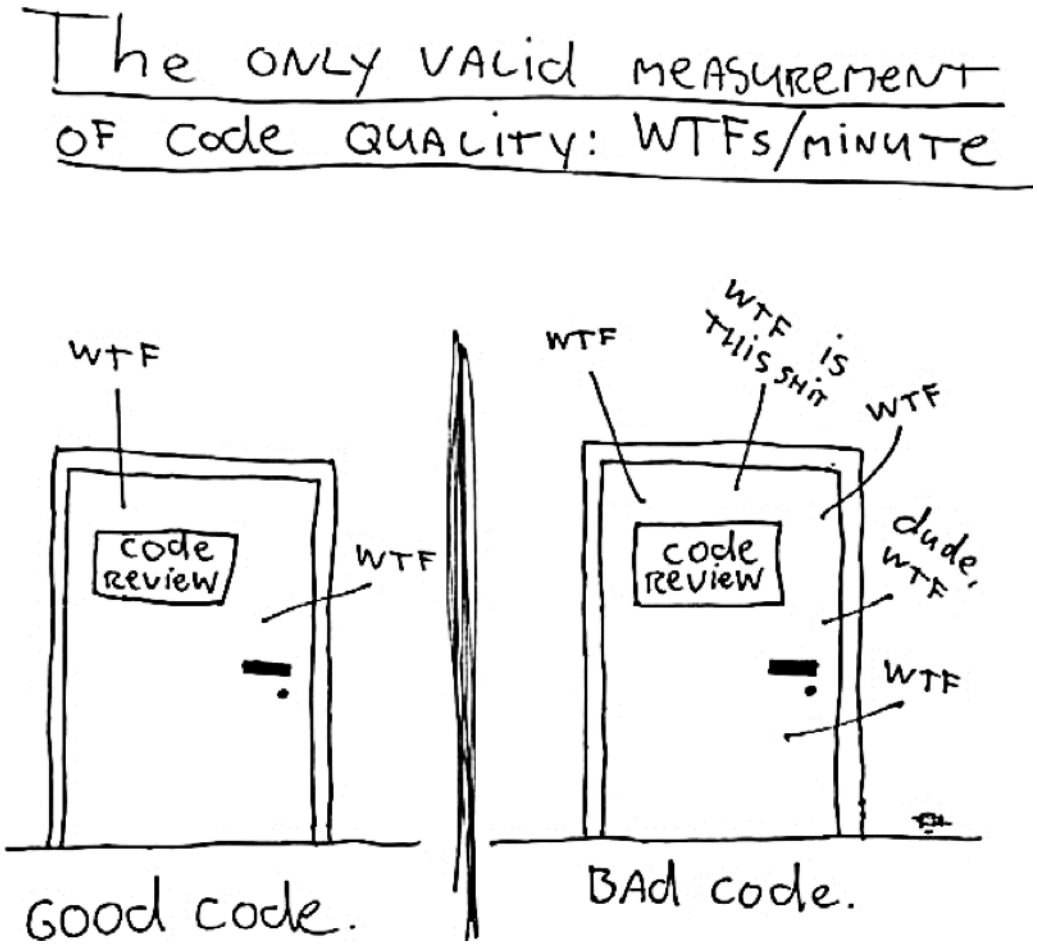
Software Construction

Florian Thiévent

3. Semester (HS 2018)



If you use this documentation for an exam, you should offer a beer to the authors!

Inhaltsverzeichnis

1 Software Construction

Problem Definition	**Detail Plan**	Team Management
Anforderungen spezifizieren	**Coding and Debugging**	Maintenance (Betrieb)
Planung der Anwendung	**Unit Testing**	**Integration**
SoftwareArchitektur		**Integration Testing**
		System Testing

1.1 Definition

Software construction is a fundamental act oft software engineering: the construction of working, **meaningful software** through a combination of **coding, validation and testing** by a programmer.

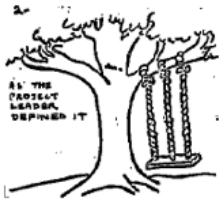
1.2 Low Level SWC

- Testing definieren
- Design und Klassen schreiben
- Erstellen und Naming der Variablen und Konstanten
- Unit Testing, Integration Testing und Debugging
- Review von anderem Code von Teammitgliedern
- Integration von Software
- Formatierung von Code und Kommentaren

1.3 Wieso ist SWC wichtig

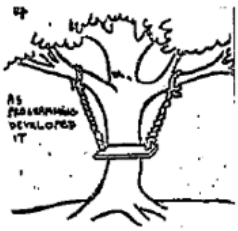
Construction is the only activity that's guaranteed to be done

As Management requested it.

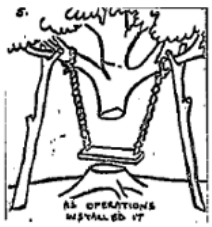


As the Project Leader defined it.

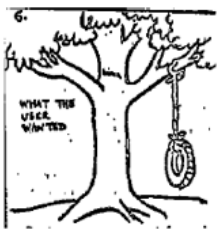
As Systems designed it.



As Programming developed it.



As Operations installed it.



What the user wanted.

1.4 Software Qualität

- Reliability: Zuverlässige Software welche fehlerfrei betrieben werden kann
- Reusability: Code Teile auch für andere Projekte nutzbar machen
- Extendibility: Erweiterbarkeit soll einfach möglich sein
- Understandability: Code soll verständlich für andere sein
- Efficiency: Geschwindigkeit
- Usability: Software einfach nutzen, auf das Zielpublikum ausgerichtet
- Portability: Einfach in eine andere Umgebung zügeln
- Functionality: Software soll funktional sein

Definition ISO: The totality of features and characteristics of a product or service that bear on its ability stated or implied needs
Definition IEEE: The degree to which a system, component, or process meets specified requirements.

1.5 Ziele

Die Software muss den Anforderungen des Kunden entsprechen. **"Good enough Software"not excellent software!**

1.6 Extreme Programming

1. Der Kunde/Auftraggeber ist immer verfügbar
2. Code wird gemäss vereinbarten Standards programmiert
3. Zuerst die Tests programmieren, dann den eigentlichen Code
4. Der produktive Code wird immer zu zweit programmiert
5. Nur ein Programmierer-Paar darf gleichzeitig Code integrieren
6. Integriere häufig
7. Jeder hat auf den gesamten Code Zugriff
8. Optimiere so spät wie möglich
9. Keine überstunden
10. Das Team folgt gemeinsamen Code-Richtlinien, so dass es aussieht, als wenn der Code von einer einzigen Person geschrieben worden wäre
11. XP Projekte werden in sehr kurzen Abständen released (von täglich bis zu maximal alle 3-4 Wochen)

2 Version Control System

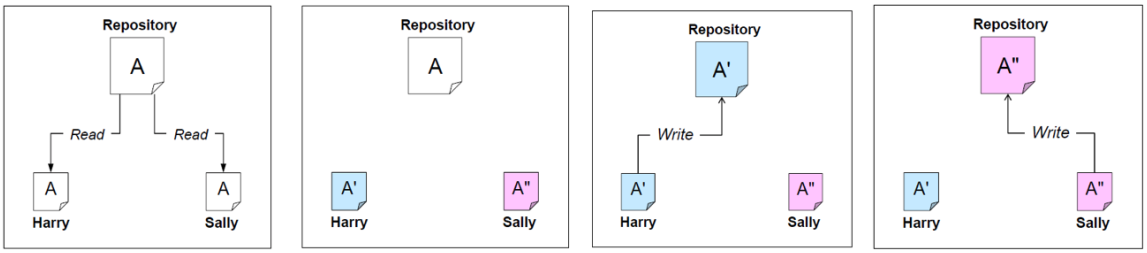
Es geht grundsätzlich um sich ständig ändernde Artefakte welche verwaltet werden müssen. Jede änderung soll eindeutig nachvollziehbar sein.

2.1 Motivation für eine VCS

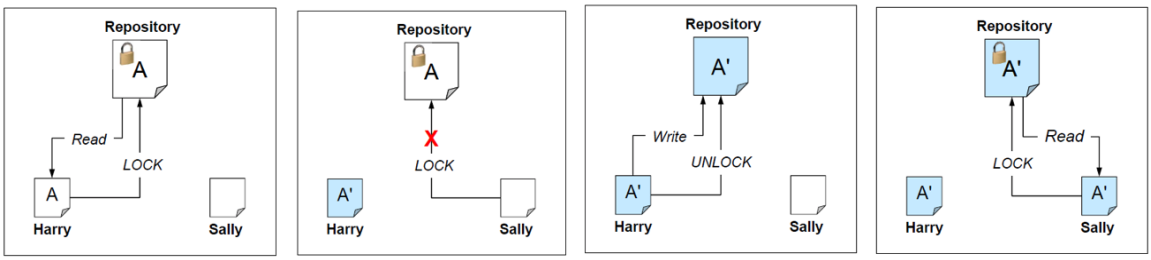
Ohne Versionsmanagement sieht der Alltag von Entwicklern so aus:

- Bugs die behoben wurden tauchen plötzlich wieder auf
- Dateien gehen verloren
- Frühere Releases der Software können nicht mehr erstellt werden
- Dateien werden äuf mysteriöse Art und Weise" verändert
- Gleicher oder ähnlicher Code existiert mehrfach in verschiedenen Projekten
- Zwei Entwickler ändern dieselbe Datei gleichzeitig ohne es zu merken

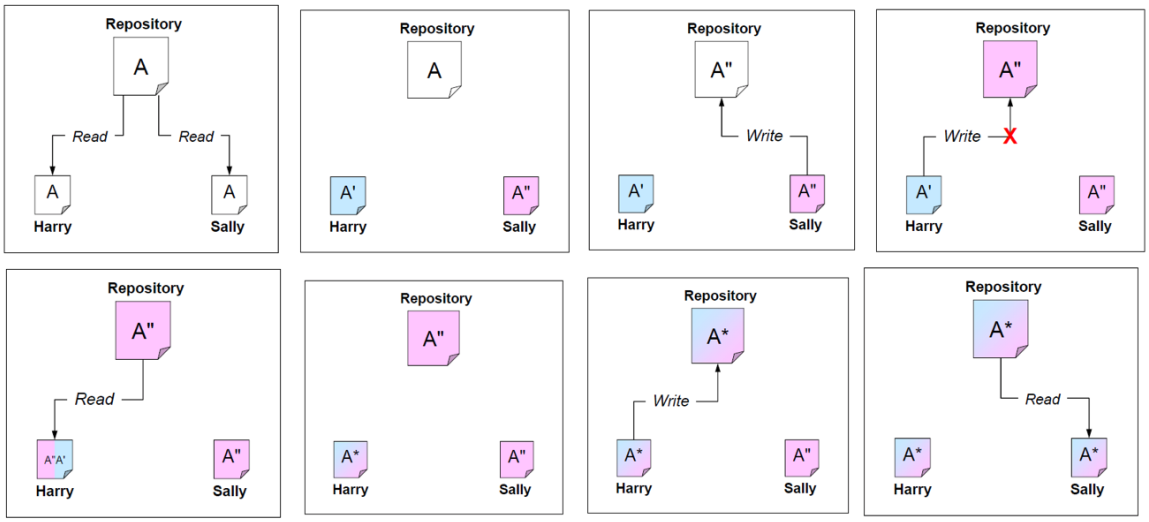
2.2 Problem of Filesharing



2.3 Lock-Modify-Unlock Solution



2.4 Copy-Modify-Merge Solution



2.5 Grundbegriffe Versions- und Release-Management

2.5.1 Version

Ein Zustand eines Konfigurationselementes mit einem klar definierten Funktionsumfang.

