

# Softwarequalität

SS 2008

Gabriele Taentzer

Philipps-Universität Marburg

---

# Organisation der LV

- Umfang: 4 SWS, 6 ECTS Punkte
- Veranstalter: Gabriele Taentzer, Stefan Jurack
- Kontakt:
  - *taentzer@mathematik.uni-marburg.de,*  
*Raum D5430, Tel: 21532*
  - *sjurack@mathematik.uni-marburg.de,*  
*Raum: D5432, Tel: 21511*
- Termine:
  - *VL: Mi 11.15 – 13 h, Hörsaal I*
  - *UE: Do 16.15 – 18 h, Seminarraum V*

# Organisation der LV

- Hauptstudium, ab 5. Semester
- Voraussetzung:
  - *Grundvorlesungen in Praktischer Informatik*
- Scheinkriterien:
  - *Übungsaufgaben*
  - *1 Kurzvortrag (ca. 20 min)*
  - *Abschlussklausur / Kolloquium*
- Homepage der LV:
  - *[www.uni-marburg.de/fb12/swt](http://www.uni-marburg.de/fb12/swt)*  
*Lehre → SS08 → Software Qualität*

# Lehrveranstaltungsstil

- Konzeptvermittlung durch Folien
- Folienkopien sind auf der Homepage verfügbar, kleinere Abweichungen (insb. Korrekturen) sind möglich
- Beispiele häufig an der Tafel, häufig mit englischen Bezeichnern
- **Zwischenfragen und Kommentare während der Vorlesung sind grundsätzlich erwünscht.**
- Literatur häufig in Englisch

# Lernziele

- Verständnis von Softwarequalität und -qualitätsmanagement
- Verständnis für die Möglichkeiten und Grenzen einzelner Methoden und Techniken zur Messung und zur Verbesserung von Softwarequalität
- Einarbeitung in spezifische Aspekte der Softwarequalität und Vorstellung dieser
- Grundverständnis für die in der LV vorgestellten Werkzeuge, durch Übungen vertieft

# Inhalt

- Einführung in das Thema Softwarequalität
- Softwarequalitätsmanagement
- Syntaktische Verfahren
  - *Softwaremetriken*
  - *Entwicklungsrichtlinien*
  - *Bad Code Smell und Refactoring*
  - *Design Patterns und Model Driven Architecture*
- Semantische Verfahren
  - *Testverfahren und Profiling*
  - *Validations- und Verifikationstechniken*
- Zusammenfassung und Ausblick

# Softwarequalität: Einführung

9. April 2008

---

# Überblick

- Warum ist Softwarequalität wichtig?
- Was ist Softwarequalität?
- Wie erreicht man Softwarequalität?

# Berühmte Software-Fehler

- Im Juni 1996 explodierte die Rakete Ariane 5 wenige Sekunden nach dem Start. Ursache war ein Softwarefehler im Trägheitsnavigationssystem der Rakete.
- Im September 1999 ging die Raumsonde Mars Orbiter der NASA kurz vor dem Ziel verloren, weil ein Fehler bei der Umrechnung von Maßeinheiten in der Support-Software der Kontrollstation aufgetreten war.

# Was ist Softwarequalität?

Ergonomie

Erlernbarkeit

Korrektheit

Zuverlässigkeit

Sicherheit

Installierbarkeit

Performance

anforderungsgerecht

Wiederverwendbarkeit

Testbarkeit

# Probleme eines SW-Projekts

- Nach Untersuchungen von Standish Group, Gartner Group, Cutter Consortium und Center for Project Management:
  - *23 % aller Softwareprojekte erfolgreich,*
  - *ca. 53 % über Budget und/oder über Zeit und*
  - *ca. 24 % abgebrochen*
- in besonderem Maße geprägt von Fehleinschätzungen: Zeit für Organisation, Kommunikation, Programmierung
- ein Programmierer produziert im längerfristigen Durchschnitt **10 LOC** pro Arbeitstag [Mayr]

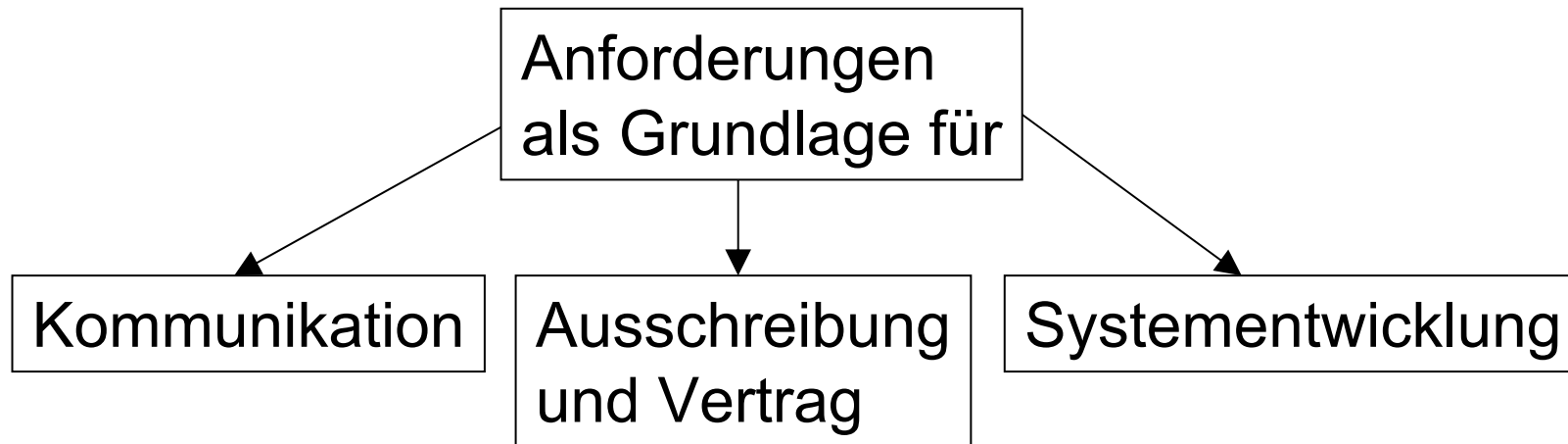
# Was ist Softwarequalität?

- Was ist das Werteverständnis der beteiligten Personen?
- Qualitätsanforderungen können sich gegenseitig widersprechen.
  - *abhängig von der Art der Software*
  - *abhängig von den Kundenwerten*
  - *abhängig von den Qualitätsvorstellungen der Entwickler*

# Anforderungsgerechte Software

- Welche Probleme gibt es bei der Anforderungsbeschreibung?
- Wie sieht eine gute Anforderungsbeschreibung aus?
- Wie sollte man mit Anforderungsänderungen umgehen?

# Bedeutung der Anforderungsbeschreibung



- Die Kosten für eine Anforderungsänderung sind um so höher, je später sie stattfindet.

# Mögliche Ausgangsszenarien

- Der Kunde überläßt die Anforderungsbeschreibung dem Entwickler.
  - *unverständlich für den Kunden, ungewünschtes System*
- Der Kunde hat eine eigene Anforderungsbeschreibung, der Entwickler übernimmt diese.
  - *unverständlich für den Entwickler, System wird nicht fertig.*
- Der Kunde kennt seine Anforderungen, der Entwickler erstellt die entsprechende Anforderungsbeschreibung.
  - *Beide Seiten haben ein gemeinsames Verständnis vom System.*

# Typische Probleme bei der Anforderungsbeschreibung

- unklare Zielvorstellungen für das System
- hohe Komplexität der zu lösenden Aufgabe
- Kommunikationsprobleme, Sprachbarrieren
- sich ständig ändernde Ziele und Anforderungen
- schlechte Qualität der Anforderungen
  - *mehrdeutig, redundant, widersprüchlich, ungenau...*
- unnötige Produktmerkmale
- ungenaue Projektplanung

# Qualitätskriterien für Anforderungen

- vollständig
- korrekt
- rechtlich klar
- konsistent
- testbar
- aktuell
- verständlich
- realisierbar
- notwendig
- bewertbar
- eindeutig

Bsp.: Das System soll schnell reagieren.

Worauf? Wie schnell?

Wie reagieren?

# Inhalt der Anforderungsbeschreibung

- Zielsetzung
- allgemeine Beschreibung
- Definitionen und Abkürzungen
- Produktumfeld
- funktionale Anforderungen
- nichtfunktionale Anforderungen
- Abnahmekriterien
- Glossar, Index, Referenzen

IEEE 830-98  
Standard für  
Anforderungs-  
dokumente

# Nichtfunktionale Anforderungen

- Nichtfunktionale Anforderungen spielen bezüglich der Softwarequalität eine spezielle Rolle. Viele Qualitätsvorstellungen der Kunden werden durch nichtfunktionale Anforderungen formuliert.
- Beispiel: Anforderungen an die Dienstqualität
  - *Bsp.: Das System muss jede Benutzeranfrage innerhalb von 30 Sekunden ausführen.*
- Beispiel: ergonomische Anforderungen
  - *Bsp.: Das System muss die gespeicherten Objekte formatiert ausgeben können.*

# Verfahren zur Qualitätsmessung

- Definition von Qualitätsmanagementprozessen
- quantitative Messungen:
  - *Softwaremetriken*
- Überprüfung syntaktischer Muster:
  - *Entwicklungsrichtlinien*
  - *Entwurfsmuster und Softwarearchitekturen*
- Beispiele und Gegenbeispiele:
  - *Testverfahren und Profiling*
- Überprüfung semantischer Eigenschaften:
  - *Validation, Verifikation*

# Kurzvorträge

- Eclipse-Plugin Metrics
- Eclipse User Interface Guidelines
- Eclipse-Plugin PMD
- Eclipse-Refactoring-Tool
- Eclipse-Plugin: Eclipse Test and Performance Tools Platform Project
- Eclipse-Plugin GUIDancer

# Zusammenfassung

- Software ist häufig fehlerhaft, manchmal mit fatalen Folgen.
- Der Begriff „Softwarequalität“ umfasst viele verschiedene Aspekte. Wir unterscheiden syntaktische und semantische.
- Erster und wesentlicher Schritt zu besserer Software: anforderungsgerechte Software
- Es gibt viele Ansätze, die Softwarequalität zu verbessern. Wir betrachten Standardtechniken, um Softwarequalität zu messen und zu verbessern.
- Nächste Woche: Softwarequalitätsmanagement

# Softwarequalitätsmanagement

16. April 2008



# Überblick

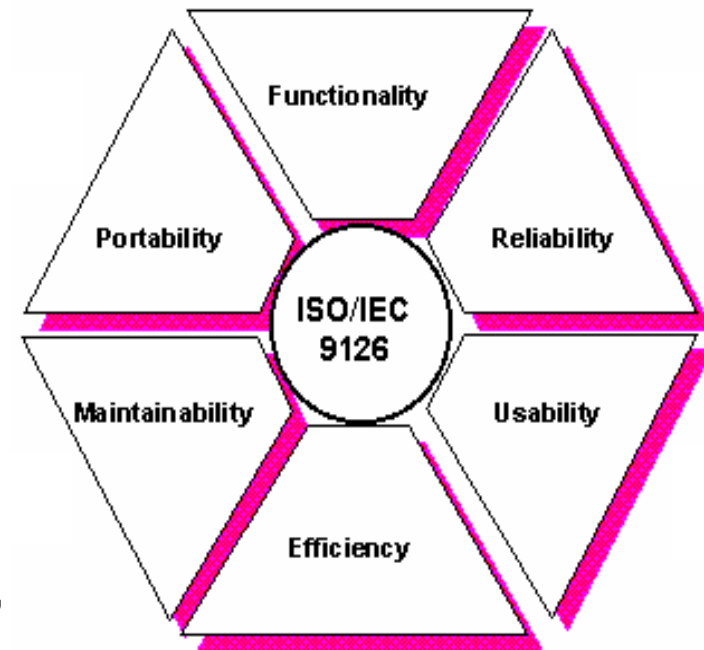
- Welche Qualitätsmodelle gibt es?
- Welche Qualitätsanforderungen leiten sich daraus ab?
- Auf welche Weise kann Qualitätsmanagement durchgeführt werden?
- Welche Standards für Qualitätsmanagement gibt es?

# Was ist Qualität?

- **transzendent**: Q. ist universell erkennbar, absolut, einzigartig und vollkommen. Sie wird nicht gemessen.
- **produktbezogen**: Q. ist eine meßbare, genau spezifizierte Größe, die das Produkt beschreibt.
- **benutzerbezogen**: Q. wird durch den Benutzer festgelegt. Verschiedene Benutzer haben unterschiedliche Wünsche und Bedürfnisse.
- **prozeßbezogen**: Q. entsteht durch die richtige Erstellung des Produkts.
- **Kosten/Nutzen-bezogen**: Q. ist eine Funktion von Kosten und Nutzen.

# Qualitätsmerkmale für Software

- **Funktionalität:** Korrektheit, Angemessenheit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit
- **Zuverlässigkeit:** Reife, Fehlertoleranz, Wiederherstellbarkeit
- **Benutzbarkeit:** Verständlichkeit, Bedienbarkeit, Erlernbarkeit, Robustheit
- **Effizienz:** Wirtschaftlichkeit, Zeitverhalten, Verbrauchsverhalten
- **Wartungsfreundlichkeit:** Analysierbarkeit, Änderbarkeit, Stabilität, Testbarkeit
- **Übertragbarkeit:** Anpassbarkeit, Installierbarkeit, Konformität, Austauschbarkeit



# Qualitätsmodelle

- Factor-Criteria-Metrics-Modell (FCM):
  - *Factor: Qualitätsmerkmal (benutzerorientierte Sicht)*
  - *Criteria: softwareorientierte Charakteristika*
  - *Metrics: Qualitätsmaße*
- Goal-Question-Metric-Ansatz (GQM):
  - *Goal: definiere die Auswertungsziele*
  - *Question: leite Fragen zur Quantifizierung ab*
  - *Metrics: leite Maße zur Beantwortung der Fragen ab*
  - *entwerfe einen Mechanismus zur Meßwerterfassung*
  - *validiere die Meßwerte*
  - *interpretiere die Meßwerte*

# Beispiel: GQM-Datenblatt

- Produkt: funktionale Spezifikation
- Qualitätsmerkmal: Wartbarkeit
- Ziel: Die funktionale Spezifikation hält vereinbarte Standards vollständig ein.
- Frage: Existiert für jeden Prozeß entweder eine Minispezifikation oder eine Verfeinerung in einem Flussdiagramm?
- Maß: Anzahl der Prozesse mit Minispec/Diagramm
- Messverfahren: hierarchische Übersicht aller Prozesse (Elementar- und andere Prozesse)

# Qualitätsmanagement

- **produktorientiert:**  
Softwareprodukte und Zwischenergebnisse werden auf vorher festgelegte Qualitätsmerkmale überprüft.
- **prozeßorientiert:**  
Methoden, Werkzeuge, Richtlinien und Standards für den Erstellungsprozeß der Software

# Qualitätssicherungsmaßnahmen

- konstruktive Maßnahmen:  
Methoden, Sprachen, Werkzeuge, Richtlinien, Standards und Checklisten, die eine bestimmte Produkt-oder Prozeßqualität garantieren
- Beispiele:
  - *Gliederungsschema für das Pflichtenheft*
  - *Verwendung einer typisierten Programmiersprache*
  - *Importierte Daten werden auf Richtigkeit überprüft.*
  - *OO-Softwareentwicklung unterstützt die Wiederverwendbarkeit von Software.*

# Qualitätssicherungsmaßnahmen

- **analytische Maßnahmen:**  
Das existierende Qualitätsniveau wird gemessen. Ausmaß und Ort des Defekts können identifiziert werden.
- **Analysierende Verfahren** sammeln Informationen ohne Ausführung der Software mit konkreten Eingaben.
- **Testende Verfahren** führen die Software mit konkreten Eingaben aus.
- *Eine vorausschauende, konstruktive Qualitätslenkung erspart viele analytische Maßnahmen.*

# Beispiele: Konstruktive und analytische Maßnahmen

- konstruktive Maßnahme: geeignete Modularisierung der Software
- analytische Maßnahme: Modultests, die Codeüberdeckung garantieren
  
- konstruktive Maßnahme: exakte Beschreibung der funktionalen Anforderungen
- analytische Maßnahme: Erstellung von anforderungsgerechten Testfällen

# Prinzipien der SW-Qualitätssicherung

- Prinzip der produkt- und prozeßabhängigen Qualitätszielbestimmung
- Prinzip der quantitativen Qualitätssicherung
- Prinzip der maximal konstruktiven Q.sicherung
- Prinzip der frühzeitigen Fehlerentdeckung und –behebung
- Prinzip der entwicklungsbegleitenden, integrierten Qualitätssicherung
- Prinzip der unabhängigen Qualitätssicherung

# Prinzip der produkt- und prozeßabhängigen Qualitätssicherung

- Jedes SW-Produkt soll nach Fertigstellung eine bestimmte Qualität besitzen.
- Über 50 % der Betriebe verzichten auf eine explizite Definition von Qualitätsmerkmalen.
- 1. Priorität: Robustheit, Verständlichkeit, Wartbarkeit und Laufzeiteffizienz
- 2. Priorität: Korrektheit, Vollständigkeit, Benutzungsfreundlichkeit
- Festgelegte Qualitätsziele bringen notwendige Planungs- und Kalkulationssicherheit.

# Prinzip der quantitativen Q.sicherung

- *„Ingenieursmäßige Qualitätssicherung ist undenkbar ohne die Quantifizierung von Soll- und Istwerten.“*  
(Rombach)
- Metriken sind ziel- und kontextabhängig.
- Kreativer Charakter vieler Aspekte der Softwareentwicklung
  - *viel höhere Anzahl der Variationsparameter als traditionell*
  - *unkontrollierte Variabilität der Entwicklungsprozesse*
- Praktisch einsetzbare Techniken, Methoden und Werkzeuge existieren und sind ausbaufähig.

# Prinzip der maximal konstruktiven Qualitätssicherung

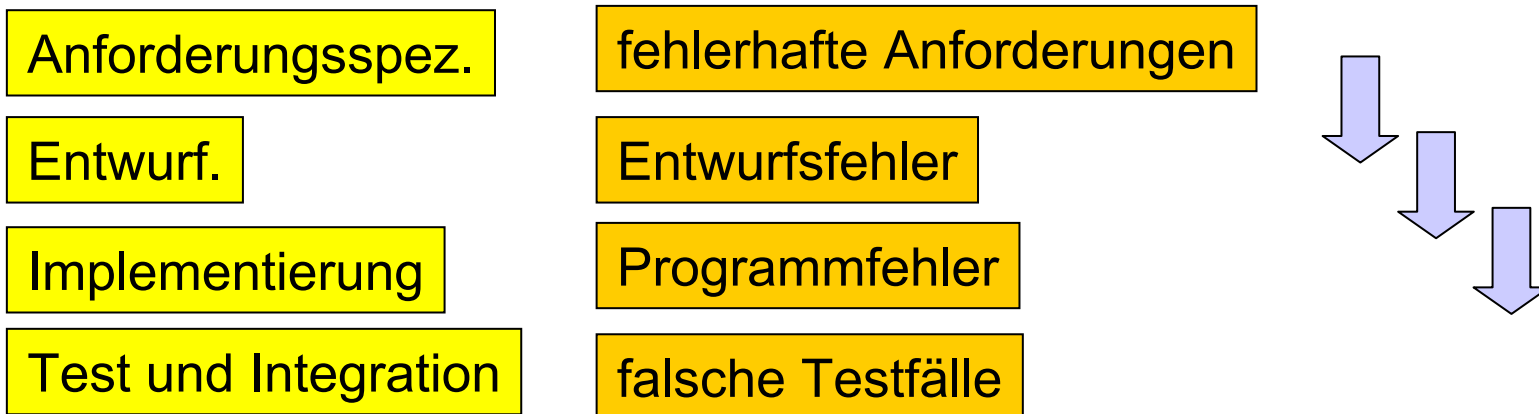
- *„Fehler, die nicht gemacht werden können, brauchen nicht behoben werden.“*
- Vorausblickende konstruktive Maßnahmen reduzieren analytische Maßnahmen.
- direkte Verbesserung der Produktqualität
- Ermöglicht analytische Maßnahmen

Beispiel:

- *explizite Typangabe für die Elemente einer Collection*
- *vermeidet Elemente mit falschem Typ in einer Collection*

# Prinzip der frühzeitigen Fehlerentdeckung und -behebung

- Ein Fehler ist
  - *jede Abweichung von den funktionalen UND nichtfunktionalen Anforderungen der Auftraggeber*
  - *jede Inkonsistenz in den Anforderungen*
- Je früher ein Fehler entdeckt wird, desto kostengünstiger kann er behoben werden.



# Prinzip der entwicklungsbegleitenden, integrierten Qualitätssicherung

- Konventionell setzt Qualitätssicherung erst am Ende des Entwicklungsprozesses ein.
- Qualitätssicherung findet immer dann statt, wenn sie angebracht ist.
- Qualitätssicherung wird nicht als Fremdkörper empfunden.
- Ein Teilprodukt steht der nächsten Iteration erst dann zur Verfügung, wenn eine bestimmte Qualität erfüllt ist.
- Das Qualitätsniveau ist zu jeder Zeit sichtbar.
- Realistische Beurteilung des Projektfortschritts möglich.

# Prinzip der unabhängigen Qualitätssicherung

- „...testing is a **destructive** process, even sadistic process...“ (Myers)
- Organisation der Qualitätssicherung:
  - *unabhängig von der Entwicklung*
  - *Teil der Entwicklung*
- Personalausstattung der Qualitätssicherung
  - *Personal nur für die Qualitätssicherung*
  - *jeder Mitarbeiter rotiert zw. Entwicklung u. QS*
  - *jede Mitarbeiterin arbeitet sowohl in der Entwicklung als auch an der QS anderer Entwicklungen*
- Vor- und Nachteile?

# Verbesserung der Prozeßqualität am Beispiel des ISO 9000-Ansatzes

- Software-Unternehmen, die Qualitätsmanagement nach ISO 9000 durchführen, können sich zertifizieren lassen.
- von ISO 9000-3 geforderte Dokumente:
  - *Vertrag Auftraggeber – Lieferant*
  - *Spezifikation*
  - *Entwicklungsplan*
  - *Qualitätssicherungsplan*
  - *Testplan*
  - *Wartungsplan*
  - *Konfigurationsmanagementplan*

# Vor- und Nachteile von ISO 9000

- stellt Anforderungen an den Entwicklungsprozess
- definiert wichtige Dokumente und ihre Inhalte
- fordert die organisatorische Unabhängigkeit der QS
- verpflichtet die Geschäftsführung zur QS
- keine saubere Trennung zwischen fachlichen, Management- und QS-Aufgaben, auch innerhalb der Dokumente
- Gefahr der „Software-Bürokratie“
- CASE-Werkzeuge zur praktischen Durchführbarkeit nötig

# Zusammenfassung

- Softwarequalität hat viele verschiedene Aspekte.
- Es werden konstruktive und analytische Verfahren zur Qualitätssicherung unterschieden.
- Prinzipien der Qualitätssicherung:
  - *explizite Definition von Qualitätsmerkmalen für SW-Produkte*
  - *Qualität soll meßbar sein.*
  - *Vorausblickende konstruktive Maßnahmen reduzieren analytische Maßnahmen.*
  - *frühzeitige Fehlerbehebung*
  - *integrierte Qualitätssicherung*
  - *unabhängige Qualitätssicherung*
- Definition der QS durch ISO 9000 Standard

# Literatur

- Balzert, Lehrbuch der Softwaretechnik 2, Spektrum Verlag
- DIN ISO 9126, Informationstechnik – Beurteilung von Softwareprodukten, Qualitätsmerkmalen und Leitfaden zu deren Verwendung
- ISO 9000, Normen zum Qualitätsmanagement und zur Qualitätssicherung
- ISO 9001, Qualitätsmanagementsysteme – Modell zur Qualitätssicherung

# Softwaremetriken

23. April 2008



# Was sind Softwaremetriken?

„Softwaremetriken messen Qualität.“

besser:

„Softwaremetriken definieren, wie Kenngrößen der Software oder des Softwareentwicklungsprozesses gemessen werden.“

## Wichtige Fragen:

- Was kann das Messen bringen?
- Wer möchte messen?
- Was kann man messen?
- Wie muß man messen?
- Welche Verfahren gibt es?

