Projektmanagement

10 Build Management und Dokumentation

Vorlesung für AI/MI 4

Prof. Dr. Thomas Allweyer

thomas.allweyer@hs-kl.de

Entscheidungen im Projekt

- **Ordnerstruktur des Projekts**
  - Wo liegen die Quelltexte?
  - Wo liegen die Binaries?
  - Wohin kopiert man Ressourcen wie Konfigurationsdateien, Grafiken
  - Wo liegen die Tests?
  - Wo liegen Bibliotheken, wie benennt man sie?
- **Wie geht man mit Abhängigkeiten (dependencies) um?**
- **Wie benennt man Klassen, Interfaces, Packages, usw.?**
- **Wie wird der Code dokumentiert?**
- **Wie wird Logging durchgeführt?**
- **Wie wird Persistenz umgesetzt?**
- **...**

Nach: Schatten et al, Best Practice Software Engineering. Spektrum Akademischer Verlag 2010.

Hilfsmittel

- **Konventionen**
  - Code-Konventionen
  - Konventionen für Ordnerstrukturen, Dokumentation usw.
- **Tools und Frameworks**
  - IDE (Entwicklungsumgebung, Integrated Development Environment)
    - Syntax-Highlighting, Überprüfung mancher Konventionen, …
  - Code-Quality-Checks
    - Z. B. Checkstyle
  - Dokumentationstools
    - Z. B. Java Doc
  - Logging-Tools
    - Z. B. java.util.logging, log4j
  - Persistenzframeworks
    - Z. B. Hibernate

Build-Management: Warum Automatisierung?

- **Wiederkehrende Aufgaben in der Implementierung**
  - Kompilieren
  - Post- und Preprocessing
  - Dokumentation generieren
  - Deployment auf Test-Server
  - Tests ausführen
  - ...
- **Während der Entwicklung häufig zu wiederholen, z. T. täglich oder öfter**
- **Manuelle Durchführung**
  - Aufwändig
  - Mögliche Fehlerquelle
  - Nicht eindeutig reproduzierbar

Nach: Schatten et al, Best Practice Software Engineering. Spektrum Akademischer Verlag 2010.

Automatisierung des Build-Prozesses

- **Automatisierbare Schritte (Beispiele)**
  - Validierung von Sourcecode
  - Code-Generierung (z. B. Konfigurationsdateien, Deployment-Deskriptor)
  - Automatisierte Code-Quality-Checks
  - Kompilieren
  - Dependency-Management
    - Verwaltung benötigter Bibliotheken (für die Kompilierung, zur Auslieferung)
  - Post-Processing von Binaries (z. B. für Aspekt-orientierte Programmierung)
  - Ausführen von Tests, Erstellen von Test-Reports
  - Generieren von Dokumentation (z. B. Javadoc, Webseiten, …)
  - Zusammenstellen der Software für die Auslieferung
  - Verpacken der Software in Installationsdateien oder Archiven
  - Hochladen auf einen Server

Nach: Schatten et al, Best Practice Software Engineering. Spektrum Akademischer Verlag 2010.

Gradle

- **Was ist Gradle?**
  - Build Management-System auf Basis von Groovy und Kotlin
  - Open Source
  - Unterstützt den automatischen Download und die automatische Konfiguration von benötigten Dateien und Bibliotheken
    - Nutzt hierfür Maven und Ivy Repositories (Maven: anderes Build-Tool, Ivy: Dependency Manager)
  - Unterstützt Multi-Projekt- und Multi-Artefakt-Builds
- **Projekte und Tasks**
  - Ein Gradle Build umfasst ein oder mehrere Projekte
  - Projekte bestehen aus Tasks
    - Beispiele für Tasks: Kompilieren, Javadoc generieren

Quelle: http://www.vogella.com/tutorials/Gradle/article.html

Tools

- **Bedienung von Gradle erfolgt standardmäßig über die Kommandozeile**
  - Z. B. ruft

    ```
    gradle build
    ```

    einen Task namens „build“ auf
- **Eclipse-Integration**
  - Aktuelle Eclipse-Versionen enthalten bereits die sogenannten „Buildship“-Tools zur Gradle-Unterstützung

Gradle in Eclipse (1)

- **Unterstützte Java-Versionen**
  - Zur Unterstützung neuerer Java-Versionen muss eine aktuelle Gradle-Version verwendet werden
  - Im Folgenden wird Gradle 7.4 und Java-Version 17 verwendet

- **Gradle-Projekt für die Java-Entwicklung erstellen:**
  - Zunächst ein reguläres Java-Projekt anlegen
  - Keine Leer- und Sonderzeichen im Projektnamen, Beginn am besten mit Kleinbuchstaben
  - Über Kontextmenü mit „Configure > Add Gradle Nature“ zum Gradle-Projekt machen
  - Über „Window > Show View > Other > Gradle > Gradle Tasks die Gradle Task View“ auswählen
  - In der Gradle Task View den Task „*Projektname*/build setup/init“ auswählen und über das Kontextmenü mit „Run Gradle Tasks“ ausführen
  - In die View „Console“ wechseln. Dort werden verschiedene Angaben abgefragt. Diese folgendermaßen beantworten:
    - Type of project: 2 (application)
    - Implementation language: 3 (Java)
    - Split functionality: 1 (no)
    - Build script DSL: 1 (Groovy)
    - Generate build using new APIs … einfach Enter drücken (no)
    - Test framework: 4 (Junit Jupiter)
    - Project name … einfach Enter drücken (vorhandener Projektname bleibt)
    - Source package… einfach Enter drücken (Package wird wie das Projekt benannt)
  - Im Kontextmenü des Projektes: „Gradle > Refresh Gradle Project“ auswählen.
  - Nun sind zwei Projekte zu sehen: Eines mit dem Projektnamen und ein zweites (Unterprojekt) namens „app“ (oder „Projektname-app)

Gradle in Eclipse (2)

- **Gradle Task View**
  - Hier sieht man die für ein Projekt verfügbaren Tasks und kann sie über das Kontextmenü ausführen
    - Z. B. „build“
    - Die Bearbeitung kann man in der View „Gradle Executions“ und ggf. in der Konsole verfolgen
- **Refresh Gradle**
  - Wenn sich die Datei „build.gradle“ ändert, muss man über das Kontext-Menü des Projekts „Gradle>Refresh Gradle Project“ aufrufen, damit die Änderungen wirksam werden
  - Oder unter Window>Preferences>Gradle „Automatic Project Synchronization“ auswählen
  - Falls neue Tasks hinzugekommen sind: „Refresh“ in der Gradle Task View

