technische universität
dortmund

Software- Engineering für langlebige Systeme

Thomas Ruhroth | Dortmund
SoSe 2015

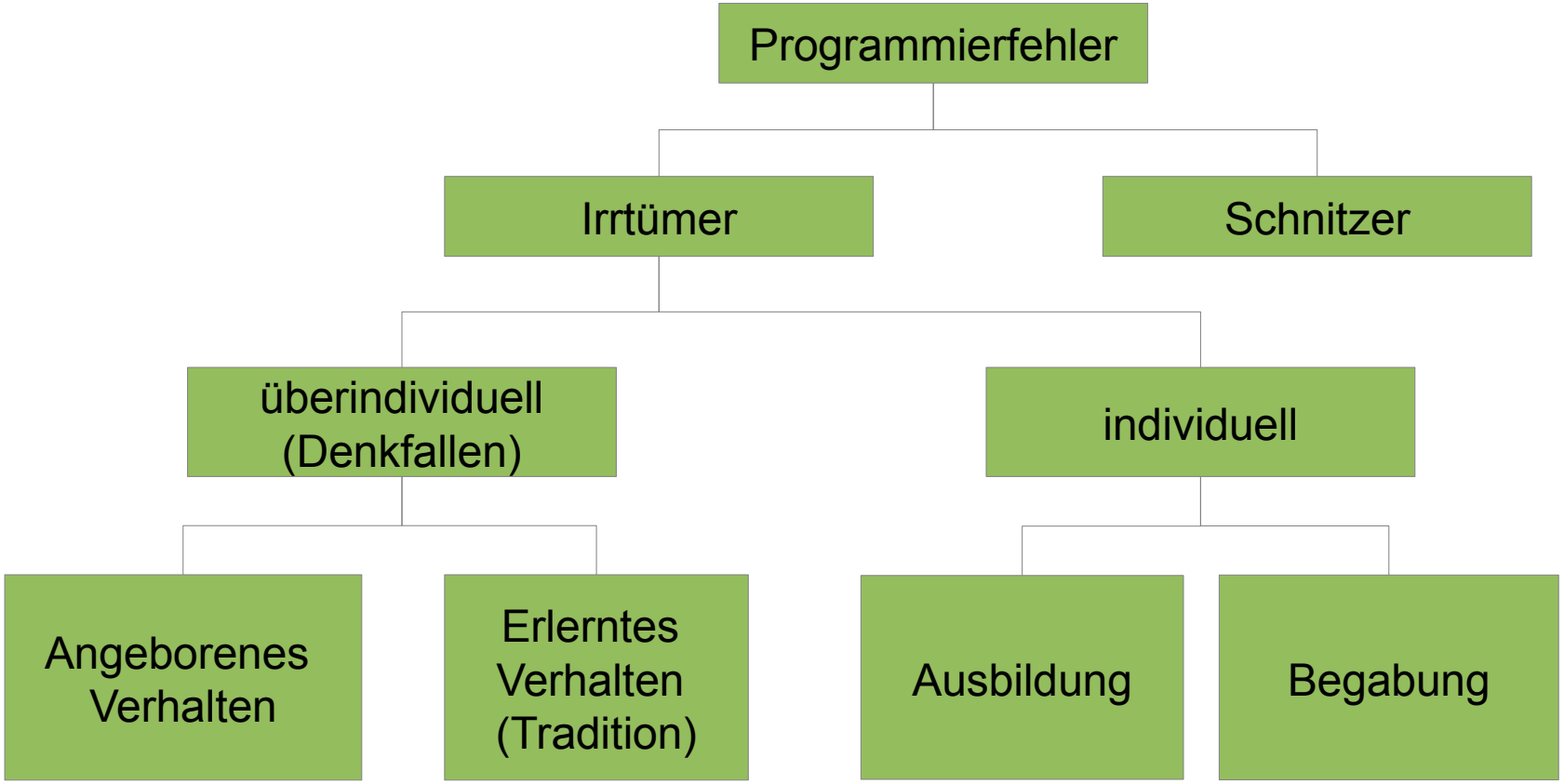
Softwarewartung auf Code-Ebene
-
Fortsetzung

VL3.2

- Kurze Wiederholung
- Dokumentation
- Hypes – Vorsicht!
- Automatisierung

- Ziele:
  - Verstehen der Zusammenhänge der Dokumentation mit Effekten der langlebigen Systeme
  - Hypes richtig bewerten können.

