

Software Engineering

Teil 5: Implementierung und objektorientierte
Programmierung

April 2018

Dr. Christian Bartelt



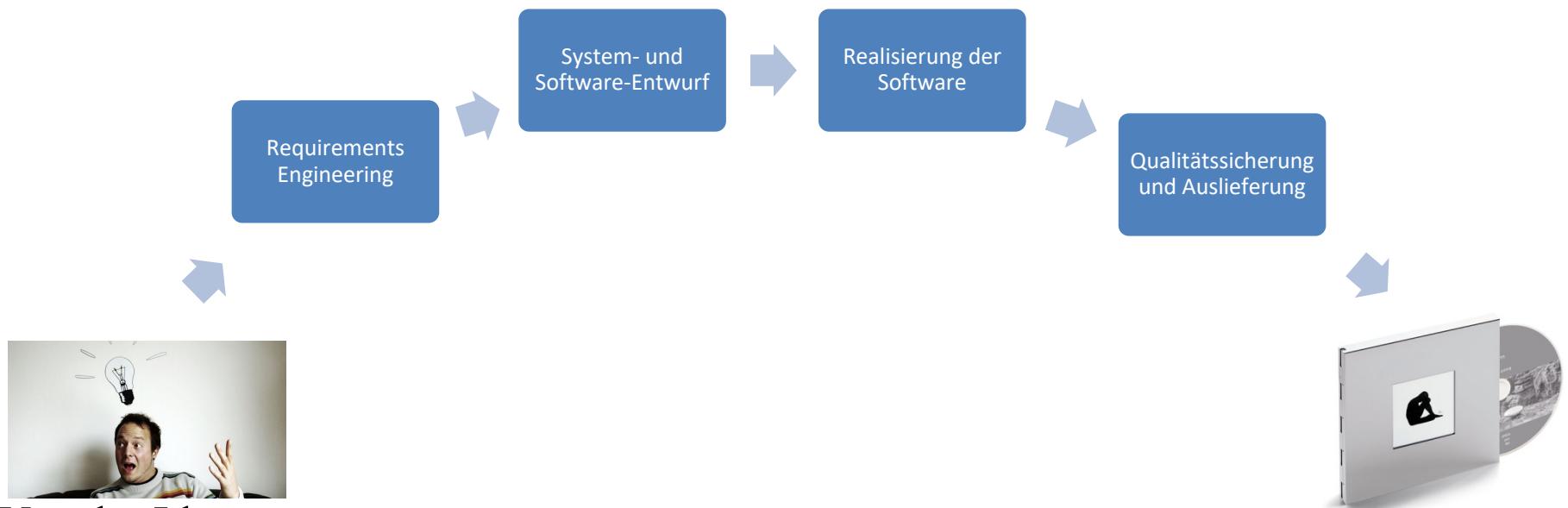
UNIVERSITATEA
BABEŞ-BOLYAI

Universität Mannheim
Institut für Enterprise Systems InES
Schloss
68131 Mannheim / Germany

UNIVERSITY OF
MANNHEIM

Zur Orientierung

Aufgabenbereiche in der Softwareentwicklung



©

Dr. Christian Bartelt

Inhalt

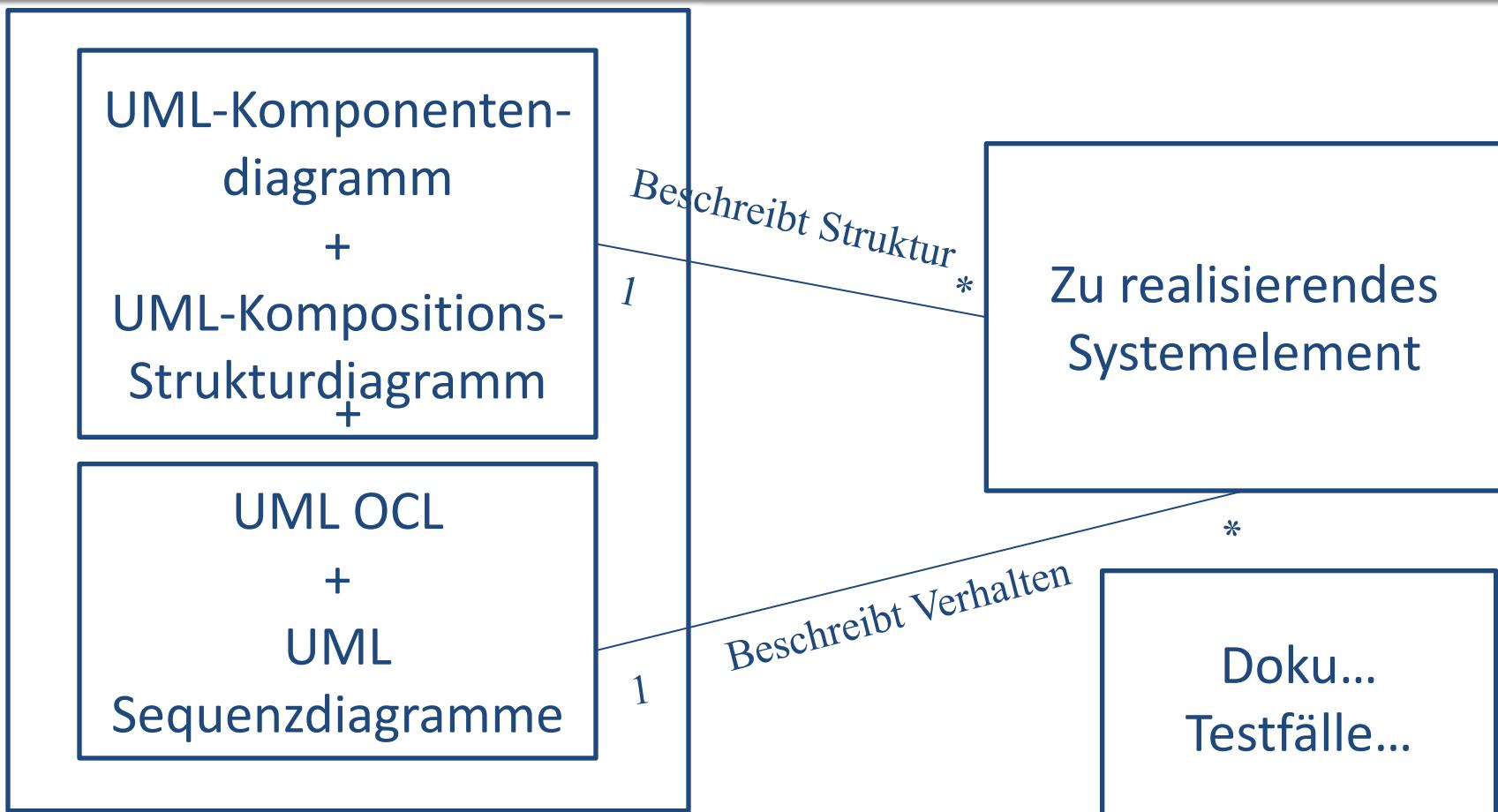
- Motivation und Herausforderungen
- Überblick und Konzepte
- Bestandteile einer Softwareproduktionsumgebung (SPU)
- Vom Design (OOD) zum Programm (OOP)



©

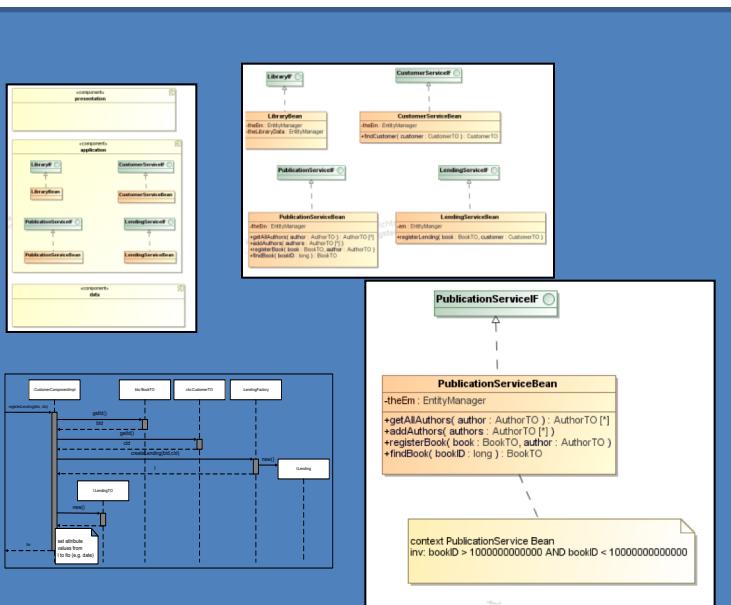
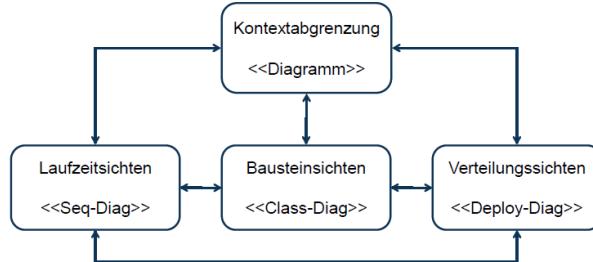
Dr. Christian Bartelt

Aufgabenstellung...



Ausgangssituation: Ergebnis der Architektur (OOD)
Zielsetzung: Wie vom OOD zum OOP und zur Auslieferung

Architektur: Objektorientiertes Design



Implementierung: Objektorientierte Programmierung

Ziel / Herausforderung:
Fehlerfrei vom OOD zum OOP und zum auslieferbaren Programm

```
public void setting() {  
    public void  
    public int  
}  
  
    // execute request  
    testURL = new HttpTestURL(requestProtocol010, requestHost);  
    performanceData.setInfoText(threadStep, testURL);  
    log("[" + threadStep + "] " + testURL.getRequestInfo());  
    testURL.execute();  
    log(" " + testURL.getShortResultText());  
}
```



Inhalt

- Motivation und Herausforderungen
- Überblick und Konzepte
- Bestandteile einer Softwareproduktionsumgebung (SPU)
- Vom Design (OOD) zum Programm (OOP)



©

Dr. Christian Bartelt

Fragestellungen und Teilaufgaben

- 1. Editieren von Code und Modellierung (UML)?
 - -> Code Editor mit Syntax Highlighting, etc.
- 2. Möglichst schematische Transformation von der Spezifikation zur Realisierung durch Code?
 - -> Code-Generierung,
- 3. Überprüfung der Korrektheit und Qualität der Realisierung?
 - -> Junit-Test, CheckStyle, etc.
- 4. Erzeugung von weiteren zusätzlichen Artefakten der Realisierung?
 - -> Java Doc
- 5. Organisation und weitgehende Automatisierung der Arbeiten?
 - -> SVN, Build, etc.



Mögliche Werkzeuge für eine SPU

- **MagicDraw, Topcased: UML-Tool**
- ERwin / DBDesigner: Datenmodellierung
- **Eclipse: OSS-Entwicklungsumgebung**
- Ant / Maven: Build Tool
- CVS, SubVersion, Git: Versions- und Konfigurationsmanagement
- XDoclet: Open Source Code-Generierungs-Engine
- Jalopy: Open Source Tool für die Formatierung von Code
- JDiff: Werkzeug zum Vergleich zweier APIs
- Jenkins: CI-Server, Continous Deployment
- CheckStyle: Überprüfung von Kodierrichtlinien
- Dokumentationswerkzeuge: javadoc und java2html
- JUnit: Open Source Framework zur Automatisierung von Unit Tests für Java
- HTTPUnit: Open Source Framework für das Testen von webbasierten Anwendungen
- JProbe: Kommerzielles Profiling Tool von Sitraka
- WebLoad: Testen der Performance und Skalierbarkeit von webbasierten Anwendungen
- JMeter: Open Source Framework zum Testen der Performance von Anwendungen
- Clover: Kommerzielles Messwerkzeug zur Kodeüberdeckung
- JDepend: Open Source Tool, das auf der Basis von Metriken die Qualität von Java Code misst
- JavaNCSS: Open Source Tool für Messen von Code (lines of code, Anzahl der Klassen)
-



Inhalt

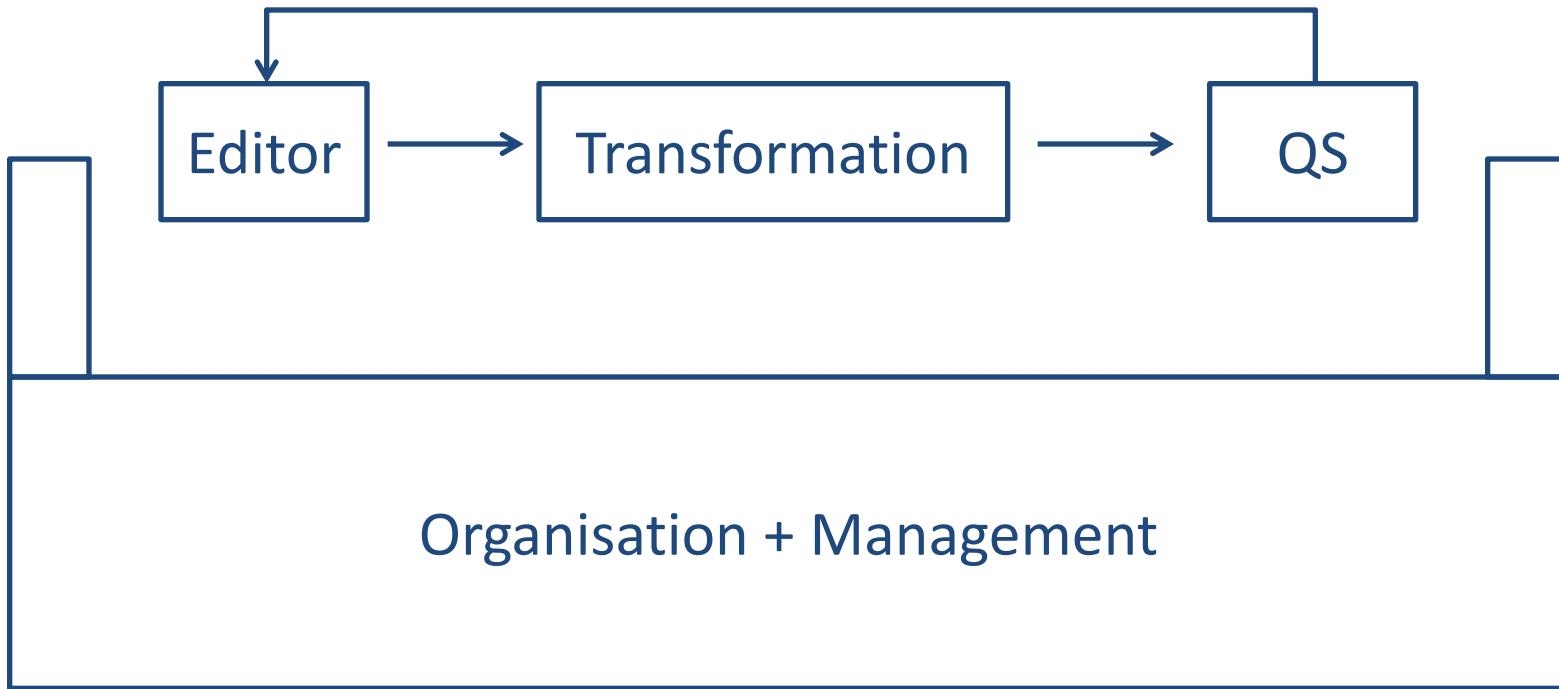
- Motivation und Herausforderungen
 - Überblick und Konzepte
 - Bestandteile einer Softwareproduktionsumgebung (SPU)
 - Editoren für alle Programmteile
 - Generierung von Programmteilen und weiteren Artefakten
 - Überprüfung von Korrektheit und Qualität
 - Unterstützung für Organisation und Management der Entwicklung
 - Vom Design (OOD) zum Programm (OOP)
-



©

Dr. Christian Bartelt

Überblick: Bestandteile einer SPU



Inhalt

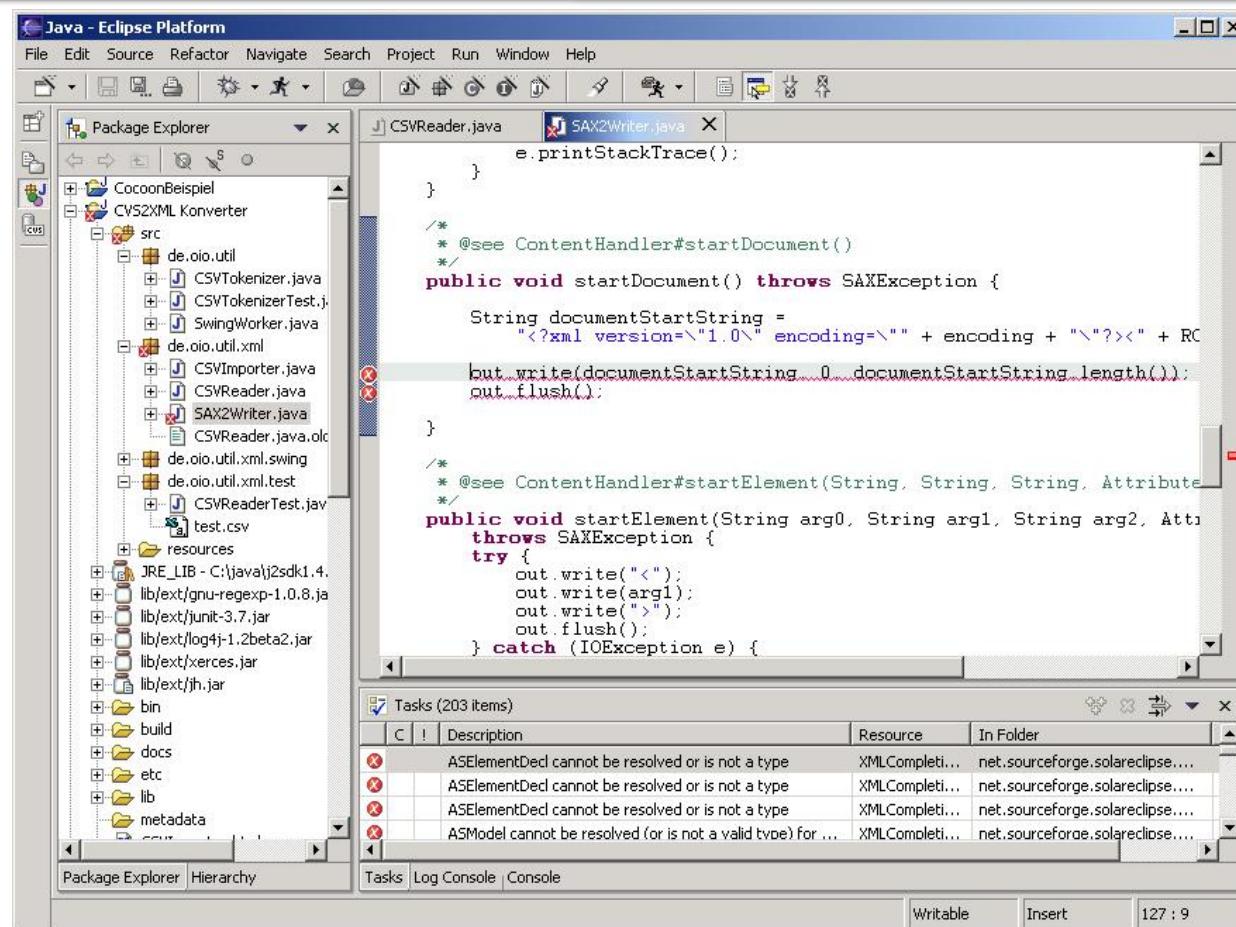
- Motivation und Herausforderungen
 - Überblick und Konzepte
 - Bestandteile einer Softwareproduktionsumgebung (SPU)
 - Editoren für alle Programmteile
 - Generierung von Programmteilen und weiteren Artefakten
 - Überprüfung von Korrektheit und Qualität
 - Unterstützung für Organisation und Management der Entwicklung
 - Vom Design (OOD) zum Programm (OOP)
-