2.5.2 Release

Bezeichnet eine veröffentlichte Version eines Konfigurationselementes. Oft wird ein Release auch ausserhalb der Software-Entwicklungsorganisation zusammengestellt.

2.5.3 Major

Die Idee, das Major-Versionen (mindestens teilweise) inkompatibel sind

2.5.4 Minor

änderungen dieser Nummer sind rückwärts kompatibel.

2.5.5 Patch

Bug-Fixes, änderungen sind vorwärts- und rückwärts kompatibel.

2.5.6 Build

Diese Nummer gibt eine Neukompilierung von Sourcecode an, z.B. aufgrund von Prozessor-, Plattform- oder Compileränderungen.

2.5.7 Revision

Kleine änderung an einer Version, welche Fehler behebt, jedoch keine Einfluss auf den Funktionsumfang haben.

2.5.8 Variante

Eine Variation einer Version welche entwickelt wurde um z.B. auf einer anderen Hardware oder unter einem anderen Betriebssystem zu laufen. Oder auch um für verschiedene Benutzergruppen feine Anpassungen vorzunehmen. Beispiele: Anpassungen für Tablets, Sehbehinderte, Touchscreens.

2.6 Grundbegriffe VCS

2.6.1 Repository

Eine Datenbank in welcher Projektdateien gespeichert werden. Ein Repository vergisst nichts, d.h. es ist nicht möglich eine Datei zu überschreiben. Vielmehr wir einfach eine neue Version der Datei gespeichert, die alte bleibt weiterhin im Repository und es kann auch weiterhin zugegriffen werden.

2.6.2 Working Copy

Eine *lokale Kopie* aller relevanten Projektdateien. Der Entwickler arbeitet immer auf dieser lokalen Kopie. Es wird also *nie* direkt auf den Dateien im Repository gearbeitet.

2.6.3 Checkout / Clone

Ist die Bezeichnung für einen Vorgang wenn eine Working Copy vom Repository bezogen wird. Dies ist eine reine Leseoperation auf dem Repository. Dabei wird auf der Entwicklermaschine eine neue Working Copy angelegt.

2.6.4 Commit / Push

Bezeichnet den Vorgang wenn eine Datei oder ein ganzes Set an Dateien (neu oder geändert) *mit einer Beschreibung* ins Repository gespeichert wird. Man spricht auch davon diese unter Versionskontrolle zu stellen. Dies ist eine Schreiboperation auf dem Repository die *atomar* erfolgt, d.h. pro Commit werden entweder alle oder gar keine Dateien ins Repository gespeichert. Damit ist sichergestellt, das das Set an Dateien konsistent ins Repository gespeichert wird. Es ist nicht möglich, dass durch zwei gleichzeitige Commits die Dateien durcheinandergebracht werden.

2.6.5 Update / Fetch / Pull

Bezeichnet den Vorgang wenn Dateien aus dem Repository mit der eigenen Working Copy abgeglichen werden. Indem andere Entwickler ihre Arbeit committen erhält das Repository neue Versionen welche auf den Working Copies der anderen Entwickler noch nicht vorhanden sind. Der Abgleich findet auf der Working Copy statt, für das Repository ist ein Updatevorgang somit eine reine Leseoperation.

2.6.6 Revision, Version

Jeder Commit verändert den Inhalt des Repositories und erzeugt somit eine neue Version oder Revision des Repositories die eindeutig identifizierbar sein muss. Dies ist notwendig um später wieder auf einen bestimmten Stand der Arbeit zurückzukehren zu können. Manche Repositories verwenden mehrstellige Versionsnummern (z.B. CVS), andere nummerieren die Commits einfach durch (z.B. Subversion) oder vergeben einen Hash (z.B. Git) als Identifikation.

2.6.7 Entwicklungsverlauf (Baseline, Codeline, Line of Development

Dies sind Bezeichnungen für eine Menge von relevanten Projektdateien die zusammen gehören und die miteinander weiterentwickelt werden. Eine Working Copy enthält all diese Dateien einmal. Ein Entwicklungsverlauf bezeichnet aber die gesamte Historie dieser Dateien während des Entwicklungsverlaufes. D.h. er enthält alle relevanten Projektdateien in allen Versionen für einen bestimmten Entwicklungsverlauf. Ein Entwicklungsverlauf ist eindeutig über seinen Revisionsverlauf gekennzeichnet. Der Hauptentwicklungsverlauf wird in Subversion oder CVS auch einfach *trunk* genannt und in Git heisst er üblicherweise *master*.

2.6.8 Branch

So wird eine Verzweigung von Entwicklungsverläufen genannt. Branches sind selber auch wieder Entwicklungsverläufe. Sie enthalten selber eine eigene, von anderen Entwicklungsverläufen unabhängigen Historie. Je nach Versionsverwaltungssystem werden Branches unterschiedlich eingesetzt. Ein mögliches Szenario ist z.B. wenn die Entwicklung an einer Version 2 weiterläuft (Hauptentwicklungsverlauf) und gleichzeitig die alte Version 1.x noch weiter gewartet werde soll (Branch).

2.6.9 Merging

Bezeichnet den Vorgang zwei Entwicklungsverläufe zu vereinen. Dazu müssen Dateien in unterschiedlichen Versionen zusammengeführt werden. Dies ist in der Regel ein manueller Vorgang der nur bedingt automatisiert werden kann. Wie beim Branching wird auch das Merging unterschiedlich unterstützt von den verschiedenen Versionskontrollsystemen.

2.6.10 Tag, Label

Identifiziert bzw. markiert eine bestimmte Revision eines Entwicklungsverlaufes oder Configuration Items und fügt ihm noch zusätzliche Informationen hinzu (z.B. gibt ihm einen bestimmten Namen oder Bemerkung wie beispielsweise: „Release für Demo anlässlich Veranstaltung xxx“). Tags sind konzeptionell keine Entwicklungsverläufe sondern nur eine Momentaufnahme (Snapshot) eines solchen.

2.7 Configuration Items

Unter VCS	Nicht im VCS	Grenzfall
Sourcen	Testprotokolle (generiert)	DB
Dokumentation	Testreports (generiert)	Video
Kommunikation	Persönliche Konfig (IDE)	Audio
Tests	Javadoc → html	
Geteilte Config	DIE	
Video / Audiofiles (kleine)	Compilate	
	(.exe, .class, .jar	
	Files aus anderen Projekten	
	(Libraries, Vorlagen)	

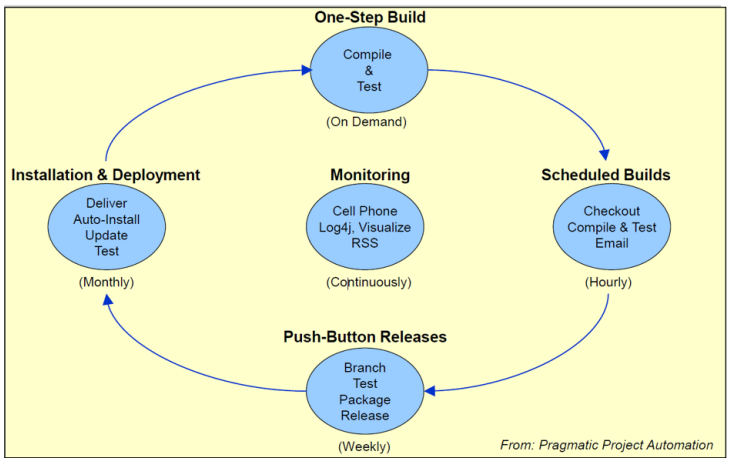
3 Build - Automation

Die Komponenten einer Software werden per Knopfdruck erstellt. Macht meist mehrer Dinge auf einmal (Code compilieren, Tests durchlaufen, Checkout, Kopie auf Server, E-Mail). Der Buildvorgang kann auch automatisch gestartet werden.

3.1 Wieso wird dies benötigt, Probleme

- Die Entwickler können die Applikation nicht zuverlässig lokal erstellen
- Fehlen einer konsistenten Versionierung
- Unit Testing ist nicht konsistent
- Status des Build ist nicht bekannt
- Abhängigkeiten von Komponenten ist nicht bekannt
- Entwicklung ist nicht transparent
- Handarbeit ist fehleranfällig
- Wiederholende Arbeit ist langweilig
- Automation ist auch Dokumentation

3.2 Build Prozess



3.3 Benötigte Komponenten

- Build Server
- Source Code Control Server
- Prozesse
- Tools
- Entwickler Verantwortung (Code checked before end of day, buildable, passed unit test)

3.4 CRISP

- Complete (Tests erfolgreich, alle Komponenten enthalten)
- Repeatable (Wiederherstellung von alten Builds, keine Binary committen, bugfixing)
- Informative (Code Dokumentation, Infos für Merging)
- Schedulable (planbar, für Akronym)
- Portable (definierte Portabilität, auch auf anderen System funktional)

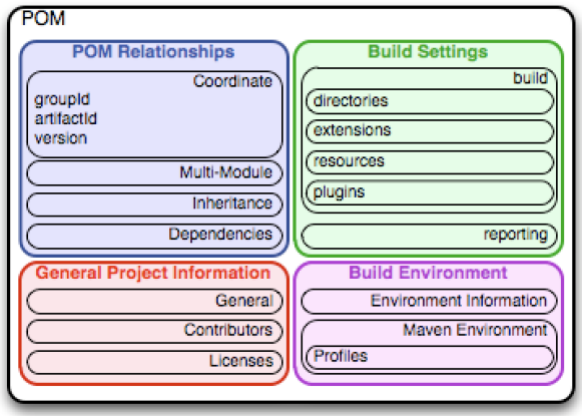
3.5 Maven

Maven ist ein Build Automation Tool und der de-facto Standard in Java Projekten. Maven setzt auf deklarative Konfiguration und folgten dem Ansatz Konvention vor Konfiguration. **Put the source in the correct directory and Maven will take care of the rest**.

