4 CCD - Clean Code Development

Autor: Prof. Dr. Stefan Edlich

4.1 Überblick und Lernziele CCD

The following learning unit is to be understood as a kind of checklist for the responsible handling of code and the design of good code / good software.

The community now has a global understanding of best practices, rules, bad-smell and coding guidelines that ensure good, modifiable, sustainable and readable code.

This knowledge is not coherent, but exists selectively. For example, there are overlaps in class design, but most of the rules mentioned here are relatively independent of each other.

The student's goal is therefore not only to understand them, but also to create his or her own kind of "cheat-sheet" and to follow the guidelines in his or her practical work.

Lernziele

Learning Objectives In this learning unit, the main point is not to code blindly, but to observe oneself on a kind of meta-level. Ideally, the person should split into two persons - as in the XP procedure model: A) A person who is coded and pays more attention to the current syntax and the specific algorithm, and (B) a person who observes as a meta instance and gives clues.

In this module all the skills are taught to be a perfect B-person or to check yourself as a B-person. In addition, this B-person learns all the principles, practices and "smells" to be able to propose and implement competent improvements at all levels. From the pure code level to architecture and management.

Gliederung

Gliederung This learning unit is divided into the following parts:

- Introduction
- Coding Fundamentals
- Coding Basics
- Code Quality
- Architecture / Class-Design
- Packages
- Productivity and
- Management

Zeitumfang

Time Requirements
It takes about 60 minutes to work through the learning unit and about 130 minutes to complete the exercises.

4.2 Einleitung zu Clean Code Development

Title: Introduction to Clean Code Development

Despite the development of better MDA tools (Model Driven Architecture) and programming languages on an ever higher level, it will still be hand coded in many years to come.

And that good code is important and that good quality features should be fulfilled, is also signed by everyone. Nevertheless, there are already many companies that had to file for bankruptcy due to bad code. Everyone knows the problem, but no one fights it at the root. Everyone of us has seen foreign code before and surely got to see a lot of bad code.

How can this all happen?

1. On the one hand, from the ignorance of the developers
2. Secondly, from the pressure of the market to offer "something" (a prototype?!) as quickly as possible.

In addition, the code complexity and thus the maintainability, readability, testability and modifiability of code increases with time and increasing bad code. Often, this can no longer be controlled. A significant number of companies around the world are involved in debugging and maintaining their software. Software maintenance is estimated to account for more than 80% of the cost of software. Managers, companies and often developers are not aware of the importance of a high technical debt.

Anmerkung

Technical Debts
This refers to dangers/problems/consequences resulting from badly written software.
More information:

1. #
2.

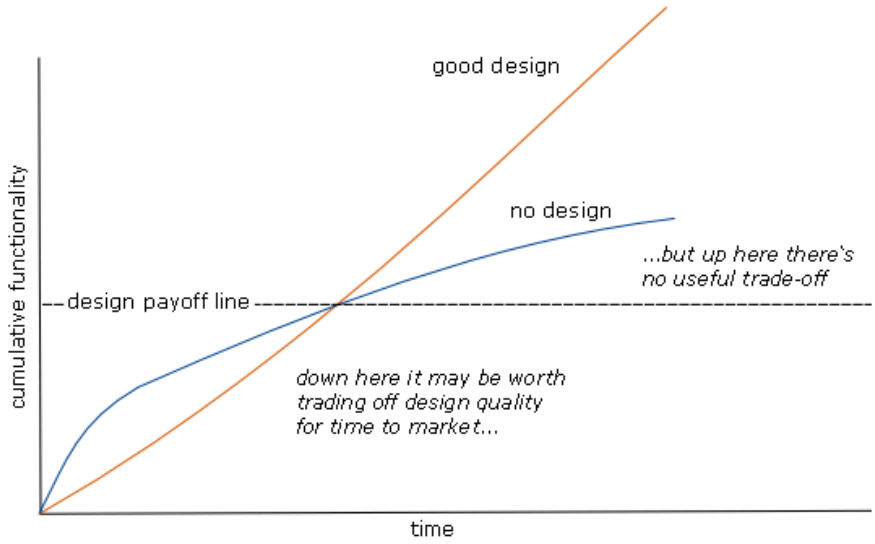
Managers often consciously make the decision:

' First develop quickly and earn money.
Repair and build in quality can be done later.'

In practice, however, it is all too often the case that there will be no future in which money and time will rain.

In the following graphic you can see a comparison of two projects which differ from each other by "good design" and "no design". At the beginning,"no design" with a steeper upward curve surpasses the functionalities of "good design". However, productivity is reduced over time due to poorer maintainability and modifiability. For example, this is about the effort to introduce new features. At the same time, the slope of the curve decreases and the time it takes to earn money from the product decreases. Therefore, there is usually a break-even point at which it is no longer worthwhile to enter Technical Debt. But where is this break-even point. After an in-depth discussion with experts such as Martin Fowler, it is concluded that this point comes much earlier than managers would suggest.

Please have a look at the folliwing graphics: (Quelle: Bliki):

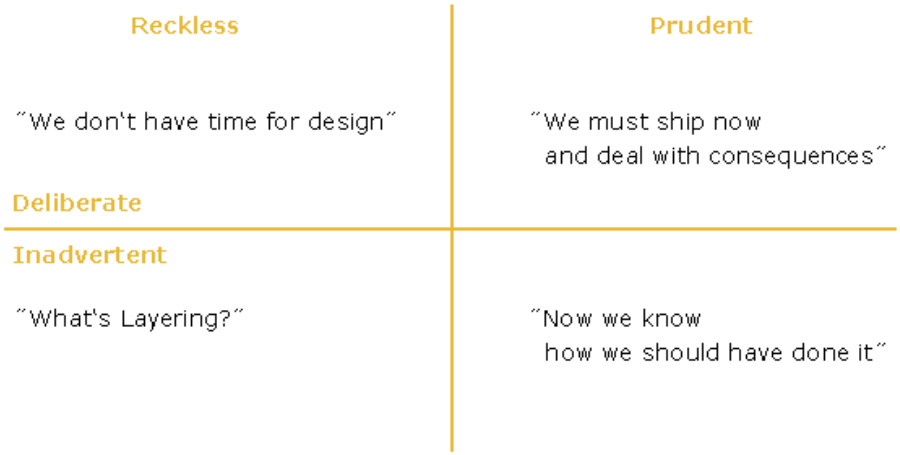


Comparison of two projects: 'no design' und 'good design'

The goal is to encourage developers to develop sustainable code whose quality is so good that technical debts are minimal. On the other hand, there must be a counterbalance to managers who prefer fast money and unsustainable value.

Of course, it is also legal to say that you can take the initial money with you and then let the company go bankrupt. Only then should you tell the developers from the beginning. However, since this is not an option for most companies, the question arises as to why and on what basis decisions are made against good design/good code and thus for implicit technical debts.

Here is a good overview by Martin Fowler (Quelle: Bliki):



Technical Debt Quadrant

The axes are labeled as follows:

- Reckless (rücksichtslos / unverantwortlich)
- Prudent (vernünftig / besonnen)
- Deliberate (absichtlich / bewusst)
- Inadvertent (unbeabsichtigt / ungewollt)

The graphic can be interpreted as follows when transferred to the Technical Debt and this learning unit:

1. **The Reckless / Deliberate** are the ones who want to take the fast money with them and deliberately and ruthlessly accept a code debt. However, it is deliberately not to be aware of a later failure, to be able to display or to take into account it. With "We don't have time for design" (or even good code or zero technical debt) it is expressed that the speed and hardness of the market does not permit any other action. So you have to act in this way and, once again, you are not actually responsible.

1. *'The Reckless / Inadvertent* are here rather the ignorant ones, who can't understand quality, code quality, design or technical debts. Mostly these are people from other disciplines who don't want to deal with them at all. What's Layering "not only expresses the question of what layer architectures (such as N-Tier or MVC) are, but also somehow a form of ignorance according to the motto:" I don't care anyway ".

1. **The Prudent / Deliberate** persons are well aware of the danger of bad code' somehow'. Nevertheless, the problem is - similar to the pyramid scheme - displaced and pushed backwards. Motto: Either it won't be so bad or we will get it under

control somehow. Therefore,"We must ship now and deal with the consequences" applies here. Delivering is therefore still the number one priority.

1. **The Prudent / Inadvertent**' people are the ones who have consciously, perhaps even from the very beginning, the quality in mind, who recognize errors and now at least hope to have time and money to address the problems. Nevertheless,"Now we know, how we should have done it" shows that despite good will, things have gone wrong.

So developers should do their job with the knowledge:

- I have to convince the managers that sustainability = low technical debt is an advantage. Fortunately, you can talk to most (IT) managers and often find a solution in the sense mentioned here.
- I know that 90% of the time code is read and not only written by me.
- I know that the cost of code maintenance and enhancement is more than 80% of the total software cost.

According to
Robert C. (Bob) Martin and Maartin Fowler
it is our duty to defend the / our code with passion!

Die nachfolgenden Auflistungen sind daher:

- **Principles / Practices**: Best practices rules that have proven themselves. Many of these rules are design patterns or good manners among classes / packages.
- **Smell**: This is about what happens if the rules are not followed. So a rather inverse wording, from which the rule is derived.

4.3 Fundamentals

Code is read 5-8 times more than written. And code maintenance is one of the major problems of the industry. Code that is still modern today will be' legacy' tomorrow. Even for the developer himself, speaking clear and logical code is a help. Everyone knows the effect of having to read their own code months later and having a hard time getting in. To code with a lot of technical debts you lose the "contact" much faster and to understand and expand / change it is also more difficult.