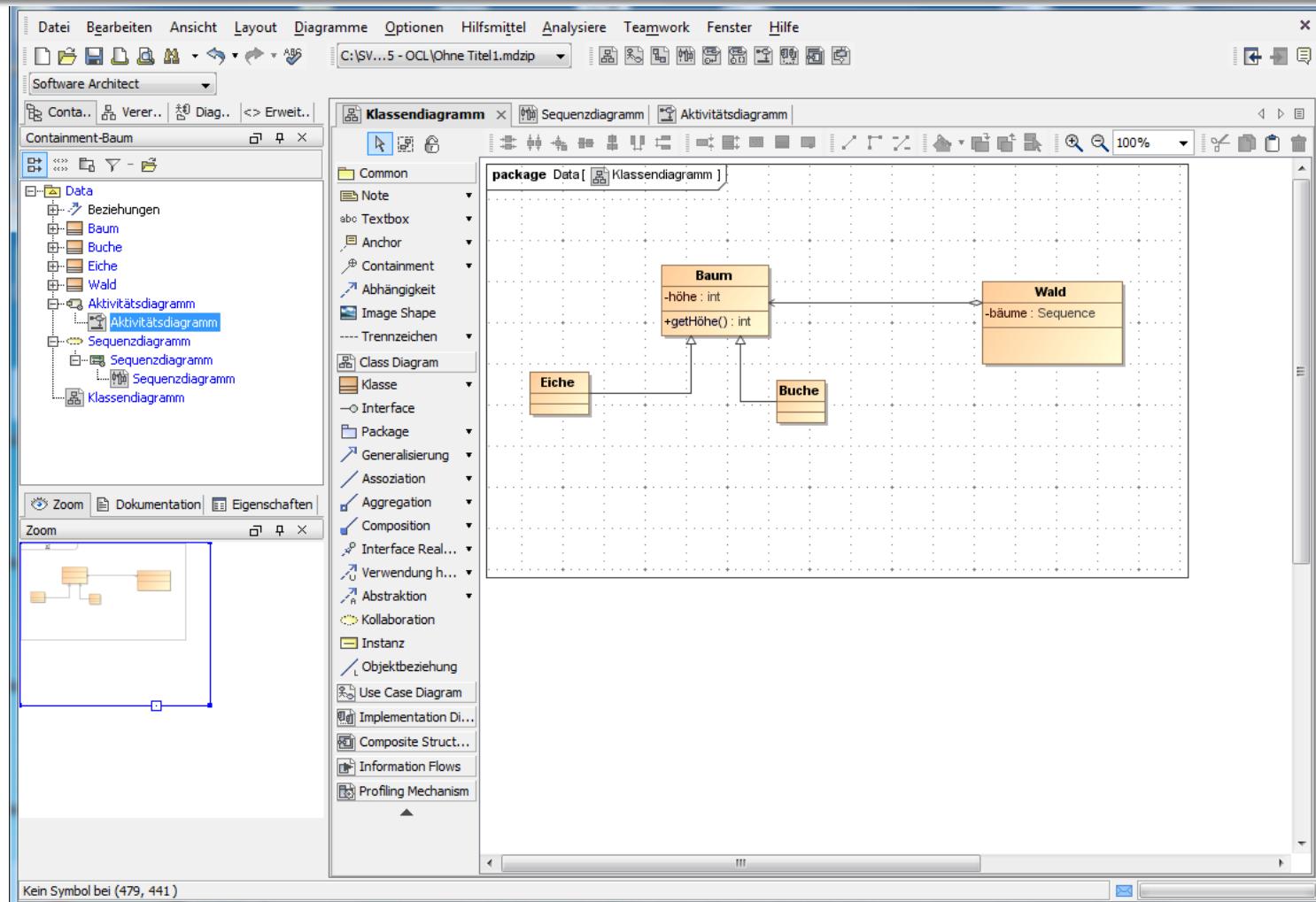
©

Dr. Christian Bartelt

Editoren für Programmcode, Modelle, Konfigurationsdateien, etc. (1)



Editoren für Programmcode, Modelle, Konfigurationsdateien, etc. (2)



©

Dr. Christian Bartelt

Inhalt

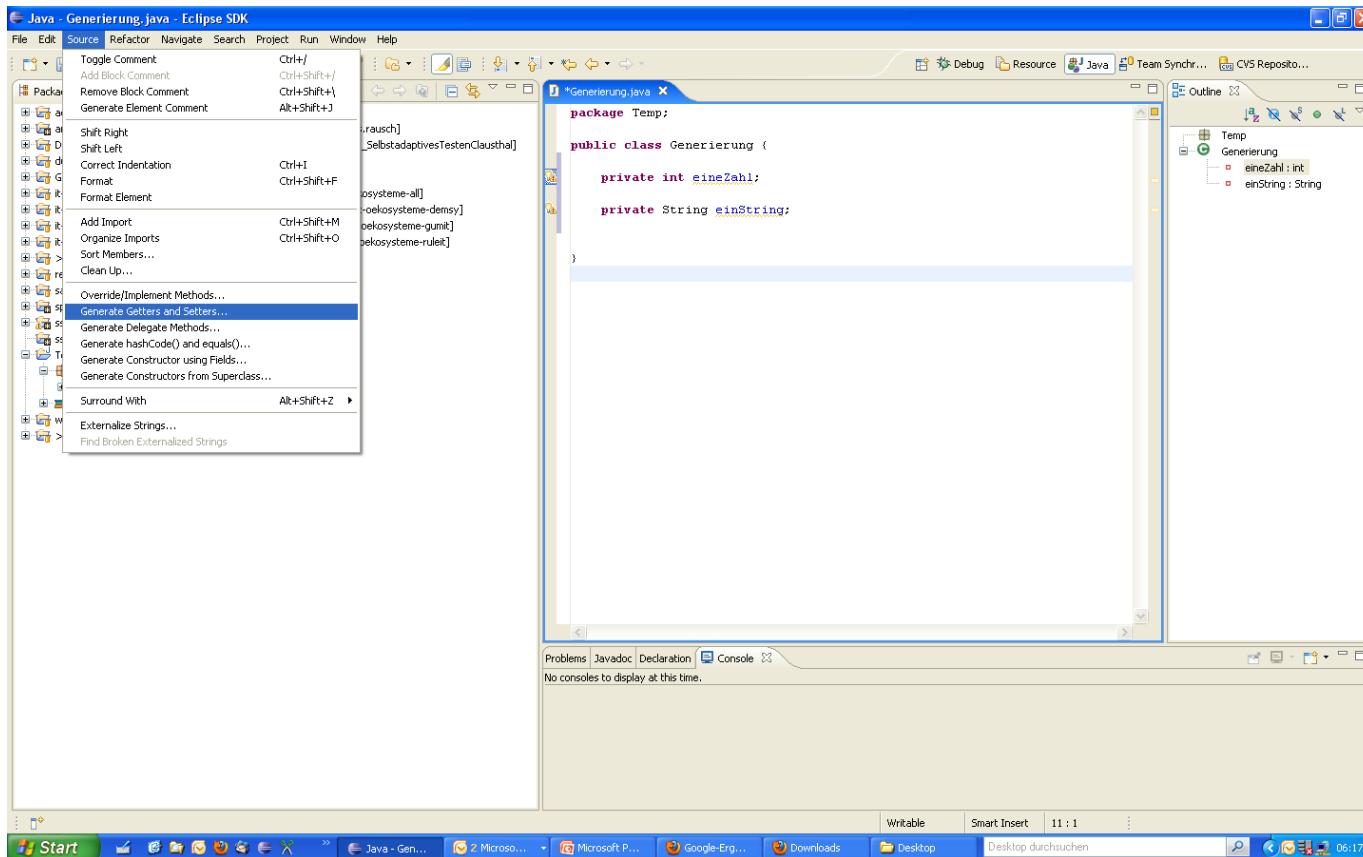
- Motivation und Herausforderungen
 - Überblick und Konzepte
 - Bestandteile einer Softwareproduktionsumgebung (SPU)
 - Editoren für alle Programmteile
 - Generierung von Programmteilen und weiteren Artefakten
 - Überprüfung von Korrektheit und Qualität
 - Unterstützung für Organisation und Management der Entwicklung
 - Vom Design (OOD) zum Programm (OOP)
-



©

Dr. Christian Bartelt

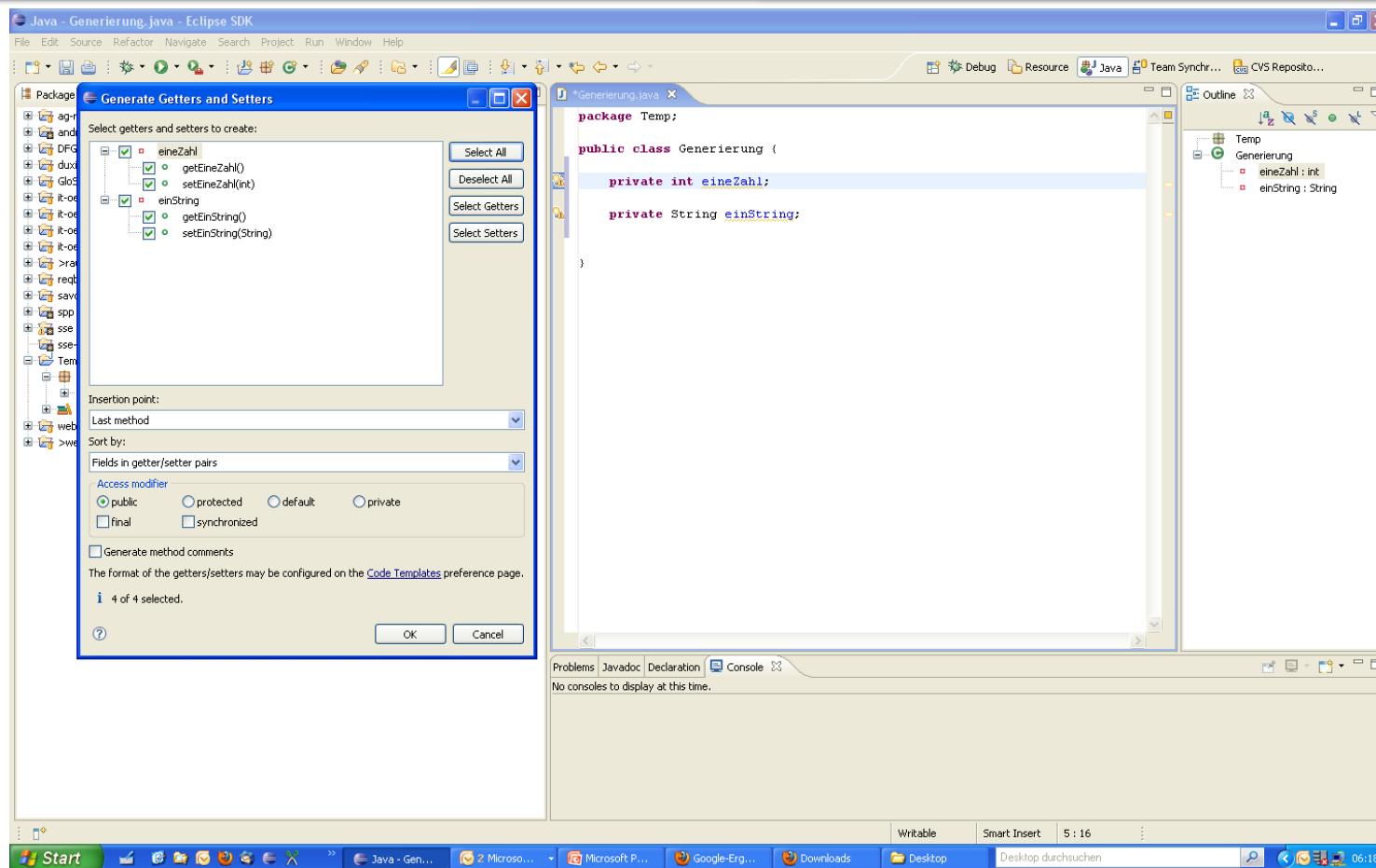
Generierung von Programmteilen (Code, Skripte, etc.) (1)



©

Dr. Christian Bartelt

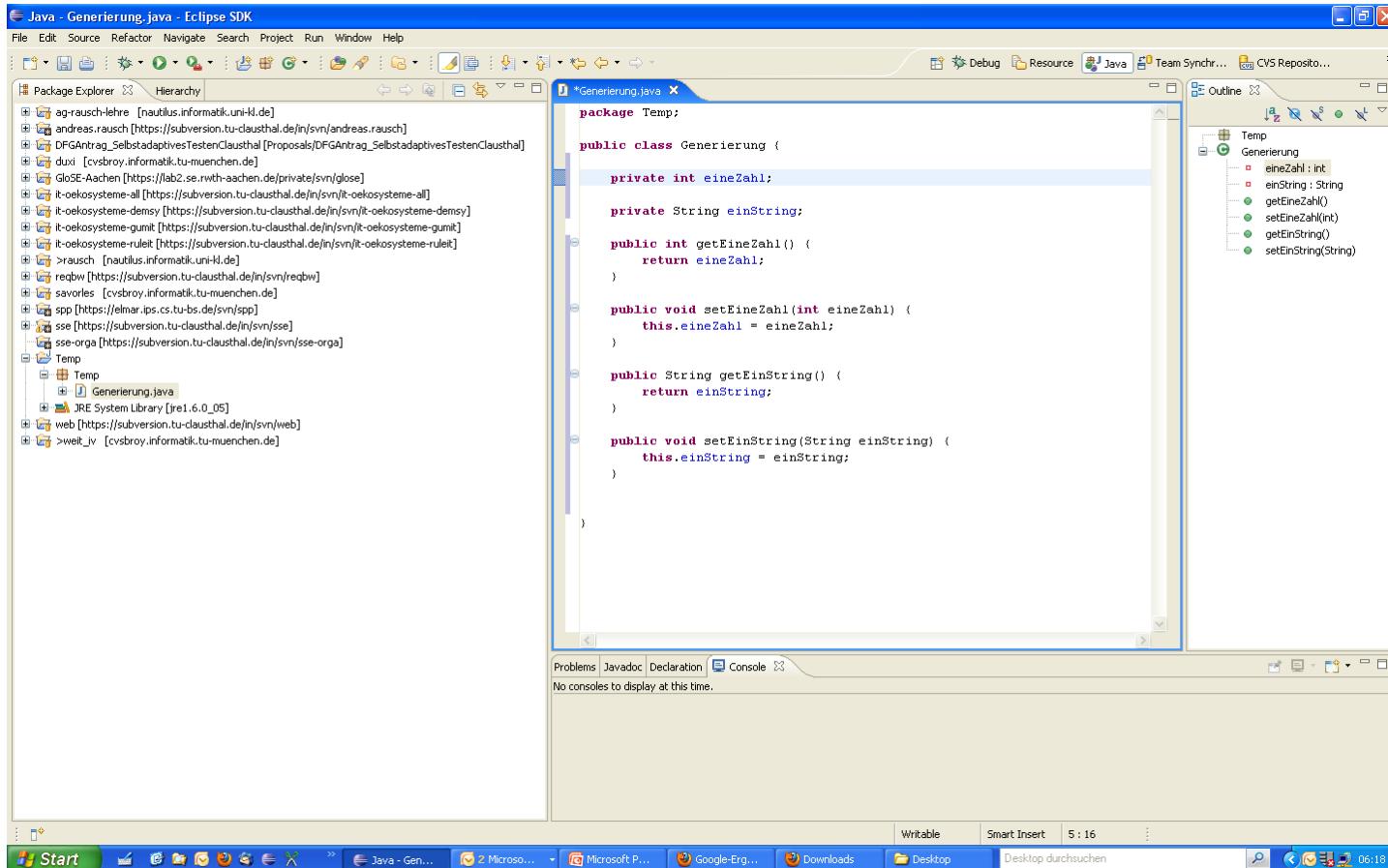
Generierung von Programmteilen (Code, Skripte, etc.) (2)



©

Dr. Christian Bartelt

Generierung von Programmteilen (Code, Skripte, etc.) (3)



©

Dr. Christian Bartelt

Generierung von Programmteilen (Code, Skripte, etc.) (4)