3.5.1 Project Model (POM)

Enthält die Informationen für den Output.

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>ch.fhnw.imvs</groupId>
    <artifactId>myapp</artifactId>
    <version>1.0-SNAPSHOT</version>
    <packaging>jar</packaging>
    <name>My First App</name>
    <url>http://maven.apache.org</url>
</project>
```

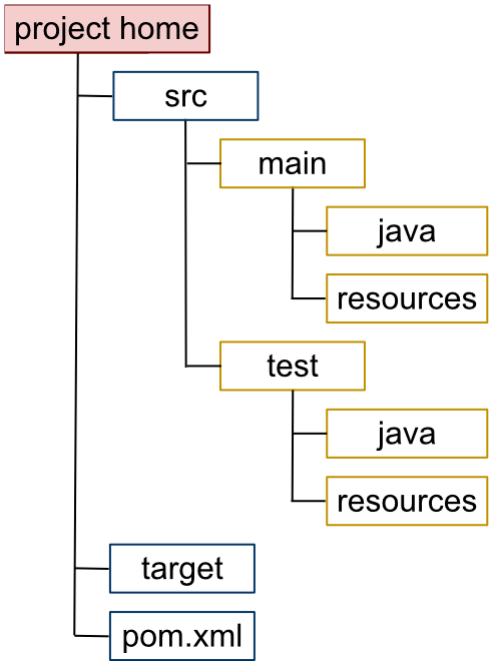


Pflichtfelder: groupeId, artifactId, version und packaging

3.5.2 Standard Build Prozess

Begriff	Beschreibung
validate	check if project is valid and all necessary information is available
process-resources	convert and filter resource files
compile	compile source code
test-compile	compile test code
test	Execute tests
package	package the artifact
integration-test	execute integration tests
install	copy artifact into the local repository
deploy	publish artifact in the remote repository

3.5.3 Directory Structure



- Java-Source (to be delivered)
- Non-Source Files (properties, icons . . .)
- Test Classes
- Resources for testing only
- generated files

3.5.4 Fallbeispiel POM

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.x
    <modelVersion>4.0.0</modelVersion>
    <groupId>ch.fhnw.swc</groupId>
    <artifactId>MRS08</artifactId>
    <version>0.1-SNAPSHOT</version>

    <properties>
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    </properties>

    <build>
        <plugins>
            <plugin>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.3</version>
                <configuration>
                    <source>1.8</source>
                    <target>1.8</target>
                </configuration>
            </plugin>
            <plugin>
                <artifactId>maven-project-info-reports-plugin</artifactId>
                <version>2.8</version>
            </plugin>
            <plugin>
                <groupId>org.codehaus.mojo</groupId>
                <artifactId>jaxws-maven-plugin</artifactId>
                <version>1.12</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-javadoc-plugin</artifactId>
                <version>2.10.3</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-surefire-plugin</artifactId>
                <version>2.18.1</version>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-checkstyle-plugin</artifactId>
                <version>2.16</version>
                <configuration>
                    <configLocation>src/main/config/swc_checks.xml</configLocation>
                </configuration>
            </plugin>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-assembly-plugin</artifactId>
                <version>2.2.1</version>
                <configuration>
                    <archive>
                        <manifest>
```

```
                    <mainClass>ch.fhnw.edu.rental.MovieRentalApplication</mainClass>
                </manifest>
            </archive>
            <descriptorRefs>
                <descriptorRef>jar-with-dependencies</descriptorRef>
            </descriptorRefs>
        </configuration>
        <executions>
            <execution>
                <phase>package</phase>
                <goals>
                    <goal>assembly</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
    <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-javadoc-plugin</artifactId>
        <version>2.10.3</version>
        <configuration>
            <stylesheet>maven</stylesheet>
            <doctitle>Movie Rental System - Software Construction Lab</doctitle>
            <footer>Copyright 2015 Christoph Denzler, Martin Kropp - IMVS, FHNW</footer>
        </configuration>
        <executions>
            <execution>
                <id>generate-javadocs</id>
                <phase>generate-sources</phase>
                <goals>
                    <goal>javadoc</goal>
                </goals>
            </execution>
            <execution>
                <id>attach-javadocs</id>
                <phase>package</phase>
                <goals>
                    <goal>jar</goal>
                </goals>
            </execution>
        </executions>
    </plugin>
</plugins>
</build>
<dependencies>
    <dependency>
        <groupId>org.dbunit</groupId>
        <artifactId>dbunit</artifactId>
        <version>2.5.1</version>
    </dependency>
    <dependency>
        <groupId>org.hsqldb</groupId>
        <artifactId>hsqldb</artifactId>
        <version>2.3.2</version>
        <scope>runtime</scope>
    </dependency>
    <dependency>
```

```
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.11</version>
            <scope>test</scope>
        </dependency>
        <dependency>
            <groupId>org.slf4j</groupId>
            <artifactId>slf4j-nop</artifactId>
            <version>1.7.12</version>
            <scope>runtime</scope>
        </dependency>
        <dependency>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-javadoc-plugin</artifactId>
            <version>2.10.3</version>
            <exclusions>
                <exclusion>
                    <groupId>org.slf4j</groupId>
                    <artifactId>slf4j-jdk14</artifactId>
                </exclusion>
            </exclusions>
        </dependency>
        <dependency>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-checkstyle-plugin</artifactId>
            <version>2.16</version>
        </dependency>
    </dependencies>
</project>
```

4 Clean Code

Clean code is **simple** and direct. Clean code reads like well-written prose. Clean code never obscures the designer's intent but rather is full of **crisp** abstractions and **straightforward** lines of control.

4.1 Grundsätze

- Name der Variabel, Methode oder Klasse sollte den Zweck beinhalten.
- Variabeln sollten möglichst nahe zum Aufrufpunkt deklariert werden
- Callers (Aufrufen) sollten oberhalb der Callees (Aufgerufene) deklaiert sein
- Lines short (80 Characters)
- Define Tabs vs. Spaces and Line Wrapping/Breaking Rules
- Use the Code Convention of the Programming Language
- Checkstyle integrieren
- Class Documentation (performance, memory consumption, persistence)
- Interface Doc (Explain what is done)
- Method Doc (precondition, postcondition, invariant) =¿ bei Kommentaren Subjekt auslassen
- Use Javadoc
- Funktionen sollten nur eine Sache Machen
- Negative Konditionen sollten verhindert warden

4.2 Wieso CleanCode

- 80% of the lifetime cost of a piece of software goes to maintenance.
- Code is written once, but read all other times
- Hardly any software is maintained for its whole life by the original author.
- Clean code improves the readability of the software, allowing engineers to understand new code more quickly and thoroughly.

4.3 Konzepte

4.3.1 Vertical Openness

Uses more vertical space (new line, brackets, etc) this improves readability, quick to see which code parts are not together.

4.3.2 Vertical density

Opennness separates concepts. Density implies association. Results in Compact Code.

4.3.3 Vertical Distance and Ordering

Concepts that are closely related should be vertically close to each other.

- Variables should be declared as close to their usage as possible.
- Instance variables should be declared at the top of the class.
- Dependent functions: callers should be above callees.

4.3.4 Horizontal Openness and Density

- Keep lines short.

  complete line should be displayable without scrolling (eg ¡120 characters per line)

- Don't try to horizontally align lists of assignments

  It draws attention to the wrong thing and can be misleading, e.g., encouraging the reader to read down a column.

- Always indent scopes (classes, methods, blocks).
- Avoid using tabs, replace them with spaces
- Define line wrapping, line breaking rules, e.g.:

  After comma

  Before operator, i.e. operators come first on new line

- Define clear spacing rules, e.g.

  Put spaces around = to accentuate the distinction between the LHS and RHS.

  Don't put spaces between method names and parenthesis, or parenthesis and parameter lists - they're closely related, so should be close.

  Use spaces to accentuate operator precedence, e.g., no space between unary operators and their operands, space between binary operators and their operands.

4.3.5 Team Rules

- Every team should agree on a coding standard
- Beware of code formatting standards getting religious.
- If the language you're using has a code convention (like Java's), use it!

5 Continous Integration

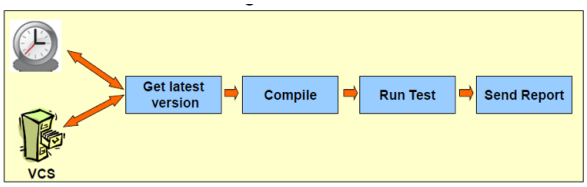
Integration ist verschiedene Module zusammen zum Laufen zu kriegen. Die Module müssen zusammen kompiliert, gestartet, getestet und deployed werden können. **CI reduziert Risiken und deckt Fehler früh auf.**

5.0.1 Fehlerhafte Integration

- Integration Server kann den Build nicht durchführen
- Geteilte Inhalte funktionieren nicht in allen System
- Fehler bei Unit Tests
- Code Qualität failed
- Deployment failed

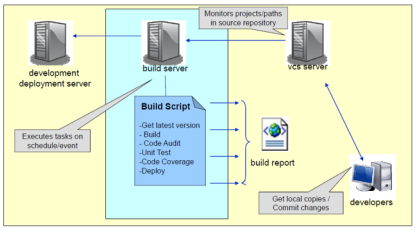
5.0.2 Arbeiten mit Continous Integration

- Nur ein Source Repository
- Automatisierter Build
- Build testet sich selbst
- Jeder commited jeden Tag
- Der Build sollte schnell gehalten werden
- Testen auf einem Klone der Produktion
- Einfach das letzte executable zu erhalten
- Jeder sieht was sich ändert
- Automatisiertes Deployment



5.1 Prerequisites of Continuous Integration

- VCS Server
- Build Server
- Deployment Server
- Automation tools
- CI tools



5.2 Jenkins

Jenkins ist ein CI Server welcher building, unit tests, code coverage und weitere Punkte zur Verfügung stellt. Gibt sofort einen Status über den Build zurück und hat ein Dashboard für die Integration von verschiedenen Projekten.

