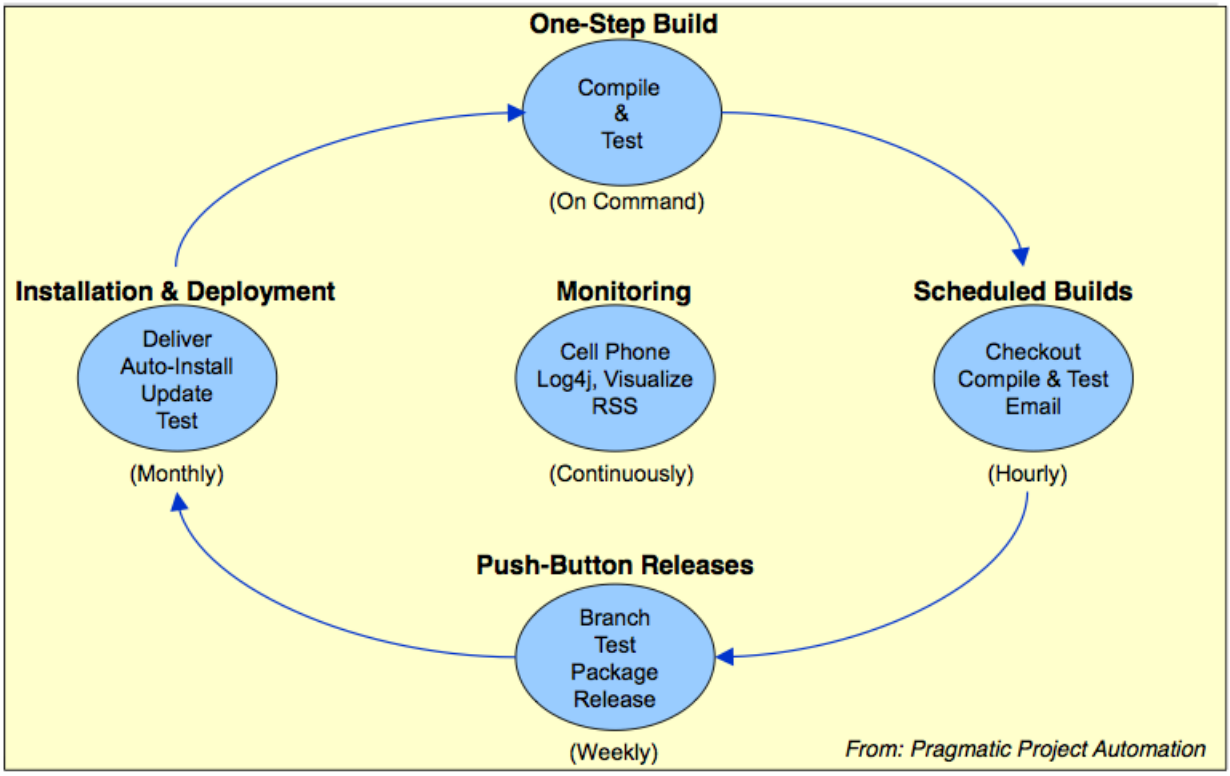
# Software-Konstruktion

Jan Fässler

2. Semester (FS 2012)

# 1 Build Automation




## 1.1 CRISP Builds

**Complete** recipe lists all ingredients

**Repeatable** version control time machine

**Informative** radiate valuable information

**Schedulable** complete and repeatable

**Portable** machine-independent

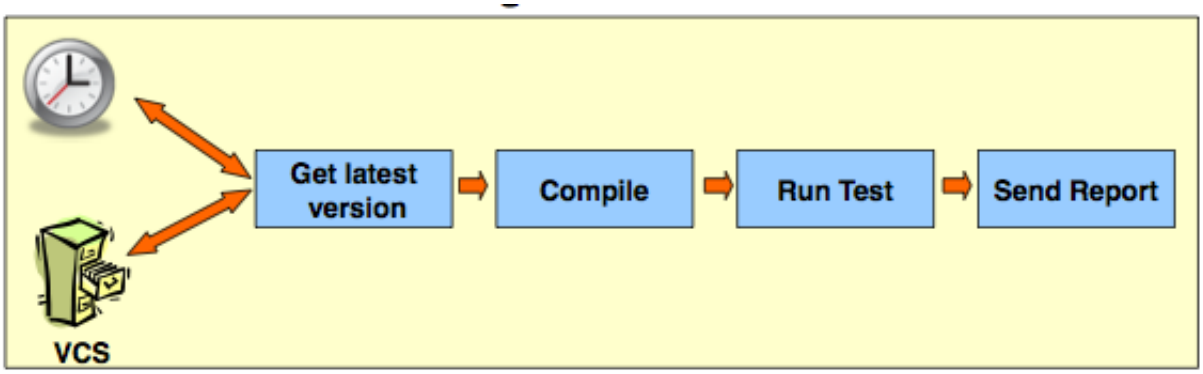
## 1.2 Ant-Script

```
1  <project name="Software Construction Lab" default="compile" basedir="..">
2      <property file="build/build.properties" />
3
4      <!-- The application's classpath -->
5      <path id="application.classpath">
6          <fileset dir="${lib.dir}">
7              <include name="**/*.jar"/>
8          </fileset>
9      </path>
10
11     <!-- The build tools classpath -->
12     <path id="build.classpath">
13         <fileset dir="${build.lib.dir}">
14             <include name="*.jar" />
15         </fileset>
16         <path refid="application.classpath"/>
17     </path>
18
19     <target name="clean">
20       <delete dir="${bin}"/>
```

```
21      <delete dir="${log}"/>
22    </target>
23
24    <target name="prepare" depends="clean">
25        <mkdir dir="${bin.classes.dir}" />
26        <mkdir dir="${bin.jar.dir}" />
27        <mkdir dir="${log.report.dir}"/>
28      <mkdir dir="${log.report.test.dir}" />
29      <mkdir dir="${log.report.checkstyle.dir}"/>
30    </target>
31
32    <target name="compile" depends="prepare" description="Compile the sources">
33        <javac includeantruntime="false" srcdir="${src.dir}" destdir="${bin.classes.dir}
              " classpathref="application.classpath" deprecation="on" optimize="off"/>
34        <copy todir="${bin.classes.dir}">
35            <fileset dir="${res.dir}">
36                <include name="**/*.xml" />
37                <include name="**/*.properties" />
38                <include name="**/*.png" />
39            </fileset>
40        </copy>
41    </target>
42
43    <target name="run" depends="jar" description="Run distributed application from jar
          file">
44        <java jar="${bin.jar.dir}/${name}-${version}.jar" fork="true" />
45    </target>
46
47    <target name="jar" depends="junit" description="Create jar distribution">
48        <jar jarfile="${bin.jar.dir}/${name}-${version}.jar" basedir="${bin.classes.dir}
              " excludes="**/*Test.class">
49            <manifest>
50                <attribute name="Main-Class" value="${main.class}" />
51                <attribute name="Class-Path" value=".
52                    ../../lib/dbunit-2.2.jar
53                    ../../lib/hsqldb.jar" />
54            </manifest>
55        </jar>
56    </target>
57
58  <target name="testcompile" depends="compile" description="Compiles JUnit Tests">