# Vor- und Nachteile von SW-Metriken

## Vorteile:

- Softwareentwicklung wird vorhersagbarer
- auf mögliche Schwachstellen wird hingewiesen
- Test- und Wartungsaufwand beurteilen
- erzieherischer Effekt auf die Entwickler

## Nachteile:

- Messen ohne Ziel
- Nutzen von Metriken oft nicht klar
- Abwehrhaltung der Entwickler
- subjektiver Einfluss der Prüfer möglich

# Wer möchte messen?

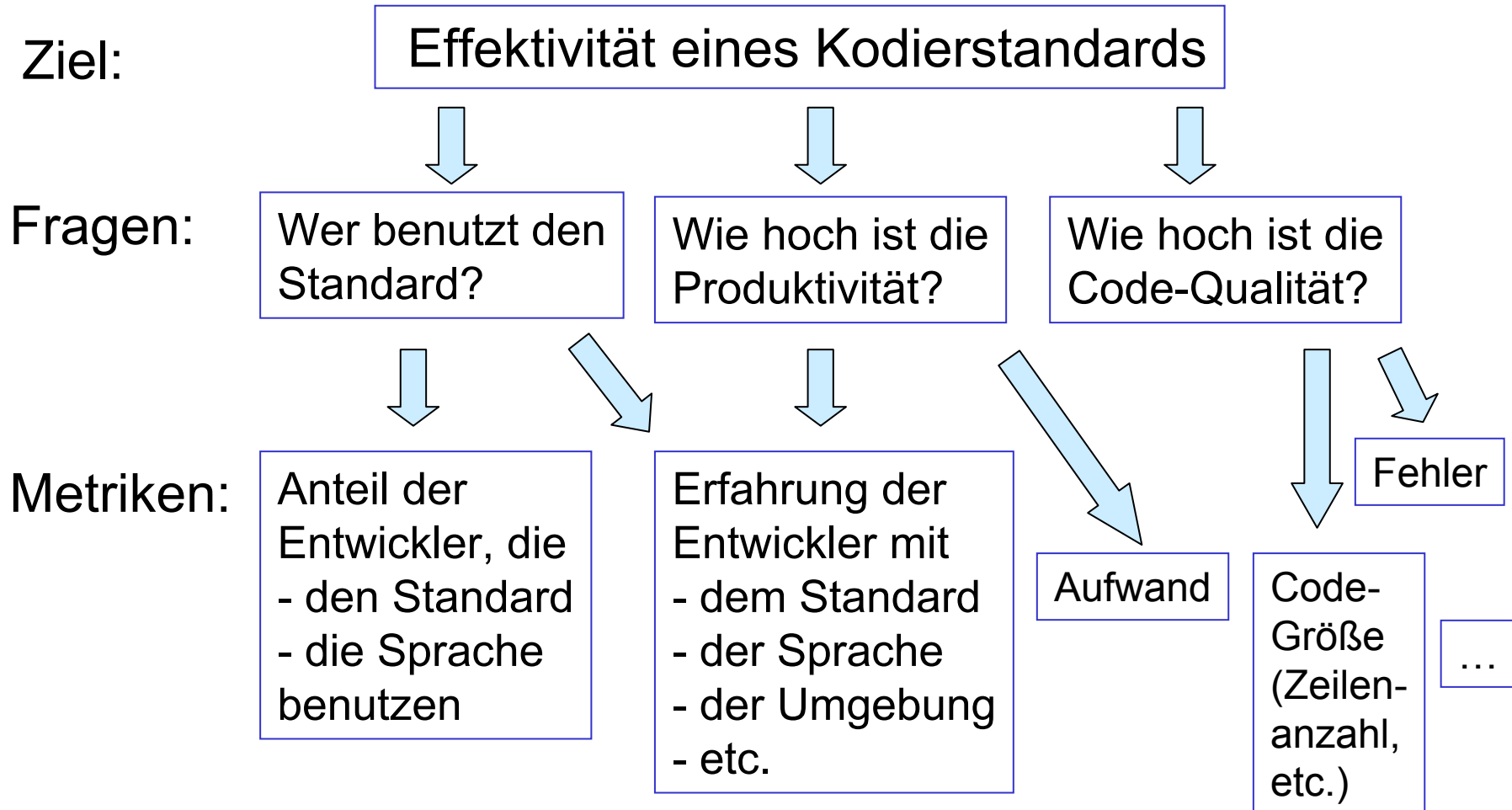
## **Manager:**

- Wie hoch sind die Entwicklungskosten?
- Wie produktiv sind die Mitarbeiter?
- Wie gut ist der entwickelte Code?
- Werden die Anwender mit dem Produkt zufrieden sein?

## **Entwickler:**

- Sind die Anforderungen testbar?
- Haben wir alle Fehler gefunden?
- Haben wir unsere Entwicklungsziele erreicht?
- Ist die Software wartbar?

# Was kann man messen?



# Szenario: Messen von Produktivität

Wie kann man die Produktivität eines Entwicklers testen?

- Anzahl von Codezeilen pro Zeiteinheit:
- Fred schreibt in 100 Tagen 5000 Zeilen Code.
- $P = 50 \text{ LOC / day}$
- Fred verdoppelt sein Programm, ohne Funktionalitätsänderung.
- $P = 100 \text{ LOC / day}$

Was wird hier gemessen?

Probleme:

- Wie ist die Anzahl von Codezeilen definiert?  
→ Exaktheit?
- Dieses Maß ist sprachabhängig.  
→ Vergleichbarkeit?
- Entwickler haben eigene Programmierstile.  
→ Normierung?
- ...

# Szenario: Messen von Produktivität (2)

## Function-Point-Analyse:

- Anzahl der Funktionspunkte pro Zeiteinheit:
- **Kategorien:** Ein- /Ausgabedaten, Abfragen, Datenbestände, Referenzdateien
- **Klassifikation:** einfach, mittel, komplex
- **Einflussfaktoren:** Kommunikation, Verarbeitungslogik,...

## Vorteile:

- misst den Wert des Outputs
- frühzeitig einsetzbar
- kann den Umfang von SW-Projekten messen
- kann Fortschritt messen
- technologieunabhängig

## Nachteile:

- stärkerer Messaufwand (Dokumentation)

# Szenario: Validation eines SW-Produkts

## Testmetriken

- Testkosten:
  - *Anzahl, Aufwand der Tests*
- Testfälle:
  - *Quantität*
  - *Komplexität: Testdaten, Intensität*
  - *Qualität: Intensität, Wiederverwendbarkeit, Anforderungen*
- Testüberdeckung:
  - *Code, Benutzerhandbuch*
- Vorteile:
  - *objektiv*
  - *zuverlässig*
  - *vergleichbar*
- Nachteile:
  - *höherer Messaufwand*

# Statische, konventionelle Metriken

- LOC - Anzahl der Codezeilen
  - *leicht zu messen*
  - *Umfang variiert mit der Sprache, Programmierstil, etc.*
- Halstead - Umfang von Ausdrücken
  - *Anzahl der Operatoren und Operanden*
  - *komplexe Strukturen nicht berücksichtigt*
- McCabe – Komplexität von Programmstrukturen
  - *Anzahl der binären Verzweigungen plus 1*
  - *je komplexer das Programm, desto höher das Risiko*
  - *ungeeignet für OO-Programme (zahlreiche einfache Methoden)*
- Hybride Metriken
  - *Kombination von mehreren Metriken*

# Objektorientierte SW-Metriken

- DIT – Depth of Inheritance Tree
  - *Anzahl der Oberklassen: Je mehr, desto fehleranfälliger*
- NOC – Number Of Children
  - *Anzahl der direkten Unterklassen: Je mehr, desto besser der Code.*
- RFC – Response For a Class
  - *Anzahl d. eigenen + aufgerufenen Methoden: Je mehr, desto fehleranfälliger*
- WMC – Weighted Method per Class
  - *Anzahl der definierten Methoden: Je mehr, desto fehleranfälliger*
- CBC – Coupling Between Classes
  - *Anzahl der benutzten Klassen: Je mehr, desto fehleranfälliger*

# SW-Metriken in der Industrie

- Welche Metriken werden eingesetzt?
  - *Größen- und Umfangsmetriken häufig, OO-Metriken wenig*
  - *Testmetriken sehr häufig: Festlegung des Testprozesses*
- Spezielle Metrik-Werkzeuge werden selten eingesetzt → Datenbank mit Tabellenkalkulation
- Analyse und Interpretation der Ergebnisse werden als problematisch empfunden.
- von den Entwicklern für das Projektmanagement durchgeführt
  - *Angst vor Kontrolle*
- Welche Gründe können gegen den Einsatz von Metriken sprechen?

# Kritik

- Das, was einen interessiert, kann man nicht direkt messen, z.B. die **Qualität** eines Produkts.
- Ausweg: **Hypothesen**, die auf Modellen basieren, Kombination ausgewählter messbarer Größen führt zu qualitativen Aussagen
- Zusammenhänge zwischen qualitativen und quantitativen Aussagen sind zum Teil recht simpel.
- Die **Ergebnisse der Messungen** müssen geeignet interpretiert werden:
  - *rückblickend: Hauptfokus auf Anomalien*
  - *planend: grobe Schätzung von Aufwand*

# Schlußbemerkungen

- **SW-Metriken** definieren, wie Kenngrößen der SW oder des SW-Prozesses gemessen werden.
- **Ziele für SW-Metriken:**
  - *Verstehen, Kontrollieren, Vorhersagen*
- **Erfolgskriterien für SW-Metriken:**
  - *exakte Messungen, automatisiert, Fokus auf Prozess / Produkt*
- **Basis-Literatur:**
  - *Fenton, Pfleeger: Software Metrics, PWS Publishing Company, 1997*

# Entwicklungsrichtlinien

30. April 2008



# Überblick

- Welche Entwicklungsrichtlinien gibt es?
- Warum gibt es Entwicklungsrichtlinien?
- Standards für Anforderungsspezifikationen
- Namensgebung
- Programmierrichtlinien für Java
- Richtlinien für die Entwicklung von graphischen Benutzeroberflächen von Eclipse-Plugins
- Versionierungsrichtlinien

# Warum Entwicklungsrichtlinien?

- konstruktive Qualitätssicherungsmaßnahme
- Für alle Phasen des SW-Entwicklungsprozesses nötig.
- Standards für Anforderungsspezifikationen
  - *strukturierte und vollständige Erfassung der Anforderungen*
  - *korrekte und genaue Anforderungen*
- Modellierungsrichtlinien kaum vorhanden.
- Programmierrichtlinien (inkl. Namensgebung)
  - *bessere Lesbarkeit des Codes*
  - *bessere Wartbarkeit*
  - *Unternehmenspolitik*
- Richtlinien für GUIs
  - *bessere Benutzbarkeit*
  - *stärkere Standardisierung*

# Arten von Programmierrichtlinien

- Namensgebung
- Layoutrichtlinien
- Strukturierungskonventionen
  - *Paket- und Klassenstruktur*
  - *Typisierung*
  - *Konventionen für Kontrollstrukturen*
- Konventionen für Kommentare

# Namenskonventionen nach Sun

- Alle Namen sollten in Englisch sein.
- Paketnamen:
  - *Keine Firmenname oder Namen von kommerziellen Produkten in Open-Source-Projekten.*
  - *Spezielle Paketnamen:*
    - internal – interne Implementierungspakete
    - tests – Paket für Testsuite
    - examples – Paket für Beispiele
  - *Paketnamen sollten nur kleingeschriebene alphanumerische Zeichen (ASCII) enthalten und Unterstrich (\_) oder Dollar (\$) vermeiden.*

# Namenskonventionen nach Sun

- **Klassennamen:**
  - *Substantive in gemischter Groß-/Kleinschreibung mit dem ersten Buchstaben jedes internen und des ersten Wortes großgeschrieben*
  - *Klassennamen sollten einfach und beschreibend sein.*
  - *Ganze Wörter verwenden, keine Abkürzungen, außer gebräuchliche wie HTML oder UML*
  - *Abkürzungen nur am Anfang groß, z.B.: „Html“ statt „HTML“*
- **Schnittstellennamen:**
  - *wie Klassennamen*
  - *Schnittstellenkonvention: „I“ als Präfix, z.B. „IWorkspace“ oder „IIndex“*
  - *Dadurch werden Schnittstellen leichter erkennbar.*

# Namenskonventionen nach Sun

- **Methodennamen:**
  - *Verben in gemischter Groß-/Kleinschreibung mit dem ersten Buchstaben jedes internen Wortes großgeschrieben*
  - *besondere Konventionen für:*
    - Getter: z.B. „getX()“
    - Setter: z.B. „setX()“
    - Prädikate: z.B. „isEmpty()“
  - *keine Objektamen in Methodennamen, also „line.getLength()“ statt „line.getLineLength()“*
  - *Funktionen werden nach ihrem Rückgabewert benannt, Prozeduren (void-Methoden) nach dem, was sie tun*
- **Konstantennamen:**
  - *komplett großgeschrieben, einzelne Wörter mit „\_“ getrennt*

# Namenskonventionen nach Sun

- Variablennamen:
  - *kurz, aber prägnant*
  - *Variablennamen der Länge 1 sollten vermieden werden, außer für „Wegwerf-“variablen*
  - *Generische Variablen heißen wie ihr Typ, z.B.: `setTopic(Topic topic)`*
  - *Werden Variablen über einen längeren Bereich genutzt, haben sie einen längeren Namen als kurz benutzte V.*
  - *Variablen für Collections im Plural, z.B. `int[] values`;*
- Klassen- und Instanzvariablen:
  - *in gemischter Groß-/Kleinschreibung mit dem ersten Buchstaben jedes internen Wortes großgeschrieben und dem ersten Buchstaben kleingeschrieben*
  - *sollten nicht mit „\_“ oder „\$“ starten*
  - *Private Klassenvariablen sollten mit „\_“ enden.*

# Namenskonventionen nach Sun

- Systemdateien und -ordner
  - *starten mit „.“*
  - *Sie sollen nicht von den Benutzern geändert werden.*
- Settings-Ordner
  - *Dateien in diesem Ordner müssen nicht mit „.“ starten. Sie werden automatisch als Systemdateien identifiziert.*
  - *Spezielle Dateien bei Eclipse:*
    - Datei: `.classpath`
    - Daten: `.project`

# Weitere Namenskonventionen

- Komplementäre Namen für komplementäre Einheiten, z.B. get/set, add/remove, start/stop, etc.
- Keine negierten Namen für boolesche Variablen, also „isFound“ statt „isNotFound“
- „No“-Endung für Variablen, die eine Anzahl darstellen, z.B. „employeeNo“
- Namen für ähnliche Konstanten sollten gleich beginnen, z.B. COLOR\_RED und COLOR\_GREEN
- Namen von Exception-Klassen enden auf „Exception“
- Default-Schnittstellenimplementierungen beginnen mit „Default“.
- Singleton-Klassen geben ihr Objekt mit „getInstance“ zurück.
- Factory-Klassen erzeugen Objekte durch „new[ClassName]“-Methoden.

# Generelle Layoutrichtlinien

- Eine Zeile ist maximal 80 Zeichen lang.
- Spezielle Zeichen wie TAB und Seitenumbruch sollten vermieden werden.
- Umbruch von zu lange Zeilen:
  - *nach einem Komma*
  - *nach einem Operator*
  - *Neue Zeile startet am Beginn des Ausdrucks in der vorigen Zeile.*

# Paket und Import-Anweisungen

- Alle Dateien sollten zu einem Paket gehören.
- Die wichtigen Importe sollten zuerst genannt werden. Import-Anweisungen sollten gruppiert werden.
- Importierte Klassen sollten immer explizit genannt werden.
  - *„import java.util.List;  
import java.util.HashSet;“  
statt „import.util.\*;“*

# Klassen und Schnittstellen

- Deklarationen von Klassen und Schnittstellen sollten wie folgt aussehen:
  - *Klassen-/Schnittstellendokumentation*
  - *class oder interface Anweisung*
  - *Klassenvariablen, geordnet nach Sichtbarkeit*
  - *Instanzvariablen, geordnet nach Sichtbarkeit*
  - *Konstruktoren*
  - *Methoden*
- Sichtbarkeitsordnung:
  - *public, protected, package, private*

# Typen

- Typkonvertierungen müssen explizit gemacht werden.
  - *„floatValue = (float) intValue;“* statt *„floatValue = intValue;“*
- Array-Information muß der Typ, nicht die Variable tragen.
  - *„int[] a = new int[20];“* statt *„int a[] = new int[20];“*

# Variablen

- Variablen sollten dort deklariert werden, wo sie gebraucht werden, und immer im kleinsten Bereich.
- Variablen haben keine mehrfache Bedeutung.
- Instanzvariablen sollten niemals als „public“ deklariert sein.
  - *Ausnahme: Klasse ist eine reine Datenstruktur*
- Variablen sollten, wann immer möglich, als “final“ deklariert sein, insbesondere Methodenparameter.

# Schleifen

- Schleifenvariablen sollten direkt vor der Schleife initialisiert werden.
- „Do-while“-Schleifen sollten vermieden werden.
- „break“ und „continue“ sollten vermieden werden.

# Bedingungen

- Komplexe Bedingungen sollten vermieden werden.  
Besser: temporäre Variablen verwenden
- Der Normalfall sollte in If-Bedingungen abgefragt werden.
- Die Bedingung sollte in einer eigenen Zeile stehen.
- Bedingungen sollten keine ausführbaren Anweisungen enthalten.
  - *„int length = list.getLength();  
if ( length != null) „ statt  
„if (list.getLength() != null)“*

# Vermischtes

- Außer 0 und 1 sollten Nummern nicht direkt im Code verwendet werden. Stattdessen besser eine Konstante deklarieren.
- Dezimalzahlen:
  - *immer mit Dezimalpunkt, also „0.0“ statt „0“*
  - *immer mit einer Ziffer vor dem Punkt, also „0.5“ statt „.5“*
- Statische Variablen sollten immer mit Klassennamen referenziert werden.
- Zwischenraum:
  - *um Operatoren und Doppelpunkt*
  - *nach Schlüsselwörtern, Semikolon und Komma*

# Kommentare

- Schwieriger Code soll nicht kommentiert, sondern verbessert werden.
  - *ausdrucksstarke Namen*
  - *explizite logische Struktur*
- Alle Kommentare in Englisch.
- Spezielle Form für Javadoc-Kommentare.
  1. *Beschreibung der Klasse / Methode / Klassenvariablen*
  2. *Beschreibung der Parameter (in, return)*
  3. *Beschreibung der geworfenen Exceptions*
- „//“ für alle Nicht-Javadoc-Kommentare
- Alle public-Klassen und alle public- und protected-Methoden in public-Klassen sollten mit Javadoc dokumentiert sein.

# Richtlinien für graphische Benutzeroberflächen

- Generelle Richtlinien
  - *Microsoft User Experience*
  - *Macintosh Human Interface Guidelines*
  - *Java Look and Feel Guidelines*
  - *Eclipse User Interface Guidelines*
- Design-Prinzipien für Benutzeroberflächen:
  - *Benutzer wird geführt*
  - *direkt*
  - *konsistent*
  - *vergibt Benutzerfehler und gibt Feedback*
  - *ästhetisch*
  - *einfach*
  - *„Corporate Design“*

# Grundsätzliches Design

- Identifikation der Schlüsselfunktionalitäten, diese sollten im Fokus stehen
- Beispiele:
  - *Java IDE: Editieren und Debugging von Java Code*
  - *Email: Erzeugen, Senden, Lesen und Antworten auf Emails*
- Fehler:
  - *Fehler, die die Aufmerksamkeit der Benutzer verlangen, sollten durch einen modalen Dialog angezeigt werden.*
  - *Programmfehler sollten auch in einer Log-Datei gespeichert werden.*

# Zusammenfassung

- Entwicklungsrichtlinien als konstruktive Qualitätssicherungsmaßnahme
- Programmierrichtlinien für bessere Lesbarkeit des Codes
- Richtlinien für graphische Benutzeroberflächen für leichtere Benutzbarkeit
- Werkzeuge, die Programmierrichtlinien prüfen:
  - *Eclipse-Plugin Checkstyle*
  - *Eclipse Plugin PMD*

# Literatur

- Code Conventions for the Java Programming Language  
<http://java.sun.com/docs/codeconv/>
- Eclipse User Interface Guidelines  
[http://wiki.eclipse.org/User\\_Interface\\_Guidelines](http://wiki.eclipse.org/User_Interface_Guidelines)
- Java Look and Feel Guidelines  
<http://java.sun.com/products/jlf/ed2/book/>

# Bad Code Smell

7. Mai 2008



# Überblick

- Was ist Bad Code Smell?
- Welche Arten von Bad Code Smell gibt es?
  - *klassenintern*
  - *klassenübergreifend*
- Welche Werkzeuge zur Analyse von Bad Code Smell gibt es?
- Eclipse-Plugin PMD

# Was ist Bad Code Smell?

- anrühige (verdächtige) Code-Stellen
- Hier lohnt es sich, genauer hinzuschauen.
- Sollten eventuell durch Refactoring-Maßnahmen restrukturiert werden.
- Kondensierung von Erfahrungswissen
- syntaktische Prüfung von Programmcode hinsichtlich bestimmter Muster
- Überblick über diese Muster in:  
M. Fowler: Refactoring, Addison-Wesley, 2005

# Klasseninterne Bad Smells

- doppelter Code
- lange Methoden
- lange Parameterlisten
- viele Änderungen einer Klasse
- temporäre Felder
- sich wiederholende Switch-Anweisungen
- Kommentare

# Doppelter Code

- eines der häufigsten Bad Code Smells
- gleicher oder sehr ähnlicher Code, der mehrfach auftritt
- Hauptgrund: Copy-and-Paste-Programmierung
  - *schnelle Wiederverwendung von Code*
- Beispiele:
  - *derselbe Codeblock in zwei Methoden derselben Klasse*
  - *oder in zwei Unterklassen einer gem. Oberklasse*
  - *oder in zwei komplett verschiedener Klassen*

# Beispiel: Doppelter Code

```
void printInformation() {
    printTitle();
    //print details
    System.out.println ("name: " + name);
    System.out.println („address: " + getAddress());
}

void printAddressBook() {
    for(int i = 1; i <= book.length; i++) {
        //print address
        System.out.println ("name: " + book[i].name);
        System.out.println („address: " + book[i].getAddress());
    }
}
```

# Lange Methoden

- Vermeidung von „Spaghetti-Code“
- je länger eine Methode, desto fehleranfälliger
- statt Kommentierung von Code-Blöcken, besser den Block als Methode rausziehen
- Prinzipielles Vorgehen:
  - *Code-Blöcke identifizieren*
  - *nötige Parameter und temporäre Variablen identifizieren*
  - *komplexe Bedingungen zerschlagen*

# Lange Parameterlisten

- Lange Parameterlisten erschweren das Verständnis von Methodenaufrufen.
- Parameter besser als globale Variablen?
- Einsparen von Parametern:
  - *Verwendung von Objektanfragen*
  - *Verwendung von Parameterobjekten*

- Beispiel:

Calendar
insertAppointment(start: Time, end: Time) showAppointments(start: Time, end: Time) cancelAppointments(start: Time, end: Time)

- Wann könnten lange Parameterlisten gewollt sein?

# Viele Änderungen einer Klasse

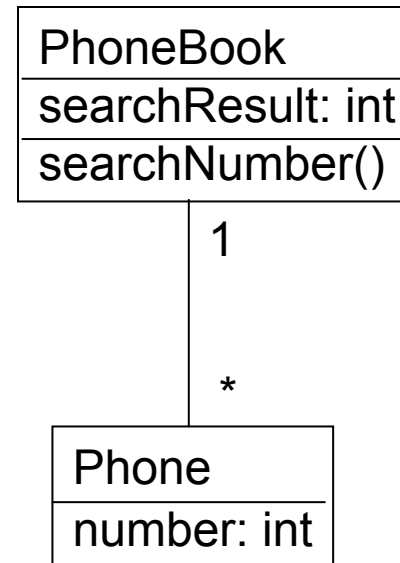
- verschiedene Änderungswünsche führen zu Änderungen derselben Klasse
- Beispiele für häufige Änderungen einer Klasse:
  - *Änderung einer Datenzugriffsklasse, falls sich die Datenbank ändert*
  - *Codeänderungen, wenn sich Geschäftsregeln ändern*
  - *Einfügen neuer Aspekte (Z.B. wird ein Logging-Mechanismus eingeführt.)*
- Beispiel:
  - *eine Klasse, die zu viele verschiedenartige Attribute hat*

Person
name: String
phone: int
fax : int
street: String
zip: int
city: String

# Temporäre Felder

- Felder einer Klasse beschreiben die Zustände ihrer Objekte.
- Manchmal sind Felder nur zeitweise gesetzt.
- Zur Vermeidung von langen Parameterlisten werden Felder erzeugt.

- Beispiel:



# Sich wiederholende Switch-Anweisungen und Kommentare

## Switch-Anweisungen:

- häufig ähnliche Switch-Anweisungen an mehreren Stellen im Code
- besser die Switch-Anweisung in eine eigene Methode extrahieren

## Kommentare:

- Kommentare sind an sich nichts Schlechtes.
- Im Gegenteil: Sie sind gut!
- Manchmal weisen Kommentare auf schlechten Code hin.
- Kruder Code soll durch Kommentare verständlicher werden.

# Klassenübergreifende Bad Smells

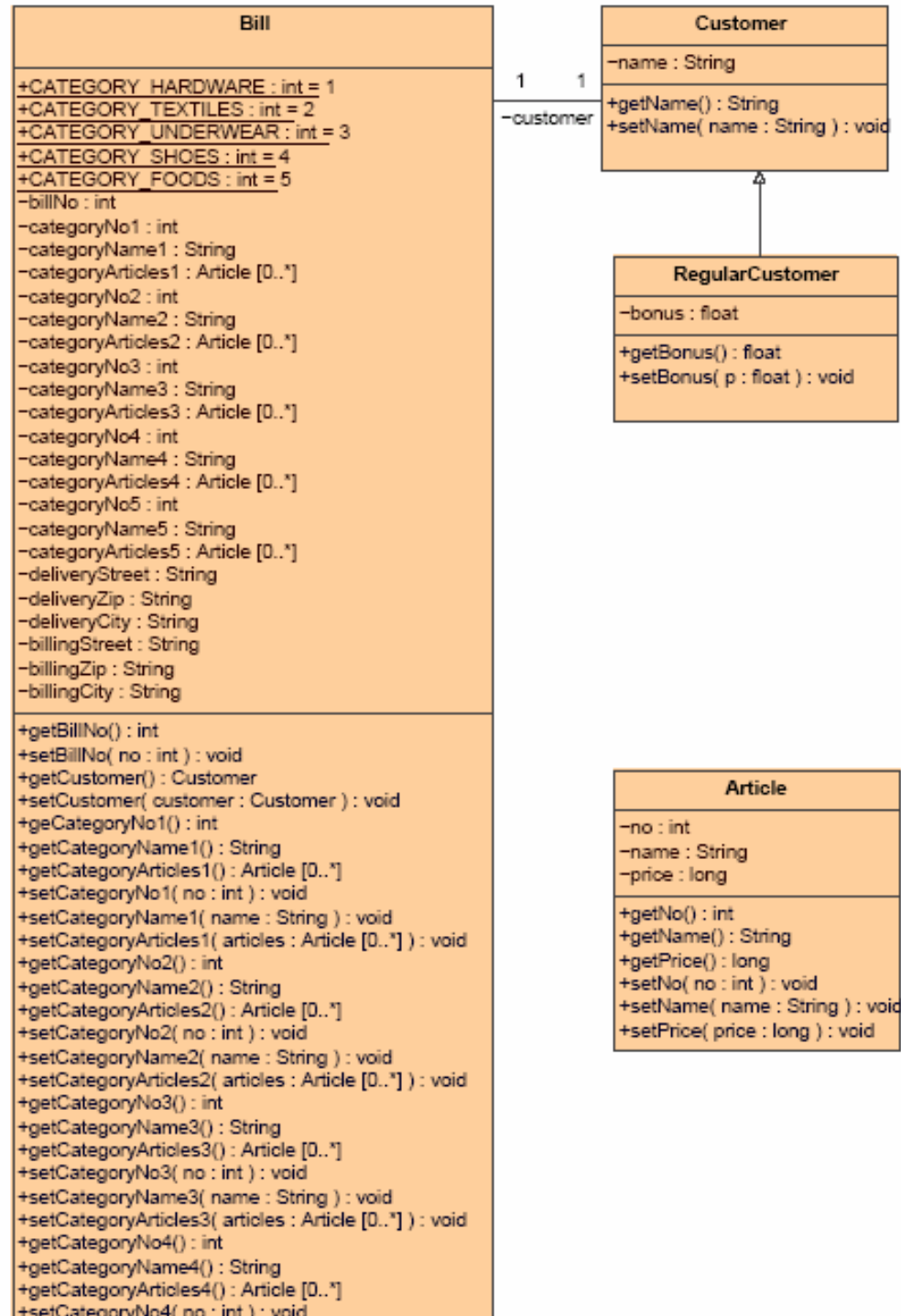
- große Klassen
- Änderungen in vielen verschiedenen Klassen aufgrund von einem Änderungswunsch
- Methoden, die zuviel auf andere zugreifen
- Datenhaufen
- faule Klassen
- Nachrichtenketten
- verweigertes Erbe
- auf Verdacht allgemein gehaltener Code

# Große Klassen

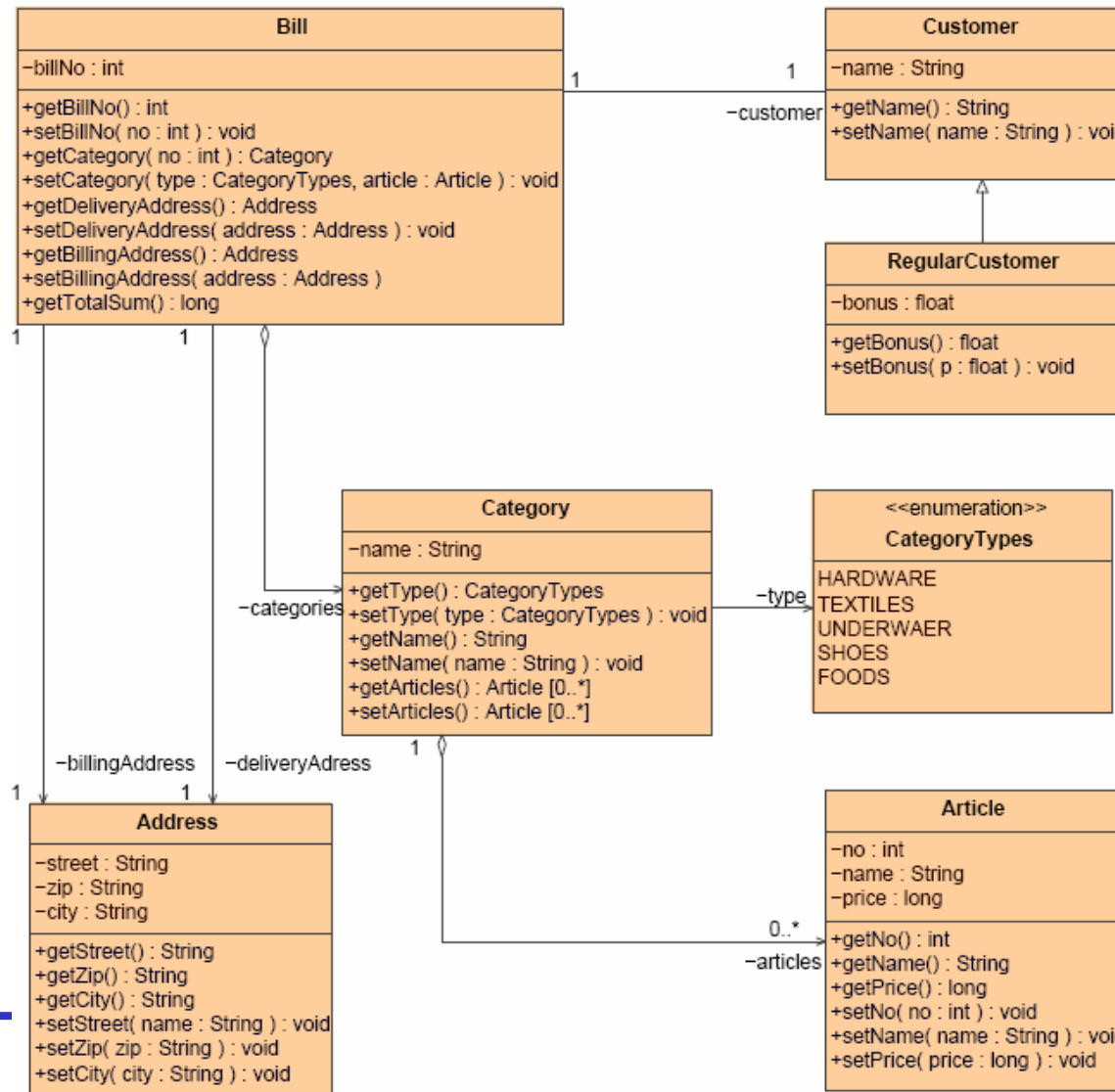
- meist zu viele Felder
- Redundanzen innerhalb der Klasse vermeiden
  - *„Turn five 100 line methods into five 10 line methods with another ten 2 line methods extracted from the original.“*
- Szenarien:
  - *Manche Felder lassen sich gut in neuen Objektklassen zusammenfassen.*
  - *Bei zu viel Code: Methoden herausziehen, durch Verkürzung der Methoden redundanten Code finden*
- Herausziehen von Schnittstellen kann aufzeigen, wie eine Klasse in mehrere Klassen zerlegt werden kann.
- Konvertierung von Konstanten in Aufzählungstypen

# Beispiel: Große Klassen

vorher



# Beispiel: Große Klassen



nachher

# Änderung viele Klassen betreffend

- Ein Änderungswunsch, der viele Klassen betrifft.
- Änderungsstellen sind schwer zu finden, es kann schnell eine vergessen werden.
- Klassenstruktur semantisch nicht passend
- Manchmal auch gewollt. Wann?
- Copy-and-Paste-Programmierung:
  - *schnelle Wiederverwendung*
  - *Wenn der Code anschließend stark modifiziert wird, ist Copy-and-Paste adäquat.*
- Boredom-Is-A-Smell:
  - *dieselben Einträge bei vielen Klassen*
  - *verbesserte Austauschformate*

# Methoden, die zuviel auf Daten anderer Klassen zugreifen

- Der Fokus einer Klasse sind ihre Daten.
- verdächtig: eine Methode, die mehr auf Daten anderer Klassen zugreift
- Methode sollte zu der Klasse gehören, deren Daten sie überwiegend verwendet

```
class Room {  
    Lecture[] lectures;  
}  
  
class Lecture {  
    int id;  
    boolean isInRoom(Room r) {  
        for(int i= 0; i < r.lectures.length; i++)  
        {  
            if (r.lectures[i].id == id)  
                return(true); }  
        return(false);  
    }  
}  
  
... if (l.isInRoom(r)) ...
```

# Datenhaufen und faule Klassen

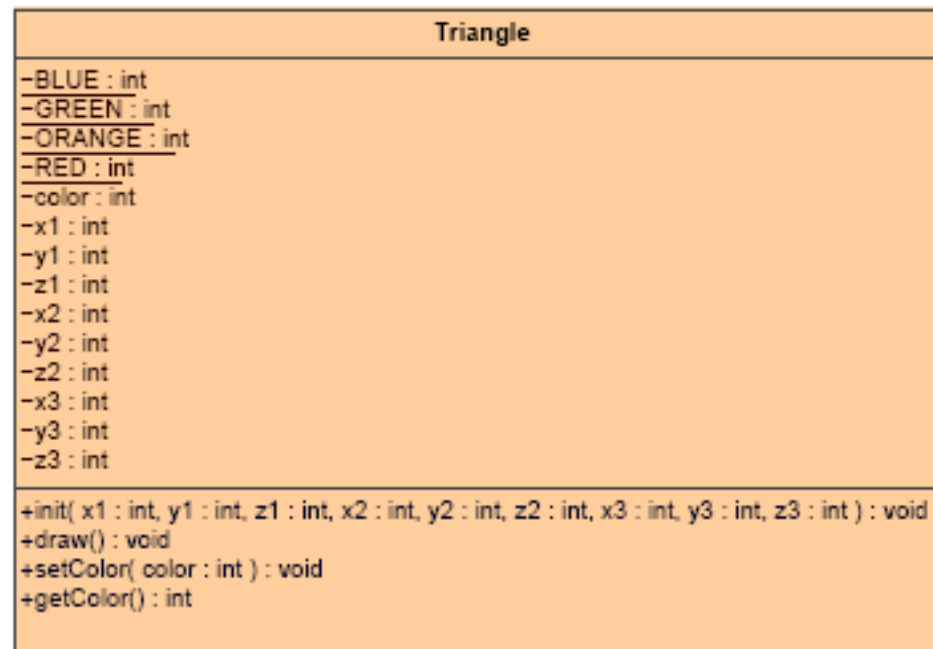
## Datenhaufen:

- Oft hängen mehrere Daten zusammen.
- Wenn Daten immer gemeinsam auftreten → neue Datenklassen
- Welche Vor- und Nachteile bringen neue Datenklassen?
- Datenklassen müssen noch mit Funktionalität angereichert werden.

## Faule Klassen:

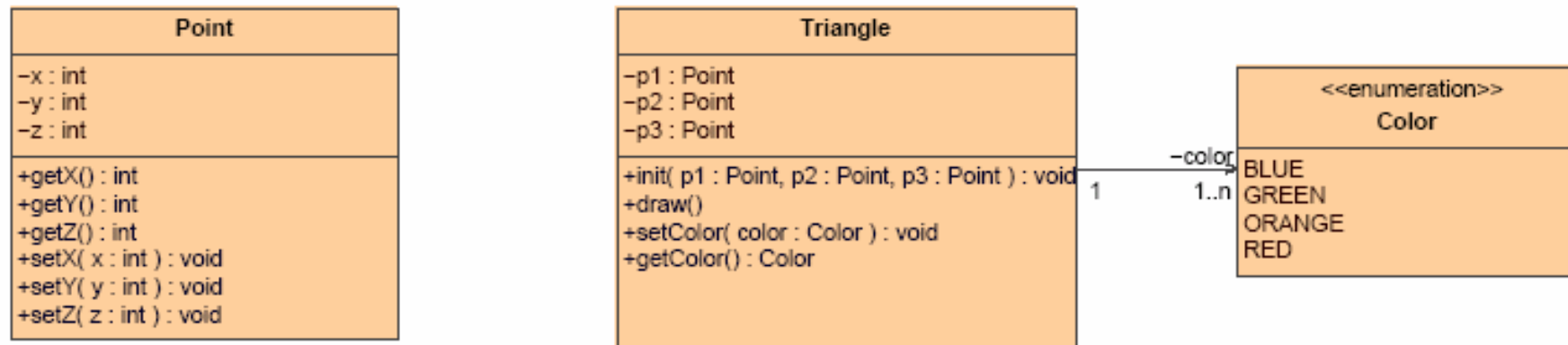
- Klassen mit zu wenig Funktionalität
- mögliche Gründe:
  - *zu ausgeprägte Klassenhierarchie*
  - *reine Datenklassen*

# Beispiel 1: Bad Code Smell

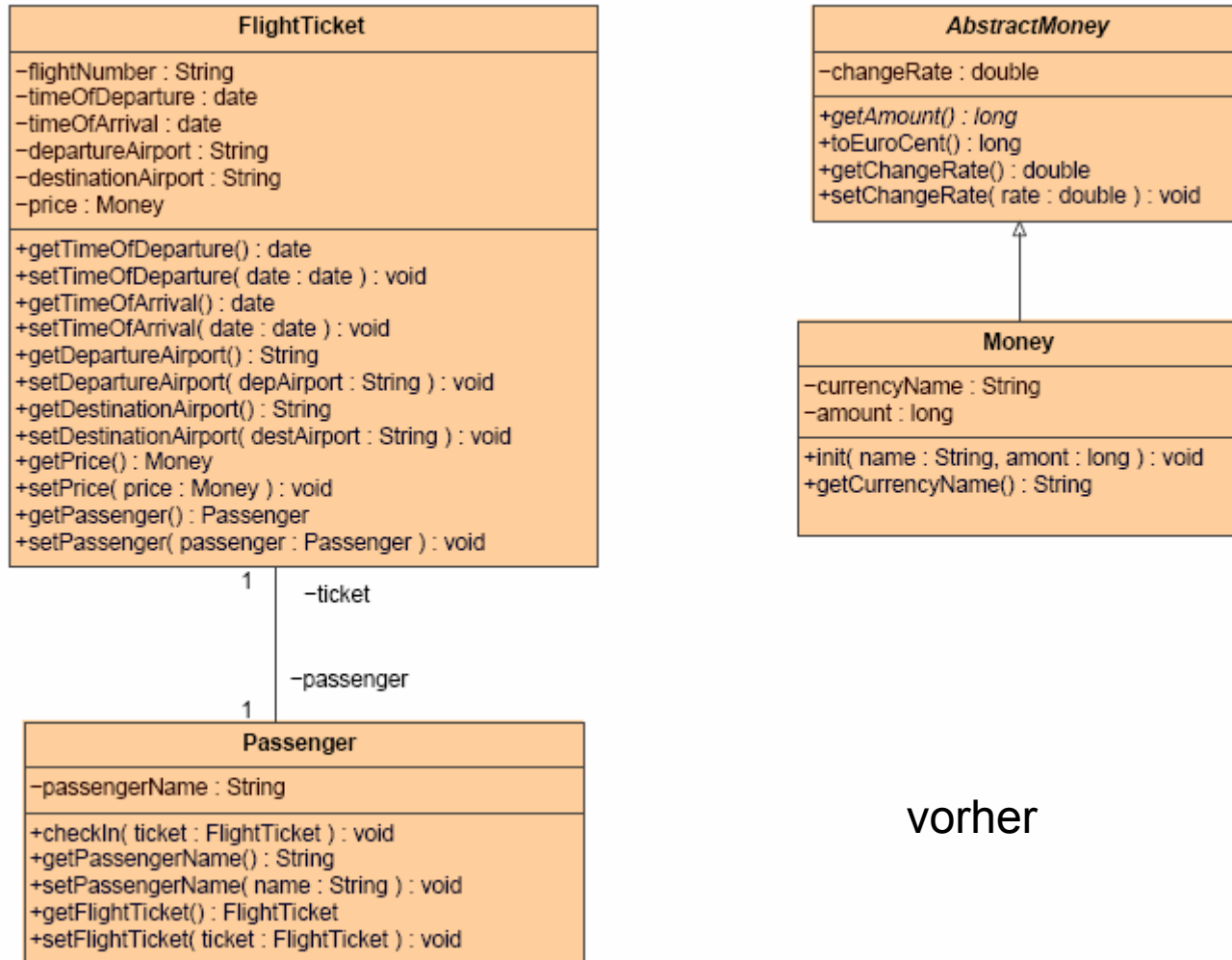


vorher

# Beispiel 1: Bad Code Smell



# Beispiel 2: Bad Code Smell



vorher

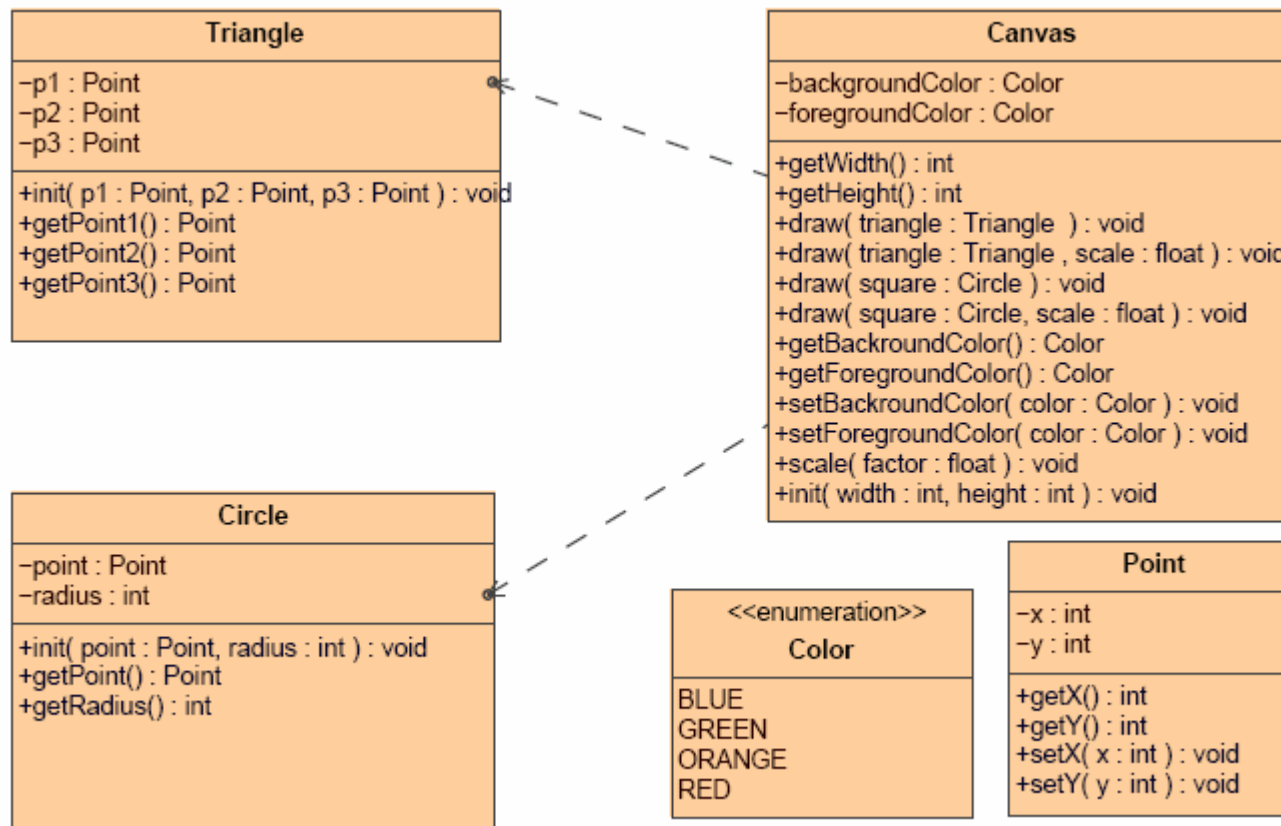
# Beispiel 2: Bad Code Smell

FlightTicket
-flightNumber : String -timeOfDeparture : date -timeOfArrival : date -departureAirport : String -destinationAirport : String -price : Money -passengerName : String
+getTimeOfDeparture() : date +setTimeOfDeparture( date : date ) : void +getTimeOfArrival() : date +setTimeOfArrival( date : date ) : void +getDepartureAirport() : String +setDepartureAirport( depAirport : String ) : void +getDestinationAirport() : String +setDestinationAirport( destAirport : String ) : void +getPrice() : Money +setPrice( price : Money ) : void +getPassengerName() : String +setPassengerName( name : String ) : void +checkIn() : void

Money
-currencyName : String -amount : long -changeRate : double
+init( name : String, amont : long ) : void +getChangeRate() : double +setChangeRate( rate : double ) : void +getCurrencyName() : String +getAmount() : long +toEuroCent() : long

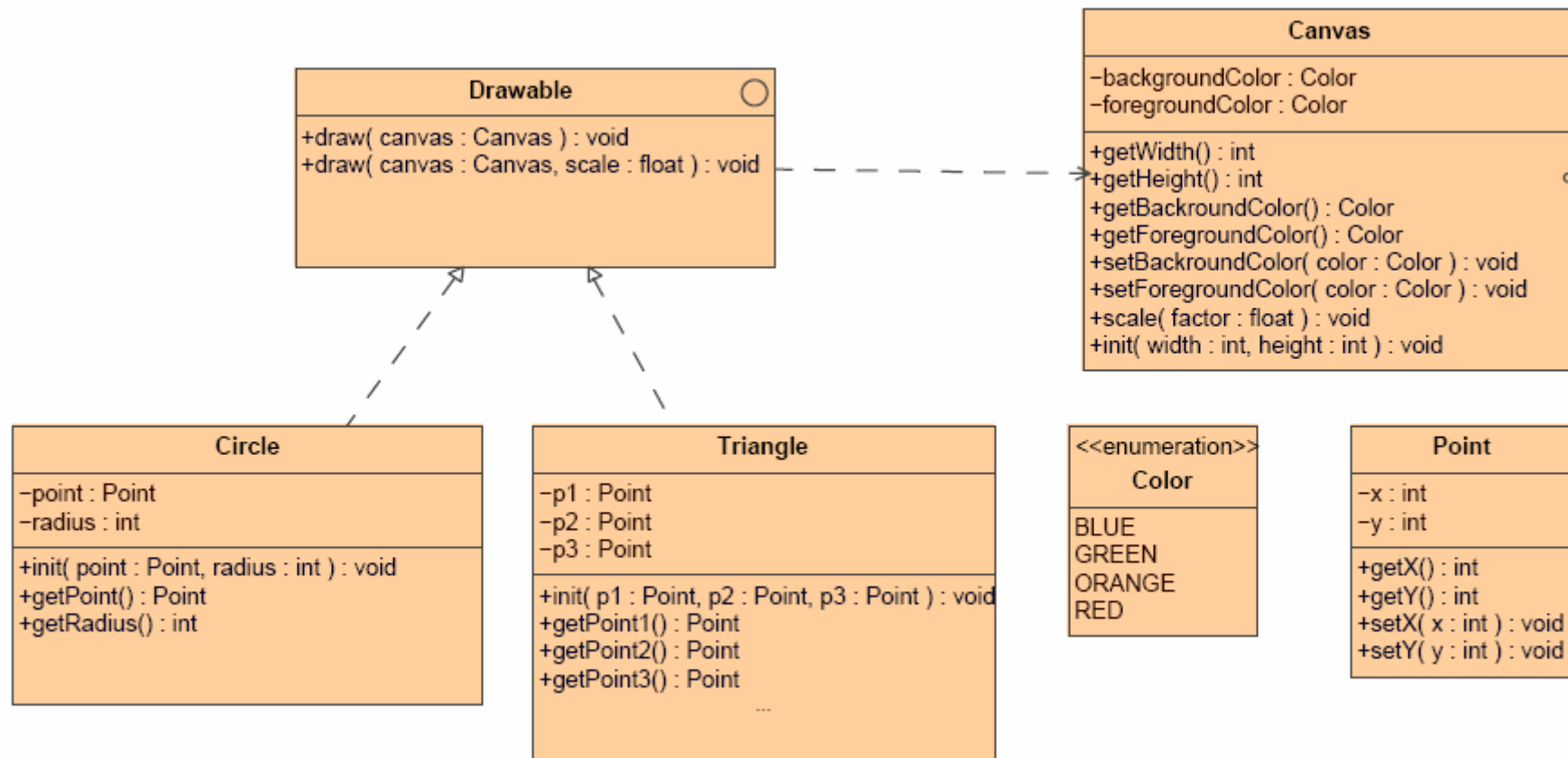
nachher

# Beispiel 3: Bad Code Smell



vorher

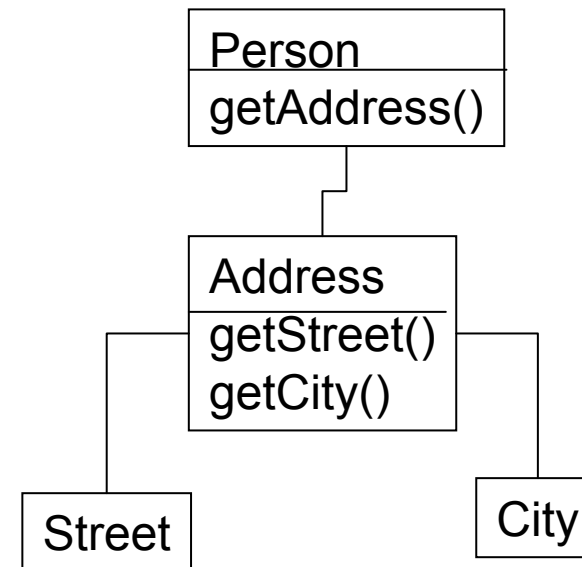
# Beispiel 3: Bad Code Smell



nachher

# Nachrichtenketten

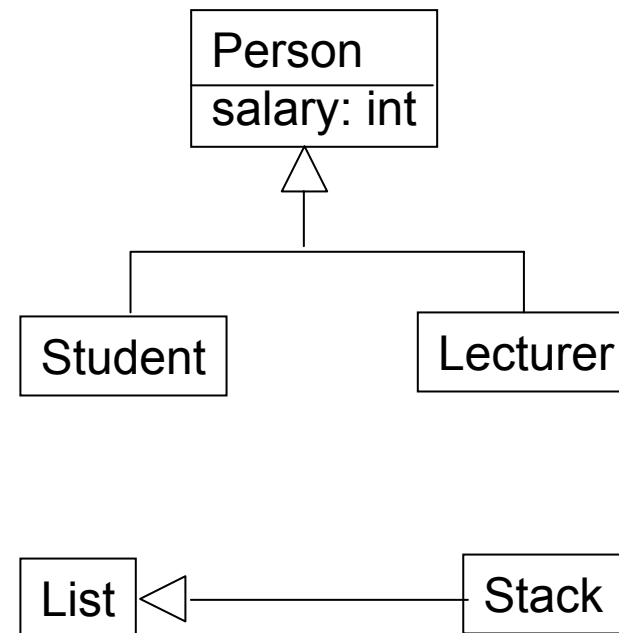
- Kaskaden von Methodenaufrufen sind verdächtig.
  - Einkapselung geht häufig mit Delegation einher.
  - Wenn eine Klasse viele seiner Methodenaufrufe an eine andere Klasse delegiert, kann sie eliminiert werden.
  - Zwei Klassen, die zu stark miteinander kommunizieren, sind verdächtig.
- Beispiel:
    - `getPerson(i).getAddress().getStreet().getHouseNumber().print();`



# Verweigertes Erbe

- Subklassen, die zu wenige Methoden und Felder von ihren Elternklassen verwenden
- Schnittstellen sollten immer wiederverwendet werden.
- Methodenrümpfe müssen nicht wiederverwendet werden.
- Eintrag ist nur für eine Schwesterklasse interessant
- durch Delegation ersetzte Vererbung

- Beispiele:



# Auf Verdacht allgemein gehaltener Code

- Code für alle Eventualitäten
  - abstrakte Klassen, die nicht viel Funktionalität enthalten
  - Parameter, die nicht gebraucht werden
  - Code, der nur in Testklassen verwendet wird
- Beispiele:
    - *Methodenname reicht nicht, um den Inhalt zu verstehen*
    - *Eine Methode wird immer mit dem gleichen Wert aufgerufen.*
    - *Abstrakte Klasse mit nur einer Unterklasse*
    - *Template, das mit nur einem Argumenttyp sinnvoll ist*

# Code Smell Metriken

- Umfangsmetriken
  - *Methoden mit mehr als 20 Code-Zeilen, mehr als 3 Parameter oder mehr als 2 Ebenen tief*
  - *Klassen mit mehr als 10 öffentlichen Methoden und einigen wenigen privaten Methoden*
- Namensgebung
  - *schlechte Namen für Klassen, Methoden oder Variablen*
- doppelte Code-Fragmente
- nicht durch Tests abgedeckte Code-Fragmente

# Werkzeuge

- Werkzeuge, mit denen man Bad Code Smell entdecken kann:
  - *Eclipse Plugin Metrics*
  - *Eclipse Plugin Checkstyle*
  - *Eclipse Plugin PMD*
  - ...

# Zusammenfassung

- Qualitätssicherungsmaßnahme für
  - *bessere Wartbarkeit*
  - *geringere Fehleranfälligkeit*
- Ziel der Analyse durch Bad Code Smell:
  - *Vermeidung von Coderedundanzen: „Once and only once“*
  - *bessere Strukturierung des Codes*
  - *Auffinden von ungenutzten Codeanteilen*
- Zwei Arten der Programmierung:
  - *pragmatisch: Code Smells werden von Fall zu Fall angesehen.*
  - *puristisch: Alle Code Smells werden eliminiert.*
- Werkzeuge können automatisch verdächtige Code-Stellen finden.

# Software-Refactoring

14. Mai 2008



# Überblick

- Was ist Refactoring und wozu dient es?
- Welche Refactorings gibt es?
  - *Refactoring-Katalog: [www.refactoring.com](http://www.refactoring.com)*
- Wann, wo und wie führt man Refactorings durch?
- Wie definiert man neue Refactorings?
- offene Probleme bzgl. Refactoring
- Werkzeugunterstützung für Refactoring
  - *Refactoring in Eclipse*
- Literatur:
  - *M. Fowler: Refactoring, Addison-Wesley, 2005*

# Was ist Refactoring?

- eine Technik im Rahmen der iterativen Softwareentwicklung
- Restrukturierung der Software nach jedem Iterationsschritt
- „Refactoring ist der Prozess, ein Softwaresystem so zu verändern, dass das externe Verhalten unverändert bleibt, der Code aber eine bessere Struktur erhält.“ (Martin Fowler)

# Wozu dient Refactoring?

- Verbesserung der Software, wobei die Funktionalität unverändert bleibt
- aus Sicht des Benutzers nicht (oder kaum) bemerkbar
- Erhöhung der Übersichtlichkeit und Verständlichkeit
- vereinfachte Wartbarkeit und Erweiterbarkeit
- verbesserte Lesbarkeit
- leichter testbar
- bessere Modularität (bessere Wiederverwendbarkeit)
  - *z.B. keine Mehrfachimplementierung*

# Welche Refactorings gibt es?

- mittlerweile ein umfangreicher Katalog
- typische Refactoring-Methoden:
  - *Umbenennungen (Felder, Methoden, Klassen, etc.)*
  - *Entfernung von redundantem Code*
  - *Verschiebung eines Symbols in ein anderes Modul (z.B. Methode in andere Klasse)*
  - *Aufteilung oder Zusammenlegung eines Moduls (z.B. Paket oder Klasse)*

# Konkrete Refactoring-Methoden

Eine Auswahl ....

Innerhalb von Methoden:

- Extract Method
- Inline Method
- Replace Temp With Query
- Inline Temp

Verschiebung zwischen Klassen:

- Move Method / Field
- Extract Class
- Hide Delegate

Datenorganisation:

- Replace Data Value With Object
- Encapsulate Field
- Replace Array with Object

Vereinfachte Bedingungen:

- Decompose Conditional
- Introduce Assertion
- Replace Conditional With Polymorphism

# Konkrete Refactoring-Methoden

## Vereinfachte Methoden:

- Rename Method
- Add/Remove Parameter
- Introduce Parameter Object
- Remove Setting Method
- Replace Error Code with Exception
- Replace Exception with Test

## Änderungen an der

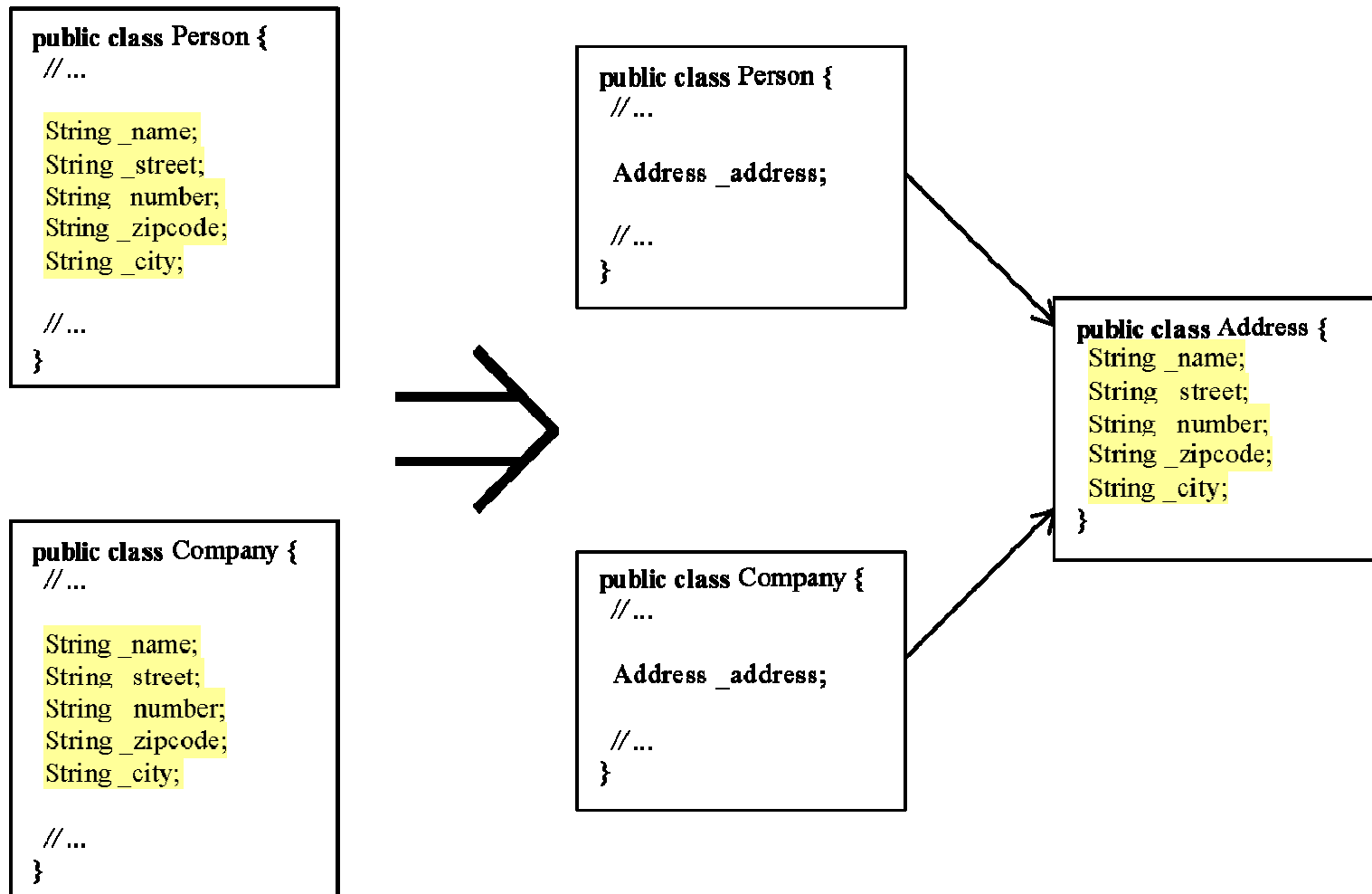
## Vererbungshierarchie:

- Pull up Field / Method
- Push Down Field /Method
- Extract Superclass / Subclass
- Replace Inheritance With Delegation

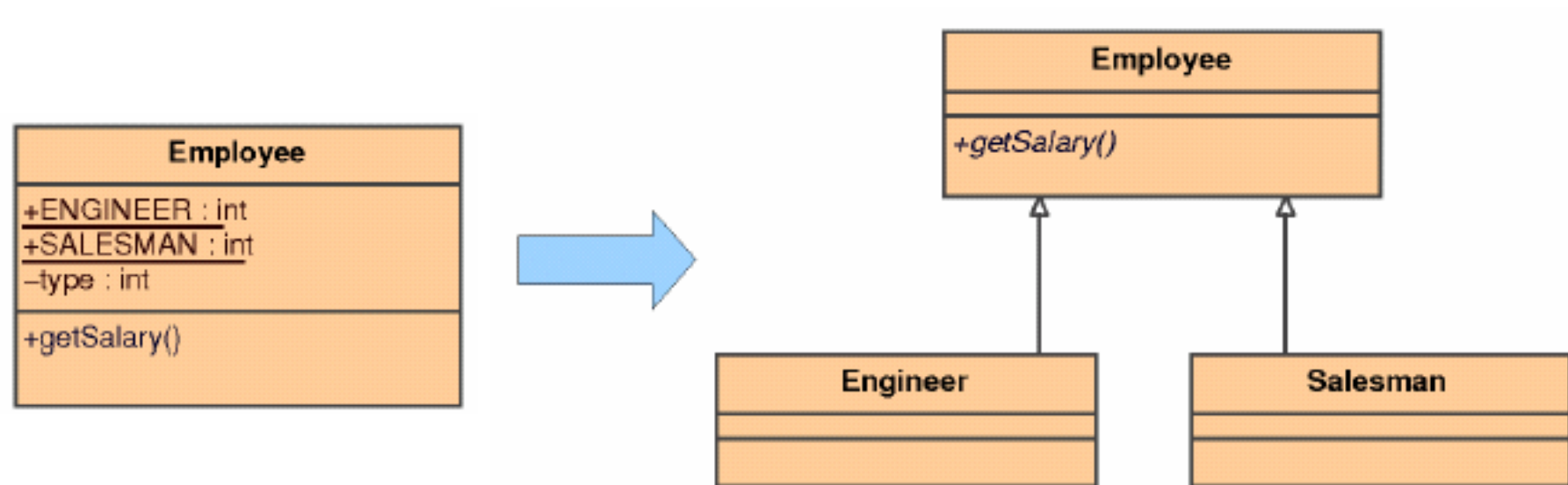
## Große Refactorings:

- Tease Apart Inheritance
- Convert Procedural Design to Objects

# Refactoring-Beispiel 1



# Refactoring-Beispiel 2



Ersetze Typkonstanten durch Unterklassen

# Beispiel: Ersetze Typkonstanten durch Unterklassen

- Ziel:
  - *Varianten sind in Unterklassen gekapselt*
- Eingabeparameter:
  - *Klasse*
  - *Typ-Konstanten dieser Klasse*
- Effekt:
  - *für jede Konstante: erzeuge Unterklasse mit Namen der Konstanten*
  - *lösche alle Typ-Konstanten*

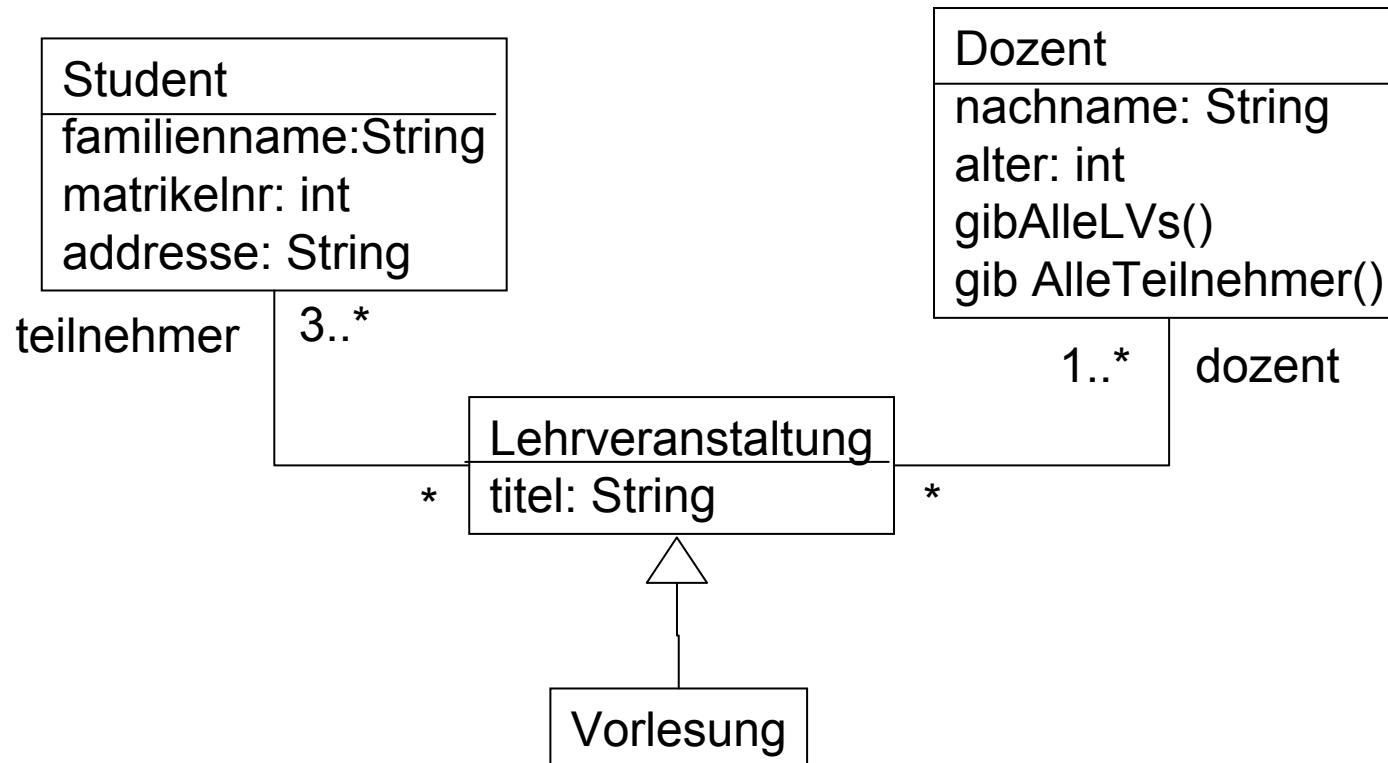
# Durchführung von Refactorings

- nach einem Iterationsschritt
- auf funktionsfähigem Code
- fortlaufend in kleinen Schritten
- Tests nach jedem Schritt:
  - *Funktionalität muss erhalten bleiben*
  - *Veränderung soll zu einem effektiveren System führen*
- Komplexe Refactorings werden in mehrere kleine Refactorings zerteilt.

# Wo führt man Refactorings durch?

- Problem: Wie findet man die Stellen, an den Refactorings durchgeführt werden sollten?
- Bad Code Smells: Anzeichen auf schlechte Programmstruktur
- Katalog über Bad Code Smells und entsprechenden Refactorings:  
<http://sis36.berkeley.edu/projects/streek/agile/bad-smells-in-code.html>
- Einführung von Design Patterns

# Refactoring-Beispiel 3



Welche Refactorings sollten hier durchgeführt werden?

# Definition von Refactorings

- Refactoring als Programm- und Modelltransformation
- Definition von Refactorings
  - *mit einer Programmiersprache implementiert*
  - *mit einer Skriptsprache beschrieben*
  - *mit der Object Constraint Language (OCL) beschrieben*
  - *mit Query/View/Transformation (QVT) beschrieben*
  - *mit Graphtransformation beschrieben*
  - *...*
- Refactoring für
  - *eine spezielle Anwendungsdomäne*
  - *eine neue Sprache*

# Qualität von Refactorings

- Verhaltensbewahrung:
  - *Was ist das Verhalten eines Systems?*  
*Auch wenn die Funktionalität gleich bleibt, kann sich der Zeit- oder Speicherplatzbedarf ändern.*
  - *Wie weist man die Bewahrung von Verhalten nach?*
- Strukturverbesserung:
  - *Wie misst man den Effekt eines Refactorings auf die Softwarequalität?*

# Zusammenfassung

- Refactoring heißt Verbesserung der Software, wobei die Funktionalität unverändert bleibt.
- konstruktive Qualitätssicherungsmaßnahme
- umfangreicher Katalog an konkreten Refactoring-Methoden
- Werkzeugunterstützung:
  - *Eclipse Refactoring-Werkzeug*
  - *weitere IDEs, wie z.B. IntelliJIDEA*
- Offene Fragen:
  - *Wie weist man Verhaltensbewahrung von Refactorings nach?*
  - *Wie führt man kohärentes Refactoring von Model und Code durch?*

# Design Patterns und Model-Driven Architecture

21. Mai 2008

---

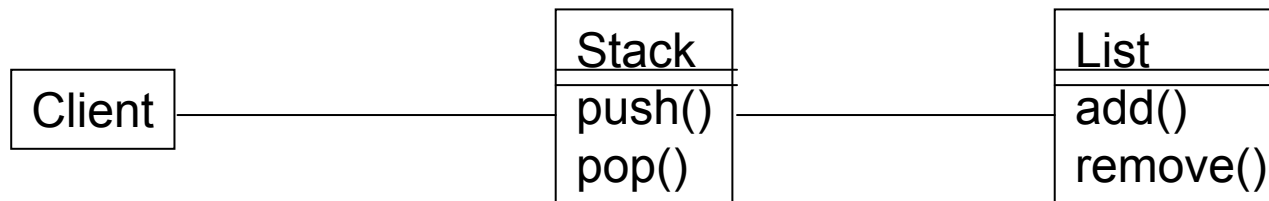
# Überblick

- Was sind Design Patterns?
- Welche Design Patterns gibt es?
- Refactoring und Design Patterns
- Diskussion: Design Patterns kontra Softwarearchitekturen
- MDA und der Traum vom automatisierten Entwickeln
- Wie funktioniert modellgetriebene Softwareentwicklung?
- Qualität von modellgetriebenen Entwicklungen

# Was sind Design Patterns?

- Muster, die sich im Softwareentwurf bewährt haben
- Gute Design Patterns entstammen direkt der Praxis.
- Antwort auf wiederkehrende Probleme im SW-Entwurf
- kein fertiges Modell, das direkt in Code übersetzt werden kann
- eher eine Art Schablone für die Lösung eines Problems
  - *Kritik: Design Patterns müssen immer wieder neu implementiert werden.*
- Literatur:
  - *E. Gamma, R. Helm, R. Johnson, J. Vlissides: Design Patterns: Elements of Reusable Object-Oriented Software (ISBN 0-201-63361-2)*

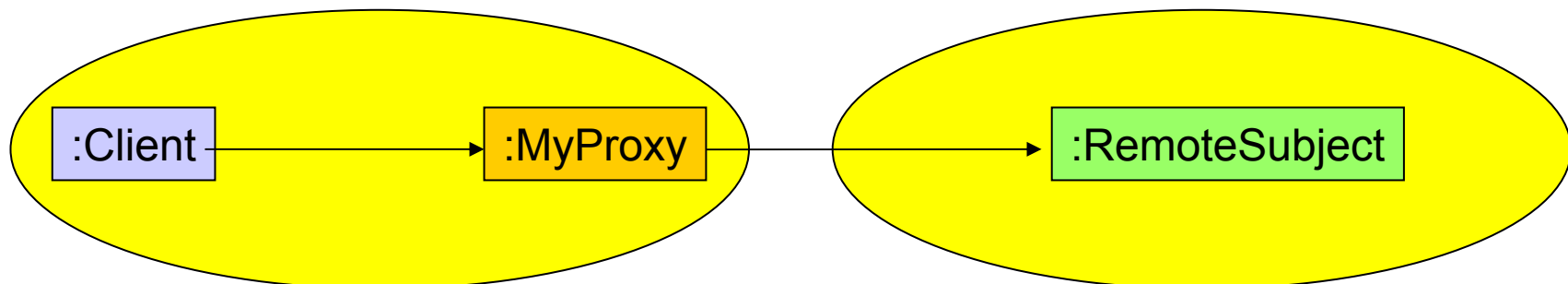
# Delegation Pattern



- Ein Objekt zeigt Funktionalität nach außen, die es intern an andere delegiert.
- Kritik an der Einführung dieses Patterns?

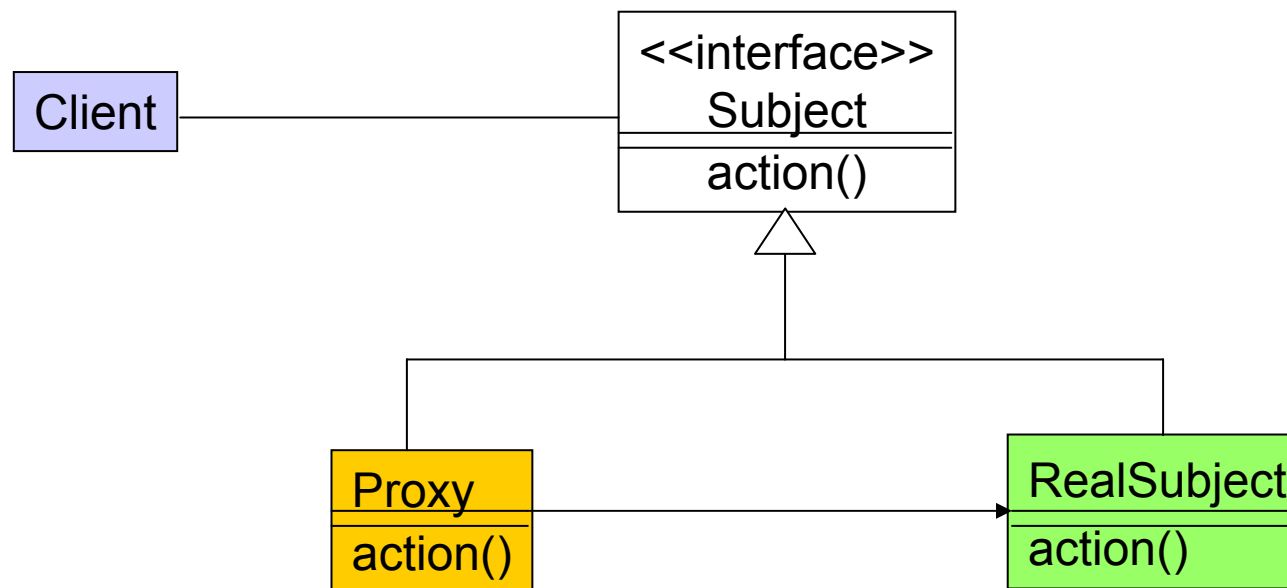
# Proxy Pattern

- Proxy als Schnittstelle zu einem anderen Ding
- typisches Beispiel: ein Remote-Proxy
  - *Ein Objekt liegt auf einer anderen Maschine. Das Remote-Proxy ist die Schnittstelle zum entfernt liegenden Objekt.*



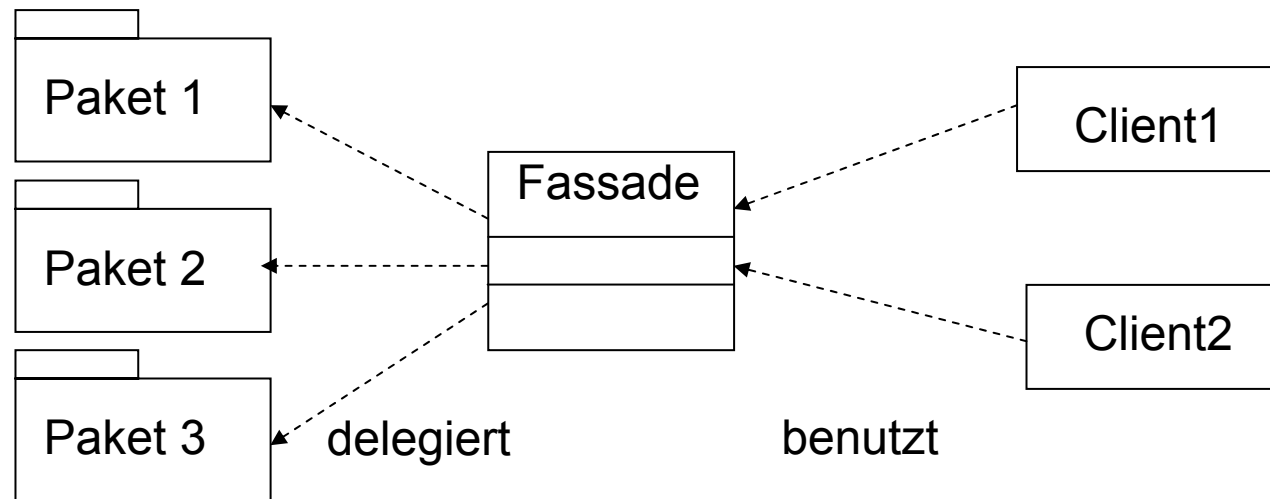
# Proxy Pattern

- Client kommuniziert nur über eine Schnittstelle mit dem Proxy bzw. realen Subjekt.



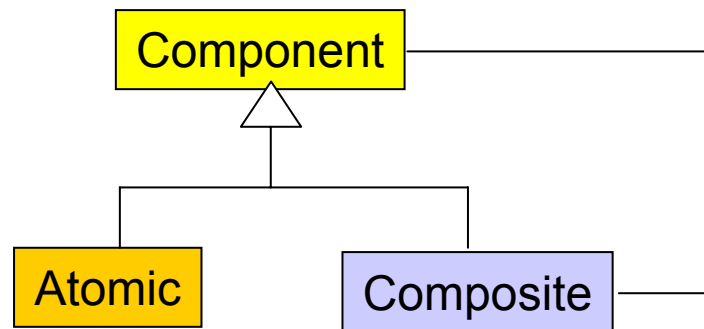
# Facade Pattern

- Eine Softwarekomponente ist sehr komplex.
- Deshalb: Bau einer Fassade, hinter der die Komplexität versteckt wird.
- Beispiel: Implementierung des UML-Metamodells

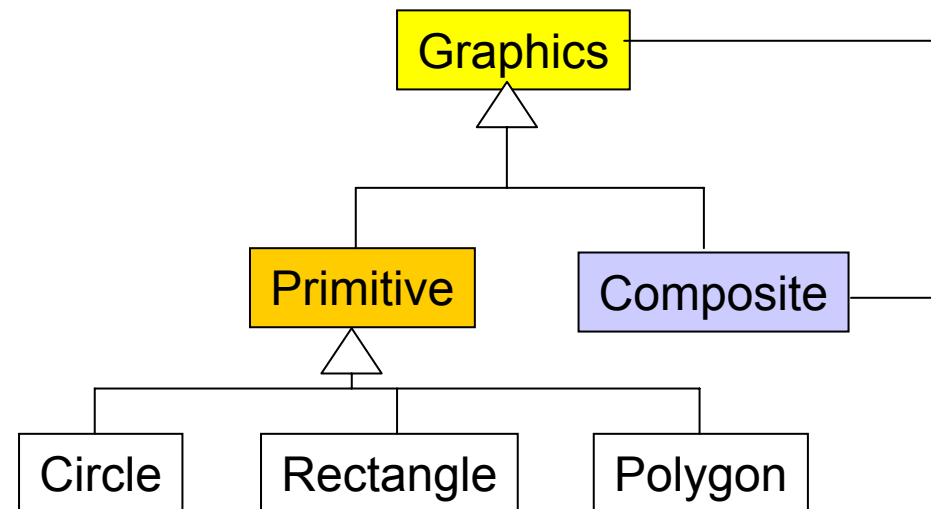


# Composite Pattern

- Einzelne Instanzen und Gruppen von Instanzen können gleich behandelt werden.



Beispiel:



# Observer Pattern



## Wirkungsweise:

- `attach()`: Observer anmelden
- `detach()`: Observer abmelden
- `notify()`: Observer benachrichtigen
  - *for all `o:Observer` {  
`o.update()` }*

## Eigenschaften:

- Broadcast-Benachrichtigung
- Subjekte niederer Schichten können mit Observern höherer Schichten kommunizieren, ohne die Schichtung zu verletzen.
- typisches Beispiel: Client-Server-Systeme

# Weitere Design Patterns

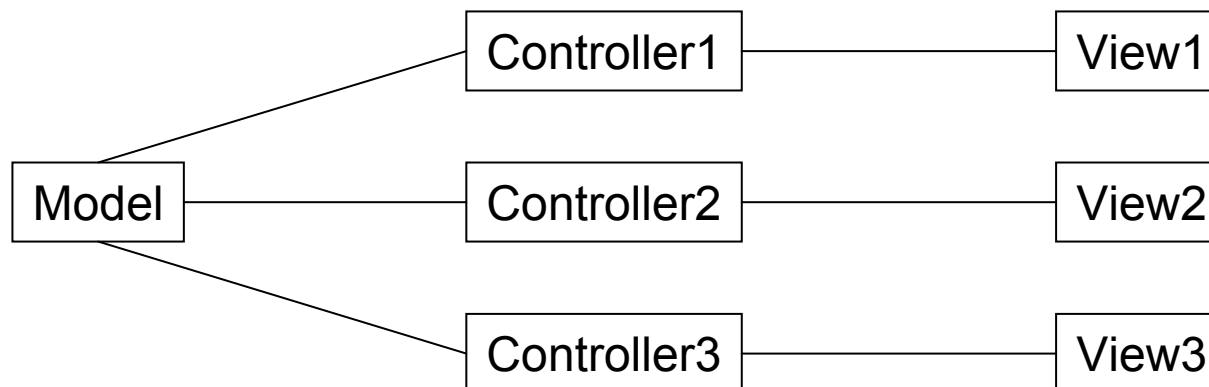
- Adapter Pattern
    - *passt eine vorhandene Schnittstelle an die Bedürfnisse der Anwendung an*
  - Decorator Pattern
    - *fügt zusätzliche Funktionalität während der Laufzeit zu*
  - Command Pattern
    - *stellt Operationen als Objekte mit Methode execute() dar*
  - Factory Pattern
    - *zentralisiertes Erzeugen von Objekten*
  - Singleton Pattern
    - *Instanziierung einer Klasse durch nur ein Objekt*
  - Strategie Pattern
    - *leichte Wahl zwischen verschiedenen Implementierungen*
- ... und viele weitere

# Design Patterns und Refactoring

- Refactoring, um Design Patterns in die eigene Anwendung zu integrieren
  - *Joshua Kerievsky, Refactoring To Patterns, Addison Wesley, 2004*
  - *Refactoring To Patterns Catalog:*  
*<http://industriallogic.com/xp/refactoring/catalog.html>*
- Auffinden von Anti-Patterns und Ersetzen durch Design Patterns
  - *Anti Patterns Catalog:*  
*<http://c2.com/cgi/wiki?AntiPatternsCatalog>*
  - *Wie weit geht dieses Verfahren?*

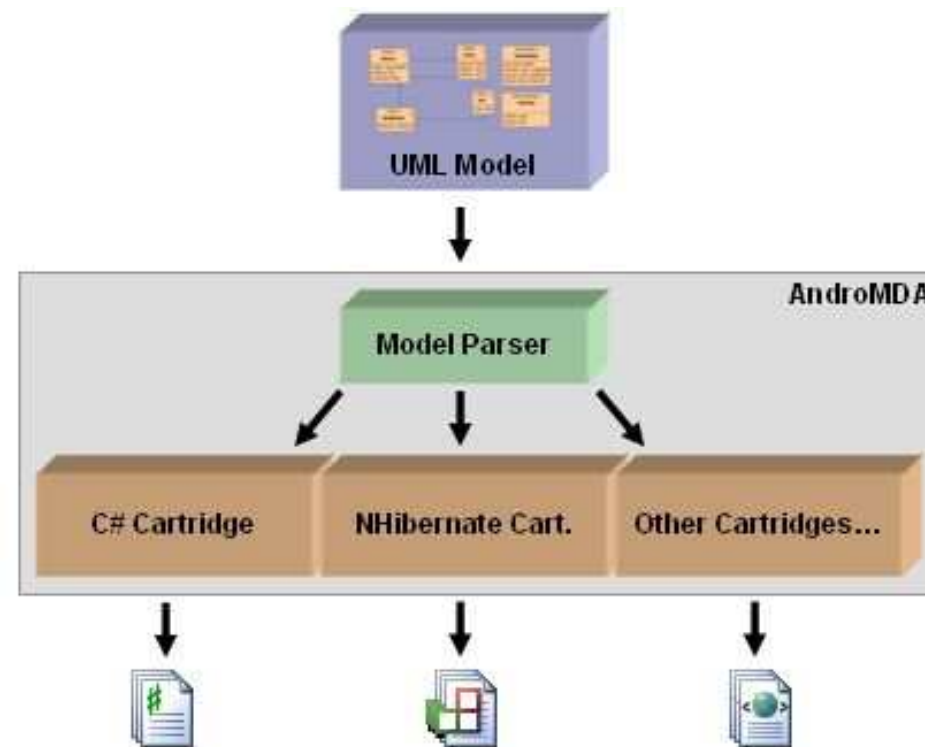
# Entwurfsmuster in der Model-View-Controller-Architektur

- Die Objekte sind voneinander getrennt: Observer Pattern
- View-Objekte können andere View-Objekte enthalten: Composite Pattern



# Idee der modellgetriebenen SW-Entwicklung

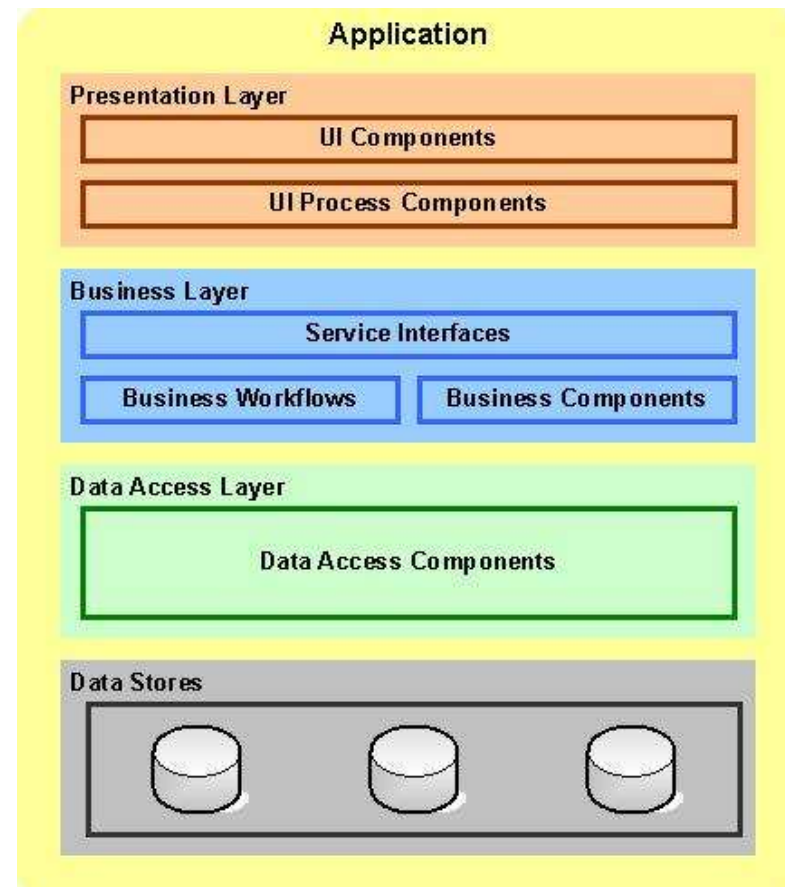
- „Model once, run anywhere.“
- Modell enthält applikationsspez. Wissen.
- Codegenerierung erfolgt (weitgehend) automatisch.
- Codegenerierung enthält Technologie-wissen.



aus „andromda.org“

# Architektur von modernen Geschäftsanwendungen

- Präsentationsschicht: Benutzerinteraktionen (Web-orientiert)
- Geschäftslogik: komplexe, länger anhaltende, zustandsbehaftete Transaktionen (Dienste)
- Datenzugriffsschicht: simple Datenzugriffe
  - *Create, Read, Update, Delete (CRUD) Operationen*
- Datenhaltungsschicht:
  - *Datenbank*
  - *Dateisystem*



aus "androMDA.org – Application Architecture"

# Was kann modellgetriebene Softwareentwicklung leisten?

- verkürzte Entwicklungszeiten
- schnelle Erstellung von Prototypen
- weniger technisches Wissen
- leichtere Umstellung auf neue Technologien
- stärkere Entkoppelung von Domänenwissen und Technologiewissen
- Kohärenz von Modell und Code
- wohldefinierte Softwarearchitektur, besserer Code
- aktuelle Dokumentation
- Unterstützung bei der Erstellung automatischer Tests

# Verwendung von MVC in MDA

- generierte Webanwendungen:
  - *Model: Daten und Geschäftslogik*
  - *View: Webpräsentation*
- generierte visuelle Editoren:
  - *Model: abstrakte Syntaxstruktur*
  - *Views: verschiedene visuelle und textuelle Sichten*
- Generierungsprozess:
  - *Model: domänenspezifisches Modell, aus dem generiert wird*
  - *Views: generierter Code für verschiedene Plattformen*

# Qualität der generierten Software

- wohldefinierte Software-Architektur
  - *Modell zwingt zur Einhaltung einer gewissen Architektur*
  - *generierter Code folgt einer vorgegebenen Architektur*
- konserviertes Expertenwissen
  - *Verwendung von verschiedenen Frameworks*
  - *Generator enthält das Expertenwissen zur richtigen Verwendung dieser Frameworks*
- stringente Entwicklungsrichtlinien
  - *Eventuell manuell erstellter Code wird an bestimmten Stellen in den generierten Code eingefügt und setzt auf klar strukturierten Schnittstellen auf.*

# Qualität der generierten Software

- Die Qualität von generiertem Code kann ebenso gut wie von manuell erstelltem sein, wenn
  - *das Modell gut ist und*
  - *der Codegenerator sorgfältig entwickelt worden ist.*
- Meist ist generierter Code systematischer und konsistenter. (warum?)
- Probleme mit generiertem Code?

# Aktuelle Dokumentation

- Das erstellte Modell ist immer aktuell und bietet einen guten Überblick über die Software.
  - *sehr kompakte Informationsdarstellung*
  - *visuelle Elemente zur Darstellung von Strukturen*
- Neben dem Code können auch die Online-Hilfe und weitere Dokumentation aus dem Modell generiert werden.
- Daneben müssen auch die verwendete Modellierungssprache und der Codegenerator dokumentiert werden.
  - *nur einmal zu dokumentieren, nicht anwendungsabhängig*

# Wiederverwendbarkeit

- MDD bietet ein hohes Maß an Wiederverwendbarkeit
  - *Aufteilung in domänen- und anwendungsspezifischen Code*
- Infrastruktur für die Erstellung von Software in einem bestimmten Bereich
  - *domänenspezifische Modellierungssprache*
  - *domänenspezifische Plattform mit Generatoren*
- Die Infrastruktur ist hochgradig wiederverwendbar.

# Portabilität, Anpassbarkeit

- aufgrund des MDA-Ansatzes
  - *durch plattformunabhängiges Modell leichte Portabilität auf andere Plattformen, die durch die MDD-Infrastruktur unterstützt werden*
  - *schnelle Portabilität auf neue oder geänderte Plattformen (durch Transformationsanpassung)*
- strukturierte Anpassung an neue Domänenanforderungen
  - *Anpassung der Modellierungssprache*
  - *Anpassung des Codegenerators*
  - *Anpassung der übrigen Infrastruktur*

# Zusammenfassung

- Design Patterns kondensieren das Wissen von erfahrenen Softwareentwicklern.
- Design Patterns und MDA-Entwicklungen tragen konstruktiv zur Qualitätssicherung bei.
- Design Patterns und Softwarearchitekturen sind Lösungen, die sich in der Praxis bewährt haben
- Codegenerierung in MDA verwendet bewährte Design Patterns und SW-Architekturen.
- MDA-Infrastrukturen sind hochgradig wiederverwendbar.

# Einführung in Testprinzipien und Profiling

28. Mai 2008

---

# Überblick

- semantische Qualität von Software
- Testmanagement
  - *Psychologie des Testens*
  - *Wirtschaftlichkeit*
- Teststrategien und -prinzipien
  - *Blackbox / Whitebox – Tests*
- Profiling
  - *Speicheranalyse*
  - *Analyse der Ausführungszeiten*
  - *Codeüberdeckung*
- Profiling-Werkzeuge

# Semantische Qualität

- Die semantische Qualität von Software bezieht sich auf die Erfüllung der Anforderungsspezifikation:
- Erfüllung der funktionale Anforderungen:
  - **Funktionalität:** Korrektheit, Vollständigkeit, Sicherheit, Fehlertoleranz
- Erfüllung der folgenden nichtfunktionalen Anforderungen:
  - **Effizienz:** Wirtschaftlichkeit, Zeitverhalten, Verbrauchsverhalten

# Nachweismöglichkeiten für semantische Qualität

- Testen: exemplarische Programmausführungen, um Fehler (Defekte) zu finden
- Zusicherungen: Konsistenzeigenschaften, die während der Programmausführung überprüft werden
- Verifikation: Nachweis, dass alle Programmausführungen bestimmte Eigenschaften erfüllen

Welche Vor- und Nachteile haben diese verschiedenen Nachweismöglichkeiten?

# Motivation – warum Testen?

- Tests sind einfach durchzuführen und mindestens so wichtig wie Programmierung.
- Tests werden oftmals vernachlässigt:
  - *meist wegen Zeit- und Spaßmangel*
- Es gibt nur beschränkte Möglichkeiten zur Durchführung von Tests:
  - *Testausgaben: direkt in Code und Ausgabe*
  - *Debugging: schrittweises Ausführen eines Programms*
  - *Profiling: Testen von Effizienzeigenschaften*
  - *Automatisches Testen mit Testfällen: Definition von Testfällen in separaten Testklassen, Zusicherung einer Testeigenschaft*

# Psychologie des Testens

- „Testen ist der Prozeß, ein Programm mit der Absicht auszuführen, Fehler zu finden.“ (Myers)
- Beurteilung von Testergebnissen:
  - *erfolgreicher Testlauf: kein Fehler gefunden*
  - *nicht erfolgreicher Testlauf: Fehler gefunden*
  - *Bei einem erfolgreichen Test wurden nur Zeit und Geld verschwendet?*
  - *Was ist ein erfolgreicher Testlauf, wenn es um Effizienz geht?*
- Wie kann man durch Tests zeigen, dass ein Programm das tut, was es tun soll?

# Teststrategien

- **Blackbox-Test:** Für diesen Test ist das interne Verhalten und die interne Struktur des Programms nicht bekannt.
  - *Testen des Ein- / Ausgabeverhaltens*
  - *vollständiges Austesten ist nicht möglich (warum?)*
  - *Wie kann man mit einer endlichen Anzahl von Testfällen maximal viele Fehler finden?*
  - *Wer sollte Blackbox-Tests durchführen?*

# Teststrategien

- **Whitebox-Test:** Definition von Testfällen unter Kenntnis des internen Verhaltens und der internen Struktur des Programms
  - *vollständiges Austesten bei vollständiger Codeüberdeckung?*
  - *Ein Programm kann auch wegen fehlendem Code fehlerhaft sein.*
  - *Codeüberdeckung sagt nichts über datensensible Fehler aus.*
  - *Wieviel Zeit verbringt eine Ausführung in einer Methode?*
  - *Wieviele Objekte wurden von welchem Typ angelegt?*
  - *Wer sollte Whitebox-Tests durchführen?*

# Testprinzipien

(nach Myers)

- Ein notwendiger Bestandteil eines Testfalls ist die Definition des erwarteten Resultats.
- Ein Programmierer sollte nicht versuchen, sein eigenes Programm zu testen.
  - *Eine Programmierorganisation sollte nicht ihre eigenen Programme testen.*
- Überprüfen Sie die Ergebnisse eines jeden Tests gründlich.
- Testfälle müssen für ungültige und unerwartete ebenso wie für gültige und erwartete Ergebnisse definiert werden.

# Testprinzipien

(nach Myers)

- Vermeiden Sie Wegwerftestfälle!
- Planen Sie kein Testverfahren unter der stillschweigenden Annahme, dass keine Fehler gefunden werden.
  - *Die Wahrscheinlichkeit für die Existenz weiterer Fehler ist proportional zur Zahl der bereits gefundenen Fehler.*
- Testen ist eine kreative und intellektuell herausfordernde Aufgabe.
  - *Ein guter Testfall ist dadurch gekennzeichnet, dass er mit hoher Wahrscheinlichkeit einen bisher unbekanntem Fehler zu entdecken imstande ist.*
  - *Ein erfolgreicher Testfall ist dadurch gekennzeichnet, dass er einen bisher unbekanntem Fehler entdeckt.*

# Welche Module sollte man bevorzugt testen?

- **Verschiedene Hypothesen:**
  - *Wenige Module enthalten die Mehrzahl der Fehler.*
  - *Wenige Module erzeugen die meisten Ausfälle.*
  - *Fehlerdichten korrespondierender Phasen sind über Releases hinweg konstant.*
  - *Viele Fehler im Modultest bedeuten viele Fehler im Systemtest.*
  - *Umfangmaße sind zur Fehlerprognose geeignet.*
  - *Viele Fehler im Test bedeuten viele Ausfälle im Feld.*

*(abnehmende Zustimmung verschiedener Autoren)*

(nach Liggesmeyer)

# Testmetriken

Wann können wir aufhören zu testen?

- Testkosten:
  - *Anzahl, Aufwand der Tests*
- Testfälle:
  - *Quantität*
  - *Komplexität: Testdaten, Intensität (Anzahl der Funktionen)*
  - *Qualität: Intensität, Wiederverwendbarkeit (Anzahl der automatisierten Testfälle), Vergleich Ist-Testfälle – Soll-Testfälle*
- Testüberdeckung:
  - *Code, Benutzerhandbuch*
- Testeffektivität:
  - *Rate der gefundenen Fehler zu nicht gefundenen Fehlern*
  - *sinkende Fehlerrate trotz steigender Testaktivität und Codeüberdeckung*

# Software Profiling

- Ein Profiler ist ein Programm, das anderen Programmen bei der Ausführung zuschaut.
- Es sammelt Informationen:
  - *zur Speicherplatzbelegung*
    - wieviele Bytes, wieviele Objekte?
  - *zur Ausführungszeit*
    - absolut, durchschnittlich?
  - *zur Codeüberdeckung*
    - welche Methode von wem aufgerufen?

# Anforderungen an Profiling-Werkzeuge

- Ein Profiling-Werkzeug sollte die folgenden Fragen beantworten können:
- Für eine konkrete Programmausführung:
  - *Welche Methode ist wie häufig aufgerufen worden?*
  - *Wieviel Zeit verbringt das Programm in welcher Methode?*
  - *Welche Methode ruft welche anderen Methoden auf?*
  - *Welche Methoden beanspruchen den meisten Speicherplatz?*
  - *Wieviele Laufzeitobjekte gibt es von welcher Klasse?*
  - *Welche Größe haben die Laufzeitobjekte?*
  - *Welcher Code ist überdeckt worden?*

# Profiling-Werkzeuge

- für Java:
  - *Eclipse Test and Performance Tools Platform Project (TPTP)*
  - *NetBeans Profiler*
  - *JProfiler*
  - ...
  
- für andere Sprachen:
  - *gprof (GNU Profiler für C und C++)*
  - *Visual Studio Team System Profiler (für .NET)*
  - ...

# Zusammenfassung

- Semantische Qualitätsprüfungen:
  - *Testen, Zusicherungen, Verifikation*
- Testen ist eine analytische Qualitätssicherungsmaßnahme.
- Eine größere Softwarekomponente kann nicht vollständig getestet werden.
  - *Testen hilft, Fehler zu finden. Es kann nicht die Abwesenheit von Fehlern zeigen.*
- Testmetriken
  - *Welche Module sollten vorrangig getestet werden?*
  - *Wann haben wir genug getestet?*
- Profiling
  - *Wieviel Zeit und Speicherplatz braucht eine Ausführung?*
  - *Welcher Code ist ausgeführt worden?*

# Testgetriebene Softwareentwicklung mit JUnit

4. Juni 2008

---

# Überblick

- Testgetriebene Softwareentwicklung
  - *Test-First-Ansatz*
- Was ist JUnit?
  - *Allgemeine Struktur von JUnit*
  - *Vor- und Nachbereitung von Testfällen*
  - *Ein Beispieltest*
  - *JUnit und Eclipse*
- Mock-Objekte

# Ein erster Test

- Wie kann man das folgende Programm testen?  
Es werden zwei ganze Zahlen eingelesen, die als Grenzen für eine Zufallszahl gewählt werden. Diese Zufallszahl soll mit Hilfe des Programms erraten werden.
- Beispiel:
  - *Grenzen: 5 und 10*
  - *Zufallszahl: 7*
  - *geratene Zahlen: 9, 6, 7*

# Mögliche Testfälle

1. zu wenige oder zu viele Grenzzahlen
2. untere Grenze größer als obere Grenze
3. untere Grenze kleiner als obere Grenze
4. Zufallszahl zwischen den angegebenen Grenzen
5. Zufallszahl direkt geraten
6. geratene Zahl zu groß oder zu klein
7. geratene Zahl außerhalb der Grenzen

# Testgetriebene Softwareentwicklung

- Test-First-Ansatz:
  - *Erst Denken, dann Programmieren.*
  - *Erst Testen, dann Programmieren.*
- Testen und Programmieren gehen Hand in Hand:  
„Test a little, write a little, test a little, write a little.“
- Vorgehensweise in einem Entwicklungszyklus:
  - *Testfälle schreiben*
  - *Testfälle ausführen: Fehler, da die Funktionalität noch fehlt*
  - *Implementieren*
  - *Testfälle ausführen: Code solange ändern, bis alle Tests durchlaufen*
  - *Refactoring*

# Was ist JUnit?

- JUnit ist ein Test-Framework.
- JUnit bietet Klassen an, um geschriebenen Quelltext leicht zu prüfen.
- JUnit unterstützt automatisiertes Testen, d.h. es verlangt während der Tests keine Benutzerinteraktion.
- JUnit verlangt ein wenig Disziplin.
- JUnit ist einfach anzuwenden.

## **Was ist JUnit nicht?**

- Ein Wundermittel – die Tests schreiben sich nicht von selbst.

# JUnit

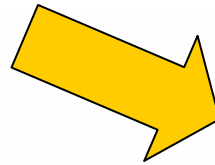
- Prinzipien:
  - *Zu jeder verfassten Klasse eine Testklasse entwerfen.*
  - *Testcode und Anwendungscode sind strikt getrennt.*
- Installation:
  - *Archiv `junit.jar` dem `CLASSPATH` hinzufügen.*
  - *Alle zum Testen notwendigen Klassen sind im Paket `junit.framework` enthalten.*
  - *Eclipse: JUnit 4 einbinden: Properties > Libraries > Add Library*
- Zum Schreiben von Tests werden lediglich benötigt:
  - *`TestCase`, `Assert`, (`TestSuite`)*
- Zum Ausführen der Tests werden benötigt:
  - *`TestRunner`*

# Beispielprogramm

```
public class GuessTheNumber {  
  
    int from = -1;  
    int to = -1;  
    int target = -1;  
    boolean initialized = false;  
    Scanner in = new Scanner(System.in);  
  
    * Initializes the game by a specific range, given by parameter[]  
    public String initializeGame(int range[]) {  
        String resultString = null;  
        if (range.length != 2) {  
            resultString = "Two values need to be entered!";  
            initialized = false;  
        } else {  
            from = range[0];  
            to = range[1];  
            target = from + (int) Math.round((to - from) * Math.random());  
            initialized = true;  
        }  
        return resultString;  
    } // initializeGame  
  
    * @param value[]  
    public String getHint(int value) {  
        String resultString = null;  
        if (this.target > value) {  
            resultString = "Value too small";  
        } else if (this.target < value) {  
            resultString = "Value too high";  
        } else {  
            resultString = "Congratulations! You found the correct number finally!";  
        }  
        return resultString;  
    } // getHint  
}
```

# Beispiel: Testfallklasse

```
public class GuessTheNumberTest {  
  
    private GuessTheNumber game;  
  
    @Before  
    public void setUp() throws Exception {  
        game = new GuessTheNumber();  
    }  
  
    @After  
    public void tearDown() throws Exception {  
    }  
  
    @Test  
    public void testInitializeGame_1() {  
        int range[] = { 1 };  
        String result = game.initializeGame(range);  
        assertEquals(result, "Two values need to be entered!");  
    }  
  
    @Test  
    public void testInitializeGame_2() {  
        int range[] = { 0, 10 };  
        String result = game.initializeGame(range);  
        assertNull(result);  
    }  
}
```



Runs: 6/6    ❌ Errors: 0    ❌ Failures: 0

game.test.GuessTheNumberTest [Runner]

- testInitializeGame\_1
- testInitializeGame\_2
- testInitializeGame\_4
- testGetHint\_1
- testGetHint\_2
- testGetHint\_3

# Beispiel: Fehlgeschlagener Testfall

```
@Test
public void testInitializeGame_3() {
    int range[] = { 10, 0 };
    String result = game.initializeGame(range);
    // Der "from"-Wert ist größer als der "To" Wert, d.h. der Wertebereich
    // ist falsch. Das wird in unserem Beispiel nicht abgefangen,
    // entsprechend liefert dieser Test einen Fehler.
    assertNotNull(result);
}
```

Runs: 7/7    ❌ Errors: 0    ❌ Failures: 1

game.test.GuessTheNumberTest [Runner]

- testInitializeGame\_1
- testInitializeGame\_2
- testInitializeGame\_3
- testInitializeGame\_4
- testGetHint\_1
- testGetHint\_2
- testGetHint\_3

Failure Trace

! java.lang.AssertionError:  
at game.test.GuessTheNumberTest.testInitializeGame\_3(GuessTheNumberTest.java:51)

# Weitere Tests

```
@Test
public void testGetHint_1() {
    testInitializeGame_4();
    int t = game.getTarget();
    t = t + 1;
    String s = game.getHint(t);
    assertEquals(s, "Value too high");
}

@Test
public void testGetHint_2() {
    testInitializeGame_4();
    int t = game.getTarget();
    t = t - 1;
    String s = game.getHint(t);
    assertEquals(s, "Value too small");
}

@Test
public void testGetHint_3() {
    testInitializeGame_4();
    int t = game.getTarget();
    String s = game.getHint(t);
    assertEquals(s,
        "Congratulations! You found the correct number finally!");
}
```

# Aufbauende Tests

```
@Test
public void testInitializeGame_3() {
    int range[] = { 10, 0 };
    String result = game.initializeGame(range);
    // Der "from"-Wert ist größer als der "To" Wert, d.h. der Wertebereich
    // ist falsch. Das wird in unserem Beispiel nicht abgefangen,
    // entsprechend liefert dieser Test einen Fehler.
    assertNotNull(result);
}

@Test
public void testInitializeGame_4() {
    testInitializeGame_2();
    // falls jener Test fehlschlägt, brauch der Aktuelle nicht durchgeführt
    // werden
    int t = game.getTarget();
    assertTrue((t >= 0) && (t <= 10));
}
```

# Definition von Testfällen

- Testfälle werden in einer normalen Klasse erstellen.
  - *Kennzeichnung der Testmethoden durch @Test*
- Framework führt die definierten Testfälle aus.
  - *nur die mit @Test markierten Testfälle*
- Assert für den Vergleich von Soll- mit Istwerten
  - *zu überprüfende Testbedingung*

# Grundsätzlicher Testablauf

- Jeder Test wird gekapselt:
  - *Vor einem Test können die Werte mit `setUp()` initialisiert werden.*
    - `@Before` für Methoden, die vor jedem Test ausgeführt werden sollen.
    - `@BeforeClass` für die einmalige Ausführung von statischen Methoden
  - *mit `runTest()` der Test durchgeführt und*
  - *Mit `tearDown()` kann nach einem Test aufgeräumt werden.*
    - `@After` und `@AfterClass` analog
- Beispiele für den Auf- und Abbau von Tests:
  - *Öffnen und Schließen eines Files*
  - *Öffnen und Schließen von Kommunikationsverbindungen*

# Assert und Fail

- Assert = Behauptungen, die erfüllt sein müssen.
  - **True**: Wahrheit
  - **False**: Falschheit
  - **Null**: Wert gleich Null.
  - **NotNull**: Wert nicht gleich Null.
  - **Same**: Referenz stimmt überein.
  - **NotSame**: Referenz stimmt nicht überein.
  - **Equals**: Ruft `Object.equals` auf.
- Beispiele:
  - `assertTrue(expected.equals(result)) ;`
  - `assertEquals(a,b) ;`
  - `assertEquals("a=b", a, b) ;`
- **AssertionFailedError**, wenn Test fehlschlägt.
- Fail = Test schlägt fehl

# Fehler oder Fehler?

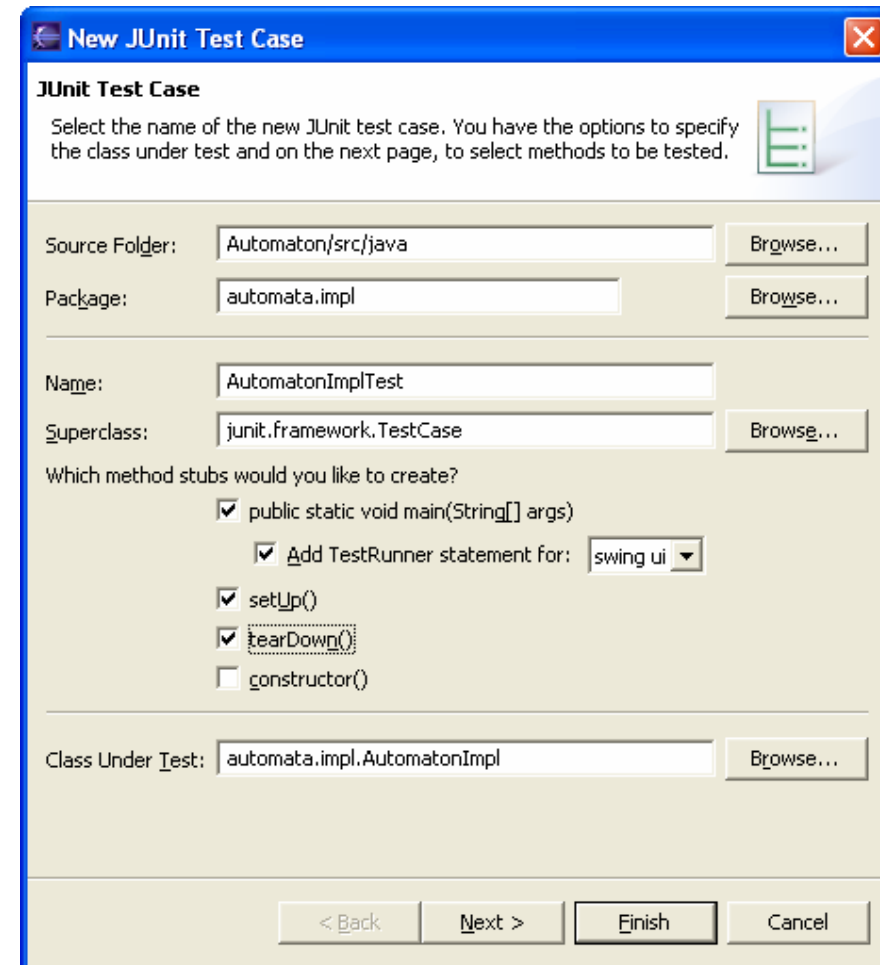
- JUnit unterscheidet zwei Arten von Fehlern:
  - failures: *Fehler, die durch die negative Auswertung einer zuvor gestellten Behauptung entstanden sind.*
  - errors: *Fehler, die unerwartet entstanden sind, wie z.B. eine **ArrayIndexOutOfBoundsException***
- Die Klasse **TestFailure** dient nur zur Speicherung der Fehler im Vector.

# Testsuiten

- Testsuiten dienen dazu, verschiedene Tests in einer bestimmten Reihenfolge aufzurufen.
- Suiten können dazu verwendet werden, verschiedene Klassen eines Paketes bzw. Projektes auf einmal zu testen.
  - *Paketttests, Klassentests, Methodentests*
- Die main-Methode wird dabei in eine neue Klasse verschoben, welche
  - *die Tests der einzelnen Klassen übernimmt.*
- Der **TestRunner** führt den eigentlichen Test durch.
  - *Text basierend*
  - *Swing basierend*

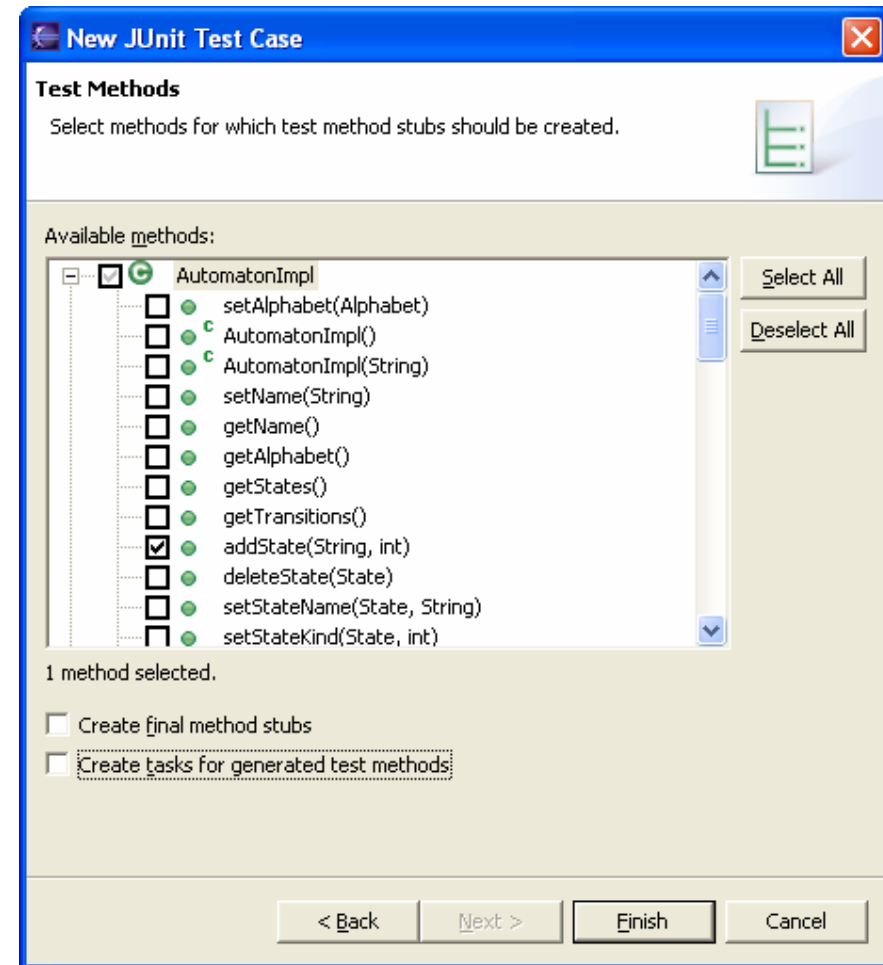
# JUnit Testfälle mit Eclipse anlegen

- Parallele Teststruktur anlegen:
  - *src*
    - java (eigentlichen Sourcen)
    - test (enthält die gleiche Paketstruktur wie java)
  - *classes*
- Zu testende Klasse auswählen:
  - *File* → *new* → *JUnit Test Case*
  - *Name der Testklasse: <Klasse>Test*



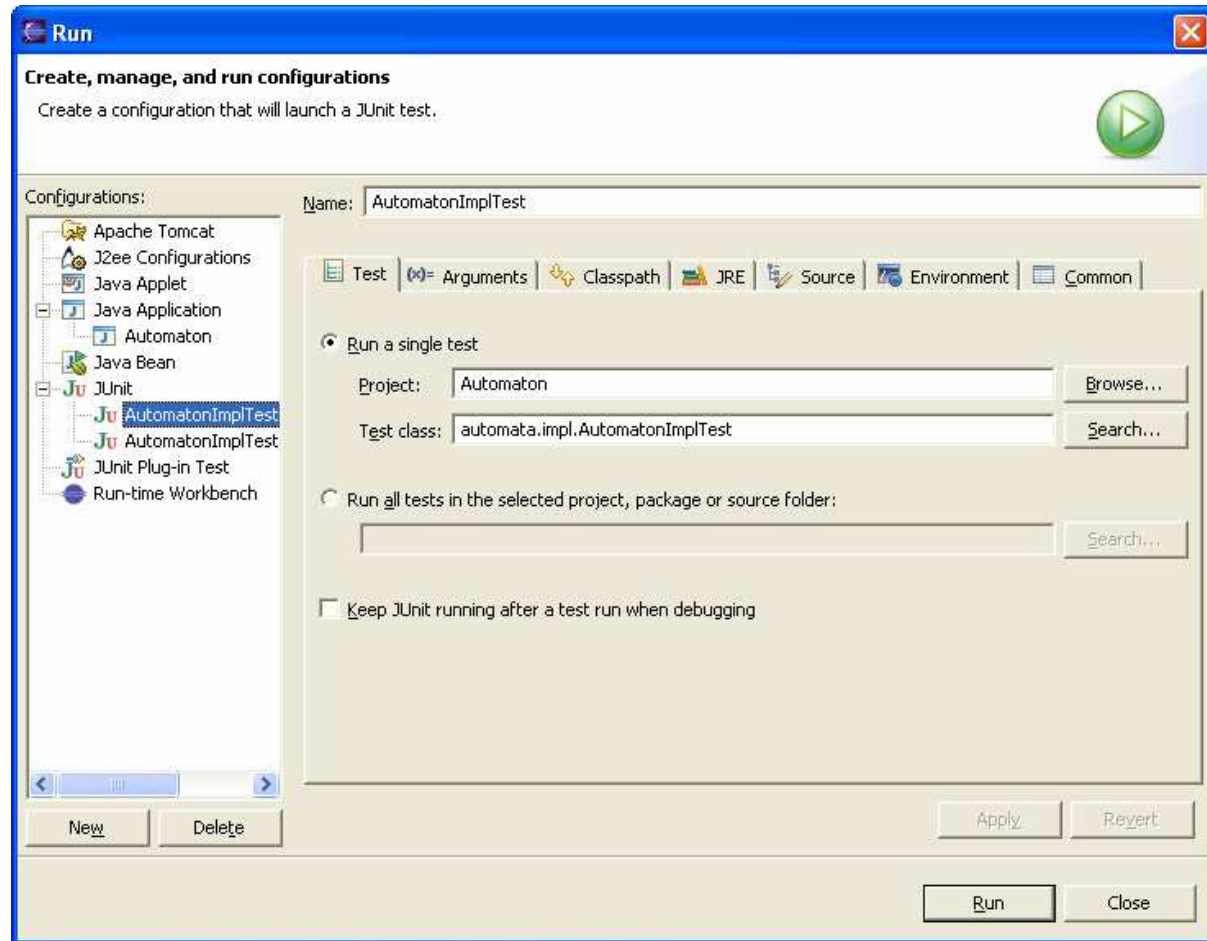
# JUnit Testfälle mit Eclipse anlegen

- Auswahl der zu testenden Methoden



# JUnit Testfälle in Eclipse ausführen

- Durchführen von Tests:
- zu testende Klasse:
- Run → Run As → JUnit Test
- Alternativ auch zu testen:
  - *Paket*
  - *Methode*entsprechend auszuwählen



# Erweiterungen

- Was tun, wenn die zu testende Methode privat ist?
- Testen von Eclipse Plugins
- Was tun, wenn die zu testenden Daten noch nicht zur Verfügung stehen?

# Testen von privaten Methoden

- Da Testmethoden in eigenen Paketen liegen, können private Methoden nicht direkt aufgerufen und getestet werden.
- Lösung:
  - *Benutzung von Reflection-Klassen, um die zu testende Methode vorübergehend öffentlich zu machen*

# Beispiel: Private Methoden testen

```
public class A {  
    public static void main(String[]  
                               args) {  
        System.out.println(getInt());  
    }  
  
    private static Integer getInt() {  
        return new Integer(1);  
    }  
}
```

```
public void testGetInt() {  
    A a = new A();  
    Integer ergebnis= null;  
    try {  
        Class clazz = A.class;  
        Method method = clazz.  
            getDeclaredMethod("getInt",  
                               new Class[]{});  
        method.setAccessible(true);  
        ergebnis = (Integer)  
            method.invoke(a, new Object[]{});  
    } catch (Exception e) {  
        e.printStackTrace();  
        fail();  
    }  
    assertEquals(new Integer(1), ergebnis);  
}
```

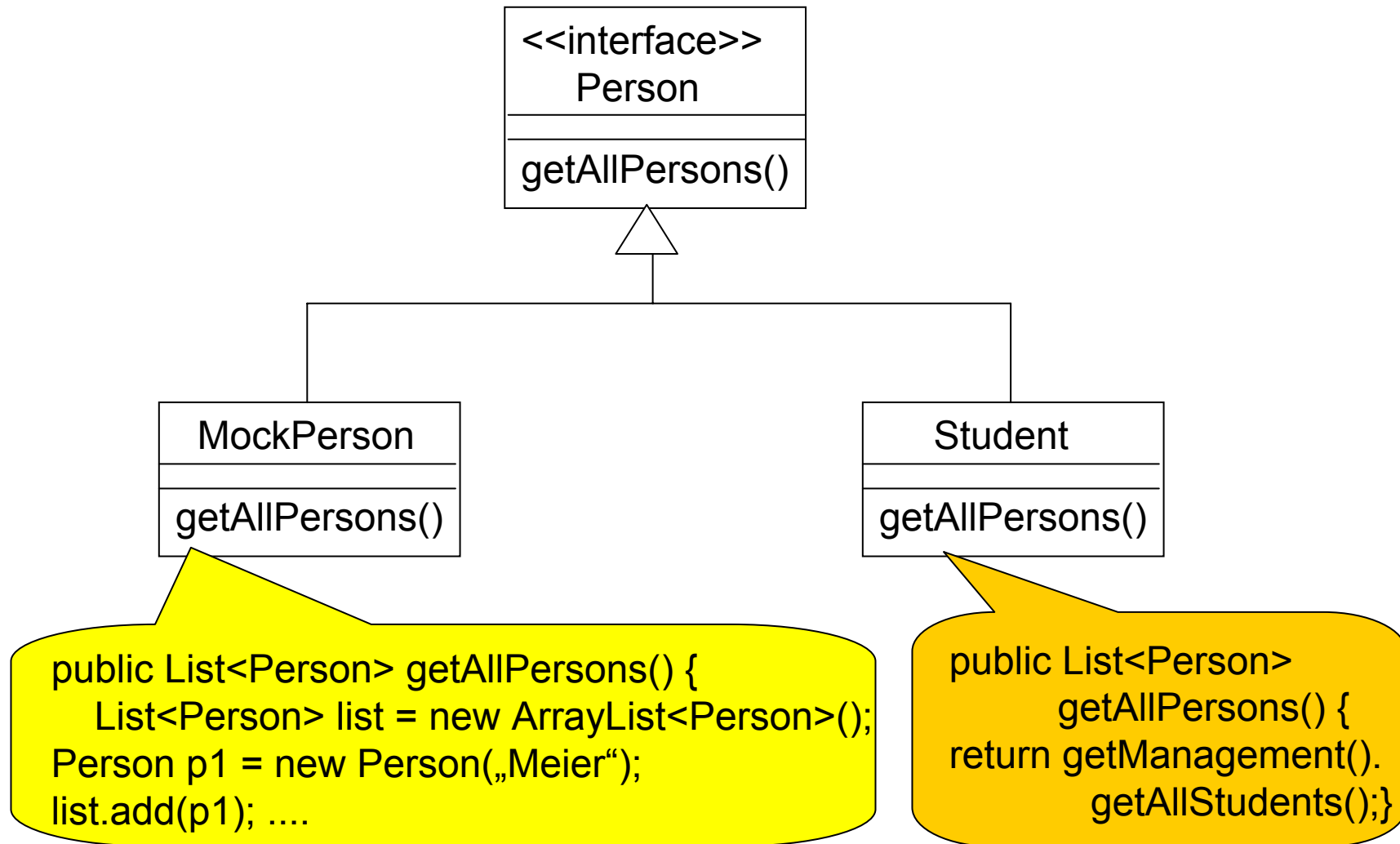
# PDE JUnit

- zum Testen von Eclipse Plug-ins
- Testausführung:
  - *Run As > JUnit Plug-in Test*
  - *Eine neue Eclipse Runtime-Umgebung wird gestartet.*
- Workspace:
  - *Alle Tests werden im neuen Workspace gestartet.*
  - *Für komplexere Tests muß der Workspace vorbereitet werden.*
  - *Im setUp() ein neues TestProjekt starten, das im tearDown() wieder abgebaut wird.*

# Mock-Objekte

- Was tun, wenn bestimmte Ressourcen noch nicht zum Testen zur Verfügung stehen?
  - *Beispiele: Datenbanken, Server, Softwarekomponenten*
  - *Einsatz von Mock-Objekten*
- Mock-Objekte:
  - *Attrappen, die echte Objekte simulieren*
  - *Sie implementieren dieselben Schnittstellen wie die „echten“ Objekte.*
- Mock-Objekte unterstützen den Komponententest. (Warum?)
- weitere Informationen: [www.mockobjects.com](http://www.mockobjects.com)

# Beispiel: Mock-Objekte



# Zusammenfassung

- Automatische Tests unterstützen den Entwickler während der gesamten Entwicklung:
  - *schnelle Wiederverwendung*
  - *vom eigentlichen Code getrennt*
- JUnit
  - *Ein einfach strukturiertes, gut durchdachtes Werkzeug*
  - *Tests verlangen, nachdem sie einmal verfasst wurden, vom Entwickler kein aktives Denken mehr (außer im Fehlerfall).*
  - *JUnit als „defacto“-Standardwerkzeug*
  - *Mehr Informationen: [www.junit.org](http://www.junit.org)*
- Mock-Objekte: Attrappen für echte Objekte

# Testen von graphischen Benutzeroberflächen

11. Juni 2008

---

# Überblick

- Testarten
  - *Methoden-, Klassen-, Komponenten-, Systemtests*
- Motivation für automatisches Testen von graphischen Benutzeroberflächen
- Entwicklungsprinzipien für GUIs
- Capture / Replay – Testmethode
  - *automatisches Testen von GUIs*
  - *realitätsgetreue Simulation von Anwendereingaben*
- Unit-Test für Oberflächen
- Test-Werkzeuge für Oberflächen

# Graphische Oberflächen: Was ist zu testen?

- Funktionalität:
  - *Alle graphischen Komponenten zeigen das spezifizierte Verhalten.*
- graphisches Design:
  - *benutzerfreundlich*
  - *konsistent*
- Verschiedene Benutzer testen:
  - *Benutzer mit verschiedenen Erfahrungen*
  - *Benutzer haben verschiedenen Verwendungszwecke.*
- Verschiedene Plattformen

# Motivation: Automatisches Testen von graphischen Oberflächen

- Testen von graphischen Oberflächen
  - *meist Blackbox-Tests*
  - *Komponenten- oder Systemtests*
- Manuelles Testen von graphischen Oberflächen
  - *zeitraubend und teuer*
  - *arbeitsintensiv*
  - *fehleranfällig, da hoch monoton*
  - *häufig Wegwerftests*
- deshalb: Werkzeugunterstützung für automatisches Testen von graphischen Oberflächen wünschenswert

# Entwicklungsprinzipien für GUIs

- Entwicklung von Anwendungen nach dem Model-View-Controller-Prinzip
- Deshalb bedeutet Testen von GUIs:
  - *kein Testen der Logik*
  - *Testen der korrekten Abbildung von unterliegenden Daten an der Oberfläche*
- Die pixelgenaue Darstellung der Oberfläche soll nicht getestet werden. (Warum?)

# Beispiel: Login Screen Test



jfcUnit – [jfcinit.sourceforge.net](http://jfcinit.sourceforge.net)

- Welche Testfälle sind zu untersuchen?

# Capture / Replay - Testen

- Capture:
  - *Alle Benutzeraktionen während eines Tests werden aufgezeichnet.*
    - ausgewählte Elemente
    - auswählende Aktionen
    - Eigenschaften der ausgewählten Elemente
- Programmierung:
  - *Erzeugen von Templates aus den aufgezeichneten Daten*
- Checkpoints zur Definition von Werten für Funktionalitätstests
- Replay:
  - *Testablauf aufgezeichnet und realistische Werte gesetzt*
  - *beliebige Wiederholung der Tests*

# Probleme mit Oberflächentests

- Erkennen verschobener Objekte
  - *Kommen die Tests mit einer anderen Anordnung der GUI zurecht?*
- richtige Zeitverzögerungen
  - *Ist die Abfolge nicht nur logisch, sondern auch zeitlich richtig?*
  - *Eingaben können beim Abspielen eines Tests schneller oder langsamer bereitgestellt werden.*
- Vorbedingungen für bestimmte Funktionalität
  - *Welche Vorbedingungen für welche GUI-Elemente?*
  - *Sind Vorbedingungen intuitiv erkennbar?*
- andere Einflüsse
  - *Stören keine anderen Programme die Tests?*
  - *Testrechner entspricht dem Anwendungsrechner?*

# Unit-Tests für GUIs

- Unit-Tests sind Whitebox-Tests. (Warum?)
- sehr ähnlich zu normalen Unit-Tests
- spezielle Features:
  - *Hält Referenzen auf Fenster und Dialoge*
  - *Findet graphische Komponenten*
  - *erzeugt Ereignisse (z.B. Mausclick)*
- JUnit Framework für Swing-basierte Oberflächen
  - *jfcUnit (URL: [jfcunit.sourceforge.net](http://jfcunit.sourceforge.net))*

# Beispiel: Set Up für einen Unit-Test

```
import junit.extensions.jfcunit.*;
import junit.extensions.jfcunit.finder.*;
import junit.extensions.jfcunit.eventdata.*;

public LoginScreenTest extends JFCTestCase {
    private LoginScreen loginScreen = null;

    public LoginScreenTest( String name ) {
        super( name ); }

    protected void setUp( ) throws Exception {
```

<http://jfcunit.sourceforge.net/>

```
        super.setUp( );
        setHelper( new JFCTestHelper( ) );
```

# Beispiel: Tear Down für einen Unit-Test

```
protected void tearDown( ) throws Exception {  
    loginScreen = null;  
    getHelper.cleanup( this );  
    super.tearDown( );  
}
```

<http://jfcunit.sourceforge.net/>

# Beispiel: Ein GUI-Testfall

```
public void testUserAndPasswordEmpty() {  
    JDialog dialog;  
    ....  
    NamedComponentFinder finder = new NamedComponentFinder  
        (JComponent.class, "PasswordTextField" );  
    JTextField passwordField = (JTextField) finder.find ( loginScreen, 0);  
    assertNotNull( "Could not find the passwordField", passwordField );  
    assertEquals( "Password field is empty", "",  
        passwordField.getText( ) );  
    ...  
}
```

<http://jfcunit.sourceforge.net/>

# Beispiel: Ein GUI-Testfall

```
...
getHelper().enterClickAndLeave( new MouseEventData( this,
                                                    enterButton ) );
DialogFinder dFinder = new DialogFinder( loginScreen );
showingDialogs = dFinder.findAll();
assertEquals( "Number of dialogs showing is wrong", 1,
                                                    showingDialogs.size( ) );
dialog = ( JDialog )showingDialogs.get( 0 );
assertEquals( "Wrong dialog showing up", "Login Error",
                                                    dialog.getTitle( ) );
getHelper().disposeWindow( dialog, this );
}
```

<http://jfcunit.sourceforge.net/>

# Test-Werkzeuge für Oberflächen

- Capture / Replay – Tests
  - z. B. *GTT : Open Source Framework (Java / Swing)*
- Unit-Tests
  - z. B. *JFCUnit: Open Source Framework für Unit-Test für Java/Swing-Oberflächen*
  - z. B. *GUIDancer (Java / Swing)*
- umfangreiche Liste von GUI Testwerkzeugen:
  - *[http://en.wikipedia.org/wiki/List\\_of\\_GUI\\_testing\\_tools](http://en.wikipedia.org/wiki/List_of_GUI_testing_tools)*
- Demo
  - *GUIDancer (URL: [www.bredex.de/en/guidancer](http://www.bredex.de/en/guidancer))*

# Zusammenfassung und Ausblick

- Systematische GUI-Tests sind zeitaufwendig und monoton, deshalb automatische Tests.
- Zwei Hauptansätze:
  - *Capture / Replay: Abläufe aufzeichnen und wiederverwenden, Varianzprobleme in Layout und Zeit*
  - *Unit Tests: Simulieren von GUI-Aktionen, spezieller Test-Thread*
- Trend: intelligente Capture /Replay Werkzeuge
  - *Behebung der Varianzprobleme*
  - *Simulation von Benutzerverhalten*

# Design-By-Contract

18. Juni 2008



# Überblick

- Defensive Programmierung
  - *Vor- und Nachteile*
- Design-By-Contract: Konzepte
  - *Zusicherungen*
  - *Vor- und Nachbedingungen für Methoden und Invarianten*
  - *Was passiert, wenn ein Vertrag verletzt ist?*
  - *Verträge und Vererbung*
- Design-By-Contract for Java

# Design-By-Contract

- konstruktive Qualitätssicherungsmaßnahme
- Qualitätskriterien:
  - *Zuverlässigkeit: Korrekte und robuste Software*
  - *Wiederverwendbarkeit: Komponenten mit wohldefinierten Eigenschaften*
- von Bertrand Meyer entwickeltes Prinzip zur Entwicklung von zuverlässiger Software
  - *Literatur: Bertrand Meyer: Applying "Design by Contract", in Computer (IEEE), 25, 10, October 1992*  
*<http://se.ethz.ch/~meyer/publications/computer/contract.pdf>*
- in der Programmiersprache Eiffel zentral verankert
  - *mittlerweile auch in anderen Programmiersprachen, z.B. in Java*

# Design-By-Contract

“One should not write a class without a **specification** - a **contract**. The contract lists

- *the internal consistency conditions that the class will maintain (the **invariant**) and,*
- *for each operation, the correctness conditions that are the responsibility of the client (the **precondition**) and*
- *those which the operation promises to establish in return (the **postcondition**).*

Writing a class without its contract would be similar to producing an engineering component (electrical circuit, VLSI chip, bridge, engine...) without a spec. No professional engineer would even consider the idea.”

Bertrand Meyer, Description of Design By Contract

# Defensive Programmierung

- Wie schreibt man zuverlässige Software?
  - *Alle Eventualitäten müssen adäquat behandelt werden.*
- Gängige Praxis:
  - *möglichst viele Checks einbauen („Sicher ist sicher.“)*
  - *auch redundante (Motto: Wenn die Checks nicht helfen, dann schaden sie wenigstens nicht.)*
- Problematik dieses Ansatzes:
  - *produziert viel (redundante) Software*
  - *produziert zusätzliche Komplexität und eventuell Ressourcen-Engpässe*

# Beispiel: Verträge

## Versenden eines Briefes:

### Verpflichtungen:

- Kunde:  
Brief ist nicht schwerer als 20g, DIN C6 groß, Kunde zahlt 0,55 Euro
- Dienstleister:  
Befördert einen Brief innerhalb von einem Tag

### Nutzen:

- Kunde:  
Brief wird innerhalb von einem Tag zum Empfänger befördert.
- Dienstleister:  
Muss zu schwere, zu kleine oder zu große Briefe, die nicht bezahlt sind, nicht befördern

# Design-By-Contract-Prinzipien

- separate Spezifikation der Software
  - *Vor- und Nachbedingungen für Methoden*
  - *Invarianten*
- Formulierung von Bedingungen durch Zusicherungen (Assertions)
  - *eingeführt von Floyd (1967), Hoare*
  - *Liste von Booleschen Ausdrücken*
- Vorbedingungen:
  - *müssen beim Methodenaufruf erfüllt sein*
- Nachbedingungen:
  - *müssen nach Ausführung einer Methode erfüllt sein*

# Design-By-Contract im Softwareentwicklungsprozess

- Während der Softwareentwicklung:
  - *erst Spezifizieren mit Verträgen*
  - *dann Implementieren*
  - *Software mit Verträgen laufen lassen*
    - Ausführung stoppt, sobald eine Bedingung verletzt
    - Spezialbehandlung bei Verletzung der Bedingungen
- Installation beim Kunden:
  - *alle Verträge ausgeschaltet*

# Beispiel: Einfügen eines Baumknotens

Der Einfüge-Vertrag:

- Nutzer der Methode:
  - *Verpflichtung: Referenz „node“ zeigt auf ein Objekt.*
  - *Nutzen: bekommt einen Baum mit „node“ als neues Kind des aktuellen Knotens*
- bereitgestellte Methode:
  - *Verpflichtung: fügt den neuen Knoten als Kind des aktuellen ein*
  - *Nutzen: muss nichts tun, falls „node“ nicht auf ein Objekt zeigt*

```
void putChild(node: Node){  
  // Vorbedingung: node != null  
  // node einfügen  
  // Nachbedingungen:  
  // node.parent == this  
  // this.noOfChildren ==  
  //           this.noOfChildren + 1  
  return;  
}
```

# Verletzung von Zusicherungen

- Verletzung der Vorbedingung:
  - *Fehler beim Aufrufer (Kunden)*
- Verletzung der Nachbedingung:
  - *Fehler in der Methode (Dienstleister)*

Bei Verletzung der Vorbedingung:

- Methode muss nichts unternehmen.
- also z.B. nicht:

```
if (newNode != null) {  
    //.....  
} else  
    //.....  
}
```

- Entweder Vorbedingungen oder Ausnahmebehandlung im Code, nicht beides
- vermeidet redundanten Code

# Nachweis von Zusicherungen

Wer sollte prüfen?

- 2 Programmierstile:
  - *fordernd: starke Vorbedingungen*
  - *tolerant: schwache Vorbedingungen*

Der fordernde Stil ist erfolgreich, da wohldefinierte Aufgaben ausgeführt werden und nicht jeder mögliche Fall abgehandelt wird.

- Muss jeder Nutzer sich wiederholende Prüfungen durchführen?
  - *Nein: Vorbedingungen müssen wiederholt garantiert, aber nicht getestet werden.*
- Falls verschiedene Nutzer dieselben Prüfungen durchführen müssen:
  - *Methode toleranter schreiben (Vorbedingungen abschwächen)*

# Welche Sprache für Assertions?

- Boolesche Ausdrücke mit
  - *Methodenaufrufen*
- Prädikatenlogik 1. Stufe?
  - *nicht ausreichend, Gegenbeispiel:*

```
class AcyclicGraph {  
    // not cyclic  
}
```

*hier wäre Prädikatenlogik 2. Stufe nötig*

# Klasseninvarianten

- Eine Invariante muss nach Erzeugung jedes Objekts der Klasse erfüllt sein.
- Eine Invariante, die beim Aufruf einer öffentlichen Methode der Klasse erfüllt ist, muß auch nach deren Beendigung erfüllt sein.
- Invarianten beschreiben tiefere Eigenschaften einer Klasse.

- Beispiel:

```
class Dictionary {  
    // 0 <= count &&  
    // count <= CAPACITY  
    private int count;  
    private static int CAPACITY  
                                = 10000;  
  
    //...  
}
```

# Beobachtung von Zusicherungen

- keine Zusicherungen aktiv
  - *Zusicherungen haben keinen Effekt auf die Programmausführung.*
- nur Vorbedingungen
  - *Normalfall*
  - *Auswertung der Zusicherung bei Methodeneintritt*
- Vor- und Nachbedingungen
  - *Vorbed. bei Methodeneintritt*
  - *Nachbed. bei Methodenaustritt*
- plus Invarianten
  - *Auswertung der Invarianten bei Methodenein- und -austritt*
- Verletzung einer Zusicherung:
  - *Wurf einer Exception*
- Gründe für Verletzungen:
  - *falsche Strategie → Objekte in einen konsistenten Zustand, neuer Versuch*
  - *organisierte Panik → Objekte in einen konsistenten Zustand, Fehlermeldung*
  - *falscher Alarm → eventuell kleinere Korrekturen, Programmausführung fortsetzen*

# Dokumentation durch Verträge

- Verträge definieren einen Standard für die Dokumentation von Klassen.
- Erstellung einer Kurzform der Klassen
  - *Klassendeklaration*
  - *Methodenköpfe*
  - *Javadoc*
  - *Vertragsinformationen*

# Verträge und Vererbung

- Beispiel:

```
class OrderedBinaryTree extends BinaryTree {  
  //...  
  public void put_child(Node newNode) {  
    //...  
  }  
}
```

hat eventuell eine  
andere Semantik

- Unterverträge für eine Unterklasse:

- *Vorbedingungen dürfen nicht verstärkt werden.*
- *Nachbedingungen dürfen nicht abgeschwächt werden.*
- *Beispiel: neue Nachbedingung: Vorgängerknoten ist kleiner und Nachfolgerknoten ist größer.*

# Zusammenfassung

- Die Spezifikation von Invarianten, Vor- und Nachbedingungen ist eine grundlegende Qualitätssicherungsmaßnahme.
- Ähnliches Vorgehen wie bei testgetriebener Entwicklung
  - *Spezifizieren*
  - *Implementieren*
  - *Prüfen*
- Realisierung der Design-by-Contract-Idee in Java
  - *Zusicherungen (Assertions) in Java 5*
  - *Contract4J: <http://www.contract4j.org/>*
  - *jContractor: <http://jcontractor.sourceforge.net>*
  - *JMSAssert: <http://www.mmsindia.com/JMSAssert.html>*

# Formale Verifikation von Software

25. Juni 2008



# Überblick

- Wann ist formale Softwareverifikation sinnvoll?
- Welche Techniken gibt es?
- Was ist Model Checking und wie kann man es zur Verifikation einsetzen?
- Welche Probleme gibt es bei der formalen Verifikation von Software?

# Berühmte Software-Fehler

- Airbusabsturz Toulouse, 1994:
  - *Software-Fehlverhalten ist mitverantwortlich für den Absturz eines A340 bei Toulouse, 7 Tote*
- Pentium-Bug, Intel, 1994:
  - *spezielle Division verursachen Fehler und Kosten von knapp 500 Mio Dollar*
  - *seitdem: Einsatz von formaler Verifikation bei Intel*
- Deutsche Telekom, 1996:
  - *durch Spezifikations- und Softwarefehlern werden in 550 Vermittlungsstellen falsche Gebühren berechnet. Dadurch Kosten von ca. 70 Mio DM (geschätzt).*

# Validation und Verifikation

- Validation: „Bauen wir das richtige Softwareprodukt?“
  - *Wird die Software das tun, was die Nutzer fordern?*
- Verifikation: „Bauen wir das Softwareprodukt richtig?“
  - *Wird die Software der Spezifikation entsprechen?*
- Welche Qualitätskriterien werden jeweils bedient?

# Software-Verifikation

- Qualitätskriterien:
  - *Zuverlässigkeit: Korrekte und robuste Software*
  - *Wiederverwendbarkeit: Komponenten mit wohldefinierten Eigenschaften*
- analysierende Qualitätssicherungsmaßnahme
- Software-Verifikation
  - *Ziel: Zusicherung der spezifizierten Anforderung*
  - *Ansätze:*
    - dynamische Verifikation: Tests
    - statische Verifikation: syntaktische Verfahren wie Metriken, **formale Verifikation (semantisch)**

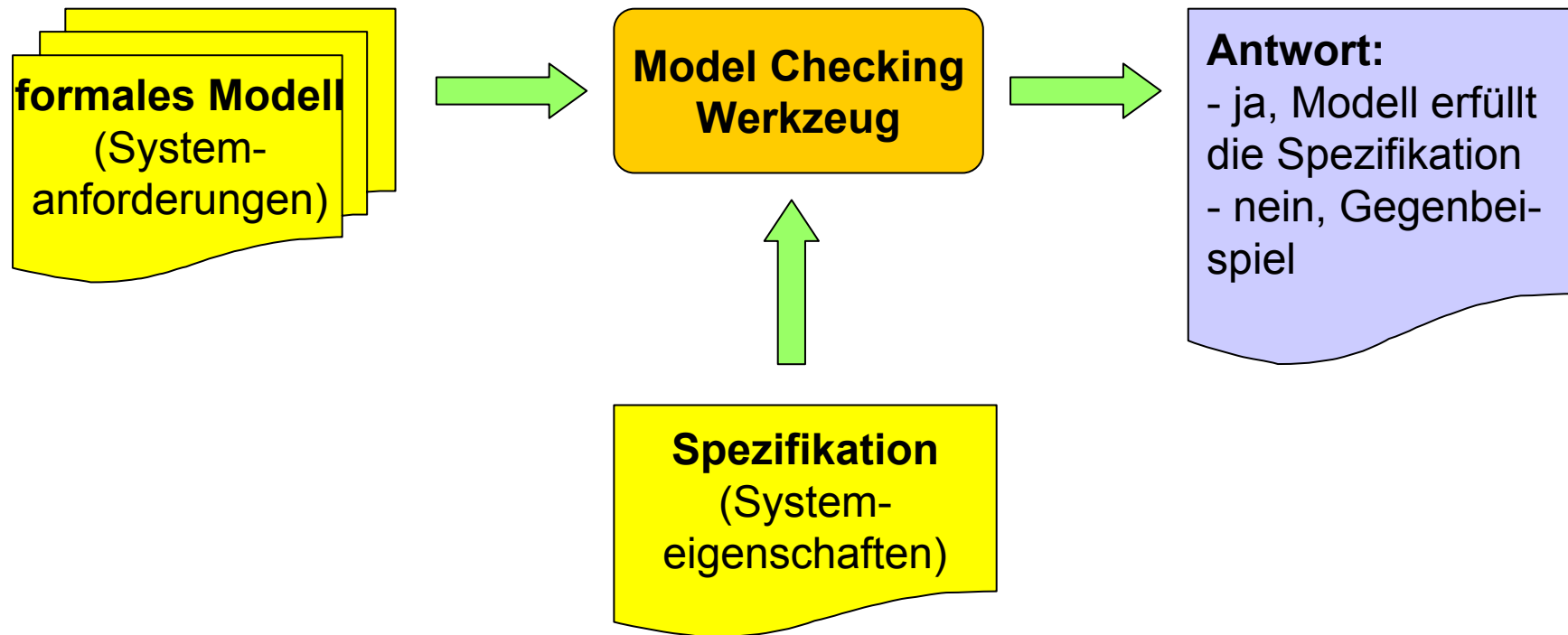
# Formale Software-Verifikation

- Korrektheitsnachweis eines Algorithmus in Bezug auf eine formale Anforderungsspezifikation
- Ansätze zur formalen Verifikation:
  - *Model Checking*
  - *Theorembeweiser*
- Formale Spezifikationsmethoden:
  - *für die Softwaresysteme:*
    - Automaten: endl. Zustandsautomaten, Transitionssysteme, Automaten mit Zeit, etc.
    - Prozessalgebren
  - *für die zu zeigenden Eigenschaften:*
    - Logiken: temporale Logiken

# Beispielanwendungen für formale Verifikation

- Übertragungsprotokolle in Kommunikationssoftware
- eingebettete Systeme z.B. im Automobil, Bahn- und Luftfahrtbereich
  - *Bremssysteme*
  - *Crash-Kontrolle für Airbags*
- digitale Schaltkreise
- Systemsoftware
  - *Compiler*

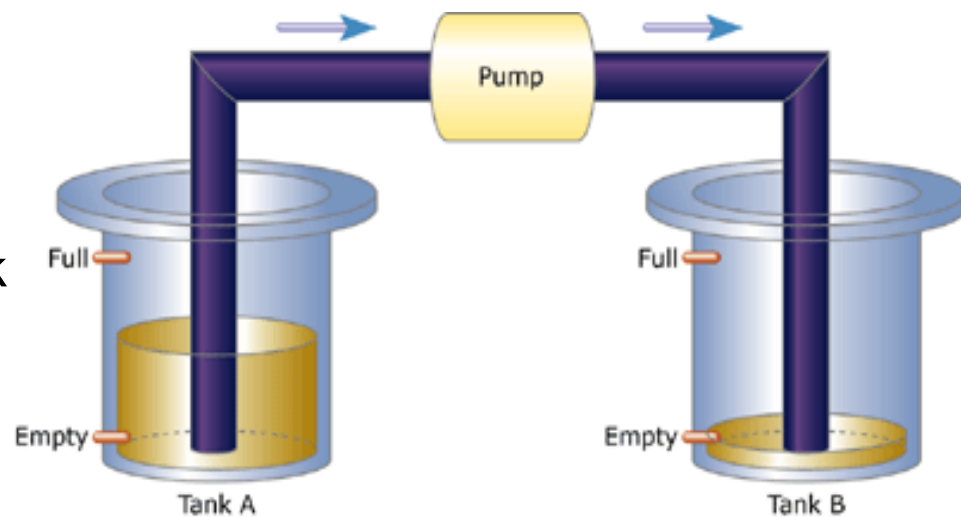
# Model Checking



# Beispiel für Model Checking

- Jeder Tank hat zwei Anzeigen:
  - *leer*
  - *voll*
- Tank okay, falls zwischen leer und voll
  1. beide Tanks leer
  2. Pumpe an, wenn ein Tank A nicht leer
  3. Pumpe aus, wenn Tank A leer oder B voll
  4. Pumpe nicht ausschalten, wenn aus

Einfaches Pumpsystem mit zwei Tanks



www.embedded.com

# Beispiel: Formales Modell

**MODULE main**

**VAR**

ta : {empty, ok, full};

tb : {empty, ok, full};

p : {on, off};

**ASSIGN**

next(ta) := case

ta = empty : {empty, ok};

ta = ok & p = off : {ok, full};

ta = ok & p = on : {ok, empty,  
full};

ta = full & p = off : full;

ta = full & p = on : {ok, full};

1 : {ok, empty, full};

esac;

ta - Tank A

next(tb) := case

tb = empty & p = off : empty;

tb = empty & p = on : {empty, ok};

tb = ok & p = off : {ok, empty};

tb = ok & p = on : {ok, empty, full};

tb = full & p = off : {ok, full};

tb = full & p = on : {ok, full};

1 : {ok, empty, full};

esac;

next(p) := case

p = off & (ta = ok | ta = full) &  
(tb = empty | tb = ok) : on;

p = on & (ta = empty | tb = full) : off;

1 : p; -- keep pump status as it is

esac;

**INIT**

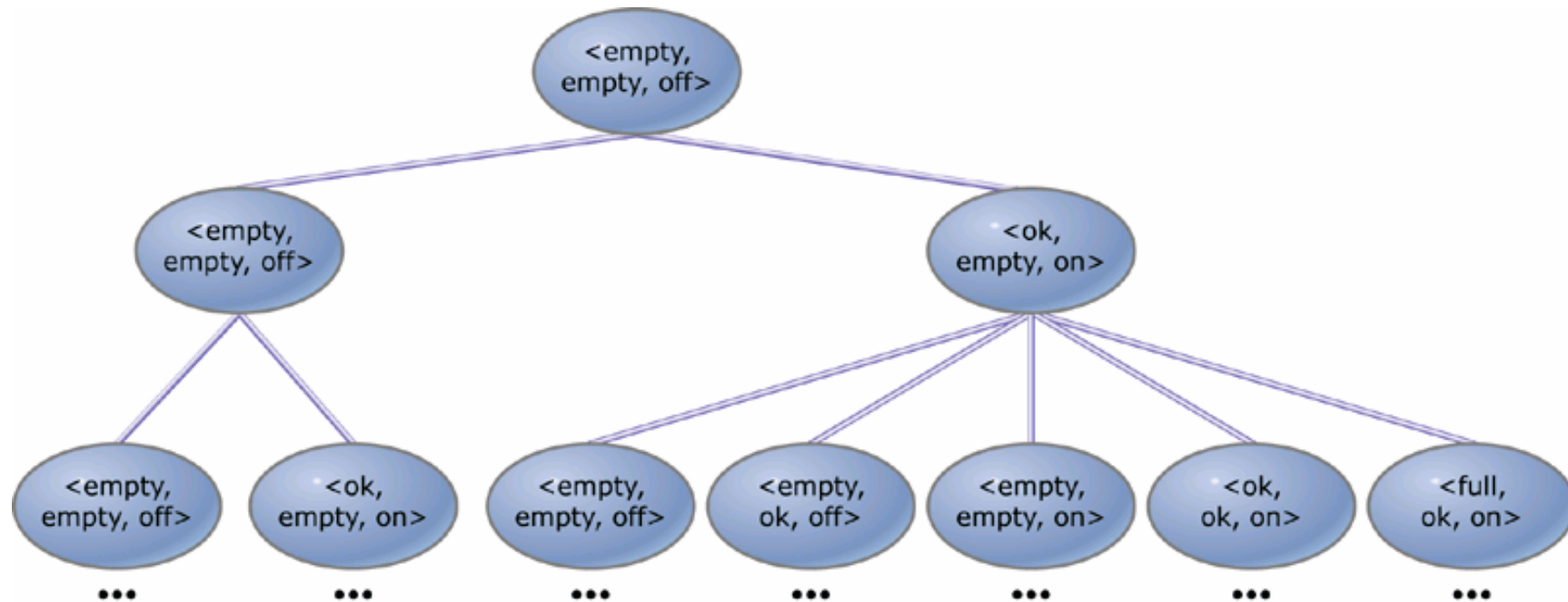
(p = off)

tb - Tank b

p - Pumpe

www.embedded.com

# Beispiel: Zustandsraum



www.embedded.com

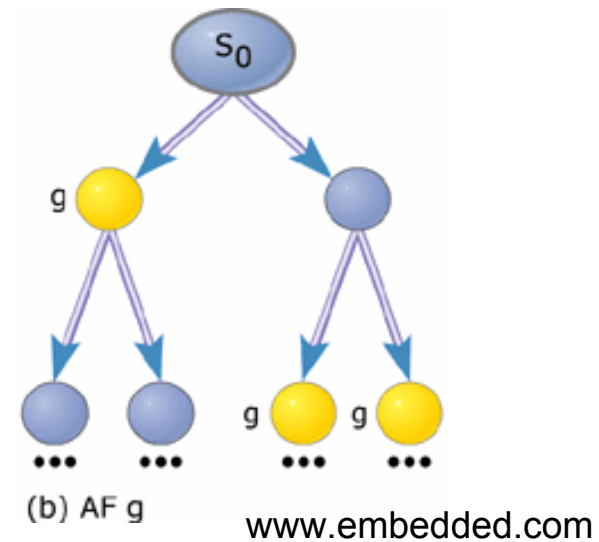
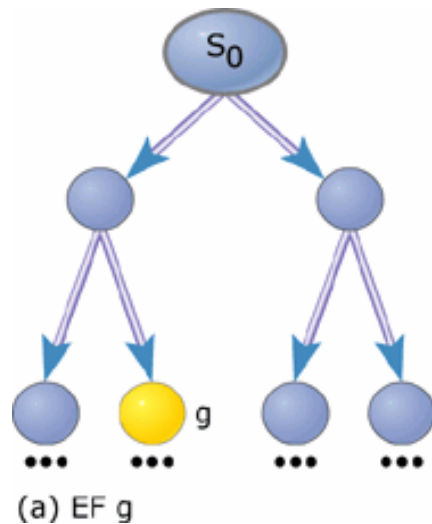
Anfang des Zustandsraums für ein einfaches Pumpsystem

# Beispiel: Systemeigenschaften

- Informelle Beschreibung:
  - *Tank A ist leer oder Tank B ist voll, wenn die Pumpe aus ist.*
  - *Tank B kann jederzeit okay oder voll werden.*
- Spezifikation:
  - *mit Computational Tree Logic (CTL), einer speziellen temporalen Logik*
  - **SPEC**
    - $AG AF (p = off \rightarrow (ta = empty \mid tb = full))$*
    - $AG (EF (tb = ok \mid tb = full))$*

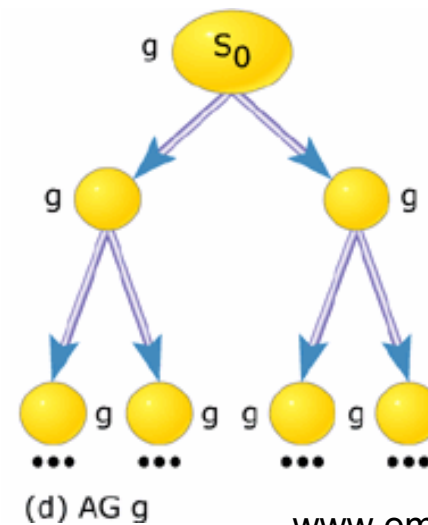
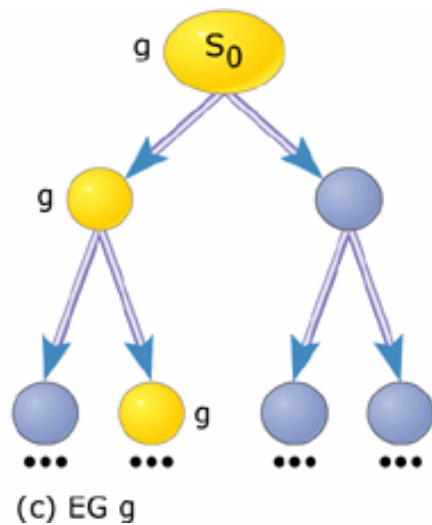
# Temporale Operatoren in CTL

- $EF \varphi$  wahr, falls es in einem Pfad einen Zustand gibt, der  $\varphi$  erfüllt.
- $AF \varphi$  wahr, falls es in jedem Pfad (der bei  $S_0$  startet), einen Zustand gibt, der  $\varphi$  erfüllt.



# Temporale Operatoren in CTL

- EG  $\varphi$  wahr, falls es einen Pfad (bei  $S_0$  startend) gibt, so dass jeder Zustand in diesem Pfad  $\varphi$  erfüllt.
- AG  $\varphi$  wahr, falls jeder Zustand in jedem Pfad (der bei  $S_0$  startet)  $\varphi$  erfüllt.



www.embedded.com

# Beispiel: Nachweis von Systemeigenschaften

- SPEC AF ( $p = \text{on}$ )
  - *Egal, wie sich das System verhält, die Pumpe ist irgendwann mal an.*
  - *Gilt diese Eigenschaft?*
- Ausgabe des Model Checkers SMV:
  - ***-- specification AF p = on is false***  
***-- as demonstrated by the following execution sequence***  
***-- loop starts here***  
***state 1.1:***  
***ta = empty***  
***tb = empty***  
***p = off***
- Spec AF ( $p = \text{off}$ )
  - *Diese Eigenschaft ist erfüllt.*

# Praktische Anwendung von Model Checking

- Modellierung des Systems:
  - *Jedes Model-Checking-Werkzeug hat seine eigene Sprache.*
  - *Beziehung: Informelle Anforderungen – formales Modell unklar*
  - *Nicht alle Anforderungen sind modellierbar*
- Spezifikation der Systemeigenschaften:
  - *ähnliche Probleme, manche Eigenschaften sind nicht spezifizierbar*
- Zustandsraum des modellierten Systems:
  - *Anzahl der Zustände kann extrem hoch sein.*
  - *Model Checker braucht lange oder gibt auf.*
  - *Reduzierung des Zustandsraums nötig*

# Wann lohnt sich formale Verifikation?

- spezielle Anwendungen:
  - *sicherheitskritische Systeme*
  - *andere kritische Anforderungen (z.B. finanzielle und rechtliche)*
- Verbesserung der Anforderungsspezifikation:
  - *präzise und klare Spezifikation erstellbar*
- Reduzierung von Testaufwand:
  - *Vermeidung von Testfällen*
  - *automatische Testfallgenerierung*

# Grenzen der formalen Methoden

- Mögliche Gründe für Fehler:
  - *Software ist nicht korrekt (d.h. erfüllt nicht die Spezifikation):*
    - Formale Verifikation kann diese Fehler finden.
  - *Software ist nicht angemessen (d.h. die Spezifikation ist fehlerhaft):*
    - Formale Spezifikation/Verifikation kann diese Fehler vermeiden/finden.
  - *Betriebssystem, Compiler oder Hardware sind fehlerhaft:*
    - kann im Normalfall nicht vermieden werden
- keine volle Spezifikation/Verifikation
  - *Softwaresystem ist zu komplex:*
    - Einschränkung auf wichtige Komponenten/Eigenschaften

# Zusammenfassung

- Anwendung von formaler Verifikation ist aufwändig
- Formale Verifikation lohnt sich speziell für sicherheits- und kostenkritische Systeme.
- Formale Verifikation wird meist mit Model Checking durchgeführt.
- Literatur:
  - *W. Reif: Software-Verifikation und ihre Anwendungen, it+ti Themenheft, Oldenbourg Verlag, 1997*  
[http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/1997-sw\\_verif\\_anwend/](http://www.informatik.uni-augsburg.de/lehrstuehle/swt/se/publications/1997-sw_verif_anwend/)
  - *W. Reif: Formale Methoden für sicherheitskritische Software – Der KIV-Ansatz, Informatik Forschung & Entw., Vol. 14, No 4, 1999*  
<http://www.springerlink.com/content/vget375dd50vf740/>
  - *G.K. Palshikar: An Introduction to Model Checking*  
<http://www.embedded.com/showArticle.jhtml?articleID=17603352>

# Softwarequalität: Zusammenfassung und Ausblick

2. Juli 2008

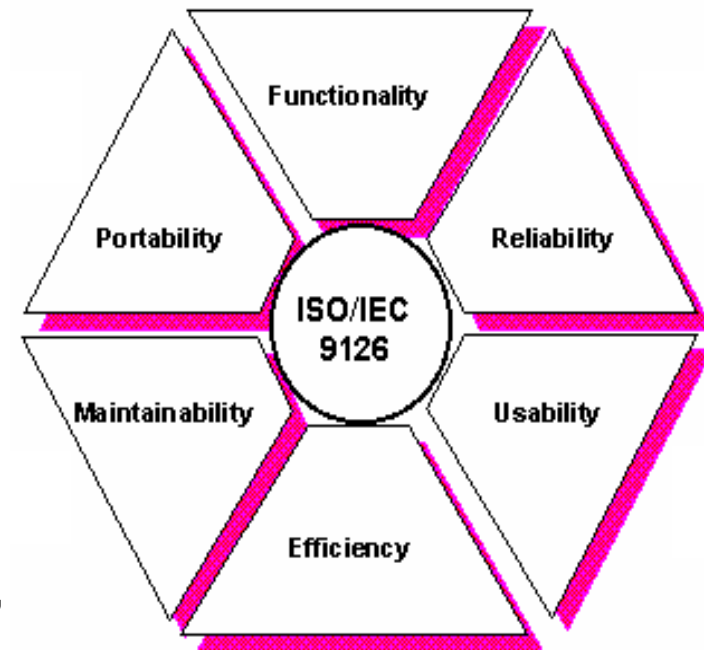
---

# Überblick

- Rückblick:
  - *Qualitätskriterien*
  - *Qualitätsmanagement*
  - *Qualitätssicherungsmaßnahmen*
- Thesen zur Softwarequalität
- Ausblick:
  - *Lehrveranstaltungen im WS 2007/08*
  - *Fortgeschrittenenpraktika / Diplomarbeiten*

# Qualitätsmerkmale für Software

- **Funktionalität:** Korrektheit, Angemessenheit, Interoperabilität, Ordnungsmäßigkeit, Sicherheit
- **Zuverlässigkeit:** Reife, Fehlertoleranz, Wiederherstellbarkeit
- **Benutzbarkeit:** Verständlichkeit, Bedienbarkeit, Erlernbarkeit, Robustheit
- **Effizienz:** Wirtschaftlichkeit, Zeitverhalten, Verbrauchsverhalten
- **Wartungsfreundlichkeit:** Analysierbarkeit, Änderbarkeit, Stabilität, Testbarkeit
- **Übertragbarkeit:** Anpassbarkeit, Installierbarkeit, Konformität, Austauschbarkeit



# Prinzipien der SW-Qualitätssicherung

- produkt- und prozeßabhängige Qualitätszielbestimmung
  - *Welche Qualitätskriterien haben Priorität?*
- quantitative Qualitätssicherung
  - *Metriken, Testfälle*
- frühzeitige Fehlerentdeckung und –behebung
  - *präzise Anforderungserhebung, Spezifikation*
- maximal konstruktive Qualitätssicherung
- entwicklungsbegleitende, integrierte Qualitätssicherung
- unabhängige Qualitätssicherung
  - *Welche QS-Maßnahmen sollten nicht vom Entwickler durchgeführt werden?*

# Qualitätssicherungsmaßnahmen

- **konstruktive Maßnahmen:**  
Methoden, Sprachen, Werkzeuge, Richtlinien, Standards und Checklisten, die eine bestimmte Produkt-oder Prozeßqualität garantieren
- **analytische Maßnahmen:**  
Das existierende Qualitätsniveau wird gemessen. Ausmaß und Ort des Defekts können identifiziert werden.
- *Eine vorausschauende, konstruktive Qualitätslenkung erspart viele analytische Maßnahmen.*

# Qualitätsmodell GQM

## Goal-Question-Metric-Ansatz (GQM):

- Goal: definiere die Auswertungsziele
- Question: leite Fragen zur Quantifizierung ab
- Metrics: leite Maße zur Beantwortung der Fragen ab
- entwerfe einen Mechanismus zur Meßwernerfassung
- validiere die Meßwerte
- interpretiere die Meßwerte

# Verfahren zur Qualitätsmessung

- quantitative Messungen:
  - *Softwaremetriken*
- Überprüfung syntaktischer Muster:
  - *Entwicklungsrichtlinien*
  - *Bad Code Smell*
  - *Entwurfsmuster und Softwarearchitekturen*
- Beispiele und Gegenbeispiele:
  - *Testverfahren und Profiling*
- Überprüfung semantischer Eigenschaften:
  - *Design-By-Contract, Verifikation*

# Thesen zur Softwarequalität

- „Die Softwarequalität steigt, wenn eine Firma A das Fachkonzept erstellt und eine andere Firma B die Software dazu entwickelt.“

*Nicht unbedingt:*

- *Es muss eine phasenübergreifende Begriffsklärung vorhanden sein.*
- *Mindestens ein Entwickler / begleitender Nutzer sollte das ganze Projekt hindurch beteiligt sein.*
- *Einzelne Phasen sollten durch unabhängige Gutachter geprüft werden.*

# Thesen zur Softwarequalität

- Am Ende eines Tages sollte man sagen können:  
„Mein Programm ist heute ein bisschen besser als  
gestern.“
  - *Verbesserung des Modells*
  - *Verbesserung des Codes*
  - *Verbesserung des Entwicklungsprozesses*
  - *Verbesserung der Testfälle*
  - *Verbesserung des Teams*

# Thesen zur Softwarequalität

- „Wer etwas macht, macht Fehler.“  
Software sollte auch ausgeliefert werden, wenn sie Bugs enthält.
  - *manche Fehler sind akzeptabel (welche?)*
  - *Besser ein Produkt veröffentlichen, dessen Fehler bekannt, aber harmlos sind, als ein vermeidlich „fehlerfreies“ Produkt.*
  - *Das Risiko neue, schlimmere Fehler einzubauen beim Versuch bekannte Fehler zu entfernen, ist hoch.*
  - *Wenn man jeden bekannten Fehler vorher entfernen möchte, kann die Entwicklung zu lange dauern.*

# Thesen zur Softwarequalität

- „Open-Source-Software hat eine höhere Qualität als proprietäre Software.“
  - *Da die Open-Source-Software einsehbar ist, kann auch ihre innere Qualität beurteilt werden.*
  - *Proprietäre Softwarehersteller neigen zur Verwendung von eigenen Standards. Für Open-Source ist Kompatibilität und Interoperabilität wichtig.*
  - *Fehler können von Kunden gefunden und behoben werden (funktioniert nur bei bestimmten Kunden (welchen?))*

# Thesen zur Softwarequalität

- „Extreme Programming ist „Hacking-As-Usual“.
  - *extreme Intensität, extreme Identifikation mit den Werten*
  - *extreme Disziplin bei der Softwareentwicklung*
  - *testgetriebener Ansatz*
  - *„Aufräumen“ nach jedem Entwicklungsschritt (Refactoring)*

# Lehrveranstaltungen im WS 08/09

- Modellgetriebene Softwareentwicklung
  - *6 SWS, 9 Kreditpunkte*
- Seminar zu einem spezifischen Thema in der Softwarequalität
  - *2 SWS, 3 Kreditpunkte*

# Themen für Praktika, Bachelor- und Diplomarbeiten

- Syntaktische Qualität von Software-Modellen
  - *Modellmetriken*
  - *Bad Model Smell*
  - *Modell-Refactoring*
  
  - *neue Refactoring-Methoden für UML-Modelle (mit Identifizierung von Bad Model Smell)*
  - *Implementierung von Modell-Refactorings und Bad Model Smells*
    - für UML2-Modelle
    - für andere Modelle (enthält die Definition einer Modellierungssprache)