Weitere Werkzeuge zur Generierung von Dokumentation

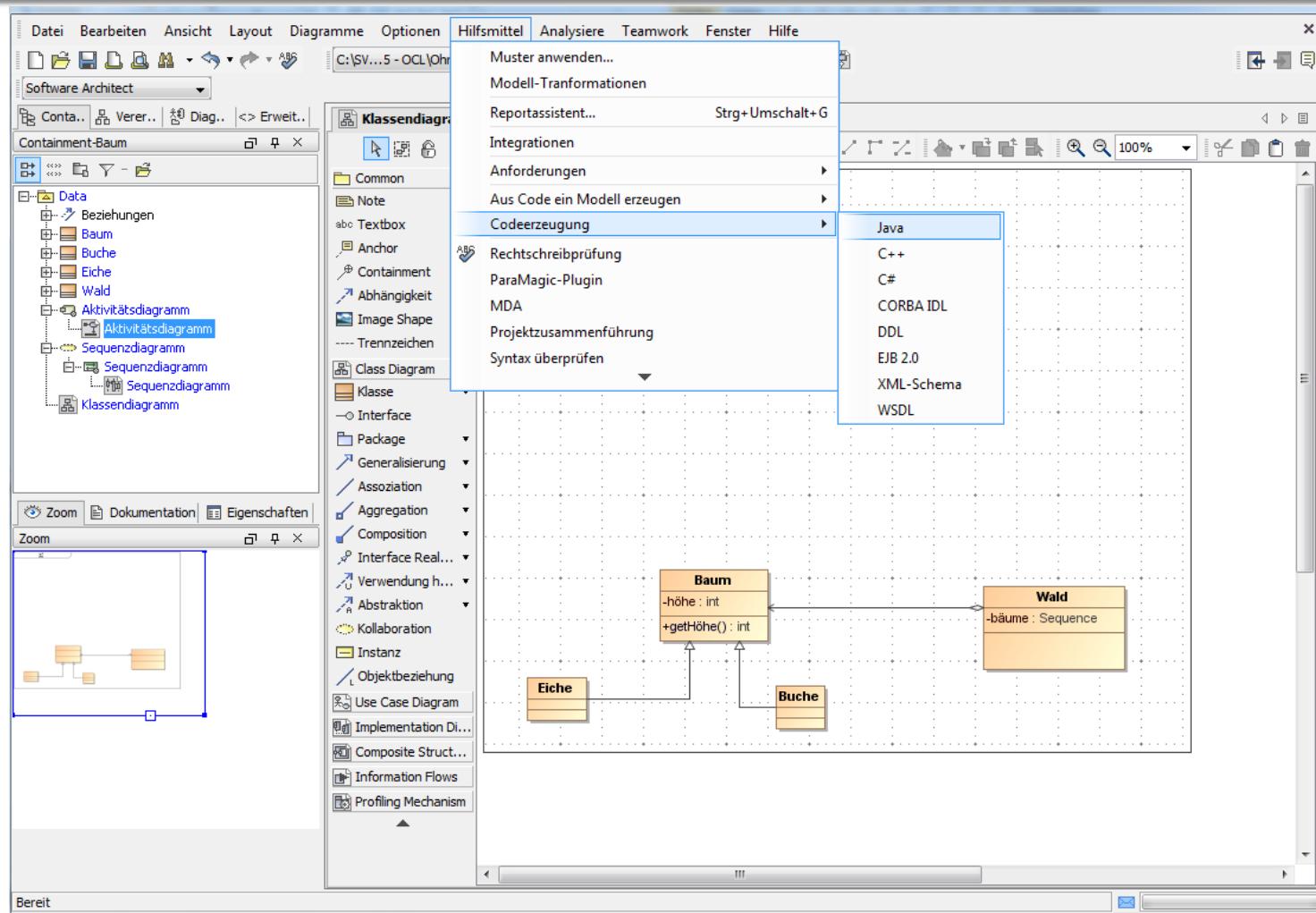
- javadoc
- java2html
- ...



©

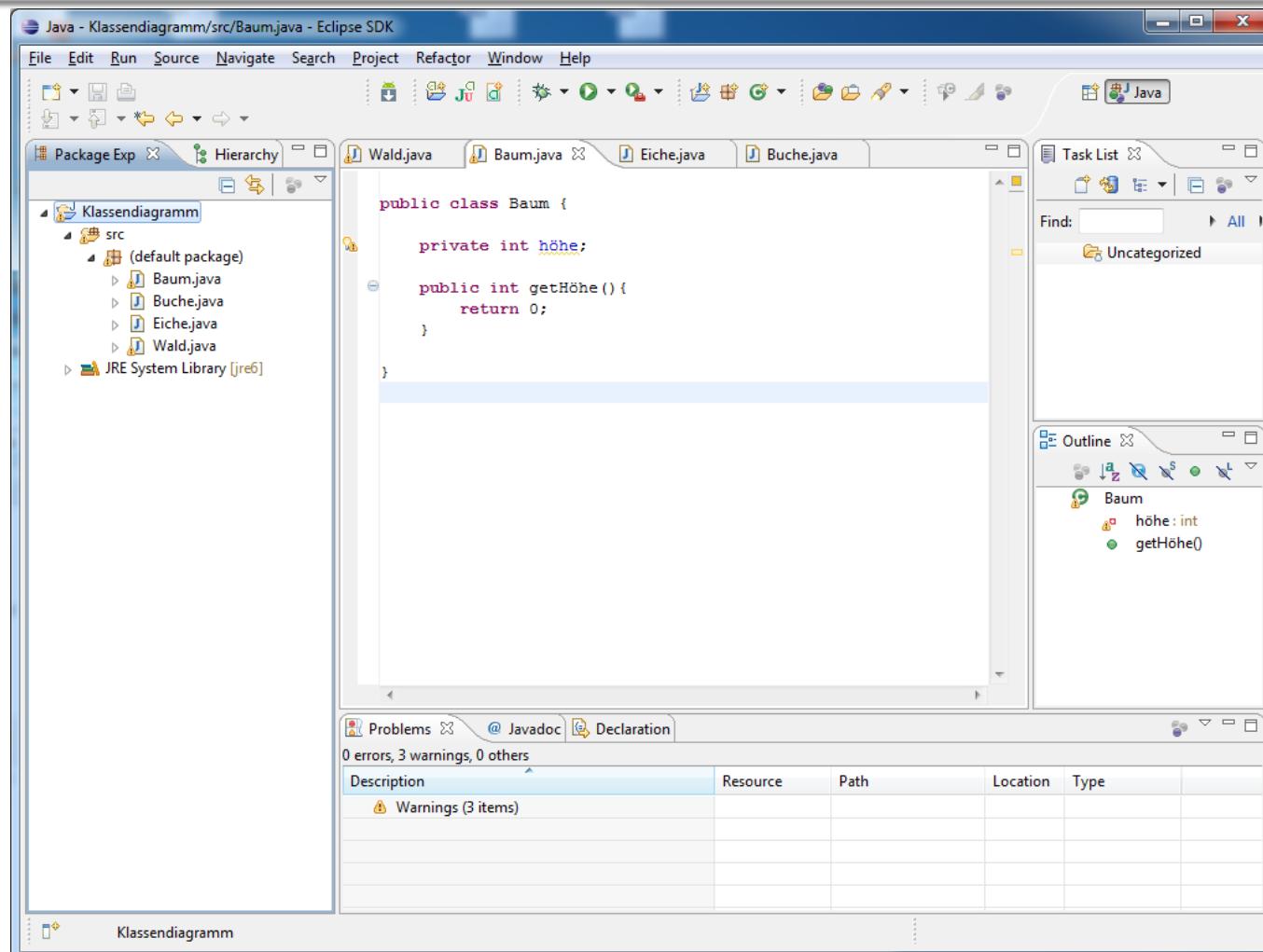
Dr. Christian Bartelt

Generierung von Programmteilen (Code, Skripte, etc.) (4)



©

Generierung von Programmteilen (Code, Skripte, etc.) (5)



©

Dr. Christian Bartelt

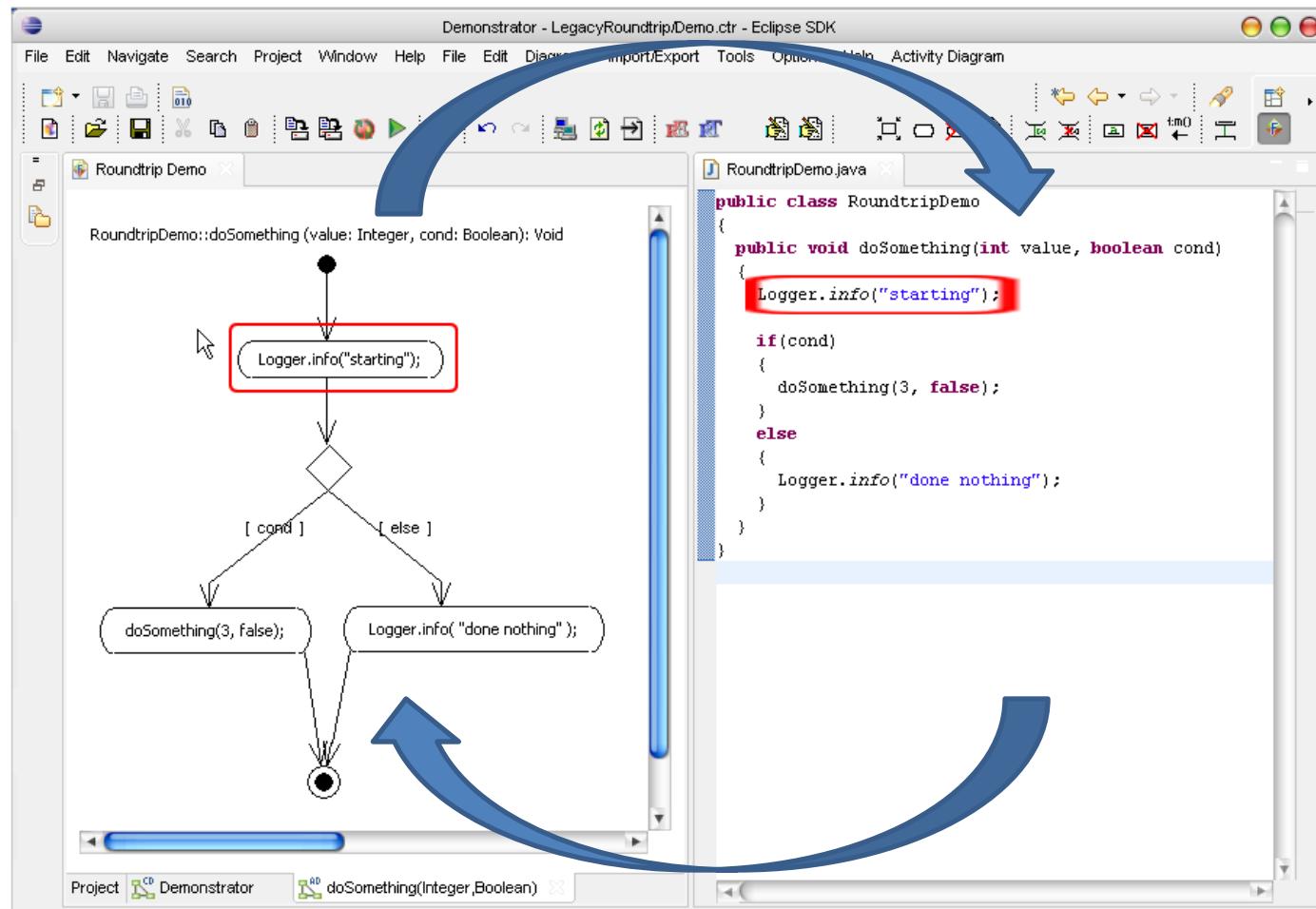
Beispiel: Round-Trip Engineering (1)

- Code-Generierung ist nur die halbe Miete
 - Änderungen am generierten Code oder am Modell (nach Generierung) können zu Inkonsistenzen führen
- Daher: Generierung von Code aus Modellen (Code Generierung) und von Modellen aus Code (Reverse Engineering)
 - Dies ermöglicht simultane Änderungen an Modell und Code (Änderungen an Code / Modell werden automatisch in Modell / Code nachgezogen)
- Viele Tools betrachten nur Struktur, kein Verhalten



Beispiel: Round-Trip Engineering (2)

- Am Beispiel UML-Lab (<http://www.yatta-solutions.com>):



©

Dr. Christian Bartelt

Inhalt

- Motivation und Herausforderungen
 - Überblick und Konzepte
 - Bestandteile einer Softwareproduktionsumgebung (SPU)
 - Editoren für alle Programmteile
 - Generierung von Programmteilen und weiteren Artefakten
 - Überprüfung von Korrektheit und Qualität
 - Unterstützung für Organisation und Management der Entwicklung
 - Vom Design (OOD) zum Programm (OOP)
-



©

Dr. Christian Bartelt

Kommentierung von Code: Metainformation und Kommentare...

```
/*
 * @(#)Observer.java          1.14 98/06/29
 * Copyright 1994-1998 by Sun Microsystems, Inc., ...
 */
package java.util;

/*
 * A class can implement the <code>Observer</code> interface when it
 * wants to be informed of changes in observable objects.
 *
 * @author Chris Warth
 * @version 1.14, 06/29/98
 * @see    java.util.Observable
 * @since  JDK1.0
 */
public interface Observer {
    /**
     * This method is called whenever the observed object is changed. An
     * application calls an <tt>Observable</tt> object's
     * <code>notifyObservers</code> method to have all the object's
     * observers notified of the change.
     *
     * @param o  the observable object.
     * @param arg an argument passed to the
     *            <code>notifyObservers</code> method.
     */
    void update(Observable o, Object arg);
}
```

Beispiel: Klasse

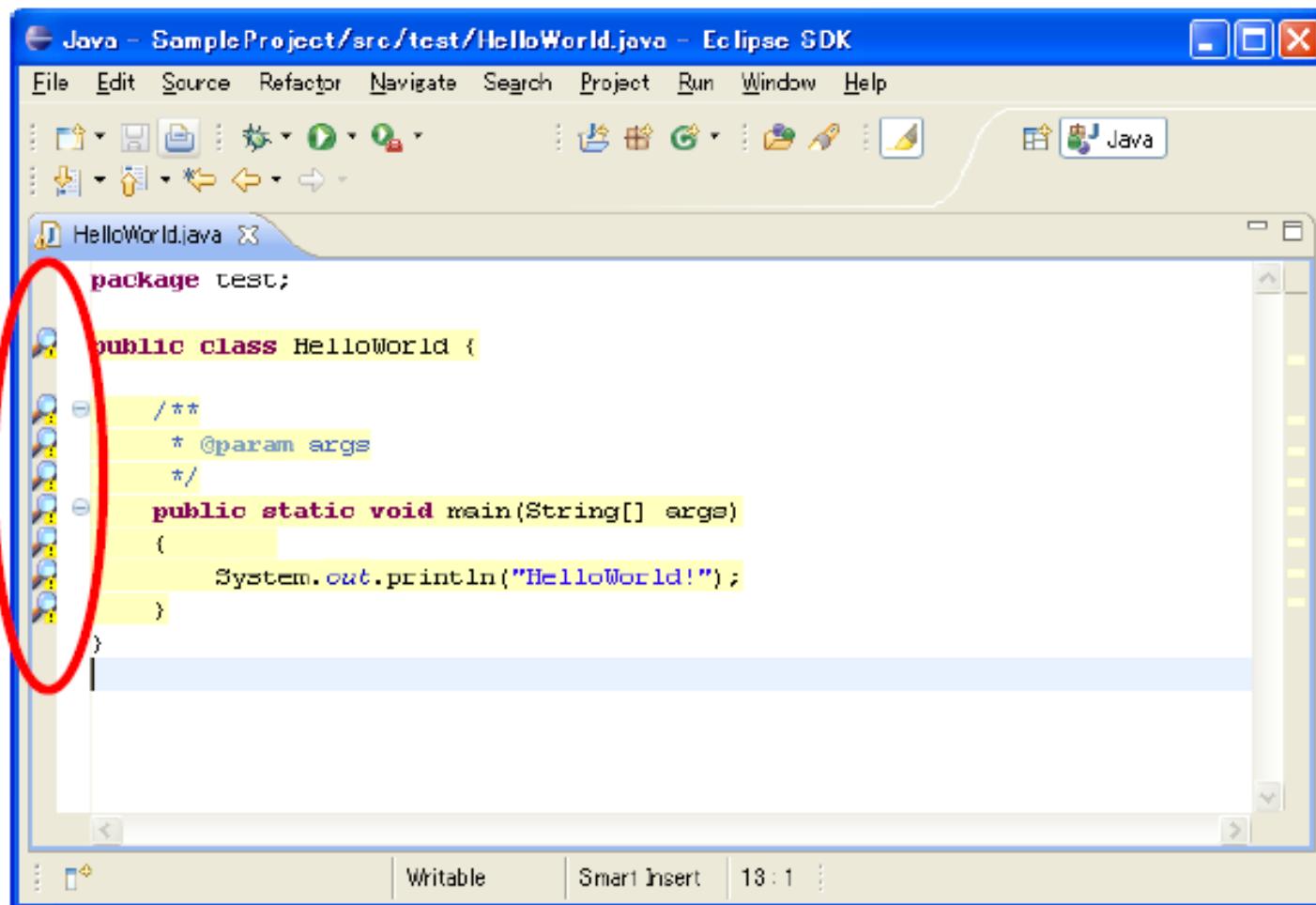
Meta-
Informationen für
Konfig-
Management

Schnittstellen-
Spezifikation



Werkzeuge zur Code-Formatierung...

Checkstyle... (1)



The screenshot shows the Eclipse IDE interface with a Java file named `HelloWorld.java` open. The code is as follows:

```
package test;

public class HelloWorld {
    /**
     * @param args
     */
    public static void main(String[] args)
    {
        System.out.println("HelloWorld!");
    }
}
```

A red circle highlights the vertical margin on the left side of the code editor, where several small icons are displayed, likely representing different code analysis or toolbars.