5.3 Jenkins Componenten und Add Ons

Components	Add Ons
CI Server (Monitors the VCS and Execute Build Script)	Unit Testing
Management Console	Test Code Coverage
Dashboard	Analysis
Email Notification	Ant
	Google Calender, and, and, and

6 Unit Testing

Testing von Softare ist extrem wichtig. Zum einen können Fehler aufgedeckt und korrigiert werden. Somit können Kosten eingespart werden. Falls es um Spezialsoftware geht können sogar Menschenleben oder Umweltkatastrophen entstehen bzw. verhindert werden.

- Validation: Did you build the right thing?
- Verification: Did you build it right?

6.1 Fault, Error and Failure

6.1.1 Fault

A static defect in the software = the actual mistake or bug in the code.

6.1.2 Software Error

An incorrect internal state that is the manifestation of some fault.

6.1.3 Software Failure

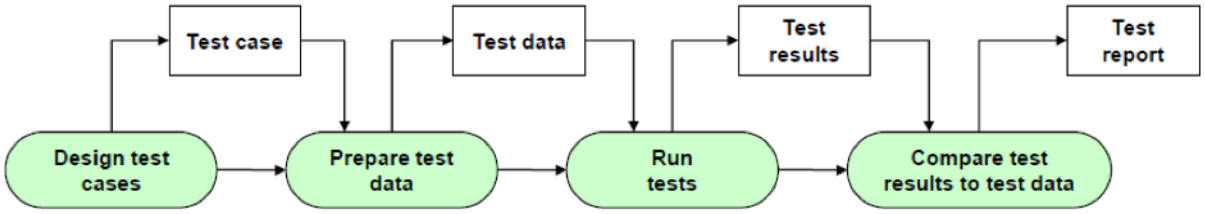
External incorrect behavior with respect to the requirements or other description of the expected behavior.

6.2 Ablauf

- Wird auf jede unit (Klasse, manchmal Methoden) durchgeführt
- In isolation
- Unit Test baut eine eigene Testumgebung auf
- Checkt das Resultat mit den erwarten Werten
- Test Spezifizierung
- Test Implementierung
- Setup und Cleanup definieren

6.3 Schlechte Ausreden gegen Unit Testing

- It takes too much time to write the tests
- It takes too long to run the tests
- It's not my job to test my codeI
- don't really know how the code is supposed to behave so I can't test it
- But it compiles
- I'm being paid to write code, not to write tests
- I feel guilty about putting testers and QA staff out of work
- My company won't let me run unit tests on the live system



6.4 JUnit

Standard in Java

6.4.1 Gute JUnit Tests

- Automatic (Invoking the tests and checking the results)
- Thorough (test everything that is likely to break)
- Repeatable (ablte to run over and over again)
- Independant (no test relies on another test)
- Professional (use same standard as for production code)
- Do not test compiler, setters/getters

6.4.2 Equivalent Klassen

Die Eingabewerte von Tests müssen in verschiedene Equivalent Klassen eingegrenzt werden. In diesen Klassen soll der Grenzwert erkannt werden und zudem auch um die Grenzwerte getestet werden (off-by-one-errors)

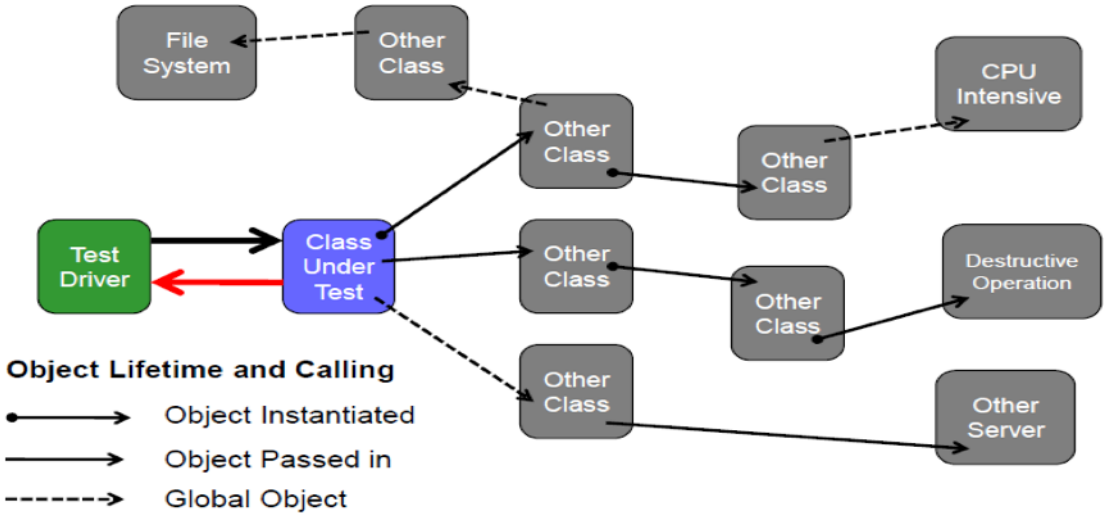
6.4.3 Right BICEP

- Right –Sind die Resultate korrekt
- B –Sind alle boundary (Grenzwerte) korrekt
- I –Können auch Inverse Beziehungen getestet werden
- C –Cross check
- E –Können Errors erzwungen werden
- P –Performance sind wie erwartet (was passiert wenn Input verdoppelt, verdreifacht wird)

6.4.4 Grundbegriffe

- Class under Test (CUT)
- Method under Test (MUT)
- Test Case (Spezifische daten welche für Test ausgewählt wurden @Before, @After,@Test)
- Test Suite (ein Set von verschiedenen Test Cases)
- Test Fixture (Die ganze Test Klasse)

6.5 Testing in Isolation



Some things are difficult to test in isolation

- Configuration
- Database Access
- Network Access

In general: How to test a class that depends on other components?

6.5.1 Test Doubles

A *Test Double* is any object or component that is installed in place of the real component for the express purpose of running a test.

6.5.2 Test Doubles in Unit Testing

- **Dummy objects** are passed around but never actually used. Usually they are just used to fill parameter lists.
- **Stubs** are minimal implementations of interfaces or base classes. Methods returning void will typically contain no implementation at all, while methods returning values will typically return hard-coded values.
- **Spies** similar to a stub, but a spy will also record which members were invoked so that unit tests can verify that members were invoked as expected.
- **Fakes** contain more complex implementations, typically handling interactions between different members of the type it's inheriting.
- **Mocks objects** pre-programmed with expectations about the methods that will be invoked. The expected arguments for a call and the resulting return values or exceptions

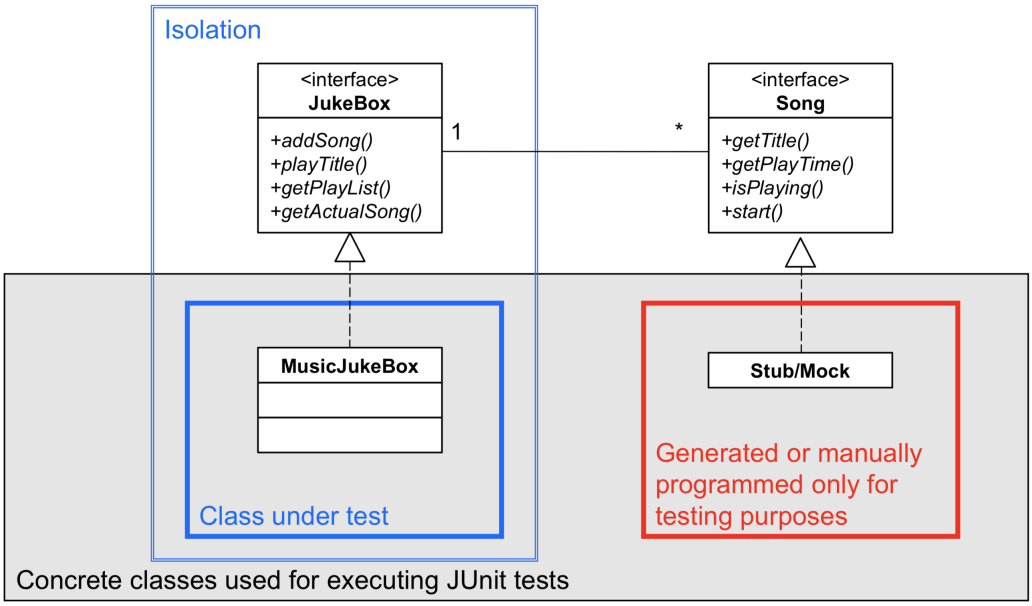
6.5.3 Test Doubles - Stubs

- Stub is a dumb object!
- A Stub is just an object that returns the same value over and over again.
- Allows implementation of tests without production code.

For example:

```
class EmailStub implements EmailService{
   void sendMail(String adr; String Text) {;}
}
```

6.5.4 Stubs - Sample Application



6.6 Mock Testing

- Mock objects simulate parts of the behaviour of domain objects.
- Classes can be tested in isolation by simulating their collaborators with mock objects.
- Takes classes out of a production environment and puts them in a well defined lab environment.

6.6.1 When to use Mock Objects

- The real object has non-deterministic behaviour.
- The real object is difficult to set up.
- The real object is slow.
- The real object has a user interface.
- The test needs to ask the real object about how it was used (-¿ if callback function was called)
- The real object does not yet exist.