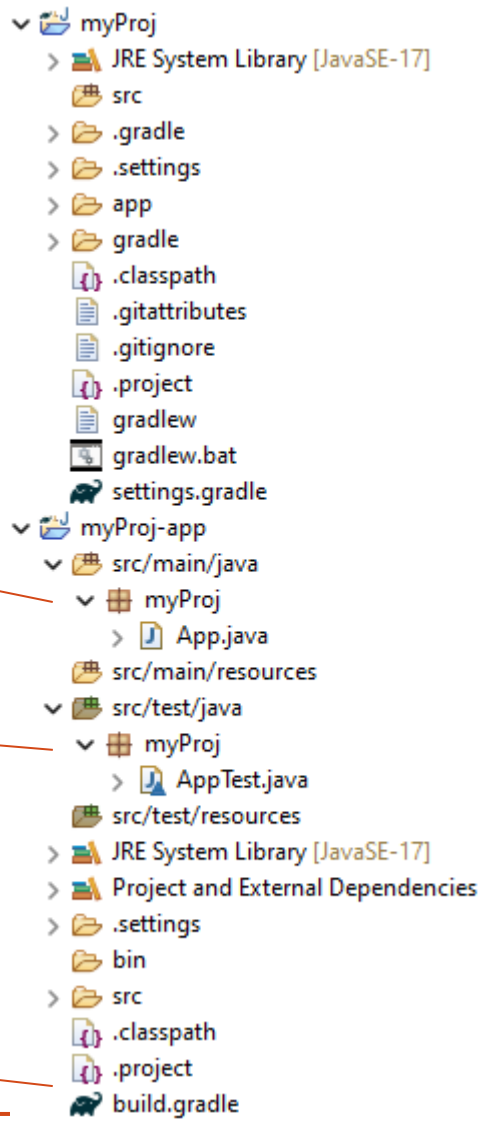
Gradle in Eclipse (2)

- **Gradle setzt eine bestimmte Struktur für Java-Projekte voraus**
  - Diese wurde durch die Ausführung des Tasks „init“ angelegt

Hier schreibt man seinen Quellcode hinein

Hier schreibt man seine Tests hinein

„build.gradle“ – Datei zur Beschreibung des build-Prozesses



Gradle Wrapper

- **Was ist ein Gradle Wrapper?**
  - Ein Skript, das eine festgelegte Gradle-Version ausführt (und sie hierfür ggf. herunterlädt).
  - gradlew (Shell Script), gradlew.bat (Windows Batch File)
  - Wird auf der Kommandozeile genauso ausgeführt wie der Befehl „gradle“, also z. B. „`gradlew build`“ statt „`gradle build`“
- **Vorteile**
  - Andere Entwickler können Gradle Tasks ausführen ohne eine bestimmte Gradle-Version installieren und einrichten zu müssen
  - Die build-Vorgänge für das Projekt sind standardisiert und werden daher von allen genau gleich durchgeführt.
    - Daher sollte man Gradle in der Regel über den Wrapper nutzen
- **Zu jedem Gradle-Projekt kann man einen Gradle Wrapper anlegen**
  - Hierfür gibt es einen Task „wrapper“

Gradle-Tasks mit Parametern

- **Bei vielen Gradle-Tasks kann man Parameter angeben**
  - Z. B. für den Task „help“. Man erhält mit

    ```
    gradle help --task build
    ```

    Informationen über den build-Task

- **Eclipse unterstützt nur manche Parameter**
  - Beim Start aus Eclipse lassen sich über das Kontextmenü mittels „Run Configuration“ Argumente eintragen
  - Bei vielen Parametern funktioniert das nicht

- **Lösung**
  - Kommandozeile im Projektverzeichnis öffnen und „gradlew“ verwenden:

    ```
    gradlew help --task build
    .\gradlew help --task build
    ```
    (in PowerShell)

  - Dann ist keine separate Gradle-Installation erforderlich

Build-Datei

- **Der Build-Prozess wird in einer Datei „build.gradle“ beschrieben**
  - Im Wurzelverzeichnis des Projekts
  - Hierfür wird eine eigene Syntax verwendet, ggf. kann man zur Ergänzung auch Groovy- (bzw. auch gewöhnliches Java) oder Kotlin-Code nutzen

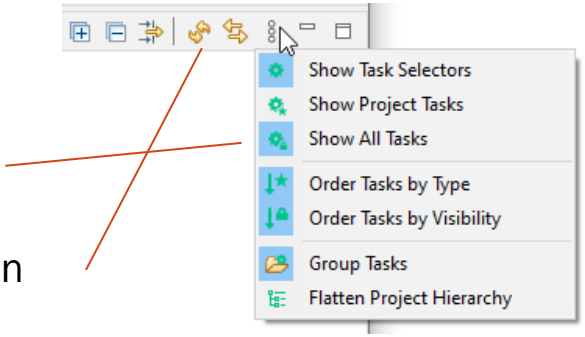
Definition eines Tasks

- **Beispiel eines Tasks (in der Datei „build.gradle“)**

```
task hello {
        doLast {
                println 'Hello Gradle'
        }
}
```

Definiert letzte Aktionen in dem Task
(hier gibt es allerdings keine vorhergehenden)

- **Damit der Task „hello“ in der Task-View sichtbar wird:**
  - Es muss im View Menu „Show All Tasks“ ausgewählt sein
  - Über das Kontext-Menü des Projekts „Gradle > Refresh Gradle Project“ aufrufen
  - Anschließend in der Gradle Task View ein Refresh durchführen
  - Der Task erscheint dann unter „other“



- **Nun kann man den Task „hello“ ausführen und die Ausgabe in der Konsole ansehen**

Definition von Tasks (2)

- Kommentare wie in Java:
  // bzw. /* */
- Tasks können Beschreibungen enthalten und in Gruppen eingeordnet werden
  - Gruppen werden in Gradle Task View als Ordner dargestellt.

```
/*
* Task "hello" definieren
*/

task hello {
	group = 'my group' // Task erscheint in Gruppe "my group"

	description = 'Says hello'
	doLast {
		println 'Hello Gradle'
	}
 }
```