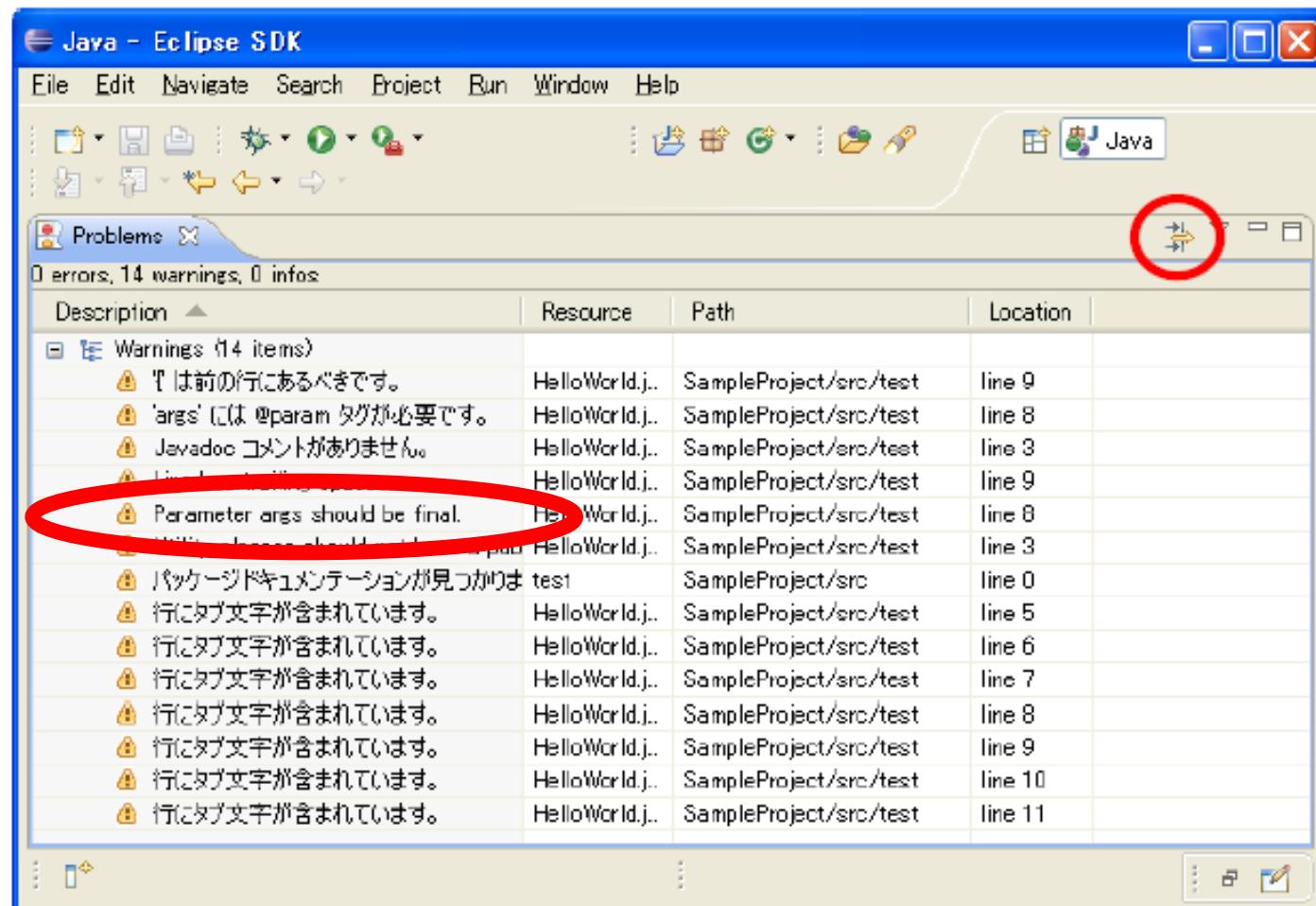


©

Dr. Christian Bartelt

Werkzeuge zur Code-Formatierung...

Checkstyle... (2)



Werkzeuge zur Code-Formatierung...

Checkstyle... (3)

		Rules	Example
Naming Conventions	Classes	<ul style="list-style-type: none"> • Use nouns • Use whole words-avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML). ✓ Begin with upper case letters • First letter of each internal word capitalized 	QueueBlock, ReversalADT, LinkedList ,CustomerAccount, not MaintainCustomerData
	Methods	<ul style="list-style-type: none"> • Use verbs ✓ Begin with lower case letters • First letter of each internal word capitalized 	examine, delete, isEmpty, toString
	Instance Variables	<ul style="list-style-type: none"> • Use nouns, plurals for collection classes ✓ Begin with lower case letters • First letter of each internal word capitalized • Does not start with underscore _ or dollar sign \$ characters, even though both are allowed. • One-character variable names should be avoided except for common temporary variables. 	items, firstAction, list Common temporary variables names Integers: i, j, k, m, and n Characters : c, d, and e
	Constants	<ul style="list-style-type: none"> ✓ All characters in upper case • Words are separated by underscores ("_") 	private static final int MAX_SEQUENCE
	Files	<ul style="list-style-type: none"> ✓ File names should be the same as the (principal) class stored in the file. 	
Layout	Classes & Interfaces	<ul style="list-style-type: none"> ✓ No space between a method name and the parenthesis "(" starting its parameter list ✓ Open brace "{" appears at the end of the same line as the declaration statement ✓ Closing brace "}" starts a line by itself indented to match its corresponding opening statement, except when it is a null statement the "}" should appear immediately after the "}" 	<pre>public class QueueBlock { . . protected Object [] items; . .</pre>



Richtlinien für den Code - Kommentare

- Prinzip der integrierten Dokumentation:
 - Kommentare im Code sind leichter zu warten
 - Kommentare sollten parallel zum Code entstehen
 - "Nach-Dokumentation" funktioniert in der Praxis nie!
 - Werkzeuge zur Generierung von Dokumentation (z.B. javadoc)
- Idealzustand:
 - Kommentare zu Klassen und Methoden stellen eine vollständige und eindeutige Spezifikation des Codes dar
- Kommentare sollen nicht:
 - den Code unlesbar machen
 - z.B. durch Verzerrung des Layouts
 - redundante Information zum Code enthalten
 - Schlechter Kommentar:
 - `i++; // i wird hochgezaehlt`
- Lesbarer kommentarfreier Code ist besser als formal kommentierter, aber unlesbarer Code.



Richtlinien für den Code - Typischer Einsatz von Kommentaren

- "Vorspann" von Paketen, Klassen, Methoden etc.
 - Zweck, Parameter, Ergebnisse, Exceptions
 - Vorbedingungen, Abhängigkeiten (z.B. Plattform), Seiteneffekte
 - Version, Änderungsgeschichte, Status
- Formale Annahmen (assertions):
 - Vorbedingungen, Nachbedingungen
 - Allgemeingültige Annahmen (Invarianten)
- Leseerleichterung
 - Zusammenfassung komplexer Codepassagen
 - Überschriften zur Codegliederung
- Erklärung von einzelnen Besonderheiten des Codes
 - z.B. schwer verständliche Schritte, Seiteneffekte
- Arbeitsnotizen
 - Einschränkungen, bekannte Probleme
 - Offene Stellen ("!!!"), Anregungen, Platzhalter



©

Dr. Christian Bartelt

Richtlinien für den Code - Hinweise zum Verfassen von Kommentaren

- Phrasen statt Sätze: kein Problem!
 - Kürze und Übersicht zählt hier mehr als literarischer Anspruch.
- Deskriptiv (3. Person), nicht preskriptiv (2. Person)
 - Bsp: "Setzt die Kontonummer." statt: "Setze die Kontonummer."
- Unnötigen Rahmentext vermeiden:
 - Bsp.: "Setzt die Kontonummer." statt:
"Diese Methode setzt die Kontonummer"
- Verwendung von "this" bzw. "diese/r/s"
 - Bsp.: "Gets the version of this component." statt:
"Gets the version of the component."
 - Bsp: "Ermittelt die Version dieser Komponente." statt:
"Ermittelt die Version der Komponente."



Richtlinien für den Code - Hinweise zur Formatierung

- Einheitliche Formatierung verwenden !
 - Werkzeuge ("pretty printer", "beautifier")
- Gemäß Schachtelungstiefe einrücken
 - Genau festgelegte Anzahl von Leerzeichen (besser als Tabulatoren!)
 - Formatierungsprobleme bei zu tiefer Schachtelung deuten oft auf Strukturprobleme des Codes hin !
- Leerzeilen verwenden (einheitlich)
 - z.B. vor und nach Methoden
 - Aber: Zusammenhängender Code soll auf einem normalen Bildschirm darstellbar bleiben!
- Leerzeichen verwenden (einheitlich):
 - z.B. Operatoren und Operanden durch ein Leerzeichen trennen
 - z.B. keine Leerzeichen vor Methodenparametern, nach Casts
- Einheitliche Dateistruktur verwenden
 - z.B.: Je Klasse eine .java-Datei, je Package ein Verzeichnis
 - "package"-Statement immer als erste Zeile (noch vor "import")



Richtlinien für den Code - Wahl von Bezeichnern

- Einheitliche Namenskonvention verwenden !
- Bezeichner sollen:
 - natürlicher Sprache entnommen sein (bevorzugt Englisch)
 - Ausnahmen: Schleifenvariablen, manche Größen in Formeln
 - aussagekräftig sein
 - leicht zu merken sein
 - nicht zu lang sein, wenn häufig verwendet
 - Kurze Bezeichner nur bei sehr kleinem Scope (Schleifenvariablen)
- Beispiele:
 - x1, x2, i, j, k
 - customername, CustomerName, customerName, cust_name
 - kontoOeffnen, oeffneKonto, kontoOeffnung, kontoGeoeffnet
 - CONSTANT



©

Dr. Christian Bartelt

Richtlinien für den Code - Beispiele für Namenskonventionen

- Klasse:
 - Substantiv, erster Buchstabe groß, Rest klein
 - Ganze Worte, Zusammensetzung durch Großschreibung
 - Bsp: Account, StandardTemplate
- Methode:
 - Verb, Imperativ (Aufforderung), erster Buchstabe klein
 - Lesen und Schreiben von Attributen mit get/set-Präfix im Namen
 - Bsp: checkAvailability(), doMaintenance(), getDate()
- Konstante:
 - Nur Großbuchstaben, Worte mit "_" zusammengesetzt
 - Standardpräfixe: "MIN_", "MAX_", "DEFAULT_", ...
 - Bsp.: NORTH, BLUE, MIN_WIDTH, MAX_WIDTH,
DEFAULT_SIZE,
- Attribute
 - Mit führendem Underscore
 - Bsp: _availability, _date



Richtlinien für den Code - Änderungsfreundlicher Programmcode (1)

- Wahl von Variablen, Konstanten und Typen orientiert an der fachlichen Aufgabe, nicht an der Implementierung:
 - Gutes Beispiel (C):
 - `typedef char name [nameLength]`
 - `typedef char firstName [firstNameLength]`
 - Schlechtes Beispiel (C):
 - `typedef char string10 [10]`
- Symbolische Konstante statt literale Werte verwenden, wenn spätere Änderung denkbar.
- Algorithmen, Formeln, Standardkonzepte in Methoden/Prozeduren kapseln.
- An den Leser denken:
 - Zusammenhängende Einheit möglichst etwa Größe eines typischen Editorfensters (40-60 Zeilen, 70 Zeichen breit)
 - Text probehalber vorlesen ("Telefon-Test")



©

Dr. Christian Bartelt

Richtlinien für den Code - Änderungsfreundlicher Programmcode (2)

- Strukturierte Programmierung
 - Kein "goto" verwenden (in anderen Sprachen als Java)
 - "switch" nur mit "break"-Anweisung nach jedem Fall
 - "break" nur in "switch"-Anweisungen verwenden
 - "continue" nicht verwenden
 - "return" nur zur Rückgabe des Werts, nicht als Rücksprung
- Übersichtliche Ausdrücke:
 - Möglichst seiteneffektfreie Ausdrücke
 - Schlechtes Bsp.: `y += 12*x++;`
 - Inkrementierung/Dekrementierung besser in separaten Anweisungen
 - Fallunterscheidungsoperator (ternäres "? : ") sparsam einsetzen
- Sichtbarkeitsprüfungen des Compilers ausnutzen:
 - Variablen möglichst lokal und immer "private" deklarieren
 - Wiederverwendung "äußerer" Namen (Verschattung) vermeiden



Fehler im Code durch statische Code-Analyse finden...

```
20⊕  public static String getMD5Hash(String input)
21  {
22
23      MessageDigest md5 = null;
24      try {
25          md5 = MessageDigest.getInstance("MD5");
26      }
27      catch (NoSuchAlgorithmException e)
28      {
29          e.printStackTrace();
30      }
31
32      md5.update(input.getBytes(), 0, input.length());
33
34      String hash = new BigInteger(1, md5.digest()).toString(16);
35
36      while (hash.length() < 32)
37          hash = "0" + hash;
38
39      return hash;
```



Beispiel: Tool findbugs (1)

The screenshot shows the FindBugs interface with the following details:

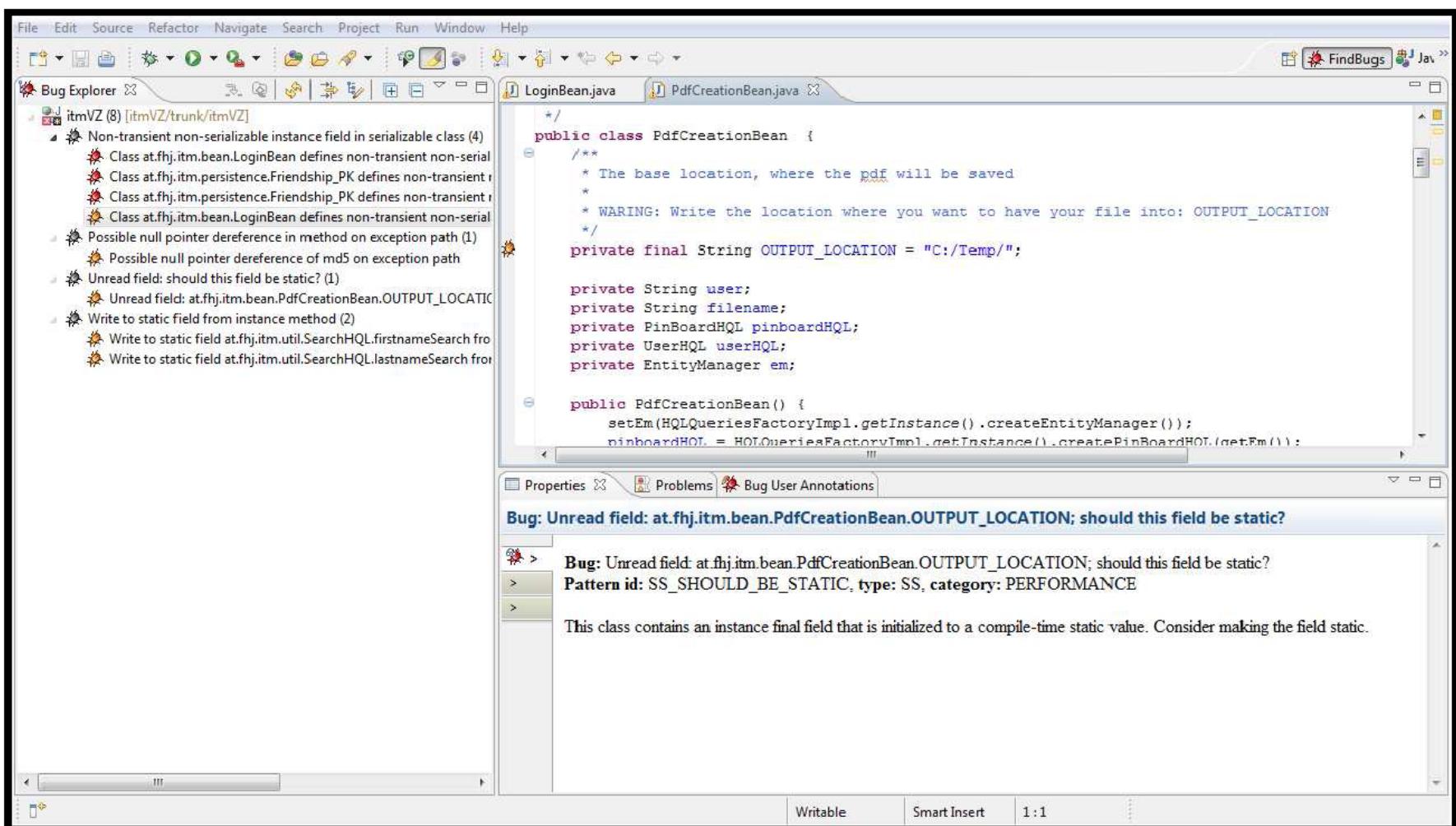
- Menu Bar:** Datei, Bearbeiten, View, Navigation, Bewertung, Hilfe
- Search Bar:** Class search strings: []
- Filter Bar:** Kategorie, Fehler-Art, Fehler-Muster, ↗, Bug Rank, Class
- Bug Summary:** Fehler (13)
 - Correctness (1)
 - Null pointer dereference (1)
 - Possible null pointer dereference in method on exception path (1)
 - Possible null pointer dereference of md5 in at.fh.itm.util.MD5H...
 - Bad practice (3)
 - Incorrect definition of Serializable class (3)
 - Malicious code vulnerability (6)
 - Performance (1)
 - Dodgy (2)
 - Code Editor:** MD5Hash.java in at.fh.itm.util
 - Line 32: md5.update(input.getBytes(), 0, input.length());
 - Message Area:** Possible null pointer dereference of md5 in at.fh.itm.util.MD5Hash.getMD5Hash(String) on exception path
Dereferenced at MD5Hash.java:[line 32]
In method at.fh.itm.util.MD5Hash.getMD5Hash(String) [Zeilen 22 - 39]
Value loaded from md5
Known null at MD5Hash.java:[line 29]
 - Description of Error:** Possible null pointer dereference in method on exception path
A reference value which is null on some exception control path is dereferenced here. This may lead to a NullPointerException when the code is executed. Note that because FindBugs currently does not prune infeasible exception paths, this may be a false warning.
Also note that FindBugs considers the default case of a switch statement to be an exception path, since the default case is often infeasible.



©

Dr. Christian Bartelt

Beispiel: Tool findbugs als eclipse Plugin



©

Dr. Christian Bartelt

Was findbugs noch alles leistet...