59    <echo message="Compile Tests" />
60    <javac includeantruntime="false" srcdir="${test.dir}" destdir="${bin.classes.dir}"
          classpathref="build.classpath" deprecation="on" optimize="off"/>
61    <echo message="Copy compiled classes to bin directory" />
62    <copy todir="${bin.classes.dir}">
63            <fileset dir="${res.dir}">
64                <include name="**/*.xml" />
65                <include name="**/*.properties" />
66                <include name="**/*.png" />
67            </fileset>
68            <fileset dir="${test.dir}">
69                <include name="**/*.xml" />
70            </fileset>
71        </copy>
72  </target>
73
74  <target name="junit" depends="testcompile" description="Runs JUnit Tests">
75       <junit haltonfailure="yes" printsummary="yes">
76          <classpath>
77            <path refid="build.classpath"/>
78            <pathelement location="${bin.classes.dir}" />
79          </classpath>
80          <formatter type="xml"/>
81            <batchtest fork="true" todir="${log.report.test.dir}">
82                <fileset dir="${test.dir}" includes="**/*Test.java"/>
83            </batchtest>
84        </junit>
85  </target>
86
```

```
87   <target name="javadoc" depends="prepare" description="Creates the javadoc">
88     <delete dir="${doc.api.dir}"/>
89       <javadoc sourcepath="${src.dir}" destdir="${doc.api.dir}" windowtitle="${name} API
         ">
90         <doctitle><![CDATA[<h1>${name} API</h1>]]></doctitle>
91         <bottom><![CDATA[<i>Copyright &#169; 2012 Dummy Corp. All Rights Reserved.</i
             >]]></bottom>
92         <tag name="todo" scope="all" description="To do:"/>
93         <link offline="true" href="http://download.oracle.com/javase/6/docs/api/"
               packagelistLoc="C:\tmp"/>
94         <link href="http://developer.java.sun.com/developer/products/xml/docs/api/"/>
95       </javadoc>
96   </target>
97
98   <taskdef resource="checkstyletask.properties" classpath="${build.lib.dir}/checkstyle
         -5.5-all.jar" />
99   <target name="checkstyle" depends="compile" description="Generates a report of code
         convention violations.">
100    <checkstyle config="${build.dir}/swc_checks.xml">
101      <fileset dir="${src.dir}" includes="**/*.java"/>
102      <classpath>
103        <pathelement location="${bin.classes.dir}"/>
104      </classpath>
105        <formatter type="xml" tofile="${log.report.checkstyle.dir}/checkstyle_report.xml
             "/>
106        <formatter type="plain" tofile="${log.report.checkstyle.dir}/checkstyle_report.
             txt"/>
107        <formatter type="plain" />
108    </checkstyle>
109  </target>
110
111  <target name="all" depends="checkstyle,javadoc,jar"/>
112
113 </project>
```

