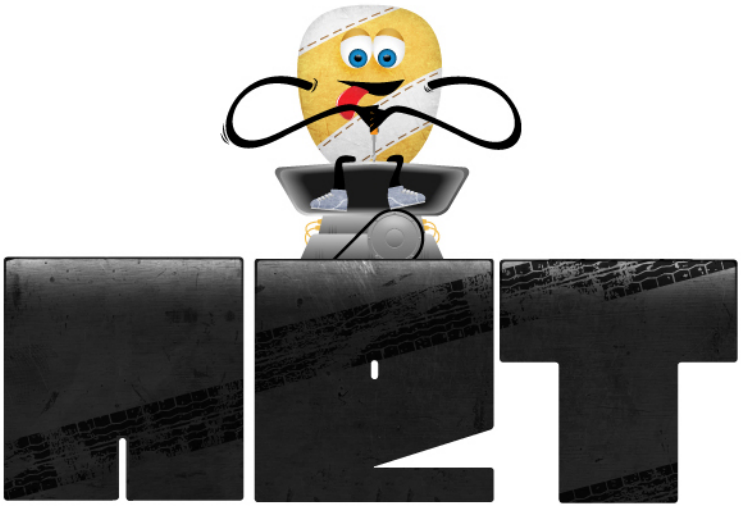
3 MET - Software- und Architekturmetriken



Title: Software- and Architecture Metrics

Autor: Prof. Dr. Stefan Edlich

3.1 Überblick und Lernziele MET

Title: Overview and Goals MET

The "Software Metrics" learning unit is a supplement to the topic of software quality, which also includes the "Testing" learning unit. The point is to make missing software quality or software complexity visible. Violations of rules or problems in the code

are listed and optimally visualized. With this help, developers, architects or project managers can set the course for new quality better.


Lernziele

Lernziele

This session has two objectives:

1. They learn about the importance of metrics as a measure of quality and how to assess them in practice.
2. You'll learn the basics and how to calculate them.
3. You'll learn about code metrics and how to use the tools.
4. You'll learn about architectural metrics and use tools to visualize them.


Gliederung

Gliederung

- After a short introduction to metrics, simple metrics such as McCabe metrics are first treated. After the Halstead metrics and code coverage, the different types of rule violations are discussed, which are aggregated by many tools as metrics.
- In the second part, JDepend, coupling metrics and the question of guilt - the Technical Debt - follow.
- Finally, current available tools for Java are listed and compared.


Zeitumfang

Zeitumfang

It takes about 120 minutes to work through this session. The application of a tool such as XRadar, Sonar or SonarJ to an existing project takes three hours each.

3.2 Literatur MET

Links

- The Classis: http://www.laputan.org/mud/
- http://akvo.org/blog/the-ball-of-mud-transition/
- http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/
- http://blog.sei.cmu.edu/post.cfm/field-study-technical-debt-208

3.3 Motivation

Metrics is anything measurable. This refers to **measures or ratios**, which occur in all fields such as music, physics, mathematics, etc.


Definition

Softwaremetric
A software metric is a measure that says something about the software.


Definition

"A software metric is a function that maps a software unit into a numerical value. This calculated value can be interpreted as the degree of fulfillment of a quality characteristic of the software unit."
(IEEE Standard 1061,1992)


Beispiel

A simple metric would be, for example, the number of code lines or the number of classes in an OO project. But of course, the topic of metrics goes much further - right up to architectural metrics that tell us something about the state of the software system's architecture.

Since a good software architecture means maintainability, changeability and testability, this topic is also referred to as "software stability". It describes how stable the software system reacts to changes, i. e. to its further development.

Basically, there are many types of metrics related to software development. A software project consists of many elements, for example:

- Sourcecode
- Binary Code
- external Libs
- Buildinformation
- Versioning Information
- Forms of meta- and additional Information (e.g. Annotations, JavaDoc, etc.)

This also means that many metrics can be measured. In this session we will look at some of them, such as:

- Complexity metrics
- Test metrics
- Architectural metrics
- Coupling metrics
- Style metrics

In computer science there are more types of metrics than just software or code metrics. Many metrics are particularly well known from project management, such as the function-point metrics or COCOMO for effort estimation.

We will first concentrate on the simple software metrics and then get to know the architecture metrics.

Hinweis

But: Good metrics are not synonymous with good quality. Good quality does not necessarily mean good metrics!

Experience from many projects also shows that the metric size and quality characteristics correlate quite often with each other!

As today's large projects are being developed and quality has to be checked, the visualization of quality metrics is becoming more and more important. This will be dealt with towards the end of the module.

Vertiefung

Interview with Jonathan Aldrich on the topic of Static Analysis (45 min):

http://www.se-radio.net

3.4 Einfache Metriken

Title: Simple Metrics

It is often helpful to have an overview of very simple code metrics. These include the following metrics:

Lines of Code

The number of lines of code - also known as LOC - is a known measure when it comes to comparing project sizes. For example, checkstyle has 22,369 lines of code and Apache Tomcat already has 159,364 lines of code, which makes it one of the larger projects. It is important to know whether all lines of code are included. If all file lines are included, Tomcat will have 314.461 lines, which can be due to documentation or configuration files.

Code lines can have completely different expressions in different languages. This is the expressiveness of a language. 1 is often referred to as C, modern dynamic languages often have an expression of 7 and more. This means that coding can possibly be done on average seven times as efficiently.

For example, compare the code line in Clojure with a possible implementation in an old basic dialect:

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

Hinweis

The function returns all odd squares between 0 and 100. For code lines, the expressiveness of the language should always be taken into account.

Comments

Number of lines describing the program. As a rule of thumb, this number should be between 30% and 60% of the code lines.

Methods/ Functions

The number of methods or functions of a program or project.

Classes

The number of classes used is of course also a measure of complexity. Around the year 2000, Germany's largest Java software project (Cheops) had about 30,000 classes, which is quite a lot.

Packages

The number of packages or namespaces are simply calculated here.

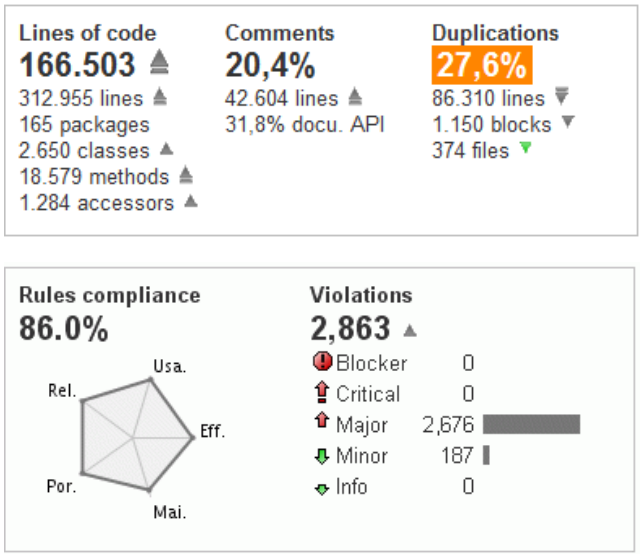
Files

Files are self-explanatory, but unfortunately they do not provide a good measure for the complexity of a software. For example, there are Ruby programs that are packed in a file but are still very complex (e. g. Kirbybase).