Another major problem of the Technical Debt is that managers often rely on the lock-in effect. Once a customer has bought a product, he usually "hangs" in it for 10 years. This is the argument that developers face when it comes to eliminating technical debts. Often

it only helps here if you argue with hard numbers. For example, now 2 developer weeks and then in 2 years no 2 developer months.

References:

- Reduce the Cognitive load

4.4 Coding Basics

Kill Useless Code

Kill Useless Code

Under developers it is sometimes common practice to leave code without a task in the source file. Usually this is also code found by Coverage Tools and cannot be found by any test.

It is assumed that this code belongs to a later to-do. Then a "ToDo:" (e. g. as Eclipse tag) should also be displayed here. And of course it has to be ensured that the corresponding item ends up in backlog lists somehow.

Delete Useless Comments

Delete Useless Comments

This refers to the old discussion: A camp is of the opinion that everything should be documented (i. e. every command line, however understandable it may be). On the other hand, there are those who argue that the code must be written so clearly that no comment is needed. The reader can thus recognize

1. The meaning of the code line and

Information about the actual implementation

However, there is agreement that code should be sufficiently documented at least at the package or class level in order to get an abstract introduction to the sources. This applies in particular to API design.

Two Examples:

Code

```
// Get to know how much was deleted.
x = tmpObj.getDelI();
```

```
// Get to know how much was deleted.
itemsDeleted = Repository.getLastDelItems();
```

Übung CCD-01: Diskussion

Discuss this topic with your fellow students or lecturers. Make yourself a few key points.

- Referenz: "5 reasons to avoid code comments"

Invest in a Precise Naming

Invest in a Precise Naming

This is not only about variables, but also about fields, methods, interfaces, classes or packages. Here too, it should be borne in mind that

- the code is mostly read and therefore has to transport as much information and context as possible.
- On the other hand, however, it must not be too long, so that manual refactoring or enhancements do not take up too much time or make the line unreadably long.

Here it is really worthwhile to find the precise description that represents the optimum of both poles. Names should be self-explanatory. The naming must be consistent. The aim is to achieve symmetry.

Here are some general rules:

- Try to find the best level of abstraction.
- Interfaces should describe the functionality they abstract (Clonable has . clone () method)
- Classes describe the type of implementation. (SSDStream implementations IStream)
- Methods describe what they do but never how they do it.
- Do not encode any type or scope information into the name.
- If something has a side effect, this should be described in the name.

Anmerkung

Scope

dt. = Bereich, means the area of visibility

The last point is coded in some modern languages like Ruby or D as extra characters. In Ruby there are the methods
capitalize (String)
and
capitalize! (String)
. Here, the rules make it immediately obvious which of the two methods has a character that changes from the beginning.

There are standards for all languages, but these differ widely (e. g. Java, Ruby, D, Clojure) but must be applied consistently.

The same applies to methods: you should complete a task. See also point:

→ Single Level of Abstraction (SLA)

Code should read like normal English text / prose:


Code

```
if(plane.altitude() < 100 * tolerance)
 generiereWarnung("Proximity Alert!");
```

Avoid Magic Numbers

Avoid Magic Numbers

Have a look at the following example:


Code

```
MoveList ml[50];
```

In extremely many programs you will find' Magic Numbers'. In 50% of the cases you might even be able to find out with a little effort what this number means and why 40 or 70 was not taken here. In many cases, you will never be able to find out.

On the other hand, the following is much more enlightening:

Code

```
// (in the sourcefile consts)
immutable MAXDEPTH = 50; // For dynamic.d how far will we search
//in chess at maximum

// (now in the file dynamic.d)
MoveList ml[MAXDEPTH];
```

The problem is that very few developers have enough empathy to imagine that another developer spends a lot of time trying to figure out the cause of the magic number. And it is not only about time, but also about the inner attitude and acceptance of third parties to the currently written code.

Prefer Polymorphism to if/else or switch/case

Prefer Polymorphism to if/else or switch/case

But what is the problem? Every developer is used to
if/else
write chains and find nothing. And of course, code fragments are difficult to polymorphise when it comes to pure numbers and / or arithmetic.

Nevertheless, with
switch-case
behavior is often hard wired. In many cases, variants are much more elegant in which the behavior called after
if/else
or in
switch-case
constructs can be called much better by a polymorphic behavior.

No Sideeffects

Ensure that "Changes have Local Consequences"

Besides the classic NullPointer exceptions, side effects are probably one of the main reasons for errors in large software systems.

A typical example is that other modules or methods rely on variables such as age, but they can then assume negative values by mistake. Often, counters are increased for certain actions. Then also during testing. Then a counter that is too high can have negative effects again, etc.

From this burning problem, new research areas and programming languages have developed. In research, the researchers are trying to prove the correctness and' side-effect freedom'. Programming languages such as Haskell or Clojure are inherently almost free of side-effects (unless you misuse some constructs of the language). In functional programming, a method must not have side effects. It should be pure. For this reason, variables have almost been abolished there. If programs have to change the state' global', this is done via different identities of a variable and strictly controlled and threadsafe.

Fields shall define state

Fields shall define state

Fields are often misused to hold temporary data. Fields should represent strict states of the object.

It helps here,

1. local variables,
2. or extract another class that can calculate or maintain temporary states.

Correct Exception Handling

Think about Exception Handling

Exceptions should always be collected as specifically as possible. Only catch all possible types and only exceptions if necessary. Exceptions only catch exceptions if you are on the right level, i. e. you can do something decent with them. Otherwise, the next higher level should do something about it. Never abuse exceptions for regular control flow. Exceptions must be dealt with properly. Otherwise, the system is threatened to remain in an inconsistent state.

Know and apply refactoring patterns

Every developer should:

1. the Refactoring Catalogue (weblink) and have read the relevant literature on this subject.
2. know the refactoring menus from the current IDE inside out and have gained practical experience with them.

Only with this knowledge, these tools and a version control system underneath can code be changed efficiently.

4.4.1 Coding Source

Lokale Deklaration

Vertical Separation

If possible, local variables should always be declared where they are used. A collective declaration at the beginning of the block must be avoided. The declaration should be done before or in the first use and then have as small a scope as possible.

Questions:

1. Why are collective declarations at the beginning of a block usually not useful?
2. What does the length of the scope of a variable depend on?
3. Should variables be explicitly destroyed?

Explanatory Variablen

Explanatory Variables

As long as the program does not have to be 100% optimized for performance (and this is rarely necessary), you should implement the optimal number of explanatory variables. This makes the individual steps understandable, easy to read and maintain.

Beispiel:

Code

```
float net = gross * 0,85;
float tax = net * 0,75;
float fee = tax * 0,1;
```

versus

```
result = input * 0,85 * 0,1 * 0,75;
```

Nesting

Nesting

Deep nested code should take on ever more specific and detailed tasks. The abstraction level or the call probability should be higher at higher levels.

Separate Multi-Threaded Code

Separate Multi-Threading Code

Multi-threaded code should not be mixed with' normal' code. A thread should (and often must) always be encapsulated in its own class at startup.

Code Example:

Code

```
public class SwtThread extends Thread {
 public void run() {// here it happens
 }
}

class X {
 Thread thread;
 void foo() {
 thread = new SwtThread ();
 thread.start();
 }
}
```

4.4.2 Coding Conditionals

Encapsulate Conditionals

Encapsulate Conditionals

Reading complex logic operations is always much more time-consuming than reading a good, explanatory method.

It should therefore always be considered whether it makes sense to encapsulate complex logic.

Example:

Code

```
if (this.ShouldBeDeleted(timer))
```

liest sich einfacher als

```
if (timer.HasExpired && !timer.IsRecurrent)
```

(C) CC Cheat Sheet

Avoid Negative Conditionals

Avoid Negative Conditionals

Negative logical expressions are more difficult to read than positive expressions. Avoid it if possible.

Example:

Code

```
not > 0 versus <= 0 oder
not positive(x) versus nevative(x) // if 0 does not play a role
```

ncapsulate Boundary Conditions

Encapsulate Boundary Conditions

Boundary conditions for e. g. loops like
while / do-loop / loop-do
should be clearly marked and listed in one place. Under no circumstances be hidden in the core code of the loop. For example, if the loop needs to be changed, refactoring would be more difficult.

Example:

Code

```
nextLevel = level + 1; // gefolgt von
```

```
this.callLevel(nextLevel);
```

4.4.3 Coding Principles

Don´t Repeat Yourself (DRY) (rot)

Don´t Repeat Yourself (DRY)

Analyses have shown that copy-and-paste programming is one of the most common sources of error!

This practice and code repetitions involve a variety of problems:

You are guaranteed to forget to turn a screw at some point, because it's not really thought of. The code no longer needs to be maintained because changes to the basic structure of the algorithm must be made everywhere.

What you should do instead is to encapsulate or parameterize the algorithm in a class. This is based on the DRY principle: avoiding the same code points where possible, whereby this principle does not only apply to code points. DRY is one of the most important principles of Ruby on Rails: definitions have to be done only once. In contrast, there are many specifications in which names, code or configurations must occur several times in different places. This is then a pleasant field for code or configuration generators. But of course, the automatic and invisible generation in the program is always better than repetitions (with generator help) considering the DRY principle. The EJB 2 specification is a nice negative example, since descriptors and interfaces and classes often have the same information content.

Example:

Code

```
code code
```

Einfachheit als oberstes Prinzip (rot)

Keep it simple, stupid (KISS)

Schaut man sich die Liste der Gründe an, warum IT Projekte laut IT-Managern angeblich scheitern, so ist von Komplexität fast nie etwas zu hören. Dennoch zeigen auch hier Studien, dass die intrinsische Komplexität ein nicht zu unterschätzender Faktor ist. Menschen sind anscheinend nicht unbedingt perfekt darauf ausgelegt, die Architektur

von tausenden Klassen zu überblicken und diese inhärente Komplexität zu beherrschen ("komplexity kills").

