Seminar Software Engineering
Sommersemester 2005 (IMS)

# XP: Java-Tools und Werkzeuge

Von: Massimo Acquasanta & Tobias Schmidt

# Agenda

- XP-Praktiken
- Tools und Werkzeuge
  - o Refactoring-Tool
  - o CVS
  - o JUnit
  - o Ant
- Fazit



# Kapitel XP-Praktiken

07.06.2005 Massimo Acquasanta und Tobias Schmidt

# Die XP-Praktiken (1)

## Zusammenfassung: Die 12 XP-Praktiken

- Planungsspiel (planning game)
- kurze Releasezyklen
- Metapher
- einfaches Design
- Testen
- Refactoring

07.06.2005 Massimo Acquasanta und Tobias Schmidt

XP ruht auf vier Werten:

**Einfachheit:**

**Entwicklungsprozess soll einfach sein, es sollen einfache Lösungen gefunden werden, weil diese schneller und kostengünstiger zu realisieren. → Einfacher erklären, warten und weiterentwicklen**

**Kommunikation:**

**Projektmitglieder sollen intesnsiv miteinander kommunizieren, Missverständnisse & Unklarheiten ausgeräumt**

**Gute Kommunikation bedeutet verzicht auf Teil der Doku**

**Feedback:**

**Qualität wir über Feedback gesichert**

**Mut:**

**Mut braucht man für die vorherigen 3 Werte.**

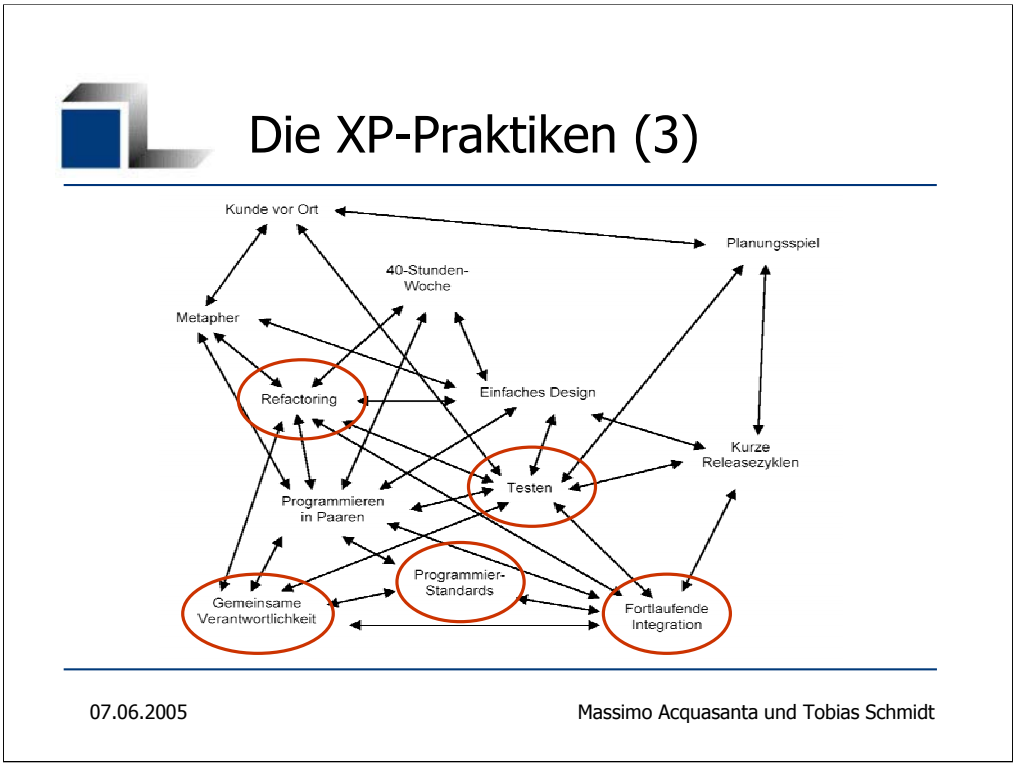
**Einfacheit → könnte zu einfach sein, Entwickler wollen Struktur um spätere Änderungen leicht einzufügen**

**Feedback → negative Bewertung könnte als persönliche Kritik aufgefasst werden**

**Kommunikation → kann sich herausstellen, dass man Missverständnis aufgesessen ist, und an SW ändern muss**

# Die XP-Praktiken (2)

- Programmieren in Paaren (pair programming)
- gemeinsame Verantwortlichkeit
- fortlaufende Integration
- 40-Stunden-Woche
- Kunde Vor-Ort
- Programmierstandards




**Folgerung**: Alle Praktiken stützen sich gegenseitig



# Kapitel Tools und Werkzeuge

07.06.2005 Massimo Acquasanta und Tobias Schmidt

# Tools und Werkzeuge

- Unterstützung durch/bei:
  - Refactoring
  - Versionsverwaltung
  - Testen
  - Build-Prozess

# Refactoring (1)

Was ist Refactoring?

- Verbesserung des Code-Designs ohne die Funktionalität zu verändern
- Code einfach wie möglich halten
- selbstdokumentierend

→ hohe Kosten für Änderungen vermieden

## Refactoring (2)

- Typische Refactorings:
  - Umbenennen von Klassen, Operationen und Attributen
  - Aufsplitten einer Operation in mehrere Operationen
  - Sicheres Löschen von Klassen, Operationen etc.
  - Verschieben von Operationen und Attributen
  - Extrahieren von Oberklassen



Martin Fowlers Refactoring Katalog umfasst knapp 100 Refactorings (http://www.refactoring.com/catalog/)

# Refactoring (3)

Wie wird Refactoring durchgeführt?

1. Sicherstellen, dass Unit-Test für die zu ändernde Funktion existert
2. Änderungen durchführen
3. Unit-Tests ausführen
4. Schritte 2-4 wiederholen, bis Refactoring beendet

## Versionsverwaltung (1)

- Versionsbibliothek (Repository)
- Komponentenorientierte Sichtweise
- Checkout: Ausleihen von Komponenten aus der Bibliothek zwecks Bearbeitung
- Commit: Einbringen von bearbeiteten Komponenten
- Komponenten erhalten Versionsnummern

07.06.2005 Massimo Acquasanta und Tobias Schmidt

**Häufig Client-Server-System**

**Repository zentral auf Server**

**Jeder Entwickler erhält lokale Arbeitskopie**

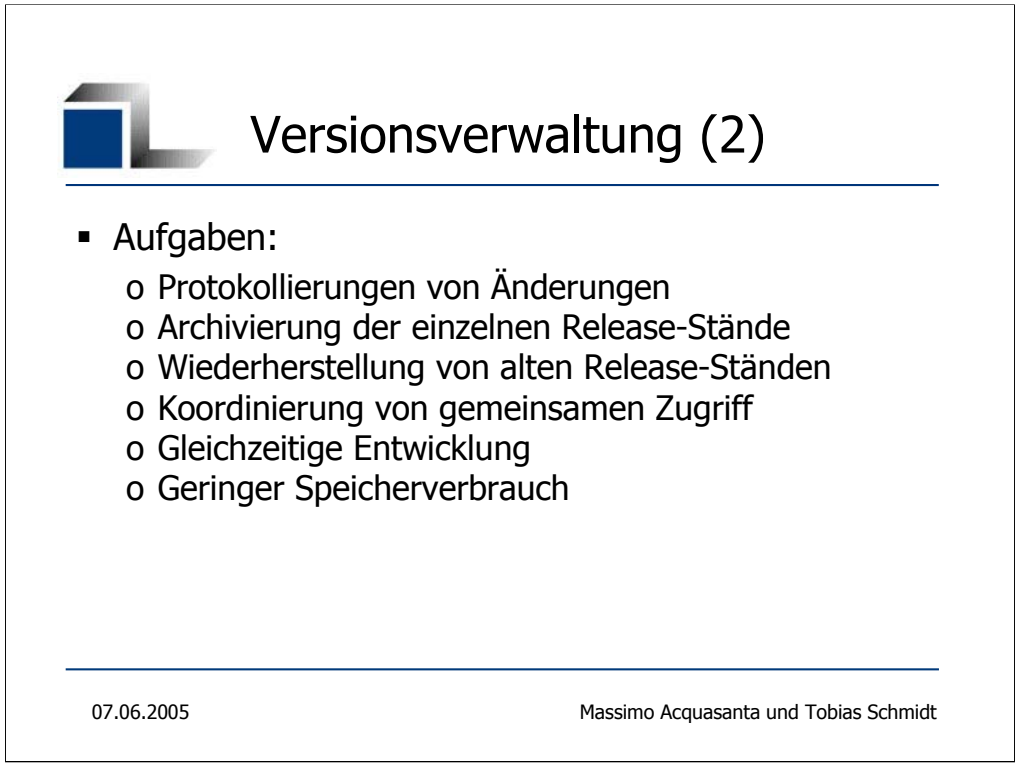
Wozu:

**Alte Versionen gehen verloren**

**Speicherverbrauch**

**Änderungen schwierig zu lokalisieren**

**Mehrbenutzerzugriff**




**Aufgaben**:

**Protokollierungen** der Änderungen - Es kann jederzeit nachvollzogen werden, wer wann was geändert hat.

**Wiederherstellung** von alten Ständen einzelner Dateien - Somit können versehentliche Änderungen jederzeit wieder rückgängig gemacht werden.

**Archivierung** der einzelnen Release-Stände eines Projektes - Dadurch ist es jederzeit möglich auf alle ausgelieferten Versionen zuzugreifen.

**Koordinierung** des gemeinsamen Zugriffs von mehreren Entwicklern auf die Dateien

**Gleichzeitige Entwicklung** mehrerer *Entwicklungszweige* eines Projektes - Hier wird der Entwickler bei der Übernahme von einzelnen Änderungen zwischen den Zweigen und der Hauptversion unterstützt.

(z.B. stabile Release-Version und Entwicklerversion mit größeren, nicht getesteten Änderungen)

**Geringer Speicherverbrauch** - speichert üblicherweise nur die Unterschiede zwischen zwei Versionen um Speicherplatz zu sparen. Dadurch kann eine große Zahl von Versionen archiviert werden.

Durch dieses Speicherformat kann jedoch nur mit der Software des Versionsverwaltungssystem auf die Daten zugegriffen werden, die die gewünschte Version bei einem Abruf unmittelbar aus den abgelegten Schnipseln "zusammenbaut".

# Versionsverwaltung (3)

- Gleichzeitiges Bearbeiten
  - Checkout:
    - 1. Variante:
      - Erster Checkout sperrt den Schreibezugriff
      - Zweiter Checkout erhält nur Lesezugriff
    - 2. Variante:
      - Jeder Checkout bildet eine neue Variante
      - Beim Commit müssen die Versionen zusammengeführt werden
  - Commit:
    - 2-Wege-Vergleich:
      - Direktvergleich der beiden Varianten
    - 3-Wege-Vergleich:
      - Vergleich der zwei Varianten mit der Ursprungsvariante



2-Wege Vergleich meist schlechte Ergebnisse → warum???

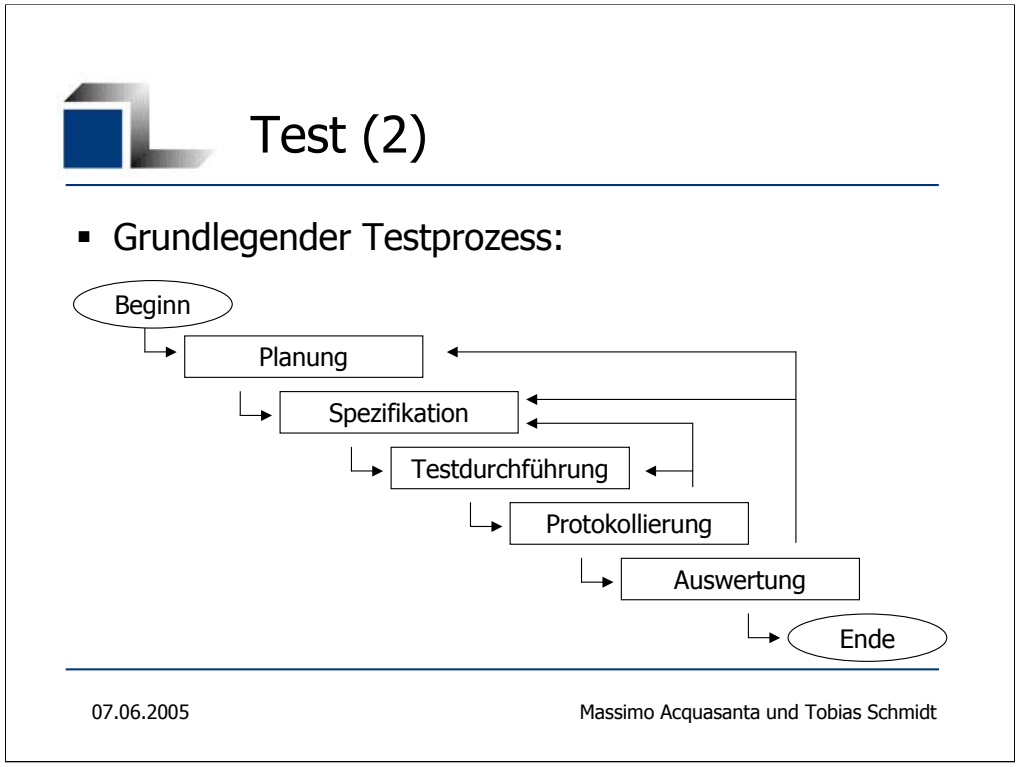
## Test (1)

- Gesamter Prozess, der alle Aktivitäten umfasst, die notwendig sind, um:
  - Fehler aufzudecken
  - korrekte Umsetzung der Anforderungen nachzuweisen
- Dient zur Qualitätssicherung
- Bestimmen von Testobjekten
- Festlegen von Testfällen

07.06.2005 Massimo Acquasanta und Tobias Schmidt

Testobjekt: Komponenten, Teil- oder System das einem Test unterzogen wird

Testfälle: Festlegen der Vorbedingungen, Eingaben, Ausgaben und Nachbedingungen → Verhalten des Testobjekts wird festgelegt




Testprozess soll sicherstellen, dass die Beschäftigung mit dem Testen der Software schon frühzeitig erfolgt und zu einem planvollen Vorgehen während der Projektlaufzeit führt.