Correctness: Wichtigste Kategorie, enthält offensichtliche Fehler im Code

Bad Practice: Verletzung von empfohlener, grundlegender Codeanwendung

Dodgy: fragwürdige, verwirrende Codestellen

Multithreaded Correctness: siehe Correctness, nur bezogen auf Multithreading

Performance: Codestellen, die die Performance verschlechtern

Security: Codestellen, die ein Sicherheitsrisiko darstellen (z.B. Passwort mitten im Quelltext)

Malicious Code Vulnerability: Verletzbarkeit durch bösartigen Code

Internationalization: z.B. Probleme bei verschiedenen Encodings und internationalen Sonderzeichen

Experimental: enthält noch nicht endgültige Bug Patterns, werden gerade entwickelt



Inhalt

- Motivation und Herausforderungen
 - Überblick und Konzepte
 - Bestandteile einer Softwareproduktionsumgebung (SPU)
 - Editoren für alle Programmteile
 - Generierung von Programmteilen und weiteren Artefakten
 - Überprüfung von Korrektheit und Qualität
 - Unterstützung für Organisation und Management der Entwicklung
 - Vom Design (OOD) zum Programm (OOP)
-



©

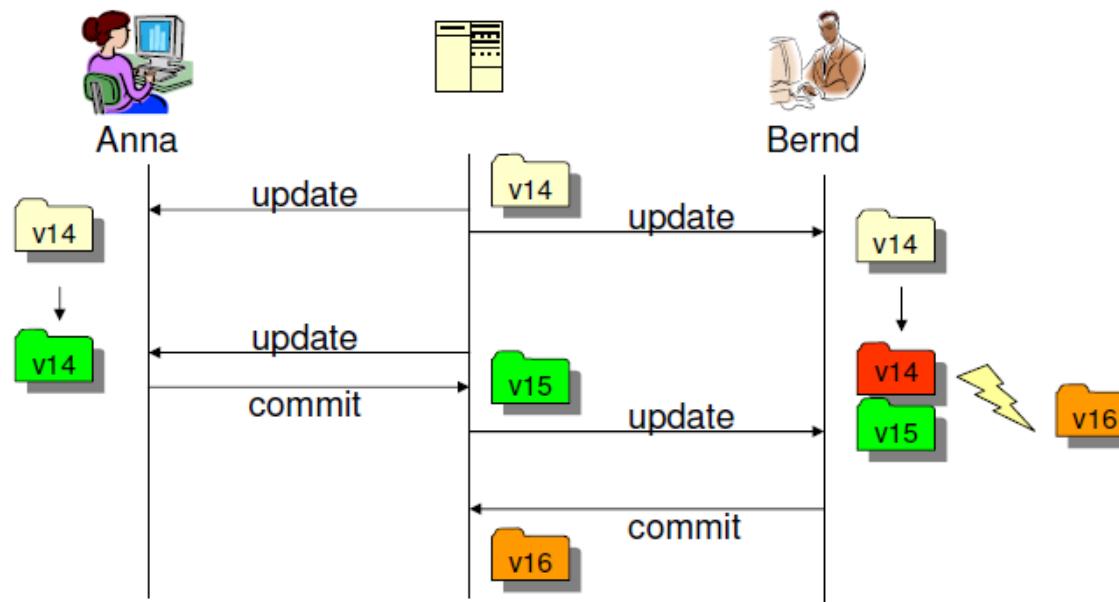
Dr. Christian Bartelt

Gemeinsames Dateiarchiv mit Versionierung

Beispiel: Subversion (1)

Das Problem....

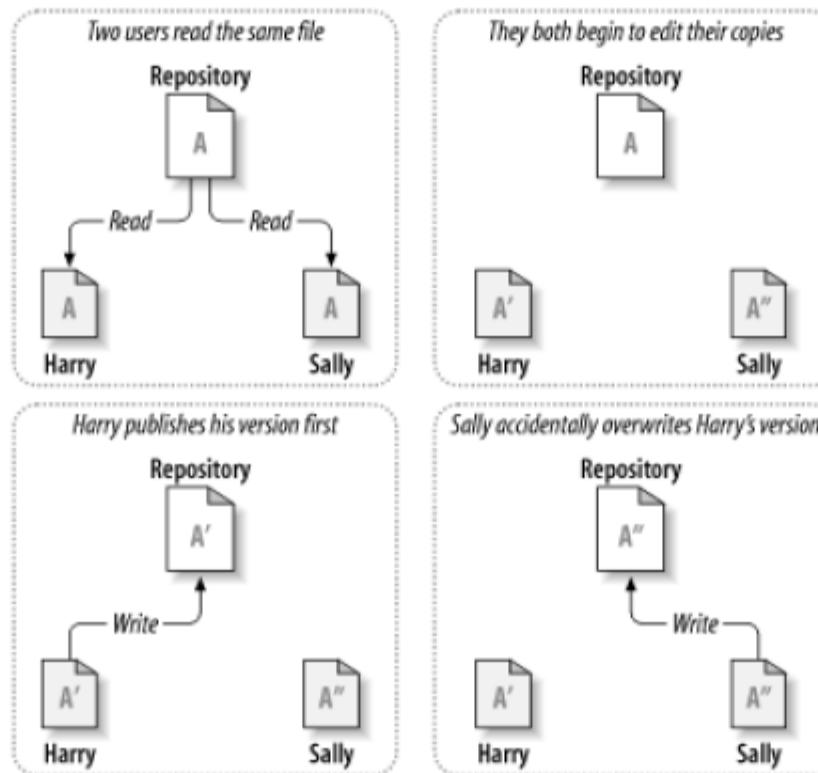
- Parallele Bearbeitung von Quellcode wird problematisch, wenn zwei Personen gleichzeitig dieselbe Datei ändern.



Gemeinsames Dateiarchiv mit Versionierung

Beispiel: Subversion (2)

Das Problem....



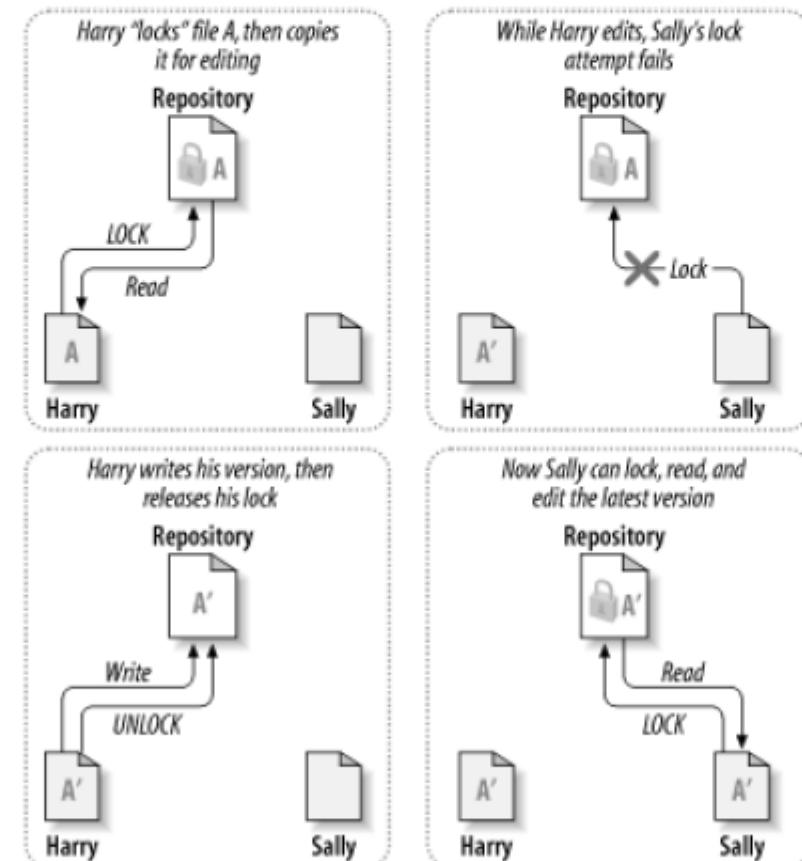
©

Dr. Christian Bartelt

Gemeinsames Dateiarchiv mit Versionierung

Beispiel: Subversion (3)

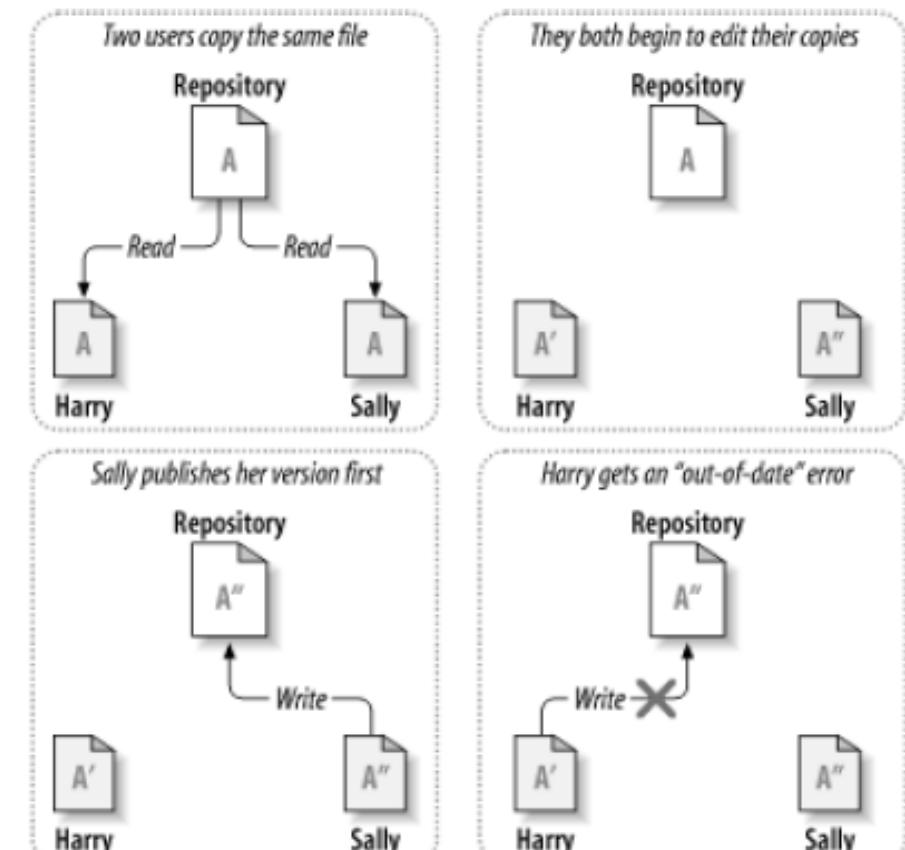
1. Lösung: lock-modify-unlock solution



Gemeinsames Dateiarchiv mit Versionierung

Beispiel: Subversion (4)

2. Lösung: copy-modify-merge solution



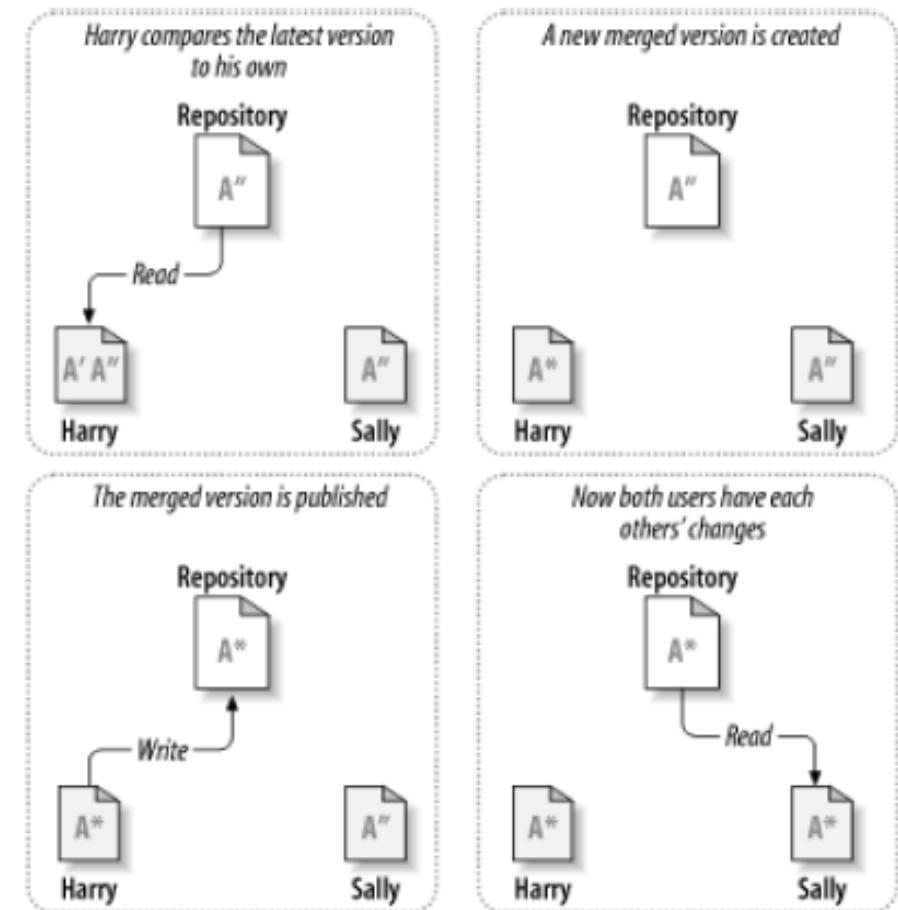
©

Dr. Christian Bartelt

Gemeinsames Dateiarchiv mit Versionierung

Beispiel: Subversion (5)

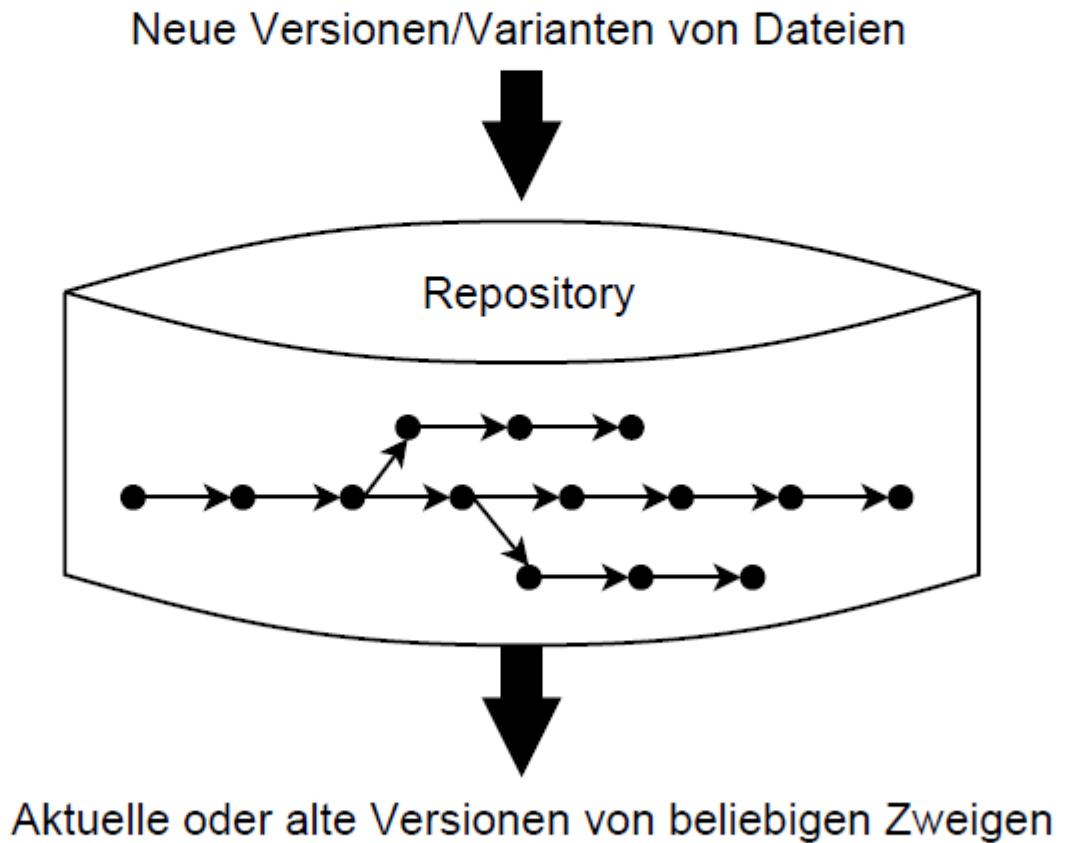
2. Lösung (cond.):
copy-modify-merge
solution



Gemeinsames Dateiarchiv mit Versionierung

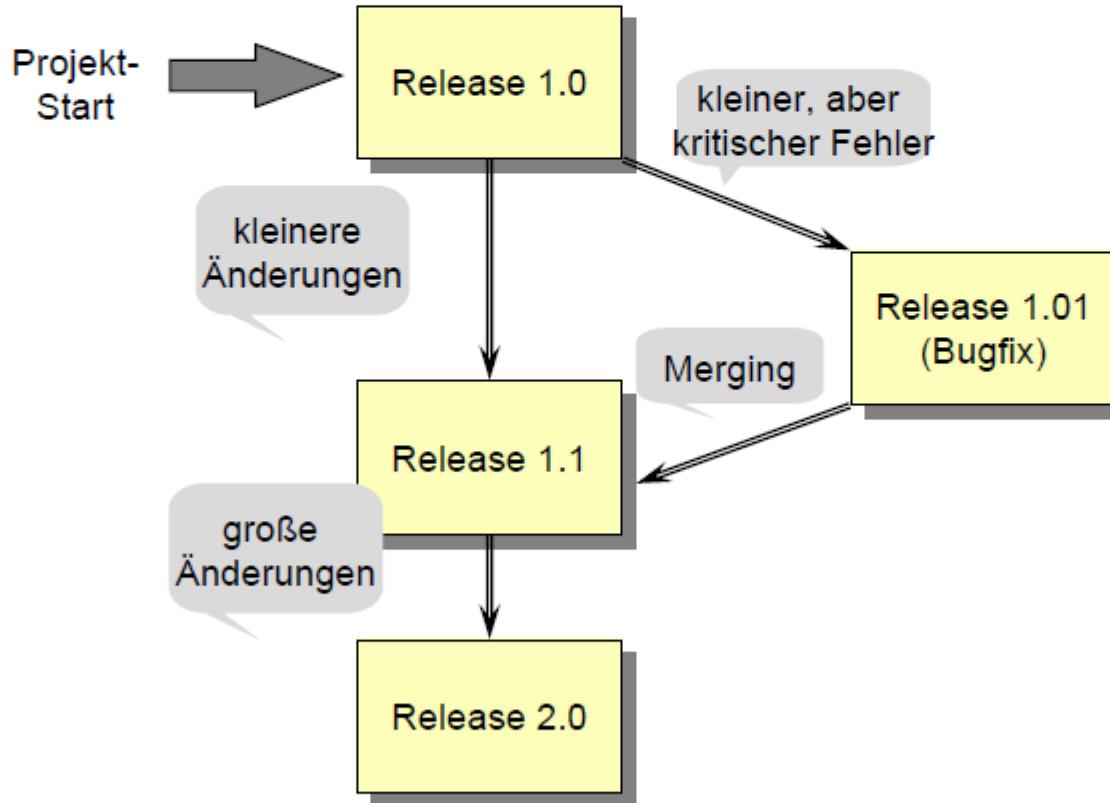
Beispiel: Subversion (6)

Von einer Datei kann es nicht nur einen Versionsstrang geben, sondern auch u.U. eine Reihe von Nebensträngen (Branches). Diese können auch wieder zusammengeführt werden!