6.6.2 Pros and Cons of Mock Objects

- Pros

  Emphasize that testing is about isolation

  Simplifies handling of interfaces with many methods

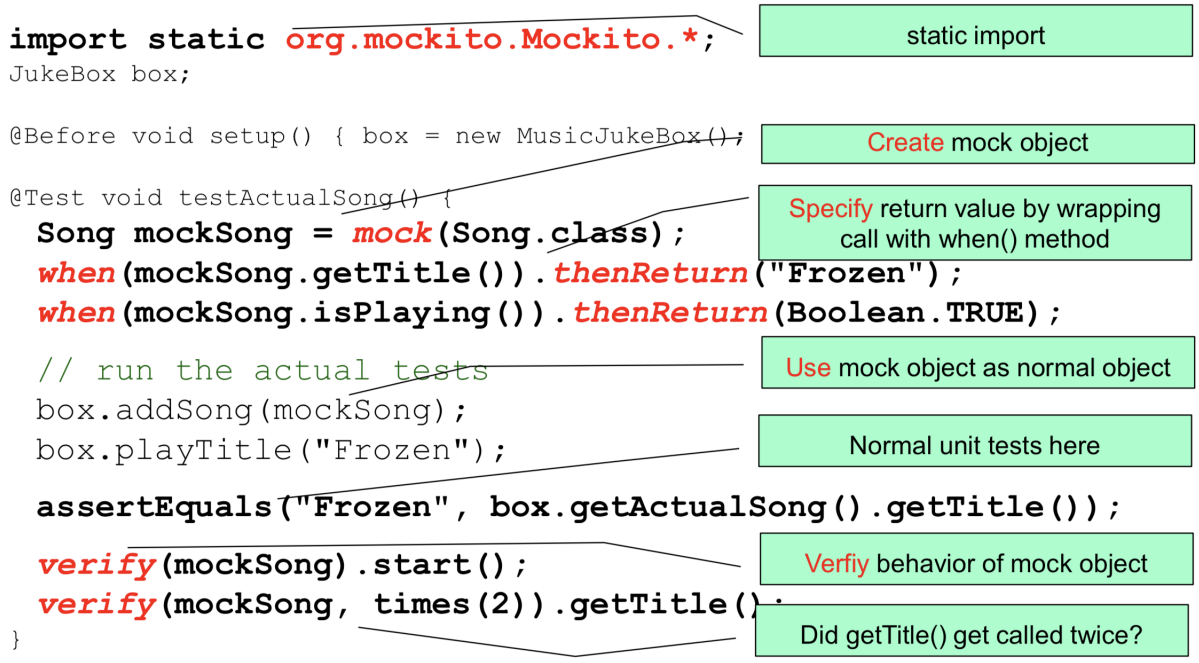
  Can enable near-instant testing even of code that uses resource-bound APIs such as JDBC

- Cons

  Can mirror the implementation too closely, making atest suite fragile

  Mocking can become complex with APIs like JDBC

6.7 Mockito



6.7.1 Mocking Usage Pattern

1. Create a mock object for the interface or class we would like to simulate
2. Specify the expected behavior of the mock
3. Use the mock in standard unit testing, as if it was a normal object
4. Verify behavior

6.7.2 Mockito Benefits

- No hand-writing of classes for mock objects needed.
- Supports refactoring-safe mock objects: test code will not break at runtime when renaming methods or reordering method parameters
- Supports return values and exceptions.
- Supports order-checking of method calls, for one or more mock objects.
- Supports checking of the number of actual calls of the mock objects

7 Javadoc

- Is a separate program that comes with the JDK
- Reads your program, makes lists of all the classes, interfaces, methods, and variables, and creates HTML pages displaying its results
- Its output is very professional looking

This makes `you` look good ... and it also helps keep your manager from imposing bizarre documentation standards

- Write comments for the programmer who uses your classes

Anything you want to make available outside the class should be documented. It is a good idea to describe, for your own use, private elements as wel. `javadoc` can be set to generate documentation for: only public elements

- public and protected elements
- public, protected, and package elements
- everything–that is, public, protected, package, and
- private elements

7.1 Tags In javadoc Comments

Use the standard ordering for javadoc tags In method descriptions, use:

```
@param p A description of parameter p.
@return A description of the value returned (unless the method returns void).
@exception e Describe any thrown exception.
@see Adds a "See Also" heading with a link or text entry that points to reference
```

Don't use tags unless you maintain them!

7.2 Know Where To Put Comments!

javadoc comments must be immediately before: a package (only in package-info.java)

- a class
- an interface
- a constructor a method
- a field

Anywhere else, javadoc comments will be *ignored*!

7.3 Hints

- Keep comments up to date
- Use the word "this "rather than "the "when referring to instances of the current class.
- Hence, `this object` has an especially clear meaning in comments Example: `Decides which direction this frog should move.`(As a comment in the `Frog` class)
- Summarize the essence of your method, field or package comment in the first sentence.
- Javadoc copies the first sentence into a summary paragraph.
- Include examples if they are helpful.

8 Software Smells & Refactoring

Is the process of changing a software system in such a way that it does not alter the external behavior of the code, yet improves its internal structure. *Martin Fowler, 1999*

8.1 Code Smell

A *Code Smell* is a hint that something *might* be wrong with the code.

8.2 How do i find Code Smells?

- Sniff out the "bad smells in code"! (Check out Martin Fowler, Cap 3)
- Analyze your code
  - Manually, by looking at the code
  - Automated, using software analysis tools
    - Checkstyle for java
    - Spotbugs for Java
    - PMD for Java

8.3 Some Typical Code Smells

- "Too Much" Code Smells Duplicated Code
  - Long Method
  - Large Class
  - Long Parameter List
  - Feature Envy
  - Switch Statements
  - Parallel Inheritance Hierarchies
- Code Change Smells
  - Divergent Change
  - Shotgun Surgery
- "Not enough" Code Smells
  - Empty Catch Clause
- Comment Smells
  - Need To Comment
  - Too much Comments

8.4 Why Refactoring

- Context: You need to modify existing code
  - extend/adapt/correct/...
- (Bad) Solution:
  - Just add new features
- Consequence:
  - Design decays
  - Duplicated code
  - Long methods / classes , ...

- (Good) Solution:

  first make code simpler, prepare for more complexity resp. → Refactor

  Add new features

- Consequence:

  Code stays simple

8.5 Benefits of Refactoring

- Refactoring improves the design of your system

  Fight code rot!

- Refactoring makes your software easier to understand

  because structure is improved

  duplicated code is removed

  etc.

- Refactoring helps you to find bugs

  because it promotes a deep understanding of the code

- Refactoring saves development time

  because good code is easier to evolve

  because your code/design is more robust

8.6 Prerequisites for Refactoring

8.6.1 Automated measures

- Version Control System
- Continuous integration
- Tests
- Coding standards

8.6.2 Social measures

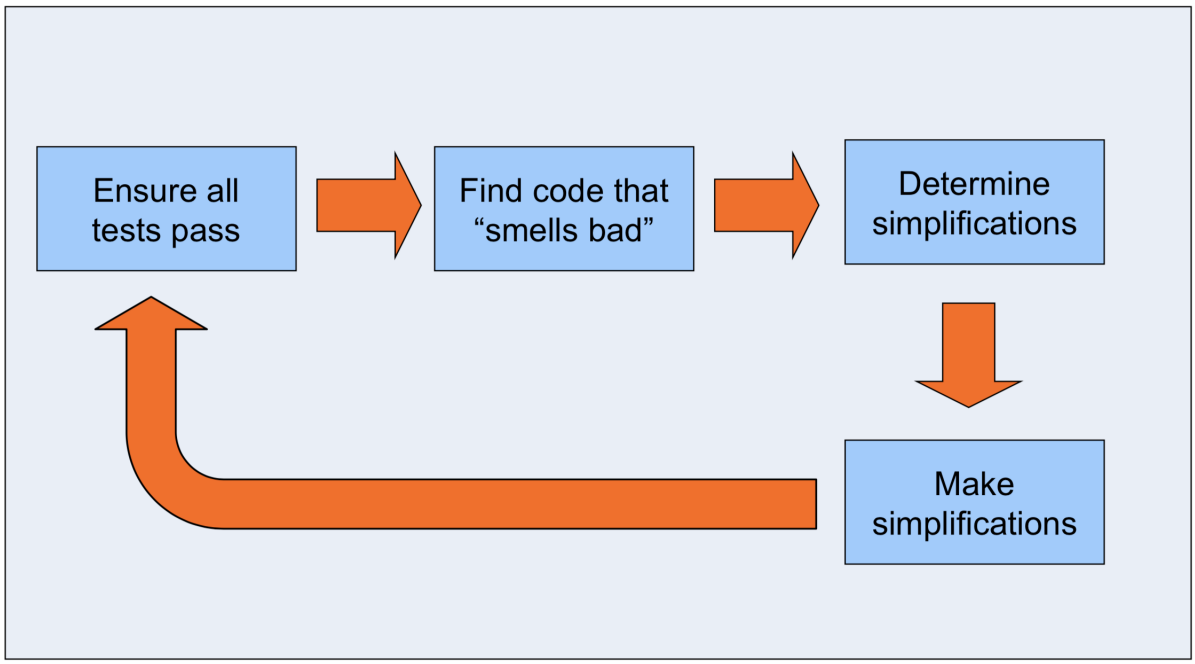
These are in descent order of importance

- Collective ownership
- Rules of Simplicity
- Sustainable Pace
- Pair programming

8.6.3 The two Hats of Refactoring

- Don't try to clean the code when wearing the function hat.
- Don't try to add features when wearing the refactoring hat.