Programmierfehler [Gram90]



Hintergrundwissen [Balz]

Jeder Mensch benutzt bei seinen Handlungen bewusst oder unbewusst angeborenes oder erlerntes Hintergrundwissen.

Hintergrundwissen gemeinsames Wissen einer Bevölkerungsgruppe, z.B. den Programmierern

Programmierfehler lassen sich nun danach klassifizieren, ob das Hintergrundwissen für die Erledigung der Aufgabe angemessen ist oder nicht.

Einstellungen [Balz]

Einstellungen führen zu einer vorgeprägten Ausrichtung des Denkens. Sie können zurückgehen auf

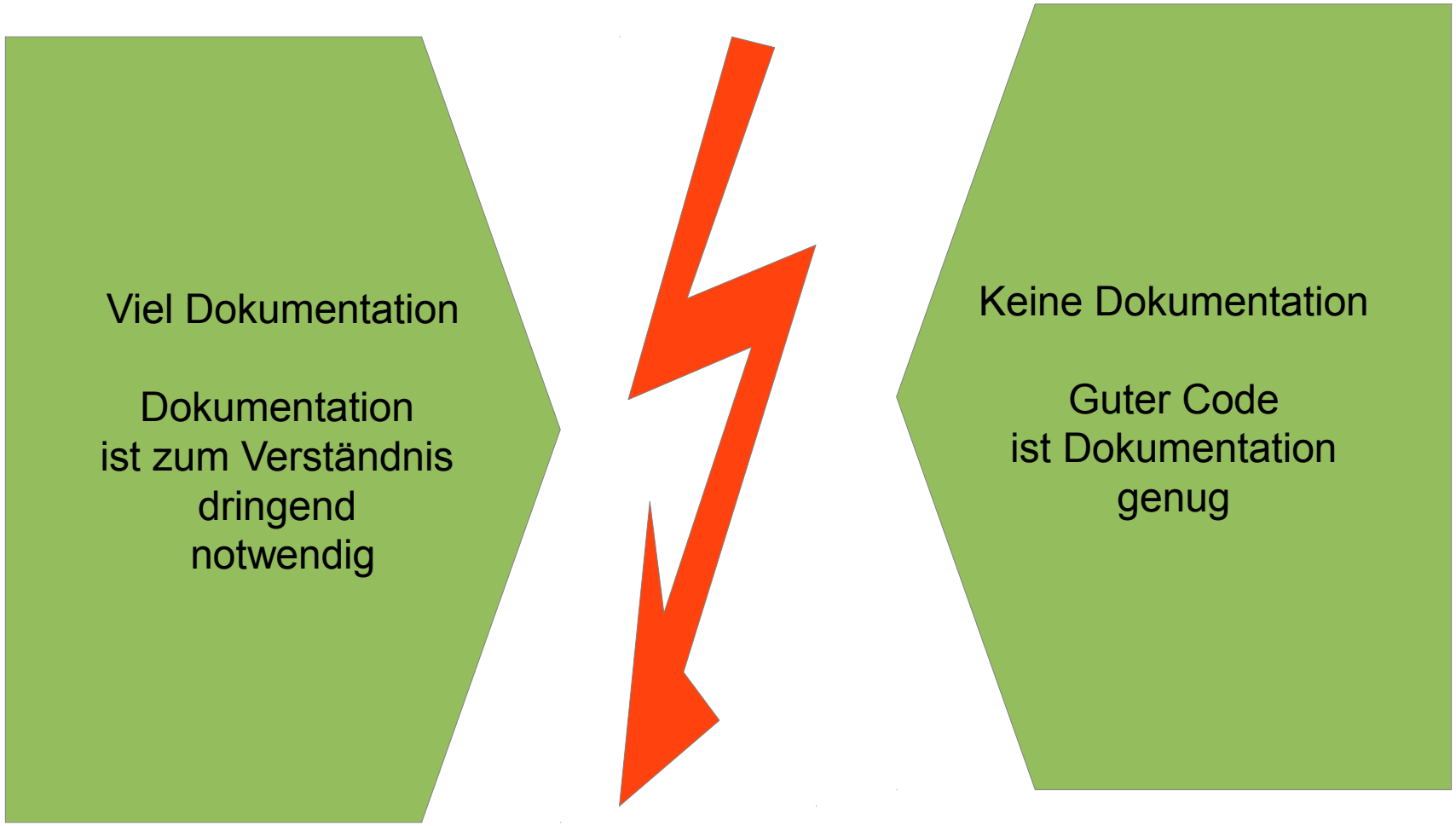
- die Erfahrung aus früheren Problemlöseversuchen in ähnlichen Situationen,
- die Gewöhnung und Mechanisierung durch wiederholte Anwendung eins Denkschemas,
- die Gebundenheit von Methoden und Werkzeugen an bestimmte Verwendungszwecke,
- die Vermutung von Vorschriften, wo es keine gibt (Verbotsirrtum).