# 2 Continuous Integration




- Maintain a single source repository.
- Automate the build
- Make your build self-testing
- Everyone commits every day (at least!)
- Every commit should build the mainline on an integration machine
- Keep the build fast
- Test in a clone of the production environment
- Make it easy for anyone to get the latest executable
- Everyone can see what's happening
- Automate deployment

## 2.1 Benefits

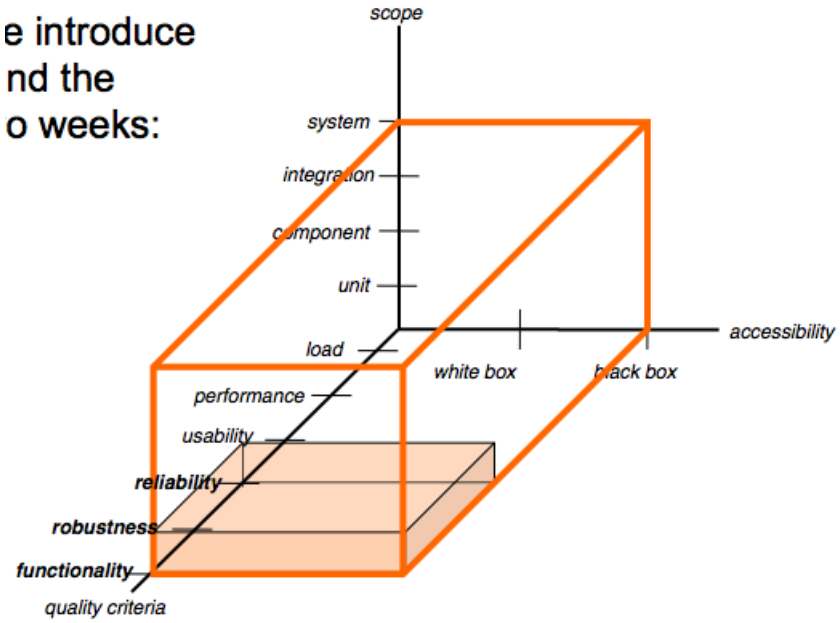
- Reduced Risks
- Always be aware of current status of the project
- Less time spent investigating integration bugs
  - Integration testing performed early
  - Integration bugs caught early
- Less time wasted because of broken code in version control system
- Prove your system can build!
- Increase code quality with additional tasks
- Discover potential deployment issues

## 2.2 Obstacles

- Tough to move an existing system into CI
- Systems that rely on server components
- Db-based systems need to be up-to-dated

# 3 Unit Testing

e introduce
nd the
o weeks:




## 3.1 Good Tests

**Automatic**
...invoking the tests and checking the results

**Thorough**
...test everything that is likely to break ⇒ Code coverage

**Repeatable**
...able to run over and over again, producing same results

**Independent**
...no test relies on an other test

**Professional**
...use same professional standards as for the production code

## 3.2 Test Class

A test class is responsible for testing a unit

- Therefore, usually one test class is responsible for testing a single class
- A test class is a normal Java class. It can have fields, methods, constructors, inner classes...
- If the test class is declared in the same package as the class under test, then it can access its default and protected methods!

## 3.3 Set Up

In order to prepare each test, a setup method is called before each test method

- In the setup method the test harness is initialized.
- The existence of a setup method guarantees, that each test method starts with exactly the same environment.