8.7 Refactoring Workflow



8.8 When?

- Do refactor
  - Before or after you add functionality
  - When you learn something about the code
  - As a consequence of a bug fix
  - As a consequence of code review
- Do not refactor
  - When the tests are not passing
  - When you should re-implement the code

8.9 Problems with Refactoring

- Taken too far, refactoring can lead to incessant tinkering with the code, trying to make it perfect
- Databases can be difficult to refactor code is easy to change; databases are not
- Refactoring published interfaces can cause problems for the code that uses those interfaces

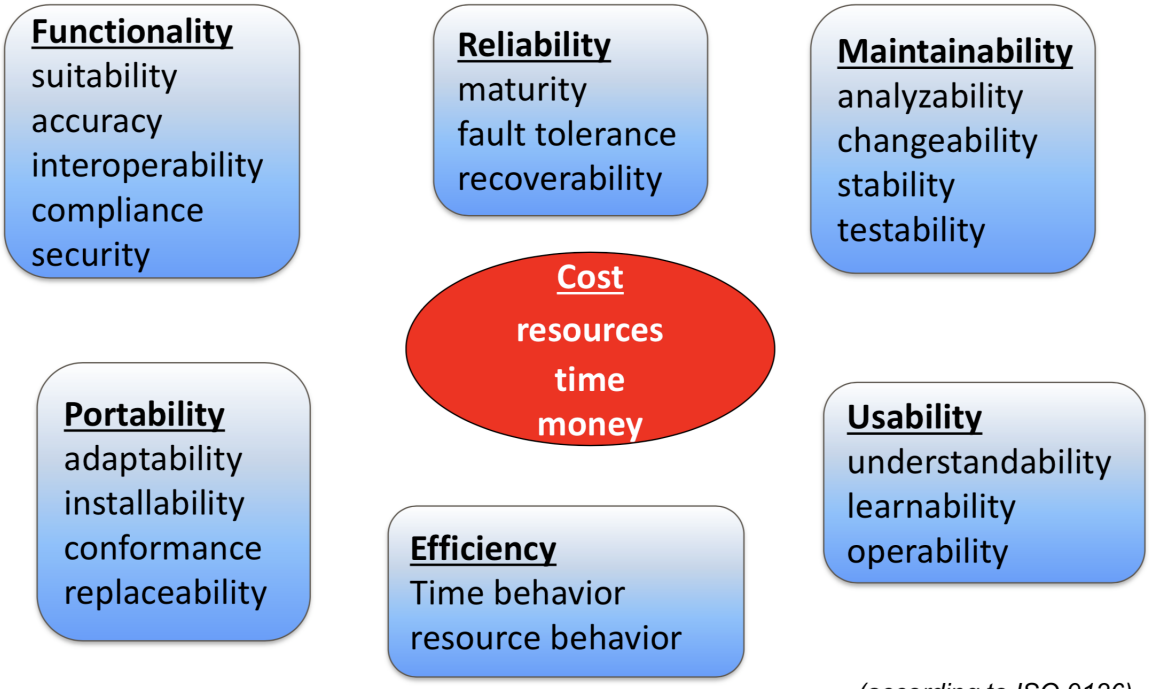
9 Metrics

Top 5 reasons why IT projects fail:

1. Communication problems
2. Bad planning / lack of planning
3. Lack of ressources
4. Requirements and goals are not clear
5. Politics

So code quality isn't an issue here?

9.1 Software Quality



9.2 Characteristics of useful Metrics

A useful metric has four characteristics

- **Measurable:** If you can't measure it, you can't use it!
- **Independent:** A metric must be independent of any influence of the project members! If they can manipulate the metric, it gets useless!
- **Reliable:** Conclusions drawn from metrics must be plausible. Therefore the raw data must be reliable.
- **Accurate:** Metrics must be accurate, to be useful. Even if the precision varies!

9.3 Static Software Metrics

- Size Metrics

  Lines of Code; Number of Statements, Fields, Methods, Packages

- Dataflow Metrics
  - Cyclomatic Complexity (McCabe Complexity)
  - Dead Code detection Initialization before use
  - In Java: done by the compiler!
- Style Metrics
  - Number of Levels (nesting depth)
  - Naming conventions, formatting conventions
- OO Metrics
  - No. of classes, packages, methods Inheritance depth / width
- Coupling Metrics
  - LCOM: lack of cohesion metrics
  - No. of calling classes / no. of called classes Efferent / Afferent couplings
- Statistic Metrics
  - Instability
  - Abstractness

9.4 Dynamic Software Metrics

- Performance Metrics Time spent
  - Memory consumption (also memory leaks)
  - Network traffic
  - Bandwidth needed
  - No. of clicks to perform a task
- Other Dynamic Metrics
  - No. of tests executed, No. of failed tests
  - Code coverage

9.5 What else can be measured

- During the software project
  - Project costs vs. estimated project costs
  - Costs per functionality
  - Delivered functionality vs planned functionality
  - Costs of separate activities (meeting-time vs. development time)
- During development
  - Person months/years spent on development
  - Change requests implemented
  - Impact of change requests per Module
  - Found bugs per Code volume
  - Bugs per Module

Time to solve bugs / cost to solve bugs

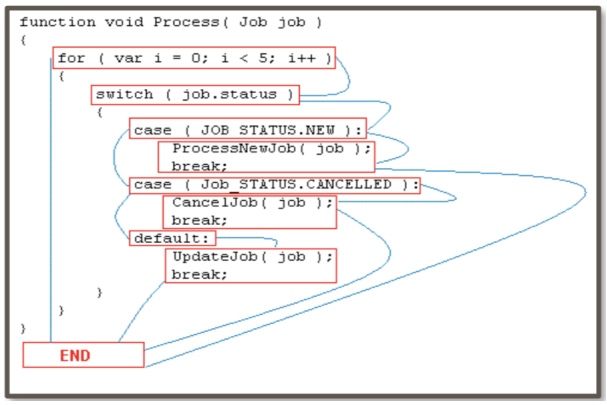
- **It is important to set the metrics in context!**

9.6 Cyclomatic Complexity (McCabe)

- Cyclomatic Complexity is the number of possible execution paths through the code.
- For a single method the Cyclomatic number N is defined as follows:

```
    N = b + 1 whereas b is the number of binary decision points (if, while, for, case)
N < 10 => code well readable
```

- Criticism: switch-statements are well readable but boost the cyclomatic number



Ablauf zu zyklischen Komplexität

9.6.1 Decision Points

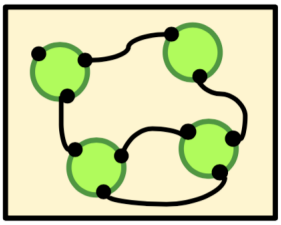
Typische Entscheidungspunkte sind:

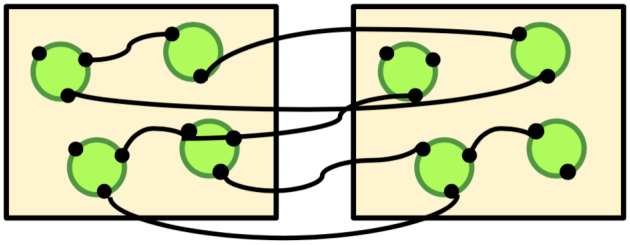
- if
- for
- while
- case

Wird oft falsch gemacht:

- switch-Statement selbst ist kein Entscheidungspunkt, wohl aber jedes einzelne case- Statement. Denn erst bei den case-Statements steht die Bedingung, welche entscheidet, ob der nachfolgende Block ausgeführt wird.
- Else und default sind keine Entscheidungspunkte, die Entscheidung wird im if, bzw in allen anderen case-Statements getroffen. Else und default sind dann nur noch die übrigbleibende Alternative.
- break: springt zwar auch im Kontrollfluss über folgende Statements hinweg, ist aber kein Entscheidungspunkt, da der Sprung unbedingt ist.

9.7 Cohesion and Coupling Metrics





- Cohesion

  Measure how closely related the methods of a class are.

- Coupling / Dependency

  Measure the degree to which each class relies on other classes

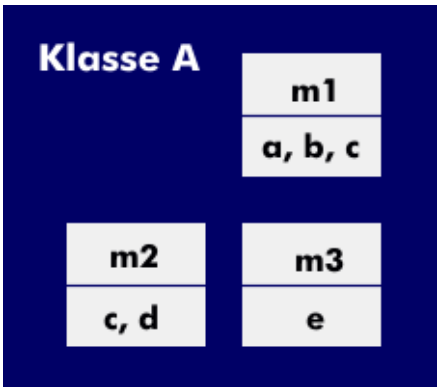
- Low coupling usually correlates with high cohesion (which is optimal)
- Metrics: LCOM (Lack of Cohesion in Method) several LCOM metrics!

9.8 LCOM - Lack of cohesion in methods

Zur Messung von Softwareprodukten werden sogenannte Software-Metriken verwendet, die unterschiedliche Eigenschaften von Softwareprodukten und -prozessen quantifizieren. Eine objektorientierte Metrik berücksichtigt bei der Messung von Software die Zusammenfassung von Datenstrukturen und der darauf anwendbaren Methoden zu einem Objekt, dessen Beziehungen zu anderen Objekten sowie die generellen Strukturmerkmale objektorientierter Programmierung. Die Metrik nach Lack of Cohesion in Methods (LCOM) ist ein Maß für die Kohäsion - d.h. den Zusammenhalt - von Methoden in der betrachteten Klasse. Weitere objektorientierte Software-Metriken sind Weighted Methods for Class (WMC), Response for a Class (RFC), Coupling Between Objects (CBO), Depth of Inheritance Tree (DIT) und Number of Children (NOC).

LCOM-Definition: LCOM ist gleich der Paare von Methoden ohne gm. Instv. Abzüglich der Paare von Methoden mit gm. Instv. gm. Instv. ist die gemeinsame Instanzvariable.

Wichtig bei dieser Methode ist es, darauf zu achten, dass jeweils Paare von Methoden betrachtet werden. Da ein negativer Wert von LCOM unsinnig ist, wird LCOM=0 gesetzt, wenn die Subtraktion einen negativen Wert ergibt.



Gemäß anliegender Darstellung ist eine Klasse A mit 3 Methoden m1(), m2() und m3() gegeben, die jeweils auf bestimmten Instanzvariablen operieren.

Beispiel Klasse mit Methoden und Instanz-variablen:

Paare: m1/m3, m1/m2, m2/m3, mit m1/m2 als gemeinsame Instanzvariable c.

Paare von Methoden ohne gm. Instv. = 1+0+1, Paare von Methoden mit gm. Instv. = 0+1+0

LCOM(A) = 2-1 = 1.

Anwendung: Ansatz zur Überprüfung der Kapselung einer Klasse. Dies aus der Überlegung heraus, dass ein hoher Wert von LCOM auf eine schlechte Kapselung hindeutet, da offensichtlich viele Methoden auf disjunkten Attributmengen operieren.

10 Logging

10.1 Why logging?

- Logging is a technique for Monitoring and Debugging applications
- Debuggers focus on the state of a program **now**

  A stack trace can tell you only how the program got here directly

  It is not possible to detect what was before this actual call
- Logging provides information about the state of a program or a data structure **over time**

  Logging can reveal several classes of errors and information that debuggers cannot.