Duplications

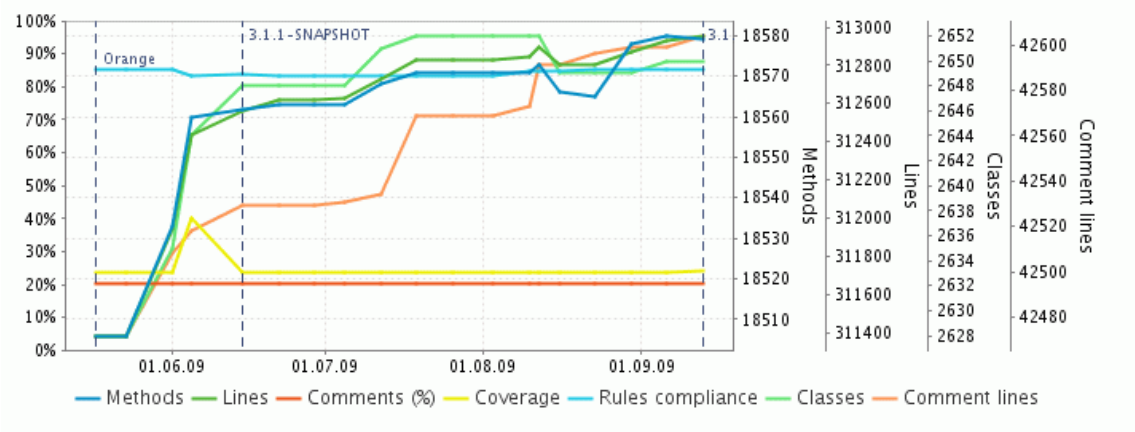
Some good tools search the code for similar building blocks. This can be done on different levels that search for identical lines of code, similar blocks or even similar files. In addition, this provides good information as to whether refactoring could be carried out or not. The DRY principle (Don't Repeat Yourself) was often violated.

For example, the same blocks can be combined in one method, which usually increases the maintainability of the code considerably. Here is an example of an automatic analysis of basic metrics:



Analysis of Basic Metrics

It is helpful if all metrics in the history can be tracked. With good tools, this is easily possible and you get graphically prepared output like the following figure.



Basis Timeline

Some tools also offer a graphical overview in which the metrics are displayed as weighted areas. With a click on the surface (similar to the tools for hard disk usage) you can "drill" yourself into the region or the error.

This list will never be complete. In the code there will be more and more characteristics that can be examined or transformed into metrics (number of to-do marks, annotations, etc.).

3.4.1 McCabe Metrik

Title: McCabe Metric

One of the best-known metrics is the McCabe metric named after Thomas J. McCabe. This is a pure code metrics that is relatively easy to determine. The idea is to automatically determine how complex the code of a software unit (e. g. method or class) is, and this can be done over 10,000 classes.

The basic assumption is that complexity can be calculated by the number of potential pathways that can be taken in a program. In this case, this means how many control flows are possible in the program graph.

Definition

McCabe Cyclomatic Complexity

Every conditional construct (whether it's a loop or an if/else/switch) increases the complexity by 1 (therefore it should be called Conditional Complexity).

The equation is:

Formel

$$M = E - N + P$$

Legende:

- M = the Cyclomatic Complexity,
- E = the number of Edges in the graph,
- N = the number of nodes in the graph and
- P = the number of exit points (return, last command, exit, etc.)

As a rule, exit point P is reconnected to the entry point and counted as an edge.

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

Aufgabe

Cyclomatic Complexity How high is the Cyclomatic Complexity in the example above?

Draw the graph!

Try to solve the problem yourself before looking at the sample solution.

Time needed: 10 min

Solution

$M = 9 - 7 + 1 = 3$

Usually, code with a higher McCabe complexity also has a lower cohaesion (see following chapters).

3.5 Halstead Metrik

Title: Halstread Metric

One of the lesser-known metrics is the Halstead metric. Nevertheless, it is implemented in many metric tools because it is easy to calculate by machine. It is also a static process in which the code is not executed, but the source code is analyzed.

This metric can be calculated as follows:

- $\mathbf{n_1}$ => the number of different' *operators* used in the code. So these are orders like that:

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

and break, case, return, try, but also private, friend, static. But no types.

- $\mathbf{n_2}$ => the number of different' *operands* used in the code.

So type declarations (int, bool, void) and all constants, type names and identifiers that are not reserved names.

- $\mathbf{N_1}$ => the total number of operators in the code.
- $\mathbf{N_2}$ => the total number of operands in the code.

From these four values, the following figures can then be calculated:

Program length:	$N = N_1 + N_2$
Amount of the vocabular:	$n = n_1 + n_2$
Halstaed-Volume:	$V = N * \log_2 n$
Halstaed-Lengh:	$L = n_1 * \log_2 n_1 + n_2 * \log_2 n_2$
Difficulty:	$D = (n_1 / 2)* (N_2/n_2)$
expense:	$E = D * V$

measure figures

The sources even list an estimate for the number of expected bugs.

This metric provides only lexical dimensional data. The actual complexity is therefore only estimated. In particular, dynamic programming languages that introduce or reduce extreme complexity with constructs such as currying, fibers, yield and closures are certainly avoiding the attempt to measure complexity accurately.

However, the absolute number is not necessarily relevant. Whether one module has a D of 65.25 and another module has a D of 67.78 is not so important. It is more important to get a feeling for these numbers and to be able to correctly evaluate the exceeding of orders of magnitude as an alarm signal.

Vertiefung

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

Artikel: Komplexität und Qualität von Software von XAVIER-NOËL CULLMANN und KLAUS LAMBERTZ verifysoft.com mscoder.pdf (737 KB)

3.6 Code Coverage / Test Coverage

There are two different areas of coverage:

1. The cover where the code will pass through during the expiration.
2. The test coverage: which controls/checks how well the tests, for example, to record the methods used.

If coverage is spoken of, software engineers usually refer to test coverage. Nevertheless, one should specify exactly what is meant.

Code Coverage
How often a method or piece of code is run through is also investigated by means of a metric. This can be very useful, for example, to check processes or to debug.


Beispiel

Example from the documentation of the programming language D
Here is an example from the documentation of the programming language D, which has built such a feature directly into the compiler. After executing the program, defined code points are output with the specification of how often the line was run through:

```
|/* Eratosthenes Sieve prime number calculation. */
|
|import std.stdio;
|
|bit flags[8191];
|
|int main()
5|{ int i, prime, k, count, iter;
|
1| 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
```

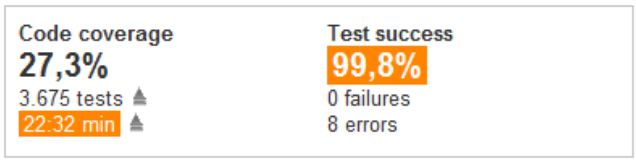
(Quelle: http://www.digitalmars.com/d/2.0/code_coverage.html)

Test-Coverage

Test Coverage has already been addressed in the TST Testing session. It is an important metric that often refers to unit tests - but of course tests can also refer to other units:

- Code blocks - have ifs, switches or loops covered or tested code blocks?
- Have larger units such as classes or packages been sufficiently tested? If so, what percentage?

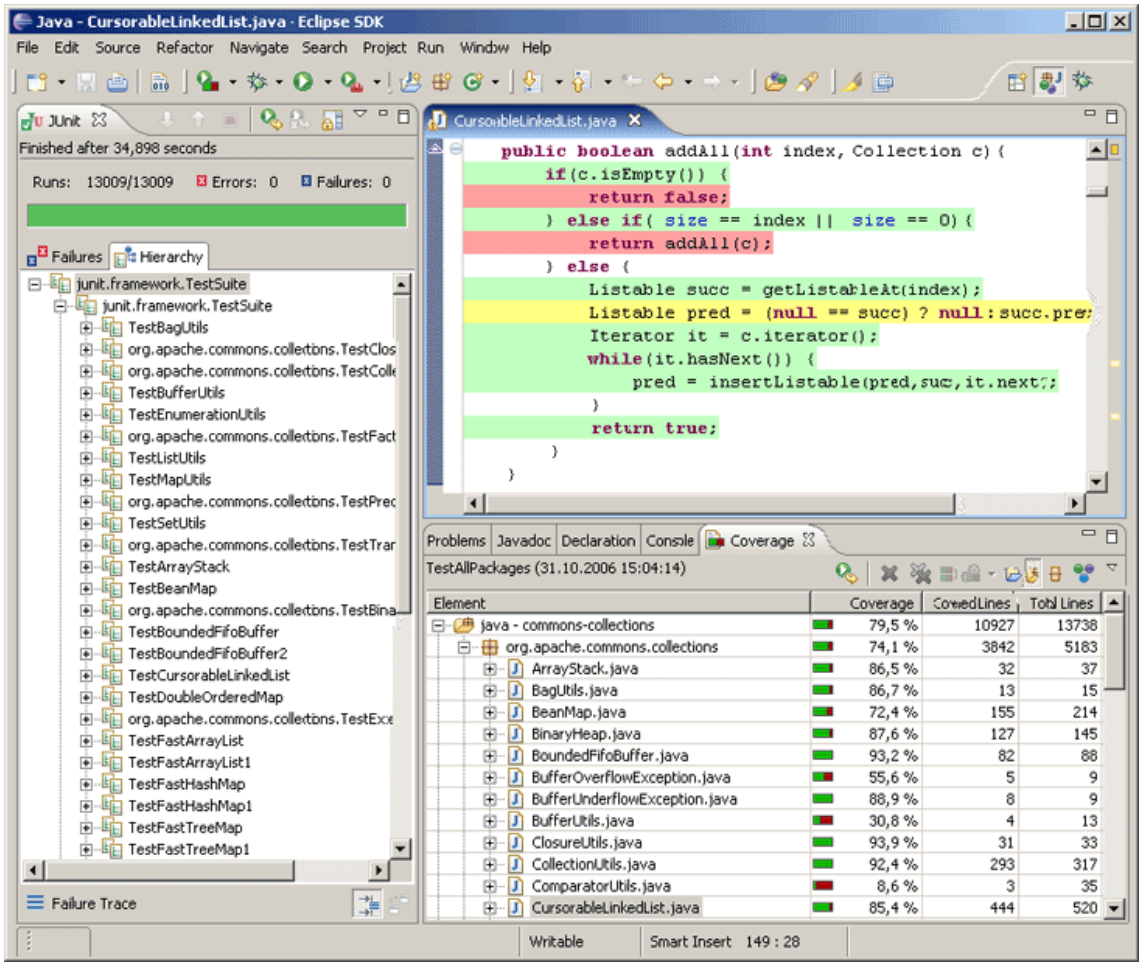
Such tools can be ideally built into a build management tool (Ant, Maven) and then provide meaningful reports:



Code-Coverage Report

While working on the code, such a metric can be very helpful. Therefore, there are plug-ins in every known development environment that provide these results.

For example, the inclined student is advised to install Emma or the eclemma (http://www.eclemma.org) plug-in. This can be done in a few minutes and you will get the first readings as shown in the picture:



Coverage-Metrik in Eclipse

3.7 Regelverletzungen

Title: Rule Violations

For many years, there have been excellent tools for checking code and programming style. Violations of the rules are usually displayed as aggregated metrics. The developer can list them individually and then edit the problems afterwards.

There are many types of rule violations, such as violating the framework of a style guide or violating rules that the community or you have defined as valid.

For Java are the most popular tools:

- Checkstyle
- PMD and
- Findbugs

An overview of other tools can be found on the Internet at:

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

Here are some examples of what can usually be checked by the tools:

Documentation in "JavaDoc":

- Is there a package.html?
- Is the JavaDoc Code well formatted?
- Are methods / Variabls well commented?

Java **Naming Coneventions**:

- Do they conform to the naming conventions regexp "^[a-z]+(\.[a-z][a-z0-9]*)*$"?

Is the "header" and "imports" okay?

- Avoidance of imports with asterisks ("de. vfh. swt. *")
- No specification of wrong imports
- No redundant imports
- No unused imports

Specification of "size restrictions":

- The number of executable code pieces can be limited (e. g. the number of methods)
- File size and number of lines can be limited
- The length of the method can be limited. Note: This test forces the refactoring Extract Method.
- A limit for the number of parameters can be set.

Check for WhiteSpaces or tabs,

Check the **order** of the **Modifier** in classes: For example, for Java the following applies

1. public
2. protected
3. private
4. abstract
5. static
6. final
7. transient
8. volatile
9. synchronized
10. native
11. strictfp

There must be no "empty blocks".

Many other "coding standards" are checked:

- Avoidance of empty statements.
- Local variables that should not change their value should be final.

Identical naming within blocks (shadowing) must be avoided.

- Don't use Magic-Numbers. Avoid the use of superfluous and therefore ineffective numbers (except -1,0,2) if possible, and use primarily declarations in final variables that can be supplemented by explanations.
- Missing default in switch construction
- Control variables should not be modified

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

- Unnecessary throws-declarations
- Find unnecessary code:

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

- Correct name definition for JUnit 3 tests.
- Are there equal strings in the code?

A good class design can also be checked:

- Visibility of class variables
- Use final
- Is the class geared to inheritance?

It is not only possible to set any regular expressions for the analysis of a line. Any number of rules can be implemented and extended.

Beispiel

" *Checkstyle example for inefficient code*'
As you can see here, today's tools also find inefficient code. Here is another example from Checkstyle:

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

This example should be replaced with:

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

Some tools such as Checkstyle contain the following metrics:

1. Number of **boolsch equations** as &&, ||, &, | und ^ in a command block
2. **ClassDataAbstractionCoupling**: Hint: This is the **Ce** from JDepend!
3. **ClassFanOutComplexity**: Hnt: This is the **Ca** from JDepend!
4. **CyclomaticComplexity**: Well we know this.
5. **NPathComplexity**: The number of possible processes. This corresponds to the McCabe metric.
6. **JavaNCSS**: Number of lines of code. With restrictions on method=50, Klasse=1500, FileMax=2000

Question: Which values do you think are useful for point 6 and the other metrics?

Résumé:

The power and usefulness of style analysis tools should not be underestimated for code and system quality. As an experienced developer, you're usually the first to be hit in the head when many thousands of error messages from these tools are displayed at the beginning. Often the wish to "comment out" some tests in the configuration file appears, which turns them off. It would be better to stop here and think about this carefully. Usually it is better to edit the cause of the error in the code rather than reducing the error check.

Tools such as checkstyles should always be included in the build initially or, if necessary, run as a plugin at the touch of a button. This can be difficult at the beginning, but in the long run this approach pays off, especially with large projects. Particularly because style analysis tools also recognize things that promote good architecture.


Beispiel

Practical Example for Checkstyle
The following shows the configuration and results of Checkstyle. At the beginning, you should consider whether this should be done under ANT (or a build management system) or as a plug-in in your preferred IDE.

- Download of **checkstyle-all-*.jar**
- Declaration of the task:

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

- Definition of the 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>
```

- Including the analysis definition. E. g. sun_checks. xml (possibly take out the things that don't work...). The file has approximately the following content:

```
<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"/>
...
```

As you can see here, concrete tests are defined for test terms. All tests are usually well documented and well-founded (http://checkstyle.sourceforge.net/checks.html).

Analysis of the result file checkstyle_errors. xml or the output on the console.

The XML file then looks something like this:

```
<?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>
 ... and more files...
 <error line="44" column="35" severity="error"
 message="&apos;71776119061217280L&apos; should be defined by
 a constant."
  source="com.puppycrawl.tools.checkstyle.checks.coding.MagicNumberCheck"/
>

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

 <error line="10" column="29" severity="error"
  message="The parameter arenaCmd should be declared as
 &apos;final&apos;."
   source="com.puppycrawl.tools.checkstyle.checks.FinalParametersCheck"/
>
```

```
            <error line="33" column="9" severity="error"
            message="The Method &apos;identify&apos; has not been designed
            for inheritance - must be abstract, final or empty."
source="com.puppycrawl.tools.checkstyle.checks.design.DesignForExtensionCheck"/
            >
```

Vertiefung

Sun Code Conventions Checkstyle PDM Findbugs

3.8 JDepend

One of the oldest and most famous metrics was developed in the 90s by Mike Clark. He provided a tool for code analysis, which can be found in many other frameworks, IDEs and tools in many languages.

The word Depend is already contained in the term JDepend, which is translated as Dependency. It is therefore primarily a question of analysing the dependency of the components of a mostly larger software system.

According to Mike Clark, JDepend's task is to do the following:

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."

Again, the source code has to be parsed and metrics have to be extracted from each class, which are presented below. These metrics allow you to measure and assess the extensibility, reusability and maintainability of the code. Better said, the metrics may indicate that the code may have is not easily expandable, reusable or maintainable.

In the field of research of software technology, the term "software stability" is also used in this context, i. e. how stable the software as a whole is.

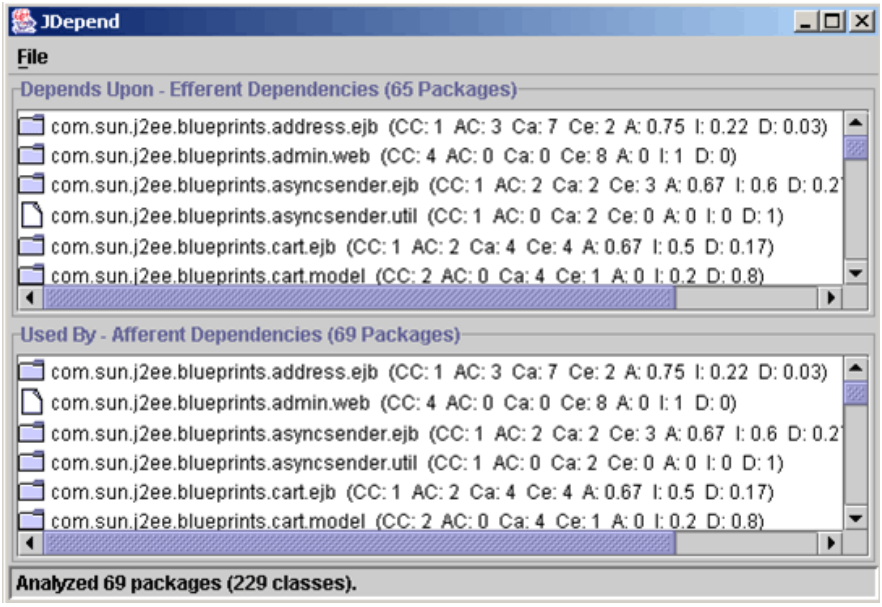
What are the objectives of the JDepend metrics?

1. JDepend promotes Design by Contract. It motivates you to develop stable packages - in the form of interfaces or abstract classes. The goal is that the unstable (frequently changing) classes are dependent on stable classes and not on other concrete

implementations. In this case, different people can also program in parallel / simultaneously against stable interfaces.

2. The independence of the packages is thereby promoted. This applies to both your own packages and external packages. The dependencies to external packages (e. g. commons. jar) can be made visible and, if necessary, isolated. This means that the external functionality can be encapsulated with stable components (e. g. interfaces or abstract classes for facades on the library). Independent, internal packages can also be developed much better in parallel - possibly even in a separate release cycle.
3. JDepend makes cycles visible. This will be discussed in more detail later. Cycles are components that depend on each other cyclically. These can be classes or packages, for example. For the changeability and maintainability of code, practical experience shows that cycles are "deadly".

Let's have a look at the metrics generated by JDepend:



Ausgabe der JDepend-Metriken

As you can see, JDepend initially refers to packages, that is, package dependency analysis. However, the view can be enlarged and reduced in size, i. e. you can also analyze at class level. As a rule, however, large projects are usually carried out at package level.

3.8.1 JDepend-Metriken MET

Title: JDepend Mectrics MET

Let's look at some of the metrics of JDepend.

The total number of concrete classes, abstract classes and interfaces:
This number alone is an indicator of whether the package is already too large, for example.

Dependency/Afferent Couplings (Ca):. Afferent couplings, which can be translated as "bringing", provide information about how many other packages depend on the package currently being considered. To put it bluntly: how many other packages will it "bang", if I change something in this package. Of course, the aim is to keep this figure as low as possible. J. Clark even writes here about the responsibility of this package for its services to others.

Efferent Couplings (Ce): Leading out / leading away "(e. g. from the biology of nerve cells). Here you can specify how many other packages my classes need from the packages considered. To put it bluntly: how sensitive I am when other classes change. How strong will it bang when other classes change? This seems to be a measure of the package's independence.

Abstractness (A): This number reflects a ratio and the values are between zero and one (including). A is defined as AC/ (CC+AC). This means: What is the ratio of abstract classes to all classes? Therefore, a package with metric 1 is a completely abstract package, which only defines but does nothing really. If the metric is zero (which unfortunately is too often the case), there are only concrete classes. In connection with the instability applied to axes, the points of the packages would then "stick"to the axes.

Instability (I): The instability is a little more interesting. It is calculated according to the formula I = Ce / (Ce + Ca). But what does this mean? The denominator is the total number of all dependencies aka Ce + Ca. So both how sensitive other packages are to changes in me and how sensitive I am when other packages change (where I am the package under consideration). Here a 0 would be a stable package and a 1 would be a completely unstable package.

Distance from the Main Sequence (D): This means distance to the main line. The main line is an idealized line with A+I=1 (see also pictures on the next page). This value is attributed to the property of describing how much the package is balanced between abstractness A and stability. Optimally balanced would be a package which lies exactly on this line, i. e. the distance d is minimized. The completely abstract package A=1 is also completely stable (I=0). On the other hand, a package that is only concretely very unstable, because it does not depend on interfaces or abstract classes.

Package Dependency Cycles: The output of cycles is also very interesting and important. If packages contain many cycles, class changes cause chain reactions to further changes. The developers are then only debugging and are not making any progress as regards content. An often observed phenomenon in many companies.

There are many cases of cycles:

- Cycles can be between classes. As a rule, classes of different packages are considered here. However, cycles between classes within a package are also dangerous. For example, a class X from package A requires a class Y from package B. This is a "direct cycle".
- Cycles can also occur between**'' more than two**' different packages.
  Z. B. A -> B -> C -> A. This means that all three classes are involved in one cycle.
- Finally, there are indirect cycles. If there is still a class P in the above example that requires package A, it is also indirectly dependent on this cycle and is marked accordingly by JDepend / Analysis tools.

3.8.2 Zyklenanalyse als Teil des Buildzyklus

Title: Cycle analysis as part of the build cycle

Tools that determine such metrics should of course be included in the build cycle to be available to developers, project managers or quality assurance within the framework of the continuous integration process. This means that the entire JDepend process (or other tools that determine the same or similar metrics) can be started and, for example, a *. txt report can be generated for shipping. Calling JDepend from build management tools such as ANT is therefore relatively simple:

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

```
</target>
```

Source: http://clarkware.com/software/JDepend.html

Beispiel

Some other tools allow you to insert the cycle search as JUnit tests. Here is an example:

```
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);
 }
}
```

Source: http://clarkware.com/software/JDepend.html

In the example, the sources are read in the upper area, then JDepend is called completely and the system checks whether cycles are included. These can then be output completely using data structure p. On the JDepend website there are even more examples of how to test for d, for example (you will find an explanation later on). There are tools available on the market for many languages that do this (such as Macker for Java).

The following figure shows how instability has been measured many years ago in community tools using javaforge.com.

SCM Commits | Browse Repository | Browse Commits | SCM Settings

Yesterday & Today	This Week	Last 30 Days	All
3	3	3	3

Source Code Summary

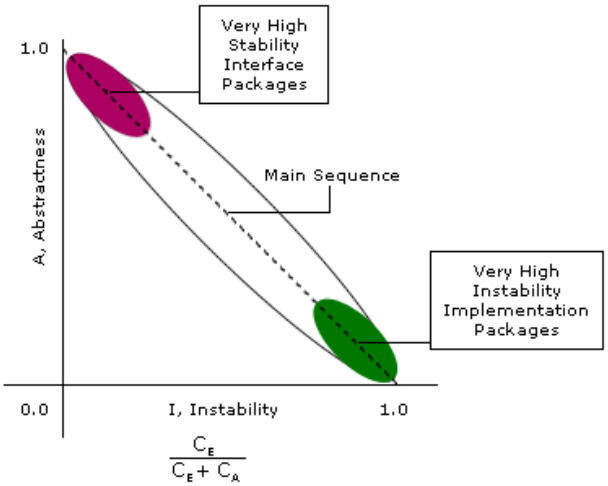
Category	Files	Lines	Bytes
Summary:	0	0	0

Source Metrics Summary

Action	Description	Value
Trends | Details	Package instability ratio	0%
Trends | Details	Percentage of duplicate code files	0%
Trends | Details	Total number of duplicate lines	
Trends | Details	Total number of violations	
Trends	Total number of copy paste blocks	
Trends | Details	Total number of packages	
Trends | Details	Number of checkstyle errors	

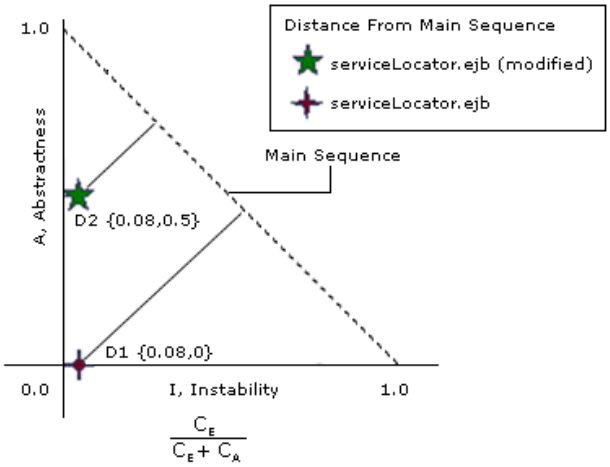
Measurement of the Instability

Analysis of the graphics



Main Sequence Graphics

In the first figure you can see the Main Sequence, which traces the line **A+I**. This shows that packages cannot be completely abstract (' *A=1*) and completely stable (*I=1*) at the same time. There should always be a balanced amount of concrete classes on the one hand and abstract classes / interfaces on the other.



Distance from Main Sequence

(Star: Class A) de.vfh.swt.metriken.cpool; Cross: B) Clas sde.vfh.swt.metriken.media)

In the second picture we see a package A, which is quite far left upside. From this it can be concluded that there are obviously many abstract classes and a high stability (I is small).

The second point has few abstract and too many concrete classes. There is a high instability here (I is large). From all points a d, i. e. the distance to the "Main Sequence", can now be calculated. Afterwards, this should be considered individually and summed up if necessary.

Packages with low abstraction should depend on packets with high abstraction!

In an ideal world, there would only be implementation classes that depend on abstract classes. In the real world, however, you have to make compromises...

JDepend measures the distance of the package to the "Main Sequence" D.

Hinweis

> The greater the distance D between the package and the main sequence, the greater the probability that the package can tolerate a review or refactoring.

These are, of course, only indications that need to be examined in concrete terms and, in special cases, can be devoid of any basis. Nevertheless, experience shows that packages with a large D can certainly tolerate refactoring. Similar to the indication of good test coverage.

3.9 Kopplungsmetriken

Title: Coupling Metrics

Another very interesting metric area is the coupling. This refers to the degree to which components are linked with each other.

There are two different areas:

Cohesion: translates approximately context / binding power

Coupling: is about coupling / connection. Often referred to as the dependency.

Cohesion examines, for example, the methods of a class / component and how they relate to each other. The goal is that one class / component performs a task and not several. If this is the case, there is a high degree of cohesion. These classes are more maintainable, changeable and of course easier to understand. On the other hand, there is usually a small coupling on the other hand.

The opposite example would be a class with low cohesion. Here, the class and its methods would fulfil many tasks. Most of the time there is a large coupling, since more external components need this class. The present component is therefore much more difficult to change and understand. Refactoring should redesign the class so that it only fulfils one task.

A core element of cohesion is whether the component can be meaningfully cut into independent parts and whether the set of both parts provides the same service as before.

Further material is available on this topic here:
http://www.aivosto.com/project/help/pm-oo-cohesion.html

3.9.1 Coupling / Kopplung

Title: Coupling Example

There are different types of measurement with the coupling, because you can look at different coupling variants and these can be weighted differently. For example, the communication of the modules such as data exchange, parameter transfer, the use of further data, etc. can be considered.

Here is one example.

Pressmann *Pr09* gave the idea to do the following:

- d_i: Amount of Input Paramters Data
- c_i: Amount of Input Parameters Control Structures

- d_o: Amount of Output Parameter Data
- c_o: Amount of Output Parameter Control Structures
- g_d: Amount of global used Data
- g_c: Amount of global used Control Structures
- w: Amount of called Modules (this is the Ce from our JDepend)
- r: Amount of Modules, that call the current module (this is the Ca from our JDepend)

This metric also considers control data, which means that when a module transmits control information to another module, this is weighted more strongly than when only simple processing data is considered.

Pressmann defined coupling like this:

Formel

$$C = 1- 1/(d_i + 2*c_i + d_o + 2*c_o + g_d + 2*g_c + w + r)$$

A side-effect free method would be a method with an input parameter (d_i=1) and an output parameter (d_o=1) such as the calculation of an interest value. Since this is called by a module, (r=1) the denominator must be three in the above formula. As a result, the result is usually a minimum C of 0.67. Of course, this value does not exceed one.

Beispiel

Let's take another example:

There are three input parameters and one output parameter, a control parameter and no global variables are used. The module uses two other modules and is called at four other places in the code.

It follows:

$C = 1 – 1 / (3 + 2*1 + 1 + 2*0 + 0 + 0 + 2 + 4) = 0{,}917$

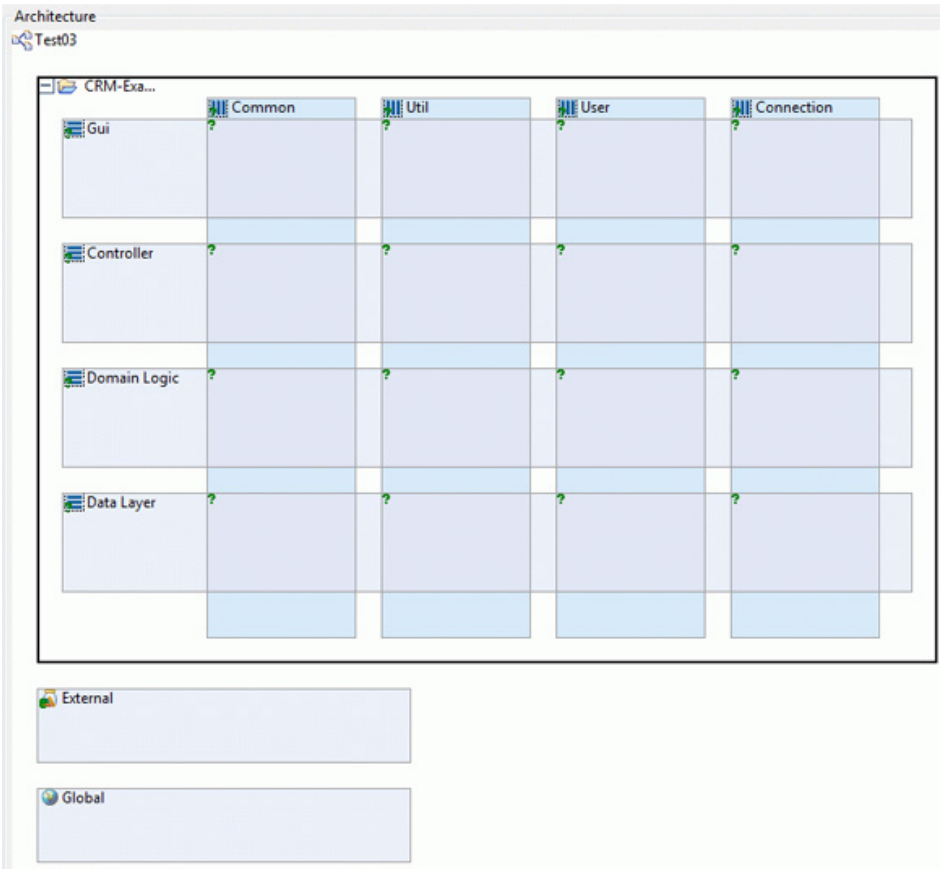
Surely, one can argue about whether the given weighting is reasonable. Other metrics might weight Ca and Ce more heavily than simple input parameters (data).

3.9.2 Die ACD Metrik

Title: The ACD Metric

Another - perhaps even more accurate - metric is measured by SonarJ and comes from John Lakos[*Lak96*]. We will dig a little deeper here.