Gemeinsames Dateiarchiv mit Versionierung

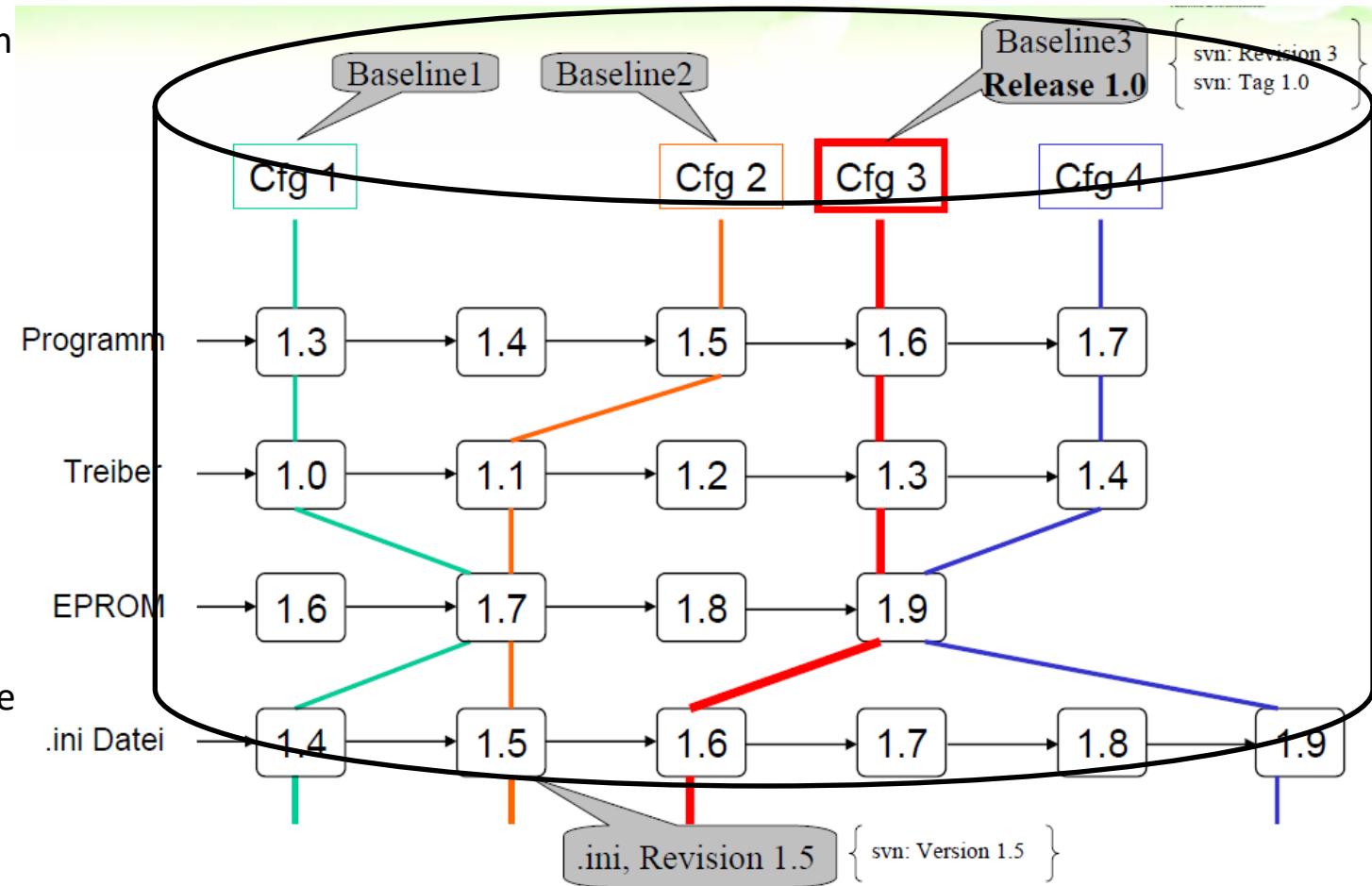
Beispiel: Subversion (7)



Gemeinsames Dateiarchiv mit Versionierung

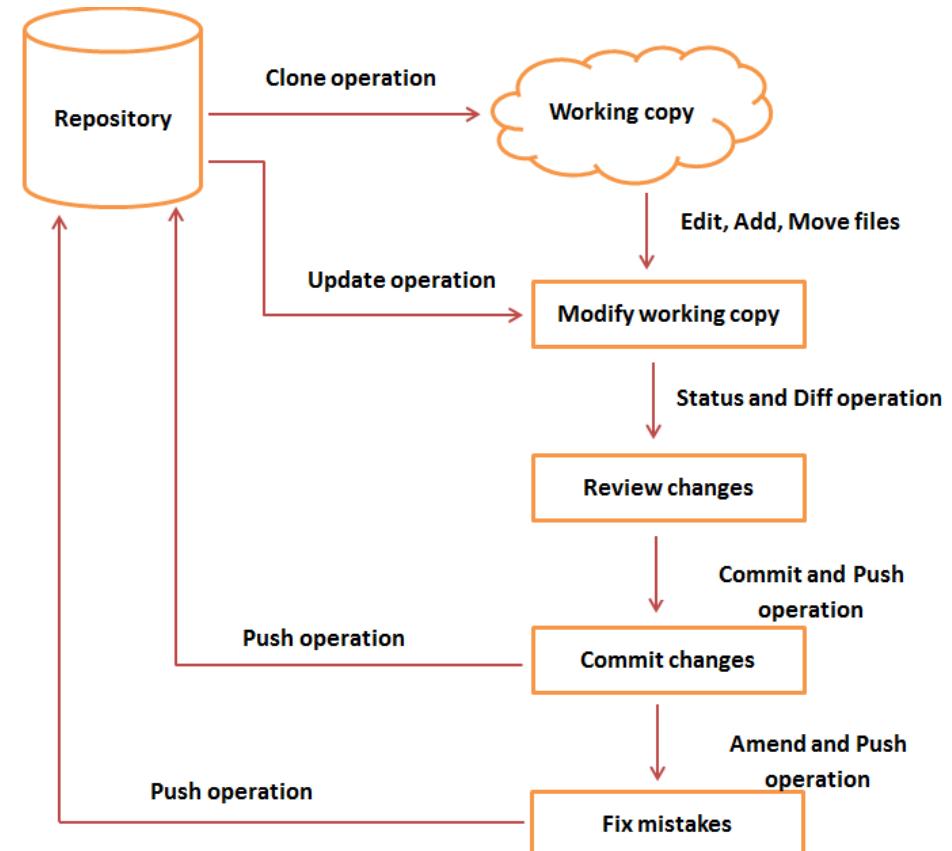
Beispiel: Subversion (8)

In einem Repository sind eine Menge von Datei-Versionssträngen abgelegt. Eine Konfiguration (Baseline) umfasst eine Menge von Dateien in einer Version! In SVN kann man hierzu einen Tag definieren! Mit Konfigurationen kann man ganze Auslieferungsstände nachvollziehbar markieren!



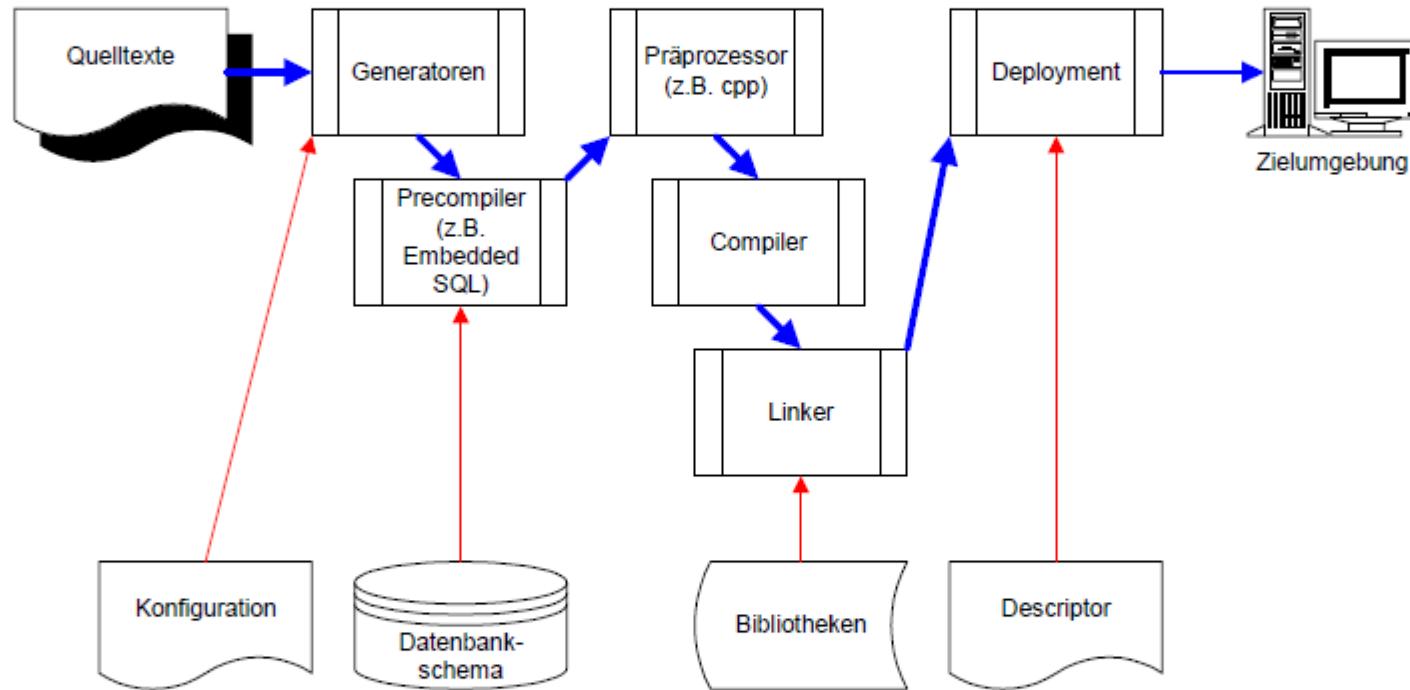
Dezentrale Versionierung mit Git

- Kostenlos (Open Source)
- Schnell
- Einfach (Branching)
- Lokale Repositories als Backups
- Sicher (SHA1)



Der Build-Prozess

Ein Beispiel



Der Build-Prozess umfasst, abhängig von der Art der Quelltexte, viele Schritte, die von weiteren Einstellungen beeinflusst werden.

Der Build-Prozess

Komplexität des Build-Prozesses

- Build-Prozess umfasst viele Schritte
 - Abläufe hängen ab von der Art der Dateien
 - Es bestehen Abhängigkeiten zwischen den betroffenen Dateien (sowohl Ausgangs- als auch Zwischendateien)
 - Beim Build-Prozess sollen meistens, aufsetzend auf einem alten Stand, nur die notwendigen Schritte für die Erzeugung des neuen Stands durchgeführt werden
- Automatisierung notwendig



Der Build-Prozess

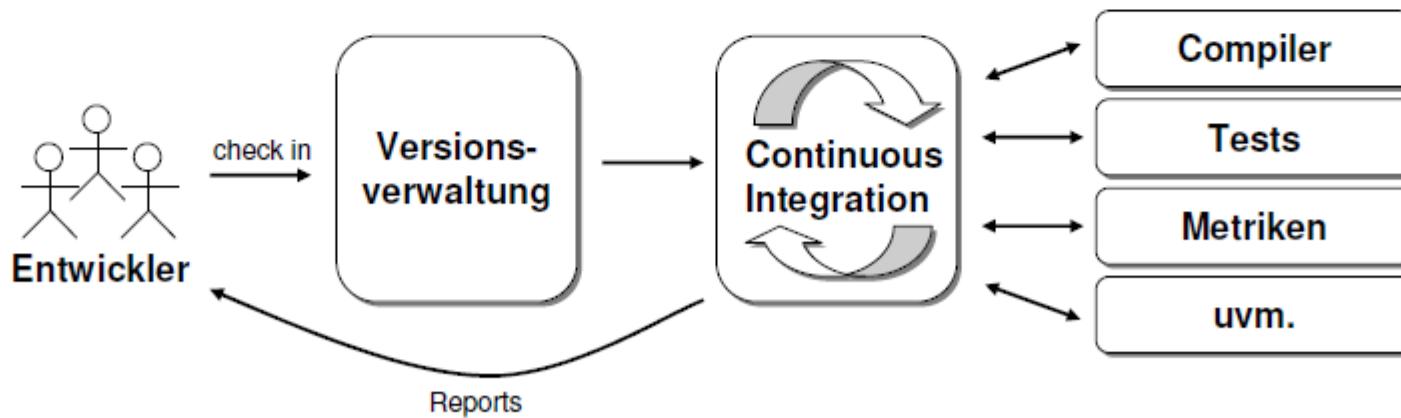
Automatisierung durch Werkzeuge

- Das Tool für die Automatisierung muss
 - die Schritte bezogen auf die Dateiarten beherrschen
 - die Abhängigkeiten zwischen den Dateien kennen
 - (möglichst universell einsetzbar sein)
 - Das bekannteste Tool: „make“
 - alt bekannt, flexibel konfigurierbar über Makefiles
 - arbeitet hinter den Kulissen vieler IDEs
 - Sukzessive Ablösung durch ANT (another neat tool)
 - Entstanden bei/für die Entwicklung von Tomcat
 - Seit Januar 2000 als eigenständiges Tool verfügbar
 - Das Build Tool der Java-Gemeinde
-
- Maven ist in Eclipse mit installiertem SDK bereits enthalten



Und dann alles zusammen bringen: Continuous Integration (1)

- Kontinuierliche Integration (Continuous Integration)
- Fortlaufende Neubildung des Softwaresystems bei jedem *Checkin* in der Versionsverwaltung
- Ausführung von Tests
- Ermittlung von Metriken
- Erzeugung von Reports (z.B. per Webfrontend oder E-Mail)
- Erzeugung von Dokumentation (z.B. Java-Doc oder LaTex-basiert)



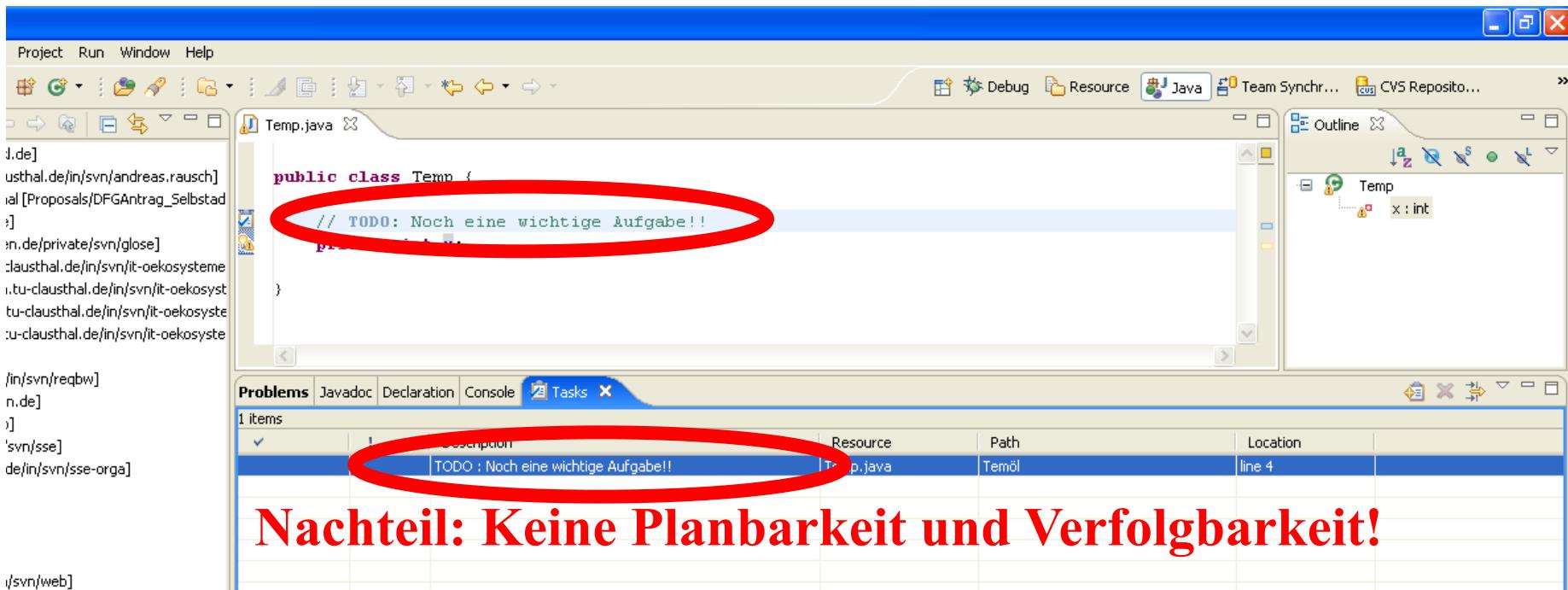
Und dann alles zusammen bringen: Continuous Integration (2)

- Beispiel: CruiseControl (CC)
 - Entwickler commiten Überarbeitungen ins SVN
 - CC überprüft in Intervallen, ob sich Daten im SVN geändert haben
 - Wenn es Änderungen gibt:
 - Build anstoßen
 - Buildergebnisse ablegen, Logging, etc. (Beispiele: jar, unit test report, code coverage report, code quality metrics)
 - Benachrichtigung (z.B. E-Mail, IM)



Organisation der Entwicklungsaufgaben

- Wenn mehrere Menschen an einem Projekt zusammenarbeiten, dann muss die Arbeit organisiert werden...
- Wer macht was, etc....
- Einfache Variante: Eclipse TODO Marker...



Nachteil: Keine Planbarkeit und Verfolgbarkeit!