10.2 Value of Logging

- Logging is invaluable in any system that needs to know how

  an application is used

  who is using the application

  sequences of event

  how fast the system is

  how well a systems scales
- Especially for

  real-time systems

  event-based application

  distributed environment

  (concurrent processes)

10.3 Approaches to logging

Do not use `System.out.println`. Use a Standart Framework such as Log4j2, slf4j, Logback, Standard Java Logging

10.3.1 Benefits using Logframeworks

- Configuration of the logging features is externalized
- Log messages can be prioritized
- Logging supports different message layouts
- Enabling / disabling at runtime
- Support different output targets
- Different output levels
- supports message caching in order to prevent performance decrease

10.4 Log4j*

Log4j is an Apache Open Source Project. It's actual Release is 2.x and you can find Documentation under http://logging.apache.org/log4j/2.x/. Originally developed by IBM at Zurich Research Labs. It is the father of many other logging Frameworks like: log4php, log4cxx, log4net, ...

10.4.1 Basic Usage

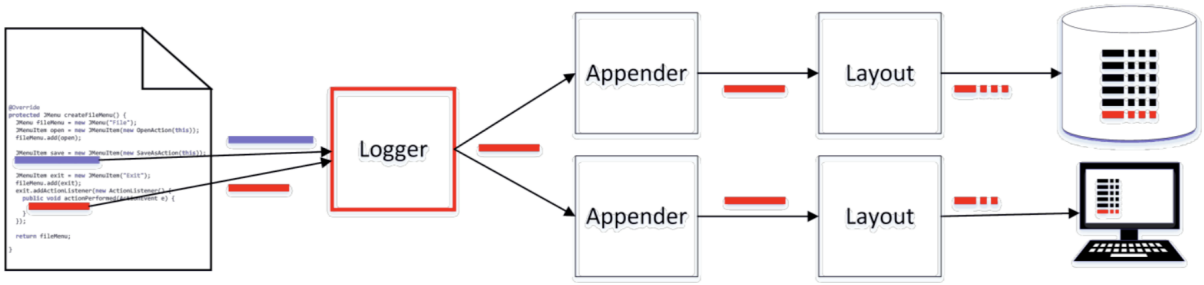
```
import org.apache.logging.log4j.LogManager;
import org.apache.logging.log4j.Logger;

class Foo {
    static Logger logger = LogManager.getLogger(Foo.class);

    public Foo(){
        logger.info("Constructing foo");‘
    }
    public String doStuff(Long x){
        logger.debug("doing stuff");
    }
}
```

10.5 Concepts

- **Priority**: Determines in the code what is logged.
- **Level**: Determines in the configuration wheter something logged.
- **Layout (Formatter)**: Determines how it is logged.
- **Appender**: Determines where it is logged.
- **Logger**: Writes a log entry to an appender in a given layout it the appropriate level is met by the priority of the log-instruction.



10.6 Logging Priorities

10.6.1 Defect logging

- FATAL: In case of very severe errors from which your application cannot recover.
- ERROR: When an error is encountered (but you application can still run).
- WARN: Log requests to alert about harmful situations.

10.6.2 Informative logging

- INFO: Logs to inform about progress of the application at a coarse grained level.
- DEBUG: Log requests that are relevant to debug the application.
- TRACE: Log on an extremely fined grained level to trace control flow.

10.7 Logging API

Level	Command
FATAL	fatal(Object message [, Throwable throwable]);
ERROR	error(Object message [, Throwable throwable]);
WARN	warn(Object message [, Throwable throwable]);
INFO	info(Object message [, Throwable throwable]);
DEBUG	debug(Object message [, Throwable throwable]);
TRACE	trace(Object message [, Throwable throwable]);
As wrapper	log(Priority level, Object message); log(Priority level, Object message,Throwable throwable);

10.8 Logger Names

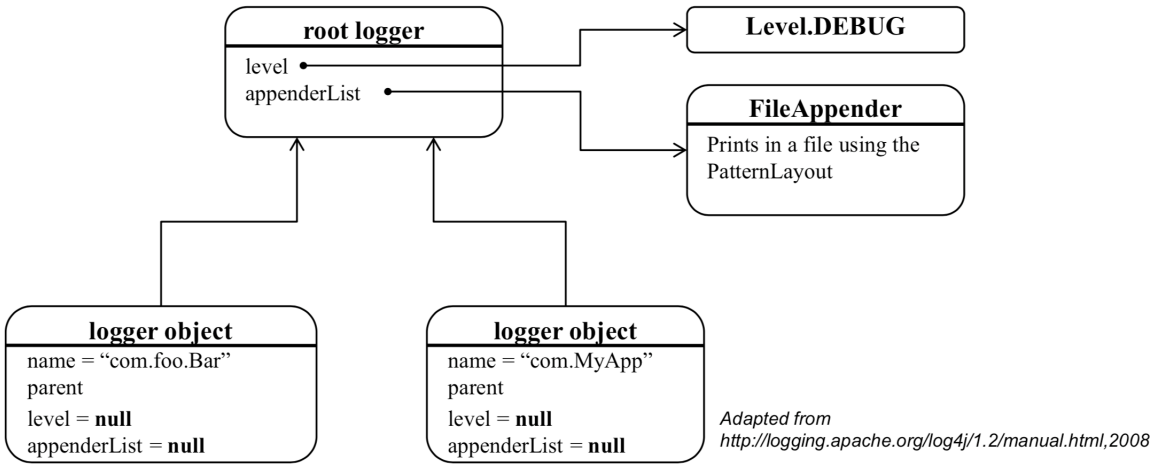
Name a Logger by locality. Instantiating a logger in each class, with the logger name equal to the fully-qualified name of the class. This is a useful and straightforward approach for defining loggers.

```
class Foo { static Logger logger = LogManager.getLogger(Foo.class); ... }
```

10.8.1 Benefits of using Fully Qualified Class Names

- It is very simple to implement.
- It is very simple to explain to new developers.
- It automatically mirrors the application's own modular design.
- It can be further refined at will.
- Printing the logger automatically gives information on the locality of the log statement.
- A Logger is an ancestor of another Logger if its name followed by a dot is a prefix of the descendant logger name.
- For example, the Logger named "com.fooïs a parent of the Logger named "com.foo.Bar". Similarly, "javaïs a parent of "java.utiländ an ancestor of "java.util.Vector". This naming scheme should be familiar to most developers.

10.9 Loggers, Appenders und Layouts



Logger sind in Hierarchien definiert. Über Properties kann man das logging zur runtime anpassen. Jedem Logger ist ein Level zugeordnet. Ein Logstatement wird nur ausgegeben, wenn seine Priorität (also Level) und seine Kategorie (logger) zutreffen. Über Appender können mehrere Output angegeben werden. Logger definieren eine Level und einen Appender.

```
<Logger name = "swc.logging.Main" level="debug" >
  <AppenderRef ref="console" />
</Logger>
```

```
<Console name="console" target="SYSTEM_ERR">
    <PatternLayout pattern="%-5p %c{1} - %m%n"/>
</Console>
```

Wenn kein Logger konfiguriert ist über eine Konfigurationsdatei oder im Sourcecode, wird der Output an den Root Logger gesendet.

10.10 Level

- Levels have the same name as priorities
- Levels are set in the configuration (log4j2.xml)
- Priorities are programmed in the code
- Levels determine which log messages will be shown in the log. They act as a filter which allows to pass logs with priorities of the same name and higher to pass to the appenders. All other logs will be ignored.

10.11 Appender

Ein Appender sendet seine empfangenen Messages an einen definierten Endabnehmer. Dies können sein:

- ConsoleAppender - Write log to System.out or System.err
- FileAppender – Write to a log file
- SocketAppender – Dump log output to a socket
- SyslogAppender – Write to the syslog.

- NTEventLogAppender – Write the logs to the NT Event Log system.
- RollingFileAppender – backup the log files after they reach a certain size.
- SMTPAppender – Send Messages to email
- JMSAppender – Send messages using Java Messaging Service

Es kann auch ein eigener Implementiert werden, je nachdem, was gebraucht wird.

10.12 Pattern Layout

Mit einem Pattern könne die Meldungen formatiert ausgegeben werden. Die Definition eines Appenders ist stark an die printf Funktion aus C angelehnt.

```
<Console name="console" target="SYSTEM_OUT">
    <PatternLayout pattern ="%d [%t] %-5p %c - %m%n"/>
</Console>
```

Dieses Pattern resultiert in der Ausgabe:

```
2000-09-07 14:07:41,508 [main] INFO MyApp - Entering application.
2000-09-07 14:07:41,529 [main] INFO MyApp - Exiting application.
```

10.13 Beispiel log4j.xml

```
<?xml version="1.0" encoding="UTF-8" ?>
<Configuration>
<Appenders>
    <Console name="console" target="SYSTEM_OUT">
        <PatternLayout pattern="%d [%t] %-5p %c - %m%n"/>
  </Console>
</Appenders>
<Loggers>
    <Root level="info" additivity="false">
        <AppenderRef ref="console" />
    </Root>
</Loggers>
</Configuration>
```

10.14 Costs of logging

- Logging performance is an issue
- When logging is turned off entirely or just for a set of priorities, the cost of a log request consists of a method invocation plus an integer comparison.
- The typical cost of actually logging is about 100 to 300 microseconds. This is the cost of formatting the log output and sending it to its target destination.

10.15 Hidden costs of logging

- Method invocation involves the ”hidden”cost of parameter construction.
- To avoid the parameter construction cost use lazy logging with lambdas:

```
logger.debug(() -> ("Entry number: " + i + " is " + String.valueOf(entry[i])));
// or before Java 1.8..
if (logger.isDebugEnabled()) {
logger.debug("Entry number: " + i + " is " + String.valueOf(entry[i]));
}
```

10.16 More Log4j2 Fearures

- Async Loggers - performance similar to logging switched off
- Custom log levels
- Automatic reload of configuration upon modification
- Java 8-style lambda support for lazy logging
- Filtering: filtering based on context data, markers, regular expressions, and other components in the Log event.
- Plugin Architecture - easy to extend by building custom components

10.17 Logging Exceptions

- Don’t use e.printStackTrace() use log.error(Ëxception message”, e) instead
- Don’t log an exception and throw same exception again
- Don’t discard the stack trace, append it!