Um Fehler durch Einstellungen zu verhindern, sollte das Aufzeigen von Lösungsalternativen und eine Begründung der Methodenwahl zum festen Bestandteil der Problembearbeitung gemacht werden.

Dokumentation

- Programmierdokumentation
  - Quell-Code
- Methodendokumentation
  - Grundlagen
  - Algorithmen und Verfahren
- Installationsanleitung
  - Installation selbst
  - Datenimport/update
  - Probleme
  - Voraussetzungen
- Benutzerdokumentation
  - Bedienung
- Datendokumentation
  - Interpretation
- Testdokumentation
  - Testdaten
  - Erfolgreiche Tests der Vergangenheit
- Entwicklungsdokumentation
  - Abweichungen in der Umsetzung
  - Modelle
  - Endscheidungen mit deren Grundlagen

Code-Dokumentation



Beide Gruppen vergessen....

... dass sich auch das Hintergrundwissen verändert!

- Sowohl „Code als Dokumentation“ als auch „Code dokumentieren“ bauen auf bewusstes und unbewusstes Hintergrundwissen auf.

Exkurs: Atomsemiotik - Grundlagen

Semiotik (von altgriechisch **σημεῖον** sēmeῖon „Zeichen, Signal“) ist die Wissenschaft, die sich mit Zeichensystemen aller Art (zum Beispiel: Bilderschrift, Gestik, Formeln, Sprache, Verkehrszeichen) befasst. Sie ist die allgemeine Theorie vom Wesen, der Entstehung (Semiose) und dem Gebrauch von Zeichen. (Wikipedia)

- Atomsemiotik ist die Bezeichnung für eine Richtung der Semiotik, die sich mit der Warnung der Nachwelt vor den Gefahren des Atommülls beschäftigt.
- Warnung vor Atommüll soll für einen Zeitraum von einer Million Jahre „funktionieren“

Exkurs: Atomsemiotik - Problem

Drei Dinge müssten der Nachwelt mitgeteilt werden:

- dass es sich überhaupt um eine Mitteilung handelt,
- dass an einer bestimmten Stelle gefährliche Stoffe lagern,
- Informationen über die Art der gefährlichen Substanzen