- A setup method must be public and is marked with the @Before annotation

```
1  @Before
2 public void setup() {
3    ...
4 }
```

## 3.4 Test Method

A unit test class consists of several test methods

- Test methods take no arguments and return nothing, i.e. a test method is void
- Each test method should be completely independent of any other test method
- Often each test method tests a single method of the class under test. IDEs such as Eclipse support this pattern.
- A test method must be public and is marked with the @Test annotation.

```
1 @Test
2 public void testY() {
3    ...
4 }
```

## 3.5 Tear Down

After each test method, a tear down method is called.

- The tear down method allows to tidy up the mess that was produced during the test method.
- The existence of teardown methods guarantees that the outcome of each test method can be cleaned up.
- A teardown method must be public an is marked with the @After annotation

```
1 @After
2 public void teardown() {
3    ...
4 }
```

## 3.6 Assertions

During each test method assertions are used to compare actual against expected results.

**assertEquals(**$<$ *Type* $>$ **expected,** $<$ *Type* $>$ **actual)**

- Compares primitive types by value
- Compares class types by calling equals
- Overloaded with all primitive types, Object and String.

**assertSame(Object expected, Object actual)**
Compares references using ==-operator

**assertNull(Object x), assertTrue(boolean b)**
Does what it says :-)

**fail()**
Unconditional failure of test

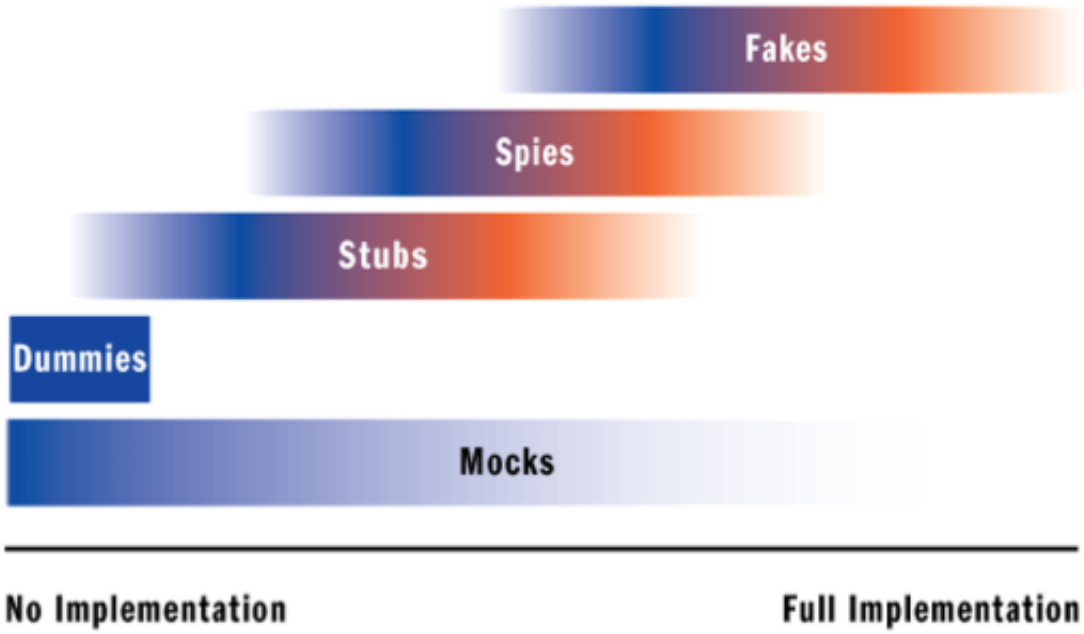
### 3.7 Exception

Exceptions can also be tested. If only one exception is expected:

- Use an expected argument to the @Test annotation
- @Test(expected=IllegalArgumentException.class)

# 4 Isolated Testing

## 4.1 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.

**Spys** 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 which form a specification of the calls they are expected to receive.

## 4.2 Mock Testing

Mock objects simulate parts of the behavior of domain objects. Classes can be tested in isolation by simulating their collaborators with Mock Objects. Takes classes out of a natural environment and puts them in a well defined lab environment.
Use Mock Tests when:

- The real object has non-deterministic behavior.
- 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
- The real object does not yet exist.