©

Dr. Christian Bartelt

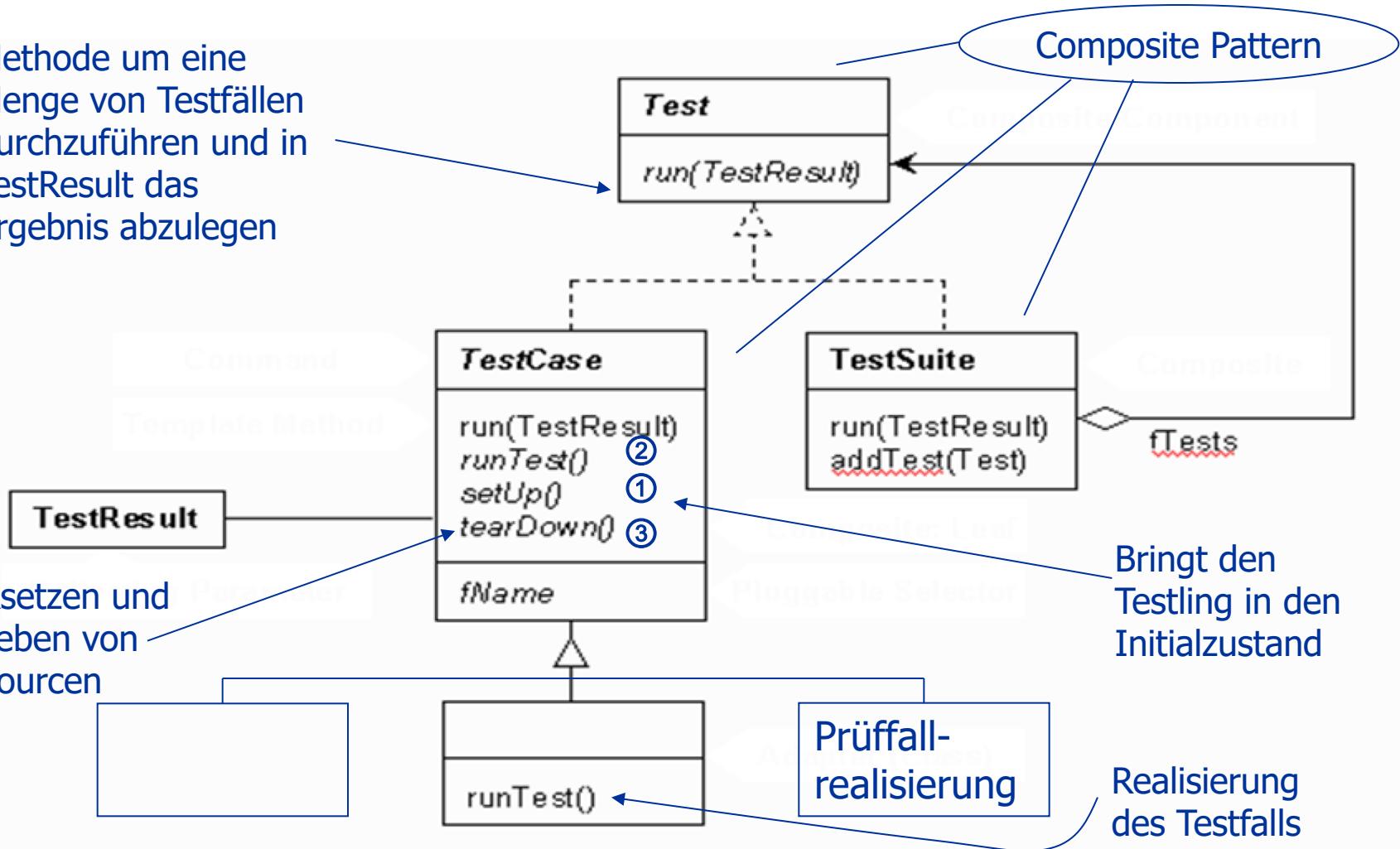
Testen des Programms mit Entwicklertests (1)

- Testen ist wichtige Qualitätssicherungsmaßnahme
- Unit Test
 - Testen von (kleinen) Einheiten in Isolation von anderen (seiteneffektfrei)
 - Unit Test in OO ist typischerweise ein Methodentest
- White Box Test
 - Implementierungsdetails des Prüflings sind bekannt
- Wichtig: wiederholbare, automatisierbare Testfallausführung
- JUnit
 - kleines Test Framework (<http://www.junit.org/>)
 - hat für Java schnell weite Verbreitung gefunden
 - Testfälle werden in Java kodiert (kein Sprachwechsel für Tests)
 - Framework kümmert sich um automatisierte Testfallausführung



Entwicklertests mit JUnit

Methode um eine Menge von Testfällen durchzuführen und in TestResult das Ergebnis abzulegen



JUnit4 Annotationen

- **@Test**
 - Kennzeichnung der Testfälle
 - Zusätzlich parametrisierbar, z.B.:
 - `@Test(timeout = 100)` schlägt fehl, wenn Ausführung länger als 100 Millisekunden dauert
 - `@Test(expected = IllegalArgumentException.class)` prüft, ob eine entsprechende Exception während der Ausführung ausgelöst wird
- Assertions
 - `assertTrue`, `assertFalse`, `assertEquals`, `assertNull`,
`assertNotNull`, `fail`, uvm. (siehe Api: `org.junit.Assert`)
- **@BeforeClass / @AfterClass**
 - Wird vor bzw. nach allen Tests einer Testklasse ausgeführt
- **@Before / @After**
 - Wird vor jedem Testfall ausgeführt
- **@Ignore**
 - Testfall vorübergehend bei der Ausführung ignorieren



Prinzip von Entwicklertests

- Identifizieren der zentralen Klassen/Module durch den Entwickler (nicht methodische Auswahl, sondern Bauchgefühl)
 - Festlegen von Testfällen oder Ad-Hoc-Tests (=> Glass-Box-Tests von Hand oder mit JUnit-Test-Tools)
- 
- i.d.R.
nicht
proto-
kolliert



©

Dr. Christian Bartelt

JUnit und Test-First

0. Wir überlegen uns erste Testfälle für die geforderte Funktionalität.

Wiederholung der Schritte 1-6:

1. Auswahl des nächsten Testfalls.
2. Wir entwerfen einen Test, der zunächst fehlschlagen sollte.
3. Wir schreiben gerade soviel Code, dass sich der Test übersetzen lässt (Signatur).
4. Wir prüfen, ob der Test fehlschlägt.
5. Wir schreiben gerade soviel Code, dass der Test erfüllt sein sollte.
6. Wir prüfen, ob der Test durchläuft.
7. Die Entwicklung ist abgeschlossen, wenn uns keine weiteren Tests mehr einfallen, die fehlschlagen können.



Anwendungsbeispiel Konto

- 0. Wir überlegen uns erste Testfälle
 - Erzeuge neues Konto (Account) für Kunden
 - Mache eine Einzahlung (deposit)
 - Mache eine Abhebung (withdraw)
 - Überweisung zwischen zwei Konten, ...
- 1. Auswahl des nächsten Testfalls: Erzeuge Konto



©

Dr. Christian Bartelt

Wir entwerfen einen kleinen Test

```
import static org.junit.Assert.*;  
import org.junit.Test;  
  
public class AccountTest {  
    @Test  
    public void testCreateAccount() {  
        Account account = new Account("Customer");  
        assertEquals("Customer", account.getCustomer());  
        assertEquals(0, account.getBalance());  
    }  
}
```

Testklassen enden mit „Test“

Testmethoden werden mit „@Test“ markiert

assertX-Methoden werden genutzt, um Ergebnisse zu prüfen

Testmethoden beginnen mit „test“:
Sie führen die getesteten Methoden aus und prüfen das Ergebnis



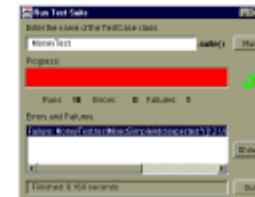
Wir schreiben gerade so viel Code, dass sich der Test übersetzen lässt...

```
public class Account {  
    public Account(String customer) {  
    }  
    public String getCustomer() {  
        return null;  
    }  
    public int getBalance() {  
        return 0;  
    }  
}
```

Die zu testende Klasse kennt die Testklasse nicht und kann daher auch ohne Tests eingesetzt werden.

Übersetzbarkeit bedeutet: Signaturen + Default-Return-Werte sind vorhanden

4. Wir prüfen, ob der Test fehlschlägt



Wir schreiben so viel Code, dass der Test erfüllt ist...

```
public class Account {  
    private String customer;  
    public Account(String customer) {  
        this.customer = customer;  
    }  
    public String getCustomer() {  
        return customer;  
    }  
    public int getBalance() {  
        return 0;  
    }  
}
```

Im Beispiel werden also der Konstruktor und `getCustomer` umgesetzt sowie die notwendige Datenstruktur (`String customer`) definiert.

6. Wir prüfen, ob der Test durchläuft



Wir schreiben weitere Testfälle

Mehrere Tests in einer Testklasse sind möglich

```
public class AccountTest {  
    ...  
    private Account account;  
    @Before  
    protected void setUp() throws Exception {  
        account = new Account("Customer");  
    }  
    @Test (expected = AmountNotCoveredException.class)  
    public void testWithdraw() throws Exception {  
        account.deposit(100);  
        account.withdraw(50);  
        assertEquals(50, account.getBalance());  
        account.withdraw(51);  
    }  
}
```

@Before setUp() wird vor jedem Test aufgerufen und erzeugt (gemeinsame) Ausgangsdaten
@After tearDown() wird nach jedem Test aufgerufen, um „aufzuräumen“

Auch erwartete Exceptions können getestet werden

Achtung: Dieser eine Test reicht nicht aus, um Randfälle abzudecken!



Wir organisieren die Testfälle in einer Test-Suit

```
public class AllTests {  
    public static Test suite() {  
        → TestSuite suite = new TestSuite();  
        suite.addTestSuite(AccountTest.class);  
        //weitere...  
        return suite;  
    }  
  
    public static void main(String[] args) {  
        //junit.textui.TestRunner.run(suite());  
        junit.swingui.TestRunner.run(AllTests.class);  
    }  
}
```

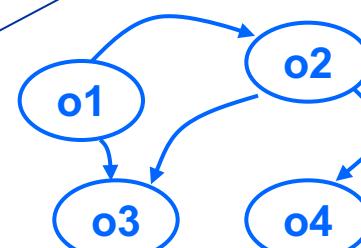
*Testklassen können in TestSuiten organisiert werden
TestSuite leitet von Test ab, daher hierarchische Struktur möglich*



Konzept von Entwicklertests

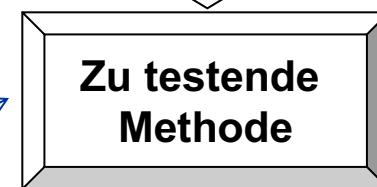
Fixture: Gruppe von Objekten, die einen Zustand beschreibt

setUp()



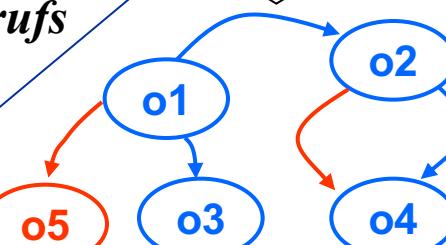
Methode operiert auf Fixture

runTest()



Fixture: Ergebnis des Aufrufs

runTest()
tearDown()



In JUnit Schnittstellen zur Umgebung sind mit „Dummies“ zu simulieren, um Nebeneffekte zu erfassen

DB-Anschluss

Protokoll

GUI



Inhalt

- Motivation und Herausforderungen
- Überblick und Konzepte
- Bestandteile einer Softwareproduktionsumgebung (SPU)
- Vom Design (OOD) zum Programm (OOP)
 - Von Aktivitätsdiagrammen Code Generieren
 - Defensives Programmieren
 - Entwicklertests
 - Refactoring und Code-Qualität



©

Dr. Christian Bartelt

Exkursion: Semantik von Aktivitätsdiagrammen

- Ein Aktivitätsdiagramm
 - stellt analog zu einer Straßenkarte den Rahmen und die Regeln von Verhaltensabläufen auf detailliertem Niveau dar
 - umfasst Start- und Endpunkte (Auf- und Abfahrten), Verzweigungen (Kreuzungen, Kreisverkehre) bestimmte Bedingungen (Einbahnstraße, Gewichtsbeschränkungen) und vieles mehr
 - stellt nicht die eigentlichen Abläufe (Straßenverkehr, der jeden Tag variiert) dar, sondern die Regeln, die alle möglichen Abläufe beschreiben
- Jetzt für die Semantik:
 - Mit Hilfe eines Aktivitätsdiagramms können primitive Operationen, Kontrollstrukturen sowie Exceptions dargestellt werden.



Darstellung von Klassen

- Wie im Programmierkurs gezeigt wurde, kann eine Java-Klasse mit Hilfe eines Klassendiagramms dargestellt werden.:

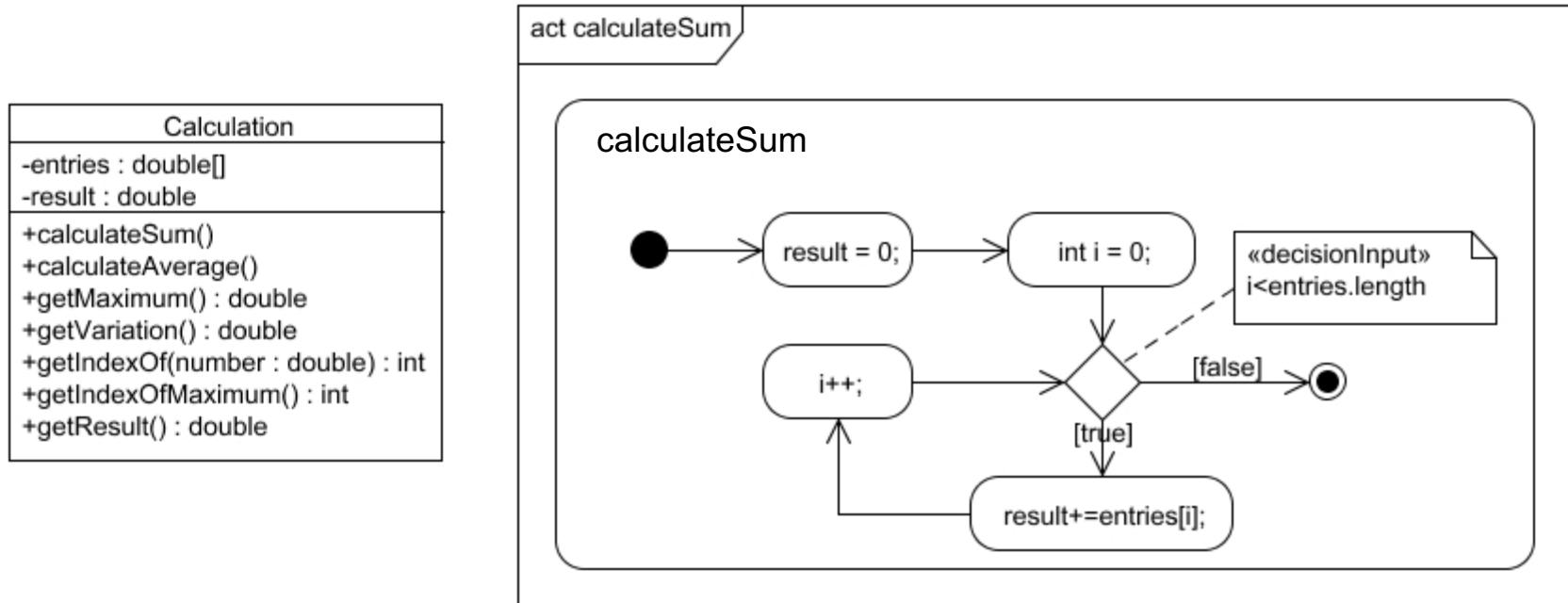
```
1 public class Calculation {  
2     private double[] entries;  
3     private double result;  
4  
5     public void calculateSum() {  
6         result = 0;  
7         for (int i = 0; i < entries.length; i++) {  
8             result+=entries[i];  
9         }  
10    }  
11  
12    public void calculateAverage() {  
13        calculateSum();  
14        result/=entries.length;  
15    }  
16    // ...
```

Calculation
-entries : double[]
-result : double
+calculateSum()
+calculateAverage()
+getMaximum() : double
+getVariation() : double
+getIndexOf(number : double) : int
+getIndexOfMaximum() : int
+getResult() : double



Darstellung von Operationen

- Es ist möglich die interne Funktionsweise der Operationen mit Hilfe von Aktivitätsdiagrammen zu modellieren.
- Die Aktivität trägt die Bezeichnung der Operation.



Aufruf anderer Operationen (1)

- Innerhalb einer Methode können andere Methoden aufgerufen werden.
- Im folgenden Beispiel wird die Methode `calculateSum()` der Klasse `Calculation` von der Methode `calculateAverage()` aufgerufen:

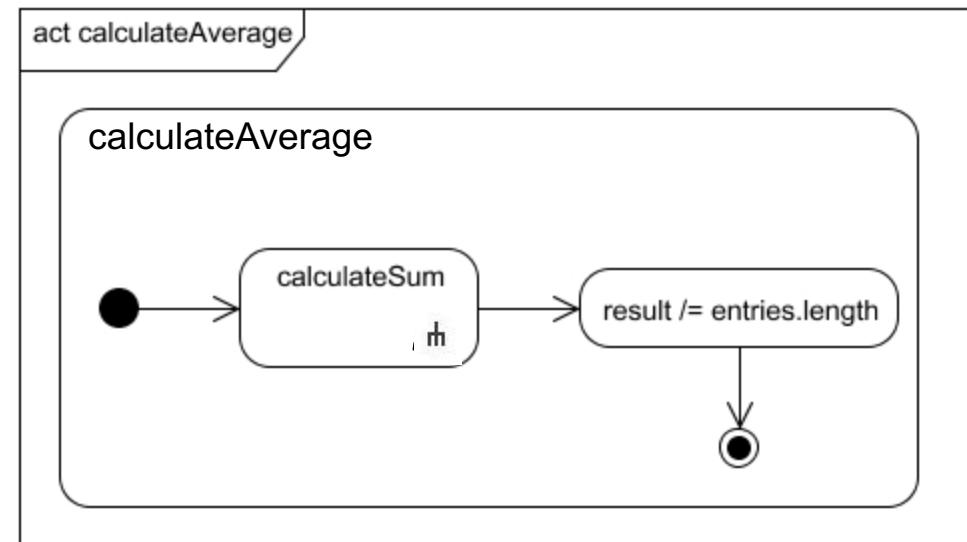
```
1 public class Calculation {  
2     private double[] entries;  
3     private double result;  
4  
5     public void calculateSum() {  
6         result = 0;  
7         for (int i = 0; i < entries.length; i++) {  
8             result+=entries[i];  
9         }  
10    }  
11  
12    public void calculateAverage() {  
13        calculateSum();  
14        result/=entries.length;  
15    }  
16    // ...
```



Aufruf anderer Operationen (2)

- Dieser Sachverhalt wird bei Aktivitätsdiagrammen durch spezielle Aktionen dargestellt, die Aktivitäten genannt werden.