Ziel ist es, das Testen nicht als "notwendiges Übel" an das Ende der Entwicklungsphase kurz vor der geplanten Auslieferung des Systems zu schieben, sondern schon so früh wie möglich so viel wie nötig zu testen.
Als zur Systementwicklung parallele Aktivität gesehen werden

**Testplanung** erfolgt zu Beginn und wird aufgrund der häufigen Änderungen steig aktualisiert

Unterteilt in:

**Ressourcenplanung**

Abschätzung der voraussichtlich benötigte Arbeitszeit, verwendete Werkzeuge und andere Hilfsmittel, zudem Zeitplan Schulungsmaßnahmen

**Teststrategie**
Die Teststrategie legt fest, welche Teile des Systems mit welcher Intensität getestet werden müssen.

**Testpriorisierung**

Priorisierung der Tests, mit dem Ziel, die kritischen Systemteile zuerst zu testen

**Werkzeugunterstützung**

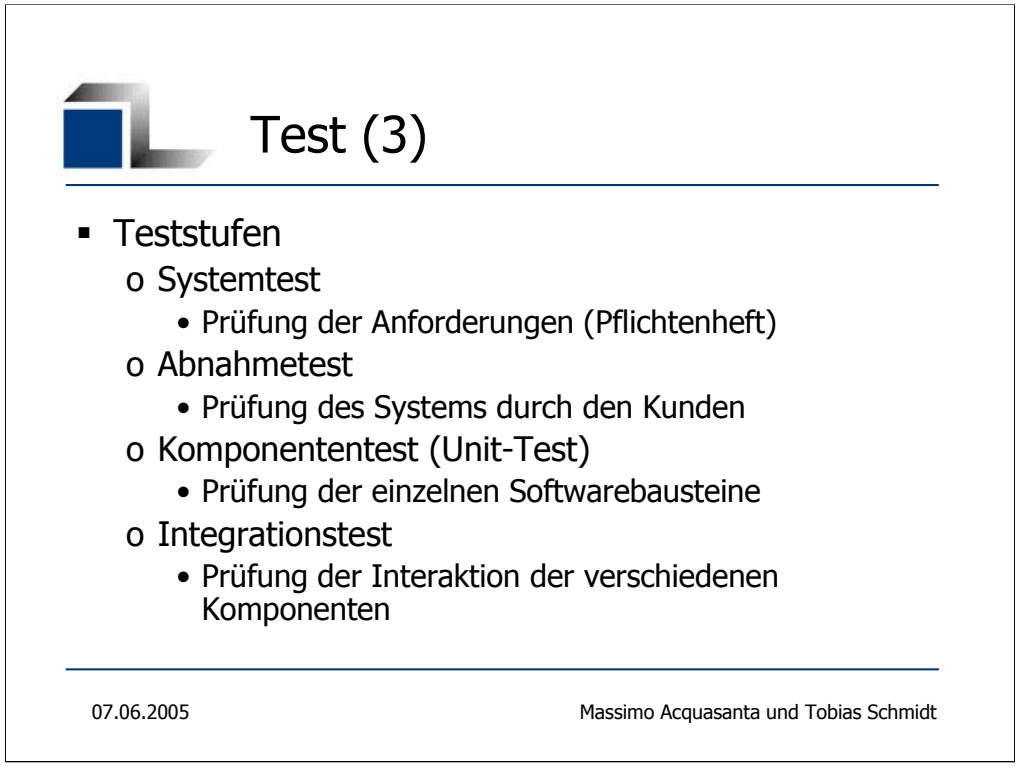
wird der *Tooleinsatz* während des Testens geklärt (reicht aus muss neues her etc)

**Testspezifikation** dient der genauen Beschreibung der Testfälle und Konkretisierung dieser

**Testdurchführung**: Abarbeitung der Testfälle nach Priorisierung. Vorraussetzung zu testende Systemteile verfügbar

**Testprotokollierung**:

Die Protokollierung der Testdurchführung hat zum Zweck, jederzeit überprüfen oder




Systemtest:

Prüfung Funktion, Leistung, Qualität des Systems,

System wird als ganzes betrachtet → interne Schnittstellen zwischen Komponenten werden nicht betrachtet

Es lässt sich die Erfüllung der festgelegten Qualitätsmerkmale prüfen bspw. Antwortzeiten, Speicherbedarf etc.

Abnahmetest:

Kunde prüft komplettes System aus Sicht des Anwenders und soweit möglich des Entwicklers, um zu sehen ob alles zu seiner Zufriedenheit

Komponententest:

Das sichtbare Verhalten einer Komponente (z.B. einer Klasse) wird verifiziert und dem Programmierer eine unmittelbare Rückmeldung darüber gegeben, ob die Komponente das geforderte Verhalten aufweist oder nicht.

Komponententests sind ein wesentlicher Bestandteil der Qualitätssicherung in der Softwareentwicklung.

Durch diese Rückmeldung wird die Wartbarkeit z.B. durch Refactoring vereinfacht bzw. erst ermöglicht.

Integrationstest:

Eine aufeinander abgestimmte Reihe von Einzeltests, die dazu dienen, verschiedene voneinander abhängige Komponenten eines komplexen Systems im Zusammenspiel miteinander zu testen.

Die erstmals im gemeinsamen Kontext zu testenden Komponenten haben jeweils einen Unit-Test erfolgreich bestanden und sind für sich isoliert fehlerfrei funktionsfähig.

//Testet Funktionen nach Einbindung der Software in die bereits vorhandene Softwarearchitektur.

## Build Prozess (1)

- Build Prozess bezeichnet Schritte, die durchzuführen sind, um ein Anwendungsprogramm zu veröffentlichen.
- Diese Schritte können beispielsweise die Code-Kompilierung, das Binden des kompilierten Codes an Bibliotheken und die Generierung einer technischen Dokumentation sein.

07.06.2005 Massimo Acquasanta und Tobias Schmidt

Beim Build können Tools wie das Java-basierte Ant eingesetzt werden.

## Build Prozess (2)

- Wozu ein Build-System?
  - konstantes Refactoring
  - gemeinsame Verantwortung
  - wechselnde Programmierpaare
  - unterschiedliche Konfigurationen



Bestandteil des Buildsystems „Cruise Control“ (http://cruisecontrol.sourceforge.net)

# Tool Refactoring (1)

Beispiel – Vor Refactoring:

```
class Person
{
  private String firstName;
  public void setFirst(String s)
  {
      this.firstName = s;
  }
}
```

# Tool Refactoring (2)

Beispiel – bessere Namen für Argumente:

```
class Person
{
  private String firstName;
  public void setFirst(String firstName)
  {
      this.firstName = firstName;
  }
}
```

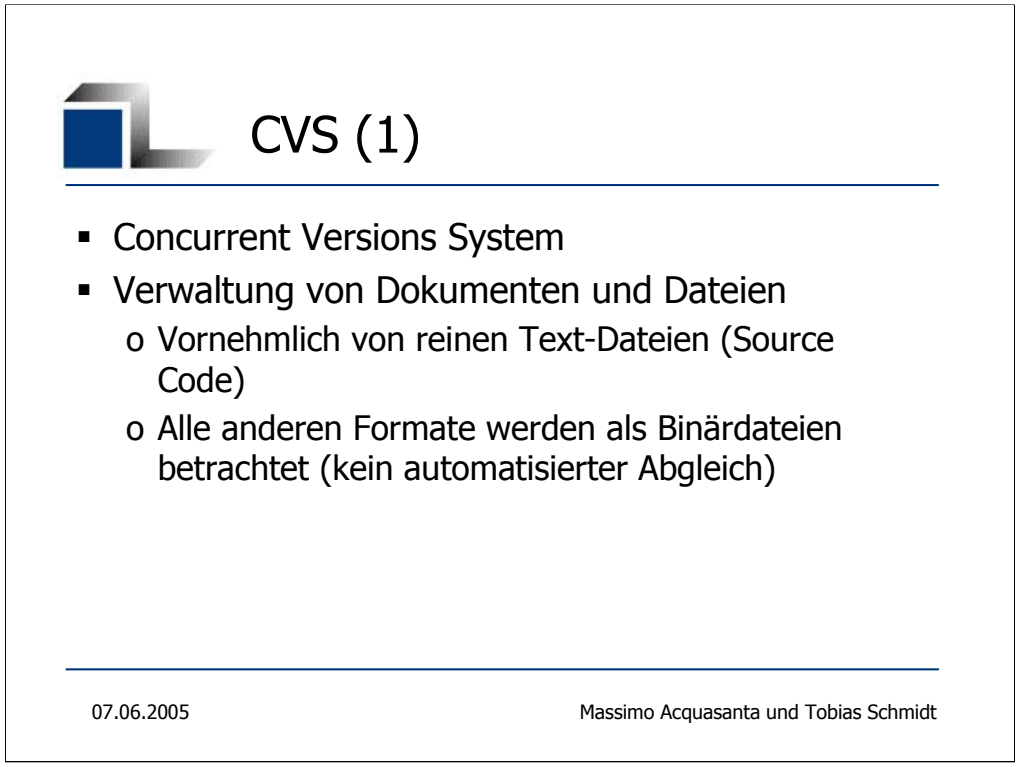
# Tool Refactoring (3)

## Beispiel – Änderung des Methodennamens

```
class Person
{
  private String firstName;
  public void setFirstName(String firstName) {
      this.firstName = firstName;
  }
}
```

# Tool Refactoring (4)

- Martin Fowlers Refactoring Katalog umfasst knapp 100 Refactorings (http://www.refactoring.com/catalog/)




WAS IST CVS?