Anmerkung

Unter der intrinsischen Komplexität versteht man die enthaltene, d.h. quasi naturgemäße Komplexität eines Problemes. So ist z.B. die intrinsische Komplexität der Lösung der Goldbachschen Vermutung einfacher als das Sieb des Erestotenes (Primzahlen finden).

Aus diesem Grunde ist es auch ein bekanntes Pattern, Komponenten so aufzuteilen, dass diese dann über ein Protokoll kommunizieren und nicht mehr via "direct call". Falls dies von der Performance her funktioniert, wird so Komplexität verringert.

Aber das KISS Prinzip ist auch mit dem XP Prinzip der (Vermeidung der) "goldenen Wasserhähne" verwandt. Also um Features, die noch keiner braucht. Es geht hier also darum, nicht nur die Menge, sondern auch die Struktur einfach zu halten und erst später zu erweitern.

KISS hilft den Code zu verstehen. Besonders von anderen Entwicklern.

Aber auch dieses Prinzip ist nicht auf Code beschränkt. Es geht auch hier um Datenstrukturen, Code, Komponenten- und Paketstrukturen und Architekturen.

Beispiel:

Code

```
function quicksort('array')
 if length('array') = 1
 return 'array' // an array of zero or one elements is already
 sorted
 select and remove a pivot value 'pivot' from 'array'
 create empty lists 'less' and 'greater'
 for each 'x' in 'array'
 if 'x' = 'pivot' then append 'x' to 'less'
 else append 'x' to 'greater'
return        concatenate(quicksort('less'),        'pivot',
 quicksort('greater'))
// two recursive calls
```

versus

```
DataProcessor.sort(myList);
```

Dieses Prinzip hängt auch mit dem YAGNI Prinzip zusammen (bzw. ist oft äquivalent).

Code soll bitte "kommunizieren, einfach und flexibel sein"

Code Communication, Simplicity, Flexibility (Beck).

Viele dieser Regeln wurden schon von
Kent Beck
in seinem Buch "Implementation Patterns" (*Addison-Wesley, 978-0321413093*) zusammengefasst.

Die Punkte sprechen die folgenden Inhalte an:

- Code Kommunikation => Selbsterklärender Code für Dritte, der Kommentare überflüssig macht
- Simplicity => KISS Prinzip
- Flexibility => Änderbarkeit und damit auch Wartbarkeit des Codes

Vorsicht vor Optimierungen! (rot)

Avoid Early Optimizations

Frühe Optimierungen neigen dazu die Dinge

1. Zu stark zu komplizieren
2. Zu viel Zeit zu investieren
3. Einen Optimierungsdruck anzunehmen, der so vielleicht gar nicht existiert.

Möglichkeiten der Optimierung: Hardware / Software (Profiler), Datenstrukturen / -typen

Lesen sie dazu:

http://www.codinghorror.com/blog/2008/12/hardware-is-cheap-programmers-are-expensive.html

Darin wird verwiesen auf (M.A. Jackson):

Hinweis

Rule 1: Don't do it.
Rule 2: (for experts only): Don't do it yet.

"Doing" versus "Calling" Code(orange)

Single Level of Abstraction (SLA)

Worüber Entwickler oftmals nicht nachdenken ist, auf welcher Abstraktionsebene der geschriebene Code liegt.

So unterscheidet man grob gesagt zwischen

A) "doing" Code: d.h. Code der wirklich etwas tut (z.B. berechnet) und

B) "calling" Code: dies ist Code, der konkreteren Code aufruft.

Entscheidend ist, dass der Developer sich klar sein sollte, welchen Code er in der Methode oder in der Funktion aufruft. Auch hier gilt es sich in eine lesende Person hineinzuversetzen und dieser die Entscheidung zu überlassen, ob man abstrakten Code oder konkreten Code lesen möchte. Man erhöht damit stark die Geschwindigkeit, in der der Code verstanden wird.

Beispiel:

Code

```
// mit Vermischung!
public int calculateSpecificTax(int income, int rate1, int
 rate2,
int amortization){
 int tmp = income * rate1 + 10;
 tmp = tmp * rate2,
 tmp = calculateAddOn(tmp); // andere Ebene!
 tmp = tmp * 0,97;
 return tmp;
}



// ohne Vermischung!
public int calculateSpecificTax(int income, int rate1, int
 rate2,
int amortization){
 int netGain = calculateNetGain(income, rate1);
 int netReturn = calculateOffsetRate(netGain , rate2);
 int grossResult = calculateAddOn(netReturn);
```

```
 return calculateFinalTax(grossResult);
}
```

Keep Configurable Data At High Levels

Keep Configurable Data At High Levels

Configurations or default values should be located at the correct level. So it is better to be read at a higher level of abstraction and then read or transferred to deeper functions. Configurations or default settings are often hidden in low-level functions, so searching for them takes a lot of time.

Configurations: be careful

Over-Configurability

Good configurations are a blessing in the first place. With property files or the numerous configuration frameworks, changeable parameters can be extracted from the code.

On the other hand, many large programs now have hundreds to thousands of setting options. And not infrequently, installations fail because of "over-configurability". This means that too much configurability is often stored externally. The complexity increases here too, and no one can see through.

There are many possibilities how to:

1. Working with default values and subsequent adjustment
2. The program can detect the optimal configuration (e. g. number of cores, availability of data / files).
3. Remove configuration parameters
4. etc.

Often it is advantageous to be able to agree on at least one framework (such as Commons Configuration, Spring or Obix).

Hidden Temporal CouplingRule

Hidden Temporal CouplingRule

If possible, the call of methods should not specify a fixed sequence. If this is the case, you have to make sure that the sequence cannot be called incorrectly.


Beispiel

Example:
kein
read();
vor
open();

Don’t Be Arbitrary

Don’t Be Arbitrary

Guidelines for coding or structuring the code must be logical and maintained. The structure must be consistently communicated by the code.

If there are fluctuations and the guidelines are not adhered to, others feel encouraged to change the code.

4.5 Code Quality

Use Constraints

Use assertions! Pre / Post Conditions

Code quality must be performed at the lowest level in the form of simple tests.

There are basically three forms of this:

1. **Pre-conditions:** Test whether all parameters contain correct values.
2. **'Post-conditions:**' Test whether all return parameters are correct.
3. **'Assertions:**' Test whether variables and objects in the function code are correct.

Besides copy and paste, return null is one of the most common programming errors. The problem with empty and unset return values is that these empty objects are not easily traced back and can be passed up. And that will be expensive.

A) The simplest approach is to check variables, parameters and return values in this way:


Code

```
int myFunc(int x, object param1) {
 if(param1 == null)
throw new IllegalArgument("myFunc arg is empty!");
```

```
 ... // code: age=x*5;
 assert(age > o);
 ... // code: result = x * param1.factor;
 if(result == null)
throw new IllegalArgument("result is null!");
 return result;
}
```

Although the success of testing is guaranteed, there are still several problems:

Coding if chains in the function is a nuisance. It pollutes the code because it is orthogonal functionality (an aspect) that has nothing to do with the actual function task.

B) This can be remedied by various validation frameworks (e. g. Apache Commons or Validation) or essays forContract-Programming.

These provide at least macros to simplify the queries.

C) Of course, it is even better if the language itself supports contract programming. There are a number of programming languages like Eiffel or D.

One exampe for this:

Code

```
long square_root(long x){
 in{assert(x >= 0);}
 out(result){assert((result * result) <= x && (result+1) *
 (result+1) >= x);}
 body{return cast(long)std.math.sqrt(cast(real)x);}
}
```

D) Another important possibility is, of course, to completely remove the aspect of testing from the function. This can be done well with AOP, for example. At least for entering and exiting the function, you can define jointpoints. See also the unitAOP - Aspect Oriented Programming.

4.5.1 Source Code Konventionen

Title: Source Code Conventions

Check your SourceCode Conventions

Nowadays, it is easy to verify source code conventions. A large number of tools are available for this purpose.

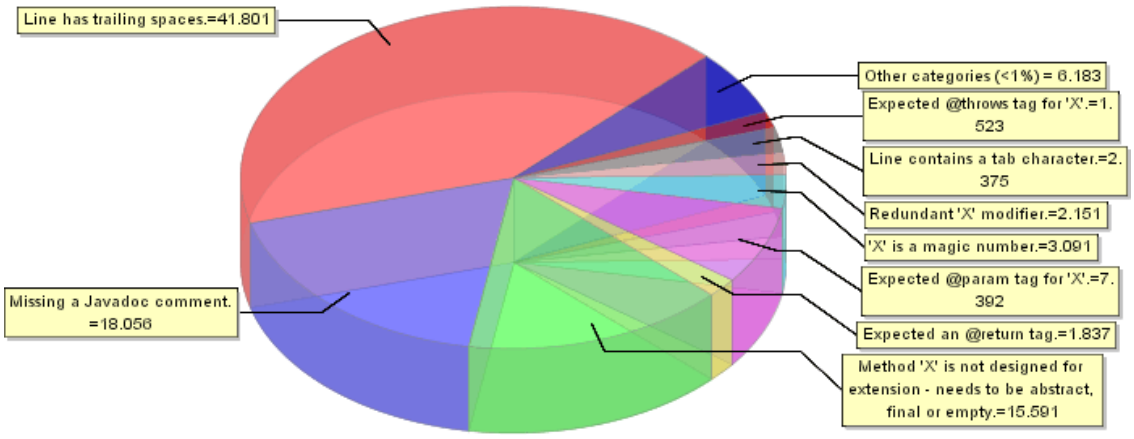
There are some well-known tools for Java alone:

- **Checkstyle**
- **PMD**
- **FindBugs**

The real problem is this:

Integrate this seamlessly into the IDE so that errors are immediately visible and Configure these tools correctly.