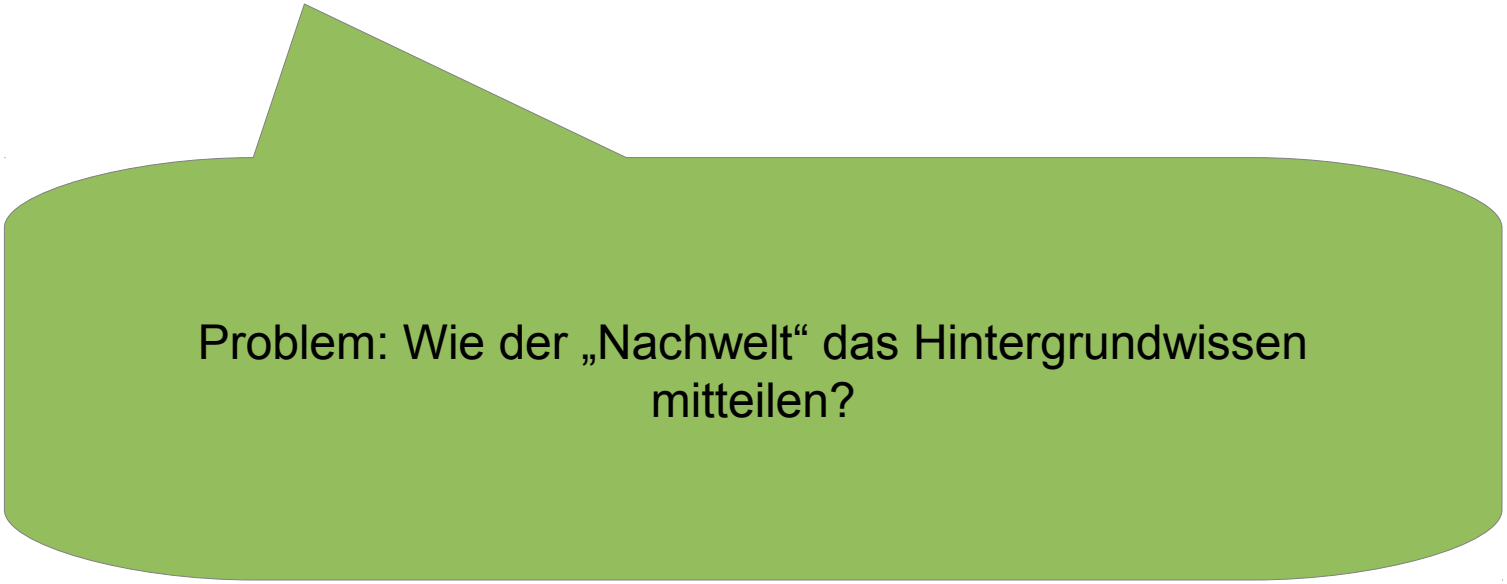
- Veränderung der Sprache
  - Heute können viele alte Sprachen nicht oder nur von Experten verstanden werden
  - Alt heißt hier max. 5000 Jahre

Exkurs: Lösungsvorschläge

- Atompriesterschaft
- mathematischer Codierung auf lebendem Trägermaterial
- Atomblumen
- Warnkreis
  - Mehre Sprachen
  - Ständige Übersetzung

Zurück zum Code

- „Sprechender Code“ und Dokumentation bauen auf Hintergrundwissen auf.
- Das Hintergrundwissen ändert sich mit der Zeit:
  - Wer kennt heute noch das Himem-Fenster und die Steuerungstechniken?



Sprechender Code

Vorteil:

- Kein Doppelter Wartungsaufwand (Code und Dokumentation)
- Häufig besser strukturierter Code
- Kein Problem mit inkonsistenten Dokumentationen

Nachteil:

- Kein Hintergrundwissen wird vermittelt
- Code mit fehlendem Hintergrundwissen nicht verständlich

Wieso ist sprechender Code heute dennoch so populär?

- Wenn Hintergrundwissen vorhanden, optimal in Bezug auf
  - Effektive Programmierung (keine Doppelwartung)
  - Keine Inkonsistenzen
  - Gute Dokumentation ist schwer
- In eng arbeitenden Gruppen beliebt
- Aus einem Problem eine Tugend machen
  - Wer schreibt schon gerne Dokumentation?

Was und wie sollte Dokumentiert werden

- Hinweise auf Hintergrundwissen
  - z.B. Pattern, Schichtenarchitektur, ...
- Nutzung und Rollen großer Strukturen
  - z.B. Visitor-Pattern (Rolle: Besucher, Besuchter)
- Algorithmen
  - z.B. Rotationsmatix
- Versteckende Vereinfachungen
  - z.B. x = 1 statt x = sin(y*y) + cos (y*y)
- Ungeprüfte Annahmen
  - z.B.: Diese Funktion kann nur für Werte größer 0 genutzt werden.
- APIs zu anderen Teil-Systemen
- Sprache: Englisch
- Ansonsten Regeln zum sprechenden Code einhalten

Sprechender Code

- Verwendung von sinnvollen, sprechenden Namen
- Jede Methode hat nur einen Zweck
- Kleine Methoden
- Lesende Methoden haben keine Seiteneffekte
  - Query/Change-Separation

Externe Dokumentation

- Nicht alles kann im Code dokumentiert werden
  - Architektur
  - Hintergrundwissen
  - Glossar
  - Modelle
- Solche Dokumente sollen „nahe“ beim Code zu finden sein
  - z.B. durch Einbettung in die Code-Versionierung

Externe Dokumentation sollte toolunabhängig sein

- Alte Formate werden Teilweise nicht mehr unterstützt
  - Früher Standard WordPerfect – Heute oft MS Office
- Forderung:
  - Dokumente
    - Als Text im Dokument lesbar sein
      - Text-Dateien
      - LaTeX/HTML
      - Mit Einschränkungen XML
    - Ein Anzeigetool ist einfach zu erstellen (voll Doku liegt vor, einfache Implementierung)
      - HTML
      - SVG
      - PDF/A (gilt nicht für normale PDF-Formate!)

Neues....

Probleme mit Hypes

Langlebige Systeme: Vorsicht vor Neuem!

- Auch heute wird Wartbarkeit häufig nicht im Sinne der Wartbarkeit von langlebigen Systemen betrachtet!
- Viele aktuelle Methoden, Praktiken und „unverzichtbare Neuheiten“ sind oft Hypes, die später geändert, korrigiert oder verworfen werden!
- Beispiel: Exceptions bei Java
  - In den 90er Hype: Explizite Exceptions
  - Heute: Expliziten Exceptions sind schlecht, lieber RuntimeExceptions

Java-Exceptions

- Explizite Exception können nur geworfen werden, wenn sie über eine throws-Deklaration der Signatur der Operation erlaubt sind.
- RuntimeException benötigen diese throws-Deklaration nicht.

Explizite Exception müssen behandelt werden

- Reagieren
  - Exception fangen und die Auswirkungen bearbeiten
- Maskieren
  - Exception fangen und eine andere Exception werfen
- Ignorieren
  - Exception nicht fangen und implizit weiter werfen

- „Reagieren“ ist immer gut und unproblematisch

Explizite Exception als Problem

- Maskieren benötigt viel Code und macht das Programm unverständlich.

- Ignorieren verhindert das verbergen des Inneren einer Komponente/Klasse
  - Geheimnisprinzip verletzt
- Exceptions müssen auch an vielen (weit entfernten) Stellen im Code Deklariert werden.

Forderung heute

- Exceptions sollen generell als RuntimeException definiert sein.

Lösungs-Tradeoff

- Häufig sind gehypte Lösungen Lösungen für ein bestimmtes Problem.
- Die Lösungen haben häufig Nachteile in anderen Bereichen
- Mit der Zeit verschiebt sich die Dringlichkeit des Problems
- Teilweise werden erst neue Probleme durch die Lösung eingeführt

Bsp: Clean Code und „Coding by Convention“

- Konvention vor Konfiguration (englisch convention over configuration oder coding by convention) ist ein Softwaredesign-Paradigma, welches zum Ziel hat, die Komplexität von Konfigurationen zu reduzieren: Solange sich die Entwickler in allen Bereichen einer Software an übliche Konventionen (beispielsweise gleichartige Bezeichner) halten, müssen diese nicht konfiguriert werden, was somit die Konfigurationen erheblich vereinfacht, ohne die Möglichkeiten der Entwickler einzuschränken. Damit unterstützt das Paradigma auch die Prinzipien KISS und Don’t repeat yourself.

Bsp: Conventions können sich ändern

- Neue Versionen
  - Neue Conventions
  - Geänderte Conventions
  - Conventions fallen weg
- Neue Entwickler kennen daher nicht unbedingt die Conventions alter Versionen

YAGNI - You Ain’t Gonna Need It

- „Du wirst es nicht brauchen“
- Änderungen nicht versuchen Vorherzusehen
  - Die meisten Vorhersagen sind falsch
  - Vorbereitungen auf Änderungen machen den Code komplexer

Dieses Prinzip ist unter anderen Namen schon alt und kann daher als gesichert gelten!

Automation

„Scripte“ statt Handarbeit

Regelmäßige Ausführung

Manche Tätigkeiten müssen Häufig durchgeführt werden:

- Bauen der Anwendung
- Durchführen von Tests
- ...

Problem:

- Nicht alle Ausführungen sind gleich
- Durchführung ist stupide
- Durchführung ist nicht nachvollziehbar

Problem-Beispiel

Eine C-Bibliothek soll ausgeliefert werden

- Der normale Auslieferungsmitarbeiter ist krank
- Sein Vertreter baut in seiner Entwicklungsumgebung die Bibliothek
- Beim Kunden führt die Bibliothek zum Absturz beim Laden der Bibliothek

- Grund: In den IDEs waren unterschiedliche Einstellung zur Code-Optimierung eingestellt. Die experimentellen Optimierungen bei der Vertretung liefen nicht in der Benutzerumgebung.

Automatisierung der Produktion

- Alle zum Erzeugen der Auslieferung benötigten Bestandteile sollen im Prozess enthalten sein:
  - Code
  - Spezielle Tools
  - Dokumentation
- Erstellung soll immer auf einer „sauberen“ Umgebung starten
- Als Ziel sollte ein vollständiges Abbild der Auslieferung entstehen (z.b. Image des Installationsmediums)
- Im Prozess sollen Tests der Anwendung enthalten sein, die eine versehentliche Auslieferung einer „unfertigen“ Anwendung verhindern.

Tools

- Make (C und UNIX)
- Ant (Java)
- Ant.Net (.Net)
- Maven (Java, C, Apache)

Bsp Ant

- Eingabe: build.xml
- Unterstützt
  - Compilieren
  - JAR-Archiv erstellen
  - JUNIT
- Verschiedene Ziele können definiert werden
- Parametrisierung einlesbar

ANT

```
<project name="MyProject" default="dist" basedir=".">
  <description>
    simple example build file
  </description>
 <!-- set global properties for this build -->
 <property name="src" location="src"/>
 <property name="build" location="build"/>
 <property name="dist"  location="dist"/>

 <target name="init">
  <!-- Create the time stamp -->
  <tstamp/>
  <!-- Create the build directory structure used by compile -->
  <mkdir dir="${build}"/>
 </target>

 <target name="compile" depends="init"
     description="compile the source " >
  <!-- Compile the java code from ${src} into ${build} -->
  <javac srcdir="${src}" destdir="${build}"/>
 </target>
```

ANT

```
 <target name="dist" depends="compile"
     description="generate the distribution" >
   <!-- Create the distribution directory -->
   <mkdir dir="${dist}/lib"/>

   <!-- Put everything in ${build} into the MyProject-${DSTAMP}.jar file -->
   <jar jarfile="${dist}/lib/MyProject-${DSTAMP}.jar" basedir="${build}"/>
 </target>

 <target name="clean"
     description="clean up" >
   <!-- Delete the ${build} and ${dist} directory trees -->
   <delete dir="${build}"/>
   <delete dir="${dist}"/>
 </target>
</project>
```

Einige Tasks

- javac zum Kompilieren von Quellcode.
- copy zum Kopieren von Dateien.
- delete zum Löschen von Dateien oder Verzeichnissen.
- mkdir zum Erstellen von Verzeichnissen.
- junit für automatisierte (JUnit-)Tests.
- move zum Umbenennen von Dateien oder Verzeichnissen.
- exec zum Ausführen von System-Programmen.
  - Achtung: Bei Benutzung dieses Tasks begibt man sich häufig in die Abhängigkeit eines Betriebssystems!
- zip zum Zippen, also zum Komprimieren von Dateien.
- cvs zum Durchführen von CVS-Operationen.
- mail zum Versenden von E-Mails.
- replace zum Ersetzen von Text in Dateien.

Fragen?

Nächste Woche
Modellebene