CVS ist ein Versionskontrollsystem. Durch die Verwendung können Sie die Änderungen bzw. die Historie Ihrer Quelldateien aufzeichnen.

Beispielsweise schleichen sich Fehler in Software ein, wenn diese modifiziert wird und der Fehler wird erst sehr viel später entdeckt, wenn die Änderung die ihn verursacht hat schon längst vergessen ist. Mit CVS sind Sie in der Lage einfach alte Versionen wieder herzustellen und nach zuschauen was denn nun die Ursache des Fehlers war. Das ist manchmal eine große Hilfe.

Sie können natürlich jede Version jeder Ihrer Dateien die Sie je erzeugt haben ablegen. Das würde aber auch eine enorme Menge Plattenplatz verschwenden. CVS speichert alle Versionen einer Datei in eine Datei und verwendet dabei eine klevere Methode wobei nur die Unterschiede zwischen den Versionen verwendet werden.

CVS Hilft Ihnen auch, wenn Sie Teil eines Entwicklungsteams sind, daß am gleichen Projekt arbeitet. Es ist so einfach die Änderungen eines Anderen zu überschreiben, wenn Sie nicht höllisch vorsichtig sind. Einige Editoren, wie GNU Emacs, versuchen sicher zustellen, daß die gleiche Datei nicht durch zwei Personen zur gleichen Zeit geändert werden kann. Wie auch immer, wenn jemand einen anderen Editor verwendet funktioniert diese Sichheitsbariere nicht mehr. CVS löst diese Problem, indem jeder Entwickler von den anderen isoliert wird. Jeder Entwickler arbeitet in seinem eigenen Verzeichnis und CVS führt die Arbeit zusammen wenn jeder Entwickler fertig ist.

CVS ging aus einer Anzahl von Shell Scripten hervor, die von Dick Grune geschrieben und in der Newsgroup comp.sources.unix im Volume 6 im Dezember 1986 gepostet. Es ist kein Code dieser Shell--Scripts in der aktuellen Version von CVS vorhanden, aber vieles des Konflikt--Auflösungsalgorithmus stammte aus diesen Scripten.

Im April 1989, hat Brian Berliner den Code für CVS designed und programmiert. Jeff Polk hat später Brian mit dem Design des Module-- und Vendor--Branch Unterstützung geholfen.

# CVS (2)

- Wichtige CVS Befehle:
  - cvs checkout
    - Daten aus dem Repository in den eigenen Arbeitsbereich
  - cvs commit
    - Änderungen am eigenen Arbeitsbereich ins Repository zurückstellen
  - cvs update
    - Zwischenzeitliche Änderungen im Repository mit den Dateien im eigenen Arbeitsbereich abgleichen
  - cvs add
    - Neue Dateien zum Repository hinzufügen

07.06.2005 Massimo Acquasanta und Tobias Schmidt

**1.2 Was CVS nicht ist?**

CVS kann eine Menge Dinge für Sie tun, aber CVS versucht nicht alles für jeden zu tun.

CVS ist kein Build System.

Die Struktur Ihres Repositorys und ihrer Module Datei sind mit Ihrem Build System verknüpft (bspw. `Makefile's), sind aber essentiell unabhängig voneinander.

CVS schreibt Ihnen nicht vor, wie Sie etwas Erstellen. Es speichert lediglich Dateien zur Wiederherstellung einer Baum Struktur die Sie vorgeben.

CVS schreibt Ihnen nicht vor, wie Sie Ihren Plattenplatz in ausgecheckten Verzeichnis verwenden. Wenn Sie Ihre `Makefile's oder Scripte in jedem Verzeichnis schreiben, so müssen diese die relative Position zu allem anderen kennen, das kann dazu führen, daß Sie das vollständige Repository auschecken müssen.

Wenn Sie Ihre Arbeit modularisieren, und ein Build--System erzeugen, das Dateien shared (via links, mounts, VPATH in `Makefile's, etc.) so können Sie Ihre Plattenbenutzung so arrangieren wie Sie es möchten.

Sie sollten aber daran denken, daß jedes System in dieser Art, einen Haufen Arbeit bei der Erstellung verursacht und auch die Pflege. CVS ist nicht für dieser Art von Aufgaben konzipiert worden, die im Zusammenhang damit entstehen.

Selbstverständlich, sollten Sie solche Tools die zur Unterstützung eines Build--System erzeugt wurden (Scripte, `Makefile's, etc. ) unter die Kontrolle von CVS stellen.

Um heraus zu finden, ob Dateien neu erstellt werden müssen wenn sich etwas ändert, ist etwas das ausserhalb von CVS gehandhabt werden muß. Ein traditioneller Ansatz ist, daß zur Erstellung make verwendet wird und ein automatisiertes Tools, um die Abhängigkeiten zu erstellen, die make verwendet.

Sehen Sie unter Abschnitt 14. How your build system interacts with CVS nach, um weitere Informationen bezüglich der Erzeugung in Verbindung mit CVS zu erhalten.

CVS ist kein Ersatz für Management

Von Ihren Managern und Projekt--Leitern wird erwartet, daß mit Ihnen häufig genug ein Austausch stattfindet damit sichergestellt ist, daß Ihnen Abläufe, Zusammenführungspunkte, Verzeigungen in der Entwicklung und Release Daten bekannt sind. Wenn das nicht gemacht wird, kann Ihnen CVS auch nicht helfen.

CVS ist ein Instrument, damit die Quellen nach Ihrer Pfeife tanzen. Aber Sie sind der Pfeifer und Komponist. Kein Instrument spielt von selbst oder schreibt seine eigene Musik.

CVS ist kein Ersatz für Entwickler Kommunikation

Wenn Entwickler mit Konflikten innhalb einer Datei konfrontiert werden, dann wird das unter den Entwicklern gelöst ohne großes Aufheben. Aber eine mehr allgemeine Definition von **Konflikten** beinhaltet Probleme, welche zu komplex sind um sie ohne eine Kommunikation der Entwickler unter einander zu lösen.

CVS ist nicht in der Lage festzustellen, ob innerhalb einer Datei oder über eine Kollektion von Dateien gleichzeitig veränderungen vorgenommen werden, die sich logisch widersprechen. Das Konzept des Konfliktes (*conflict*) ist rein textualer natur, die auftritt, wenn zwei Änderungen an der gleichen Datei so nah einander sind, daß das Mergen (i.e. diff3) verhindert wird.

CVS ist nicht in der Lage Konflikte im Bereich non-textualen oder in verteilten Konflikten der Programmlogik zu helfen.

Beispielsweise sagen wir, Sie ändern die Argumente der Funktion X, die in der Datei `A' definiert ist. Zur gleichen Zeit editiert jemand die Datei `B' und fügt einen neuen Aufruf der Funktion X hinzu, unter Verwendung der alten Argumente. Das ist außerhalb der Kompentenz von CVS.