  } catch(SQLException e){ throw new RuntimeException(”DB exception”, e); }

10.18 log4j2 Patterns Vars

Conversion	Character Effect
c	Used to output the category of the logging event.
C	Used to output the fully qualified class name of the caller issuing the logging request %C1 will output SSomeClass".
d	Used to output the date of the logging event. The date conversion specifier may be followed by a date format specifier enclosed between braces. For example, %d{HH:mm:ss,SSS}
F	Used to output the file name where the logging request was issued.
l	Used to output location information of the caller which generated the logging event.
L	Used to output the line number from where the logging request was issued.
m	Used to output the application supplied message associated with the logging event.
M	Used to output the method name where the logging request was issued.
n	Outputs the platform dependent line separator character or characters.
p	Used to output the priority of the logging event.
r	Used to output the number of milliseconds elapsed from the construction of the layout until the creation of the logging event.
t	Used to output the name of the thread that generated the logging event.
x	Used to output the NDC (nested diagnostic context) associated with the thread that generated the logging event.
X	Used to output the MDC (mapped diagnostic context) associated with the thread that generated the logging event.
%	The sequence %% outputs a single percent sign.

11 Codebeispiele

11.1 RentalTest

```
package ch.fhnw.swc.mrs.model;

import ch.fhnw.swc.mrs.api.MovieRentalException;
import org.junit.jupiter.api.BeforeEach;
import org.junit.jupiter.api.Disabled;
import org.junit.jupiter.api.DisplayName;
import org.junit.jupiter.api.Test;

import java.time.LocalDate;

import static org.junit.jupiter.api.Assertions.*;

/**
 * Unit tests for the Rental class.
 */
@DisplayName("Tests for class Rental")
public class RentalTest {

    private User mickey, donald;
    private Movie theKid, goldrush;
    private PriceCategory pc;
    private LocalDate today;

    /**
     * Creates legal User and Movie objects and sets the reference time stamp to now.
     */
    @BeforeEach
    public void setUp() {
        today = LocalDate.now();
        pc = RegularPriceCategory.getInstance();
        mickey = new User("Mouse", "Mickey");
        mickey.setId(4);
        donald = new User("Duck", "Donald");
        donald.setId(7);
        theKid = new Movie("The Kid", today, pc);
        theKid.setId(6);
        goldrush = new Movie("Goldrush", today, pc);
        goldrush.setId(2);
    }

    @DisplayName("Does Rental object get initialized correctly with constructor?")
    @Test
    public void testRentalCtorWithRentalDate() {
        Rental r = new Rental(mickey, theKid, today.minusDays(6));
        doAssertionsForTestRental(6, r);
    }

    private void doAssertionsForTestRental(int expected, Rental r) {
        // is the rental registered with the user?
        assertTrue(mickey.getRentals().contains(r));
        // has the movie's rented state been set to rented?
        assertTrue(theKid.isRented());
        // is the number of rental days set correctly?
```

```
    assertEquals(expected, r.getRentalDays());
    // has the rental date been set?
    assertNotNull(r.getRentalDate());
    // do we get the objects that we set?
    assertEquals(mickey, r.getUser());
    assertEquals(theKid, r.getMovie());
}

@DisplayName("Throws exception when ctor is called with null User ?")
@Test
public void testRentalCtorWithNullUser() {
    Throwable t = assertThrows(NullPointerException.class, () -> new Rental(null, theKid, today));
    assertEquals(Rental.EXC_USER_NULL, t.getMessage());
}

@DisplayName("Throws exception when ctor is called with null Movie?")
@Test
public void testRentalCtorWithNullMovie() {
    Throwable t = assertThrows(MovieRentalException.class, () -> new Rental(mickey, null, today));
    assertEquals(Rental.EXC_MOVIE_NOT_RENTALBE, t.getMessage());
}

@DisplayName("Throws exception when ctor is called with null Date?")
@Test
public void testRentalCtorWithNullDate() {
    Throwable t = assertThrows(IllegalArgumentException.class, () -> new Rental(mickey, theKid, null
    assertEquals(Rental.EXC_RENTAL_DATE_IN_FUTURE, t.getMessage());
}

@DisplayName("Throws exception when ctor is called with rental date in the future?")
@Test
public void testRentalCtorWithFutureDate() {
    Throwable t = assertThrows(IllegalArgumentException.class, () -> new Rental(mickey, theKid, toda
    assertEquals(Rental.EXC_RENTAL_DATE_IN_FUTURE, t.getMessage());
}

@DisplayName("Is rental duration calculated correctly?")
@Test
public void testCalcDaysOfRental() {
    LocalDate rentalDate = LocalDate.now().minusDays(6);
    Rental r = new Rental(mickey, theKid, rentalDate);

    int days = r.getRentalDays();
    assertEquals(6, days);
}

@DisplayName("Do Id getter and setter work correctly and prevent setting id once it is set?")
@Test
public void testSetterGetterId() {
    Rental r = new Rental(mickey, theKid, today);
    r.setId(11);
    assertEquals(11, r.getId());

    // setting id a 2nd time
    assertThrows(IllegalStateException.class, () -> r.setId(47));
    assertEquals(11, r.getId());
}
```

```
    @DisplayName("Do Movie getter and setter work correctly and prevent setting null Movie?")
    @Test
    public void testSetterGetterMovie() {
        Rental r = new Rental(mickey, theKid, today);
        r.setMovie(goldrush);
        assertEquals(goldrush, r.getMovie());

        Throwable t = assertThrows(MovieRentalException.class, () -> r.setMovie(null));
        assertEquals(Rental.EXC_MOVIE_NOT_RENTALBE, t.getMessage());
        assertEquals(goldrush, r.getMovie());
    }

    @DisplayName("Do User getter and setter work correctly and prevent setting null User?")
    @Test
    public void testSetterGetterUser() {
        Rental r = new Rental(mickey, theKid, today);
        r.setUser(donald);
        assertEquals(donald, r.getUser());

        Throwable t = assertThrows(NullPointerException.class, () -> r.setUser(null));
        assertEquals(Rental.EXC_USER_NULL, t.getMessage());
        assertEquals(donald, r.getUser());
    }

    @DisplayName("Does equals work correctly?")
    @Test
    @Disabled
    public void testEquals() {
    }

    @DisplayName("Is hash code calculated correctly?")
    @Test
    public void testHashCode() {
        Rental x = new Rental(mickey, theKid, today);
        theKid.setRented(false);
        Rental y = new Rental(mickey, theKid, today);

        x.setId(42);
        y.setId(42);
        assertEquals(x.hashCode(), y.hashCode());

        x.setMovie(goldrush);
        assertTrue(x.hashCode() != y.hashCode());
        y.setMovie(goldrush);
        assertEquals(x.hashCode(), y.hashCode());

        x.setUser(donald);
        assertTrue(x.hashCode() != y.hashCode());
        y.setUser(donald);
        assertEquals(x.hashCode(), y.hashCode());
    }

}
```

11.2 Checkstyle

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE module PUBLIC "-//Puppy Crawl//DTD Check Configuration 1.3//EN"
       "http://www.puppycrawl.com/dtds/configuration_1_3.dtd">

<!--
    This configuration file was written by the eclipse-cs plugin configuration editor
-->
<!--
    Checkstyle-Configuration: SWC Checks
    Description: none
-->
<module name="Checker">
    <property name="severity" value="error"/>
    <module name="TreeWalker">
        <property name="tabWidth" value="4"/>
        <module name="JavadocMethod">
            <property name="scope" value="public"/>
            <property name="excludeScope" value="protected"/>
            <property name="allowUndeclaredRTE" value="true"/>
        </module>
        <module name="JavadocType">
            <property name="excludeScope" value="private"/>
            <property name="scope" value="protected"/>
        </module>
        <module name="JavadocVariable">
            <property name="excludeScope" value="private"/>
            <property name="scope" value="protected"/>
        </module>
        <module name="JavadocStyle">
            <property name="scope" value="package"/>
            <property name="excludeScope" value="private"/>
        </module>
        <module name="ConstantName"/>
        <module name="LocalFinalVariableName"/>
        <module name="LocalVariableName"/>
        <module name="MemberName"/>
        <module name="MethodName"/>
        <module name="PackageName"/>
        <module name="ParameterName"/>
        <module name="StaticVariableName"/>
        <module name="TypeName"/>
        <module name="AvoidStarImport"/>
        <module name="IllegalImport"/>
        <module name="RedundantImport"/>
        <module name="UnusedImports"/>
        <module name="LineLength">
            <property name="max" value="120"/>
        </module>
        <module name="Indentation">
            <property name="caseIndent" value="0"/>
        </module>
        <module name="MethodLength">
            <property name="max" value="100"/>
            <property name="tokens" value="METHOD_DEF"/>
        </module>
```

```
        <module name="ParameterNumber"/>
        <module name="EmptyForIteratorPad"/>
        <module name="MethodParamPad"/>
        <module name="NoWhitespaceAfter"/>
        <module name="NoWhitespaceBefore"/>
        <module name="OperatorWrap"/>
        <module name="ParenPad"/>
        <module name="TypecastParenPad"/>
        <module name="WhitespaceAfter"/>
        <module name="WhitespaceAround"/>
        <module name="ModifierOrder"/>
        <module name="RedundantModifier"/>
        <module name="AvoidNestedBlocks"/>
        <module name="EmptyBlock"/>
        <module name="LeftCurly"/>
        <module name="NeedBraces"/>
        <module name="RightCurly"/>
        <module name="EmptyStatement"/>
        <module name="EqualsHashCode"/>
        <module name="IllegalInstantiation"/>
        <module name="InnerAssignment"/>
        <module name="MissingSwitchDefault"/>
        <module name="SimplifyBooleanExpression"/>
        <module name="SimplifyBooleanReturn"/>
        <module name="FinalClass"/>
        <module name="HideUtilityClassConstructor"/>
        <module name="InterfaceIsType"/>
        <module name="VisibilityModifier"/>
        <module name="ArrayTypeStyle"/>
        <module name="TodoComment"/>
        <module name="UpperEll"/>
        <!--- Start of Metric analysis -->
        <!-- disable metrics
            <module name="ClassFanOutComplexity"/>
            <module name="ClassDataAbstractionCoupling">
              <property name="max" value="10"/>
            </module>
            <module name="BooleanExpressionComplexity">
              <property name="max" value="4"/>
            </module>
            <module name="CyclomaticComplexity"/>
            <module name="JavaNCSS"/>
        --><!-- End of Metric analysis -->
    </module>
    <module name="FileLength"/>
    <module name="NewlineAtEndOfFile">
        <property name="lineSeparator" value="lf_cr_crlf"/>
    </module>
    <module name="Translation"/>
    <module name="FileTabCharacter">
        <property name="fileExtensions" value="*.java"/>
    </module>
</module>
```