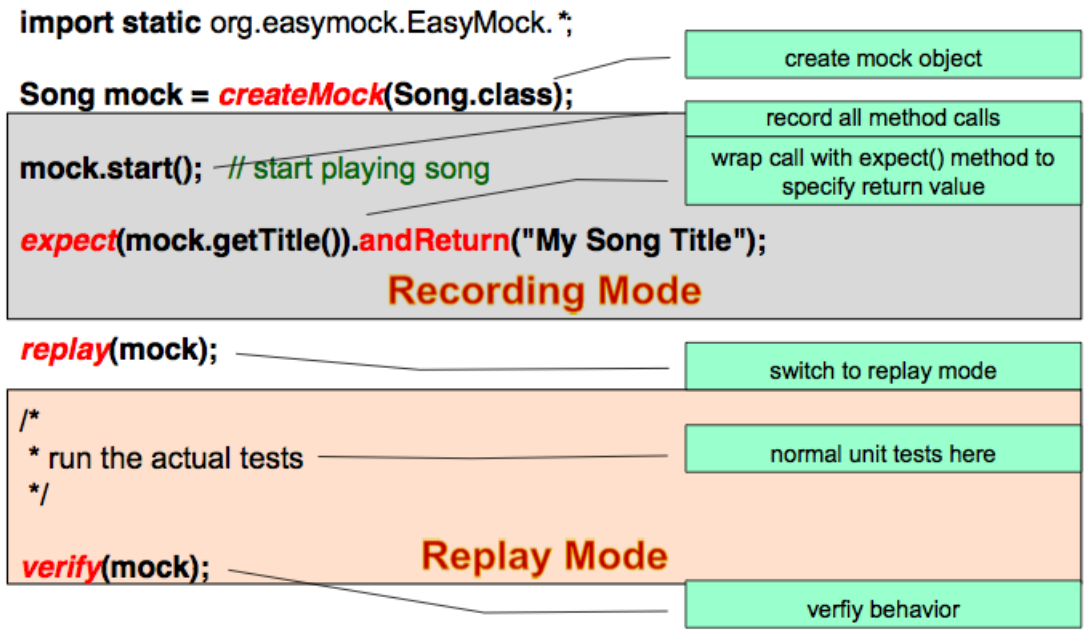
**Pros**

- Enforce the message that testing is about isolation
- Can simplify working with 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 the test suite fragile
- Mocking can become complex with APIs like JDBC

### 4.2.1 EasyMock



# 5 Software Quality Metrics

## 5.1 Software Metrics

**Size Metrics**
Lines of Code(LoC), Number of Statements, Fields, Methods, Packages

**Dataflow Metrics**

- Cyclomatic Complexity (McCabe Complexity)
- Dead Code detection
- Initialization before use

**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

**Performance Metrics (dynamic)**

- Time spent
- Memory consumption (also memory leaks)
- Network traffic
- Bandwidth needed
- No. of clicks to perform a task

## 5.2 Cyclomatic Complexity

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 ... )
$N < 10 \Rightarrow$ code well readable
Criticism: switch-statements are well readable but boost the cyclomatic number

## 5.3 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

Metrics: LCOM (Lack of Cohesion in Method)

### 5.3.1 LCOM

$LCOM^{HS} = \frac{m - avg(r(f))}{m}$

**Where**

- m is the number of methods of a class
- r(f) is the number of methods that access a field f
- Average r(f) over all fields.

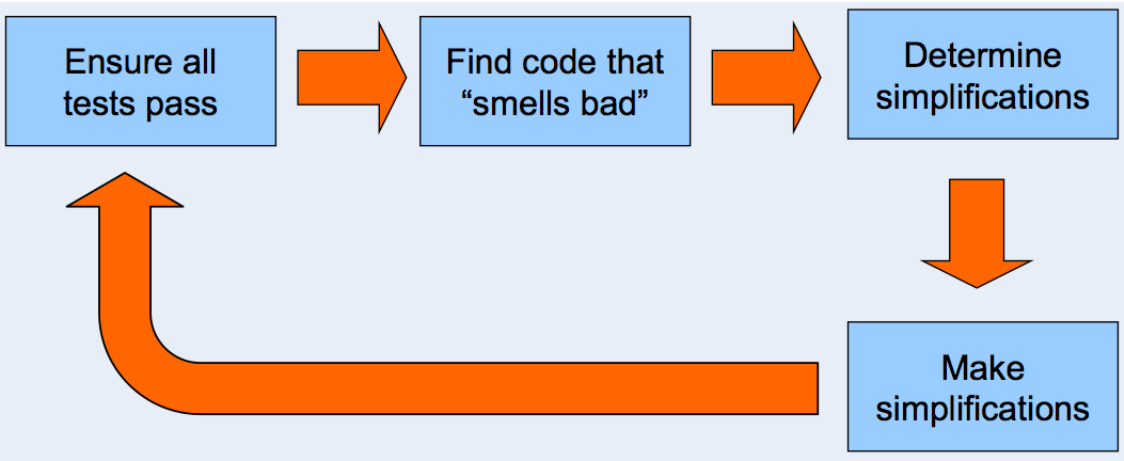
**Examples**

- Each method accesses only one field
  $LCOM^{HS}$ = (almost) 1 low cohesion
  → i.e. getters/setters are bad
- Each method reads all fields
  $LCOM^{HS} = 0$ high cohesion

# 6 Refactoring

- Refactoring improves the design of your system
- Refactoring makes your software easier to understand
- Refactoring helps you to find bugs

**Don't try to add features when wearing the refactoring hat.**




## 6.1 Problems

- Taken too far, refactoring can lead to incessant tinkering with the code, trying to make it perfect
- Databases can be difficult to refactor
- Refactoring published interfaces can cause problems for the code that uses those interfaces

## 6.2 Code Smells

**Too Much Code Smells**

- Duplicated Code
- Long Method
- Large Class
- Long Parameter List
- Feature Envy
- Switch Statements
- Parallel Inheritance Hierarchies

**Not enough Code Smells**

- Empty Catch clause

**Code Change Smells**

- Divergent Change
- Shotgun Surgery

**Comment Smells**

- Need To Comment
- Too much Comments

# 7 Coding Style & Clean Code

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

Code conventions usually cover:

**Filenames and File organization**

- Names of files, directories
- Directory structure of project
- Structure of code files

**Indentation**

- Tabs vs. Spaces
- Line length
- Line Wrapping, line breaking rules

**Naming Conventions**

- Naming conventions make programs more understandable by making them easier to read, to improve and to detect bugs

**Declarations, Statements and White Spaces**

- How many variables shall be declared per line?
- Where shall variables be declared?
- How to indent e.g. a switch statement?
- Where to put block-braces { }?
- Where to insert blank lines?
- Where to insert blank spaces in order to increase readability of expressions?

## 7.1 API Documentation

API documentation is a contract between you and the clients of your code. It specifies the purpose of the class / interface and the functionality of the methods. Provide what is really needed, Remember that anything you provide, you are stuck with debugging, maintaining and updating.

## 7.2 Class/Interface Documentation

- specifies the purpose and responsibility of the class/interface
- specifies important properties that cannot be expressed with the programming language (e.g. immutability, cloneability, singleton)

## 7.3 Method Documentation

- specifies what you expect from the caller (preconditions & parameters)
- specifies what you promise to give the caller in return (postconditions, invariants & return values)

## 7.4 JavaDoc

Javadoc is a separate program that comes with the JDK. It reads your program, makes lists of all the classes, interfaces, methods, and variables, and creates HTML pages displaying its results.
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 well

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

javadoc comments must be immediately before:

- a package (only in package-info.java)
- a class
- an interface
- a constructor
- a method
- a field

### 7.4.1 Tags In javadoc Comments

**@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 SSee Also"heading with a link or text entry that points to reference

**@author** your name

**@version** a version number or date

## 7.5 Programming Practices

**No Magic Numbers**

- Use named constants

**Do not optimize your code**

- Variable assignment. Examples
- Do NOT (try to improve run-time performance): d = (a = b + c) + r;

**Know the boolean type**

**poor:** if ($aBool == true$) { return true; } else { return false; }
**better:** $return(i < m)\&\&(j <= n)$;

**Inappropriate Static**

**poor:** ourlyPayCalculator.calculatePay(employee, overtimeRate)

**better:** Math.max(double a, double b)

**Do one thing**

- A method should do only one thing

**Avoid negative conditionals**

**poor:** if (!buffer.avoidCompaction())

**better:** if (buffer.shouldCompact())

**Avoid Output Arguments**

**poor:** appendFooter(report)

**better:** report.appendFooter()

# 8 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 ca

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

## 8.1 Benefits Using Logging Frameworks

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

## 8.2 Concepts

**Priority** Determines what is logged.

**Level** Determines whether something is logged.

**Appender** Determines where it is logged.

**Layout** Determines how it is logged.

**Logger** Writes a log entry to an appender in a given layout if the appropriate logging level is met by the priority of the log- instruction.

## 8.3 Logging Levels

**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.

**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.