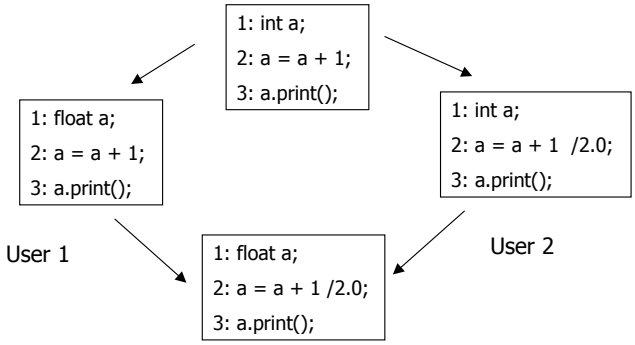
Hat jemand eine gute Idee, wie man das hier übersetzt: Gewöhnen Sie sich an, Spezifikationen zu lesen und mit Ihren Kollegen zu reden.

CVS hat keine change control(1) *Change control* kann vieles bedeuten. Es kann zuerst einmal *bug-tracking* heißen, einen Datenbestand zu haben, in dem gefundene Fehler festgehalten werden und der Zustand eines jeden (Ist der Fehler schon behoben? In welcher Release war

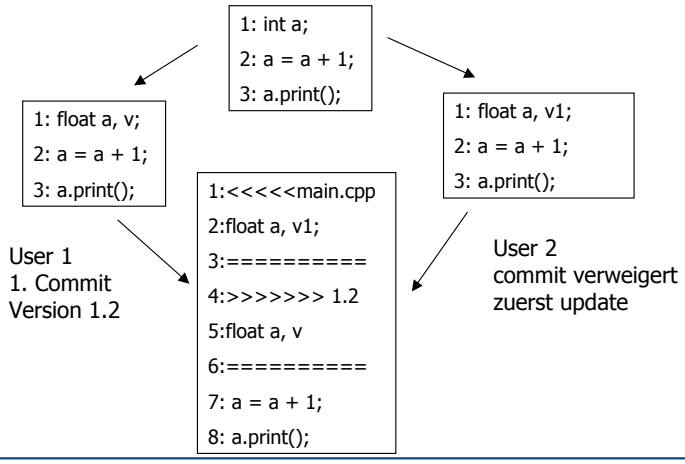
# CVS (3)

- Umgang mit Konflikten:
  - CVS kann bei textorientierten Dateien Konflikte unterschiedlicher Zeilen automatisch auflösen.
  - Konflikte der selben Zeile müssen vom Benutzer gelöst werden

# CVS (4)


1: int a;
2: a = a + 1;
3: a.print();
1: float a;
2: a = a + 1;
3: a.print();
1: int a;
2: a = a + 1 /2.0;
3: a.print();
User 1
User 2
1: float a;
2: a = a + 1 /2.0;
3: a.print();

# CVS (5)


1: int a;
2: a = a + 1;
3: a.print();
1: float a, v;
2: a = a + 1;
3: a.print();
1: float a, v1;
2: a = a + 1;
3: a.print();
1:<<<<<main.cpp
2:float a, v1;
3:==========
4:>>>>>>> 1.2
5:float a, v
6:==========
7: a = a + 1;
8: a.print();
User 1
1. Commit
Version 1.2
User 2
commit verweigert
zuerst update

# CVS (6)

- Beispiel „checkout“:

```
C:\>cvs checkout projekt
cvs server: updating projekt
U projekt/beispiel1.java
U projekt/beispiel1.class
U projekt/upload.txt
C:\>cd projekt
C:\projekt\>dir
beispiel1.java      beispiel1.class      upload.txt
C:\projekt\>
```



Erzeugt Verzeichnis „projekt“ und bevölkt es mit den Quelldateien

# CVS (7)

- Beispiel „add“:

```
C:\projekt\>cvs add beispiel2.*
examing.
cvs server: scheduling file `beispiel2.java' for addition
cvs server: scheduling file `beispiel2.class' for addition
use 'cvs commit' to add this file permanently
C:\projekt\>
```

- Binärdateien müssen mit dem Parameter –kb zum Repository hinzugefügt werden

07.06.2005 Massimo Acquasanta und Tobias Schmidt

Erzeugt Verzeichnis „projekt“ und bevölkt es mit den Quelldateien

Was kb

# CVS (8)

- Beispiel „commit“:
  C:\projekt\>cvs ci –m „neues beispiel“
  cvs commit: examing.
  Checking in beispiel2.java
  /projekt/beispiel2.java <-- beispiel2.java
  initial revision: 1.0
  Checking in beispiel2.class
  /projekt/beispiel2.class <-- beispiel2.class
  initial revision: 1.0
  Checking in upload.txt
  /projekt/ upload.txt <-- upload.txt
  new revision: 1.1 previous revision: 1.0
  C:\projekt\>

- Der Zusatz –m „neues beispiel“ fügt einen Kommentar hinzu. Wenn dieser Parameter nicht mit angegeben wird, öffnet CVS beim „commit“ automatisch ein Texteditor, der zur Eingabe auffordert.



Erzeugt Verzeichnis „projekt“ und bevölkt es mit den Quelldateien

# JUnit (1)

## Was ist JUnit?

- Open-Source-Framework zur Automatisierung von Unit-Tests in Java
- Entwickler: Kent Beck und Erich Gamma
- Integration in moderne IDEs vorhanden
- Ant-Task für JUnit verfügbar
- nicht geeignet für Tests von GUIs
- XUnit-Frameworks für weitere Sprachen verfügbar (CSUnit, utPLSQL)

07.06.2005 Massimo Acquasanta und Tobias Schmidt

Automatisierung von:
Testen einzelner Methoden,
Testen es Protokolls einer Klasse
Testen von Interaktionen zwischen Objekten

Unit-Test (Komponententest):
Wird entwickelt bei Verwirklichung Anforderung,
nach Fertigstellung von Aufgabe,
Immer wenn Defizite an anderen Testfällen.
Bei bzw. Nach Refactoring,
beim fixen von Fehlern

Rückgabewerte variablen ertesten keine layoutspezifischen Sachen

# JUnit (2)

Beispiel:

```
class Person {
  private String firstName;
  private String lastName;

  public String getFullName()
  {
    return firstName + lastName;
  }
}
```

# JUnit (3)

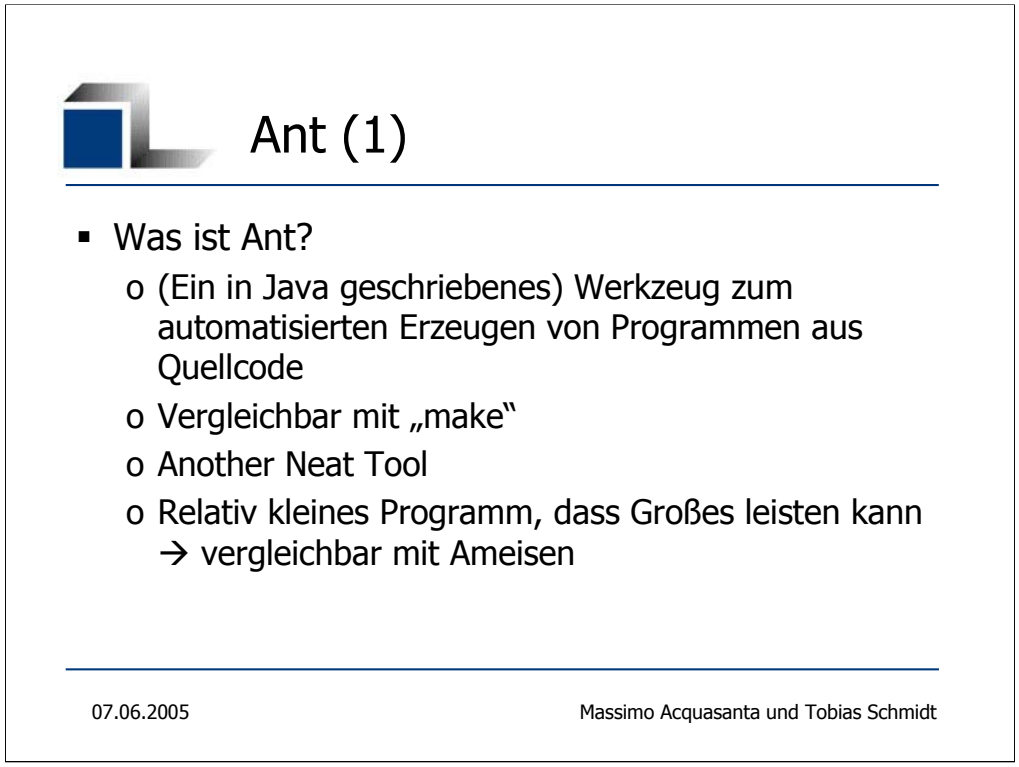
Beispiel-Test:

```
import junit.framework.TestCase;

class TestPerson extends TextCase {

  public void testGetFullName() {
      Person p = new Person(„Mia“, „Meyer“);
      assertEquals(„Mia Meyer“,
                   p.getFullName());
  }
}
```

# JUnit (4)

weitere Test-Tools:

- HttpUnit (http://www.httpunit.org)
- Mock Objects (http://www.mockobjects.com)
- Cactus (http://jakarta.apache.org/cactus)
- JUnitPerf (http://clarkware.com/software/JUnitPerf.html)




Das Programm **Ant** ist ein in Java geschriebenes Werkzeug zum automatisierten Erzeugen von Programmen aus Quell-Code. Damit erfüllt es den gleichen Zweck wie das sehr verbreitete Programm make, nämlich die automatisierte Erstellung von installierbaren Software-Paketen aus existierendem Quell-Code, Bibliotheken und sonstigen Dateien.

Ant ist Open Source, startete als Teil des Jakarta-Projekts und ist nun ein Apache-Top-Level-Projekt. Ant ist die Abkürzung für "Another Neat Tool" (engl. für "Noch ein hübsches Werkzeug") und entwickelt wurde die erste Version von James Duncan Davidson der 1999 ein Tool wie make für Java benötigte, als er die erste J2EE-Referenz-Implementierung entwickelte. Davidson gilt weiterhin als Vater von Jakarta Tomcat. Der Name "Ant" steht auch dafür, dass es, vergleichbar mit Ameisen, als relativ kleines Programm, Großes leisten kann.

Was ist Ant?

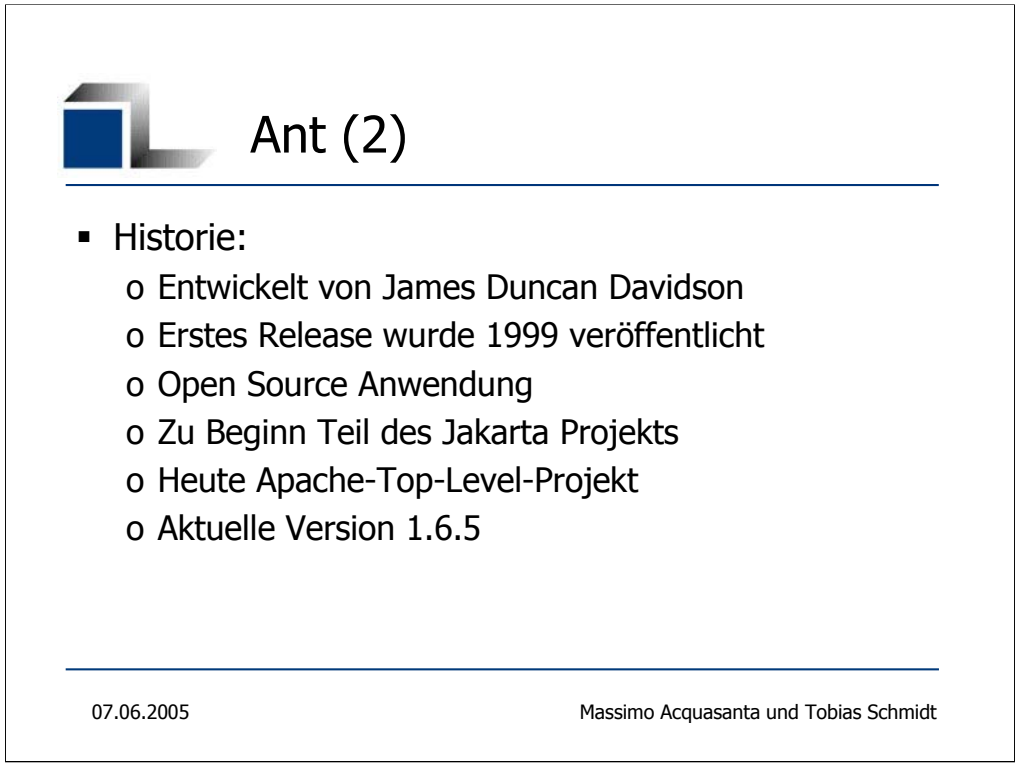
Java-basierter Ersatz für plattformspezifisches *make*

Konfiguration über XML-Dateien (*build.xml*)

Definition von *tasks*: Quellcode kompilieren, JAR-Archive erstellen, Deployment, Einbinden von Unit-Tests, CVS bedienen, ...

mehr als 100 Standardtasks vorhanden

einfach erweiterbar (*custom tasks*)




Ant ist Open Source, startete als Teil des Jakarta-Projekts und ist nun ein Apache-Top-Level-Projekt. Ant ist die Abkürzung für "Another Neat Tool" (engl. für "Noch ein hübsches Werkzeug") und entwickelt wurde die erste Version von James Duncan Davidson der 1999 ein Tool wie make für Java benötigte, als er die erste J2EE-Referenz-Implementierung entwickelte. Davidson gilt weiterhin als Vater von Jakarta Tomcat. Der Name "Ant" steht auch dafür, dass es, vergleichbar mit Ameisen, als relativ kleines Programm, Großes leisten kann.

Das **Jakarta-Projekt** ist ein Projekt der Apache Software Foundation. Es beherbergt, entwickelt und unterstützt freie Software, die in der Programmiersprache Java geschrieben wurde. Zur Zeit besteht es aus 19 Unterprojekten: BSP. Tomcat oder Watchdog

Apache Top Level Projekt???

# Ant (3)

- Funktionsweise:
  - Konfiguration über XML-Dateien (build.xml)
  - Definition von Targets
  - Erreichen dieser Targets durch Tasks
  - Möglichkeit der Erweiterung mit „custom tasks"
  - Externe Programme aufrufbar
  - Beispiele für vordefinierte Tasks:
    - *Javac* zum Kompilieren von Quell-Code.
    - *CVS* zum Durchführen von CVS-Operationen
    - *JUnit* für automatisierte (JUnit-)Tests.

07.06.2005 Massimo Acquasanta und Tobias Schmidt

Was ist Ant?