An architectural approach has already been mentioned in which components were entered into a grid and logically grouped. For example, an axis can refer to the layer model: show the view at the top and the data retention layer at the bottom. Additional layers or logical domain objects can be placed on another axis. Let's take a look at the following figure.



Definition of an architekture

For architects, it is important to know how the "call flow" is determined at this point. Can each component call up any other component according to the "spaghetti style"? Or is a top-down with left-to-right permitting much more meaningful? Of course, the latter makes more sense, which is why there are many tools - such as SonarJ or Macker - that test this kind of thing.

If you take out a component and look at which other components are called up, you can create a graph. This graph can refer to packages in large systems as well as classes in small systems.

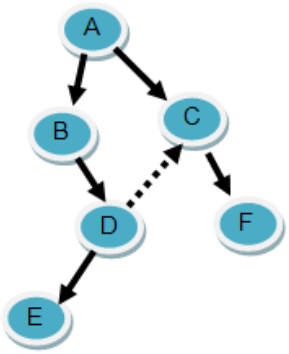
Hinweis

As already explained in the session Design, the following shows that packages should not access each other unstructured. Access to Package B from Component A (or, for example, Package en.vfh.swt.A) is better done via one or a few interfaces (quasi Facade Interfaces), which provide the functionality bundled together.

The recorded graph is itself already a measure of the degree of coupling. A large graph with many cycles is logically much harder to maintain and change, since many more code points are affected. Therefore, a small and cycle-free graph is always useful.

Beispiel

Calculation of the ACD metric How is the ACD metric calculated? Let's look at an example:



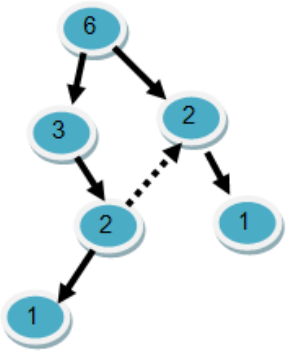
Component Graph

The figure shows some components from A to E, which can represent both classes and packages. A uses B and C. B uses D and indirectly also E. C uses F. In the first variant there are no cycles. In the second variant, D also uses C again, which creates a cycle.

ACD stands for Average Component Dependency and describes the mean component dependency according to Lakos. However, there is no distinction between control and data flow.

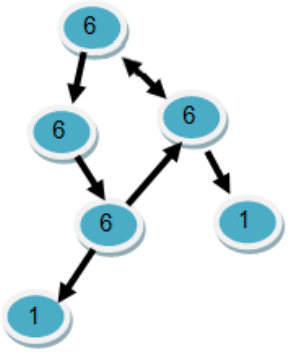
After the ACD method, each component is initially assigned an A since it implicitly depends on itself. Paint the above graph on paper and enter the numbers. Then you start from the bottom and add the sum of the underlying / dependent numbers in each component.

The following result is obtained for the first cycle-free variant:



The ACD metric is calculated by dividing the sum of all values by the number of nodes! In our example, **ACD = 15 / 6 = 2.5**. This means that on average, each node depends on itself (=1) and another 1.5 nodes.

If we now add a critical cycle from D to C and from C to A, the numbers change significantly. We obtain the following graph:



We can see here that in the cycle, each account actually depends on each other. The circle in question therefore receives a value of 6 on each node. If the new mean dependency is calculated, a completely different impression is created: "ACD = 26 / 6 = 4.33". This is a much higher value, which certainly indicates refactoring.

What happens when you compare two graphs? Let's take our graph above with a cycle and a value of 4.33. We want to compare it with a graph that looks similar, has 100 nodes but does not have a cycle.

In this case, the 100knotige graph might have a much higher ACD value, simply because of the absolute number of nodes. Nevertheless, this graph could be "cleaner" than the lower-order small cycle graph. To obtain comparable values, we have to divide by the number of nodes again. Only then do we get a relative value that is independent of the number of nodes. So we call it "rACD" and define it:

ACD = number of dependencies / number of nodes **rACD** = ACD / number of nodes

We obtain:

> Graph 1: rACD1 = 2,5 / 6 = 0,4167 Graph 2: rACD2 = 4,33 / 6 = 0,7267
>
> It shows that a maximum value of 0.5, i. e. 50%, can be reached for cycle-free graphs. However, a lower value would be better.

Übung MET-01

Cycle free graphs

Use examples to illustrate why it is impossible to achieve a value greater than 50% in cyclic-free graphs as the number of nodes grows!

Processing time: 15 minutes

Experience has shown that a rACD value for larger graphs is a very likely indication of cycles in the graphs considered.

The sonar team (after Zitzewitz, Java-Magazin 2.2007, p. 30) comes to the conclusion that further formulas can be derived from experience, which provide good values for rACDs - depending on the node size:

At 200 knots, the rACD should be less than 15%. For 1000 knots, the rACD should be less than 7.5%.

The following useful connection can be established:

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

3.9.3 Zusammenfassend

Title: Summary

The coupling metrics presented here automatically provide a good overview of the "interlocking" of components. Cycle-free components with the lowest possible rACD are helpful.

These metrics are quickly determined in modern tools and can be displayed in suitable colours. Developers and architects can immediately see which packages or components are critical so that refactoring can be carried out.

Ideally, such metrics should be used right from the start and sound the alarm with the tools, as this promotes the natural growth of a good architecture.

3.10 Technical Debt MET

Title: Technical Debt

Back in 1992 Ward Cunningham came up with the idea of describing the number of errors, rule violations and complexity as **Technical Debt**. Freely translated, this means that you are guilty because something can be improved on the technical side.

There are several reasons for this debt. For example:

- bad architecture / design
- many code metric violations
- too much complexity
- too large coupling
- cyclic dependencies

Some errors may be easier to fix. However, there is also an intrinsic complexity that is usually more difficult to resolve or reduce. Other problem areas are of a long-term nature (architecture) and some are of a short-term nature (missing JavaDoc), as this can be solved more easily.

The entire problem addresses a topic that is often found in industry. There is the pressure to get a product onto the market quickly. Therefore, a solution " *Quick-and-Dirty*" is implemented. The product is shipped with its guilt, including the dirty part that is in the code. A loan has been taken up in the same way as in Financial Accounting (debt), and now interest is to be paid. Interest rates are paid at a higher or lower level because the product - as a rule - has to be further developed. However, the higher the number of injuries, the harder it will be to develop and modify the program. For debugging, the developers now have to invest much more time.

The ideal goal for the software vendor is to deliver the product with a debt of zero. The buyer's expectations are to buy a product with the lowest possible technical debt.

This topic is often discussed in forums and other forums. It is also interesting that modern tools try to express the debt as monetary value. This means that all metrics are converted into man-days. How many man-days are needed to eliminate the number of code and architecture metrics errors? Man-days can be converted into monetary value - depending on the wage level.

Hinweis

The decisive factor is that software costs are not only made up of development costs, but that they also result in much higher maintenance costs, training costs, change costs, etc., which have to be taken into account!

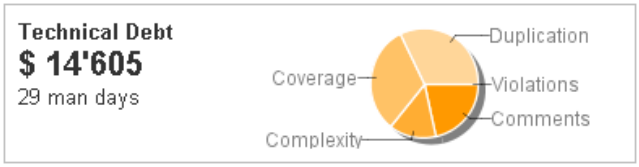
This is done with the Sonar tool as follows:

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

Where we have:

Duplications =	cost_to_fix_one_block * duplicated_blocks
Violations =	cost_to fix_one_violation * mandatory_violations
Comments =	cost_to_comment_one_API * public_undocumented_api
Coverage =	cost_to_cover_one_of_complexity * uncovered_complexity_by_tests (80% of coverage is the objective)
Complexity =	cost_to_split_a_method * (function_complexity_distribution >= 8) + cost_to_split_a_class * (class_complexity_distribution >= 60)

Sonar – Cost Calculation



Sonar - Calculation of development days

The number is intended to motivate those responsible or developers to invest in quality. This is particularly useful if the Technical Debt does not result in insignificant "interest payments" (expenses).

Of course, it is clear that this absolute number is very likely to have nothing to do with reality. There may even be cases where a high technical debt is not so bad. And here too it is like with the JDepend metrics (e. g. the d). It's not the absolute number that counts, but the experience gained with such graphics and numbers (which can be further detailed at any time with a simple click).

Such metrics and tools sensitize for the quality factor. This is usually just as important as being the first to be on the market with a product. Quality is one of the top priorities when it comes to dominating the market in the long term and asserting itself against the competition.

Vertiefung

Software Best Practices Blog Matrin Fowlers Comment Announcement bySonar Heise Artikel by Eberhard Wolff Cognitive Horizon

3.11 Werkzeuge

Title: Tools

The tools mentioned so far mostly offered pure code metrics. To improve the architecture, there are programs that integrate the low-level analysis of Checkstyle or PMD. On this page you will find examples of such tools for the Java language. For. NET and other languages like Ruby or Python there are similar tools.

The tools are divided into two license types: commercial and free programs. The free people are divided into three categories according to their abilities:

1. All-purpose tools,
2. Purely physical dependency analysis and
3. Dependency analysis against a logical structure.

The commercial tools listed here all have a graphical user interface.

- **SonarJ**: The most expensive and at the same time most powerful tool comes from the German company Hello2morrow (weblink www.hello2morrow.com). The architecture is divided horizontally and vertically to form subsystems in the levels. Java artifacts (classes and packages) are linked to it. For developers, there is an Eclipse plugin that displays architecture violations as a problem at development time. Class relationships can be virtually removed, so that the effects of refactorizations are immediately displayed. SonarJ works on class files. The source code is included to explain dependencies. An Ant Task generates HTML reports and stops the build in case of architecture violations. Some metrics are listed in the interface and in the reports.

- **Lattix LDM**: Lattix, an American company, manufactures LDM. This organizes the structure in matrices according to the DSM12 approach (http://en.wikipedia.org/wiki/Dependency_Structure_Matrix). Since no logical structure can be defined, LDM does not find any architectural violations. Consequently, automation is not possible. Packages can be used to set up rules that can be accessed. UML diagrams in the exchange format XMI should be able to be imported. In connection with MDA projects, the structure should be analysable. However, importing simple diagrams from MagicDraw in XMI format seems to be problematic.

- **Structure101**: The program was developed by the Irish company Headway Software. Like Lattix, Structur101 follows the DSM approach and also offers more visualizations of the packages. A logical model is based on the Java package hierarchy. Structure101 assumes that packages at the same level in the package hierarchy have the same logical meaning. Due to the strong reference of the logical to the physical structure, most views focus only on classes and packages. At each packet level, it identifies areas that are not related to other packages. It uses only a few metrics. You can use the command line to install it in a build system. An Eclipse plugin indicates architecture violations at the time of development.

- **STAN**: Dhe manufacturer Odysses Software[Ody] produces STAN (Structure Analysis for Java). It is available as an Eclipse plug-in and as a separate application. STAN does not allow the definition of logical structures, which is why all analyses are performed at packet level, to which various metrics are applied. Extensive reports with diagrams can be generated. With the trial version, only one test project can be reviewed.

Except for JDepend, none of the following free tools have a graphical user interface. They are designed to run with the build and generate a report, usually in HTML format. The free general-purpose tools examine architecture in different ways. Both subsequent tools include a variety of other free tools.

- **Sonar**: Sonar (http://sonar.codehaus.org/) is called "Code quality management platform". A tool for the analysis of dependencies is still missing, but it is on the request list. Many metrics are used, some of which are displayed graphically. A special feature is that each analysis result is persisted in order to display it later on timelines. Sonar runs as a stand-alone server and comes with a web application as a user interface. The project to be investigated has to be configured for Maven. The graphical presentation of Sonar is very informative and appealing. Sonar is the only platform that calculates the technical debt and displays it in the form of a monetary debt.

- **XRadar**: XRadar (http://xradar.sourceforge.net/) uses two tools, JDepend and DependencyFinder, to investigate dependencies. A logical structure can be defined in layers and their subsystems. XRadar generates extensive reports (spider diagram) and checks for architecture violations during the build with Maven or Ant. The logical structure and other diagrams are provided graphically. It offers various metrics for which limits can be configured. The configuration for a test project other than the one supplied is not easy, since error messages are lost in the long log. XRadar evaluates the reports from several reports to give a historical overview. This includes various diagrams about the development of metric results.

- **JarAnalyzer**: JarAnalyzer considers a Jar file as a component and analyzes the dependencies to other Jar files (http://www.kirkk.com/main/Main/JarAnalyzer). The result is a report and a dot file. The dot file can be converted into a graphic that displays jar files with their dependencies.

- **JDepend**: It analyzes the class files and creates a report with dependency metrics (http://clarkware.com/software/JDepend.html). All specifications refer to the package level, including the used and dependent packages. In addition, a diagram can be generated via the detour of a dot file. It displays each package and its relationships.

- **DependencyFinder**: It (http://depfind.sourceforge.net/) provides a set of investigative tools. Only a few are applied to metrics. A complete examination takes a relatively long time and results in an XML file of 100 MB. HTML reports are generated from it.

The free dependency tools of logical structure checking only deliver results in text form:

- **dependometer**: (http://dependometer.sourceforge.net) Eine logische Struktur kann in Schichten und funktionalen Abschnitten angelegt werden. Ein Lauf bricht bei maximaler Speicherzuweisung wiederholt ab und hat 4 GB Daten auf die Festplatte geschrieben.
- **Architecture Rules**: (http://dependometer.sourceforge.net) Die Konfiguration der logischen Struktur ist nicht gut dokumentiert. Folglich wird kein Ergebnisbericht erzeugt. In den Editionen der Architecture Rules fehlen brauchbare Details.
- **SA4J**: SA4J (Structural Analysis for Java) was developed by AlphaWorks / IBM (http://www.alphaworks.ibm.com /tech/sa4j), but development was discontinued in beta 2004. The installation of the program is complicated.
- **Macker**: (http://innig.net/macker) Macker only checks for violations during the build and creates an HTML report. A logical structure can be created in components. The configuration of package patterns is powerful, but not well documented for component structures. The report and a check against the logical structure run quickly. In the event of violations, the build system receives an error message with the dependencies that are not allowed.
- **japan**: (http://japan.sourceforge.net/) You can define the logical structure at package level without specifying patterns. Each dependency must be described individually. Besides Ant, IntelliJ is supported. There is no report. The expenditure is not recoverable.
- **Classycle**: (http://classycle.sourceforge.net/) An HTML report provides a detailed view of the dependencies. The logical structure can be defined in layers and components. The report and a check against the logical structure run quickly. In the event of violations, the build system receives an error message with the dependencies that are not allowed.

Without taking into account the costs, SonarJ would probably be the best tool. Sonar, XRadar, Classcycle and JarAnalyser are recommended for further testing.

The following two tables summarize the results. Thereby Kom. for commercial software and FOSS for Free Open Source Software. The line Try gives an impression of the effort required to run the tool successfully (++ = high success).

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	++	--	0	--	--	+	--
Vergleich logisch / physisch	++	--	+	--	--	++	--
Visualisierung der Abhängigkeit	++	+	+	+	--	0	+
Zyklen finden	++	+	--	0	--	+	0
Metriken	+	0	0	+	++	++	+
Berichte	0	+	0	+	++	+	+
Ergebnis im Quellcode	+	-	+	0	++	--	--
Ergebnis im Quellcode	++	--	0	+	0	++	++
Unterstützung OSGi	++	--	--	--	--	--	--
Kosten	--	-	-	-	++	++	++
Versuch	++	0	0	-	0	0	++

Software-Werkzeuge I

Kriterien/ Produkt	**JDepend**	**Dependency Finder**	**Architecture Rules**	**Macker**	**Sonar**	**Macker**
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	--	--	0	0	-	+
Vergleich logisch / physisch	--	--	--	+	-	+
Visualisierung der Abhängigkeit	0	--	--	--	--	--
Zyklen finden	+	0	--	--	--	+
Metriken	+	0	--	--	--	-
Berichte	+	-	--	-	--	+
Ergebnis im Quellcode	--	--	--	--	--	--
Ergebnis im Quellcode	++	+	+	++	++	++
Unterstützung OSGi	--	--	--	--	--	--
Kosten	++	++	++	++	++	++
Versuch	+	0	-	-	-	++

Software-Werkzeuge II

3.12 Zusammenfassung MET

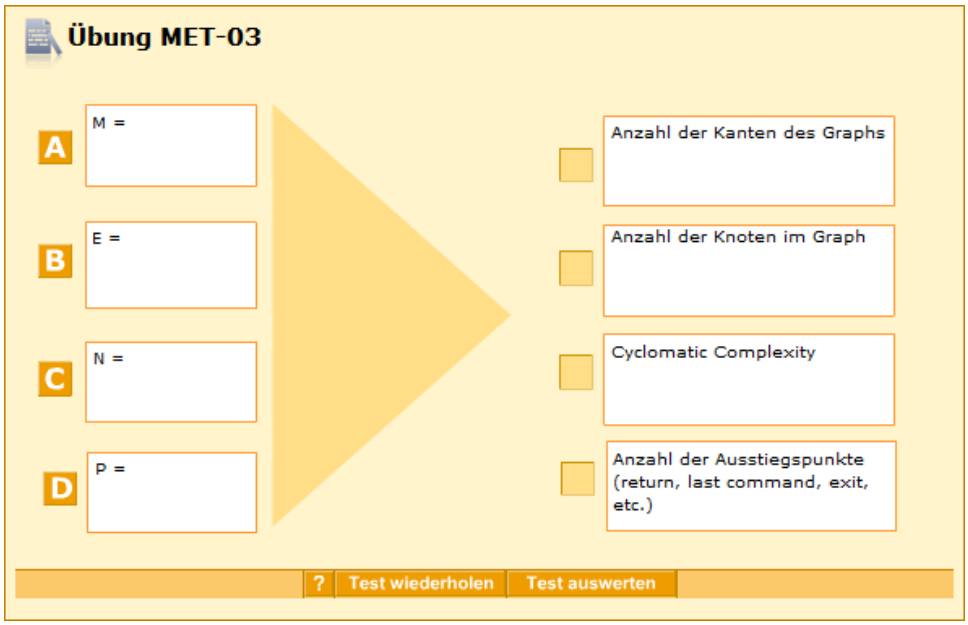
Title: Summary MET

- Metrics are an important tool for quality assurance. Most interesting is not the absolute metrics, but the reference to a problem and the relations of the metrics to each other.
- Even simple metrics like McCabe or Halstaed can be used to calculate interesting complexity metrics in relation to the code.
- Metrics for rule violations in the code are another important area. There are a variety of tools that can look over your shoulder.
- When it comes to cycles, abstract programming and package dependency analysis, JDepend is one of the most important basic tools. Many of the principles have been integrated into other tools.
- In the architecture analysis, there are tools that allow you to check accesses and thus force a better design (e. g. only top-down accesses).
- With other metrics such as the rACD measure, intrinsic module complexity can be measured. These metrics can provide architects with valuable clues to design decisions that are sometimes disadvantageous.
- With the Technical Debt, new interesting metrics are now available to measure the entirety of all violations - in the area of code metrics and architecture metrics - and state them as expenditure debt (in $ or €).

3.13 Wissensüberprüfung MET

Übung MET-02

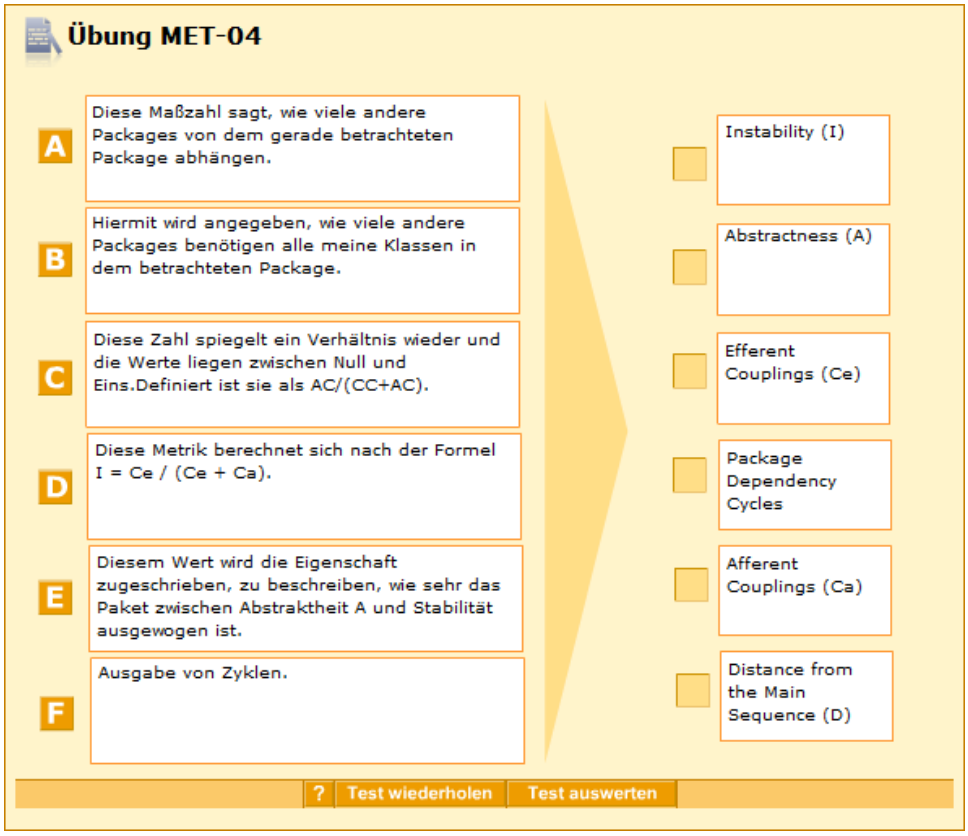
Anfang Druckversion



Ende Druckversion

Übung MET-03

Anfang Druckversion



Ende Druckversion