Verwendung von Programm-Code

- In einem Task kann gewöhnlicher Groovy- (und somit auch Java-) Code verwendet werden

```
task sayHelloInUpperCase(){

      doLast {
            String s = "hello"
            println(s.toUpperCase())
      }
}
```

Vordefinierte Task-Typen

- **Es gibt zahlreiche vordefinierte Task-Typen,**
  - z. B. zum Kompilieren, Testen, Hochladen in Repositories, Arbeiten mit Dateien usw.
  - Meist bietet jeder Task-Typ viele Einstellungs- und Parametrisierungsmöglichkeiten
    - Siehe Gradle-Dokumentation https://docs.gradle.org/current/dsl/
  - Nutzung der Task-Typen für eigene Tasks:

```
task meinTask(type: <TaskType>){
}
```

- **Beispiele für Task-Typen zur Arbeit mit Dateien**
  - Copy
  - Delete
  - Zip

Beispiele für Copy, Delete, Zip

Komplettes Verzeichnis kopieren

```
task copyFromAtoB(type: Copy){
      from file("examples/A")
      into file("examples/B")
 }

 task copyFileFromAtoB(type: Copy){
      from file("examples/A/file1.txt")
      into file("examples/B")
 }


 task copyDocsFromAtoB(type: Copy){
      from file("examples/A")
      include "*.docx"
      into file("examples/B")
 }
```

Filter: Nur die Dateien mit Endung .docx kopieren

```
task getRidOfB(type: Delete){
      delete("examples/B")
}

task archiveA(type: Zip){
      archiveFileName = "A.zip"
      destinationDirectory = file("examples")
      from("examples/A")
}
```

Weitere Beispiele:
https://docs.gradle.org/current/userguide/working_with_files.html

Task Dependencies (Abhängigkeiten)

```
task task1(){
        doLast {
                println("Task 1")
        }
}

task task2(){
        dependsOn(task1)

        doLast {
                println("Task 2")
        }
}
```

- Tasks können voneinander abhängig sein
- task2 ist von task1 abhängig
  - D. h. damit task2 ausgeführt werden kann, muss vorher task1 ausgeführt worden sein
- Gradle kümmert sich selbst darum, dass die Tasks in der richtigen Reihenfolge ausgeführt werden
  - Auch über mehrere Ebenen (z. B. könnte task1 wiederum von einem anderen Task abhängig sein usw.)
- Es handelt sich also um eine deklarative Beschreibung
  - Man beschreibt nicht, in welcher Reihenfolge die Tasks ausgeführt werden müssen, sondern, was für die einzelnen Tasks erforderlich ist
- Vorteile der deklarativen Beschreibung
  - Falls das Ergebnis eines Tasks bereits vorliegt und sich nichts geändert hat, braucht er nicht erneut durchgeführt werden. Gradle kümmert sich um diese Zusammenhänge.
  - Es ist egal, in welcher Reihenfolge die Tasks in der build-Datei definiert werden. Dies ist insbesondere nützlich, wenn in größeren Builds mehrere build-Dateien integriert werden.

Konfiguration und Ausführung

```
task myTask(){
        println("Configuring ...")

        doFirst {
                println("First actions ...")
        }

        doLast {
                println("Last actions ...")
        }
}
```

- **Konfiguration**
  - Anweisungen, die hier stehen (außerhalb von doFirst( ) und doLast( )), werden zu Beginn immer ausgeführt (für jeden Task, auch wenn er gar nicht aufgerufen wird).
  - In der Konfigurationsphase analysiert Gradle alle Abhängigkeiten und stellt die auszuführenden Tasks zusammen. Mit Hilfe von Anweisungen im Konfigurations-Teil kann dies beeinflusst werden.
- **Ausführung**
  - Für jeden ausgeführten Task werden zu Beginn die Anweisungen in „doFirst( )“ und am Ende die Anweisungen in „doLast( )“ ausgeführt.

doFirst( ) und doLast( )

- **Wieso unterscheidet man „doFirst( )“ und „doLast( )“?**
  - Man kann vordefinierten oder importierten Tasks noch Verhalten hinzufügen
  - Z. B. kann man einen Task vom Typ „Copy“ um Aktionen erweitern, die vor bzw. nach dem eigentlichen Kopieren erfolgen:

```
task copyMyTexts(type: Copy){
    from file("examples/A")
    include "*.txt"
    into file("${buildDir}/myOutput")

    println("Configuring ...")

     doFirst {
          println("Before copying ...")
    }

    doLast {
          println("After copying ...")
    }
}
```

- Und auch später kann man zu einem bereits existierenden (z. B. von Gradle vordefinierten) Task noch etwas hinzufügen, was entweder zu Beginn oder Ende eingefügt wird:

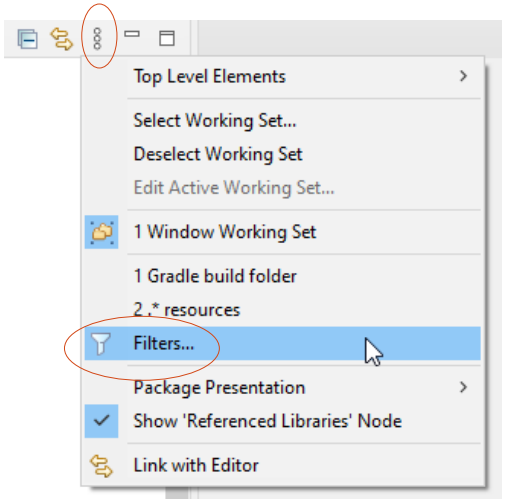
```
copyMyTexts.doLast{
      println("Something else ...")
 }
```