Java-basierter Ersatz für plattformspezifisches *make*

Konfiguration über XML-Dateien (*build.xml*)

Definition von *tasks*: Quellcode kompilieren, JAR-Archive erstellen, Deployment, Einbinden von Unit-Tests, CVS bedienen, ...

mehr als 100 Standardtasks vorhanden

einfach erweiterbar (*custom tasks*)

Im Gegensatz zu *make* ist es plattformunabhängig - zumindest solange auf der Zielplattform eine Java-Laufzeitumgebung (JRE) verfügbar ist. Gesteuert wird Ant durch eine XML-Datei, die so genannte *Build-Datei*. In dieser wird zunächst ein *Projekt* definiert, welches *Targets* (englisch für "Ziele") enthält. Diese sind vergleichbar mit Funktionen in Programmiersprachen und enthalten u.a. Aufrufe von *Tasks* (englisch für "Aufgaben"). Ein *Task* ist ein untrennbarer Arbeitsschritt. Zwischen den *Targets* können und sollten Abhängigkeiten definiert werden. Beim Aufrufen eines *Targets* löst Ant diese Abhängigkeiten auf und arbeitet die Targets entsprechend ab. Wenn man ein einzelnes Target definiert hat, welches direkt oder indirekt Abhängigkeiten zu allen anderen Targets hat, so genügt es, dieses aufzurufen und Ant führt dann alle notwendigen Arbeitsschritte in der richtigen Reihenfolge aus. Beim Projekt kann dieses Target als *default* angegeben werden.

Da es sich bei der *Build-Datei* um eine XML-Datei handelt, hängt ihre Bedeutung nicht von Tabulatorzeichen, Leerzeichen oder Pfadtrennzeichen ab, die auf unterschiedlichen Betriebssystemen unterschiedlich definiert sind.

Weiterhin ist Ant ein offenes System. Das bedeutet, dass es z.B. durch die oben genannten Tasks beliebig erweitert werden kann. Außerdem lässt es sich auch in eigene Anwendungen - z.B. Installationsprogramme - einbinden, um die verschiedensten, meist Batch-artigen, Aufgaben zu übernehmen.

**TASKS**

Diese Liste enthält einige eingebaute (engl. "built-in") Tasks von Ant. Die derzeitige Version 1.6.2 enthält über 150 Tasks, wobei man auch eigene Tasks in Java selbst programmieren kann.

*Javac* zum Kompilieren von Quell-Code.

*Copy* zum Kopieren von Dateien.

*Delete* zum Löschen von Dateien oder 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.

Flexibel einsetzbar ist auch der vordefinierte Task *Xslt* zum Umwandeln einer XML-Datei in z.B. eine HTML-Datei unter Verwendung einer gegebenen XSLT-Datei.

**Beispiele für Targets**

Während Tasks als Java-Klassen implementiert sind, werden Targets in XML definiert und rufen Tasks auf. Es können auch eigene Tasks in Form von Java-Klassen erstellt werden.

# Ant (4)

- Ant vs. Make (1)
  - Make
    - Wird in C/C++ Projekte verwendet und in Unix
    - Arbeitet Dateizentriert
      - Bsp:

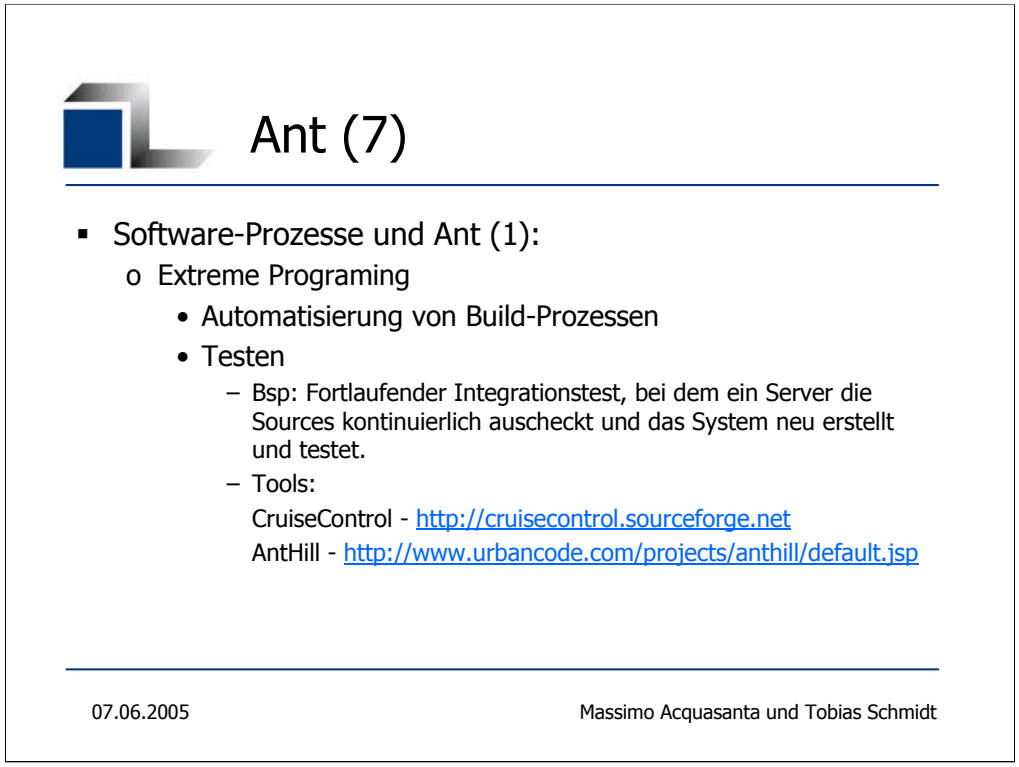
        Header-Datei geändert, so sollte auch die C–Datei, die diese Header-Datei einbindet, neu übersetzt werden
    - Konfiguration komplexer als bei Ant

# Ant (5)

- Ant vs. Make (2)
  - Java besitzt keine Dateiabhängigkeit
  - Kann trotzdem für
    - das Erstellen von Archiven,
    - die Neuübersetzung aller Quellen
    - das Erzeugen von Dokumentation
  - Funktioniert nicht so gut auf Remote-Servern für das Deployen
  - Muss für jede Plattform ein entsprechendes make-file erstellt werden
    - Tool Automake, imake

# Ant (6)

- Wieso nicht Eclipse dafür verwenden?
  - o Bietet keine Möglichkeit ein passendes Skript zu erstellen
  - o Jedes Eclipse müsste individuell eingerichtet werden
  - o Bietet selbst die Möglichkeit Ant-Skripte auszuführen



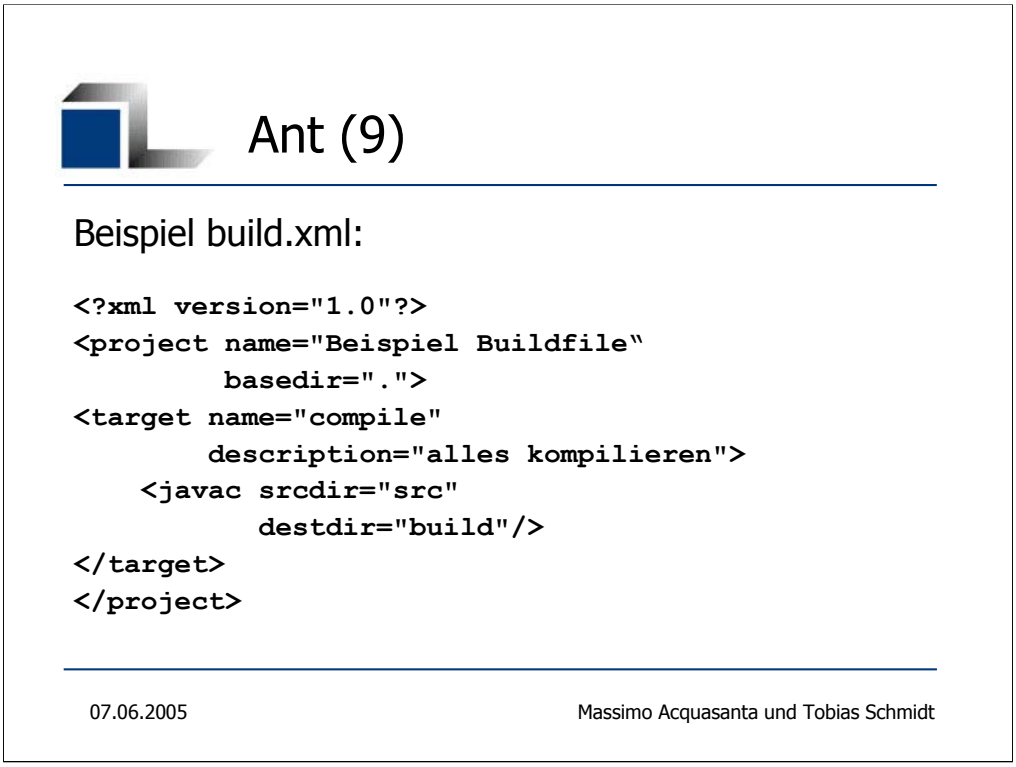

XP-Bsp: Fortlaufender Integrationstest, bei dem ein Server die Sources kontinuierlich auscheckt und das System neu erstellt und testet.

## Ant (8)

- Software-Prozesse und Ant (2):
  - RUP
    - Automatisierung der Kompilierungs- und Deployment-Stufen
    - Dateien ein- und ausgecheckt
      - Revisionskontrolwerkzeug Clear Case
    - Integration von TestSuite von Rational IBM nicht möglich
      - Bsp:
        Visual Test
        Purify



XP-Bsp: Fortlaufender Integrationstest, bei dem ein Server die Sources kontinuierlich auscheckt und das System neu erstellt und testet.




Mit Ant können Entwickler plattformunabhängige Makefiles erstellen. Die Extensible Markup Language (XML) verleiht den Buildfiles eine übersichtliche Struktur, und ihre Syntax ist einfach zu erlernen. Eingebaute Tasks erfüllen so gut wie alle Anforderungen, die Java-Programmierer an ein Maketool stellen. Für speziellere Aufgaben steht dem Entwickler die volle Mächtigkeit von Java zur Verfügung.

Mittlerweile hat die Entwicklung von Ant eine ziemliche Dynamik erreicht, und es ist für viele Open-Source-Projekte im Java-Umfeld zum Buildtool der Wahl geworden (beispielsweise Tomcat, JBoss und Castor). Einige Entwickler haben Ant bereits in ihre Lieblings-IDE integriert - etwa JBuilder, Visual Age for Java oder Netbeans -, und unter dem Namen Antidote wird an einem GUI-Frontend für die Erstellung der Buildfiles gearbeitet. (ka)

# Ant (10)

Ausgabe:

```
Buildfile: D:\prj\xp\build.xml

compile:
    [javac] Compiling 32 source files
            to D:\prj\xp\build

BUILD SUCCESSFUL
Total time: 7 minutes
```

# Ant (11)

Integration von JUnit:

```
<?xml version="1.0"?>
<project name="Ant JUnit demo" basedir=".">
  <target name="junit">
      <junit printsummary="on"
            showoutput="true"
            haltonerror="true">
          <test name="TestPerson"/>
      </junit>
  </target>
</project>
```

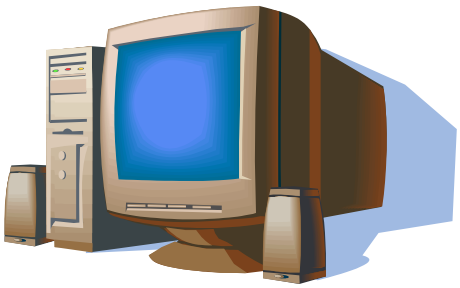
# Ant (12)

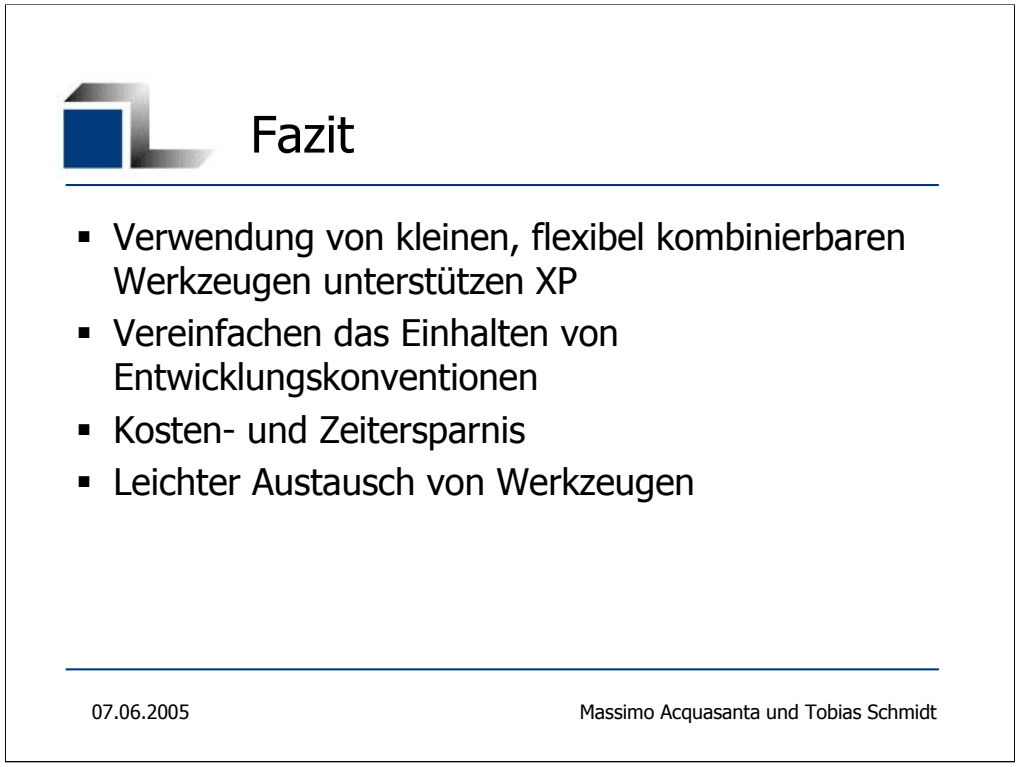
Ausgabe:

```
Buildfile: D:\prj\xp\build.xml
junit:
    [junit] Running TestPerson
    [junit] Tests run: 1, Failures: 0,
            Errors: 1, Time elapsed: 0 sec

BUILD FAILED: D:\prj\xp\build.xml:6:
              Test TestPerson failed
Total time: 266 milliseconds
```

# Ant (13)

## Demonstration






Punkt 1:

XP selbst leichtgewichtiger Prozess, deshalb passt Einsatz von kleinen flexiblen Tools

Punkt 4:

Durch Punkt 1 ist Punkt 1 erfüllt

Punkt 2:

bspw. Variablen Benennungen und Verzeichnisstrukturen

Punkt 3:

bspw. durch den Ensatz automaitisertes testen im ggt zum manuellen testen

# Quellen


- E. Hatcher, S. Loughran: Java-Entwicklung mit Ant, Manning
- A. Hunt, D. Thomas: Unit-Tests mit JUnit, Hanser
- M. Lippert, S. Roock, H. Wolf: Software entwicklen mit XP, dpunkt Verlag
- Markus Vetter, Vorlesung SME – Softwareentwicklungsmethoden und Tools, FH Mannheim WS 2004
- Walter Kriha: Software Design Patterns – Extreme Programming
  http://www.kriha.de
- Wikipedia: Die freie Enzyklopädie
  http://www.wikipedia.de
- Galileo Computing: Openbooks
  http://www.galileocomputing.de
- Computer Base: Das Online Magazin
  http://www.computerbase.de
- Software-Kompetenz: Software-Engineering-Wissensdatenbank
  http://www.software-kompetenz.de