For the latter, gigantic XML files often have to be edited in order not to be confronted with thousands of error messages every time.



Percentage distribution of error messages

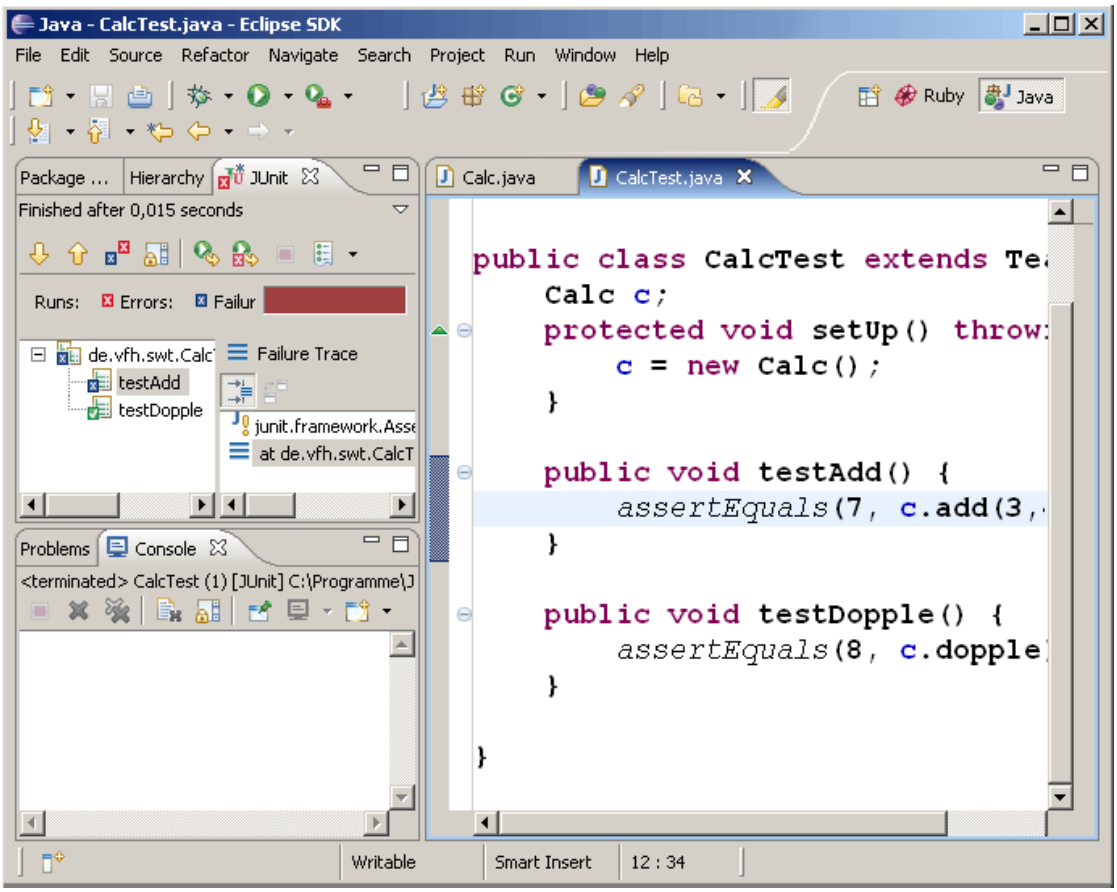
4.5.2 Automatisierte Unit Tests

Title: Automated Unit Tests

Automated tests offer double benefits:

- You save time
- They take fear

The more a codebase is subject to change, the more time is saved. Where code changes, new and old (regression tests) have to be tested again and again. Automation simply saves time. And the more complex the code, the greater the reduction in fear. If complex code needs to be changed - to add functionality, optimize it or simply correct it - there is a high risk of inadvertent errors being introduced. However, small steps of automated tests reveal these, so there is no reason to be afraid of "making things worse".
(C) CCD Westphal / Lieser



Testen mit JUnit

4.5.3 Code Coverage Analyse

Title: Code Coverage Analysis

Code Coverage is the test coverage through test cases. Code Coverage also shows the untested code points.

In practice, a test coverage of 80% is a desirable goal. A high test coverage has become an important marketing tool and plays an important role in awarding and acceptance of orders. There are excellent (also free tools) that can automate the process of code coverage analysis and perform it quite comprehensively.

Übung CCD-02: Discussion „Much helps a lot?“

A high test coverage indicates high code quality!

Would you agree with that statement? Write down briefly arguments that speak both for and against this statement. Keep your remarks ready for the next synchronous meeting.

Processing time: 10 minutes

A good and in-depth introduction to Code Coverage can be found on the Internet under:

http://www.bullseye.com/coverage.html

There are numerous code coverage tools for Java. On the Internet, for example, you can find them at the address:

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

The most important tools are:

- Clover (commecial), test version available
- Emma (emma.sourceforge.net), Open-Source
- Cobertura (cobertura.sourceforge.net/), Open-Source

Other tools are: Quilt, NoUnit, InsEct, Hansel, Jester, JVMDI Code Coverage Analyse, Grobo, jcoverage, JBlanket,...

(Please do research by yourself and find the leading tools!)

Example from Clover

Beispiel

3 Types of Measurement:

Statement Coverage

Number of statements that pass through a class

Conditional Coverage

How to run through conditional constructs (e. g. if/else). Ideally, the test should cover everything.

Method Coverage

How many methods of a class are called.

Formel

The entire coverage is derived from a more complex formula:

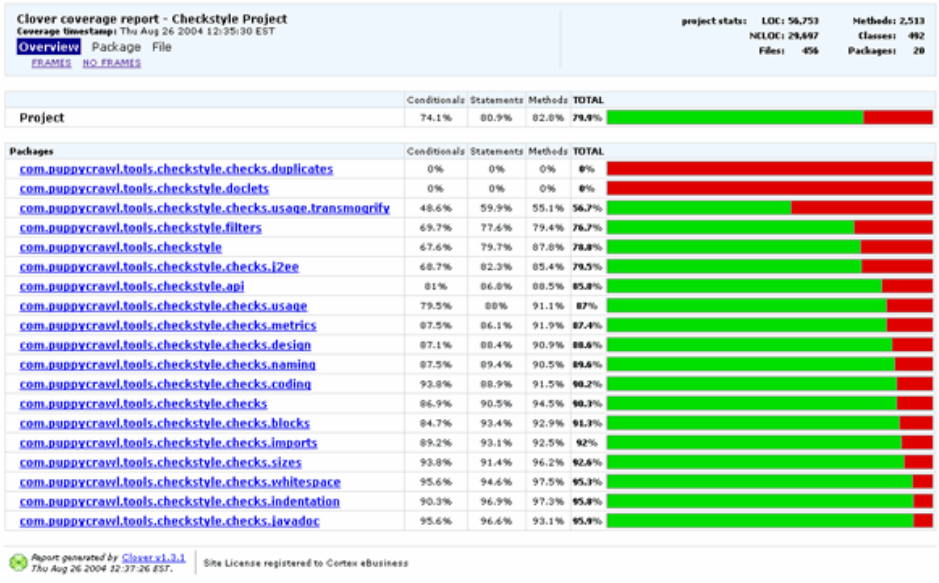
TPC = (CT + CF + SC + MC)/(2*C+S+M)

CT = conditionals that evaluated to „true“ at least once

CF = conditionals that evaluated to „false“ at least once

SC = statements covered MC = methods entered

The most important thing for the project manager/management is to get an overall overview of the coverage:



Clover coverage report - Checkstyle Project
Coverage timestamp: Thu Aug 26 2004 12:35:30 EST
Overview Package File
FRAMES NO FRAMES

project stats: LOC: 56,753 NCLOC: 29,697 Files: 456 Methods: 2,513 Classes: 492 Packages: 20

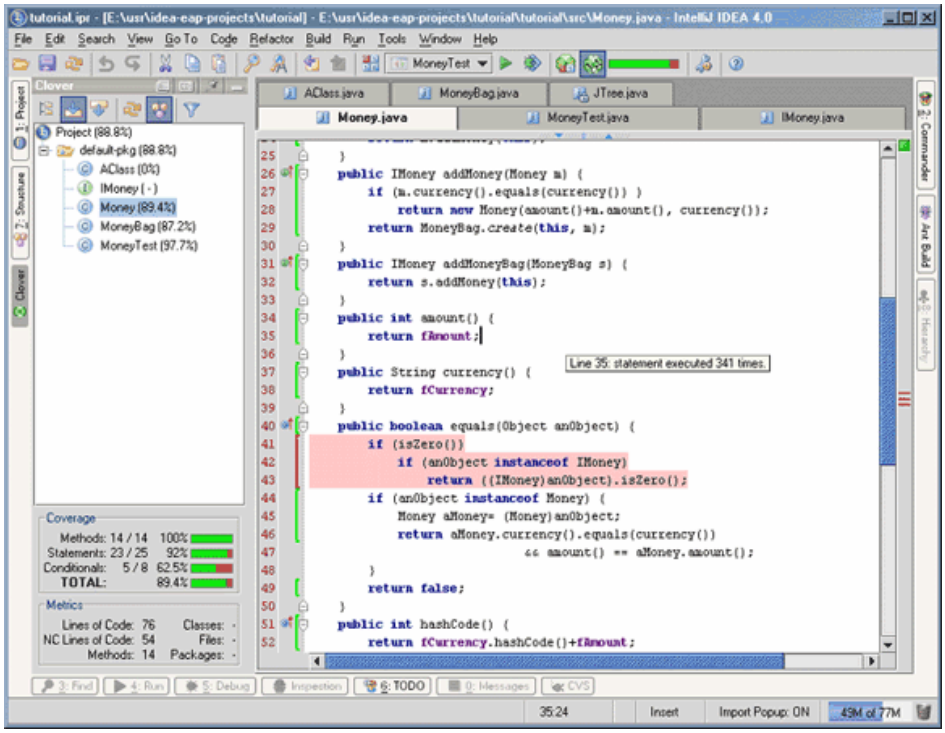
	Conditionals	Statements	Methods	TOTAL
Project	74.1%	80.9%	82.8%	79.9%

Packages	Conditionals	Statements	Methods	TOTAL
com.puppycrawl.tools.checkstyle.checks.duplicates	0%	0%	0%	0%
com.puppycrawl.tools.checkstyle.doclets	0%	0%	0%	0%
com.puppycrawl.tools.checkstyle.checks.usage.transmogrify	48.6%	59.9%	55.1%	56.7%
com.puppycrawl.tools.checkstyle.filters	69.7%	77.6%	79.4%	76.7%
com.puppycrawl.tools.checkstyle	67.6%	79.7%	87.8%	78.8%
com.puppycrawl.tools.checkstyle.checks.j2ee	68.7%	82.3%	85.4%	79.5%
com.puppycrawl.tools.checkstyle.api	81%	86.8%	88.5%	85.8%
com.puppycrawl.tools.checkstyle.checks.usage	79.5%	88%	91.1%	87%
com.puppycrawl.tools.checkstyle.checks.metrics	87.5%	86.1%	91.9%	87.4%
com.puppycrawl.tools.checkstyle.checks.design	87.1%	88.4%	90.9%	88.6%
com.puppycrawl.tools.checkstyle.checks.naming	87.5%	89.4%	90.5%	89.6%
com.puppycrawl.tools.checkstyle.checks.coding	93.8%	88.9%	91.5%	90.2%
com.puppycrawl.tools.checkstyle.checks	86.9%	90.5%	94.5%	90.3%
com.puppycrawl.tools.checkstyle.checks.blocks	84.7%	93.4%	92.9%	91.3%
com.puppycrawl.tools.checkstyle.checks.imports	89.2%	93.1%	92.5%	92%
com.puppycrawl.tools.checkstyle.checks.sizes	93.8%	91.4%	96.2%	92.6%
com.puppycrawl.tools.checkstyle.checks.whitespace	95.6%	94.6%	97.5%	95.3%
com.puppycrawl.tools.checkstyle.checks.indentation	90.3%	96.9%	97.3%	95.8%
com.puppycrawl.tools.checkstyle.checks.javadoc	95.6%	96.6%	93.1%	95.9%

Report generated by Clover v1.3.1 Thu Aug 26 2004 12:37:26 EST. | Site License registered to Cortex eBusiness

Example Report by Clover

The most important thing for the developer is to get feedback in a short period of time about which methods are not covered:



Example Report by Clover

Both graphics can be produced in just a few minutes with the Emma Eclipse plug-in Eclemma.

(Source: LE SWT Bachelor)

4.5.4 Zuerst Testen

Title: Test first

The Test First approach means to write the test first and then the code.

This has important side effects:

- First of all, you have to think about interfaces / interfaces and signatures. * From the point of view of the user of the' Unit'.
- The code is easier to read.
- The code is automatically written with better quality.
- The time it takes to find errors is drastically reduced (fast feedback - see[CIC - Continous Integration / Continous Delivery]].
- Faster error detection means enormous cost savings.

Conversely, units that are implemented first are usually much more difficult to test. Test First changes the design.

Übung CCD-03: Code-Katas

Try this with a simple code-catas.

Time needed: about 35 minutes

Anmerkung

Code-Kata
The term **Kata** means practice and is used in the context of karate. For a simple programming problem, a solution should be found within 30 minutes (up to a maximum of one hour). The solution will then be presented. The goal is to train and improve one's own skills.

4.5.5 Mockups

Use Mocks

Translation programs provide the following explanation for mock objects: dummy, counterfeit, imitation, pseudo-, sham, fake and deceive (see also the learning unit "Testing" from the bachelor's degree).

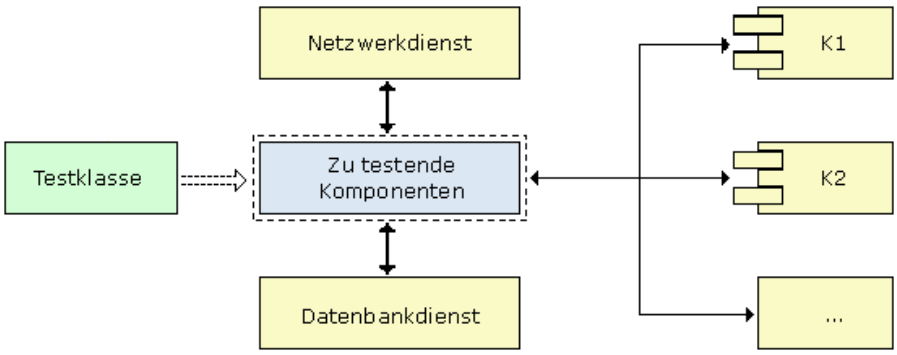
In too many cases, it is not easy to test a unit because the component has too many dependencies. Therefore, it would be desirable for the tests to have test objects that simulate a real environment for a component to be tested. These objects are called mock objects. For example, an object that pretends to be a database. This would then return any result sets, but this is not relevant for the test.

Known mock tools for Java are:

1. mockito
2. EasyMock
3. jmock

Mock tools are available for all languages. These are particularly simple for dynamic languages (e. g. FlexMock in Ruby).

The application of a test or test class to individual service components poses a problem. Service components are usually dependent on other components or services. Mock objects, on the other hand, can isolate the component to be tested and simulate a real environment. The figure below illustrates dependencies between K1, K2, database services, network services and the component to be tested.



Dependencies for a component to be tested

Having a mock at hand makes testing much more efficient. And even if you don't want to test, you can emulate functionality or behavior.

Here are two good links about mocks and stubs and their effects on testing: (thx MC!)

- Martin Fowlers Blog
- Stackoverflow Eintrag

4.5.6 Funktionale Techniken

Title: Functional Techniques

With functional programming techniques, safe programs can be developed, since, for example, freedom from side-effects is important.

See also the learning unitPPD - Programming paradigms.

Important techniques are:

- Use of first-class /Higher Order Functions: e. g. functions as argument or return value; e. g. generation of functions
- Pure Functions: Use of Recursion (or Tail-Recursion)
- Immutable Data: The use of data structures that cannot be changed.

and many other concepts such as lazy evaluation, strict evaluation, type systems, etc.

Often, performance and memory consumption are exchanged here in favor of error-free code or unclear data status. However, this is usually irrelevant as the Bottleneck is located somewhere else. However, it must of course be decided on a case-by-case basis.

4.5.7 Lange Funktionen aufteilen und Reviews

Title: Long functions and Reviews

Split long methods

Method Object

Of course, there are many refactorings at the code level. But only a few have as much impact on readability and changeability as the Method Object Pattern.

Usually, for long functions, the first thing to look for is whether the Extract Method Pattern can be used. However, this is often not simply due to the parameter lists and the naming. Here
Kent Beck
("Implementation Patterns", ISBN-13:978-0321413093) indicates the following procedure:

1. Create a class named exactly like the method: e. g. complexCalculation () becomes the class ComplexCalculator.
2. Create a field in the class for each parameter, variable, and field used in the original method. Leave the names the same.
3. Build a constructor for these variables or parameters
4. Copy the old
   complexCalculation ()
   method to this class. The parameters now become references to the fields!
5. Now replace the body of the original method with code that creates an instance of this class.

Code

```
complexCalculation(){
 new ComplexCalculatior().calculate();
}
```

1. If there were fields in the old method that were used, set them after the call:

Code

```
complexCalculation(){
 ComplexCalculator calculator = new ComplexCalculator();
 calculator.calculate();
 mean = calculator.mean;
 variance = calculator.variance;
}
```

Now check if the code still works. But after Beck, the fun only really starts now, because refactoring is now easy. Extract Method works wonderfully now, since all reference fields are in the class. Fields can often be transformed into local variables. And of course, you can now also delete fields and pass them as individual parameters into the new extracted functions to make them clearer and easier.

Reviews

Code reviews should be an integral part of a quality concept. These are already integrated in XP based procedure models as pair programming. Unfortunately, pair programming is not part of every project.

Therefore, there are often review processes at the end of the work step. Google reports that five' code approvals' are often required for the code to go into production.

Important effects here are important:

- Learning Effects / Continuing Education
- Often 90% of the errors are found, which are later found via expensive debugging. This saves a huge amount of money!
- Code ownership is changing positively.

4.5.8 Messen von Fehlern

Title: Measuring and Tracking Errors

Errors occur during software development. These occur in all phases: misunderstood or unclearly formulated requirements lead to errors as well as incorrect implementations. In the end, everything is a mistake, which leads to the customer receiving software that does not meet his requirements. Iterative approach and reflection are two building blocks that serve to improve the process. However, in order to determine whether an

improvement is actually taking place, there must be a measure of the extent to which a development for the better can be seen.

The measurement of the errors can be carried out by counting or by time measurement. The focus is not on precision as long as the measurement method provides comparable data. The tendency to develop over several iterations should become apparent. Furthermore, it is not a question of clarifying the responsibility for an error. In the end, it doesn't matter who caused the mistake, as long as the team learns from it and improves its process.

Which faults are to be measured? It is not the errors that occur during development. These are unavoidable and hopefully lead to a faultless product being delivered at the end of an iteration. Rather, it is about the errors that are reported back after an iteration by the customer or his deputy (e. g. productowner or support. These are errors that hinder the implementation of new requirements. The errors to be measured are therefore those that occur when you believe that they should not exist.

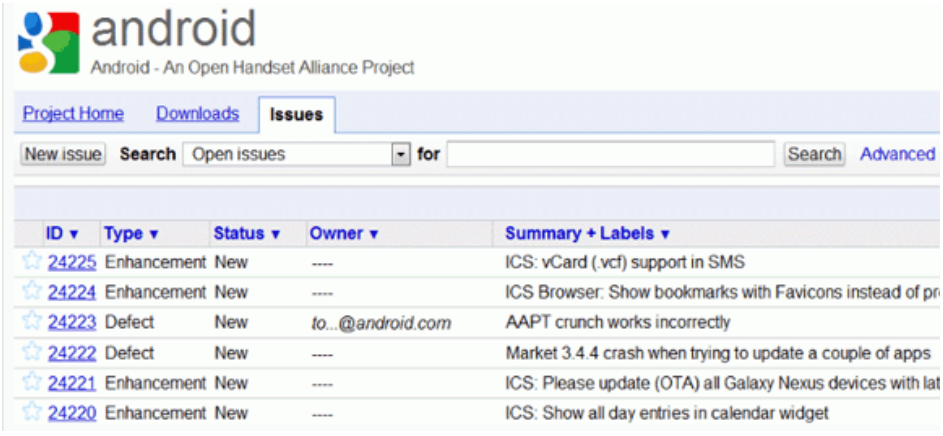
When a team reaches this point in the process and curses, because there is such a flaw in the rest of the work, is to be determined individually for each team.

(C) CCD

Issue tracking systems (or defect tracking) are widely used. All modern code hosters like Google Code, github, Jira Cloud, etc. have integrated these systems.

These systems are available out-of-the-box as a service. So there is no longer any obstacle to deployment.

Here is an example where Android OS is hosted on Google Code:



Issue Tracking System

4.5.9 Statische Codeanalyse

Title: Static Code Analysis

Static Code Analysise (Metrics) / Measure Complexity (green)

The results of a static code analysis are usually extremely informative. These are usually, for example, complexity metrics that provide strong evidence of shortcomings in code design or architecture. For example, there are the following metrics:

- Basic metrics (lines, methods, classes, packages, files, duplicates, etc.)
- McCabe Cyclomatic Complexity
- Halstead metric
- Code Coverage (as pure' visit metrics')
- Test coverage
- Style Injuries
- JDepend
- and coupling metrics

belong to the latter:

- Pressman metrics
- ACD / rARD metric

Anmerkung

rARD Metric
With this metric, each module is initially given the weight 1. Then the weight on which it depends is added for each module.

In addition to the test and readability of high-quality code (which has good metrics), Westphal also describes' evolvability' as a central element of the results of a code analysis or the subsequent refactoring.

4.5.10 Der Technical Debt Deines Produktes

Title: The Technical Debt of a Product

Measure your 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 guilt. For example:

- Bad architecture / design
- many code metrics violations
- too much complexity
- over-coupling
- cycles

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 (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? In turn, man-days can be converted into monetary value - depending on the wage level.

The decisive factor is that - as already mentioned in the introduction - software costs do not only consist of development costs, but that in individual cases they 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:

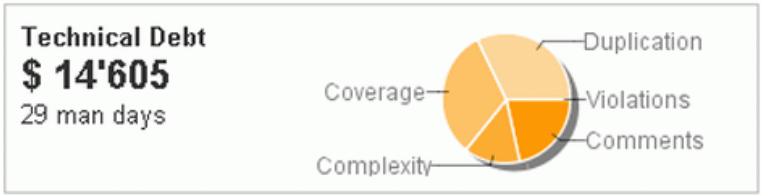
Debt(in man days) = cost_to_fix_duplications + cost_to_fix_violations + cost_to_comment_public_API + cost_to_fix_uncovered_complexity + cost_to_bring_complexity_below_threshold

As well 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
Complexity	= cost_to_split_a_class * (class_complexity_distribution >= 60)

Impact Factors for the Technical Debt Calculation

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).



Sonar - Calculation of Man-Days

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's like the JDepend metrics. 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.

Daily reflections

Good developers are characterised by being able to look at themselves from a bird's eye view. And asking yourself a lot of questions:

- What did I encode?
- How did I code?
- Did I set myself realistic goals?
- Did I implement Clean Code?
- Why could some things not be implemented?
- Did I learn?

4.6 Architecture und Class Design

Besser Instantiieren als Vererben (rot)

Favour Composition over Inheritance (FCoI)

Text will follow...

Pro Komponente nur eine Aufgabe (orange)

Single Responsibility Principle (SRP)

Text will follow...

Aufgabentrennung(orange)

Separation of Concerns (SoC)

Text will follow...

Text 1 (gelb)

Interface Segregation Principle (ISP)

Text will follow...

Geheimnisprinzip durchsetzen (gelb)

Information Hiding Principle

Text will follow...

Inversion of Control Container (grün)

Use Dependency Injection

Text will follow...

Verwende Dependency Injection (Framework!) (gelb)

Dependency Inversion Principle

Text will follow...

Text 2 (gelb)

Liskov Substitution Principle

Text will follow...

Text 3 (gelb)

Komplexe Refaktorisierungen

Text will follow...

Text 1 (grün)

Open Closed Principle

Text will follow...

Text 2 (grün)

Tell, don´t ask

Text will follow...

Text 3 (grün)

Law of Demeter

Text will follow...

Prüfe die Anwendung von Basispatterns

Check all Basic Patterns: Pluggable Selector, DSL pattern, ...

Text will follow...

Entwurf und Implementation überlappen nicht (blau)

Design and Implementation must not overlap

Text will follow...

Implementation spiegelt Entwurf (UML2Wall) (blau)

Implementation corrosponds Design

Text will follow...

Visualisiere, Messe & Constraine die Architektur (Sonar, SonarJ, JDepend,..)

Visualize, Measure and Constrain your Architecture

Text will follow...

Prüfe den Einsatz von Einterprise Patterns

Check Enterprise Patterns

Text will follow...

4.7 Packages

Put together what belongs together

Common Closure + Common Reuse Principle (CCP, CRP)

This principle means that a class should be designed for this:

A) not to be changed

B) but may indeed experience extensions

Beispiel

(ToDo)

Class: Price calculation

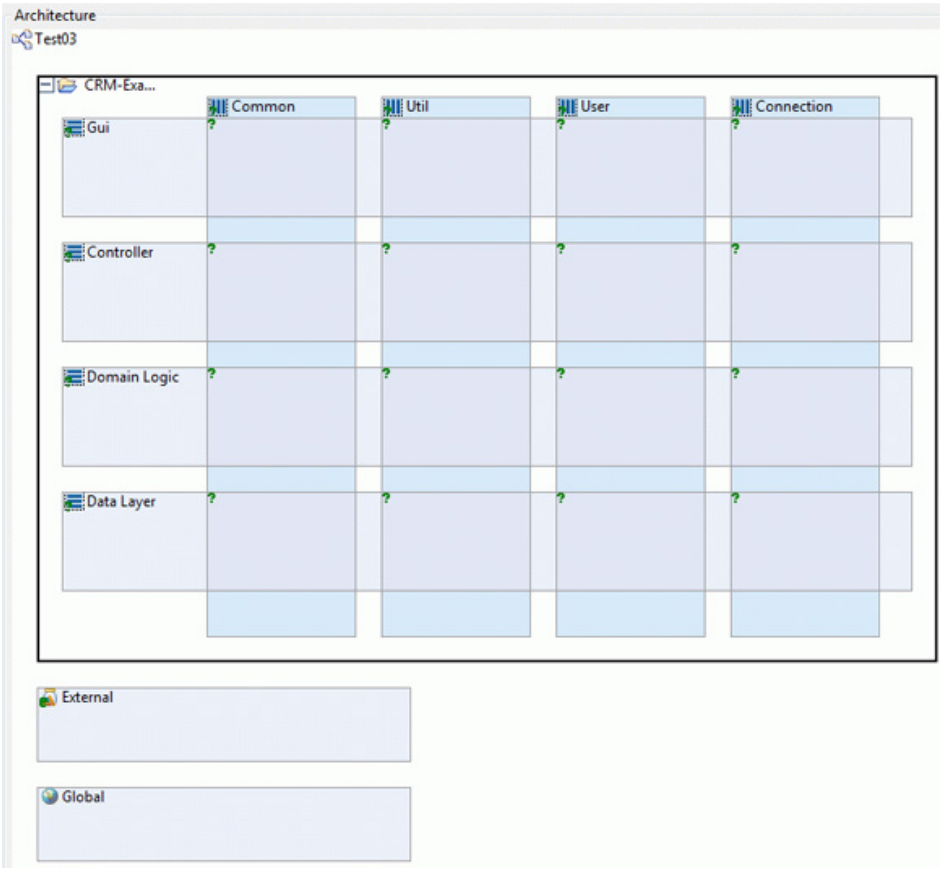
Class: Price calculation incl. Net amount with interface before

Reference: http://www.objectmentor.com/resources/articles/ocp.pdf

Visualize Package Dependencies

Dependency graphs of packages

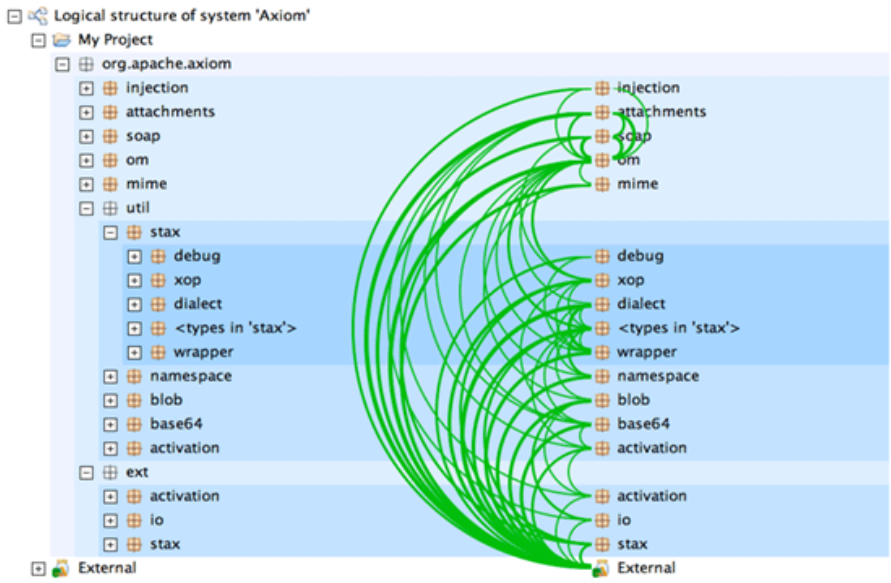
The representation of components as grids is common and helpful for the visualization of components. This display can be used to check whether packages and components really only access down or to the right! Access to the top or possibly also to the left should be avoided, as there is a risk of unchangeable spaghetti code.



Definition of an Architecture

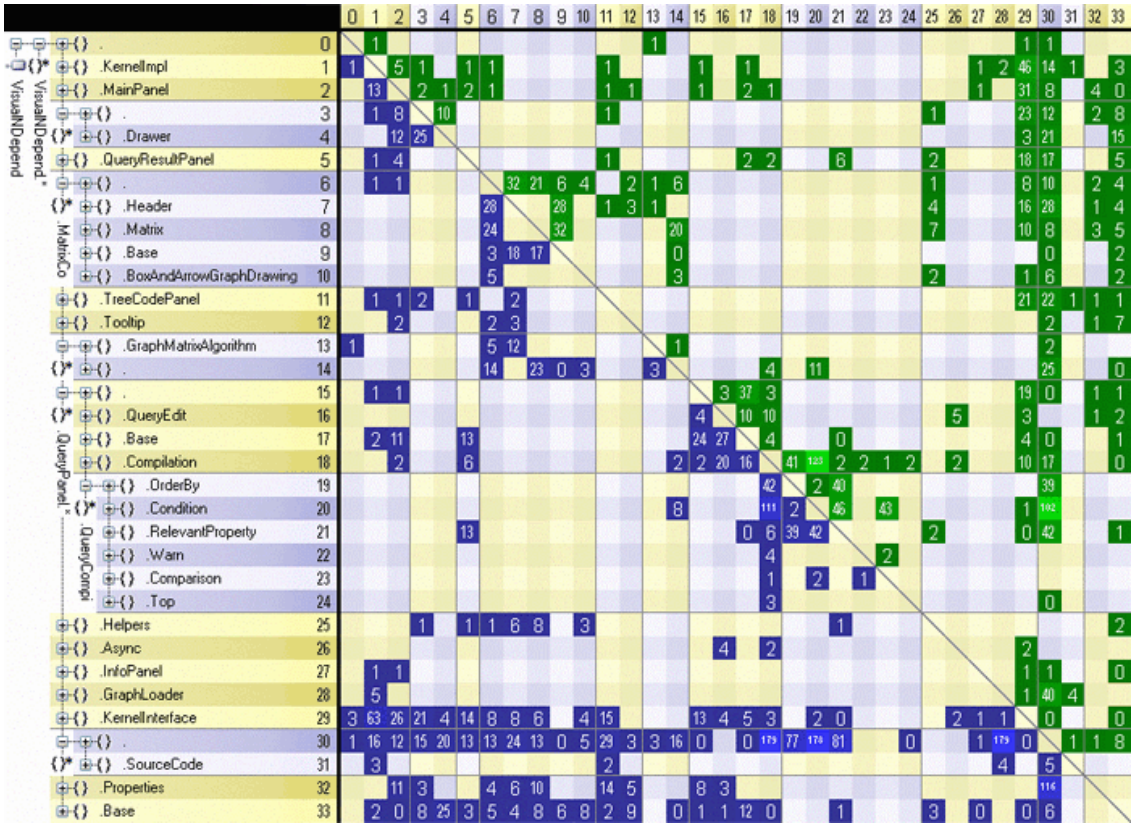
Of course, there are many forms of representation for components and their references.

Below is a graphic created with SonarJ specifically for packages:



Sonar

Below is a picture of NDepend:



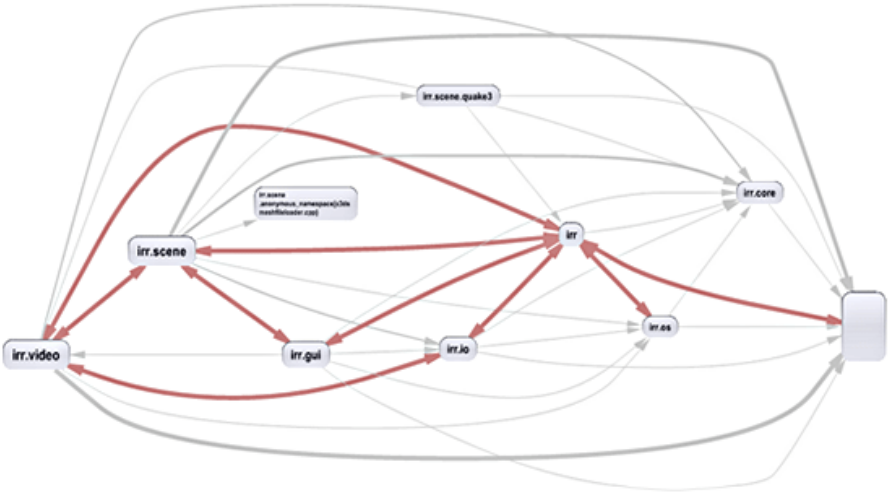
Dependency

This representation is almost equivalent to the second representation in terms of content.

The Dependencies of Packages mus be free of Cycles

Acyclic Dependencies Principle (ADP)

According to the principles of JDepend, all packages must be cyclic free. Experience has shown that the changeability and evolvability of large software with cycles decreases dramatically.



cycles

The graph shown here contains many cycles. He would certainly have a rACD metric of 50%-100%.

Anmerkung

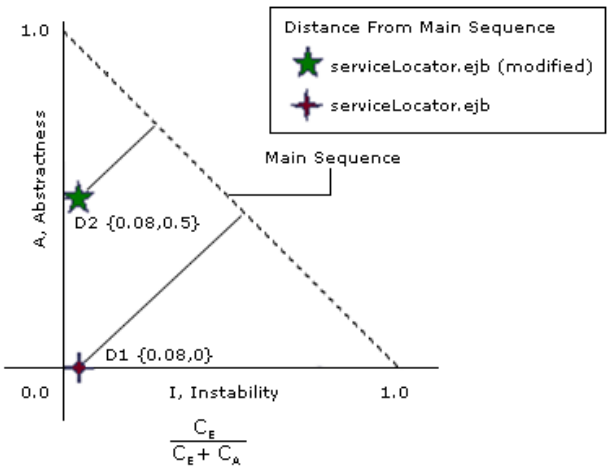
rACD (Relative mean component dependency)
The ACD metric is calculated by dividing the sum of all values by the number of nodes. This value is divided again by the number of nodes (or their square) and results in the rACD value.

In this case, it may be advisable not only to carry out a visual analysis, but also to include the freedom from cycles in the form of tests. Tools such as Macker (http://innig.net/macker/) provide Ant Tasks to enforce architecture enforcements.

A) Unstable packages depend on more stable packages / Stable Dependencies Principle (SDP)

B) The more stable, the more abstract the packages or components / Stable Abstractions Principle (SAP)

Please refer to the JDepend metrics for background information on this topic.



JDepend

In these principles, it is therefore important to look at the Ca and Ce metrics and calculate the values for the packages from them:

- Stable Packages: Usually more abstract packets, i. e. A = AC/CC+AC is high.
- Unstable packages: Are usually completely concretely implemented packages, i. e. I = Ce / (Ce + Ca) is also high.

4.8 Produktivität

Die Pfadfinderregel beachten (rot)

The Scout Rule

besser hinterlassen / broken windows!

Bekämpfe Ursachen. nicht die Wirkung (rot)

Root Cause Analysis

Text will follow...

Komponentenorientierung (blau)

Think and create components

Text will follow...

Keine goldenen Wasserhähne (blau)

You Ain´t Gonna Need It (YAGNI)

Text will follow...

Prinzip der Kleinstmöglichen überraschung (gelb)

Principle of Least Astonishment

Text will follow...

Ein Versionskontrollsystem einsetzen (rot)

Use a Version Control System

Text will follow...

Issues und Fehler werden (öffentlich) Verwaltet (orange)

Issue & Defect Tracking

Text will follow...

Effektives Buildmanagement

Use a Buildmanagement Tool

Research and industry today agree that building management is necessary from just a few classes if a serious program is to be created and more than one developer is working on it. Documentation generation, testing, creating the binaries, etc. cannot be done manually any more and can no longer be done consistently in the IDE after two developers.

Tools like:

1. Ant
2. Maven
3. rake
4. Gradle
5. (or commercial tools)

are therefore mandatory.

Automatisierte Integrationstests (orange)

Use automatic integration tests

Text will follow...

Continuous Integration / Delivery (grün / blau)

Have a continous Integration+Delivery System

ToDo...

Lote aus, ob ein DSL-Einsatz sinnvoll sein kann

Check if DSL can be useful

The coupling of components can take place in many ways:

- with direct function call
- via a communication protocol (e. g. REST)
- Event based
- and much more

However, if developers themselves or modules are to use other components with parameterized parameters, the use of aDSL can also be useful.

There are many examples of this. LINQ is - like SQL itself - such a query language, which is embedded directly into the system or programming language (like C#).

With the increasing ease with which dynamic programming languages such as Ruby or other tools (such as those of IntelliJ) can be used to create or generate DSL, it's definitely worth taking a look at.

The consideration is: how does module A want to use or manipulate module B?

4.9 Management

Kill Distraction

Kill Distraction

As a qualified developer, you also have to think about possible distractions. Nowadays, concentrated work over a defined period of time is hardly possible any more. The multitude of e-mails and events on the social web demand constant distraction and often an immediate reaction. There are many literature and references on this topic on the Internet.

The book below is highly recommended. It is about working concentrated over a defined period of 25 minutes and eliminating all distractions during this period.

Pomodoro Technique Illustrated: **"Can You Focus - Really Focus - for 25 Minutes?"** Pragmatic Bookshelf, ISBN-13: 978-1934356500

Referenz: http://www.pomodorotechnique.com

Übung CCD-04: Google „Focus“

Read more about the topic on the given website or google. You may even be able to download an app for it.

Time needed: 10 Minuten

App: Pomodoro

Fast Launcher (Ford)

Fast Launcher (Ford)

From Neil Ford there is the hint to always have a fast app launcher at hand.

This certainly applies to both web links and applications.

Become a wizard in Shortcuts

Use Keys & Shortcuts

Some studies on workplace efficiency have shown that a continuous change or break between the mouse and keyboard significantly disrupts the workflow and costs time. Therefore, it is always worthwhile to learn the shortcuts of the current development environment by heart.

Exercise CCD-05: Shortcuts

Which IDE do you use? Which shortcuts do you use to do the following:

1. Format the code?
2. Launch the application?

Stick a label on your monitor to help you with these most important shortcuts.

Time needed : 15 Minuten

Multiple Screens or at least Virtual Desktops

Multiple Screens or at least Virtual Desktops

Similar to the previous example, there are also investigations that the visual switching between the applications costs quite a lot of time. It is not without reason that stock exchange traders have therefore for several dozens of years already begun to use various monitors to distribute the information meaningfully across the entire field of vision. A good 27 inch monitors are already available for little money.

However, if the money is missing, a virtual desktop is a must to be able to switch quickly between completely set up screens.

Become a Scripting Guru

Love Macros & Shell Scripts

Good developers are often lazy people. You try to automate repetitive work steps.

There are many tools for the automation of work steps:

- Build tools / CI/CD tools (e. g. Ant)
- UNIX-Shell & Power Shell
- Scripting languages

Of all these three areas, a developer should have sound knowledge. This does not only include the tools mentioned here under one and the shell of the operating system. Nowadays, scripting languages are just as important. Therefore, the developer should have at least one dynamic scripting language, such as: Python of Ruby.

Love UNIX

Use Unix

Many professional developers work under UNIX. Why are they doing this?

UNIX is much better suited to the work of developers in many respects. Many tools - such as Git - are implemented more efficiently for UNIX than other operating systems. Of course, there are many programs with graphical gimmicks that have been ported to Windows. However, there are professionals who are much faster and more efficient when using Git or Mercurial, for example, if the corresponding commands are executed with the keyboard.

Use the best Editor and IDE

Perfect Editor, Best IDE

Every developer's goal must also be to be as efficient as possible using the best development environment. Here you should ask yourself whether the right tool is used for the current problem. Often more than 1000 € is spent on hardware, but a few Euros for the appropriate software are sometimes no longer available.

In the area of software development with Java, the world with three vendors (Eclipse, NetBeans, IntelliJ) is still quite manageable.

However, it looks different when a text with a certain format (

- . txt, *. xml, *. json, *. cvs

) has to be edited. There are a lot of editors here, which are wonderfully extensible and adapt to the context of the text (UltraEdit, Jedit, notepad++, emacs, vim, etc.).

Know Libraries / Don't Reinvent the Wheel

Know Libraries / Don't Reinvent the Wheel

The title of this tip already says it. As developers, we must not constantly reinvent the wheel. This also includes knowing the most important one or two dozen libraries and using them in the project. Of course, it has to be clarified whether a new library can be added to the project. However, if you have good dependency management, this should not be a problem.

It is exactly this topic that we will address in one of the last sessions. This is a quick look at the most important libraries in the Java environment and why this should be part of your own portfolio (e. g. the old Guava).

Iterative Entwicklung z.B. Scrum

Develop Iterative; e.g. SCRUM

For most of today's projects, iterative development with the appropriate agile project management tools is mandatory.

Unfortunately, however, it is still far too often seen today that projects are carried out without the appropriate tools or visions in terms of agility and iterative approach. What remains is often a management via Excel or web wiki.

Software development is in any case a learning process in which feedback is the most important element to get as close as possible to the final goal.

Lesen, Lesen, Lesen & Weiterbilden

Cultivate your Knowledge

If you have been working in development for a long time, it is often easy to make yourself comfortable in the given work area. You know your area, sit in a (really) safe position and see no need to look beyond the edge of your plate.

Unfortunately, this view is becoming increasingly dangerous and insecure. In a time when everything is getting faster and faster, technology is advancing and people are competing with jobs from all over the world (globalization), the pressure on each individual increases. Why can't a qualified person anywhere else in the world fill his or her own job competently?

In order to survive in this competitive environment, every developer should cultivate many of these sources and pay attention to continuing education:

1. Information through web content (RSS, hackernews, infoQ, dzone, etc.)
2. Information from trade journals
3. Information from good books, eBooks
4. Information from conferences

Visit Events and Learn from the Masters

Visit Events and Learn from the Masters

This point refers to point four of the previous list. No one is perfect in all areas. But learning is more efficient the more you learn from or engage in dialogue with experts. Therefore, the attendance of some specialist conferences and regional groups (JUG) is obligatory.

Share your Knowledge

Share your Knowledge

The open source principle has also played a pioneering role here: in reasonable companies without competitive pressure, it should be a matter of course to pass on special or learned knowledge. And this does not only add value for the company, but also for the learner by enriching their knowledge. Even the process itself, in which experience or knowledge is passed on, allows only a deep penetration of the material and also helps to qualify oneself as a knowledge mediator or trainer.

An important prerequisite for this is an open corporate culture without any power struggles. The latter suffocates every attempt to grow together and achieve common goals.

4.10 Zusammenfassung CCD

Title: Summary CCD

- The term Technical Debt and the consequences resulting from it were introduced.
- You have learned about coding basics (delete code that no longer has authorization, avoid Magic Numbers) and the subcategories:
  - Coding Source (local declaration, explanatory variables, code nesting and separation of multi-threaded code)
  - Coding Conditionals (capsules of complex logic operations, avoid negative logic operations and reveal constraints in one place)
  - Coding Principles (No repetitions alias code copy, simplicity as a guiding principle, ensuring implicit sequences)
- Furthermore, you have learned about various procedures for testing code quality, such as code convention checking, automated unit tests, code coverage analysis, the Test First approach, the use of mock tools and static code analysis (metrics).
- You now know what principles of clean code development exist in relation to software architecture, class design and packages such as task separation and dependency injection, visualize dependency dependencies of packages.
- It was explained which basic principles you should observe with regard to productivity (e. g. no golden taps, effective build management).
- You have learned about ways to work better and more efficiently than software developers (e. g. eliminating distractions, Unix is important, training).

- You now know that it is important to have a good eye for quality at all levels: from source code to architecture or working methods.

4.11 Wissenüberprüfung CCD

Übung CCD-06: Clean Code Development

1. What is the meaning of Clean Code?
2. What is the Technical Debt?
3. Please read www.clean-code-developer.de
4. Create your personal checklist for Clean Code and attach it to the wall at home via your computer. ;-)

Time needed: 60 min

4.12 Literatur CCD

Robert C. Martin:

"Clean Code"

Prentice Hall

Andrew Hunt / Dave Thomas / Ward Cunningham:

"The Pragmatic Programmer"

Addison Wesley

Neal Ford:

"The Productive Programmer"

O'Reilly

Kent Beck:

"Implementation Patterns"

Addison Wesley

Martin Fowlers Bücher über:

Patterns, E-Patterns, Refactoring, Continous Integration, Beyond Architecture, etc.

Joshua Bloch:

"Effective Java"

Prentice Hall

Buschmann, Henney, Schmidt:

"Pattern-Oriented Software Architecture"

Wiley

Weitere Referenzen:

www.clean-code-developer.de

(An deren Entstehung und Verbreitung der Autor beteiligt war)

12 Things Every Programmer Should Know

http://www.javacodegeeks.com

http://www.javacodegeeks.com/2010/12/things-every-programmer-should-know.html

97 Things Every Programmer Should Know

http://programmer.97things.oreilly.com/wiki/index.php/Contributions_Appearing_in_the_Book

O'Reilly

Weitere Weblinks:

- Artikel aus PHP Magazin
- 100 Tools for Developers
- Reducing Incidental Complexity in Our Code and in Our Teaching
- How to be a great Software Developer