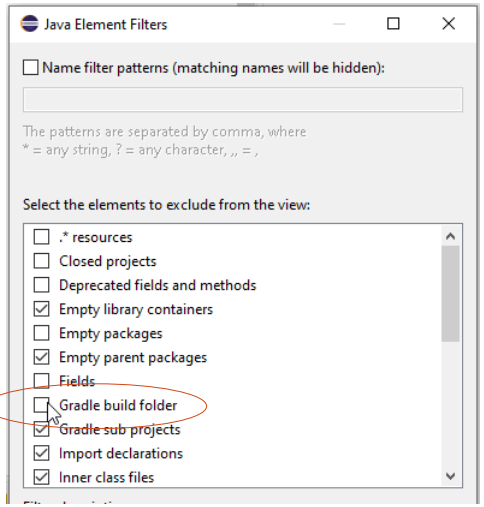
Build-Verzeichnis

- Standardmäßig erzeugt Gradle Outputs in einem Ordner „build“
- Die Variable „buildDir“ enthält den Pfad des build-Ordners
  - Verwendung in einem String mit ${buildDir}

```
task copyMyTexts(type: Copy){
    from file("examples/A")
    include "*.txt"
    into file("${buildDir}/myOutput")

    println("Configuring ...")

     doFirst {
          println("Before copying ...")
    }

    doLast {
          println("After copying ...")
    }
}
```

- Einblenden des von Gradle standardmäßig verwendeten Ordners „build“ in Eclipse
  - Im Package Explorer
    - Häkchen bei „Gradle build folder“ entfernen





Aufgabe

- **Schreiben Sie vier Tasks: task1, task2, task3 und task4**
  - Jeder Task soll während der Konfiguration eine Ausgabe der Form „Konfiguration Task Nr. x“ erstellen (x steht für die Nummer des Tasks)
  - Zu Beginn jedes Tasks soll „Starte Task Nr. x“ ausgegeben werden
  - Zum Ende jedes Tasks soll „Beende Task Nr. x“ ausgegeben werden
  - task2 hängt von task1 ab
  - task4 hängt von task2 und task3 ab
- **Überlegen Sie, was beim Ausführen jedes der erstellten Tasks ausgegeben wird und überprüfen Sie es.**

Plugins

- Plugins enthalten Sammlungen vorgefertigter Tasks für bestimmte Aufgaben
- Beispiele für Plugins für die Java-Entwicklung
  - base
    - clean (erzeugte Dateien löschen), build, check, ...
  - java (schließt base mit ein)
    - Tasks für Kompilieren, Testen, Javadoc, Archive erstellen, ...
  - application (schließt java mit ein)
    - Angabe eines mainClassName (Klasse mit main-Methode, kann über einen Task gestartet werden)
- Nutzung eines Plugins in einer build-Datei:

```
plugins {
        id 'java'
}
```

Externe Dependencies

- Mit Hilfe des Build-Skripts kann man auch externe Abhängigkeiten beschreiben, d.h. man legt fest, wo benötigte Klassen, Dateien, Bibliotheken gesucht werden.
- Dabei können benötigte Bibliotheken mitsamt ihren Abhängigkeiten automatisch heruntergeladen werden.
- Angabe von Dependencies in build.gradle:

Abhängigkeit von lokalen Dateien

```
dependencies {
    implementation fileTree(dir: 'libs', include: ['*.jar'])
    implementation 'com.squareup.okhttp:okhttp:2.5.0'
    testImplementation 'junit:junit:4.12'
}
```

Abhängigkeit von Bibliotheken aus einem Repository

Beispiele vordefinierter Konfigurationen (umfassen verschiedene Tasks) – hier Konfigurationen aus dem Java-Plugin .
„implementation“ betrifft Tasks wie „compileJava“, „testImplementation“ betrifft Tasks wie „test“

Repositories

- **Festlegung des Repositories, in dem sich die angegebenen Bibliotheken finden:**

```
repositories {
    jcenter()
}
```

- **Angabe von Repositories**
  - JCenter ist ein öffentliches Repository mit einer umfangreichen Sammlung von Open Source-Bibliotheken, vor allem aus dem Java- und Android-Umfeld.
  - Daneben gibt es noch andere, z. B. mavenCentral.
  - Für diese bekannten Repositories ist keine URL nötig, ansonsten muss eine URL angegeben werden.
  - Es lassen sich auch mehrere Repositories angeben.

- **Cache**
  - Beim ersten Ausführen eines builds werden die benötigten Dateien heruntergeladen und in einen Cache geschrieben, so dass sie nicht jedes Mal neu heruntergeladen müssen.

Java-Projekte – Struktur

- **Typische Struktur**
  - Java Source-Code unter src/main/java
  - Java Tests unter src/test/java
- **Andere Strukturen können in build.gradle definiert werden**
  - Im allgemeinen sollte man die vorgegebene Struktur so verwenden
  - Die Standard-Plugins erwarten diese Struktur
- **Angabe von Version und Java-Version**

```
version = 0.1.0
sourceCompatibility = 1.8
```

Versionsnummer des Projekts, wird z. B. beim Erstellen von jar-Files in den Dateinamen aufgenommen, z. B. myProject-0.1.0.jar

Code ist zur Java-Version 1.8 kompatibel

Aufgabe (1)

- Erstellen Sie eine Klasse mit einer einfachen Methode, z. B. zum Addieren dreier Zahlen
- Erstellen Sie eine Testklasse mit zwei Tests für die obige Methode
- Führen Sie den Task „build“ aus
- Sehen Sie sich auf der Konsole an, welche Tasks ausgeführt wurden
- Sehen Sie sich unter build/reports/tests/test/index.html den Report mit den Testergebnissen an

Aufgabe (2)

- Es soll nun die joda-Time-Bibliothek verwendet werden.
- Erweitern Sie die Klasse „App“:

```
package myProj;

import org.joda.time.LocalTime;

public class App {
    public String getGreeting() {
        return "Hello World!";
    }

    public static void main(String[] args) {
        System.out.println(new App().getGreeting());
            LocalTime currentTime = new LocalTime();
            System.out.println("The time is " + currentTime);
    }
}
```

Aufgabe (3)

- Ergänzen Sie die build.gradle-Datei um den folgenden Eintrag (in „dependencies“):

```
implementation 'joda-time:joda-time:2.12.1'
```

- Führen Sie den Task „run“ aus.

Anmerkung: Unter https://www.joda.org/joda-time/dependency-info.html wird *compile 'joda-time:joda-time:2.12.1'* angegeben. Den Task „compile“ gibt es aber in aktuellen gradle-Versionen nicht mehr. Daher wird „implementation“ verwendet.

Aufgabe (4)

- Fügen Sie noch Java-Doc-Kommentare hinzu

```java
package myProj;

import org.joda.time.LocalTime;

/**
 * This is the main class.
 * @author John Doe
 *
 */
public class App {

      /**
       * Method for greeting
       * @return a String with a Greeting
       */
    public String getGreeting() {
        return "Hello World!";
    }

    /**
     * Main method - prints the greeting and the current Time
     */
    public static void main(String[] args) {
        System.out.println(new App().getGreeting());
             LocalTime currentTime = new LocalTime();
             System.out.println("The time is " + currentTime);
    }
}
```

```java
/**
 * A calculator class
 * @author John Doe
 *
 */
public class Calculator {

      /**
       * Calculates the sum of three Integers
       * @param a first Integer
       * @param b second Integer
       * @param c third Integer
       * @return the sum
       */
      public int sum(int a, int b, int c) {
            return a+b+c;
      }
}
```

- Führen Sie den Task javadoc aus
  - Sehen Sie sich das Ergebnis unter build/docs/javadoc/index.html an

Aufgabe (5)

- Bei dem Task „build“ wird „javadoc“ nicht automatisch mit ausgeführt.
- Schreiben Sie einen Task „buildWithDoc“, bei dessen Aufruf sowohl build also auch javadoc ausgeführt werden.

Hinweis

- **Falls es bei Javadoc Probleme mit Umlauten gibt:**
  - In build.gradle Folgendes einfügen

```
javadoc {
      options.encoding = 'UTF-8'
      options.docEncoding = 'UTF-8'
      options.charSet = 'UTF-8'
}
```

Aufgabe (6)

- **Es soll noch das vordefinierte Plugin „checkstyle“ eingebunden werden. Damit kann die Einhaltung von Code-Konventionen überprüft werden**

```
plugins {
    id 'application'
    id 'checkstyle'
}
```

  - Weitere Angaben: Checkstyle-Version, kein Abbruch bei Fehlern, Pfad zur Konfigurationsdatei (mit den zu überprüfenden Regeln):

```
checkstyle{
      toolVersion '9.3'
      ignoreFailures = true
      configFile = file("${rootDir}/config/checkstyle/google_checks.xml")
}
```

  - Legen Sie die zur Verfügung gestellte Konfigurationsdatei „google_checks.xml“ in Ihrem Projekt in einen Ordner config/checkstyle.
  - Führen Sie erneut einen build aus (oder direkt den Task checkstyleMain).
  - Sehen Sie sich die Ergebnisse des Checks an

Continuous Integration - Prinzipien

- Nur ein Sourcecode-Repository im Projekt, mit dem alle arbeiten
- Automatisierter Build
- Automatisierte Tests
- Jeder Entwickler sollte mindestens einmal täglich seine Änderungen in das Repository committen
- Der Build sollte schnell ablaufen, so dass häufige Builds möglich sind
  - Z. B. nur geänderte Sourcen kompilieren
- Automatisiertes Deployment
- Der Build sollte wenigstens einmal täglich auf einem neutralen Rechner automatisiert ausgeführt werden
  - Neutraler Rechner: Kein Entwicklungsrechner, entspricht dem Zielsystem
- Die Build- und Testergebnisse werden z. B. auf einer generierten Intranetseite dem gesamten Projekt zur Verfügung gestellt

Nach: Schatten et al, Best Practice Software Engineering. Spektrum Akademischer Verlag 2010 und http://martinfowler.com/articles/continuousIntegration.html

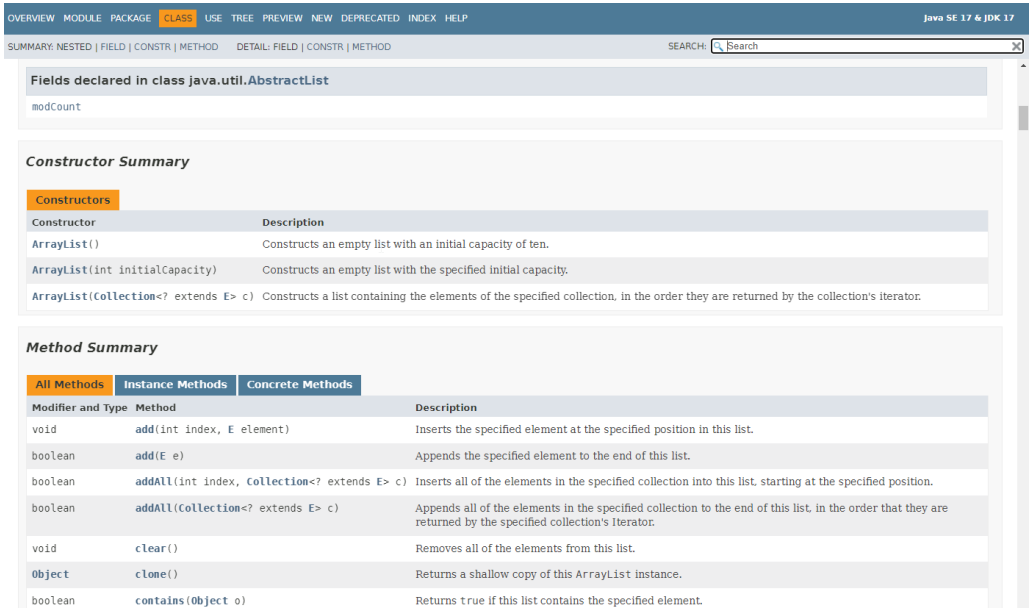
Dokumentation mit Javadoc

- Nutzen Sie die Möglichkeiten von Dokumentationstools wie Javadoc
- Verwendung von Doc Comments im Quelltext

```
/**
* Hier steht der Kommentar, der
* mehrere Zeilen überspannen kann.
*/


/** Einzeiliger Kommentar */
```

Berücksichtigte Dateien

- **Javadoc erstellt Dokumentationen aus**
  - Quellcodedateien
  - Package-Beschreibungen
  - Overview-Dateien („overview.html“)
  - Diverse weitere Dateien
- **Erzeugt eine strukturierte HTML-Dokumentation, wie sie von der Java API-Dokumentation bekannt ist.**



Quellcodedateien

- Doc Comments können vor
  - Klassen
  - Attributen
  - Konstruktoren
  - Methoden

  stehen
- Sie müssen aber direkt vor der jeweiligen Deklaration stehen

Dokumentation von packages

- In einer Datei package-info.java (oder package.html)
- Beispiel:

Ein Doc Kommentar besteht aus einer Beschreibung gefolgt von einer Tag-Section. Beide sind optional, können aber nur einmal und nur in dieser Reihenfolge vorkommen.

```
/**
 * Provides the classes necessary to create an
 * applet and the classes an applet uses
 * to communicate with its applet context.
 * <p>
 * The applet framework involves two entities:
 * the applet and the applet context.
 * An applet is an embeddable window (see the
 * {@link java.awt.Panel} class) with a few extra
 * methods that the applet context can use to
 * initialize, start, and stop the applet.
 *
 * @since 1.0
 * @see java.awt
 */
package java.lang.applet;
```

Der erste Satz der Beschreibung ist wichtig. Er erscheint in der Übersichtsdarstellung-

Nutzung von HTML-Tags

Inline-Tag

Tag-Section mit Block Tags

Package-Deklaration

Quelle: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html

Dokumentation von Klassen

```
/**
 * A class representing a window on the screen.
 * For example:
 * <pre>
 *    Window win = new Window(parent);
 *    win.show();
 * </pre>
 *
 * @author  Sami Shaio
 * @version 1.15, 13 Dec 2006
 * @see     java.awt.BaseWindow
 * @see     java.awt.Button
 */
class Window extends BaseWindow {
   ...
}
```

Quelle: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html

Dokumentation von Methoden

- Wichtig ist die Dokumentation von öffentlichen Methoden, die von anderen Entwicklern genutzt werden sollen. Sie sollten anhand der Beschreibung erkennen können, was man der Methode übergeben muss, was sie tut, und was sie zurückgibt.

Für jeden Parameter:
Name und Beschreibung

Beschreibung der Rückgabe

Welche Exceptions werden unter welchen Bedingungen geworfen?

```
/**
 * Returns the character at the specified index. An index
 * ranges from <code>0</code> to <code>length() - 1</code>.
 *
 * @param     index  the index of the desired character.
 * @return    the desired character.
 * @exception StringIndexOutOfRangeException
 *              if the index is not in the range <code>0</code>
 *              to <code>length()-1</code>.
 * @see       java.lang.Character#charValue()
 */
public char charAt(int index) {
   ...
  }
```

Quelle: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html

Dokumentation von Attributen

```
/**
* The X-coordinate of the component.
*
* @see #getLocation()
*/
int x = 1263732;
```

Quelle: http://docs.oracle.com/javase/7/docs/technotes/tools/windows/javadoc.html

Javadoc Tags (Auswahl)

@author *name*	Autor(en)
@deprecated *text*	Das Element ist veraltet und sollte nicht mehr benutzt werden
@exception *klassenname beschreibung*	Exception und unter welchen Bedingungen sie geworfen wird
@param *name beschreibung*	Parameter mit Name und Beschreibung der Bedeutung und erlaubter Werte
@return *beschreibung*	Rückgabewert mit Beschreibung der Bedeutung und möglicher Werte
@see *reference*	Verweis. Mögliche Angaben: String, url, package.class#methode
@since *versions-bezeichnung*	Seit dieser Version ist das Element in der Software enthalten
@throws *klassenname beschreibung*	Synonym zu exception
@version *versions-bezeichnung*	Version der Software, zu der der Code gehört
{@link *package.class#member label*}	Ähnlich @see, aber inline-Link im Text

Aufgabe

- Gegeben ist die folgende Methode

```
public String wandleZahlInString(int zahl, int laenge) throws WertebereichException {
    if(zahl>=0){
        String ergebnis = String.valueOf(zahl);
        int urspruenglicheLaenge = ergebnis.length();
        if(urspruenglicheLaenge<=laenge){
            for (int i=urspruenglicheLaenge+1; i<=laenge; i++){
                ergebnis = "0" + ergebnis;
             }
             return ergebnis;
         } else {
             throw new WertebereichException("Zahl ist länger als angegebene Länge " + laenge);
         }
    } else {
        throw new WertebereichException("Negative Zahl");
    }
}
```

- Ermitteln Sie, was die Methode genau macht und schreiben Sie einen Javadoc-Kommentar dazu.

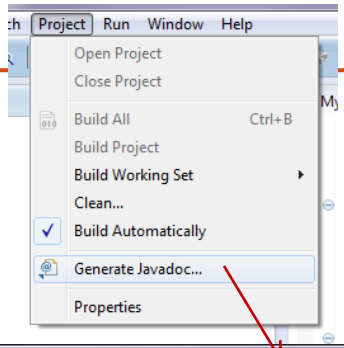
Javadoc-Unterstützung in Eclipse

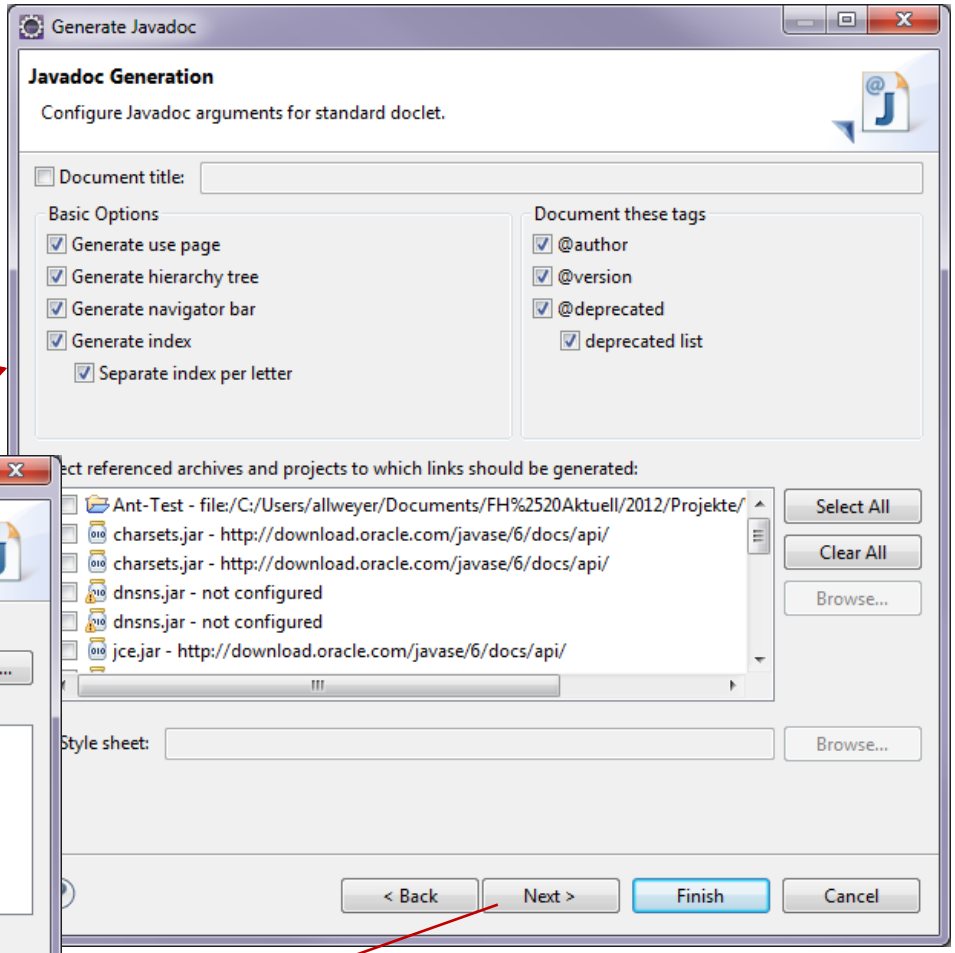
Automatisch generierte Kommentar-Gerüste

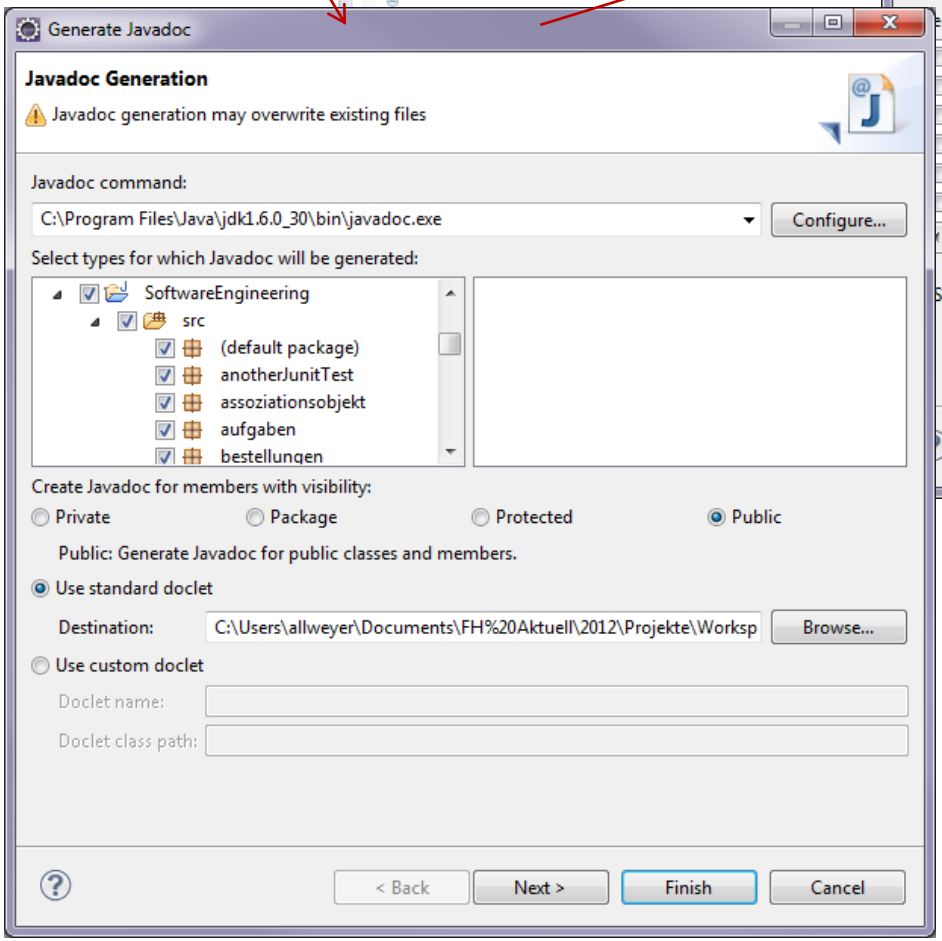
```
/**
 * @author allweyer
 *
 */
public class Sums {

    /**
     * @param n1
     * @param n2
     * @return
     */
    public int add(int n1, int n2){
        return n1 + n2;
    }
}
```

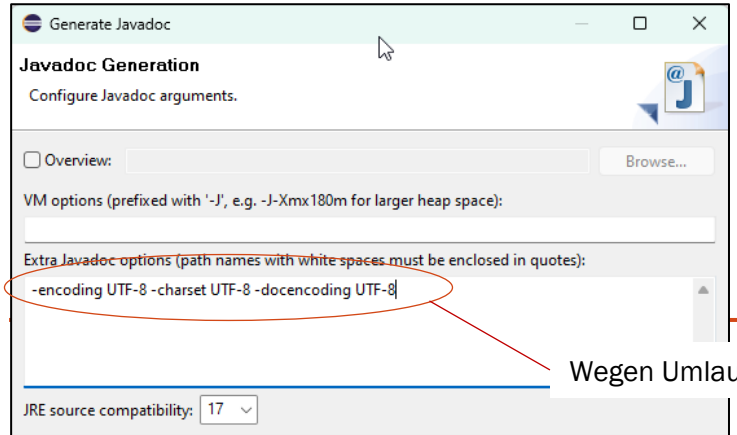
- Kommentar-Templates unter Preferences > Java > Code Style > Code Templates > Comments
- Erzeugen von Attribut- oder Methoden-Kommentar-Gerüsten über das Kontext-Menü Source > Generate Element Comment (Alt Shift-J)

Javadoc-Generierung in Eclipse





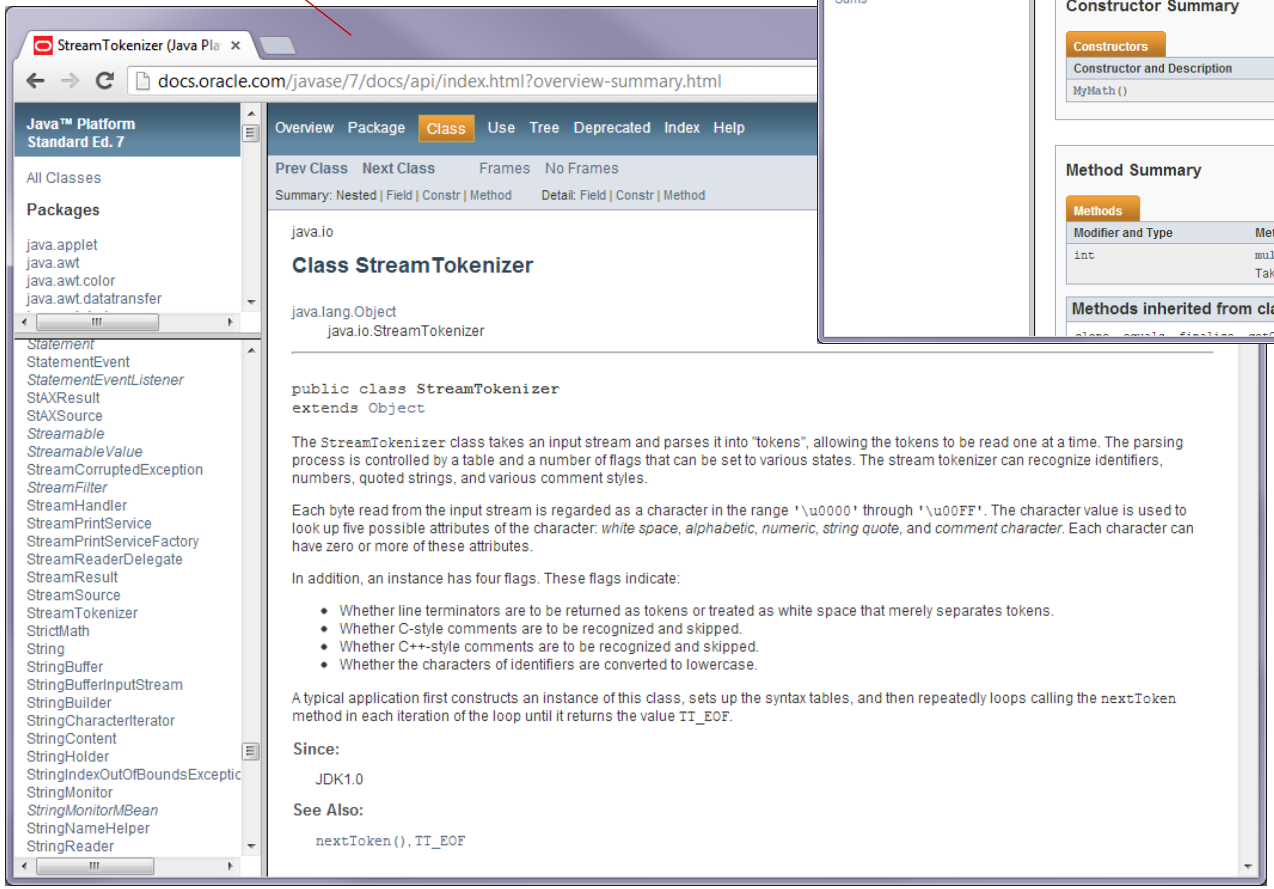


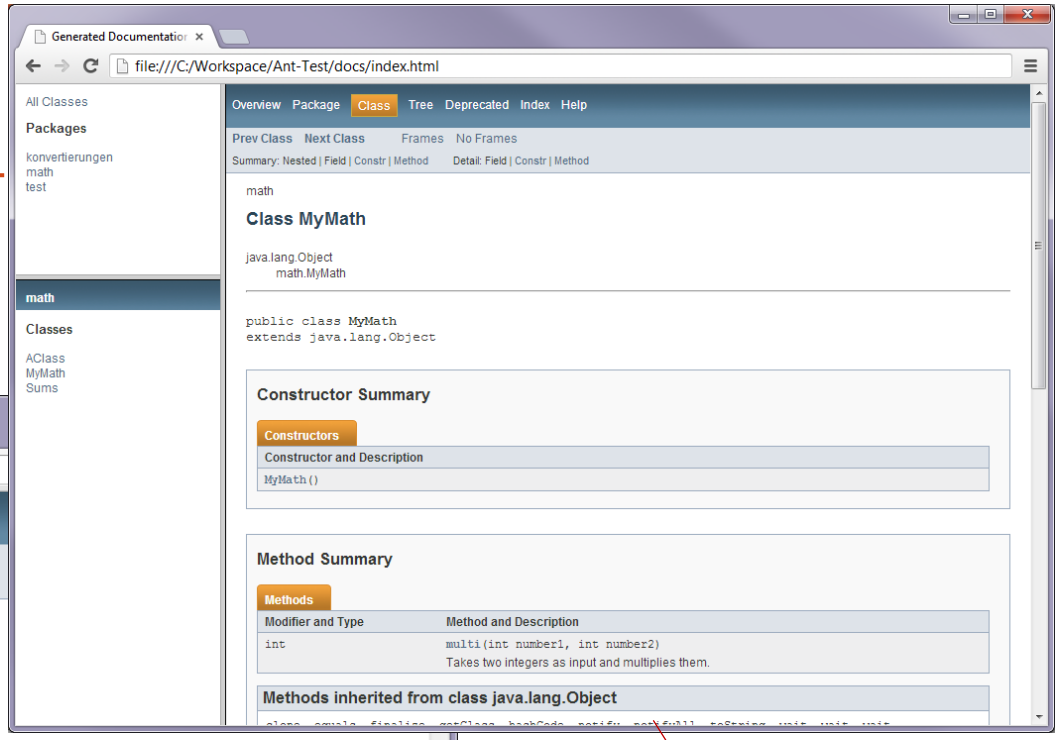


Wegen Umlauten

Javadoc

Java API Dokumentation





Selbst generiertes Javadoc