Calculation
-entries : double[]
-result : double
+calculateSum()
+calculateAverage()
+getMaximum() : double
+getVariation() : double
+getIndexOf(number : double) : int
+getIndexOfMaximum() : int
+getResult() : double



Darstellung eines Rückgabewertes (1)

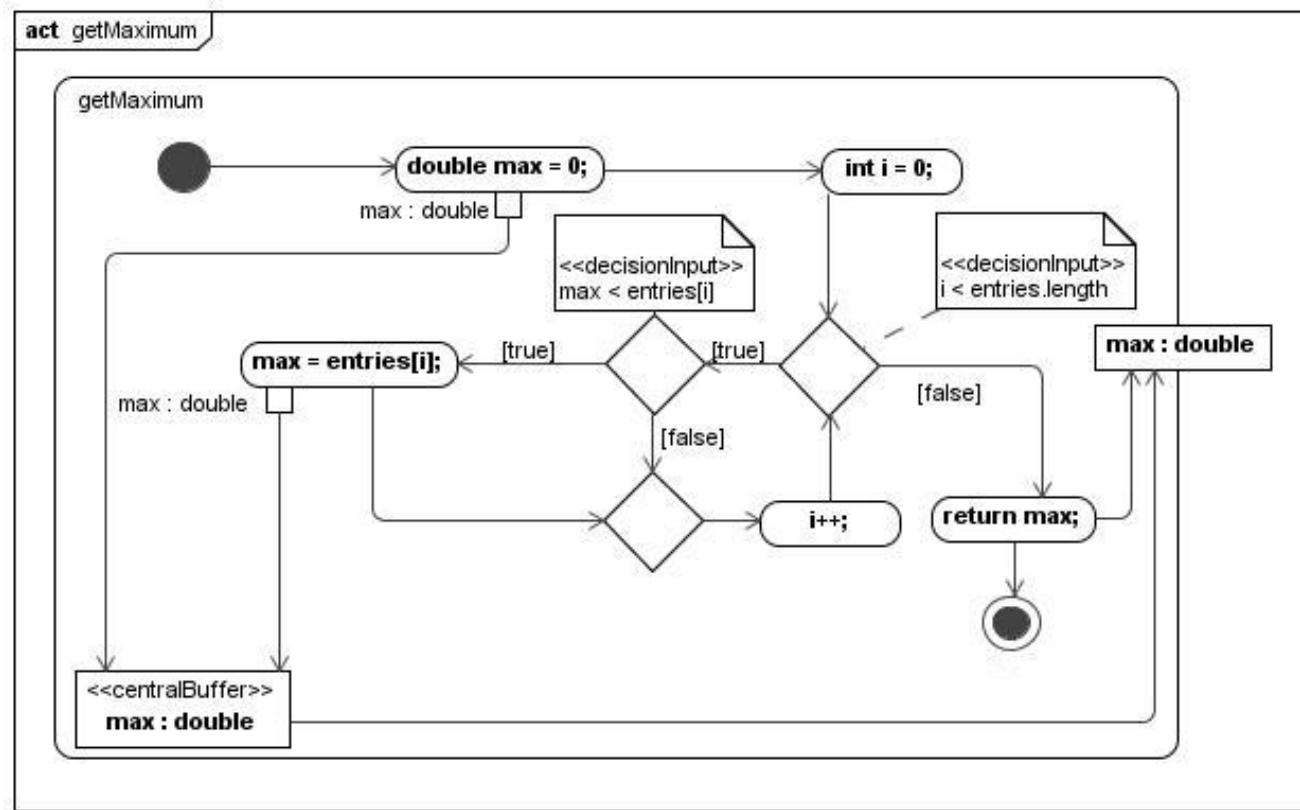
- In Java besteht die Möglichkeit einer Methode einen Rückgabewert zu geben.
- Im folgenden Beispiel gibt die Methode `getMaximum()` das Maximum eines Arrays zurück.

```
1 public class Calculation {  
2     private double[] entries;  
3     private double result;  
4  
5     // ...  
6     public double getMaximum() {  
7         double max = 0;  
8         for (int i = 0; i < entries.length; i++) {  
9             if(max<entries[i]) max = entries[i];  
10        }  
11        return max;  
12    }  
13    // ...
```



Darstellung eines Rückgabewertes (2)

- In UML kann dieser Sachverhalt mit sogenannten Ausgabepins dargestellt werden:



Verwendung eines Rückgabewertes (1)

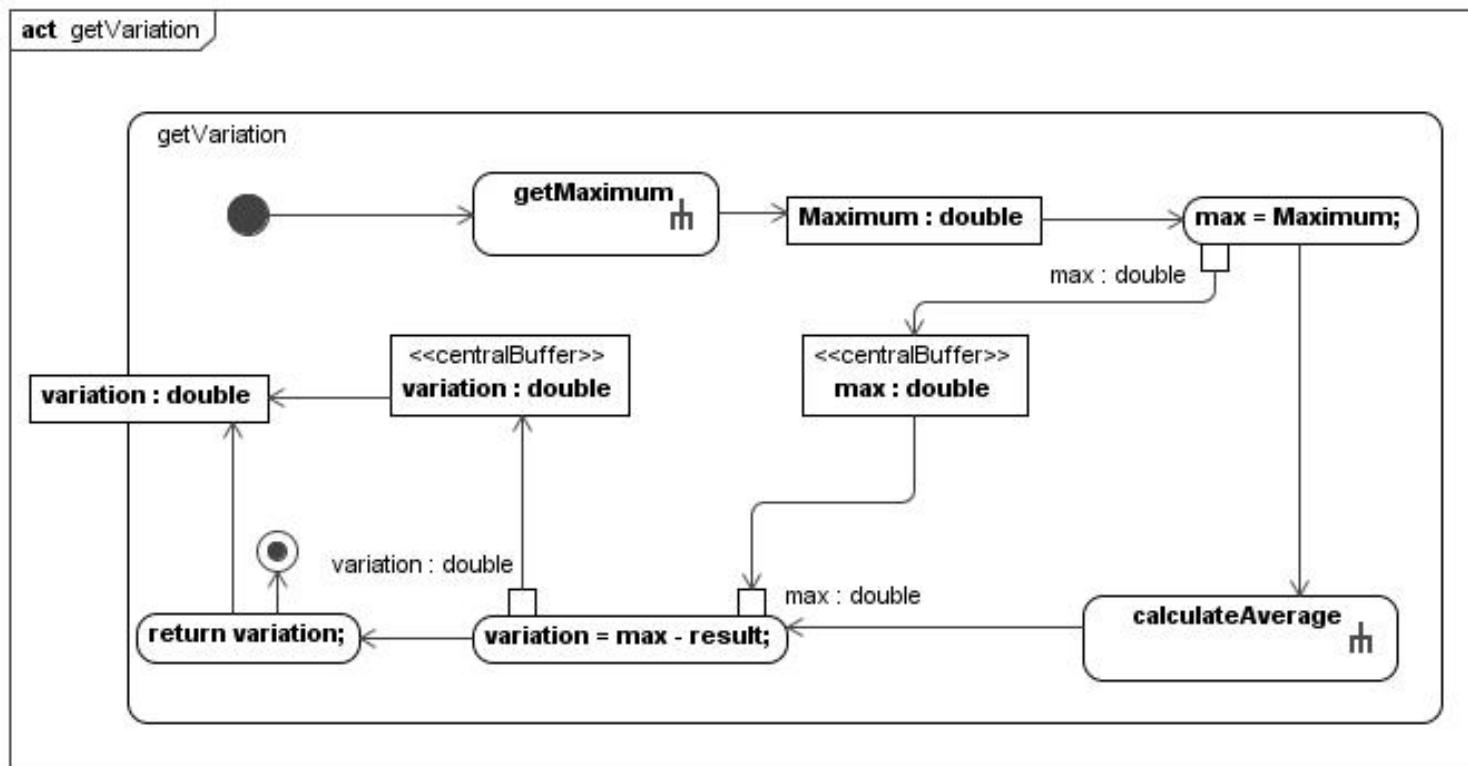
- Die Methode `getVariation()` der Klasse `Calculation` ruft die Methoden `calculateAverage()` und `getMaximum()` auf.
- Der Rückgabewert der Methode `getMaximum()` wird ebenfalls verwendet.

```
1 public class Calculation {  
2     private double[] entries;  
3     private double result;  
4  
5     // ...  
6     public double getVariation() {  
7         double variation;  
8         double max;  
9         max = getMaximum();  
10        calculateAverage();  
11        variation = max - result;  
12        return variation;  
13    }  
14    // ...
```



Verwendung eines Rückgabewertes (2)

- Die Operation `getVariation()` wird als Aktivitätsdiagramm folgendermaßen modelliert:



Darstellung Parameterübergabe (1)

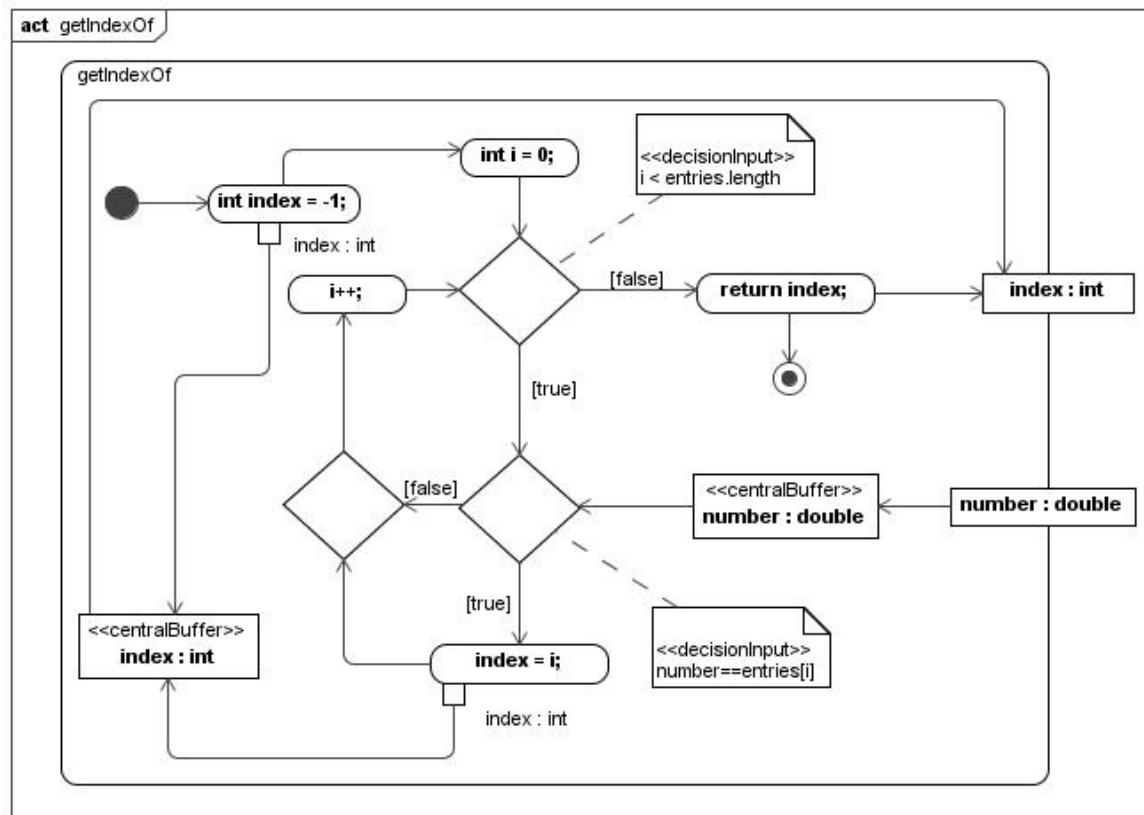
- In Java besteht die Möglichkeit, den Methoden einer Klasse Parameter zu übergeben.
- Im folgenden Beispiel wird der Methode `getIndexOf()` der Parameter `number` übergeben:

```
1 public class Calculation {  
2     private double[] entries;  
3     private double result;  
4  
5     // ...  
6     public int getIndexOf(double number) {  
7         int index = -1;  
8         for (int i = 0; i < entries.length; i++) {  
9             if (number == entries[i]) index = i;  
10        }  
11        return index;  
12    }  
13    // ...
```



Darstellung Parameterübergabe (2)

- Die Parameter der Methoden können in UML ähnlich wie die Rückgabewerte als sogenannte Eingabepins dargestellt werden.



Übergabe eines Parameters (1)

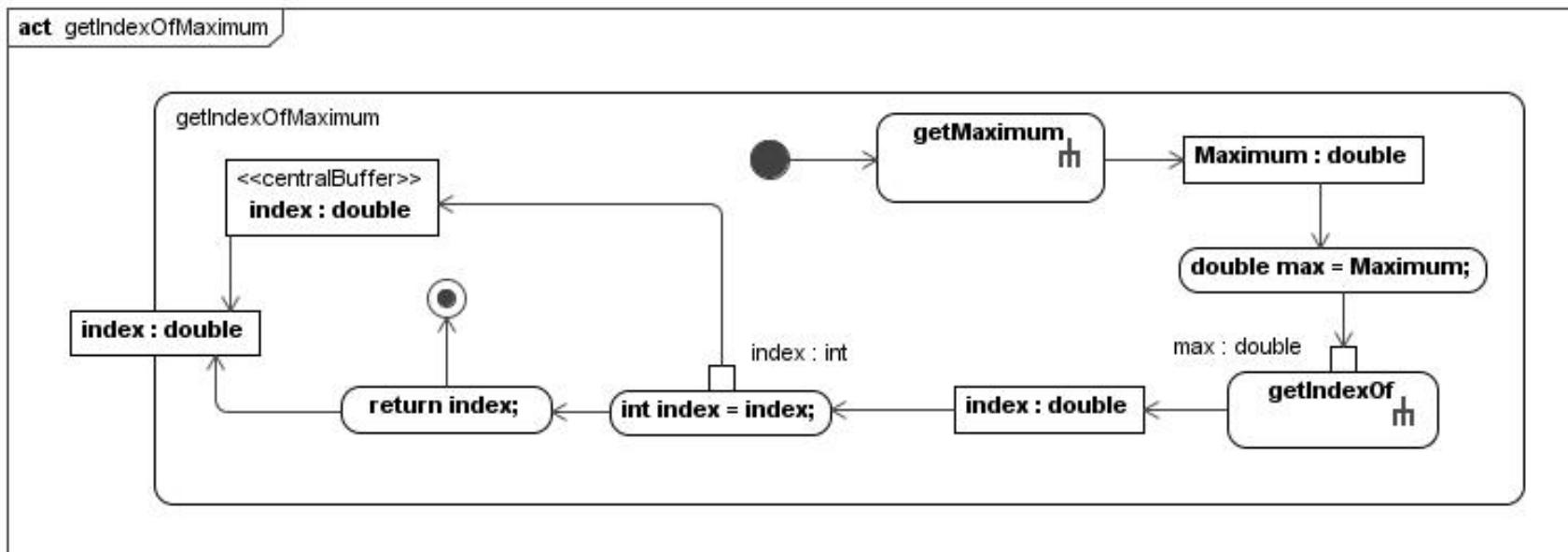
- Die unten abgebildete Methode `getIndexOfMaximum()` übergibt der Methode `getIndexOf()` einen Parameter.
- Der Rückgabewert der Methode `getIndexOf()` ist ebenfalls der Rückgabewert der Methode selbst.

```
1 public class Calculation {  
2     private double[] entries;  
3     private double result;  
4  
5     // ...  
6     public int getIndexOfMaximum() {  
7         double max = getMaximum();  
8         int index;  
9         index = getIndexOf(max);  
10        return index;  
11    }  
12    // ...
```



Übergabe eines Parameters (2)

- Sowohl die Parameter als auch die Rückgabewerte werden als Pins oder Objektknoten dargestellt.



Methodenaufruf anderer Klassen (1)

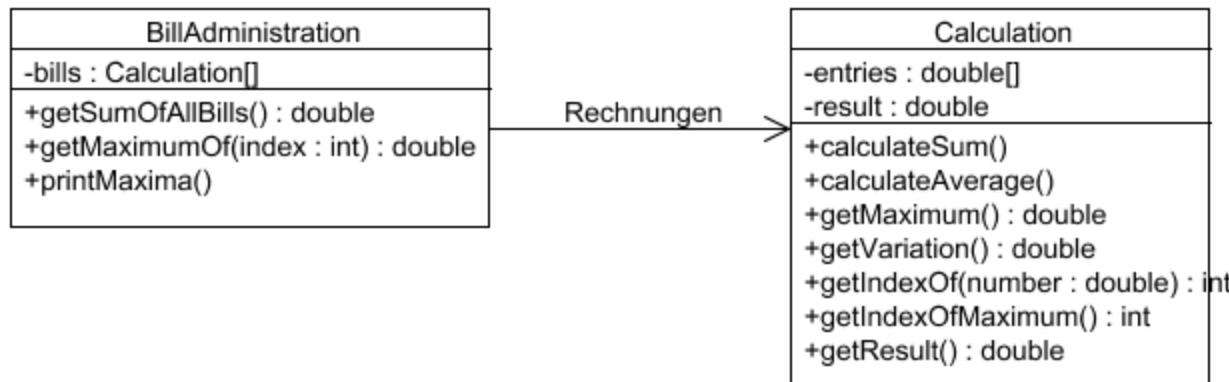
- Die Methoden einer Klasse können Methoden anderer Klassen aufrufen.
- Im folgenden Beispiel verwendet die Klasse BillAdministration die Klasse Calculation.

```
1 public class BillAdministration {  
2     private Calculation[] bills;  
3  
4     public double getSumOfAllBills() {  
5         double result = 0;  
6         for (int i = 0; i < bills.length; i++) {  
7             bills[i].calculateSum();  
8             result += bills[i].getResult();  
9         }  
10    }  
11    }  
12    //...
```



Methodenaufruf anderer Klassen (2)

- Die Klassen stehen damit untereinander in Beziehung.
- Die Klasse BillAdministration ist assoziiert mit der Klasse Calculation.



Methodenaufruf anderer Klassen (3)

- Als Aktivitätsdiagramm kann die Methode `getSumOfAllBills()` folgendermaßen dargestellt werden:

