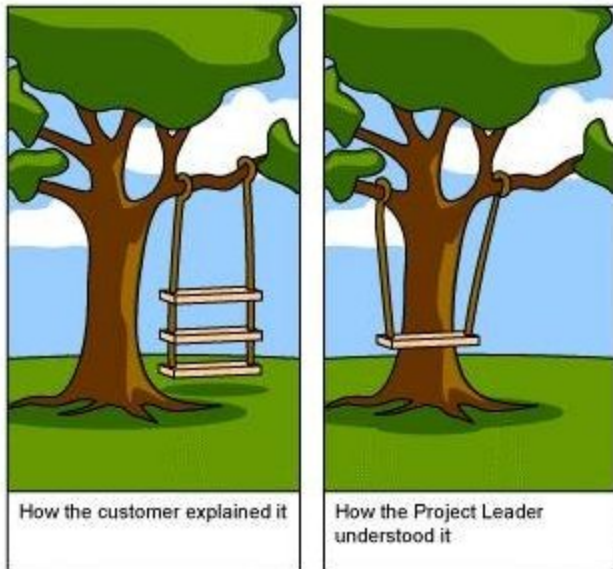


Software Engineering 1

Einleitung



Prof. Dr. Andreas Judt

Software Engineering – Merkmale

- Software Engineering (SE) umfasst Methoden, Techniken und Instrumente zur Erstellung von auf Maschinen lauffähigen Programmen (Software).
- Ziel von SE ist die Vermeidung von Fehlern, Reduktion von Aufwänden und Erhöhung des Kundennutzens von Software..

Software Engineering – Merkmale

- SE beinhaltet das Management von Entwicklungsprojekten für Software.
- SE umfasst die Inbetriebnahme, den Betrieb und die Wartung von Software.
- SE beschreibt bewährte Vorgehensmodelle und definiert Qualitätsstufen für den Software-Entwicklungsprozess

Definition: Software Engineering

- The application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software; that is, the application of engineering to software.
- The study of approaches as in 1.

IEEE-Standard 610-1990

IEEE = Institute of Electrical and Electronics Engineers, Inc.

Software Engineering != Softwaretechnik!

- Softwaretechnik:
 - Modelle, Techniken, Sprachen und Werkzeuge zur Entwicklung von Softwaresystemen
- Software Engineering:
 - Ingenieurmäßige Methodik und Vorgehensweise bei der Entwicklung von Softwaresystemen

Warum ist Softwareentwicklung schwierig?

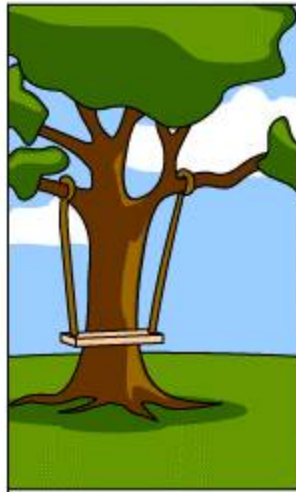
- Die Planung des Projektablaufs ist schwierig.
- Es gibt mehr als einen Lösungsansatz.
 - Oft gibt es nur eine Lösungsvorstellung.
 - Welcher Ansatz ist technisch machbar?
- Mehrere Varianten von Hard- und Software sind geeignet.
 - Welche davon halten ihre Versprechen?

Warum ist Softwareentwicklung schwierig?

- Es treten unvorhergesehene technische Probleme auf.
- Verwendete Technologien werden während der Entwicklungsphase abgelöst.
- Die Software soll auch auf zukünftigen Plattformen funktionieren.



How the customer explained it



How the Project Leader understood it



How the Analyst designed it



How the Programmer wrote it



How the Business Consultant described it



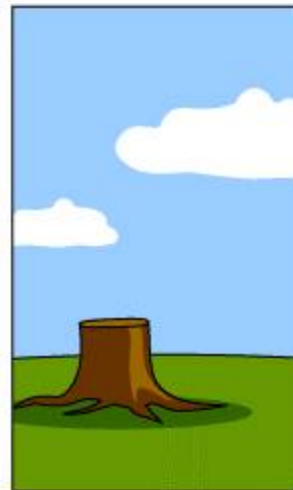
How the project was documented



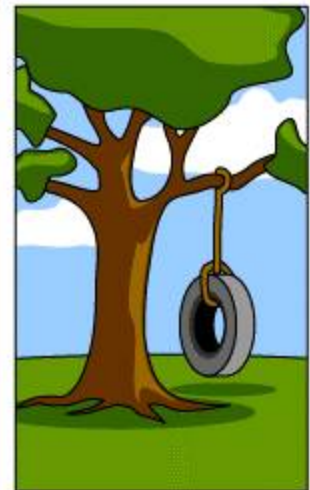
What operations installed



How the customer was billed



How it was supported



What the customer really needed

Phasen in der Softwareentwicklung

- Planung
- Definition
- Entwurf
- Implementierung
- Test
- Übergang in den Betrieb
 - Abnahme und Einführung
 - Wartung und Pflege

Vorgehensmodelle (Auszug)

- völlig unkoordiniert
- Wasserfall
- V-Modell XT
- Synchronisieren und Stabilisieren

Rollenverteilung in der Softwareentwicklung (Auszug)

- Analyst
 - Anforderungen, Budget, Termine, Änderungen
- Architekt
 - Modellierung
- Programmierer
 - Implementierung in einer konkreten Technologie
- Tester
 - Überprüfen der Anforderungen

Typische Anfängerfehler in Software-Projekten

- zuerst die Hardware beschaffen
- zu Beginn des Projekts eine Technologieentscheidung treffen
- Programmieraufwände nicht schätzen
- zu Projektbeginn Fertigstellungstermine zusagen
- wegen Zeitmangels auf Testen verzichten
- Dokumentation am Ende des Projekts erstellen

Management-Fehler



www.dilbert.com
scottadams@aol.com



9-3-07 © 2007 Scott Adams, Inc./Dist. by UFS, Inc.



© Scott Adams, Inc./Dist. by UFS, Inc.

Softwarequalität - Fragestellungen

- Wie kann man Software vergleichen?
- Wie wird Software gemessen?
- Welche Architektur ist besser?
- Erfüllt eine Software die gestellten Anforderungen?
- Wie kann man Software verbessern?
- Wann ist eine Lösung optimal?

Beliebte Irrtümer

- Mit Model Driven Architecture (MDA) kann aus einem Modell eine vollständige Software generiert werden.
 - Beispiel: UML ermöglicht die Erzeugung von Klassen und Rümpfen.
- Die Fehlerfreiheit von Software kann bewiesen werden.
- Meine Lösung ist die beste.

Selbstkontrolle

1. Was unterscheidet Software Engineering von Softwaretechnik?
2. Nennen Sie die Phasen der Softwareentwicklung.
3. Warum ist Softwareentwicklung schwierig?
4. Warum kann die Projektlaufzeit nicht beliebig durch den Einsatz von Personal verkürzt werden?
5. Wo sehen Sie Grenzen bei der Straffung von Projekten?

Kontakt

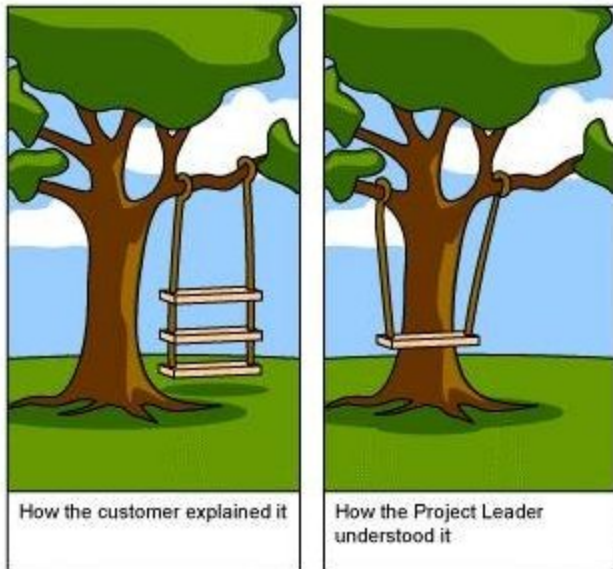
Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Software Engineering 1

Planung



Prof. Dr. Andreas Judt

Aufgaben in der Planungsphase

- Vorgehensweise festlegen
 - auch Vorgehensmodell oder Prozessmodell
- Analyse und Beschreibung der Zielsetzung
 - Lastenheft und Glossar
 - Machbarkeitsstudien (auch Durchführbarkeits-Untersuchungen bzw. Voruntersuchungen)
- Entwicklungsaufwand schätzen
 - Zeitplanung
 - Anzahl der Mitarbeiter
 - Aufgabenverteilung

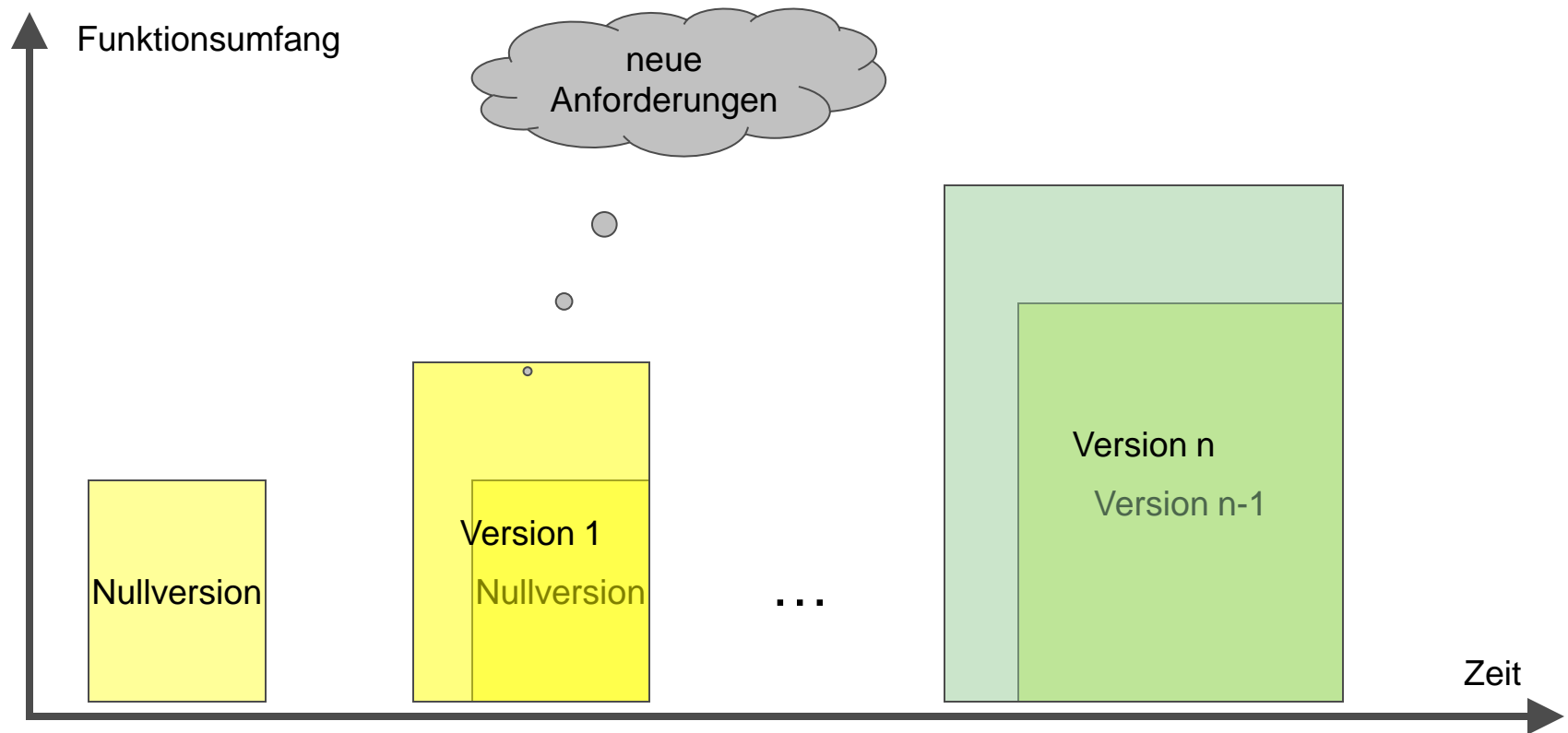
Vorgehensweise / Prozess-Modell

- Legt den organisatorischen Rahmen fest.
 - Welche Aktivitäten sollen in welcher Reihenfolge von welchen Mitarbeitern bearbeitet werden?
- Definiert die Ergebnisse aller Teilschritte.
 - auch Artefakte (engl. artifacts)
- Aktivitäten werden von Mitarbeitern ausgeführt, die jeweils eine oder mehrere Rollen einnehmen.
 - Beachtung von Methoden, Richtlinien, Konventionen, Checklisten und Mustern
- Typische Prozess-Modelle:
 - Wasserfall-Modell, V-Modell (XT), Synchronisieren und Stabilisieren

Iterative Vorgehensweise: evolutionär

- Ausgangspunkt: Kern- und Muss-Anforderungen des Auftraggebers
 - wird als Nullversion des Produkts entwickelt
- Auftraggeber verfeinert seine Anforderungen aus den Erfahrungen mit der Nullversion.
 - Entwicklung einer Produktversion
- Guter Ansatz, wenn Auftraggeber noch nicht alle Anforderungen kennt.
 - Eventuell ist die Nullversion zu schwach: Nicht alle Kern- und Muss-Anforderungen wurde umgesetzt!
 - Zwischen den Versionen kann eine vollständige Überarbeitung möglich sein!

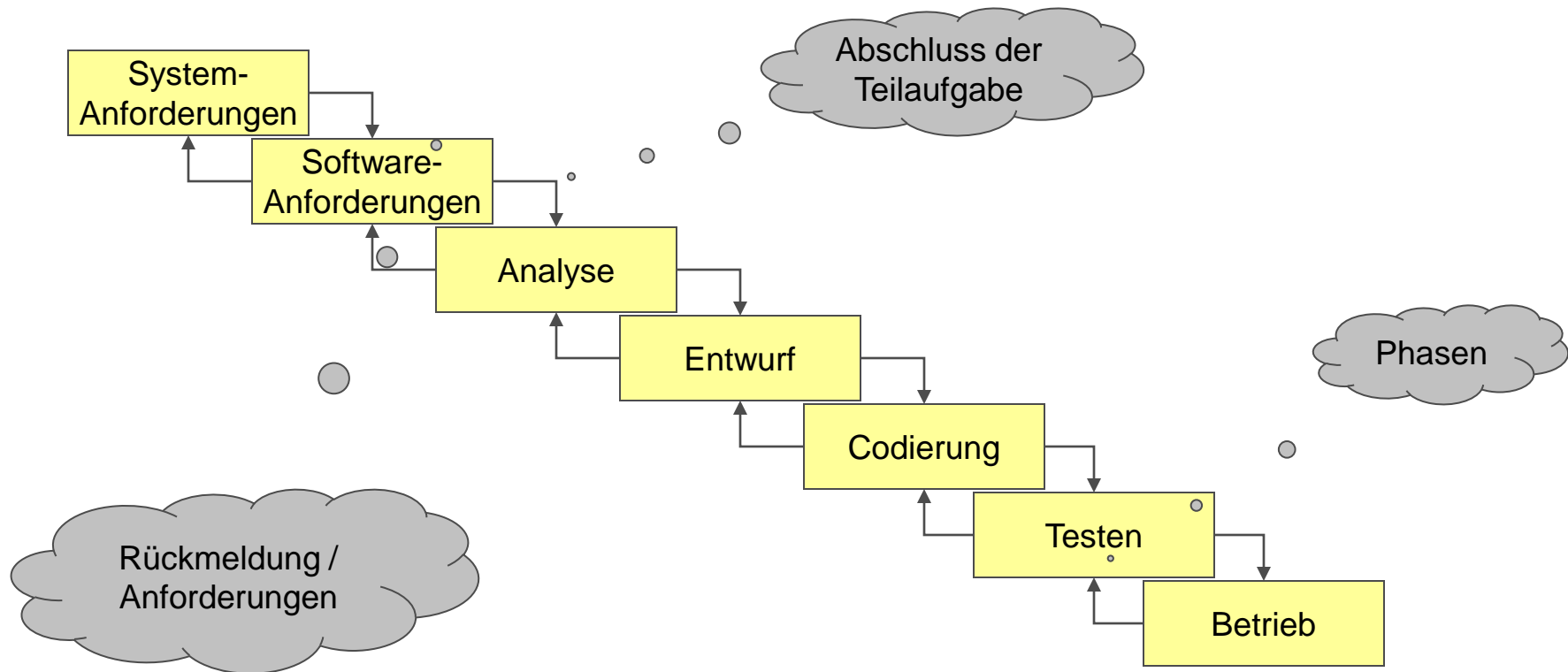
Iterative Vorgehensweise: evolutionär



Iterative Vorgehensweise: inkrementell

- Für die Nullversion sind die vollständigen Anforderungen bekannt.
 - Nur ein Teil davon wird für die Version umgesetzt.
- Erfahrungen des Auftraggebers fließen in die nächste Version ein.
 - Kenntnis aller Anforderungen verhindert hohe Überarbeitungsaufwände.
 - Inkrementelle Erweiterungen lassen sich leicht einbauen.

Vorgehensmodell: Wasserfall



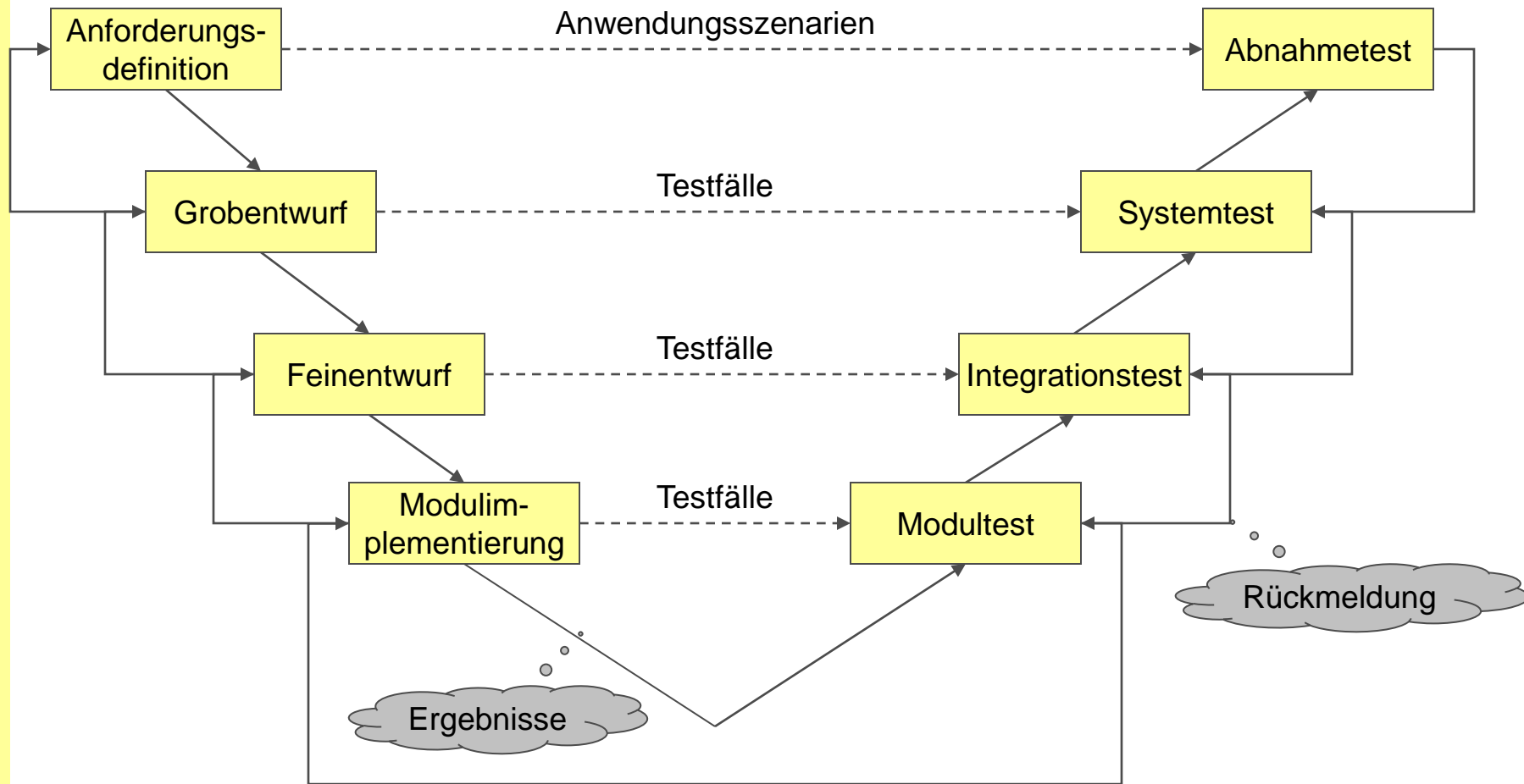
Vorgehensmodell: Wasserfall

- Jede Phase muss vollständig abgeschlossen sein, bevor die nächste Phase beginnt.
- Ergebnisse einer Phase fallen immer nur in die nächste Phase.
 - Rückkopplung und Anforderungen nur in die angrenzende, übergeordnete Phase.

Vorgehensmodell: Wasserfall

- Wasserfall ist dokumentgetrieben
 - Am Ende jeder Phase steht ein Ergebnisdokument.
- Bewertung: Weit verbreitete Vorgehensweise mit klaren Nachteilen.
 - Trennung in Phasen nicht immer sinnvoll
 - Sequenzielle Ausführung von Phase nicht immer sinnvoll
 - Das eigentlich zu entwickelnde Softwaresystem könnte vernachlässigt werden.
 - Risikofaktoren werden evtl. zu wenig berücksichtigt.

Vorgehensmodell: V



Vorgehensmodell: V

- Erweiterung des Wasserfall-Modells um Qualitätssicherung
 - V-Modell 92 später V-Modell 97
 - für Bundeswehr, anschließend für Behörden entwickelt („Vorgehensmodell“)
 - eigentlich für eingebettete Systeme
- Legt alle Aktivitäten und Ergebnisse im Entwicklungs- und Pflegeprozess fest.
- besitzt 4 Teilmodelle:
 - Systemerstellung
 - Qualitätssicherung
 - Konfigurationsmanagement
 - Projektmanagement

Vorgehensmodell: V

- Bewertung: Vorteile
 - sehr gute Darstellung der Vorgehensweise
 - Möglichkeiten zum Maßschneidern des Modells auf Projektanforderungen
 - bildet einen Quasi-Standard für die Entwicklung von Softwaresystemen
 - geeignet für große Projekte

Vorgehensmodell: V

- Bewertung: Nachteile
 - Bürokratischer Aufwand bei kleineren Projekten unrealistisch hoch
 - ohne CASE-Werkzeug (Computer Aided Software Engineering) kaum verwendbar
 - zu viele Rollen in den Teilmodellen

Vorgehensmodell: V XT

- 2005 veröffentlicht, XT = eXtensible Tailoring
 - sehr stark anpassbar an Projektanforderungen
- Anpassung an neue Methoden der Software-Entwicklung
 - z.B. Extreme Programming und Agile Software-Entwicklung
- Erweiterung des V-Modell 97
 - Dokumentation wurde vereinfacht
 - Unterscheidung in Auftragnehmer und Auftraggeber
- wird als neuer Entwicklungsstandard propagiert
 - speziell bei öffentlichen Auftraggebern

Übung: V-Modell XT

- Beschreiben Sie das V-Modell XT. Recherchieren Sie dafür in der gängigen Literatur bzw. im Internet.
- Teilen Sie den Kurs dafür in Gruppen mit maximal 3 Studenten.
 - Verteilen Sie die inhaltliche Arbeit auf die Gruppen.
- Jede Gruppe präsentiert ihre Ergebnisse in einem Vortrag (5-10 Minuten)
- Informationen zum V-Modell XT finden Sie u.a. unter:
<http://v-modell.iabg.de>

Vorgehensmodell: Rational Unified Process (RUP)

- etabliert von Rational, seit 2003 zu IBM gehörig
- objektorientiertes Vorgehensmodell
 - anwendungsfall-basiert (use case)
 - Menge der Anforderungen beschreibt das zu entwickelnde Softwaresystem
 - architektur-zentriert
 - Modellierung erfolgt mit UML
 - iterativ und inkrementell
 - Entwicklungsprozess wird in Zyklen iteriert.
 - Jeder Zyklus inkrementiert den Funktionsumfang.
 - Zyklen werden so lange ausgeführt, bis alle Anwendungsfälle entwickelt wurden.

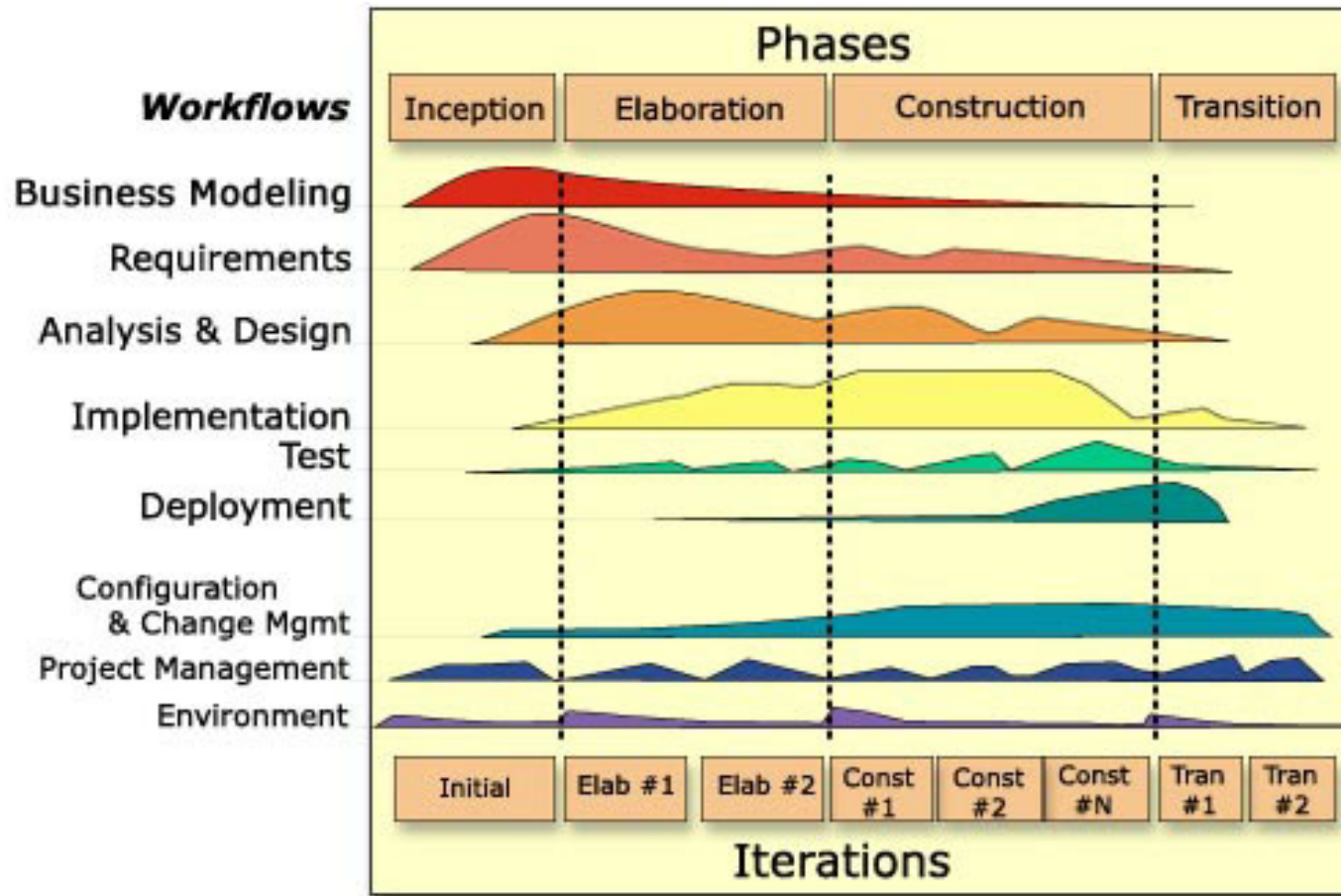
Vorgehensmodell: Rational Unified Process (RUP)

- Phasen eines Zyklus:
 - Einführung (engl. inception)
 - Anwendungsfälle auswählen, Systemarchitektur entwerfen, Risikoanalyse, Präsentation beim Kunden
 - Ergebnis: Zieldefinition
 - Entwurf (engl. elaboration)
 - Werkzeuge und Ressourcen festlegen, Hauptrisiken identifizieren, Prototyp der Architektur entwickeln
 - Ergebnis: Architekturbeschreibung
 - Konstruktion bzw. Entwicklung (engl. construction)
 - Implementierung und Test, Behandlung der Hauptrisiken
 - Ergebnis: lauffähige Software
 - Übergang (engl. transition)
 - Übergabe der Software an den Kunden, Dokumentation, Schulung, Datenübernahme aus Altsystemen
 - Ergebnis: Projektabschluss

Vorgehensmodell: Rational Unified Process (RUP)

- Umsetzung von Projekterfahrungen (engl. best practices) in folgende Schritte:
 - Geschäftsprozessmodellierung (engl. business modelling)
 - Anforderungsdefinition (engl. requirements)
 - Analyse und Design (engl. analysis and design)
 - Implementierung (engl. implementation)
 - Test (engl. test)
 - Betrieb (engl. deployment)
- begleitende Schritte:
 - Konfigurations- und Änderungs-Management (engl. configuration and change management)
 - Projektmanagement (engl. project management)
 - Umwelt (engl. environment)

Vorgehensmodell: Rational Unified Process (RUP)



Vorgehensmodell: Rational Unified Process (RUP)

- Bewertung: Vorteile:
 - Risiken werden bereits früh erkannt und behandelt.
 - Projekterfahrungen sichern die Praxistauglichkeit.
 - Weiterentwicklungen durch neue Erfahrungen.
 - UML ist heute weit verbreitet und kann als Standard für die Modellbildung angesehen werden.
- Bewertung: Nachteile:
 - Phasen und Zwischenziele (Meilensteine) sind schwer zu definieren.

Vorgehensmodell: Synchronisieren und Stabilisieren

- Vorgehensmodell von Microsoft
 - Synchronize and stabilize
- Vision: Software im Internetzeitalter entwickeln
 - Marktentwicklung schwer abzuschätzen
 - kurze Produktlebenszyklen
 - hoher Konkurrenzdruck
 - unbekannte Kundenwünsche
 - schnelle Veränderungen
- Ziel: große Flexibilität und hohe Geschwindigkeit bei professionellem Vorgehen
 - Mitarbeiter haben sehr viel Autonomie, müssen aber ihre Ergebnisse täglich synchronisieren.

Vorgehensmodell: Synchronisieren und Stabilisieren

- Eigenschaften
 - Zwischenstände (engl. builds) werden täglich erzeugt
 - 3-4 Zwischenziele (Meilensteine)
 - Frühe Alpha- und Betaversionen
 - Kleine Teams arbeiten parallel
 - Anforderungen werden kontinuierlich ermittelt
 - Keine vollständige Anforderungsbeschreibung zu Beginn des Projekts
 - Vision, was entwickelt werden soll
 - Spezifikation des Funktionsumfangs erst am Ende des Projekts
 - Stabilisierung bei Erreichung des Zwischenziels

Vorgehensmodell: Synchronisieren und Stabilisieren

- Entwicklungszyklus:
 - Entwurf
 - Implementierung
 - Test auf Verwendbarkeit
 - Funktionaler Test
 - Fehlerbehebung
 - Integration in Gesamtsystem
 - Stabilisierung

Anforderungen formulieren: Lastenheft

- Ergebnis der Planungsphase
 - ergänzt um Begriffsdefinitionen (Glossar)
- Definiert alle fundamentalen Eigenschaften des Produkts
 - auch als Basisanforderungen bezeichnet
 - auf hohem Abstraktionsniveau
- Lastenheft wird auch als grobes Pflichtenheft bezeichnet.

Gliederung des Lastenhefts

- Zielbestimmung
 - Ziele, die erreicht werden sollen
- Produkteinsatz
 - Anwendungsbereiche und Zielgruppen
- Produktübersicht
 - grafischer Überblick, z.B. Umweltdiagramm
- Produktfunktionen
 - typische Arbeitsabläufe, Akteure, Geschäftsprozesse, Schnittstellen, Datenflüsse
- Produktdaten
 - langfristig zu speichernde Daten mit Schätzung des Umfangs (Mengengerüst)
- Produktleistungen
 - Leistungsanforderungen mit Berücksichtigung von 5.
- Qualitätsanforderungen
 - Qualitätsstandards, z.B. Zuverlässigkeit, Benutzbarkeit, Effizienz
- Ergänzungen
 - spezielle Anforderungen

Schätzung von Aufwänden

- Warum Aufwände schätzen?
 - zu erwartende Kosten bei Eigenentwicklung
 - erforderliche Zeit und Ressourcen
 - Kostenvergleich mit Standardsoftware
 - Planung von Teilschritten und Terminen
- Ziel: Bestimmung vor der Umsetzung:
 - Komplexität
 - Aufwand
 - Zeit

Definition: Programmzeile

- Beschreibt einen Anteil geistigen Eigentums
- Source Line of Code (SLOC)
 - soll eigentlich eine Programmanweisung darstellen
- Programmzeilen ohne geistiges Eigentum:
 - Leerzeilen
 - reine Kommentarzeilen
 - triviale Programmzeilen, z.B. Zeilen, die nur Klammern enthalten
 - Anweisungen für den Debugger
- SLOC sind unterschiedlich komplex, z.B.
 - `x+=1;`
 - `for (int i=0;i<x;i++) { v[i] = f(x*i,n); }`

Schätzmethode: COCOMO 2

- Constructive Cost Model, Schätzverfahren auf der Basis von SLOC, 1995
 - Erweiterung des COCOMO Modells von 1981 auf objektorientierte Entwicklung
 - basiert auf SLOC und Parametern über den geschätzten Projektverlauf, die Qualität des Entwicklerteams und des Projektumfelds
- Schätzmodelle:
 - Application Composition: stark werkzeuggestützte Entwicklung
 - Early Design: in Analyse- oder Prototyp-Stadium
 - Post Architecture: nach Fertigstellung der Architektur
 - einigermaßen empirisch gesichert

Schätzmethode: COCOMO 2

- Basiert auf empirischen Ergebnissen:
 - durchschnittliche Entwickler liefern 350 SLOC pro Mannmonat (MM)
 - Entwicklungen mit hohem Wiederverwendungsanteil benötigen etwa 25% der Aufwände einer Neuentwicklung
- Durch die Parametereinstellungen ist die Varianz der Schätzung enorm hoch
 - Produktivität weicht in der Praxis enorm von 350 SLOC/MM ab, z.B. bei sicherheitskritischen Anwendungen der Luft- und Raumfahrt: 10 SLOC/MM

Schätzmethode: Funktionspunkt-Analyse (FP)

- Idee der Methode: Aufwand hängt von Umfang und Schwierigkeit ab.
 - Umfang wird nicht aus LOC, sondern aus den Anforderungen ermittelt
- Anforderungen werden kategorisiert:
 - Eingabedaten
 - Ausgabedaten
 - Abfragen
 - Datenbestände
 - Referenzdateien

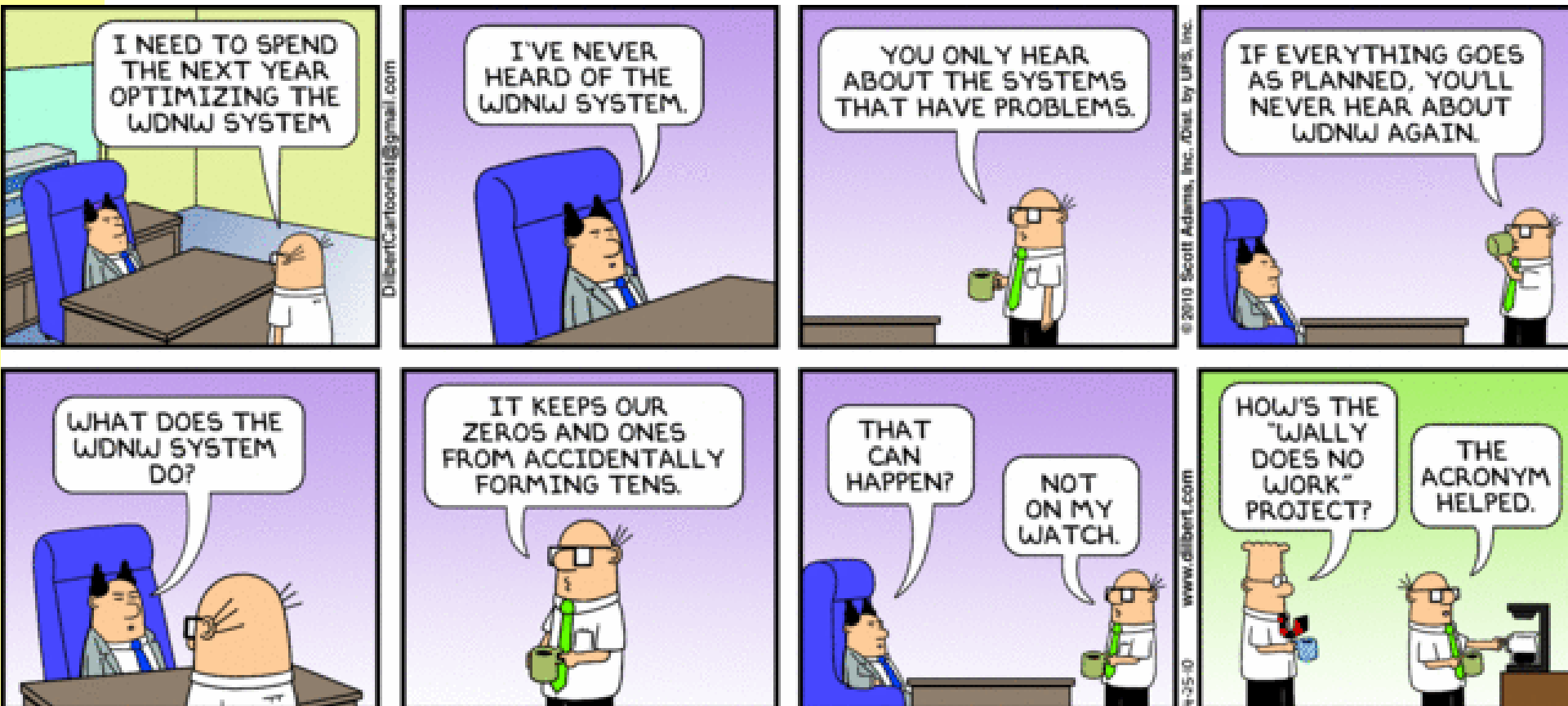
Schätzmethode: Funktionspunkt-Analyse (FP)

- Jede Anforderung wird mittels Richtlinien klassifiziert:
 - einfach
 - mittel
 - komplex
- FP werden durch Gewichtungsfaktoren für Kategorie und Klassifizierung berechnet.
- Umrechnung von FP in Mannmonate (MM) erfolgt durch unternehmensspezifische Abbildung
 - z.B. 50 FP \Leftrightarrow 5 MM, 2000 FP \Leftrightarrow 175 MM
- allgemein akzeptiert: 1 FP entspricht etwa 30 SLOC Java und 29 SLOC C++

Schätzmethode: Funktionspunkt-Analyse (FP)

- **Bewertung: Vorteile**
 - Anforderungen leichter zu ermitteln als SLOC
 - sehr gut an Projektumfeld anpassbar
 - Schätzung kann im Projektfortschritt iterativ verfeinert werden
 - einfach umsetzbar
- **Bewertung: Nachteile**
 - Nur Gesamtaufwand schätzbar
 - keine Berücksichtigung von Qualitätsanforderungen
 - Bei lückenhaften Pflichtenheft wird der Aufwand erheblich unterschätzt!

Zeitplanung mal anders



Selbstkontrolle

1. Welche Aufgabe hat die Planungsphase?
2. Wann ist es sinnvoll, nach dem Wasserfallmodell zu entwickeln?
3. Was ist der Unterschied zwischen V-Modell und V-Modell XT?
4. Vergleichen Sie die Schätzmethode Funktionspunktanalyse und COCOMO2.
5. Wie kann man mit COCOMO2 die Projektkosten deutlich unterschätzen?
6. Begründen Sie, warum die Projektkosten bei der Schätzung mit Funktionspunkten typischerweise zu gering ausfallen.

Kontakt

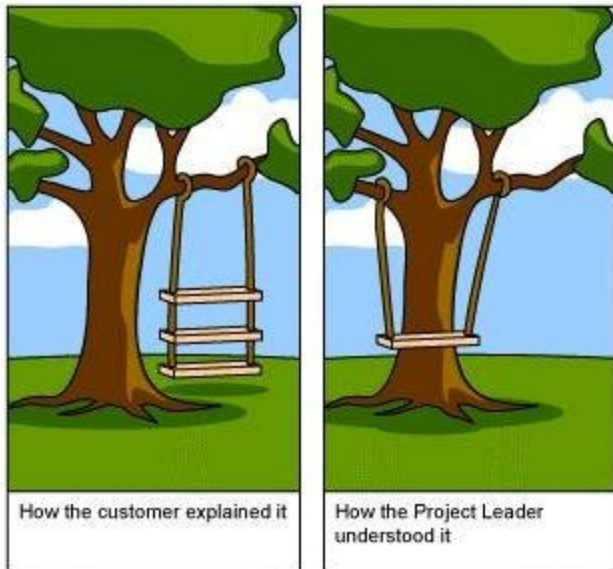
Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

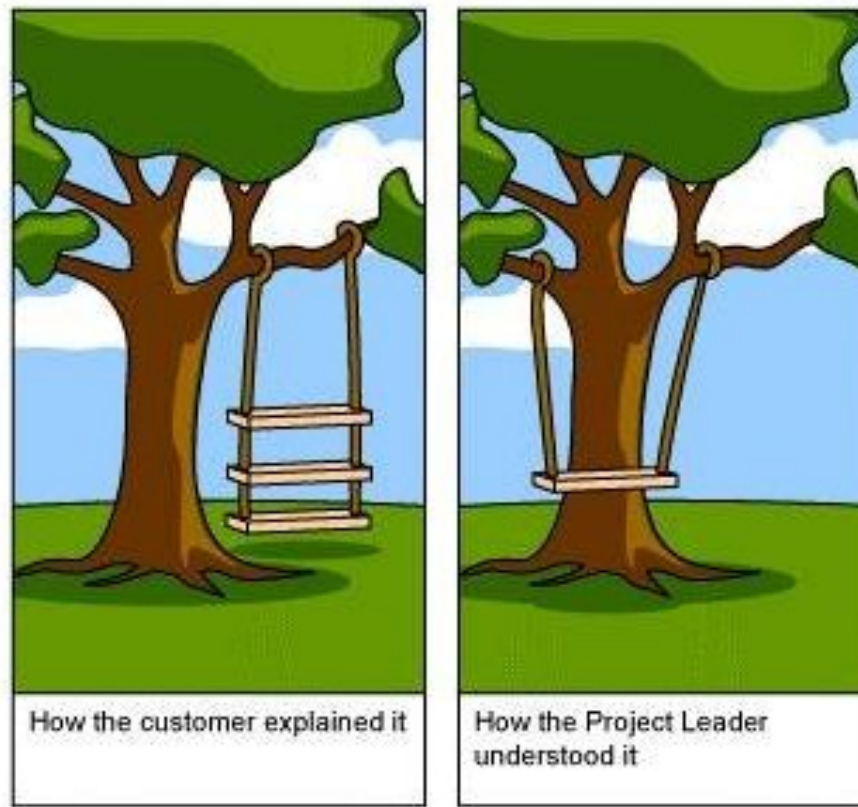
Software Engineering 1

Definition



Prof. Dr. Andreas Judt

Was soll entwickelt werden?



Was soll entwickelt werden?

- Funktionsweise des zu entwickelnden Systems wird aus den Anforderungen (engl. requirements) des Auftraggebers entwickelt.
 - qualitative und quantitative Eigenschaften eines Produkts aus Sicht des Auftraggebers
- Systemanalyse (engl. requirements engineering)
 - systematische Vorgehensweise, um Anforderungen in einem iterativen Prozess (Definieren des Produkts) zu ermitteln

Was soll entwickelt werden?

- Ziel der Definitionsphase:
 - vollständiges, konsistentes und eindeutiges Produktmodell erzeugen
- Ergebnis der Definitionsphase: Produkt-Definition
 - Pflichtenheft und ggfs. erweitertes Glossar
 - Produktmodell (hängt von der verwendeten Methode ab)
 - Prototyp der Benutzerschnittstelle oder Pilotsystem (Kernfunktionalität)
 - Benutzerhandbuch

Formulierung von Anforderungen

- Unterscheidung in der Formulierung
 - textuell oder grafisch
 - informal oder formal
- textuelle Beschreibung
 - beschreibt Anforderung in natürlichsprachigem Text

Formulierung von Anforderungen

- grafische Beschreibung
 - legt Anforderungen durch Symbole und Linien zwischen den Symbolen fest
 - Symbole und Linien werden mit Texten bezeichnet
- formale Beschreibung
 - Verwendung einer textuellen formalen Sprache, die durch eine Grammatik festgelegt wurde.
 - Grammatik beschreibt die Gültigkeit von Zeichenketten und Worten, also syntaktisch formal, nicht semantisch

Pflichtenheft

- Pflichtenheft
 - verbale Beschreibung der Anforderungen an ein neues Produkt
- Verschiedene Gliederungsschemata:
 - IEEE SRS (Software Requirements Schema) von 1984
 - IEEE Standard 830 von 1998
 - andere (eigene) Strukturen
- Eigenschaften des Pflichtenhefts
 - möglichst standardisierte Struktur
 - Nummerierung der Anforderungen
 - gute Lesbarkeit
 - wird unmittelbar nach Abschluss der Planungsphase erstellt
 - Anforderungen müssen ausreichend detailliert sein

Beispiel für Gliederungsschema eines Pflichtenhefts

1. Zielbestimmung

1.1. Musskriterien

1.2. Wunschkriterien

1.3. Abgrenzungskriterien

Ziele, die mit dem Produkt nicht erreicht werden sollen

2. Produkteinsatz

2.1. Anwendungsbereiche

2.2. Zielgruppen

ggfs. erforderliche Qualifikation beschreiben

2.3. Betriebsbedingungen

physikalische Umgebung, tägliche Betriebszeit, Beaufsichtigung des laufenden Systems

Beispiel für Gliederungsschema eines Pflichtenhefts

3. Produktübersicht

alle wichtigen Geschäftsprozesse in Form eines
Übersichtsdiagramms

4. Produktfunktionen

Detaillierung der Funktionen aus dem Lastenheft, ggfs. Gliederung
nach Geschäftsprozessen, Listen, Reports

Beispiel für Gliederungsschema eines Pflichtenhefts

5. Produktdaten

langfristig zu speichernde Daten aus Benutzersicht mit Mengengerüst

6. Produktleistungen

Leistungsanforderungen bzgl. Zeit oder Genauigkeit für in 5. genannte Datenmengen

7. Qualitätsanforderungen

Qualitätsmerkmale und einzuhaltende Standards

Beispiel für Gliederungsschema eines Pflichtenhefts

8. Benutzerschnittstelle

Anforderungen an die Interaktion mit dem Benutzer

9. Nichtfunktionale Anforderungen

Anforderungen, die sich nicht auf die Funktionalität oder Benutzerschnittstelle beziehen

z.B. Barrierefreiheit, Einhaltung von Normen und Gesetzen, Zertifizierung, Sicherheit, Plattformunabhängigkeit

Beispiel für Gliederungsschema eines Pflichtenhefts

10. Technische Produktumgebung
bei Client/Server Systemen getrennt angeben!

10.1 Software
Betriebssystem, Laufzeitumgebung, Datenbank, etc.

10.2 Hardware
CPU, Peripherie, minimale Konfiguration

10.3 Organisatorische Randbedingungen
Voraussetzungen für den Produkteinsatz

10.4 Produkt-Schnittstellen
Schnittstellen für Wartung und Integration

Beispiel für Gliederungsschema eines Pflichtenhefts

11. Spezielle Anforderungen an die Entwicklungsumgebung

11.1 Software

Software-Werkzeuge und Fremdprodukte

11.2 Hardware

11.3 Organisatorische Randbedingungen

11.4 Entwicklungs-Schnittstellen

Zugang zu bestehenden Systemen, z.B. einer Datenbank

Beispiel für Gliederungsschema eines Pflichtenhefts

12. Gliederung in Teilprodukte

Teilprodukte aus Sicht des Auftraggebers und Reihenfolge, in der die Teile entwickelt werden.

13. Ergänzungen

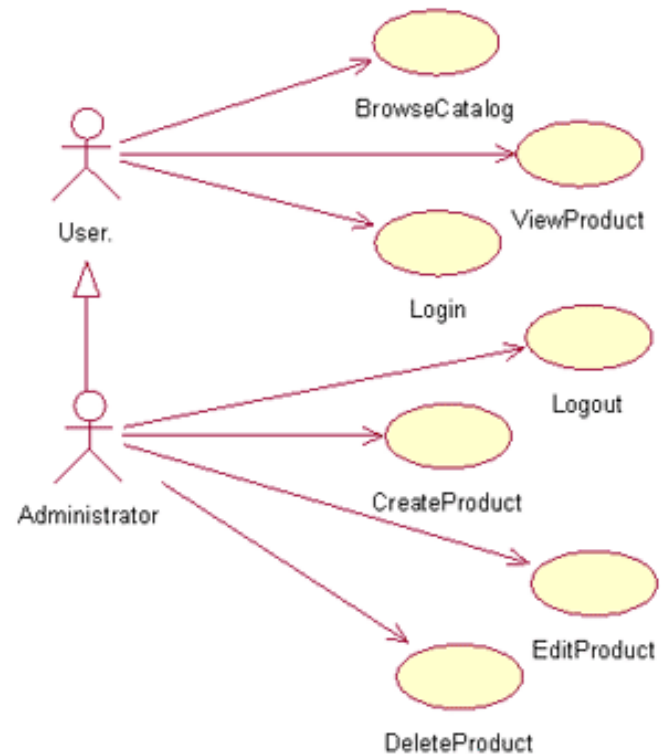
z.B. Festlegung von Installationsbedingungen, etwa baulich und räumliche Voraussetzungen, Testdaten, Normen, Vorschriften, Patente, Lizenzen und externe Unterstützung

Sichten und Analysemethoden

- Verschiedene Sichten:
 - funktional: Funktionsbäume (Selbststudium), Geschäftsprozesse, Datenflussdiagramme (Selbststudium)
 - objektorientiert
 - algorithmisch
 - regelbasiert (hier nicht relevant)
 - Stichwort: Entwicklung regelbasierter Expertensysteme
 - zustandsorientiert
 - Automaten
 - Petri-Netze
- Analysemethoden
 - objektorientiert
 - strukturiert (Selbststudium)
 - echtzeit-orientiert (hier nicht relevant)

Geschäftsprozesse (use cases)

- Geschäftsprozess (engl. use case):
 - Arbeitsablauf, der mithilfe einer Software durchgeführt wird und manuelle oder organisatorische Anteile besitzen kann.
 - beschreibt die Benutzerkommunikation mit dem Softwaresystem
- Akteur (engl. actor):
 - Benutzer oder externes System, das mit dem zu modellierenden System kommuniziert
 - Akteure befinden sich immer außerhalb des Systems!
- Beschreibung:
 - Gerundium
 - Substantiv und Verb
 - Anfangs- und Endpunkt



Geschäftsprozess – eine Schablone

- Name des Geschäftsprozesses
 - Was wird getan?
- Ziel
 - globale Zielsetzung bei erfolgreicher Ausführung
- Kategorie
 - primär (häufig benutzt), sekundär (selten benutzt), optional
- Vorbedingung
 - erwarteter Zustand, bevor der Geschäftsprozess beginnt
- Nachbedingung Erfolg
 - erwarteter Zustand nach Ausführung = Ergebnis des Geschäftsprozesses
- Nachbedingung Fehlschlag
 - erwarteter Zustand wenn Geschäftsprozess das Ziel nicht erreichen konnte

Geschäftsprozess – eine Schablone

- Akteure
 - Rollen von Personen oder andere Systeme
- Auslösendes Ereignis
 - Ereignis, nachdem der Geschäftsprozess gestartet wird
- Beschreibung
 - textuelle Beschreibung des Standardfalls, also der am häufigsten auszuführen ist
- Erweiterungen
 - textuelle Beschreibung der Nicht-Standardfälle
- Alternativen
 - alternative Verarbeitung im Standardfall

Geschäftsprozesse – analytisches Vorgehen

- Formulierung
 - so formulieren, dass der Auftraggeber sie verstehen kann
 - Kommunikation der Akteure mit dem System steht im Mittelpunkt
 - keine interne Struktur oder Algorithmen beschreiben
 - Standardfall immer vollständig spezifizieren
 - maximal eine Seite Beschreibung
 - auf potenzielle Fehlerquellen prüfen
- Bewertung
 - im Mittelpunkt stehen zentrale Abläufe, nicht elementare Funktionen
 - Konzentration auf die Standardabläufe
 - falsche oder unsachgemäße Anwendung leicht möglich
 - z.B. zu hoher Detailgrad, Verwendung von Geschäftsprozessen als Kontrollstrukturen, Abbildung auf die Benutzerschnittstelle

Objektorientierte Sicht

- Ausgangspunkt: Objektorientierte Analyse
 - Vorlesung Software Engineering 2
- Formulierung von
 - Assoziationen (Beziehung zwischen Objekten),
 - Kardinalitäten (Anzahl von assoziierten Objekten),
 - Aggregationen (Sonderfall der Assoziation, „besteht aus“),
 - Kompositionen („ist Teil von“ – existenziell)
- Formulierung von OO-Modellen
 - Sequenzdiagramm
 - zeitlicher Aspekt: Reihenfolge und Verschachtelung von Operationen
 - Kollaborationsdiagramm
 - Verbindung zwischen Objekten
 - Angabe der Reihenfolge durch Nummerierung

Datenorientierte Sicht

- Entity Relationship Model (ER-Model)
 - auch „Information Modeling“
 - Ziel: permanent gespeicherte Daten und Beziehungen untereinander beschreiben
 - Ergebnis: konzeptionelles Modell, das gegen Veränderungen der Funktionalität weitgehend stabil ist
- ER-Konzept vergleichbar mit OO-Konzept
 - Unterschied: Operationen und Nachrichten fehlen
 - Erweiterung um Aggregation und Vererbung: semantische Datenmodellierung
- Unternehmensdatenmodell
 - Informationsstrukturen und Geschäftsprozesse aller Bereiche mit Schnittstellen zueinander

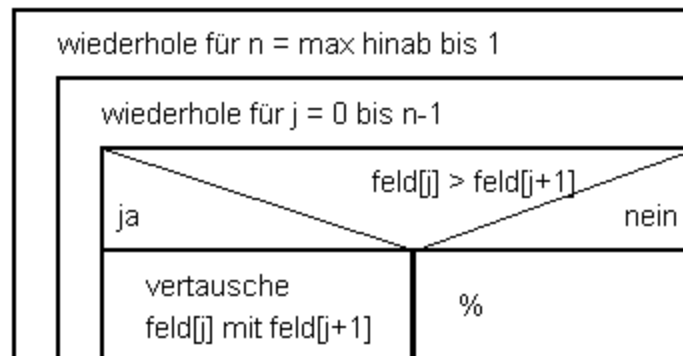
Algorithmische Sicht

- Aufbau von Kontrollstrukturen
 - Typen:
 - Sequenz
 - Auswahl
 - Wiederholung
 - Aufruf anderer Algorithmen
 - Ergebnis (verschiedene Notationen):
 - Struktogramm
 - Pseudo-Code
 - Programmablaufplan
- Aufbau von Entscheidungstabellen oder –bäumen
 - Darstellungen sind äquivalent

Algorithmische Sicht: Bubblesort

```
BEGIN BUBBLESORT
firstindex = 1
lastindex = ARRAYLENGTH(element)
WHILE lastindex > firstindex
  currentindex = 1
  WHILE currentindex < lastindex
    IF element(currentindex) > element(currentindex + 1) THEN
      SWAP(element(currentindex), element(currentindex + 1))
    ENDIF
    currentindex = currentindex + 1
  ENDWHILE
  lastindex = lastindex - 1
ENDWHILE
END BUBBLESORT
```

Bubble-Sort



Algorithmische Sicht: Entscheidungstabelle

Name der Entscheidungstabelle		Regel 1	Regel 2	Regel k
wenn ...	Bedingung 1	J	N	N
wenn ...	Bedingung 2	J	J	N
wenn ...	Bedingung n	N	N	N
dann ...	Aktion 1	X		
dann ...	Aktion 2		X	X
dann ...	Aktion m			X

Algorithmische Sicht: Entscheidungstabelle

- beschreibt Aktionen, die bei erfüllten oder nicht erfüllten Bedingungen vorzunehmen sind
- Vollständige Entscheidungstabelle:
 - formal: alle Kombinationen der Bedingungen sind beschrieben
 - inhaltlich: alle möglichen Problemkombinationen sind aufgeführt
- Optimierung:
 - bei identischen Aktionen sind nur gleiche Bedingungen relevant. Ungleiche Bedingungen als irrelevant markieren.

Algorithmische Sicht: Entscheidungstabelle

- Anzahl der Regeln für vollständige Entscheidungstabelle: z^n
 - z = Anzahl der Zustände
 - n = Anzahl der Bedingungen
 - Zustandsmenge $\{ja, nein\}$: $z=2$
- Ursprüngliche Anwendung: Entwurf von Logikschaltungen
 - Zustandsmenge: $\{0,1\}$: $z=2$
 - Bedingungen \Leftrightarrow Eingänge, Aktionen \Leftrightarrow Ausgänge, Aktion ausgeführt: Ausgang = 1
 - Anzahl der Bedingungen: Zahl der Eingänge
 - Anzahl der Aktionen: Zahl der Ausgänge

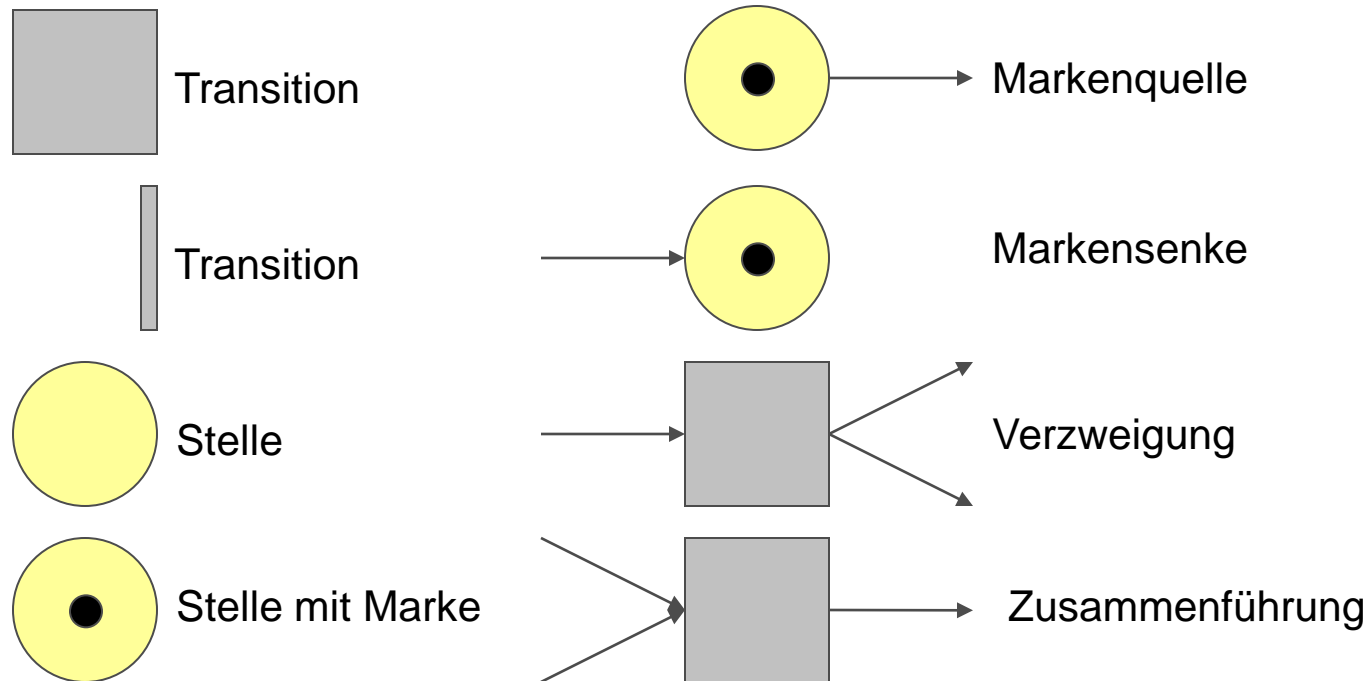
Beispiel: Zustandstabelle

aktueller Zustand	Ereignis	Aktion	Folgezustand
	Start	Initialisieren	Zustand 1
Zustand 1	Ereignis 1	Aktion 1	Zustand 5
Zustand 1	Ereignis 2	Aktion 2	Zustand 3
Zustand 2	Ereignis 3	Aktion 3	Zustand 1
Zustand 3	Ereignis 2	Aktion 4	Zustand 4
Zustand 3	Ereignis 4	Aktion 5	Zustand 3
Zustand 4	Ereignis 4	Aktion 5	Zustand 1
Zustand 5	Ereignis 1	Aktion 2	Zustand 5
Zustand 5	Ereignis 5	Aktion 6	Zustand 2

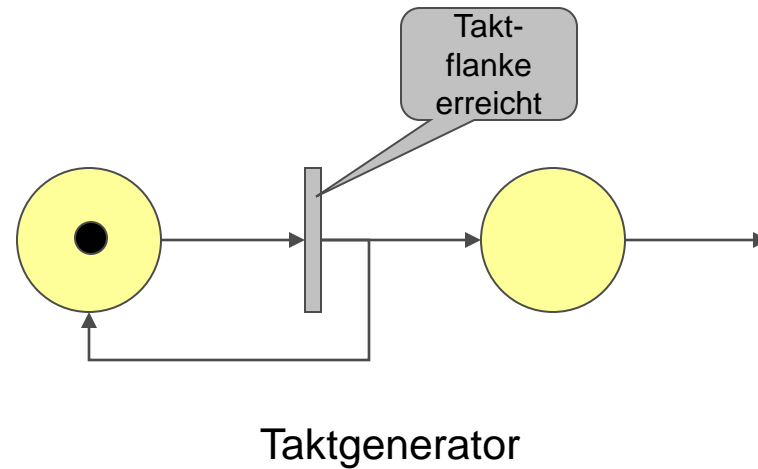
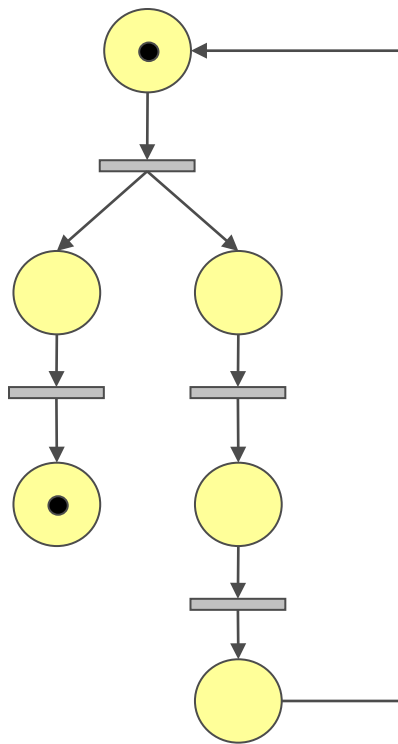
Petri-Netze

- gerichteter Graph mit 2 verschiedenen Knoten: Stellen und Transitionen
 - Stelle = Platz = Zustand: als Kreis dargestellt
 - Transition = Hürde = Zustandsübergang: als Quadrat dargestellt
- Transitionen
 - verarbeiten Daten
 - besitzen Eingabe- und Ausgabestellen
- Verhaltensbeschreibung
 - Stellen werden markiert: Marken (engl. token)
- Schaltregel
 - Eine Transition kann feuern (oder schalten), wenn jede seiner Eingabestellen mindestens eine Marke enthält.
 - Feuert eine Transition, wird aus jeder Eingabestelle eine Marke entfernt und zu jeder Ausgabestelle eine Marke hinzugefügt.

Petri-Netze: Notation



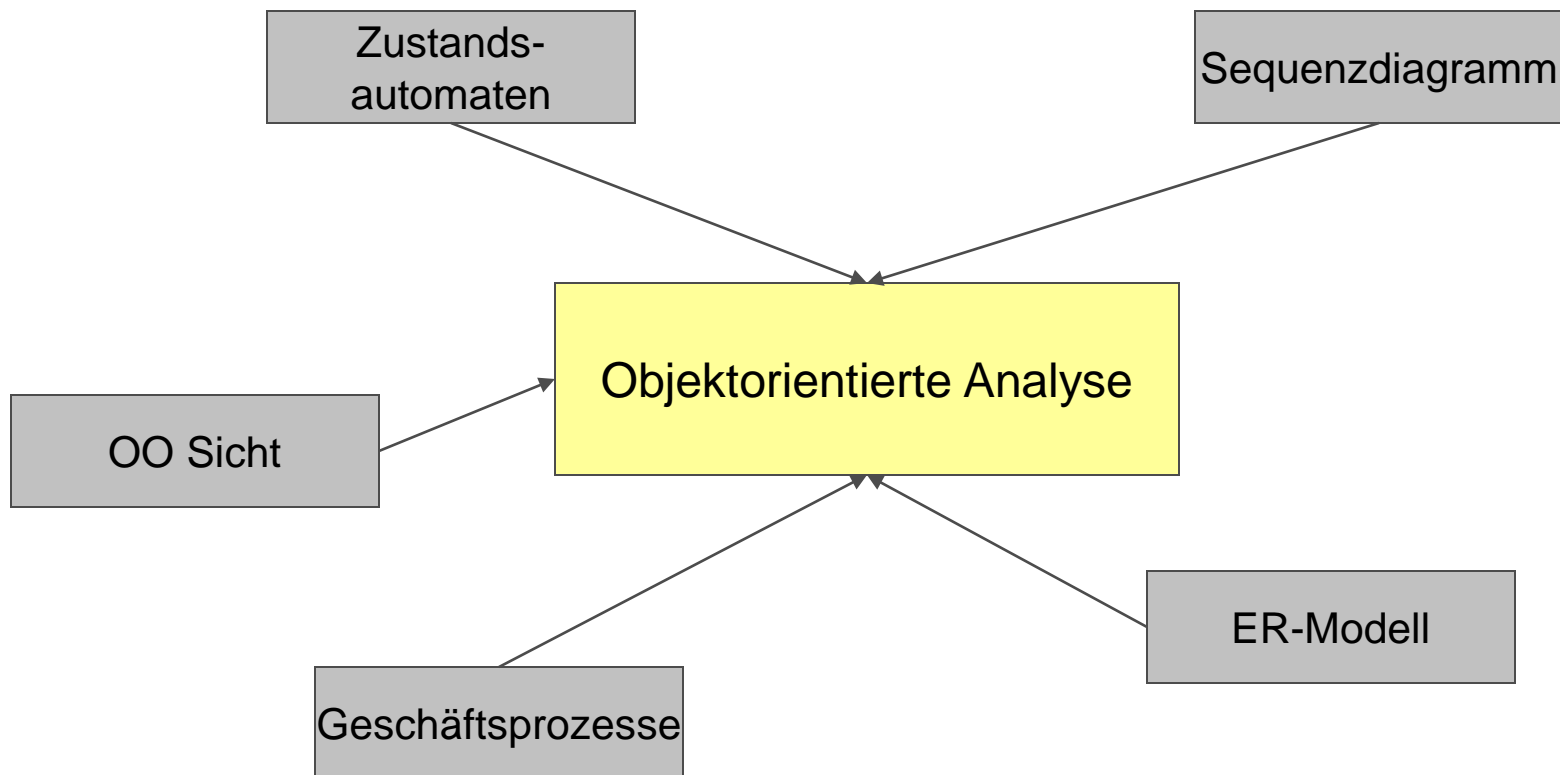
Petri-Netze: Beispiele



Petri-Netze: Bewertung

- Vorteile
 - sehr einfach, gut darstellbar
 - sehr gute Werkzeugunterstützung
 - auch Codegenerierung teilweise möglich
 - solides theoretisches Fundament (PSPACE-vollständig)
 - weit verbreitet zur Modellierung kooperierender Prozesse
 - viele Varianten und Spezialisierungen, z.B. Echtzeit-Erweiterungen
 - Abbildung auf Zustandsdiagramme möglich
- Nachteile
 - werden sehr schnell komplex und unübersichtlich
 - Erzeugung neuer Prozesse kaum übersichtlich darstellbar
 - komplexe Netze sind nicht intuitiv verständlich

Objektorientierte Analyse (OOA)



Objektorientierte Analyse (Auszug)

- Analyse im Großen
 - Geschäftsprozesse aufstellen
 - Pakete (Module) bilden

- Erzeugen eines statischen Modells
 1. Klassen identifizieren
 2. Assoziationen identifizieren
 3. Attribute identifizieren
 4. Vererbungsstrukturen identifizieren
 5. Assoziationen vervollständigen
 6. Attribute spezifizieren
 7. Muster identifizieren

Objektorientierte Analyse (Auszug)

- Erzeugen eines dynamischen Modells
 1. Szenarios erstellen
 - Sequenzdiagramm, Kollaborationsdiagramm, Aktivitätsdiagramm
 2. Zustandsautomat erstellen
 - für jede Klasse: Zustandsdiagramm (Lebenszyklus)
 3. Operationen beschreiben
 - Notation nach je nach Komplexitätsgrad wählen
- Ergebnis der OOA
 - UML Modell des Zielsystems

Viele Sichten, viele Analysen...

Wie wird nun ein konsistentes
Softwaresystem entwickelt, dass
alle Anforderungen erfüllt?

Selbstkontrolle

- Was ist Ziel der Definitionsphase?
- Wie unterscheidet sich das Pflichtenheft vom Lastenheft?
- Wie wird ein Geschäftsprozess beschrieben?
- Wie würden Sie eine Zustandstabelle in ein Petri-Netz überführen?
- Überführen Sie das Petri-Netz Taktgenerator in eine Zustandstabelle.

Kontakt

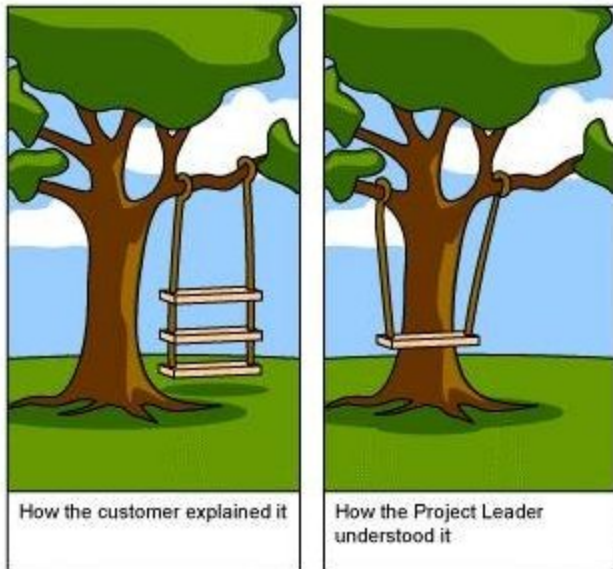
Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Software Engineering 1

Entwurf



Prof. Dr. Andreas Judt

Aufgabe des Entwurfs

- In der Entwurfsphase (Entwurf, engl. design) wird aus den Anforderungen eine Software-Architektur entwickelt.
 - baut auf den Ergebnissen der Definitionsphase auf
 - Entwurf ist der Ausgangspunkt für die Implementierung
- Aufgaben des Entwerfers = Software-Architekt
 - Erstellen der Software-Architektur mit Netzwerkverteilung, Verbindung zur Benutzerschnittstelle und Datenbank
- Der Software-Architekt trifft Grundsatzentscheidungen.

Einflüsse aus der Definitionsphase

- Einsatzbedingungen
 - z.B. nebenläufig, verteilt, echtzeitfähig, parallel
- Umgebungs- und Randbedingungen
 - z.B. Zielplattform, Hardware, Betriebssystem, Laufzeitumgebung
- nichtfunktionale Anforderungen und Qualitätsvorgaben
 - z.B. Wiederverwendbarkeit, Barrierefreiheit, leicht verständliche Architektur, Migrationsfähigkeit

Grundsatzentscheidungen

- Datenhaltung
 - Speicherung in einer Datei, Verwendung von XML
 - Entwurf einer relationalen Datenbank
 - Entwurf einer objektorientierten Datenbank
- Verteilung im Netz
 - Client/Server Architektur
 - Web-Architektur
 - Fat Client Architektur
 - Service-orientierte Architektur (SOA)

Grundsatzentscheidungen

- Benutzerschnittstelle
 - grafische Schnittstelle (engl. GUI, graphical user interface)
 - Auswahl der Technologie
 - Spracherkennungssystem
 - Kommandozeile
- Infrastruktur
 - Kommunikation, z.B. SOAP, RMI, CORBA
 - Sicherheitsaspekte
 - Rahmenwerke (engl. frameworks)

Aufbau von Klassenbibliotheken

- Klassenbibliothek: organisierte Software-Sammlung, aus der Entwickler Funktionalität verwenden
 - Einsparung von Lern- und Entwicklungsaufwand
 - Einsatz verschiedener Varianten möglich
 - Meistens hohe Qualität der Implementierung
- Nachteile:
 - Bibliotheken sind oft sehr komplex
 - hoher Einarbeitungsaufwand
 - möglicherweise Namenskonflikte mit anderen Bibliotheken

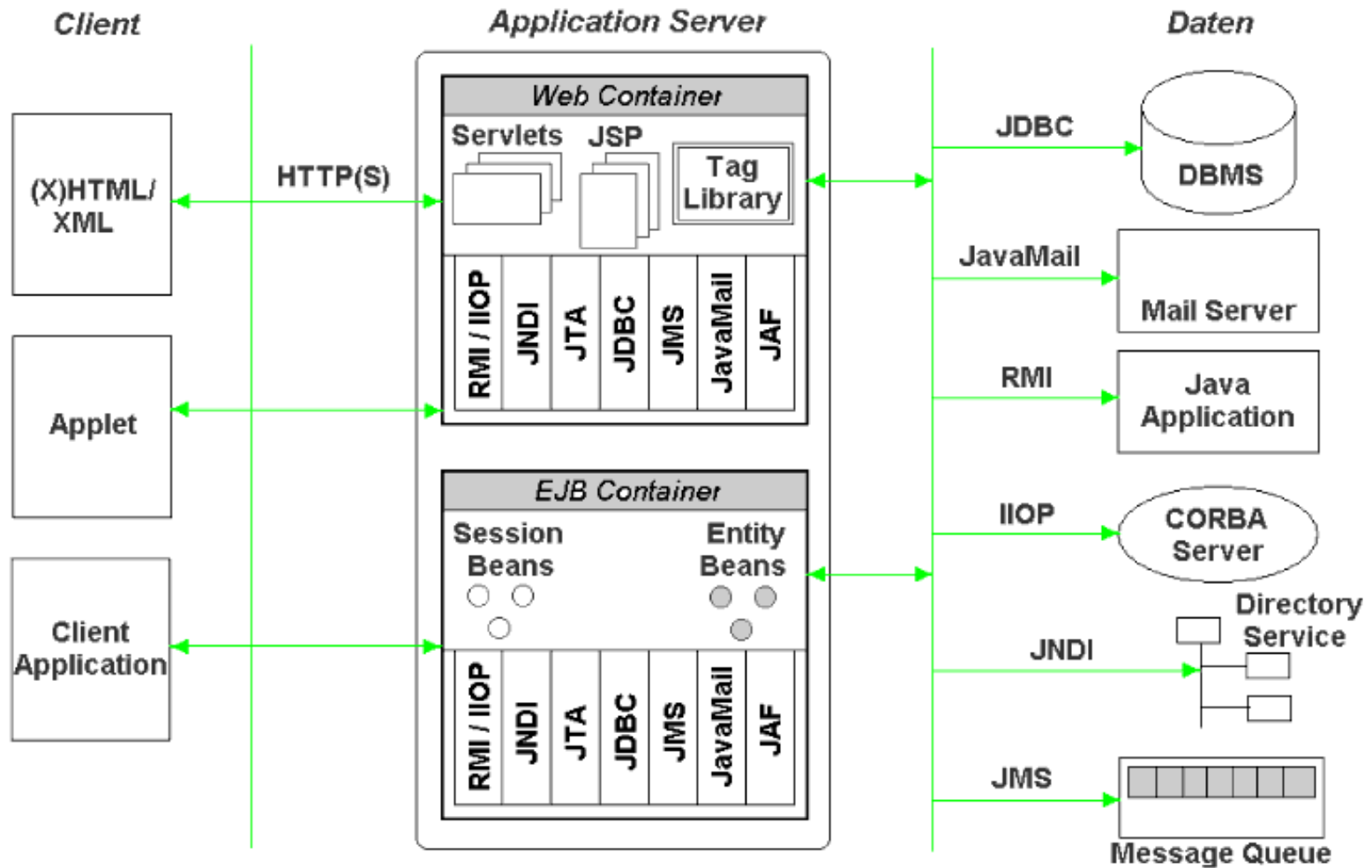
Rahmenwerke

- Rahmenwerk, engl. framework
- Unterschied zur Klassenbibliothek
 - Rahmenwerk bestimmt die Software-Architektur
- Eigenschaften
 - erweiterbares System kooperierender Klassen
 - implementieren einen wiederverwendbaren Entwurf für einen Anwendungsbereich
 - besteht aus konkreten und abstrakten Klassen, die Schnittstellen definieren

Entwurf mit einem Rahmenwerk

- Bildung von Unterklassen, um Rahmenwerk zu verwenden und anzupassen
- Selbst definierte Unterklassen empfangen und verarbeiten Nachrichten aus den Klassen des Rahmenwerks
 - Hollywood-Prinzip: „Don't call us – we'll call you.“
- Software-Architekt kann sich auf die Funktionalität des Zielsystems konzentrieren.

Rahmenwerk: Beispiel JEE



Entwurfsmuster

- Entwurfsmuster, engl. design patterns
 - bewährte generische Lösungen für wiederkehrende Aufgaben
 - definieren wiederverwendbare Standardlösungen
 - bieten alternative Entwürfe
 - bilden ein gemeinsames Vokabular von Problemlösungen bei Architekten und Entwicklern
 - Muster werden klassifiziert nach ihrem Zweck.
- Es gibt heute mehrere Tausend Muster für verschiedenste Aufgabenbereiche.

Entwurfsmuster: Beispiele

- Abstrakte Fabrik
 - Abstrakte Klasse definiert das Ergebnis eines Erzeugungsprozesses.
 - Konkrete Unterklassen erzeugen das Ergebnis, ohne das der Auftraggeber darüber Informationen erhält.
 - Auswahl der konkreten Fabrik kann z.B. durch Konfiguration erfolgen.
- Singleton
 - Klasse, von der es höchstens ein Objekt geben kann. Auf das Objekt kann global zugegriffen werden.

Entwurfsmuster: Beispiele

- Beobachter
 - bei Änderung eines Objekts werden alle abhängenden Objekte benachrichtigt und automatisch aktualisiert
- Kompositum
 - z.B. Java Swing
- Model-View-Controller (MVC)

Komponenten

- Komponente (, engl. component; auch Halbfabrikat)
 - abgeschlossener, binärer Software-Baustein
 - besitzt anwendungsorientierte, semantisch zusammengehörende Funktionalität
 - stellt Funktionalität über Schnittstellen bereit
 - hohe Wiederverwendbarkeit

Komponenten

- Komponentenbasierte Softwareentwicklung
 - schnelle, preiswerte Softwareentwicklung mit Halbfabrikaten
 - oft visuelle Entwicklung mit CASE Tools möglich
 - Komponenten werden für den Einsatz ggf. konfiguriert
- Beispiele für komponentenbasierte Softwareentwicklung
 - GUI, z.B. Java Swing
 - Java Beans, Enterprise Java Beans (EJB)
 - Web Services

Analyse von Komponenten: Introspektion

- Integration in CASE Tools (computer aided software engineering) und visuelle Programmierung erforderte maschinelle Analyse der Bedienung
 - Software wird von Software inspiziert!
- Einhaltung von Namenskonventionen liefert semantische Informationen
 - Semantik kann detaillierter in beigefügten Info-Klassen beschreiben werden,
- Für inspizierbare Komponenten können automatisch Parameter-Editoren erzeugt werden.

Selbstkontrolle

1. Welche Aufgabe hat die Entwurfsphase?
2. Welche Grundsatzentscheidungen trifft der Software-Architekt?
3. Was unterscheidet eine Klassenbibliothek von einem Rahmenwerk?
4. Wie kann man Programmteile automatisiert verwenden, ohne die Quelle zu kennen?

Kontakt

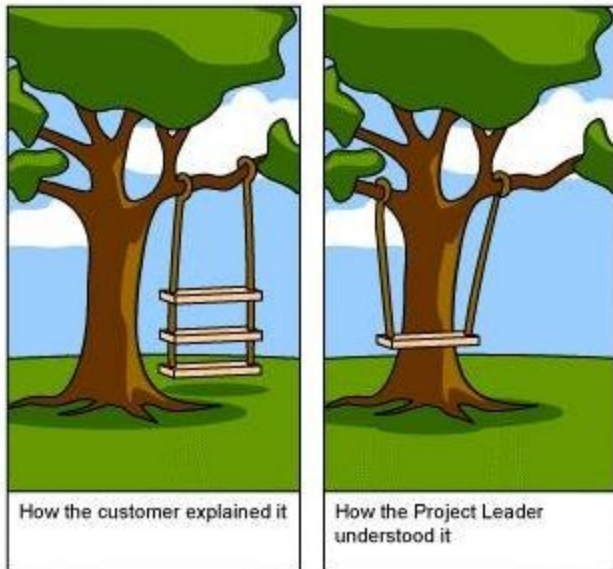
Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

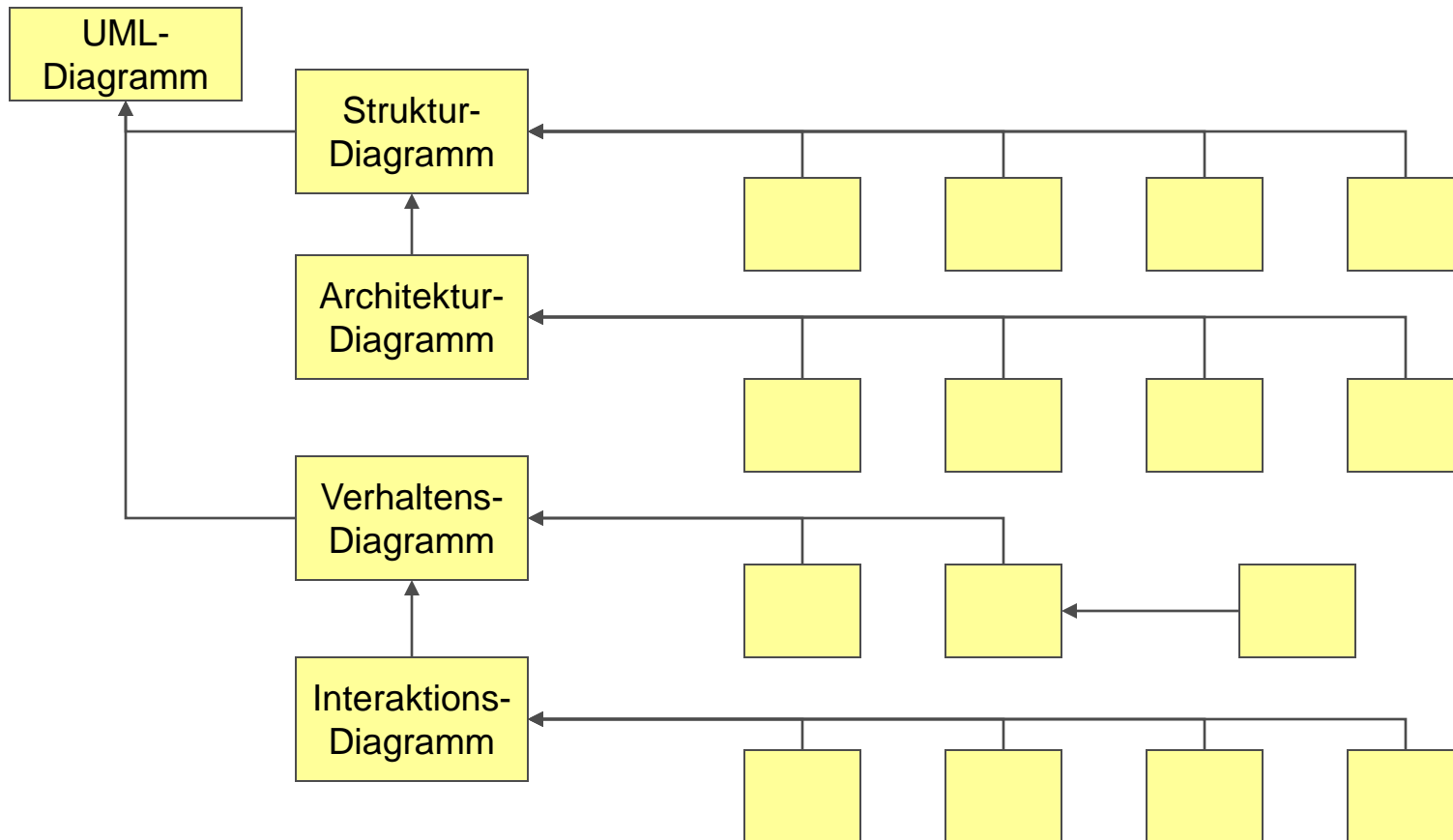
judt@dhbw-ravensburg.de

Software Engineering 1

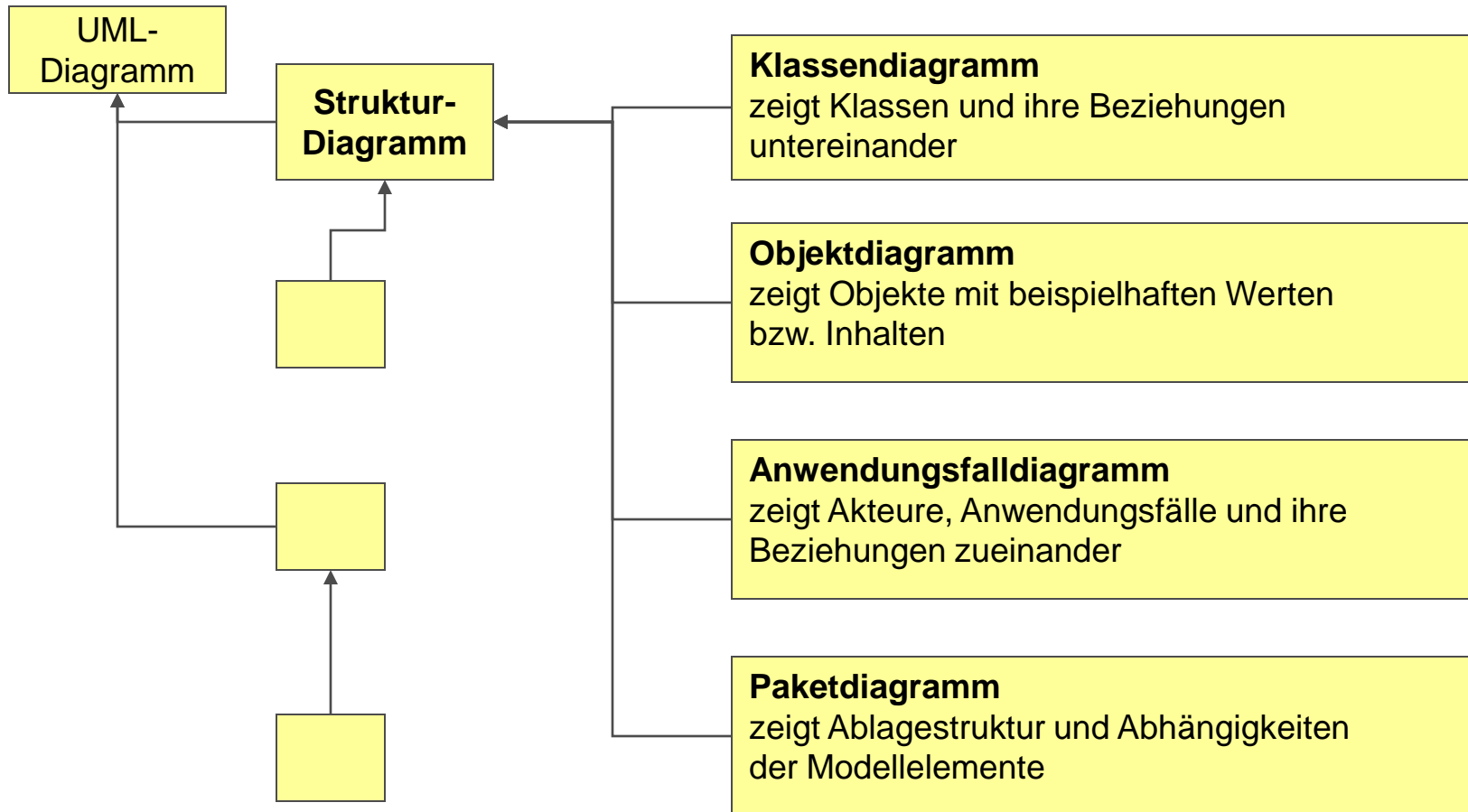
Unified Modeling Language (UML)



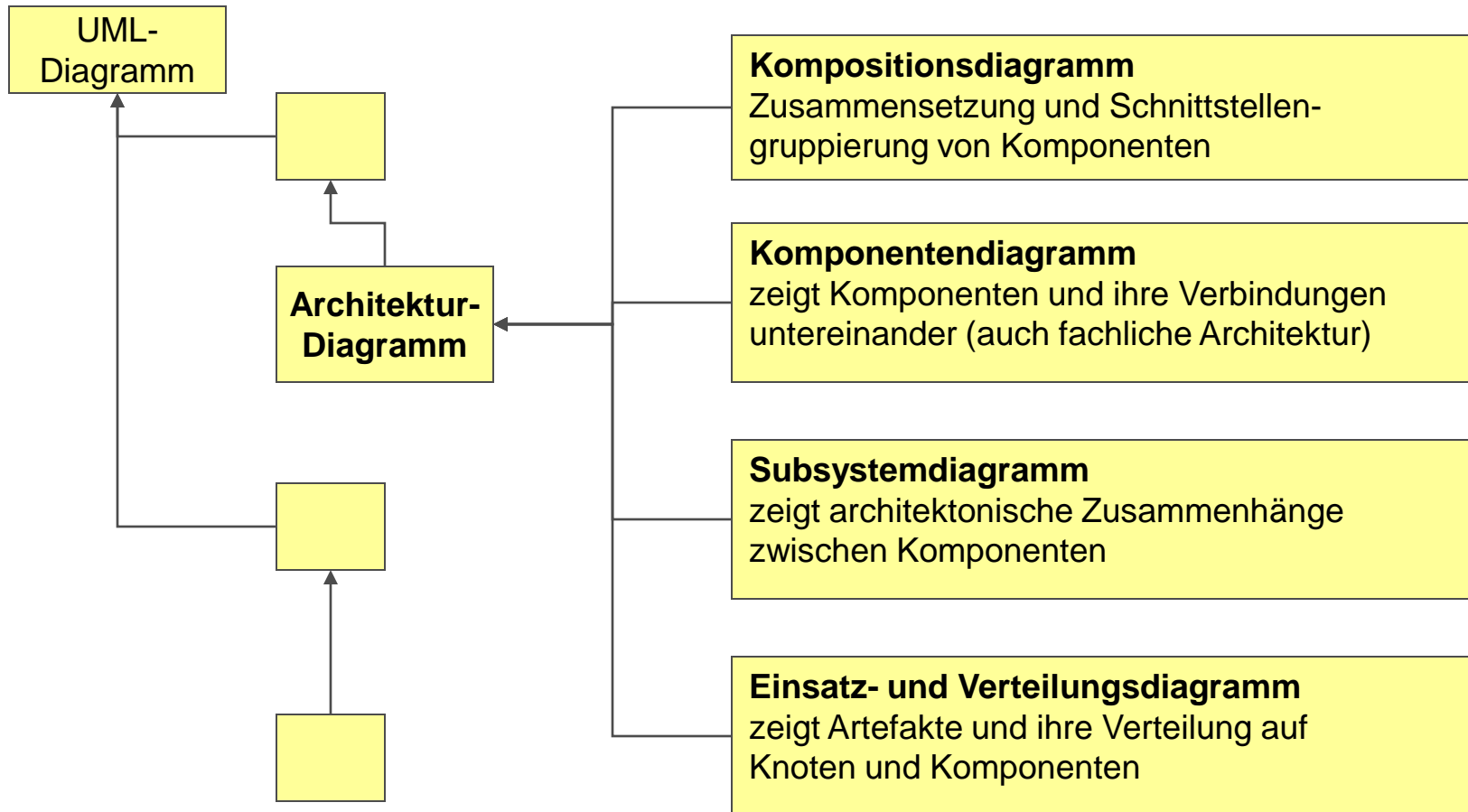
Prof. Dr. Andreas Judt



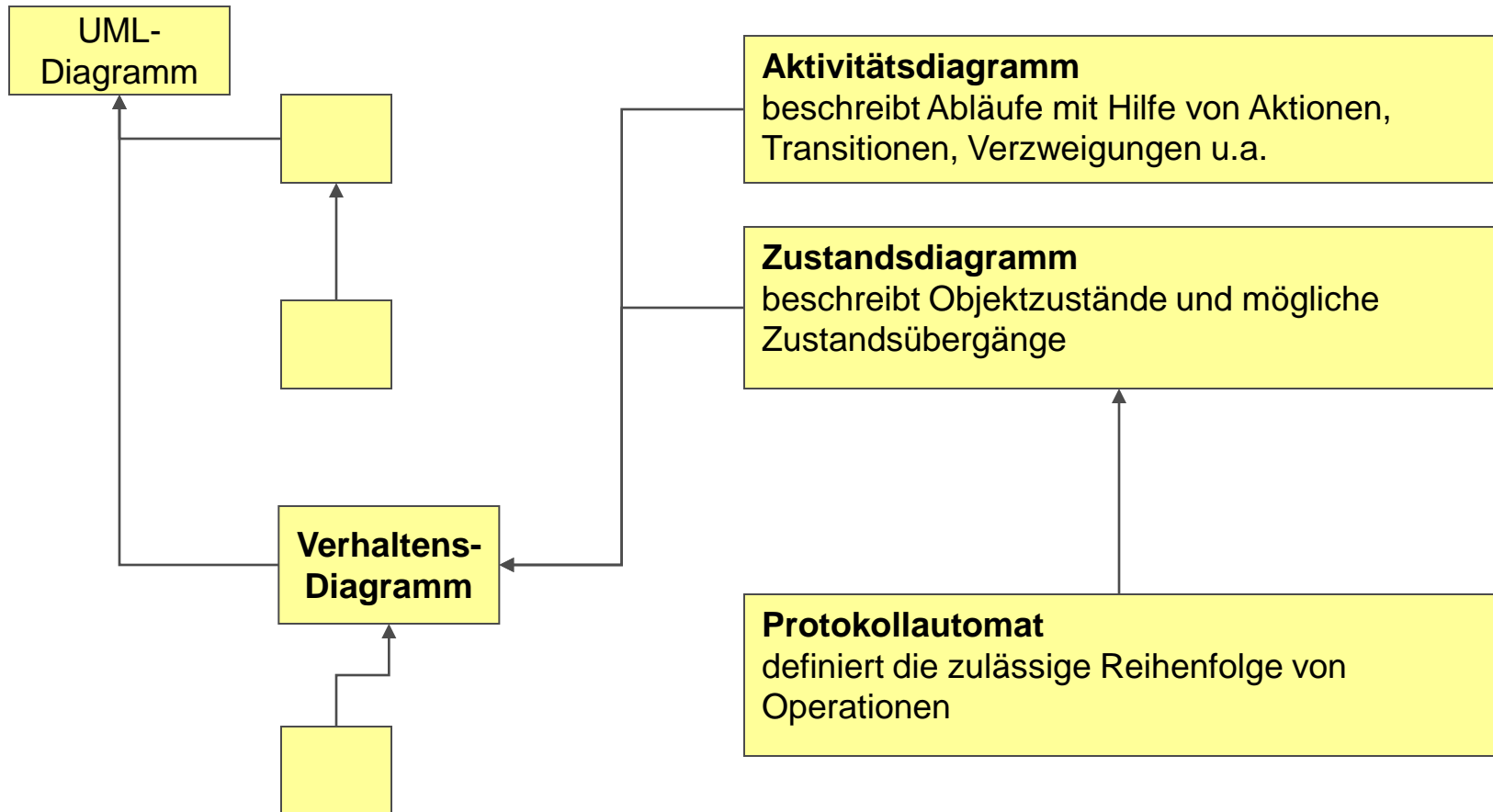
Strukturdiagramme



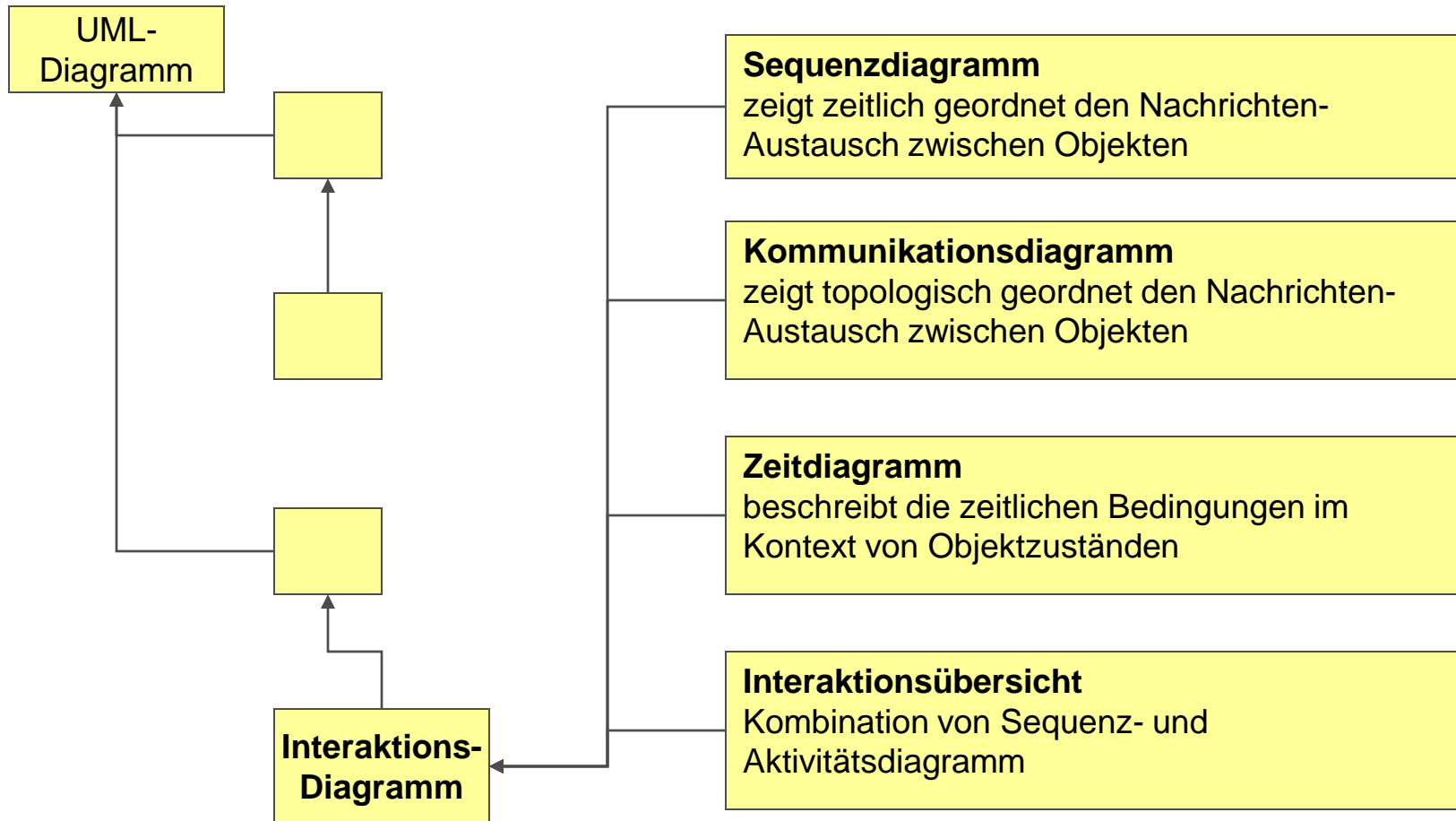
Architekturdiagramme



Verhaltensdiagramme

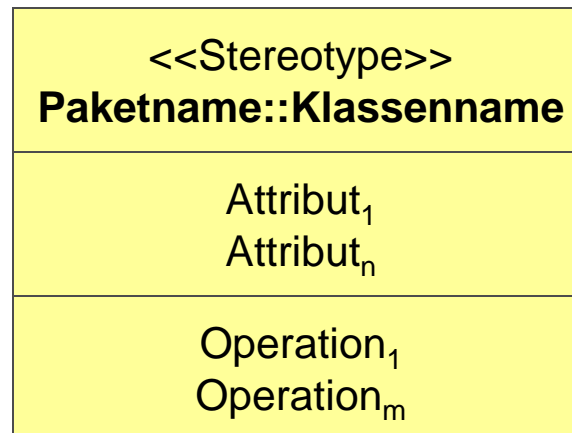


Interaktionsdiagramme



Klassen

- Eine Klasse definiert Attribute, Operationen und die Semantik einer Menge von Objekten.
 - wird oft auch als Bauplan bezeichnet

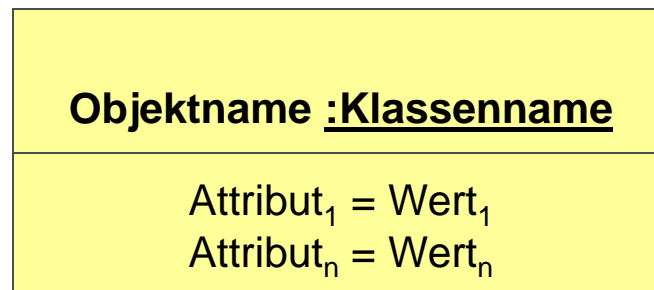


Klassen: Objekte

- Objekte
 - sind Exemplare einer Klasse
 - werden mit einem Konstruktor hergestellt
 - existieren erst zur Laufzeit eines Programms
 - empfangen Nachrichten

Klassen: Objekte

- Objekte werden in UML durch ihre Eigenschaften beschrieben.
 - Objektname (= Name des Zeigers)
 - Werte aller Attribute
- Die Beschreibung ist nur zu einem festgelegten Zeitpunkt gültig!



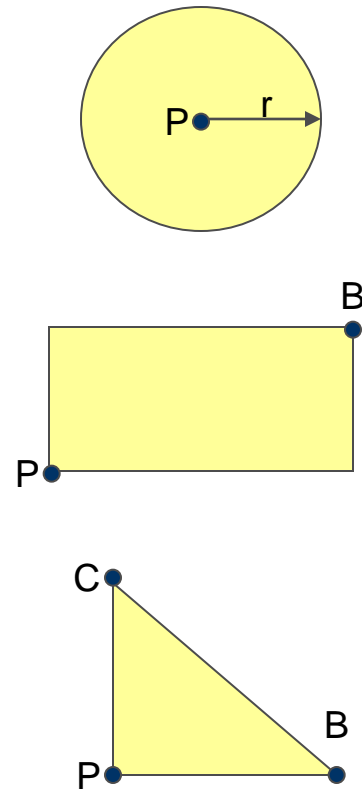
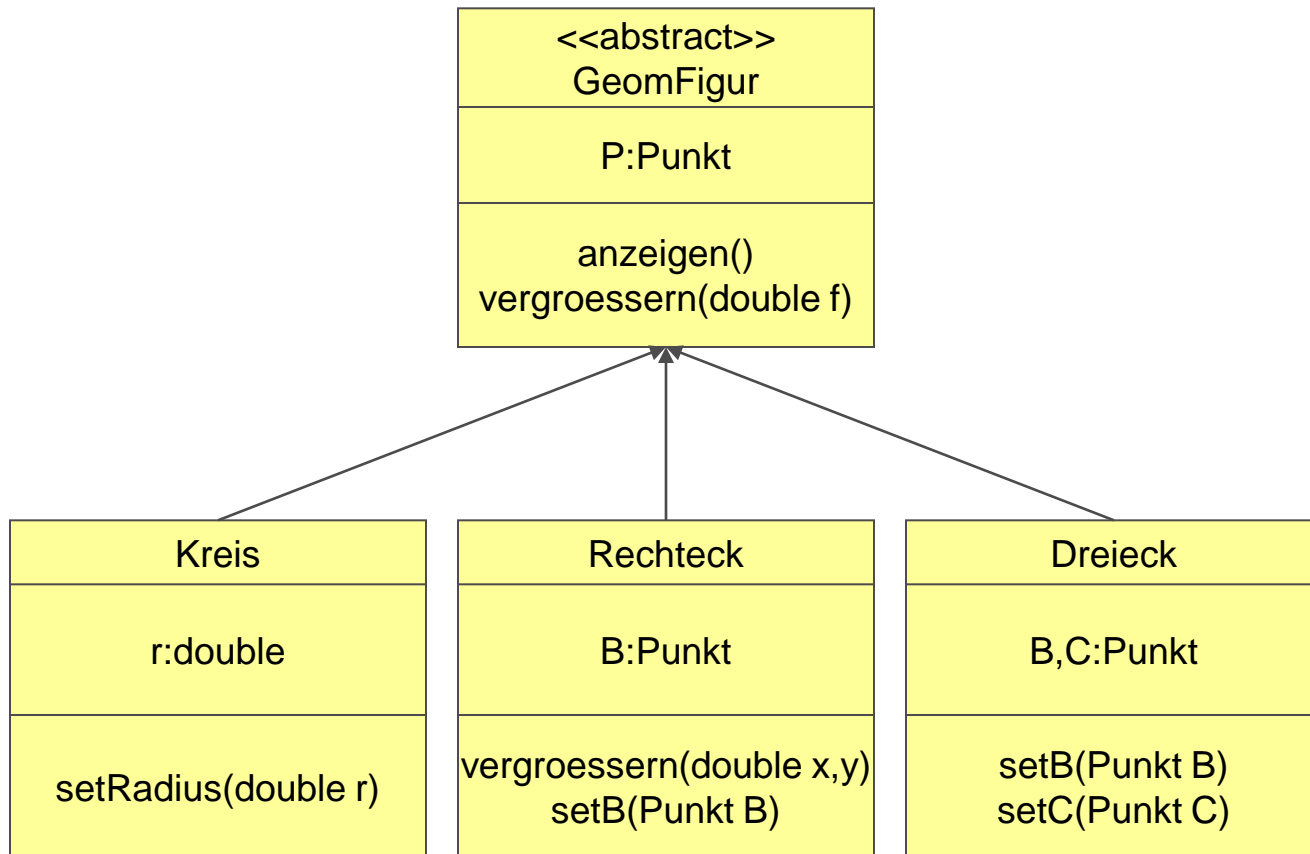
Klassen: Attribute und Operationen

- Attribute
 - sind Eigenschaften in Form von Daten
 - sind primitive Datentypen oder Objekte
 - besitzen in jedem Objekt verschiedene Werte
- Operationen
 - sind Dienstleistungen, die von Objekten angefordert werden können
 - sind werden über ihre Signatur definiert:
 - Name
 - Parameter
 - ggfs. Rückgabetyt (nicht in Java!)

Vererbungsbeziehung

- Eine Vererbung spezialisiert eine Klasse.
 - Die entstehende Unterklasse (oder abgeleitete Klasse) besitzt alle Eigenschaften der Oberklasse.
- Es gilt das Substitutionsprinzip.
 - Objekte von Unterklassen müssen zu jedem Zeitpunkt anstelle von Objekten ihrer Oberklassen eingesetzt werden können.
- Objektorientierung wird oft falsch eingesetzt.
 - Die Praxis zeigt, dass Vererbung nur selten eine Problemlösung ist.

Beispiel: geometrische Figuren



Vererbung: Restriktionen und Probleme

- Vererbung ist einfach, aber nicht ohne Tücken!
- Unterklassen müssen alle Zusicherungen ihrer übergeordneten Klassen erfüllen.
 - Zusicherungs-Verantwortlichkeits-Prinzip
- Eine Unterklasse darf nicht nur um Zusicherungen spezialisieren!

Klassifizierung von Klassen und Objekten

- Klassen und Objekte besitzen unterschiedliche Verwendungszwecke.
- Klassifizierung ist der Weg zu strukturierten Modellen und standardisierten Lösungen.
 - z.B. Entwurfsmuster
- Die Trennung der nachfolgenden Klassifizierung ist nicht immer scharf!
- Die Eigenschaften werden als Stereotype bezeichnet.

Klassifizierung: Entitätsklasse (engl. entity) <<entity>>

- stellt einen Sachverhalt oder realen Gegenstand dar
 - besitzt viele Attribute
 - besitzt viele primitive Operationen, z.B. set/get
 - wird in Analyse- und Strukturmodellen verwendet
 - besitzt ein einfaches Zustandsmodell
- z.B. Auto, Benutzer, Bestellung

Klassifizierung: Steuerungsklasse (engl. control) <<control>>

- definiert einen Ablauf, eine Steuerung oder eine Berechnung
 - besitzt wenige oder keine Attribute
 - existiert oft nur für eine Berechnung
 - wird ggfs. in einem Pool bereitgestellt
 - greift auf Entitätsklassen zur Anforderung von Daten zu und speichert Ergebnisse darin zurück
 - besitzt keine Zustände
- z.B. Klassen in mathematischen Bibliotheken

Klassifizierung: Schnittstellenklasse (engl. interface) <<interface>>

- abstrakte Definition von Schnittstellen
 - keine Attribute
 - keine Assoziationen
 - definiert eine Menge von Operationen
 - werden von Entitäts- und Steuerklassen implementiert
- z.B. java.rmi.Remote

Klassifizierung: Schnittstellenobjekt (engl. boundary) <<boundary>>

- liefert eine Sicht auf eine Menge anderer Objekte
 - bildet eine Zusammenstellung von Eigenschaften
 - delegiert Operationen weiter
 - haben keine Zustände
 - sind gewöhnlich als Singleton implementiert
 - siehe Entwurfsmuster
- z.B. Schnittstelle zu Hardware

Klassifizierung: Typ (engl. type) <<type>>

- definiert eine Menge von Attributen und Operationen.
 - kann (anders als Schnittstellenklassen) Attribute und Assoziationen besitzen
 - besitzt oft abstrakte Operationen
 - wird üblicherweise von Entitätsklassen implementiert und besitzt einen ähnlichen Namen
 - Analysemodelle bestehen hauptsächlich aus Typen
- z.B. Person, Beitragszahler

Klassifizierung: primitive Klasse (engl. primitive) <<primitive>>

- ist eine elementare Klasse in einer Programmiersprache
 - besitzt keine eigenen Persistenzmechanismen
- z.B. `java.lang.String`

Klassifizierung: Datentyp oder Datenstruktur (engl. data type) <<data Type>>

- Klasse, die einfache Attribute oder primitive Klassen enthält
 - besitzt wenige Attribute und einfache Operationen
 - besitzt oft Methoden zur Umwandlung in andere primitive Klassen
 - z.B. getIntValue, asDate
- z.B. Geld, KundenNr, Dauer

Klassifizierung: Aufzählung (engl. enumeration) <<enumeration>>

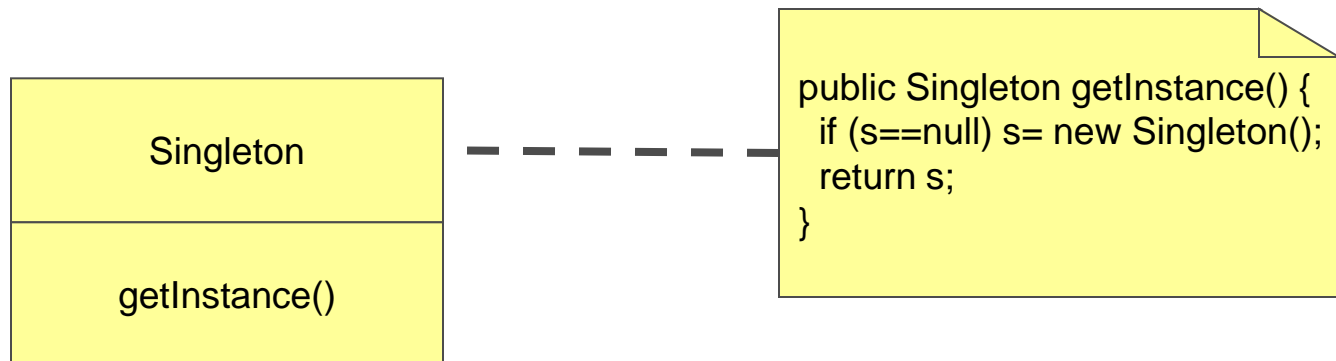
- ist eine aufzählbare Wertmenge
 - ist oft dynamisch änderbar
 - wird für die Deklaration von Attributen benötigt
 - besitzt eine frei definierbare Reihenfolge
 - in der Praxis oft als Folge von definierbaren int-Konstanten realisiert
 - z.B. true=42, false=0
(vgl. Douglas Adams: Per Anhalter durch die Galaxis)
- z.B. Entscheidung = {ja, nein, vielleicht}

Weitere Eigenschaften (Stereotype) von Klassen

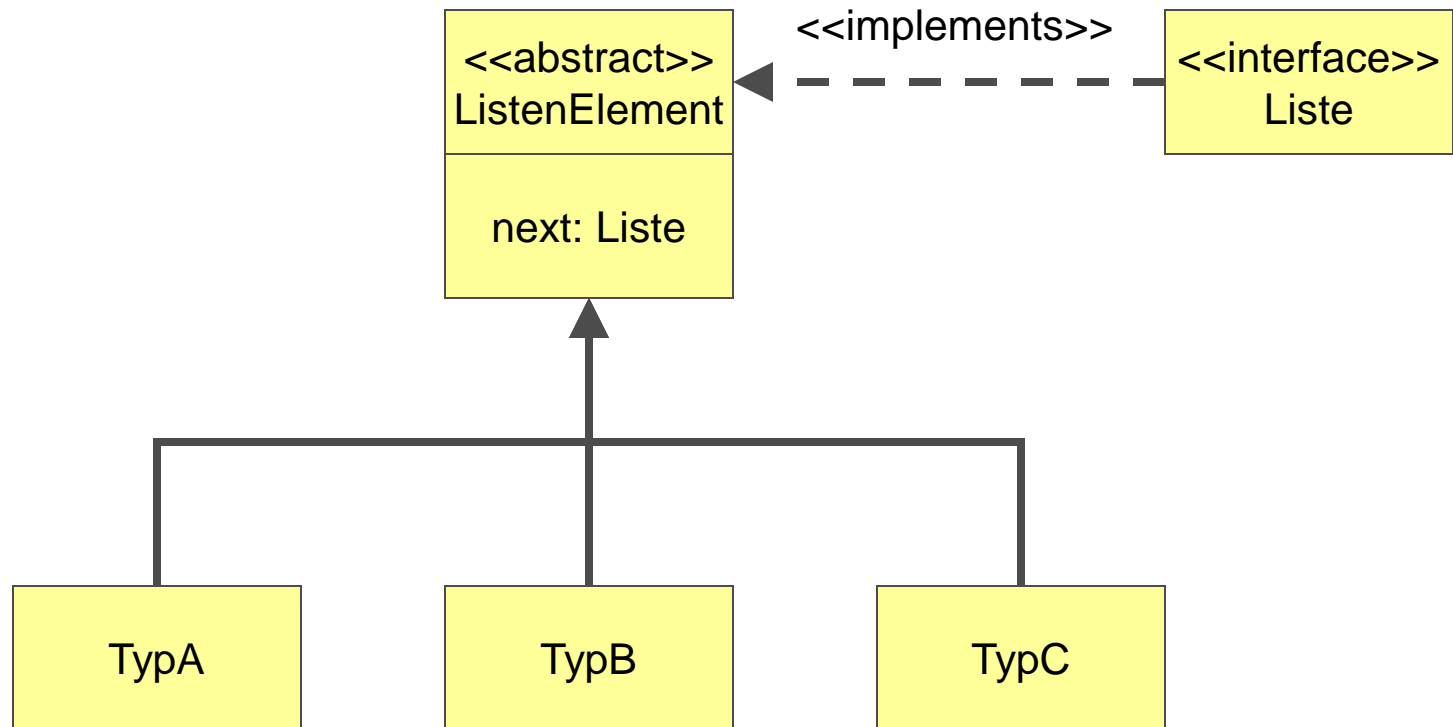
- <<abstract>>
 - definiert eine abstrakte Klasse
- <<focus>>
 - enthalten die primäre Programmlogik
- <<auxiliary>>
 - Hilfsklassen, die Fokusklassen unterstützen
- <<implements>>
 - kennzeichnet in einer Abhängigkeit, welche Klasse eine Schnittstelle implementiert

Notizen

- Notizen erlauben eine zusätzliche informelle Beschreibung einer Klasse, z.B.
 - mit einem Freitext
 - mit einem Quelltext einer konkreten Programmiersprache
- Notizen sehen aus wie Zettel mit Eselsohr.



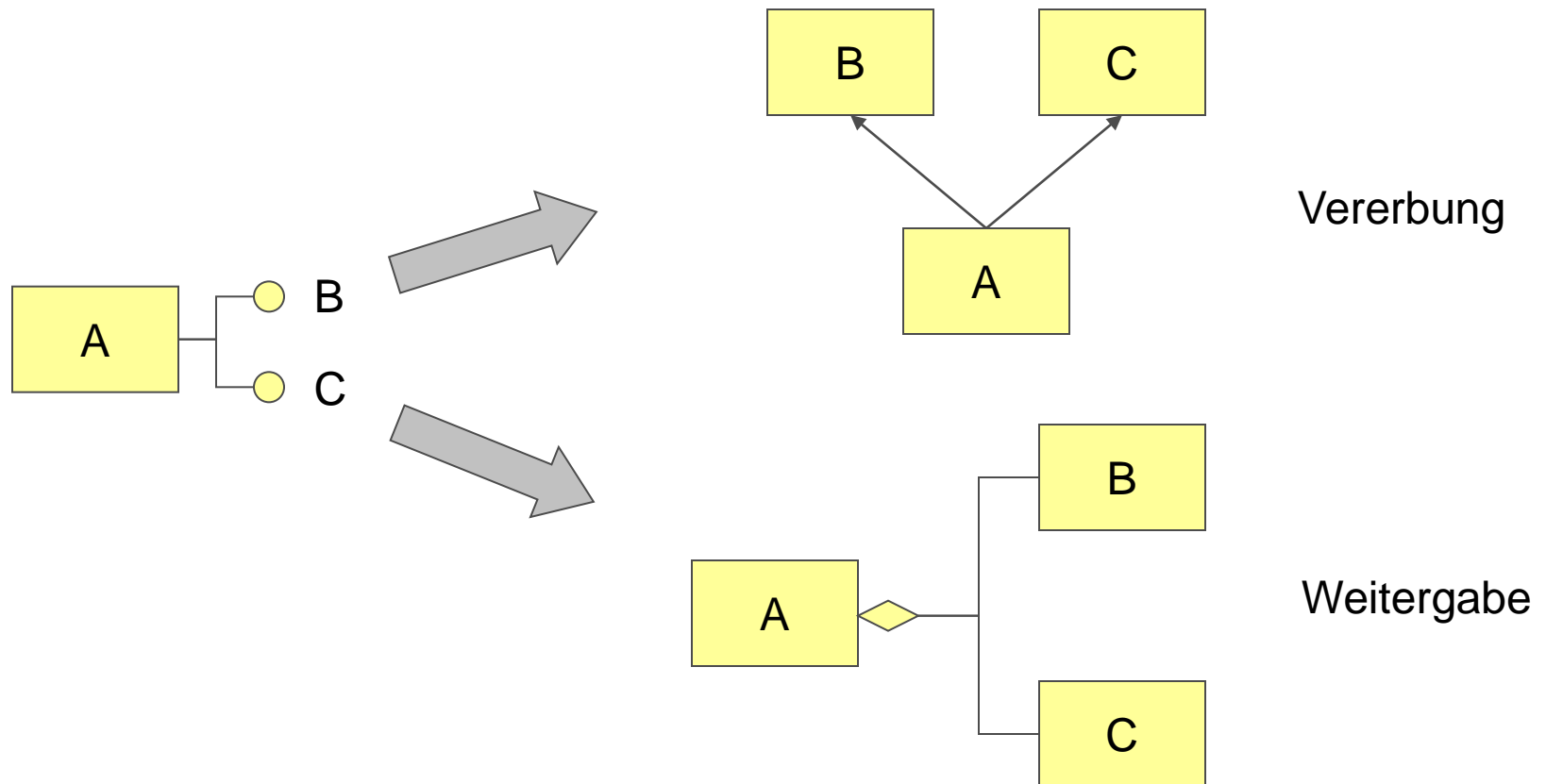
Beispiel für eine Schnittstellenimplementierung



Delegation

- Delegation erlaubt die Mitbenutzung von Eigenschaften anderer Klassen
 - durch Bildung einer Unterklasse (ggfs. mit Mehrfachvererbung)
 - durch die Implementierung einer Delegat-Schnittstelle
- Bildung von Unterklassen
 - Substitutionsprinzip beachten!

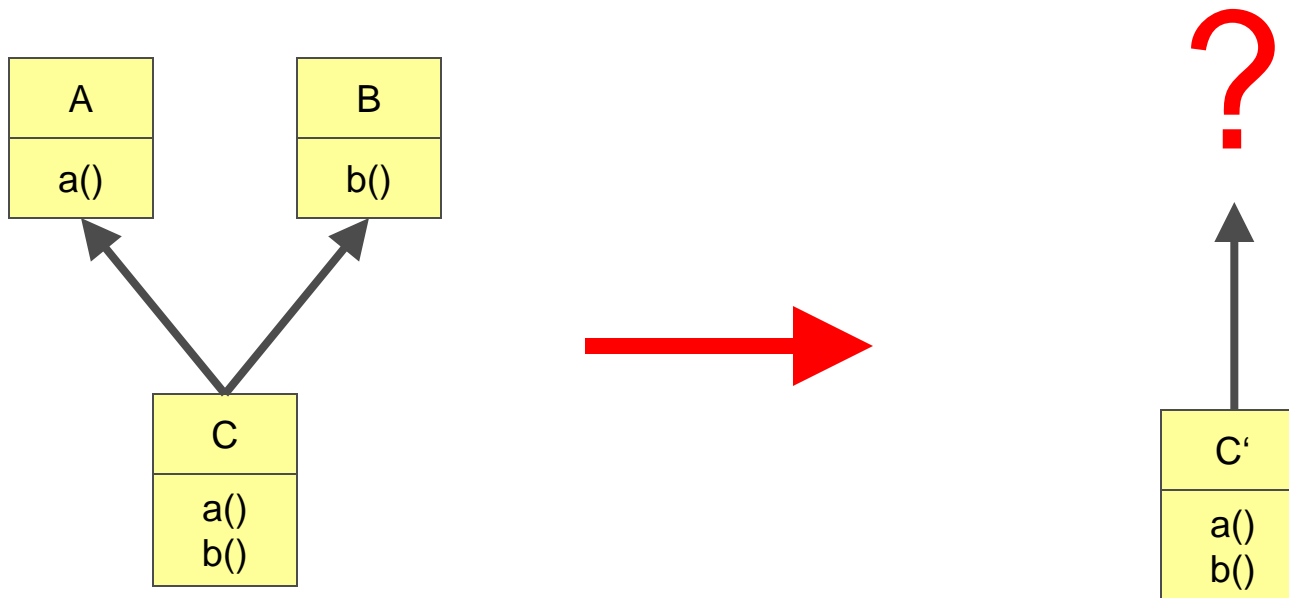
Delegation



Übung: Mehrfachvererbung

- Einige Programmiersprachen erlauben Mehrfachvererbung, einige nicht.
 - Mehrfachvererbung wird oft als schlechter Entwurfstil angesehen.
- Ihre Aufgabe:
 - Finden Sie ein Beispiel für die Problematik von Mehrfachvererbung.
 - Entwerfen Sie ein Klassenmodell, mit dem Sie Mehrfachvererbung in Einfachvererbung überführen können.

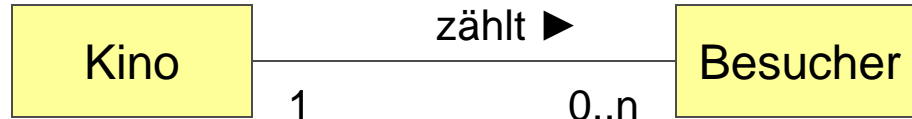
Übung: Mehrfachvererbung



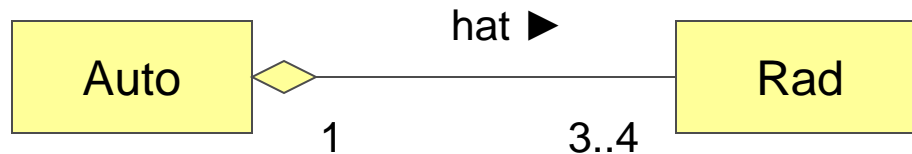
Assoziation, Aggregation und Komposition

- Eine Assoziation repräsentiert eine Beziehung zwischen Objekten einer oder mehrerer Klassen.
 - Kardinalität: eine Mengenangabe für Objekte:
z.B. 0, 1, 0..1, 0..n, 1..n
- Eine Aggregation ist ein Spezialfall der Assoziation.
 - definiert eine Beziehung in Form einer Teil/Ganzes Beziehung
- Eine Komposition ist ein Spezialfall der Aggregation
 - definiert eine existenzabhängige Teil/Ganzes Beziehung

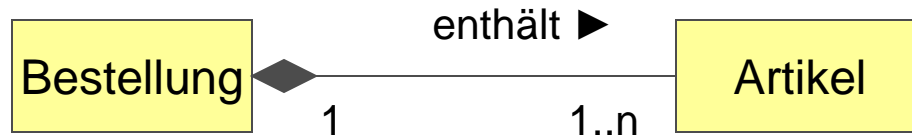
Assoziation, Aggregation und Komposition



Assoziation



Aggregation



Komposition

Klassendiagramm

- Ein Klassendiagramm beschreibt die Klassen eines Systems und ihre Beziehung zueinander.
 - Vererbungsbeziehungen
 - Assoziation, Delegation, Komposition
 - Abhängigkeiten
 - Schnittstellen
 - Kardinalitäten
 - Notizen

Objektdiagramm (engl. object diagram)

- besitzt eine ähnliche Struktur wie das Klassendiagramm
- zeigt für einen bestimmten Zeitpunkt existierende Objekte (nicht Klassen)
 - Objektdiagramm ist ein Schnappschuss des Systems zu einem bestimmten Zeitpunkt
- beschreibt insbesondere die Objektzustände mit ihren Attributwerten

Anwendungsfalldiagramm (engl. use case diagram)

- verwandte Begriffe
 - engl. use case model
 - Nutzungsfalldiagramm
- Ein Anwendungsfalldiagramm zeigt Akteure, Anwendungsfälle und ihre Beziehung zueinander.
 - zeigt, wie mit einem Anwendungsfall umgegangen werden soll
 - beschreibt keine Abläufe, sondern nur Zusammenhänge!
- spezielle Anwendungsfalldiagramme
 - Systemkontext-Diagramm

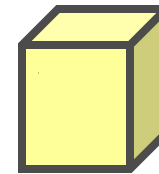
Akteur (engl. actor)

- verwandte Begriffe
 - engl. stakeholder
 - Beteiligter, Systemakteur, Ereignis, externes System, Dialog
- Akteure besitzen eine Multiplizität
 - Standardwert: 0..1 (wenn nicht explizit angegeben)
- Akteure sind in Anwendungsfalldiagrammen zu finden.

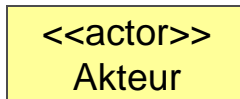
Akteure



menschlicher Akteur



Fremdsystem als Akteur



Akteur allgemein

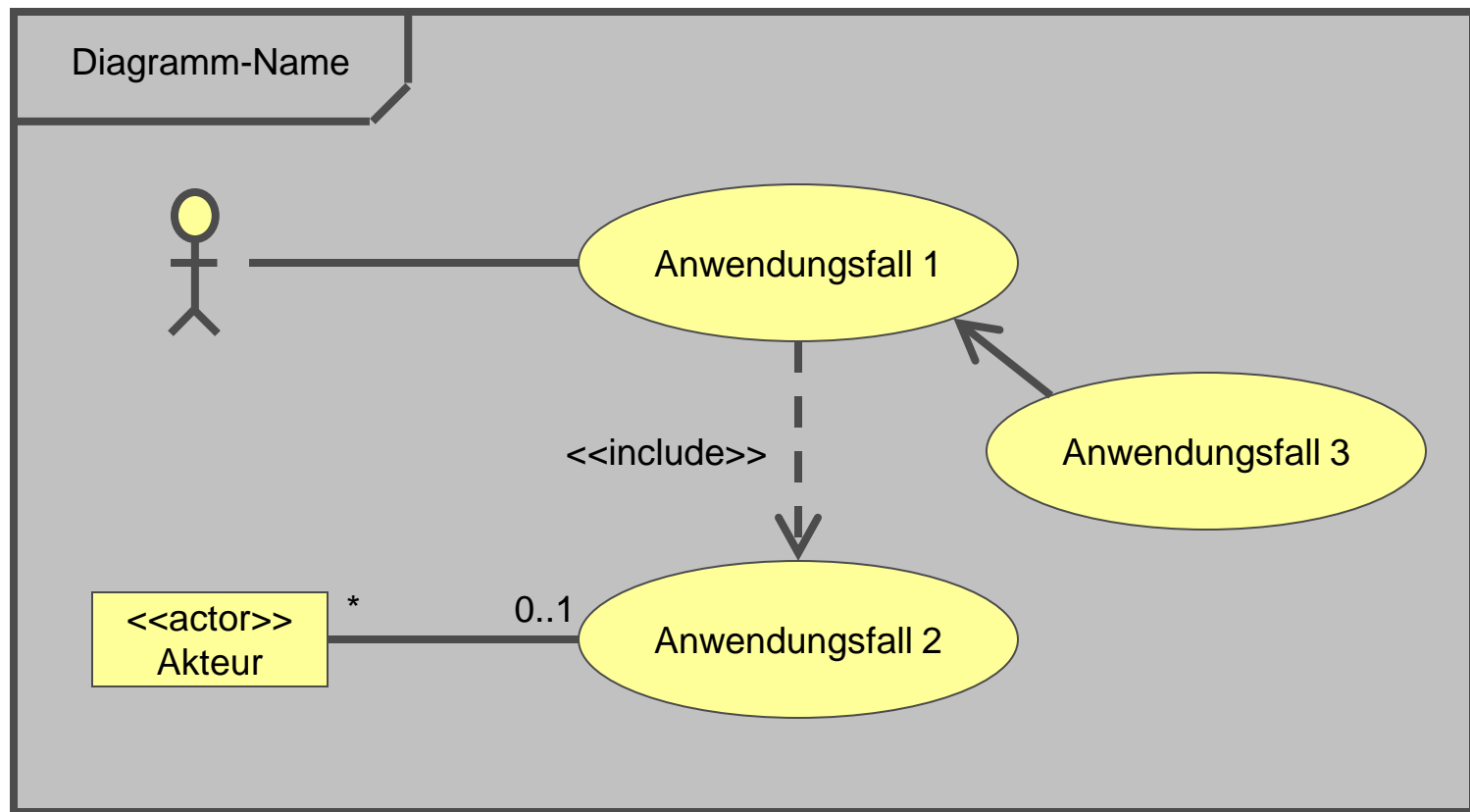


Zeit-Ereignis

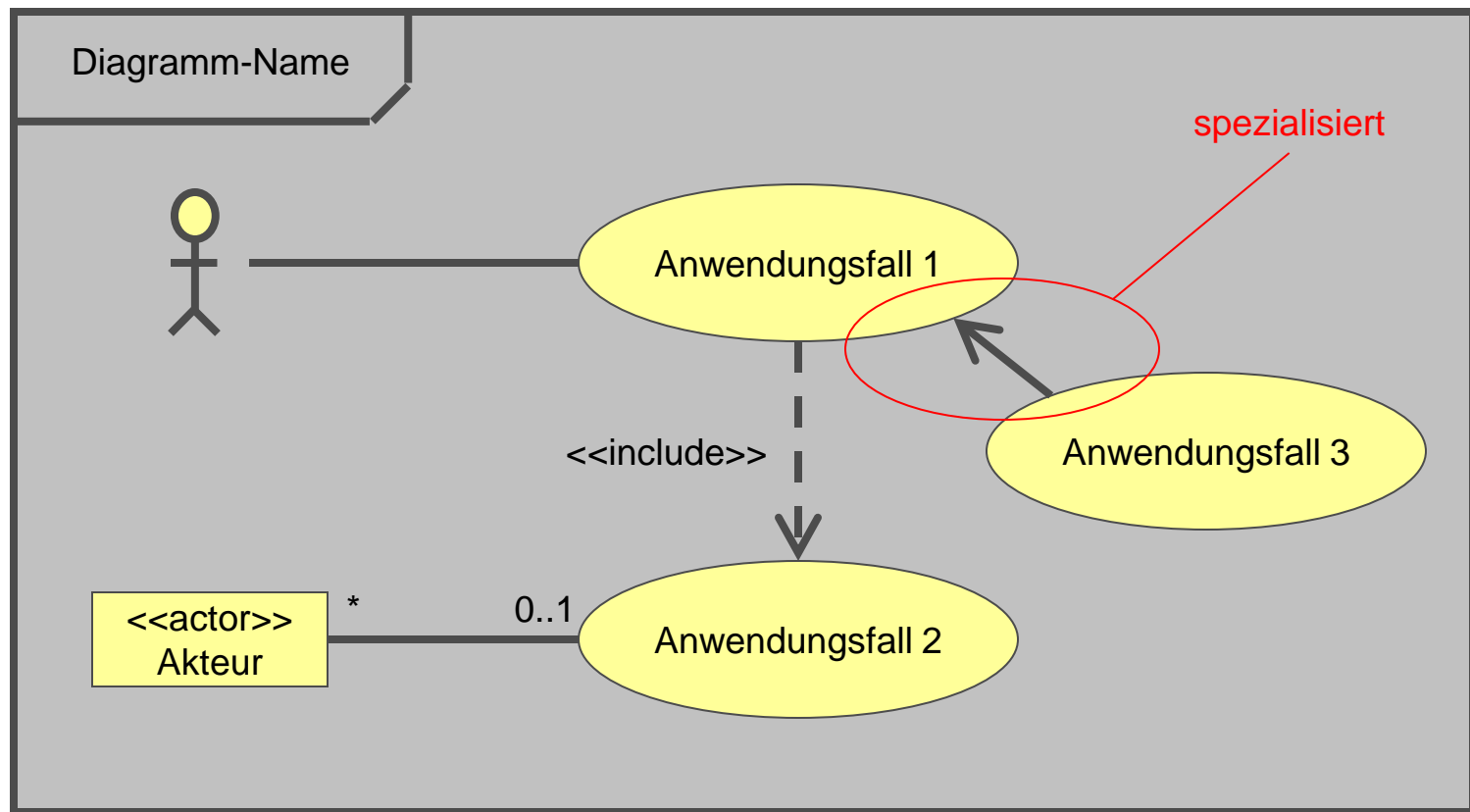
Anwendungsfalldiagramm

- spezielle Anwendungsfälle
 - Geschäftsfall
 - geschäftlicher Ablauf, der von einem geschäftlichen Ereignis ausgelöst wird und ein geschäftliches Ergebnis besitzt
 - Systemfall
 - Anwendungsfall, der speziell für außen stehende Akteure ein wahrnehmbares Verhalten beschreibt
 - abstrakter Anwendungsfall
 - Verallgemeinerung ähnlicher Anwendungsfälle
 - sekundärer Anwendungsfall
 - unvollständiger Teilablauf, der Bestandteil anderer Anwendungsfälle ist

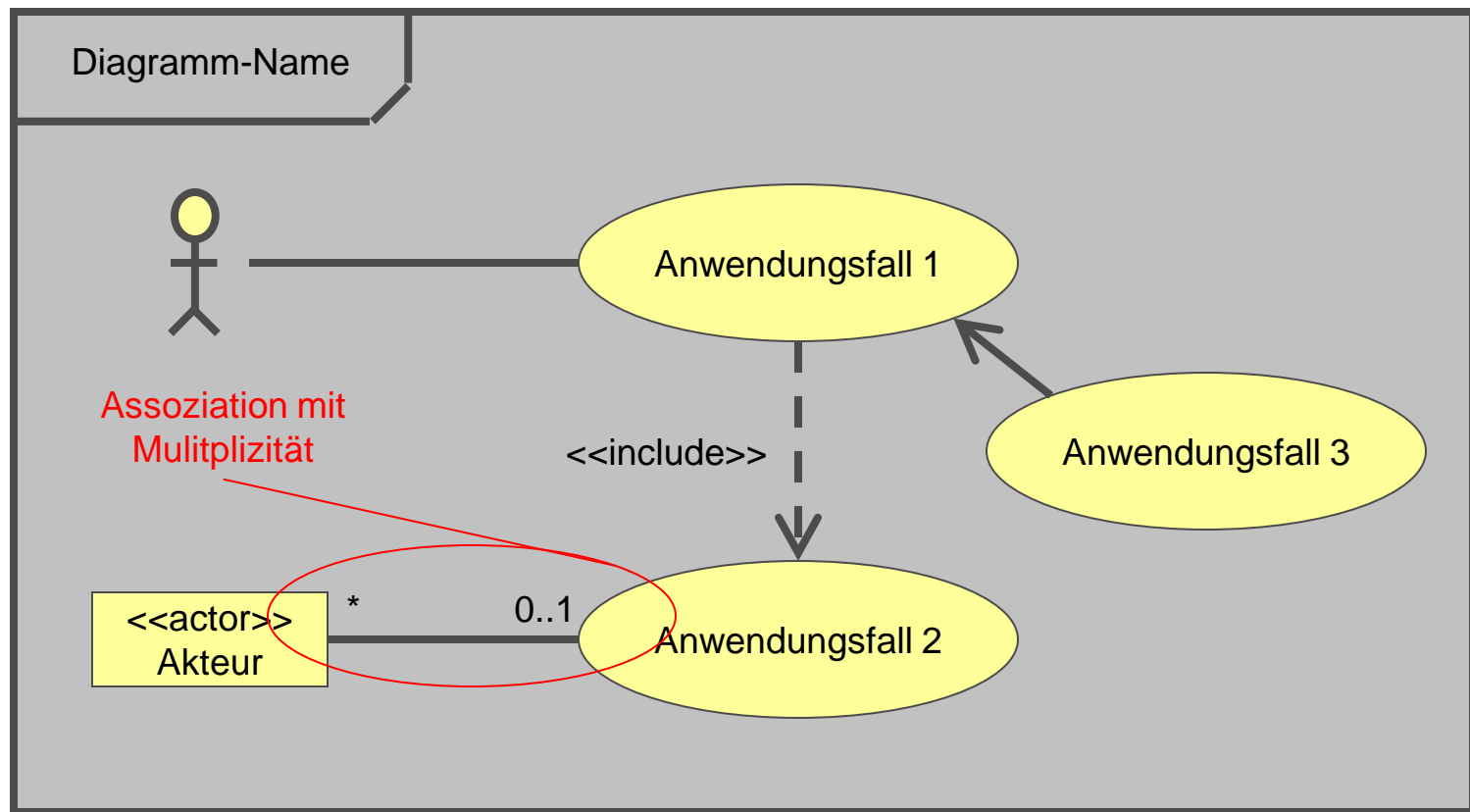
Anwendungsfalldiagramm



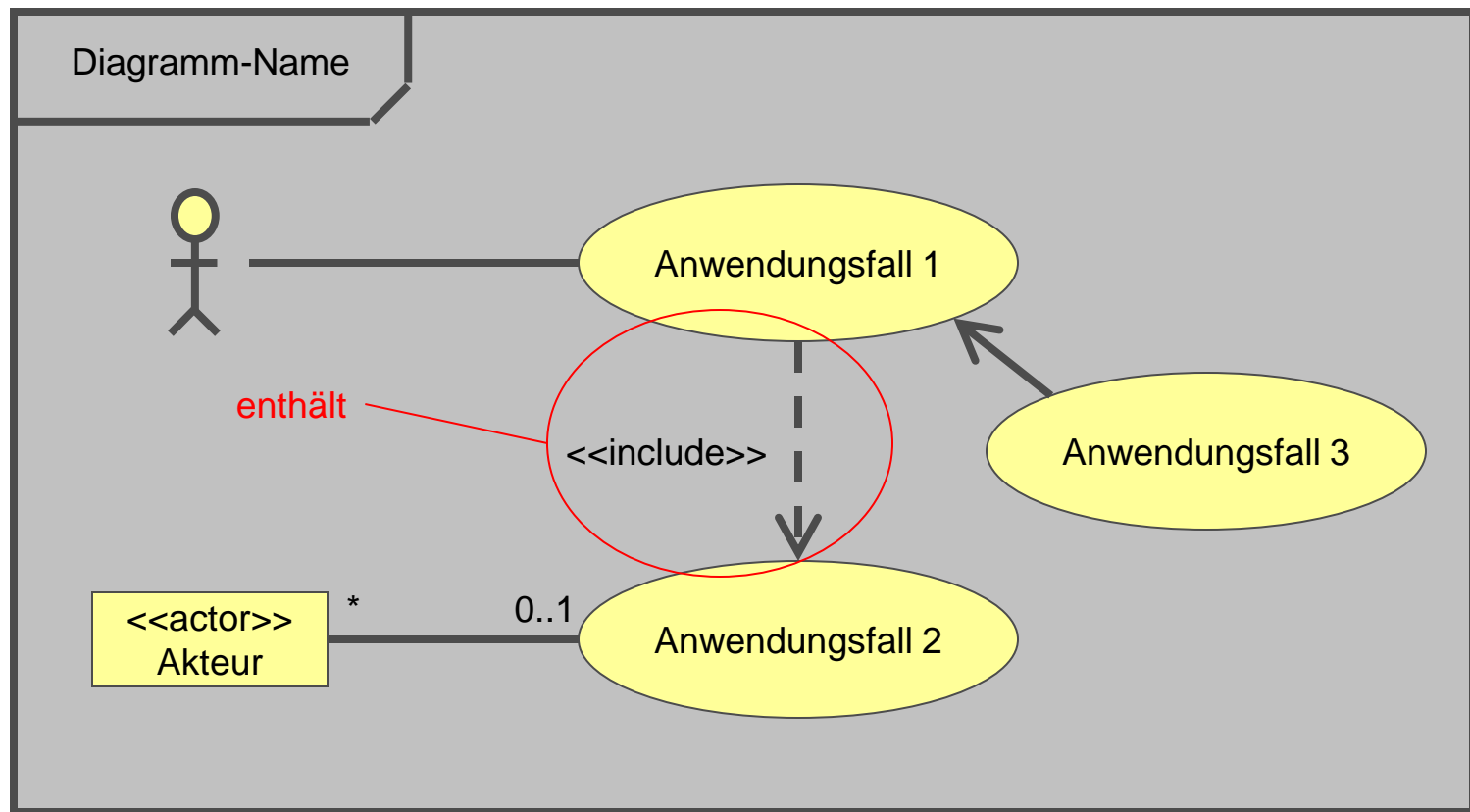
Anwendungsfalldiagramm



Anwendungsfalldiagramm



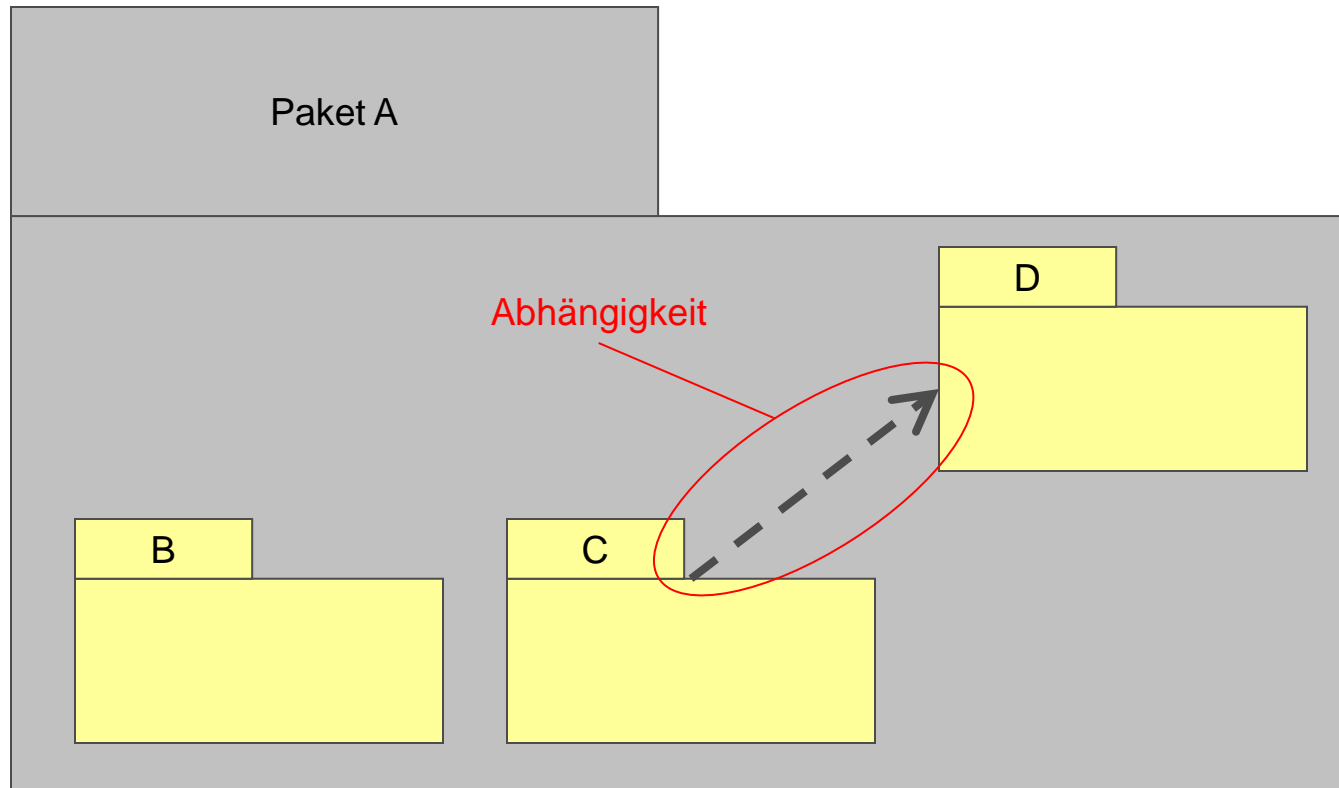
Anwendungsfalldiagramm



Paketdiagramm

- Paketdiagramme gliedern das System in überschaubare Einheiten
- Pakete können beliebige Modellteile beliebigen Typs enthalten
 - z.B. Klassen und Anwendungsfälle
- Pakete können Pakete enthalten und Abhängigkeiten besitzen
 - Abhängigkeit: ein Paket nutzt Klassen eines anderen Pakets

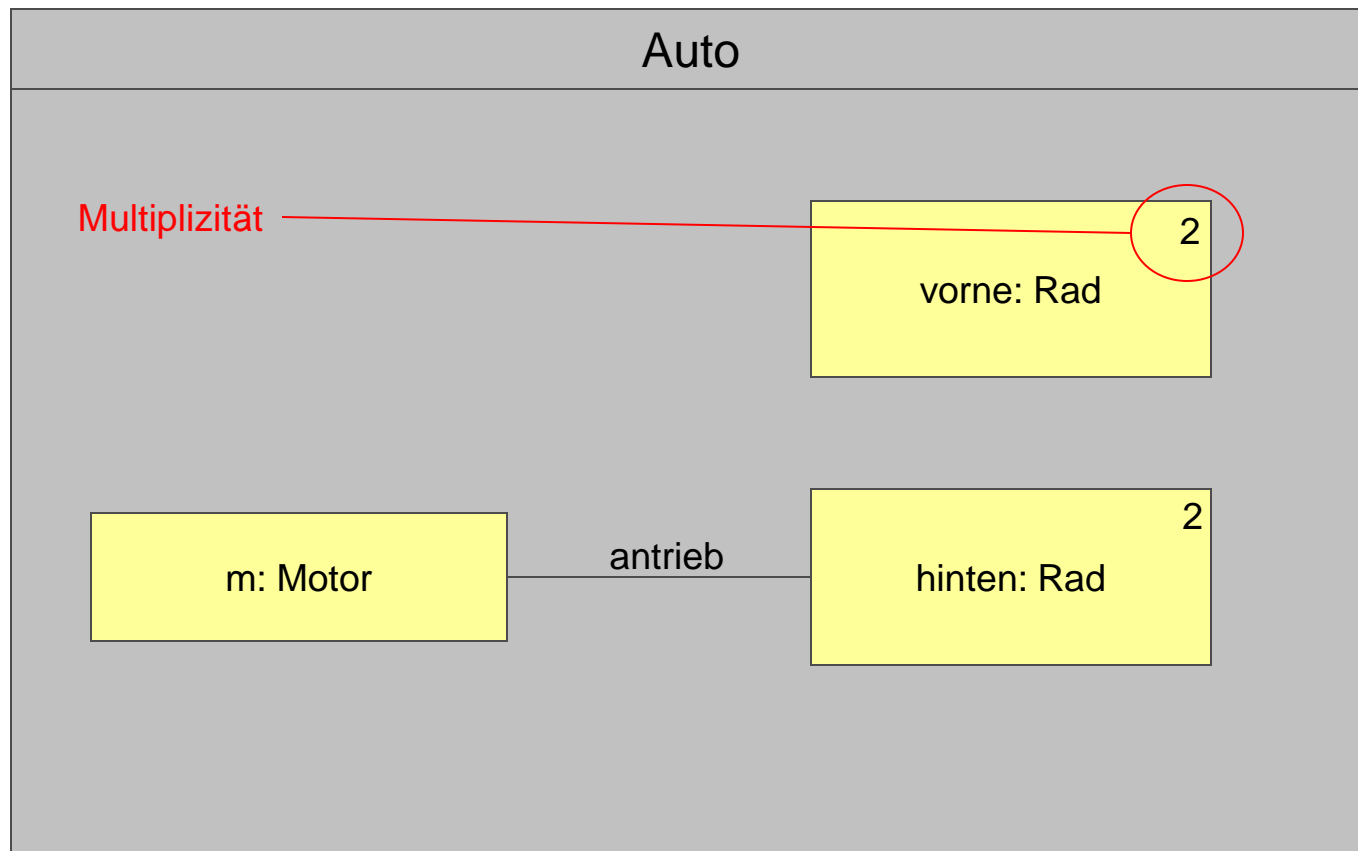
Paketdiagramm



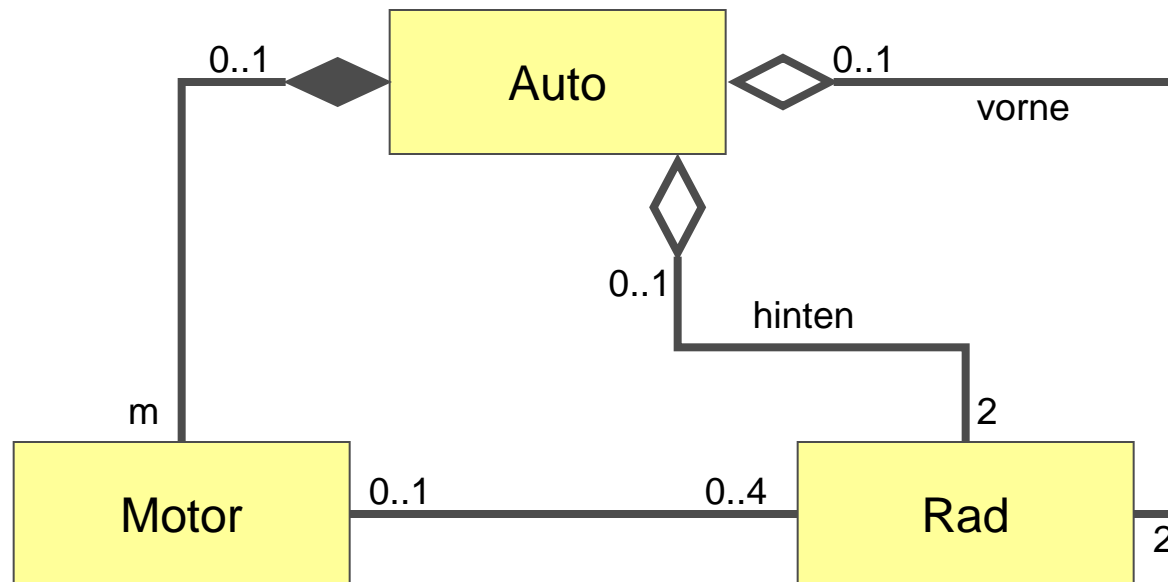
Kompositionsdiagramm

- zeigt die interne Zusammensetzung einer Komponente oder Klasse
 - Teile (engl. parts)
 - detaillierte Informationen über die Verbindung der Teile
- Warum reicht ein Klassendiagramm nicht aus?
 - Kardinalitäten von Kompositionen und Aggregationen sind zu wenig präzise.
 - Bei komplexen Klassendiagrammen können fachlich falsche Kompositionen entstehen.

Kompositionsdiagramm



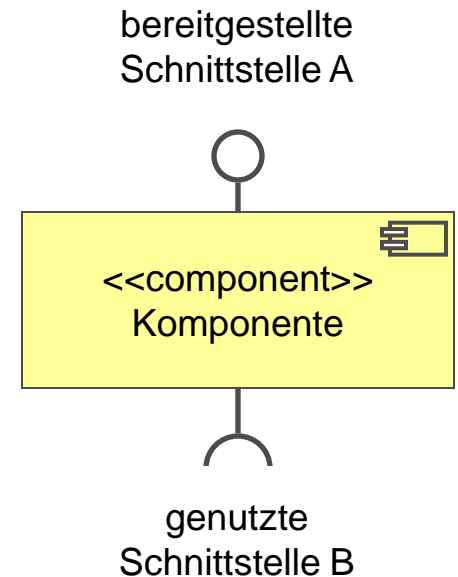
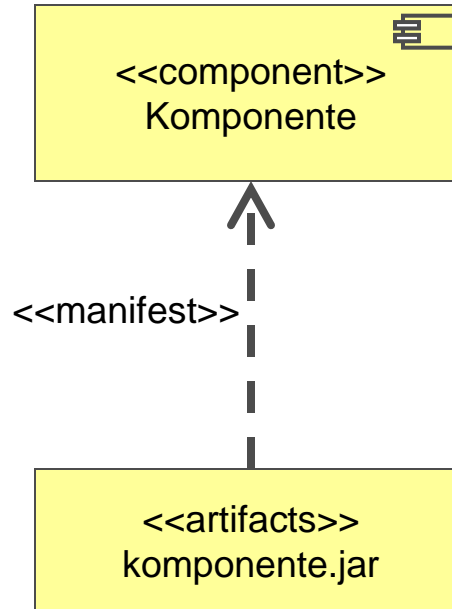
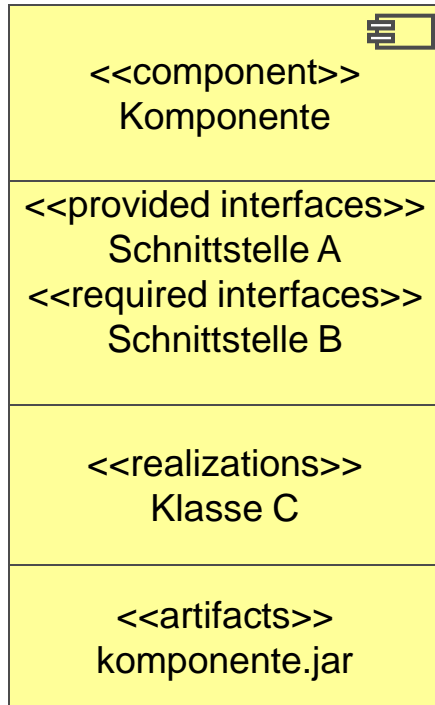
zum Vergleich: das Klassendiagramm



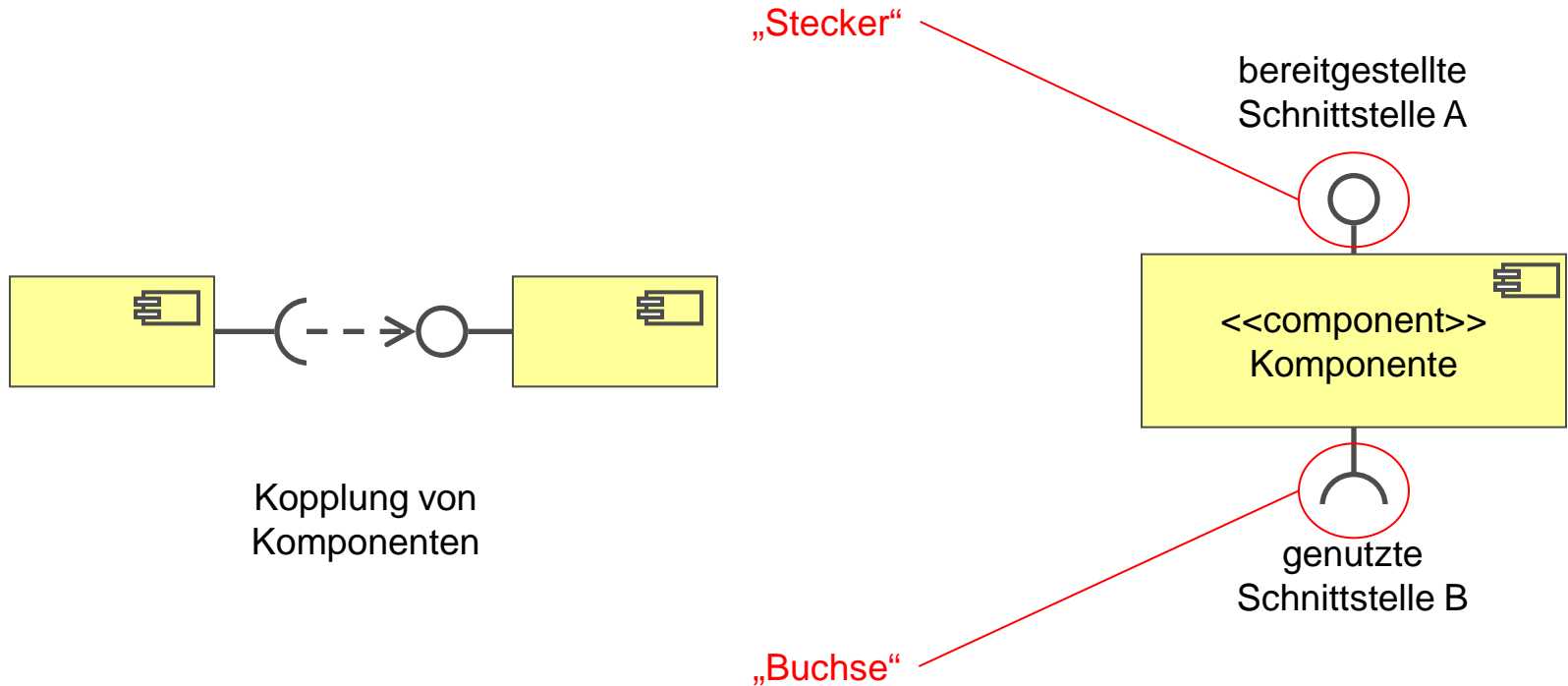
Komponentendiagramm

- Eine Komponente (engl. component) ist eine spezielle Klasse, die eine austauschbare Einheit in einem System repräsentiert, deren Bestandteile gekapselt sind.
 - Funktionalität wird über eine Schnittstelle (engl. interface) bereitgestellt
- Funktionalität, die von außen benötigt wird, wird ebenfalls über Schnittstellen definiert.

Komponentendiagramm: Notationen



Komponentendiagramm: Notationen

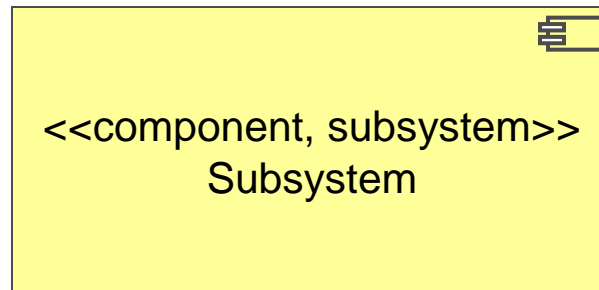


Subsystem-Diagramm

- Ein Subsystem ist eine Komponente, die eine architektonische Einheit darstellt.
 - besitzt die Merkmale einer Komponente
- Hierarchie in der Architektur

System → Subsystem → Komponente

Subsystem-Diagramm



Aktivitätsdiagramm (engl. activity diagram)

- verwandte Begriffe
 - engl. flow chart
 - Ablaufdiagramm, Programmablaufplan, Objektflussdiagramm
- besteht aus
 - Start- und Endknoten
 - Aktions-, Kontroll- und Objektknoten
 - Kontrollflüsse
- seit UML 2.0: Aktionen, Aktionsdiagramm

Aktivitätsdiagramm: Notationen



Startknoten



Kontrollfluss



Endknoten, normaler Ablauf

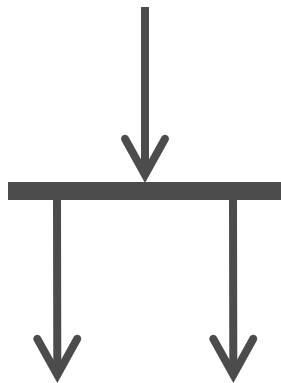
Aktivität
(=Aktion)



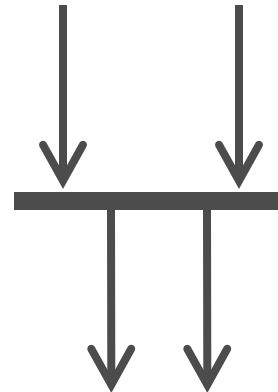
Ablaufende, z.B. Abbruch

Objekt

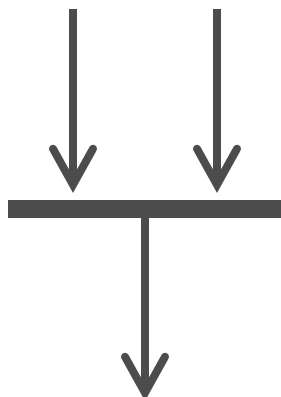
Aktivitätsdiagramm: Notationen



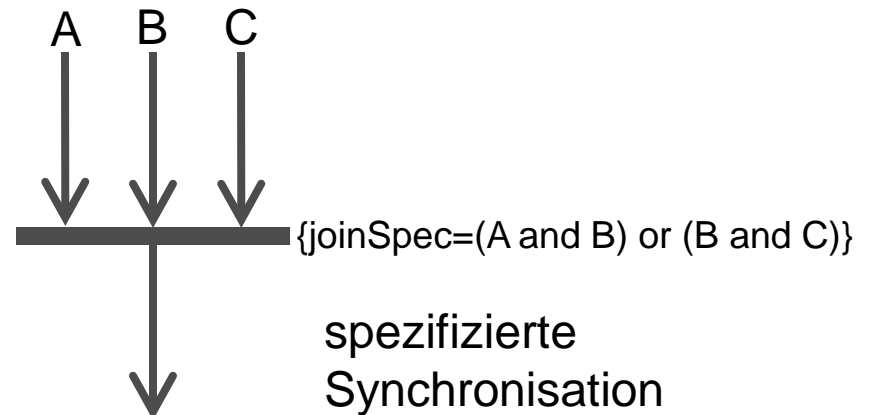
Teilung



Teilung und
Synchronisation

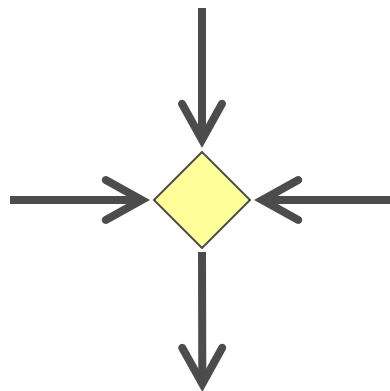
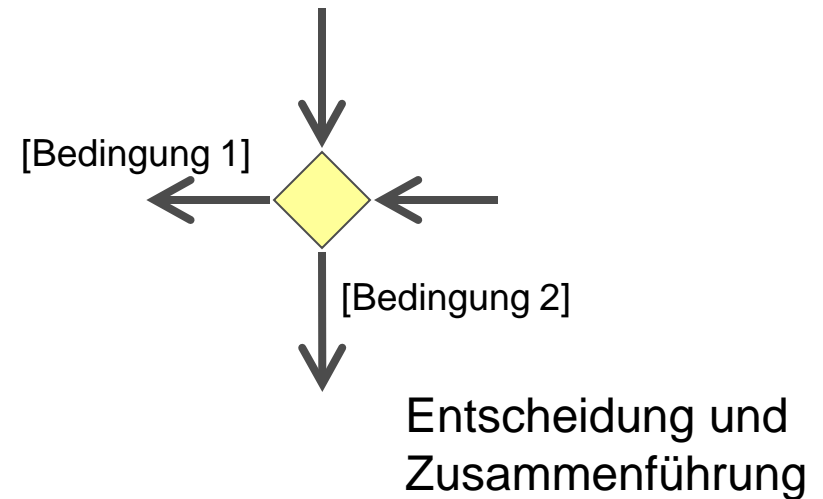
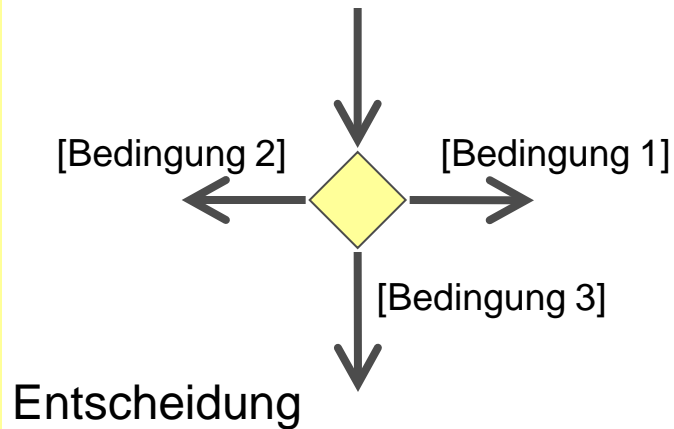


Synchronisation
(logisches UND)



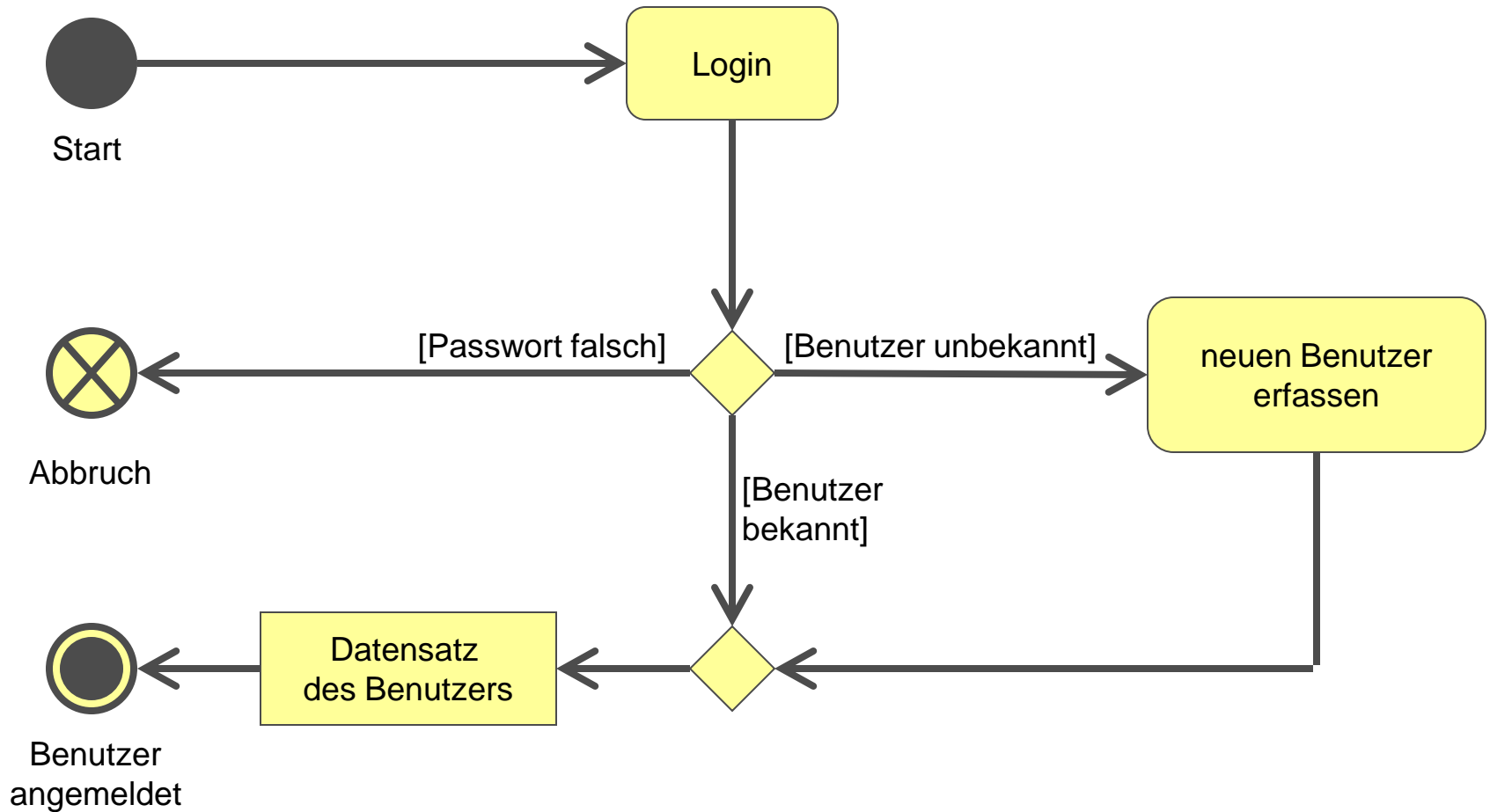
spezifizierte
Synchronisation

Aktivitätsdiagramm: Notationen



Zusammenführung (logisches ODER)

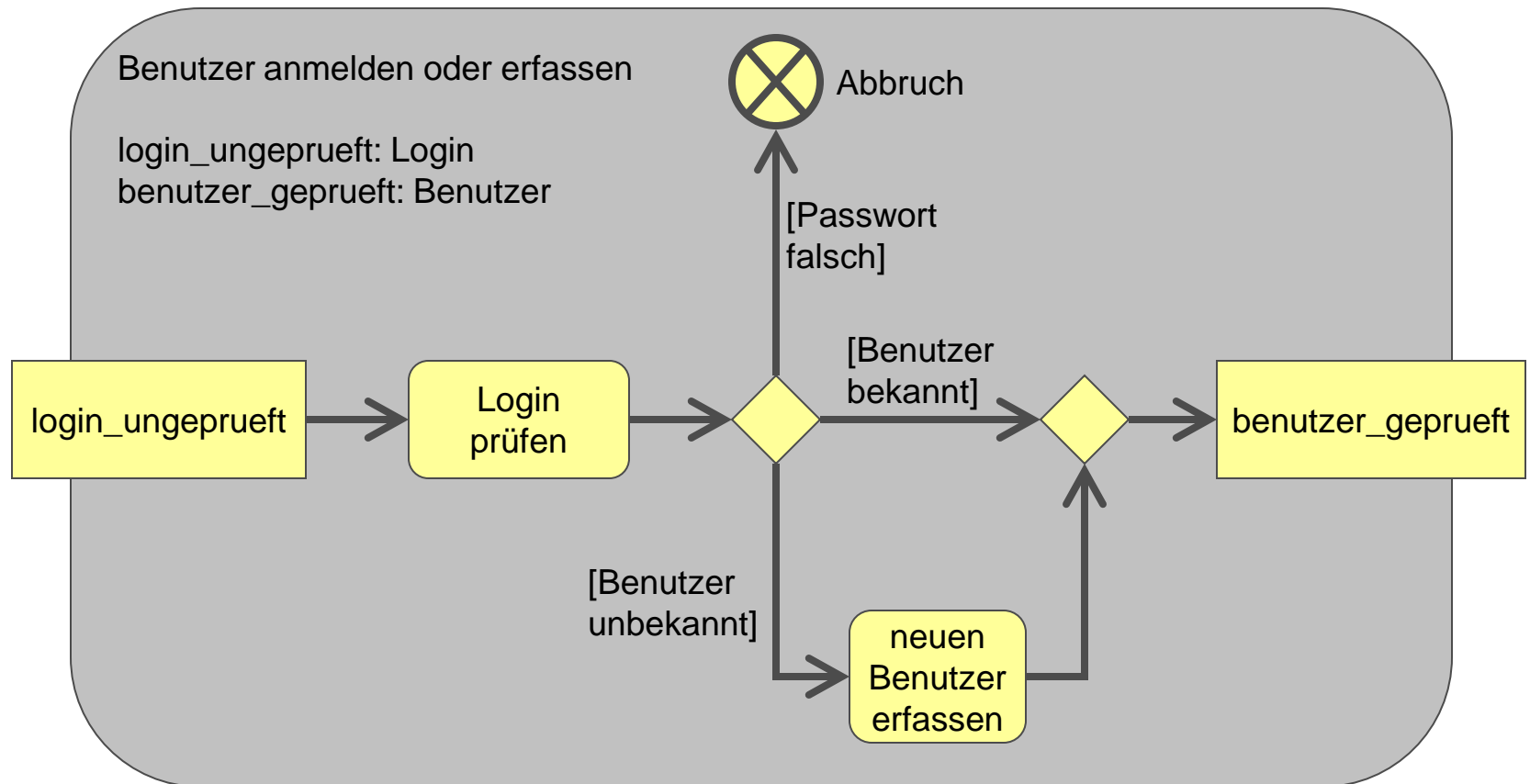
Beispiel: Benutzeranmeldung



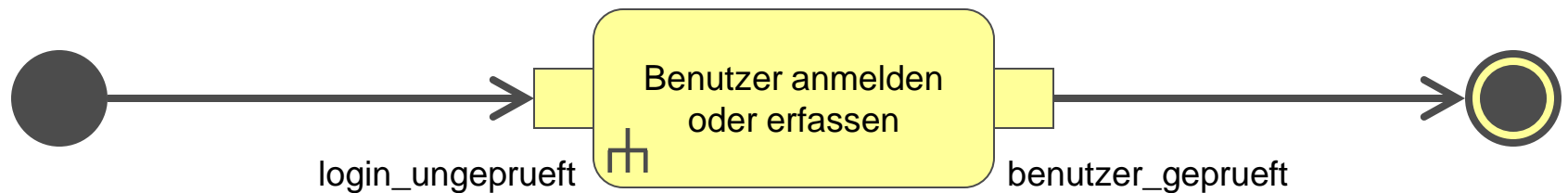
Übung

- Verfeinern Sie das Aktivitätsdiagramm der Benutzeranmeldung um folgende Eigenschaften:
 1. Ein Benutzer hat maximal 3 Anmeldeversuche. Nach dem dritten Fehlschlag wird das Konto gesperrt.
 2. Ein Administrator kann das Konto wieder entsperren.

Aktivitäten zusammenfassen



Aktivitäten zusammenfassen



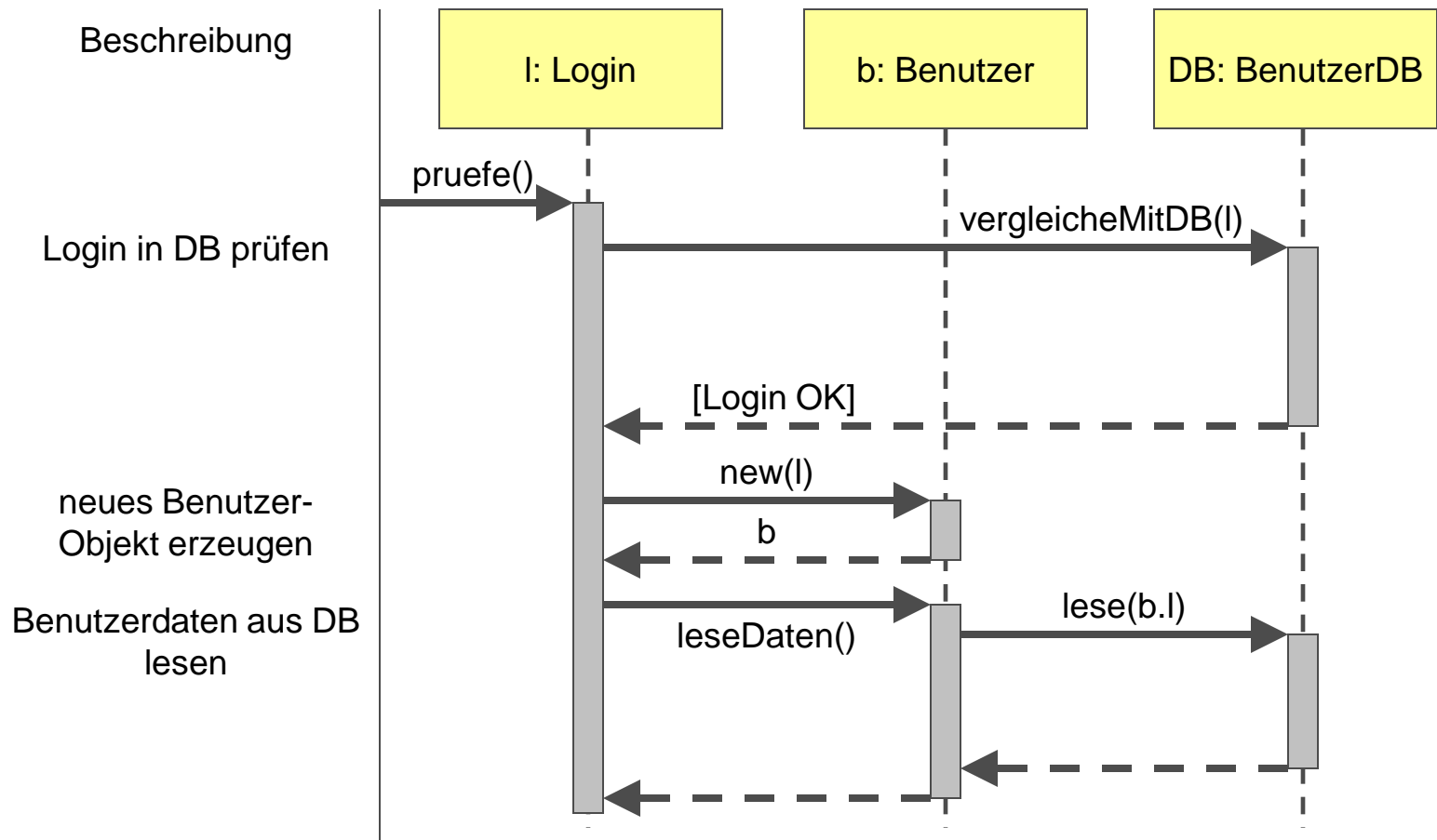
Zustandsdiagramm (engl. state diagram)

- Ein Zustandsdiagramm zeigt eine Folge von Zuständen, die ein Objekt im Laufe seines Lebens einnehmen kann und aufgrund welcher Stimuli Zustandsänderungen stattfinden.
- Ein Zustandsdiagramm ist äquivalent zu einem Zustandsautomaten (endlicher Automat)
- Elemente eines Zustandsdiagramms
 - endliche, nicht-leere Menge von Zuständen
 - endliche, nicht-leere Menge von Ereignissen
 - Zustandsübergänge
 - Anfangszustand
 - Menge von Endzuständen
- Notation
 - analog zu einem Aktivitätsdiagramm

Sequenzdiagramm (engl. sequence diagram)

- Eine Sequenz zeigt eine Reihe von Nachrichten, die eine ausgewählte Menge von Beteiligten (Rollen und Akteuren) in einer zeitlich begrenzten Situation austauscht, wobei der zeitliche Ablauf betont wird.
 - vergleichbar mit Kommunikationsdiagramm
 - Kommunikationsdiagramm zeigt die Zusammenarbeit der Rollen.
- Notation
 - Rollen werden durch gestrichelte senkrechte Linien dargestellt.
 - Oberhalb der Linie steht der Name bzw. das Rollensymbol.
 - Nachrichten werden als waagerechte Pfeile zwischen den Rollenlinien gezeichnet.
 - Nachrichten: `nachricht(argumente)`

Sequenzdiagramm



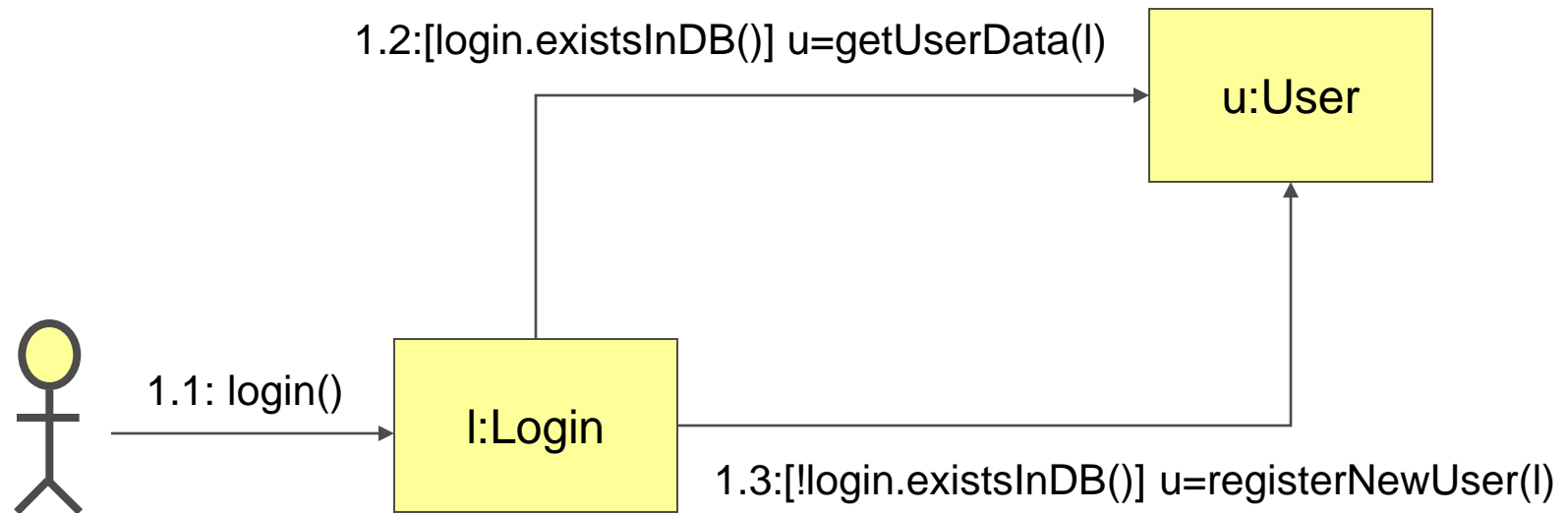
Kommunikationsdiagramm (engl. collaboration diagram)

- Ein Kommunikationsdiagramm zeigt eine Menge von Interaktionen zwischen ausgewählten Rollen in einer bestimmten begrenzten Situation (=Kontext) unter Betonung der Beziehungen zwischen den Rollen und ihrer Topographie.
 - zeitliche Abfolge der Nachrichten und Antworten werden dargestellt
 - auch in Form von Iterationen und Schleifen
- Ein Kommunikationsdiagramm ist eine Projektion des dahinter stehenden Gesamtmodells und ist zu diesem konsistent.

Kommunikationsdiagramm (engl. collaboration diagram)

- Unterschied zum Sequenzdiagramm
 - Iterationen und Schleifen sind leichter darstellbar
 - Mehrere Kommunikationspfade (= mehrere Sequenzen) können durch geeignete Nummerierung dargestellt werden.
- Gemeinsamkeit mit Sequenzdiagramm
 - beide Diagramm können Ablaufvarianten beschreiben
 - beide Diagramm eignen sich in der Praxis nicht zur vollständigen Beschreibung des Systemverhaltens
 - zeitlicher Verlauf der Kommunikation zwischen den Rollen steht im Vordergrund
 - Nachrichten werden im Diagramm nummeriert

Beispiel für ein Kommunikationsdiagramm



Interaktionsübersicht

- Eine Interaktionsübersicht ist ein Aktivitätsdiagramm, in dem Teilabläufe durch referenzierte oder eingebettete Sequenzdiagramme repräsentiert sind.
- Die Interaktionsübersicht hilft, eine Menge von Sequenzdiagrammen mit Hilfe des umgebenden Aktivitätsdiagramms in einen zeitlogischen Kontext zu setzen.

Gruppenarbeit: SysML

- Die SysML (Systems Modeling Language) ist ein Subset der UML, bietet aber auch neue Diagrammtypen.
- SysML wurde für die ingenieurmäßige Entwicklung von Systemen, z.B. für Steuergeräte in der Fahrzeugkonstruktion entwickelt.
- Beschreiben Sie die Möglichkeiten zur Modellierung mit der SysML. Recherchieren Sie dafür in der gängigen Literatur bzw. im Internet.
- Weitere Informationen: <http://www.sysml.org>

Selbstkontrolle

1. Warum kann es notwendig sein, zusätzlich zu Klassendiagrammen auch Objektdiagramme zu modellieren?
2. Welche Akteure kennen Sie?
3. Wie funktionier Ihrer Meinung nach Mehrfachvererbung, wenn zwei Oberklassen eine Operation mit gleicher Signatur vererben?
4. Warum wird ein Kompositionsdiagramm benötigt?
5. Warum sollten Sequenzdiagramme nicht für jede Operation modelliert werden?

Kontakt

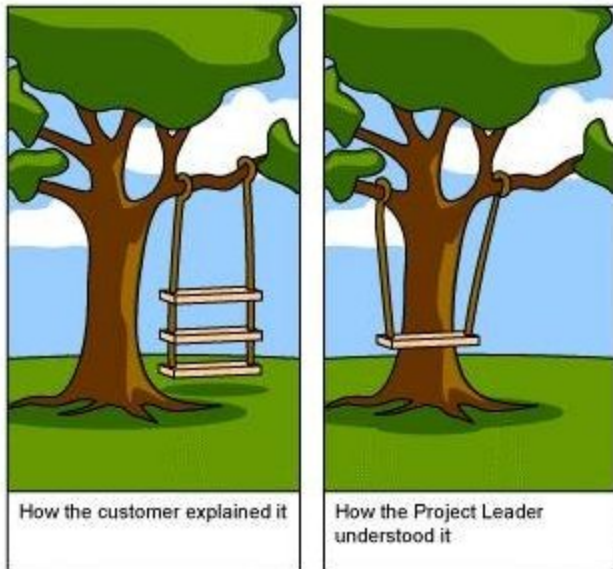
Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Software Engineering 1

Implementierung



Prof. Dr. Andreas Judt

Aufgabe der Implementierung

- Aufgaben:
 - Konzeption von Datenstrukturen und Algorithmen
 - Strukturierung des Programms durch Verfeinerung des Entwurfs
 - unter Beibehaltung von Schnittstellen und Architektur
 - Umsetzen von Abläufen in die gewählte Programmiersprache
- Ergebnis:
 - dokumentiertes Quellprogramm
 - ablauffähiges Programm (Objektcode), das aus dem Quellprogramm erzeugt wurde
 - ggfs. Installationspaket, z.B. InstallShield oder Microsoft Installer (MSI)

Testen: eines der größten Missverständnisse

- einige Lehrbücher propagieren, dass der Programmierer in der Implementierungsphase testet!
 - vgl. Balzert, Bd. 1, Kap. 4.1, Seite 1065
- Entscheidend für die Qualität des Softwaretests: Tester darf nicht an der Programmierung beteiligt sein!

Warum ???

- Bemerkung: Testen wird in einem eigenständigen Kapitel behandelt.

Was ist ein gutes Programm?

- Softwarequalität generell:
 - Softwarequalität ist kaum messbar.
- Softwarequalität plausibel:
 - Ein gutes Programm kann von anderen Programmierern verstanden und weiterentwickelt werden.
- Faustregel für die Aufwandsabschätzung:
 - Die Einarbeitung in ein fremdes Programm ist so aufwendig wie eine Neuentwicklung!
- guter Programmierstil:
 - aussagekräftige Namensgebung
 - kurze Klassen und Methoden
 - gute Dokumentation im Quellcode
 - beschreibender Programmkopf

Beispiel: Programmierrichtlinie für Java

- vollständige Richtlinie <http://java.sun.com/docs/codeconv>
- Beispiel maximaler Länge einer Quelldatei: 2000 Zeilen
- Beispiel Kopfdeklaration für eine Klasse:

```
/*  
 * Classname  
 *  
 * Version information  
 *  
 * Date  
 *  
 * Copyright notice  
 */
```

API Dokumentation mit Javadoc

- API = engl. application programming interface
- Mit Javadoc-Kommentaren können API-Dokumentationen von Klassen, Funktionsbibliotheken und Rahmenwerken automatisch aus dem Quellcode als HTML generiert werden.

API Dokumentation mit Javadoc

- Beispiel:

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

Vorschlag für Versionsnummern

- 3 Zahlen, durch Punkt getrennt
 - [Hauptversion].[Nebenversion].[Revision]
- Änderung der Revision
 - Schnittstellen und Funktionalität ändern sich nicht!
 - bei Fehlerbehebungen
 - bei internen Änderungen

Vorschlag für Versionsnummern

- Änderung der Nebenversion
 - Schnittstellen und Funktionalität können sich geändert haben
 - sollten abwärtskompatibel sein: Dokumentation beachten!
- Änderung der Hauptversion
 - grundlegende Überarbeitung von Funktionalität und Schnittstelle
 - üblicherweise stark überarbeitete Softwarearchitektur
 - nicht abwärtskompatibel

Psychologie des Programmierens: Verhaltens- und Denkmodelle

- Scheinwerferprinzip
 - bei einem hohen Informationsfluss werden nur wichtige Teile aufgenommen und verarbeitet
 - Effekt: Ausnahmen, Initialisierung und Grenzfälle werden vergessen
- Sparsamkeits- = Ökonomieprinzip
 - Ziele werden mit möglichst geringem Aufwand erreicht
 - Effekt: Lösung ist nicht ausreichend gut bzw. ineffizient

Psychologie des Programmierens: Verhaltens- und Denkmodelle

- **Strukturerwartungen = Denken in Kategorien**
 - Kategorien sollen Ordnung in eine unübersichtliche Welt bringen und strukturiertes Denken erleichtern
 - Effekt: außergewöhnliche Situationen führen zu Irrtümern, unpassenden Vorurteilen und Fehlverhalten
- **Kausalitätserwartung = lineares Ursache-Wirkung Denken**
 - übermäßige Vereinfachung komplexer Sachverhalte
 - Erscheinungen wird oft eine einzige Ursache zugeordnet
 - Effekt: komplexe Entscheidungen werden falsch getroffen

Psychologie des Programmierens: Verhaltens- und Denkmodelle

- induktive Hypothesen
 - im Besonderen wird das Allgemeine und im Vergangenen das Zukünftige erkannt
 - Effekt:
 - Überschätzung bestätigter Informationen, z.B. bei schwachen Tests
 - Gesetzmäßigkeiten werden vorschnell als gesichert angesehen.
 - Schwache Lösungen werden vom Programmierer als optimal angesehen, nur weil sie funktionieren.

Psychologie des Programmierens: Verhaltens- und Denkmodelle

- falsche Lösung
 - es wird keine Lösung gefunden, ob wohl eine existiert
 - es wird eine mangelhafte Lösung gefunden, die weit vom Optimum weg ist
- falsche Einstellung zum Problem
 - durch Erfahrungen bei früheren Problemen in ähnlichen Situationen
 - durch wiederholte Anwendung eines mechanisierten Denkschemas
 - Vermutung von Vorschriften, die nicht existieren
 - Lösung eines nahen Teilziels und Nichtverfolgen der komplexeren Gesamtlösung

Typische Programmierer-Fehler

- unnatürliche Zahlen
 - Negative Zahlen werden oft falsch behandelt.
- Ausnahmen und Grenzen
 - Normalfälle werden bevorzugt behandelt. Sonderfälle und Ausnahmen werden oft übersehen oder nur erfasst, wenn sie repräsentativ sind.
- Tücken der Maschinendarstellung
 - Zahlendarstellung im Rechner ist begrenzt und auf dem Zahlenstrahl an verschiedenen Stellen unterschiedlich präzise.
 - Bsp: $x==y \Leftrightarrow x-y < \text{epsilon} ?$

Typische Programmierer-Fehler

- unvollständige Bedingungen
 - Oft ist die Software nicht so komplex wie das zu lösende Problem.
- überraschende Variablen-Werte
 - Die Komplexität von Zusammenhängen wird nicht erfasst.
- wichtige Nebensachen
 - Vernachlässigung von Programmteilen führt zu Teilprogrammen geringer Qualität.
- alte Kommentare und Dokumentation
 - Weder Programmkommentare noch Dokumentation werden bei Änderungen aktualisiert.

Beispiel: Patriot-Raketen-Fehler

- Quelle:
 - <http://www.math.psu.edu/dna/disasters/patriot.html>
- Situation:
 - Während des Golfkriegs verfehlte am 25.2.1991 in Saudi-Arabien eine amerikanische Patriot-Rakete eine irakische Scud-Rakete. Die Scud-Rakete traf eine Kaserne und tötete 28 Soldaten.
- Ursache:
 - Ungenaue Berechnung der Zeit seit Systemstart wegen Rundungsfehler. Die Interne Uhr zählte in zehntel Sekunden.



Beispiel: Patriot-Raketen-Fehler

- Umrechnung:

- Verstrichene Zeit mit $1/10$ multipliziert, um die verstrichene Zeit in Sekunden zu berechnen. Dabei ist

$$1/10 = 2^{-4} + 2^{-5} + 2^{-8} + 2^{-9} + 2^{-12} + 2^{-13} + \dots$$

- Als binäre Festkommazahl mit 24 Stellen also

0.00011001100110011001100

- Rundungsfehler:

$$2^{-25} + 2^{-26} + \dots \approx 0.000000095 = 9.5 \cdot 10^{-8}$$

Beispiel: Patriot-Raketen-Fehler

- Effekt:

- Nach 100 Betriebsstunden $100 \cdot 60 \cdot 60 \cdot 10 = 3.6 \cdot 10^6$ ergibt sich als Fehler (Zeitdifferenz) :

$$9.5 \cdot 10^{-8} \cdot 3.6 \cdot 10^6 = 0.34 \text{ Sekunden}$$

- Scud-Geschwindigkeit $1.676 \text{ m/s} = 6034 \text{ km/h}$
- Scud in der Zeit ca. $0.57 \text{ km} = 570 \text{ m}$ weiter.
- damit außerhalb der Reichweite des „Aufspürsystems,“ der Patriot

Abnahme eines Softwareprojekts

- Aufgaben in der Abnahmephase
 - Übergabe
 - Gesamtprodukt mit Dokumentation, falls individuelle Software erstellt wurde
 - Installationspaket und Handbuch bei Vertrieb von lizenzierter Software
 - Abnahmetest und Stresstest
- Ergebnis: Abnahmeprotokoll
 - Abnahme durch den Auftraggeber ist eine schriftliche Erklärung zur Annahme des Produkts im juristischen Sinn.
- Wichtig:
 - Der Auftraggeber kann in der Abnahmephase nicht nachvollziehen, ob die Software tatsächlich getestet wurde.

Einführung in den Produktivbetrieb

- Aufgaben in der Einführungsphase
 - Installation des Produkts
 - Schulung von Anwendern und Administratoren
 - Inbetriebnahme des Produkts
- Ergebnis: Einführungsprotokoll
 - Produkteinführung ist immer eine Innovation!
- Problem:
 - Umstellung von einem bestehenden Altsystem

Umstellung vom Altsystem

- direkte Umstellung
 - Altsystem wird abgeschaltet, Neusystem in Betrieb genommen
 - höchstes Risiko, endet in der Praxis oft in einer Katastrophe
- Parallellauf
 - beide Systeme laufen produktiv und können miteinander verglichen werden
 - funktioniert nicht beim gemeinsamen Zugriff auf eine Datenbank!

Umstellung vom Altsystem

- Versuchslauf
 - versuchsweise Inbetriebnahme des Neusystems mit Daten des Altsystems
 - Anwender soll Fehler melden
 - sukzessiver Austausch von Modulen des Altsystems mit Modulen des Neusystems
 - enorm hohes Risiko, da Schnittstellengleichheit vorausgesetzt wird
- Pilotinstallationen
 - sog. Betatest
 - Vorab-Installationen, Fehler werden (automatisiert) zurückgemeldet
 - z.B. Entwicklung von Mozilla bzw. Firefox

Betrieb des neuen Systems

- Aufgaben während des Betriebs
 - Wartung
 - Fehlerbehebung
 - Leistungsverbesserung
 - Pflege
 - Änderungen
 - Erweiterungen
- Bemerkung:
 - auf 10 gefundene Fehler im Test wird 1 Fehler im Betrieb erwartet
 - Fehlersuche dauert im Betrieb 4-10 Mal länger als bei systematischem Test

Selbstkontrolle

1. Begründen Sie, warum der Programmierer für das systematische Testen ungeeignet ist.
2. Erläutern Sie, was nach Ihrer Meinung ein gutes Programm ist.
3. Begründen Sie, warum trotz überlegter Implementierung und systematischem Testen trotzdem Fehler auftreten können.

Kontakt

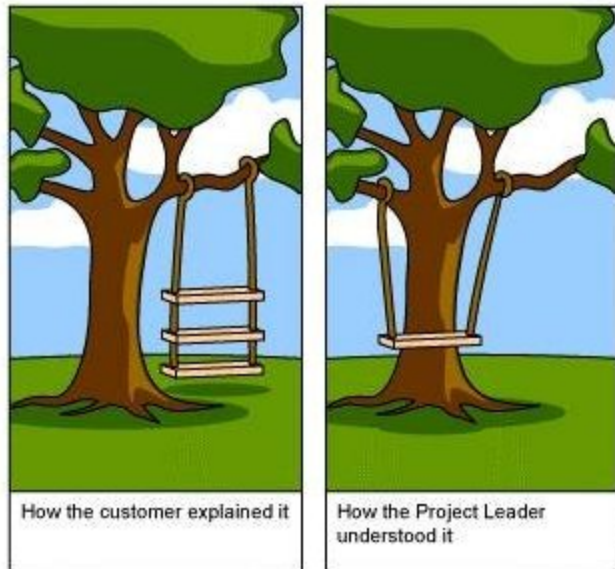
Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Software Engineering 1

Übersicht

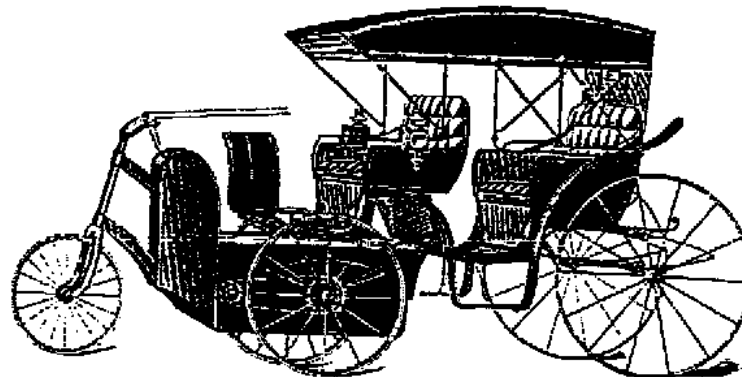


Prof. Dr. Andreas Judt

Benutzerschnittstellen?

- Erste Fragen

- Was ist eigentlich eine Benutzerschnittstelle?
- Wo findet man Benutzerschnittstellen?
- Wird Bedienbarkeit erlernt?
- Was heißt intuitiv bedienbar?
- Welche Benutzerschnittstellen hat ein Computer?



Phelps Traktor, 1901

Human Computer Interaction (HCI)

- Mensch-Maschine-Interaktion
 - Bedienung von Computern vereinfachen
 - seit Beginn des Computerzeitalters
- Bedienfreundlichkeit hat direkten Einfluss auf die Akzeptanz des Benutzers!
- zwei Richtungen der Entwicklung von Benutzerschnittstellen
 - mehr Komfort durch steigende Rechenleistung
 - Mobilisierung der Rechenleistung
 - ubiquitous computing – überall vorhandenes Rechnen

steigende Rechenleistung

- freundlicher, bedienbarer, effizienter!
- häufig nur ein einziger Weg der Bedienbarkeit
 - mangelnde Flexibilität
 - programs that do what they think you want
 - Benutzer ist bei Programmfehlern hilflos
- Gestaltungsrichtlinien
 - Wiedererkennungswert von Programmfunktionen
 - Ergonomie
 - intuitive Bedienbarkeit
 - Gewöhnungseffekte des Benutzers
- kinomatische Schnittstellen

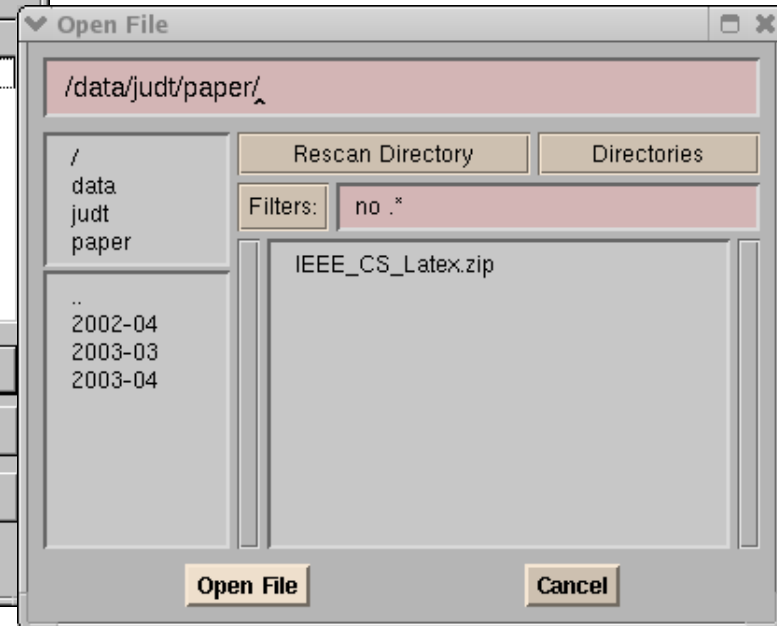
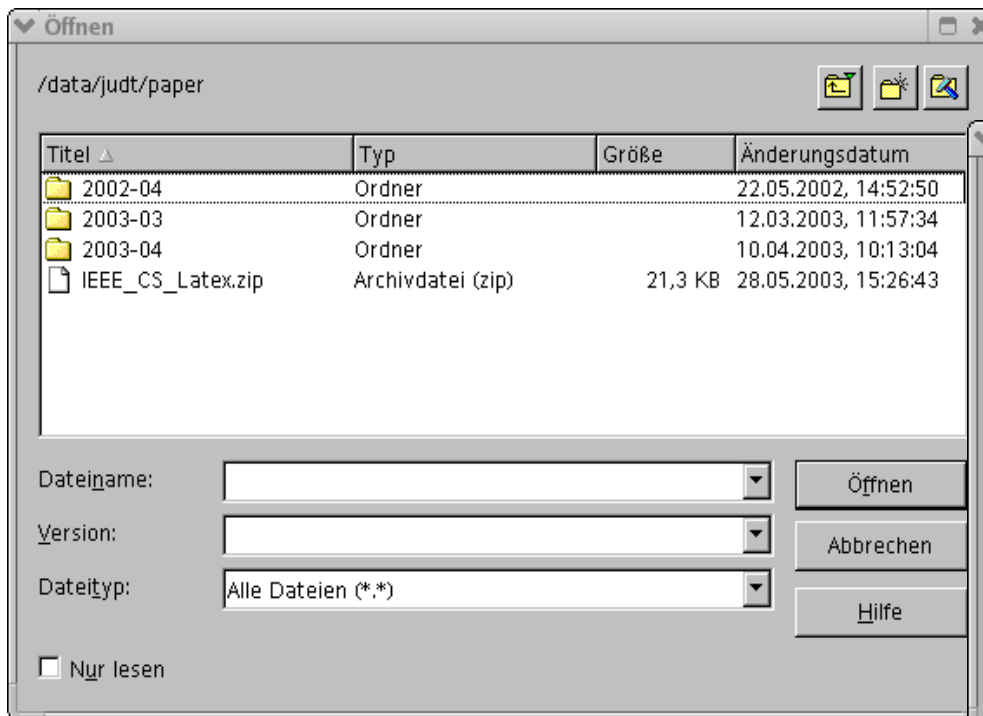
intuitive Bedienbarkeit

- Was stellt diese Benutzerschnittstelle dar?
- Welche Aktionen können ausgeführt werden?
- Was wird zur intuitiven Bedienbarkeit genutzt?



Gewöhnungseffekte

- Wie wirkt hier der Gewöhnungseffekt?



kinomatische Schnittstellen

- komfortable Benutzerschnittstellen verwenden
 - hohe Farbtiefe
 - Animationen
 - Klänge
 - Effekte
- Beispiel:
 - Datei kopieren mit Windows Explorer
 - je größer die Rechenleistung, desto mehr kinomatische Effekte
 - z.B. Windows95 gegen WindowsXP

Wo liegen die Grenzen?

- häufig mehr Rechenleistung für die Benutzerschnittstelle als für die Funktionalität
- Beispiele:
 - Kopieren im Windows Explorer
 - mit 64kb Speicher zum Mond fliegen, aber 256MB Speicher benötigen, um einen Brief zu schreiben
- Automatisierung und Kinomatisierung der Benutzerschnittstelle hat klare Grenzen
 - Rechenleistung
 - Flexibilität
 - Komplexität

Benutzerschnittstelle vs. Markterfolg

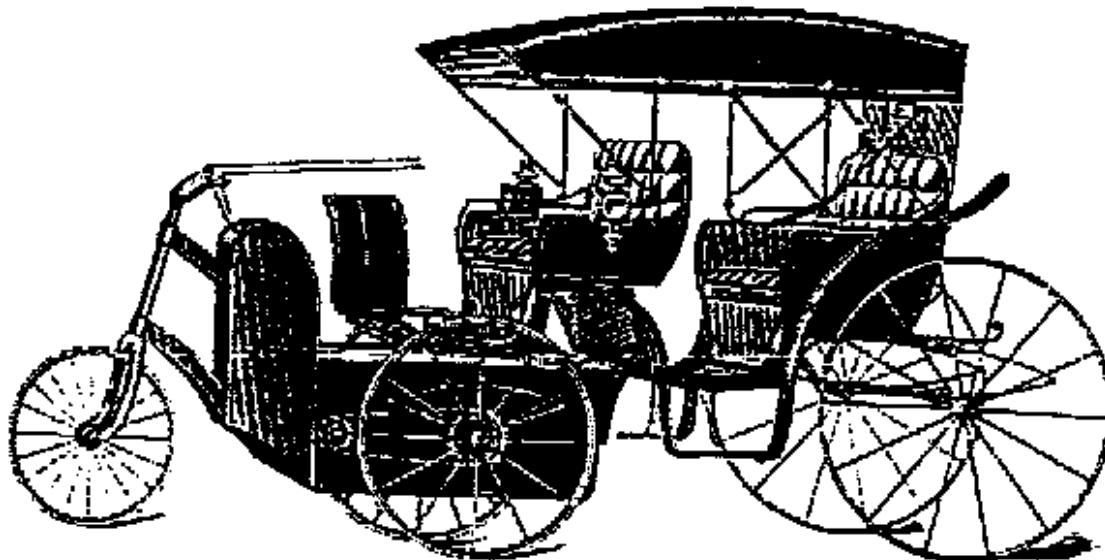
- Erfolg eines Produkts hängt von der Bedienbarkeit ab
 - Prinzip ist seit der Industrialisierung bewusst
- Benutzerschnittstelle beeinflusst seit mehr als 100 Jahren den Markterfolg verschiedenster Produkte
- Benutzerschnittstelle muss die Ausführung komplizierter Funktionen durch einfache Aktionen ermöglichen!

Markterfolg, Beispiel 1: Phelps Traktor, 1901

- 1900: Industrialisierung hatte begonnen
- Automobil seit mehr als 10 Jahren bekannt
- Phelps wollte Traktoren an Bauern verkaufen
- Problem
 - Bauern nutzen Pferdegespanne
 - Automobil wird mit Lenkrad, Pferd mit Zügeln gesteuert
 - Akzeptanz der Bauern für den Umstieg auf ein Automobil fraglich
- Lösung
 - Traktor muss lenkbar sein wie ein Pferdegespann!

Markterfolg, Beispiel 1: Phelps Traktor, 1901

- Phelps Traktor
 - Lenkung mit Zügeln
 - Aussehen ähnelt einem Pferdegespann
 - großer Markterfolg



Markterfolg, Beispiel 2: Philips ToUCam, 2001



Markterfolg, Beispiel 2: Philips ToUCam, 2001

- Webcam macht ein Foto, wenn man Cheese sagt.
- Nutzung von Rechenleistung
 - einfache Spracherkennung implementiert
 - Peripherie (Mikrofon, etc. günstig und leistungsfähig)
- benötigte Rechenleistung der Benutzerschnittstelle übersteigt deutlich die Leistung einer üblichen Webcam

Markterfolg, Beispiel 3: Apple iPod nano, 2008



Markterfolg, Beispiel 3: Apple iPod nano, 2008

- iPod nano besitzt einen Lagesensor (sog. Gyro-Sensor)
 - Automatisches Umstellen des Bilds bei Lageänderung
 - Zufällig Auswahl eines Musikstücks, wenn das Gerät geschüttelt wird
- Gyro-Sensoren ermöglichen eine einfache Bedienung durch Haptik.
 - Funktion wäre über Menü oder Stift kaum praktikabel.

Diskussion:

Beispiele für Benutzerschnittstellen

- Welche Benutzerschnittstellen kennen Sie?
 - Auto
 - Computer/Betriebssysteme/Programme
 - Automaten
 - Werkzeug
- Welche Benutzerschnittstellen sind gut/schlecht?
Warum?
- Warum benötigt man nicht für jedes Auto eine Fahrausbildung?

Geschichte der Benutzerschnittstelle

- Primitive Schnittstelle
 - 50er – 60er
- Kommandozeilen
 - 60er – 80er
- WIMP GUI
 - 80er bis heute
 - GUI based on windows, icons, menus and a pointing device
- jenseits von WIMP
 - Zukunft
 - erste Ansätze schon heute zu erkennen

Geschichte: einfache Schnittstellen

- Programmierung von Computern per Lochkarte mit Prozessorinstruktionen (Stichwort: Assembler)
- Ausgabe des Ergebnisses auf einem Zeilendrucker oder mit programmierbaren Lampen
- noch keine Benutzerschnittstelle, da keine Interaktion stattfand

Geschichte: Kommandozeilen

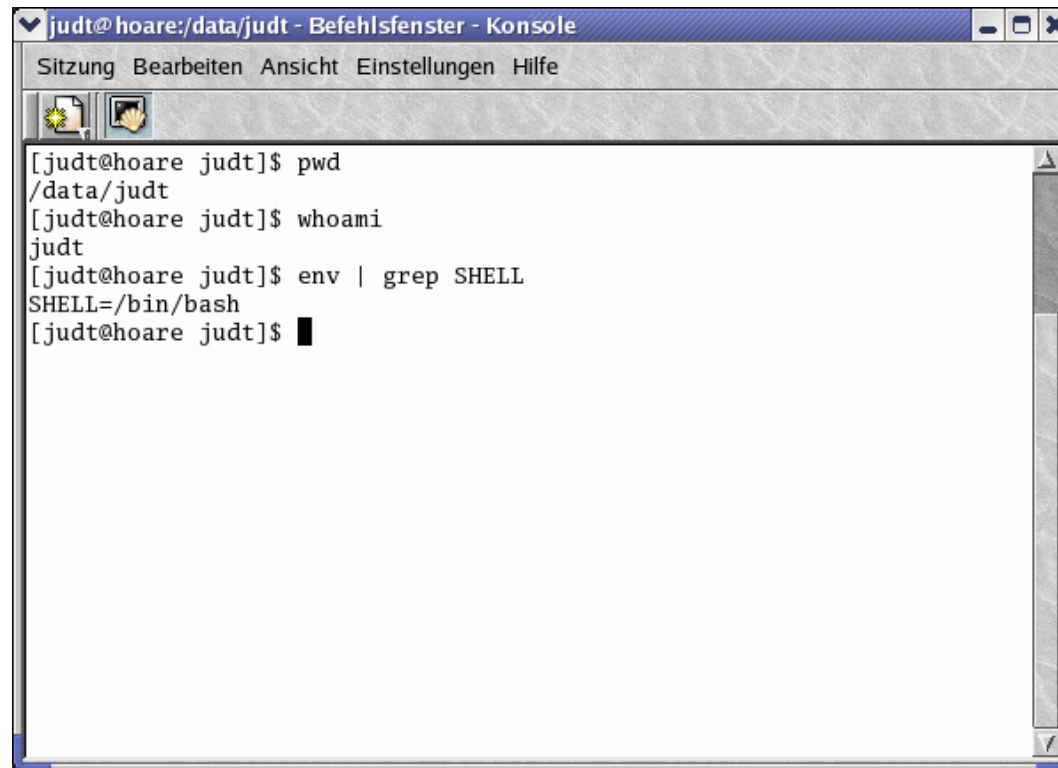
- Rechner mit Prozessmodellen
- Anwender erhielten Rechenzeit für die Ausführung von Kommandos
 - im Mikroprogramm
 - im Betriebssystem
- Eingabe über zeichenorientierte Bildschirme
 - sog. ASCII Terminals
 - z.B. DEC VT100
 - Eingabe eines einzigen Kommandos

Geschichte: Kommandozeilen

- Kommandozeilenschnittstellen sind heute noch in fast allen gängigen Betriebssystemen vorhanden
 - DOS Fenster bei Windowsvarianten
 - Terminalemulationen bei Unix, Linux, u.ä.
 - z.B. xterm, kterm
- Programm zur Interpretation der Kommandos spezifisch ladbar als Shell
 - Shell (sh),
 - Bourne Shell (bash),
 - Korn Shell (ksh),
 - C Shell (csh)

Geschichte: Kommandozeilen

- Linux kterm mit Bourne Shell (bash)



```
judt@hoare:/data/judt - Befehlsfenster - Konsole
Sitzung Bearbeiten Ansicht Einstellungen Hilfe

[judt@hoare judt]$ pwd
/data/judt
[judt@hoare judt]$ whoami
judt
[judt@hoare judt]$ env | grep SHELL
SHELL=/bin/bash
[judt@hoare judt]$
```

Geschichte: WIMP GUI

- Grafik, Fenster, Icons, Menüsteuerung, Tastatur, Maus
 - erstmals 1984 auf Apple Macintosh
 - wird bis heute in allen gängigen Betriebssystemen verwendet
 - Windows
 - Unix/Linux (mit Window Manager)
- trotz der rasanten Entwicklung der Rechenleistung bis heute fast unverändert
 - größere Farbtiefe,
 - Ton,
 - Animationen,
 - Schatten

Abstraktionsebenen bei WIMP GUI

- WIMP GUI werden als Fenstersysteme bezeichnet.
- Bedienung von GUI
 - Ein-/Ausgaben von Daten in Fenstern über die Tastatur
 - Auswahl von Funktionen in Menüs
 - Auslösen von Aktionen mit der Maus (eigentlich Zeigegerät)

Abstraktionsebenen bei WIMP GUI

- Fenstersysteme stellen Programmen verschiedene Dienstleistungen zur Verfügung
- Kategorisierung in Abstraktionsebenen
 - grafische Primitive
 - Interaktionselemente (widgets)
 - Gruppierung (layout)

Abstraktionsebene: Grafische Primitive

- Methoden zur Darstellung grafischer Elemente
 - Punkte, Linien
 - einfache geometrische Objekte
 - Schriften
 - Bilder
- Methoden arbeiten auf einer festgelegten und begrenzten Zeichenfläche
 - Fenstersysteme stellen Anwendungen die spezifischen Zeichenflächen zur Verfügung

Abstraktionsebene: Interaktionselemente

- Interaktionselemente (widgets)
 - Standardelemente zur Informationsdarstellung und Dateneingabe
 - z.B. Schaltflächen, Klappleisten, Textfelder
- Fenstersystem stellt Methoden zur Einrichtung von Interaktionselementen zur Verfügung
- Anwendung soll Interaktion mitbekommen
 - z.B. Drücken einer Schaltfläche

Abstraktionsebene: Interaktionselemente

- Anwendung soll sich nicht um die Darstellung des Interaktionselements kümmern
 - z.B. Darstellung des gedrückten Zustands einer Schaltfläche
- Fenstersystem steuert das Zeichnen der Interaktionselemente
 - z.B. nach Überdeckung, Verschiebung

Abstraktionsebene: Gruppierung

- Interaktionselemente müssen für die Anwendung angeordnet werden
- Fenstersystem stellt Methoden für die Gruppierung (layout)
- Fenstermanager baut eine hierarchische Struktur der Interaktionselemente auf
- Fenstersystem bietet auf oberster Ebene Methoden zur Verwaltung von Fenstern
 - erzeugen
 - darstellen
 - manipulieren

Abstraktionsebene: Gruppierung

- Fenstersystem entlastet Anwendung weitgehend
- Interaktionen des Benutzers, die nur ein Fenster manipulieren, werden komplett vom Fenstersystem bearbeitet
 - Größenänderung
 - Verschieben
 - Aktivieren
 - Deaktivieren
- Anwendung erfährt von der Interaktion des Benutzers nur, wenn sie informiert wird
 - Rückruf (callback)
 - z.B. Neuzeichnen von Fensterinhalten nach Vergrößerung

GUI Bibliotheken

- GUI Bibliotheken sind groß und flexibel
 - Prinzipien verstehen
 - Struktur erkennen
 - Funktionalität nachlesen
 - begleitende Literatur unverzichtbar
- Vorlesung zeigt Konzepte
 - Verständnis für den Umgang mit GUI
 - Fähigkeit erlangen, in GUI Bibliotheken nachzulesen
 - bislang unbekannte Funktionalitäten erarbeiten und anwenden können

Einführung verschiedener Technologien

- Grundlagen verstehen
 - Konzepte, Übungen
 - Strickmuster identifizieren und anwenden
 - Wissen selbständig erweitern
- API Dokumentation, Literatur
 - Funktionalitäten selbständig ausprobieren
 - unbekannte Funktionalitäten erarbeiten
 - Benutzerschnittstellen entwerfen
 - Gestalt definieren
 - verwendete Funktionalität auswählen

Geschichte: Jenseits von WIMP

- neue Interaktionstechniken gesucht
 - seit den 90ern
 - to get the interface out of your face
- neue Technologien erfordern neue Eingabetechniken
 - z.B. persönlicher Assistent
 - keine Maus
 - Bedienung mit Stift und Touchscreen
 - Tastatur häufig nur noch als Anwendungsprogramm

Geschichte: Post-WIMP Ansätze

- weg von Fenstern(2 1/2D) mit 3D Navigation
- räumliche Navigation nicht intuitiv, da Raumgefühl schwierig zu lernen
- Anwender sieht eine Projektion des Raums auf den Bildschirm
- Anwendung fehlt zur Durchsetzung als allgemein verwendete Benutzerschnittstelle

Geschichte: Post-WIMP Ansätze

- **Spracherkennung**
 - heute schon vereinzelt verwendet
 - Auskunftssysteme,
 - Buchungssysteme
 - aufgrund der benötigten Rechenleistung nur sehr begrenzt einsetzbar
- **Gestenerkennung**
 - heute noch kaum verwendet
 - erste Ansätze der Forschung
 - Interpretation von Gestensprache
 - Analyse der Gemütsstimmung des Anwenders und anschließende Anpassung der Benutzerschnittstelle

Geschichte: Post-WIMP Ansätze

- elektronisches Papier
 - Gyricon bzw. ePaper (90er)
- 1000 Mal elektronisch wiederbeschreibbares Papier
- praktische Umsetzungen werden seit mehr als 15 Jahren angekündigt

Geschichte: elektronisches Papier



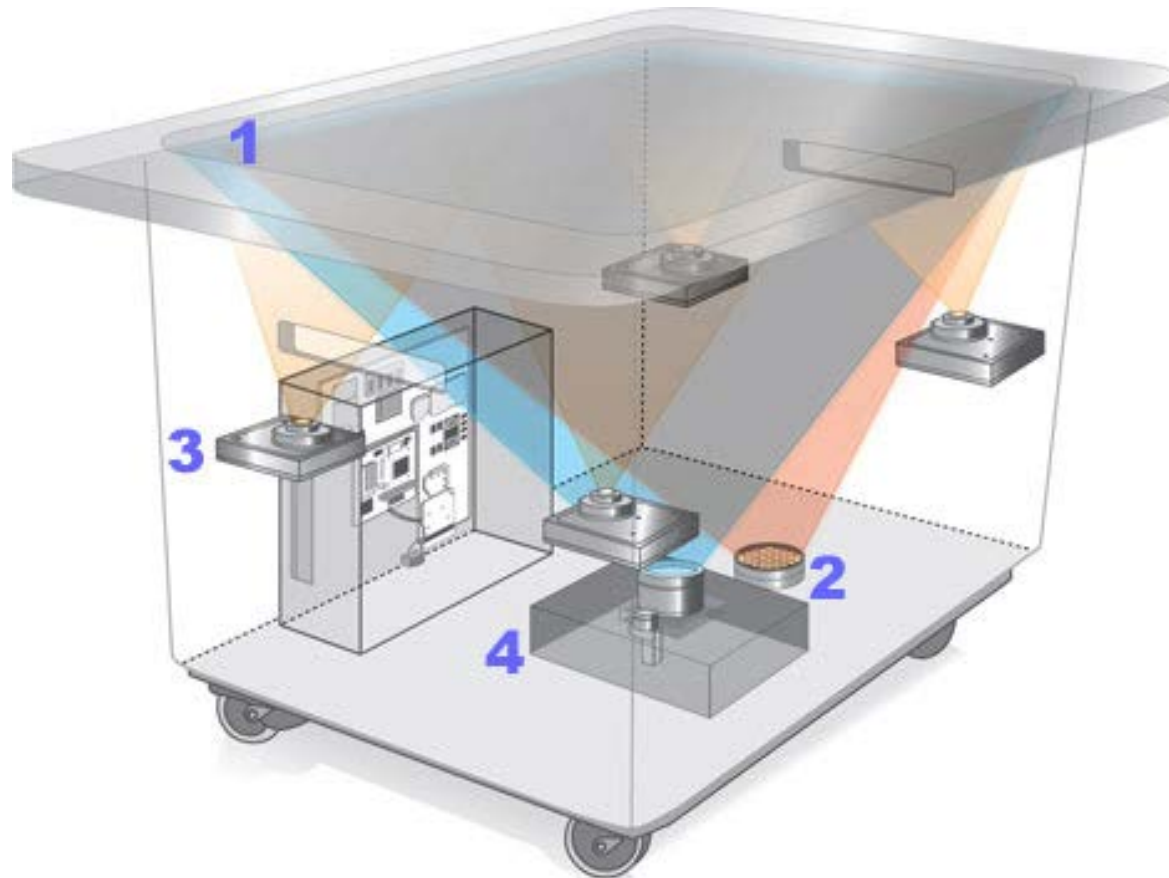
Geschichte: Post-WIMP Ansätze

- anziehbare Computer
 - eingebaut/angebracht an die Kleidung
 - keine Maus
 - Tastatur am Arm befestigt
 - kleiner Bildschirm im Blickwinkel eines Auges
 - alternativ: Laser projiziert Bild auf die Netzhaut eines Auges
- Elektronik ist in Kleidung integrierbar
 - kleine Rechner in einer Tasche
 - Leiterbahnen in den Stoff eingewebt; Stoff im Futter mit SMD Bauteilen bestückt

Post-WIMP-Ansatz: Microsoft Surface, 2008



Post-WIMP-Ansatz: Microsoft Surface, 2008



Kontakt

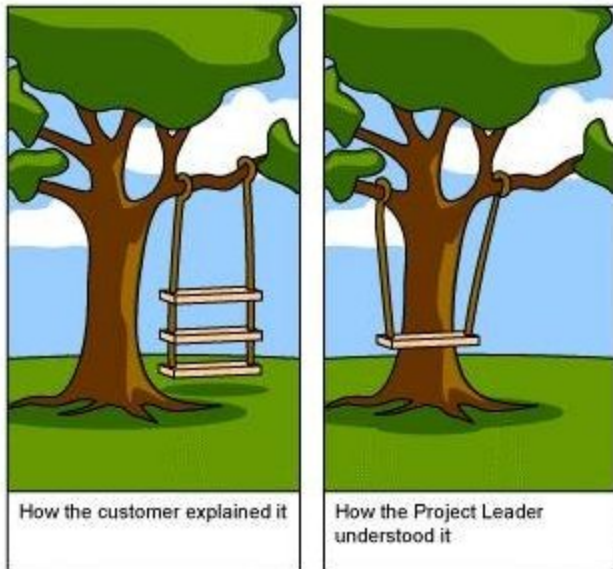
Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Software Engineering 1

Abstract Window Toolkit (AWT)



Prof. Dr. Andreas Judt

AWT: einfache GUI

- Abstract Window Toolkit (AWT)
 - einfache GUI Dienstleistungen für Java Programme
 - erzeugt unter dem jeweiligen Betriebssystem Fenster, Zeichenflächen und Interaktionselemente
 - reagiert auf Interaktionen und erzeugt Ereignisse
 - Ereignisse werden von der Programmlogik verarbeitet
 - AWT Bibliothek: `java.awt`

Interaktionselemente

- Implementierung der AWT Interaktionselemente basiert auf der Implementierung des Betriebssystems
 - Aufruf von Betriebssystemroutinen zur Darstellung
 - Darstellung der GUI sieht auf verschiedenen Betriebssystemen unterschiedlich aus
 - Ereignisse können ebenfalls unterschiedlich erzeugt werden!
- betriebssystem-unabhängige GUI-Bibliothek: Swing
 - kommt später
 - alle AWT GUI sind auch mit SWING programmierbar

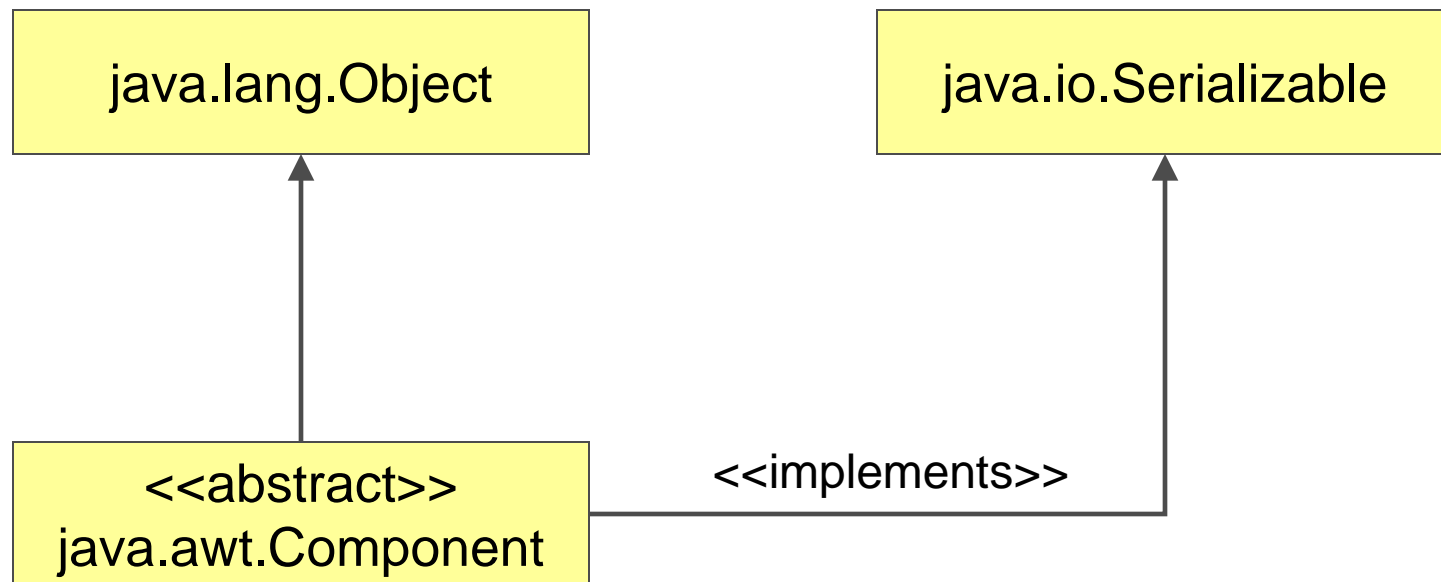
AWT Klassenhierarchie

- AWT GUI werden komponentenweise organisiert
 - einheitliche Schnittstellen garantieren Baukastensystem mit großer Flexibilität
 - Spezialisierungen der Komponenten
 - Interaktionselemente (widgets)
 - Gruppierungen (layouts)
 - Fenster
- Swing Komponenten sind Spezialisierung der AWT Komponenten

java.awt.Component

- Oberklasse der AWT GUI
- direkte Unterklasse von java.lang.Object
- abstract
- legt Grundfunktionalitäten fest
 - Darstellung aller grafischen Objekte auf dem Bildschirm
 - für Interaktionen zwischen Programmlogik und Benutzer
- implementiert java.io.Serializable

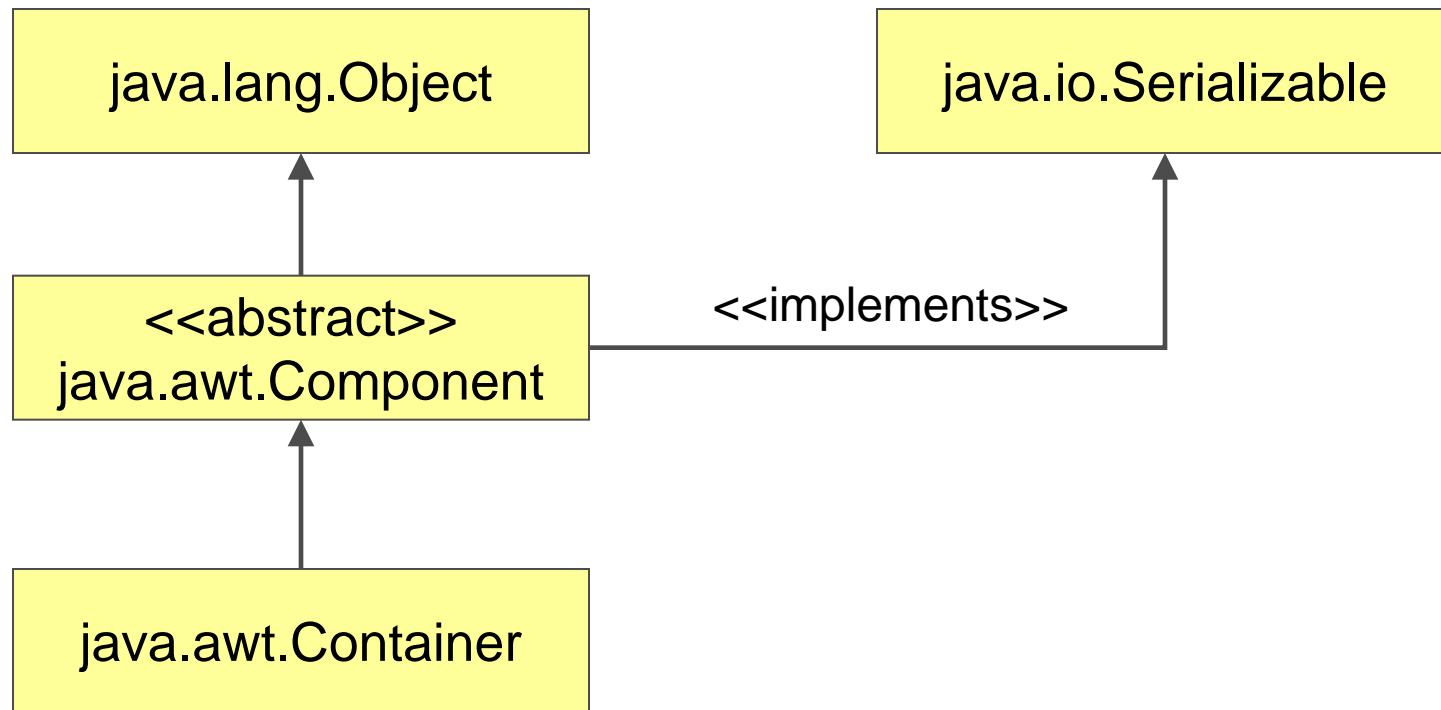
java.awt.Component



java.awt.Container

- implementiert Behälter der Gruppierung
 - konkrete Unterklasse von Component
 - Zusammenfassung grafischer Objekte vom Typ Component
- nimmt Komponenten in den Behälter auf und organisiert
 - Anordnung
 - Darstellung
 - Überdeckung

java.awt.Container



java.awt.Window

- Zeichenbereich für Interaktionselemente
- direkte Unterklasse von java.awt.Container
- java.awt.Window Fenster...
 - sind rechteckig
 - haben Rahmen
 - haben Titelleiste
- Fensterklassen sind Unterklassen von java.awt.Window
- direkte Unterklassen
 - java.awt.Frame
 - java.awt.Dialog

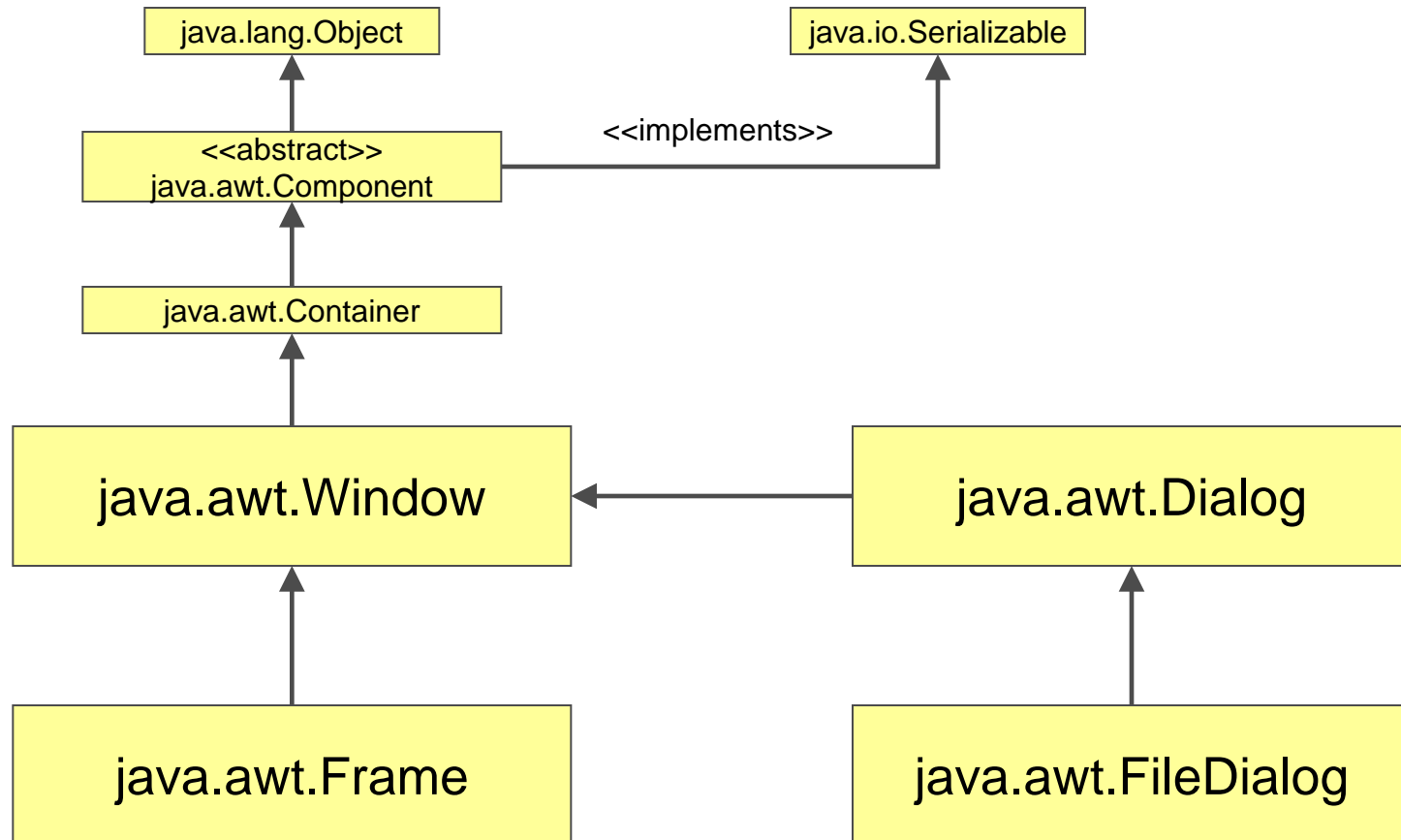
java.awt.Frame

- Implementierung für Hauptfenster (Startfenster) für Java GUI
- direkte Unterklasse von java.awt.Window
- erweitert java.awt.Window um...
 - Menüleisten
 - Vergrößern
 - Verkleinern
 - Ikonisieren
- java.awt.Frame bietet verschiedene Methoden zur Definition des Verhaltens von Fenstern

java.awt.Dialog

- Dialogfenster auf oberster Ebene
- direkte Unterklasse von java.awt.Window
- wird verwendet, um eine Entscheidung vom Benutzer zu erfragen
- hat Titel und Rahmen
- direkte Unterklasse:
 - java.awt.FileDialog
- Dialog wird mit Referenz auf ein übergeordnetes Frame oder einen übergeordneten Dialog konstruiert

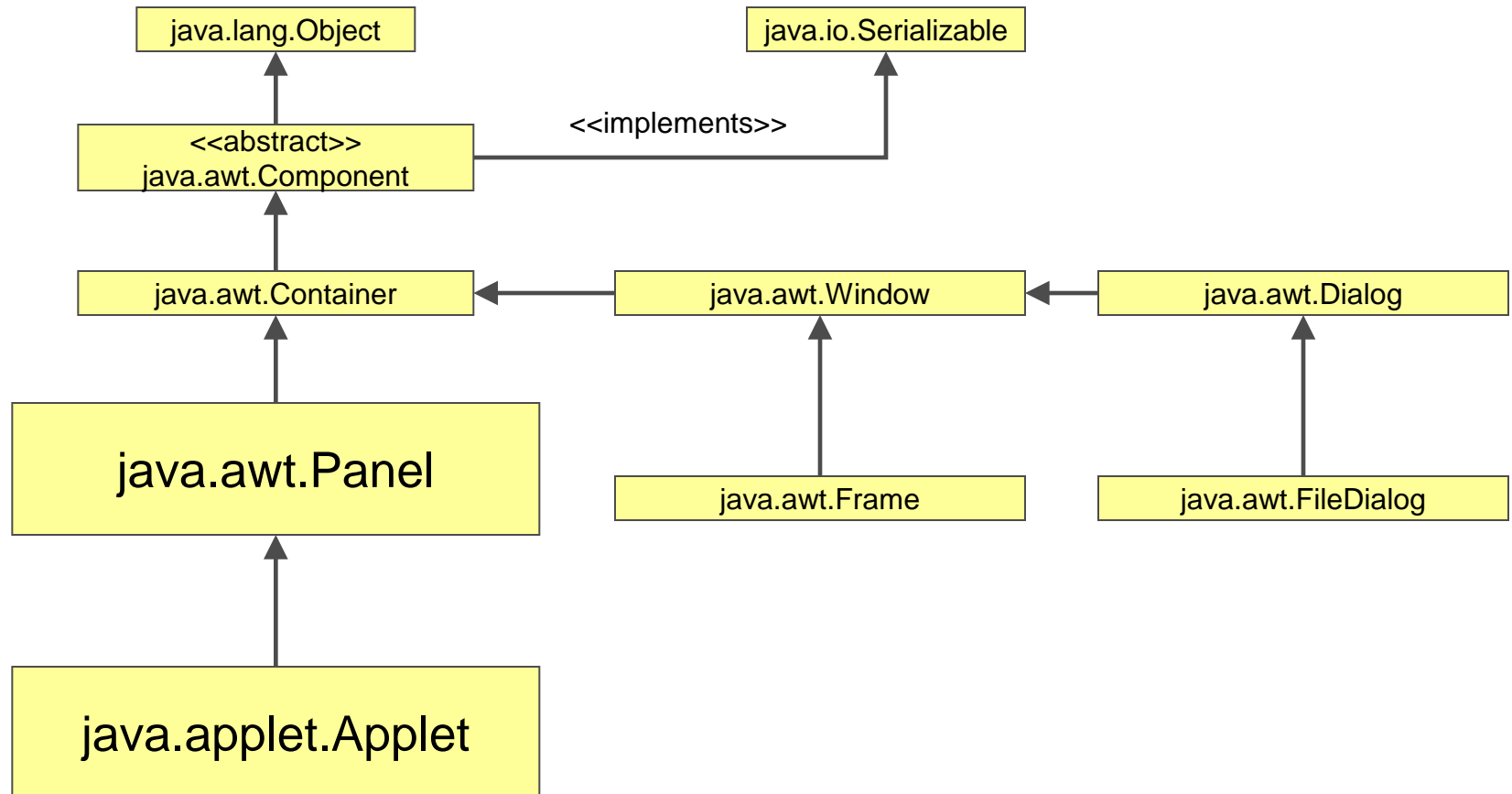
java.awt.Window



java.awt.Panel

- direkte Unterklasse von java.awt.Container
- einfachste Implementierung einer Gruppierung für Interaktionselemente
- stellt sichtbaren Zeichenbereich für Komponenten zur Verfügung
- direkte Unterklasse: java.applet.Applet

java.awt.Panel



Benutzer-Interaktionen

- Interaktionen sind Aktionen des Benutzers mit Interaktionselementen der Benutzerschnittstelle.
- Interaktionen werden als Ereignisse (events) an das Programm gemeldet.
- Das Programm implementiert spezifische Zuhörer (listener), um Ereignisse zu verarbeiten.

Ereignisse

- GUI haben mehr Flexibilität als kommandozeilenorientierte Benutzerschnittstellen
 - Eingaben in selbstdefinierter Reihenfolge
 - Vergrößern, Verkleinern, Verschieben der GUI
- Programm muss zu jedem Zeitpunkt auf verschiedenste Aktionen des Benutzers reagieren können
- Interaktionen werden als Ereignisse gemeldet
 - primitive Ereignisse
 - semantische Ereignisse

primitive Ereignisse

- Beispiele für primitive Ereignisse
 - Verschieben
 - Klicken auf den Hintergrund
 - Vergrößern/Verkleinern/Ikonisieren
- primitive Ereignisse haben keinen Einfluss auf den Programmzustand
- werden nicht weiter verarbeitet

semantische Ereignisse

- Semantische Ereignisse haben eine Bedeutung für den Programmzustand
- Beispiele:
 - Klicken auf einen Knopf
 - Eingabe von Text
- Ereignisse werden vom Betriebssystem erzeugt und an das betroffene Interaktionselement weitergeben
- Ereignisse werden zu Ereigniskategorien zusammengefasst

Kategorisierung von Ereignissen

- Welches Ereignis semantisch ist, hängt von der Implementierung des Interaktionselements ab
 - eigene Implementierung der Reaktion auf Ereignisse
- Ereignisse werden von Zuhörern (listener) verarbeitet
 - für jedes semantische Ereignis wird ein Zuhörer für die jeweilige Ereignisklasse implementiert

Ereignisse in AWT

- Ereignisse werden in Java als Ereignisklassen modelliert und als Ereignisobjekte erzeugt
- Basispaket: `java.awt.event`
- Ereignisobjekte einer Ereigniskategorie werden automatisch erzeugt, wenn der Benutzer ein Ereignis auslöst und das Interaktionselement einen Hörer implementiert.
- Basisklasse für alle Ereignisse: `java.util.EventObject`

java.awt.AWTEvent

- Oberklasse alle Ereignisse in AWT
- Quelle jedes Ereignisses lässt sich mit der Methode `getSource` ermitteln
- Konkrete Unterklassen für AWT Ereigniskategorien sind im Paket `java.awt.event` zu finden

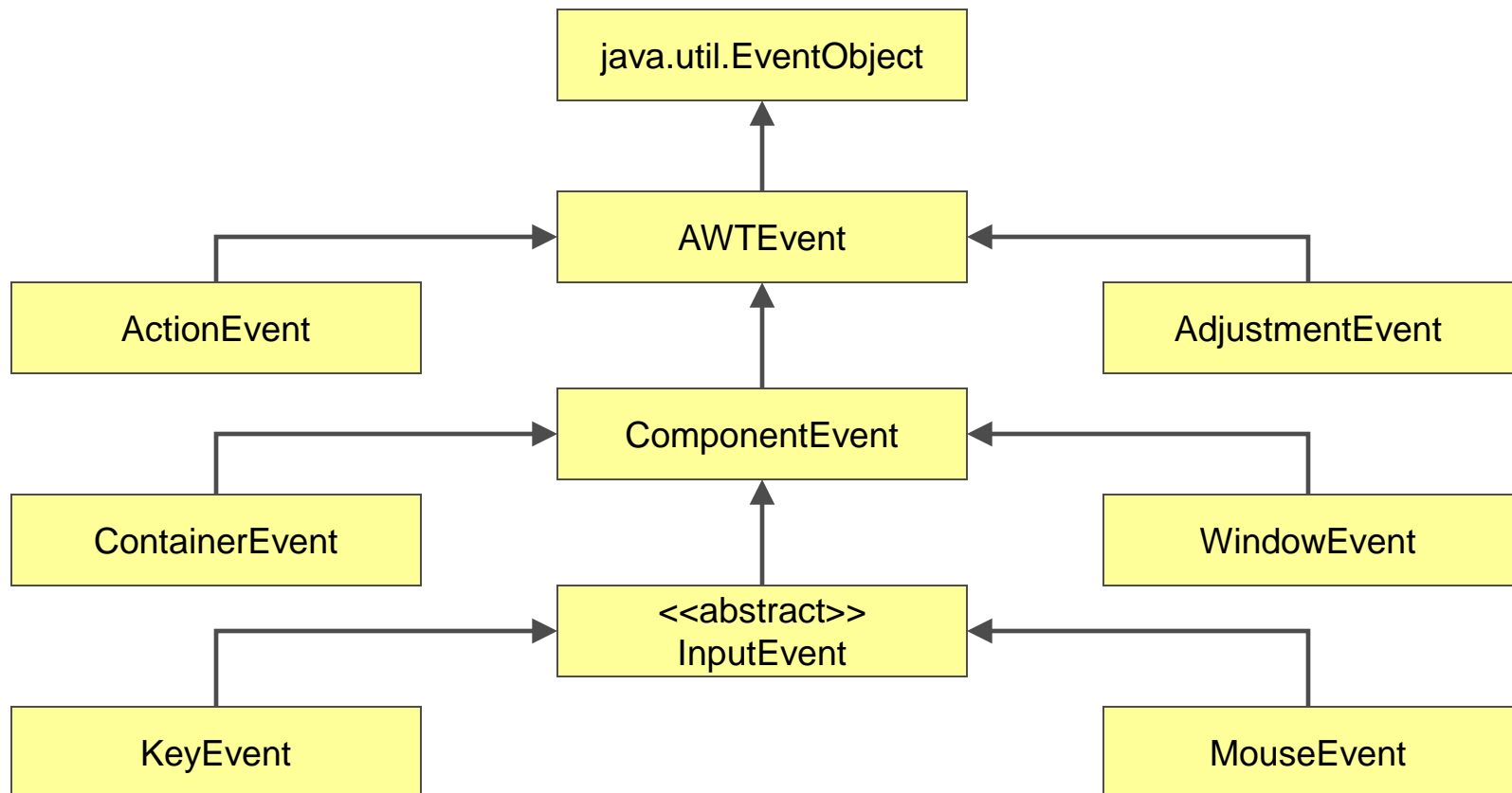
Ereignisklassen

- **ActionEvent**
 - (beliebige) Aktion
- **AdjustmentEvent**
 - Änderung an Verschiebepalken
- **TextEvent**
 - Textänderung in Eingabefeldern
- **ComponentEvent**
 - primitive Ereignisse für Component Objekte
- **WindowEvent**
 - Spezialisierung von ComponentEvent
 - primitive Fensterereignisse

Ereignisklassen

- FocusEvent
 - Fokuswechsel zwischen Interaktionselementen
- InputEvent
 - abstrakte Klasse
 - Eingabe mit Maus oder Tastatur
 - Konkrete Unterklassen:
 - MouseEvent
 - KeyEvent
- java.awt.event bietet noch wesentlich mehr Ereignisklassen
- jede Ereigniskategorie besitzt unterschiedliche Methoden zur Verarbeitung

Klassenhierarchie



Hörer

- Ereignisse werden von speziellen Klassen verarbeitet, die einen Hörer (listener) implementieren
- Hörer sind Schnittstellen (interfaces)
- alle Hörer haben das Superinterface `java.util.EventListener`
- Hörer definieren Interaktionskategorien
- zu jedem Ereignis gibt es einen passenden Hörer
- Hörer im Paket `java.awt.event`

Besonderheit:

`java.awt.event.AWTEventListener`

- kein Superinterface für AWT Hörer!
- kann alle Ereignisse von AWT verarbeiten
- wird zu Testzwecken verwendet
- alle AWT Hörer erweitern `java.util.EventListener`
- Klassenstruktur der Hörer entspricht nicht exakt der Struktur der Ereigniskategorien!

Adapter

- triviale Klassen, die Hörer implementieren
- Adapter vereinfachen die Implementierung
 - es werden nur die benötigten Methoden überschrieben
 - z.B. MouseAdapter, WindowAdapter, ...
- Hinweis
 - Welcher Listener hat keine Adapter? Warum?

Kochrezept: Ereignisse verarbeiten

- Hörerklasse für ein Ereignis implementieren
 - als Hörer Implementierung
 - als Unterklasse eines Adapters
- Hörerklasse muß die Methode zur Bearbeitung des gewünschten Ereignisses überschreiben
- Objekt der Hörerklasse erzeugen
- Hörerklasse wird dem Interaktionselement mit einer speziellen addXXXListener Methode hinzugefügt
 - z.B. addMouseListener

Beispiel: ein Knopf

```
import java.awt.*;
import java.awt.event.*;

public class Uebung extends java.awt.Frame {
    public Uebung() {
        super("Übung 4");
        Button button = new Button("Ende");
        button.addActionListener(new MyActionListener(this));
        setBounds(100,100,200,150);
        add(button);
        setVisible(true);
    }

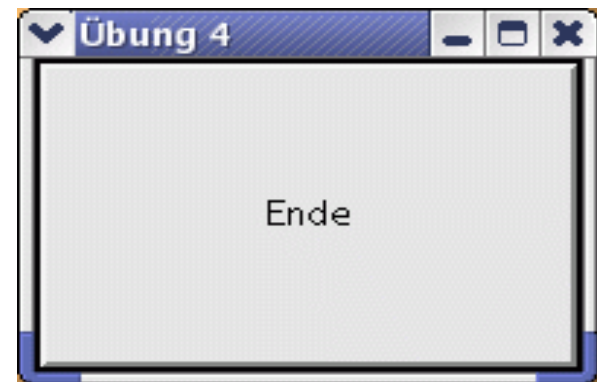
    public static void main(String[] args) {
        Uebung u = new Uebung ();
    }
}
```

Beispiel: ein Knopf

```
class MyActionListener implements ActionListener {  
  
    private Frame parent;  
  
    public MyActionListener(Frame parent) {  
        this.parent = parent;  
    }  
  
    public void actionPerformed(java.awt.event.ActionEvent e) {  
        parent.setVisible(false);  
        System.exit(0);  
    }  
  
}
```

Beispiel: ein Knopf

- ActionListener bearbeitet beliebige Ereignisse
- eigene Klasse MyActionListener überschreibt actionPerformed
- Klick auf Knopf beendet das Programm
- Gestaltung des Knopfs entspricht nicht der Vorstellung!



Übung: Ereignisse verarbeiten

- Wandeln Sie das Beispiel so ab, dass das Programm mit einem Doppelklick beendet wird.
- Bei einfachem Klick soll das Programm „Klick.“ auf die Konsole ausgeben.
- Welchen Hörer müssen Sie implementieren?
- Verwenden Sie einen Adapter.

Darstellung und Anordnung

- Interaktionselemente werden Behältern hinzugefügt
 - add Methode von `java.awt.Container`
 - Hinzufügen von beliebigen Komponenten vom Typ `java.awt.Component`
- `java.awt.Container` bietet die Verwendung eines Darstellungsverwalters (layout manager) für die Positionierung der Interaktionselemente
 - `setLayout(java.awt.LayoutManager mgr)`
 - AWT kennt verschiedene Darstellungsverwalter

java.awt.LayoutManager

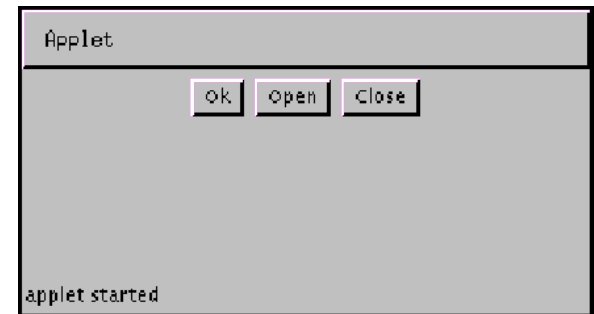
- Schnittstelle für die Implementierung von Darstellungsverwaltern
 - Methoden zur Anordnung von Komponenten in Behältern
 - Interaktionselemente und Behälter
 - java.awt.LayoutManager2 spezialisiert die Darstellungsverwaltung um hinzufügbare Platzierungsregeln
- AWT Darstellungsverwalter implementieren beide Schnittstellen

java.awt.FlowLayout

- einfachster Darstellungsverwalter
- ordnet Komponenten zeilenweise an
- neue Zeile wird begonnen, wenn verbleibender Platz zu klein für einzufügende Komponente
- Komponenten werden zeilenweise zentriert angezeigt
- Standard für Applets

java.awt.FlowLayout

```
import java.awt.*;  
import java.applet.Applet;  
  
public class myButtons extends Applet {  
    Button button1, button2, button3;  
    public void init() {  
        button1 = new Button("Ok");  
        button2 = new Button("Open");  
        button3 = new Button("Close");  
        add(button1);  
        add(button2);  
        add(button3);  
    }  
}
```



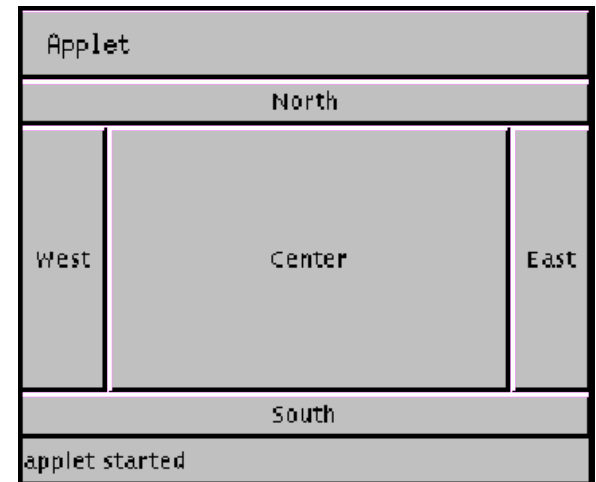
java.awt.BorderLayout

- BorderLayout teilt den Behälter in 5 Bereiche
 - NORD, SÜD, WEST, OST, MITTE
 - in jeden Bereich nur eine Komponente einfügbar
- Größe der Komponenten wird angepasst
 - NORD, SÜD horizontal
 - WEST, OST vertikal
 - MITTE horizontal und vertikal
- Bleibt ein Bereich leer, füllen die übrigen Bereiche den Behälter

java.awt.BorderLayout

```
import java.awt.*;
import java.applet.Applet;

public class buttonDir extends Applet {
    public void init() {
        setLayout(new BorderLayout());
        add(new Button("North"), BorderLayout.NORTH);
        add(new Button("South"), BorderLayout.SOUTH);
        add(new Button("East"), BorderLayout.EAST);
        add(new Button("West"), BorderLayout.WEST);
        add(new Button("Center"), BorderLayout.CENTER);
    }
}
```

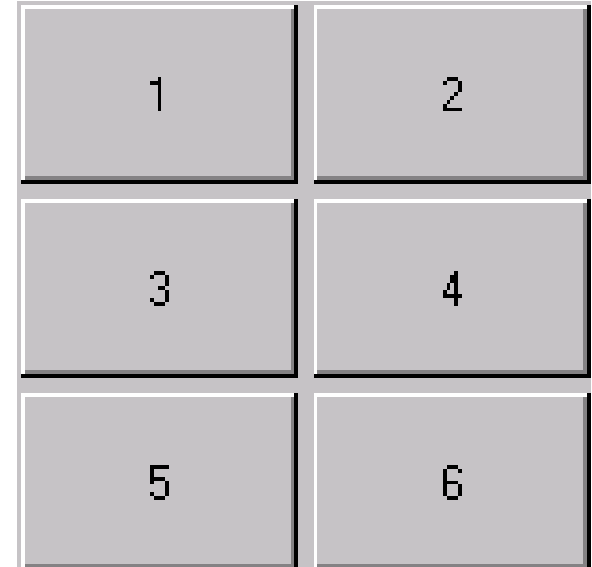


java.awt.GridLayout

- Behälter wird in gleich große Zellen geteilt
 - Angabe von Zeilenzahl und Spaltenzahl
 - Darstellungsverwalter paßt die Darstellung an, wenn die Feldzahl nicht mit der Zahl der Komponenten übereinstimmt
 - Spaltenzahl wird angepasst
 - Zeilenzahl bleibt konstant
- Komponenten werden an die Größe der Zellen angepasst

java.awt.GridLayout

```
import java.awt.*;  
import java.applet.Applet;  
public class ButtonGrid extends Applet {  
    public void init() {  
        setLayout(new GridLayout(3,2));  
        add(new Button("1"));  
        add(new Button("2"));  
        add(new Button("3"));  
        add(new Button("4"));  
        add(new Button("5"));  
        add(new Button("6"));  
    }  
}
```



java.awt.GridBagLayout

- flexibelster Darstellungsverwalter
- Zellgröße individuell einstellbar
- Komponenten über mehrere Zellen einfügbar
 - Zellen definieren Darstellungsbereich (display area)
- Anordnung von Komponenten über Regeln einstellbar
 - z.B. Gewicht, Platzhalter
- Programmierung kompliziert
 - mit GUI Editor in einer Entwicklungsumgebung verwenden

java.awt.GridBagLayout

- Programm in der Dokumentation von java.awt.GridBagLayout nachlesbar.

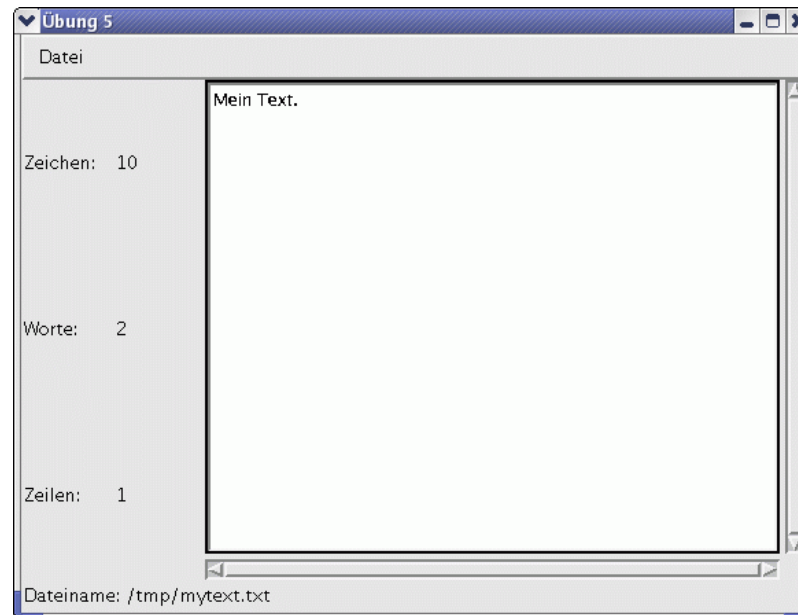


Kombination von Darstellungsverwaltern

- Darstellungsverwalter werden Behältern zugeordnet.
- Behälter können Behälter enthalten
- Behälter eines Fensters können beliebig gestaltet werden
- Darstellungsverwalter werden je nach Bedarf gewählt

Übung: Texteditor

- Schreiben Sie einen einfachen Texteditor mit folgendem Aussehen:



Übung: Texteditor

- Der Texteditor enthält
 - ein Menü zum Öffnen und Speichern des Texts und Beenden des Programms
 - eine Statuszeile, die den Dateinamen anzeigt
 - eine Spalte, die Zeichenzahl, Wortzahl und Zeilenzahl anzeigt und bei jeder Änderung des Texts aktualisiert wird
- Verwenden Sie die gezeigte Aufteilung des Fensters!
- Die Beschreibung der Komponenten finden Sie in der Java API Dokumentation.

Selbstkontrolle

1. Erläutern Sie den Unterschied zwischen Container und Komponente.
2. Warum sehen AWT GUI bei unterschiedlichen Betriebssystemen verschieden aus?
3. Was ist ein Darstellungsverwalter?
4. Was unterscheidet LayoutManager und LayoutManager2?
5. Warum besitzen nicht alle Listener einen Adapter?
6. Warum ist es sinnvoll, Komponenten in Containern zu organisieren?

Kontakt

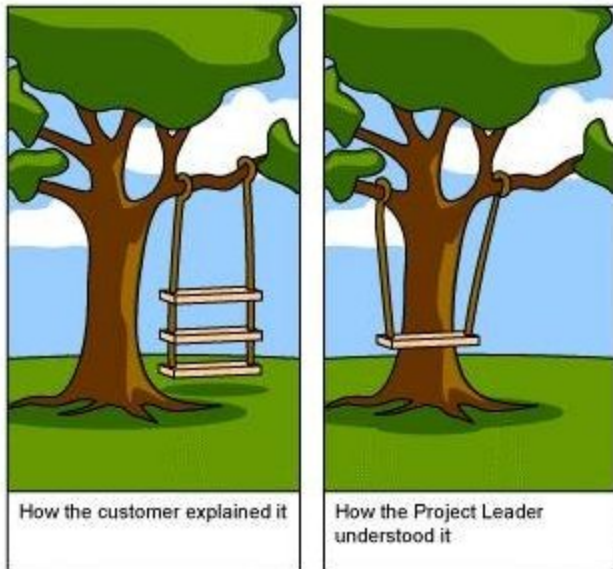
Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Software Engineering 1

Java Swing



Prof. Dr. Andreas Judt

Was ist Swing?

- komponentenbasierte GUI
 - look-and-feel zur Laufzeit konfigurierbar
 - Komponenten sind einfach erweiterbar
 - plattformunabhängiges Aussehen
- Swing Komponenten implementieren Model-View-Controller (MVC) Entwurfsmuster
 - mehr Flexibilität
 - strukturierte Programmierung der Darstellung von Informationen und Steuerung von Interaktionen
- komplexe und leistungsfähige MVC Implementierungen
 - z.B. Tabellenmodell

Swing ein Ersatz für AWT?

- NEIN, Swing basiert auf den AWT Klassen
- pures Java, keine nativen Bibliotheken
- bietet viele neue Interaktionselemente
 - Tabellen
 - Bäume
 - interne Fenster
- schärferes Konzept für leichtere Programmierung
- Swing Komponenten sind JavaBeans:
 - serialisierbar
 - beliebig in Werkzeuge integrierbar

Probleme des AWT Konzepts

- AWT - ein minimalistischer Ansatz für einfache GUI (ursprünglich für Applets!)
- Bedienung von plattformabhängigen Interaktionselementen brachte zunehmende Probleme
 - kleinster gemeinsamer Nenner statt Komfort
 - begrenzte Fähigkeiten der AWT GUI
 - benötigen viel Speicher
- Stark auseinander entwickelnde GUI der Betriebssysteme stellen ein unlösbares Problem dar!
- Entwicklung der Java Foundation Classes (JFC)

Java Foundation Classes (JFC)

- Sammlung von Bibliotheken für große Unternehmensanwendungen mit Java
 - AWT
 - Swing
 - Accessibility
 - bessere Zugriffsmöglichkeiten auf Fähigkeiten der Interaktionslemente
 - 2D API
 - 2D Grafik, Schriften, Farben
 - Drag-and-Drop
 - Bewegen von GUI Objekten, Reaktion auf Ablegen
 - Schnittstelle zu nativen Anwendungen
 - z.B. Bewegen einer Datei auf Verwaltungsprogramm

Neue Technologien: Lightweight Components

- Komponenten übernehmen ihre Darstellung selbst
 - verwenden grafische Primitive
- Klassen greifen nicht auf Systemfunktionen zur Darstellung zu
 - schnelles Neuzeichnen
 - bessere Speichernutzung
- Fast alle Komponenten sind leichtgewichtig
 - Behälter der obersten Ebene nicht: JApplet, JFrame, JDialog, JWindow
- Look-and-Feel kann GUI an jeweilige Plattform anpassen

Neue Funktionen

- neue Interaktionselemente
 - Tabellen, Bäume, Schieber, Regler, Fortschrittsanzeigen, interne Fensterverwaltung, diverse Texteingaben
- Modifikation des Aussehens
 - z.B. verschiedene Rahmen, Gestaltungsalternativen
- Tooltips
 - kleine Textflaggen zur Information eines Interaktionselements
 - wird ausgelöst, wenn Mauszeiger über dem Interaktionselement steht

Neue Funktionen

- beliebige Verknüpfbarkeit von Tastaturereignissen zu beliebigen Komponenten
 - beliebige Reaktionen in beliebigen Zuständen auf beliebige Tastenkombinationen
- Fehlersuche-Unterstützung für leichtgewichtige Komponenten

Paketstruktur (1/5)

- javax.accessibility
 - Klassen und Schnittstellen für erweiterten Komponentenzugriff (assistive technologies)
- javax.swing
 - elementare Klassen und Modelle
- javax.swing.border
 - Implementierung und Verwaltung von Dekorationen
 - Rahmen sind keine Komponenten, sondern speziell verwaltete grafische Elemente, die als Eigenschaften verwaltet werden

Paketstruktur (2/5)

- javax.swing.colorchooser
 - komfortable Farbauswahl
- javax.swing.event
 - spezifische Ereignisse und Hörer
- javax.swing.filechooser
 - komfortable Dateiauswahl

Paketstruktur (3/5)

- javax.swing.plaf
 - pluggable Look-and-Feel
 - konfigurierbare, plattformspezifische Darstellung von Komponenten
- javax.swing.table
 - Modelle und Sichten für Tabellen
 - Implementierungen für Ansicht, Selektion, Editierung
- javax.swing.text
 - Klassen und Schnittstellen für textbasierte Interaktionslemente (document/view)
 - dokumentbasierte Datenverwaltung

Paketstruktur (4/5)

- javax.swing.text.html
 - Darstellung und Formatierung von HTML Daten
- javax.swing.text.html.parser
 - Interpretation und Analyse von HTML Daten
- javax.swing.text.rtf
 - Darstellung und Formatierung von Dokumenten im Rich text Format (RTF)

Paketstruktur (5/5)

- javax.swing.tree
 - Modelle und Sichten für Baumkomponenten
 - z.B. Dateisystem, Eigenschaftsbaum
- javax.swing.undo
 - Klassen und Schnittstellen für rücksetzbare Interaktionen (undo)

Klassenhierarchie

- ähnlich aufgebaut wie AWT
- jede Swing Komponente, die ein Äquivalent in AWT hat, besitzt einen äquivalenten Namen
 - z.B. `java.awt.Frame` vs. `javax.swing.JFrame`
 - Regel: J + AWT-Name
 - äquivalente Komponenten sind Unterklassen der AWT Komponenten
- Ausnahme für äquivalentes Verhalten: `MenuBar` vs. `JMenuBar`
 - `JMenuBar` ist in Swing Unterklasse von `javax.swing.JComponent` (und damit vom `java.awt.Component`)

Klassenhierarchie

- neue Struktur für Knöpfe (button)
 - JToggleButton als Knopf mit zwei Zuständen
 - gedrückt/ungedrückt
 - Zustand gedrückt wird gehalten, bis Knopf nochmal gedrückt wird
- JToggleButton ist Oberklasse von JRadioButton und JCheckBox

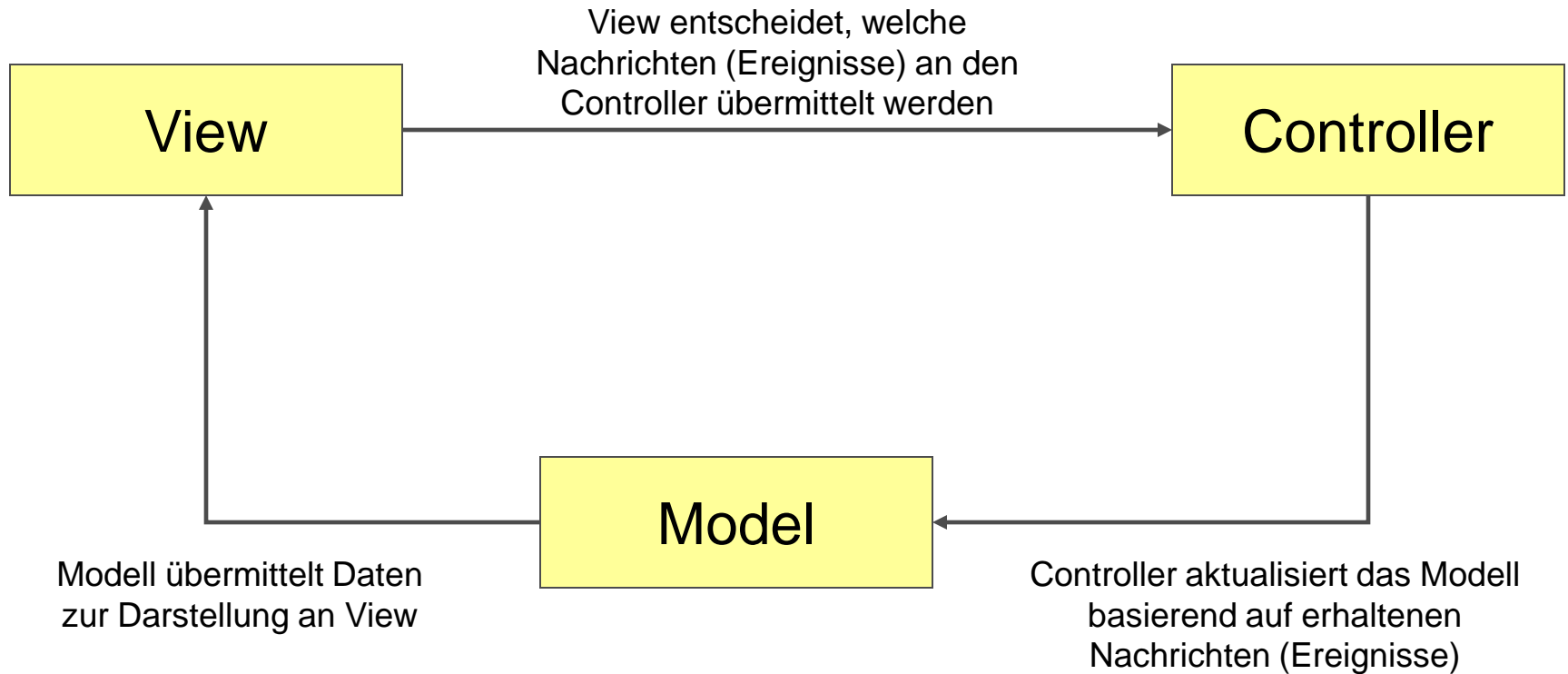
Diskussion: Unterschiede zu AWT

- Diskutieren Sie folgende Fragen:
 - Welche Swing Komponenten können in AWT GUI verwendet werden? Definieren Sie ein Kriterium!
 - Welche Bedeutung hat es, dass javax.swing.JMenuBar Unterklasse von javax.swing.JComponent ist?
 - Inwieweit verhalten sich AWT Menüs anders als Swing Menüs?

MVC Paradigma

- Model-View-Controller Entwurfsmuster
- Model (Modell) speichert die Zustandsdaten der Komponente
- View (Darstellung) legt fest, wie das Model dargestellt wird
- Controller (Bearbeiter) verwaltet die Interaktion und bietet Schnittstellen für Nachrichten an Model und View

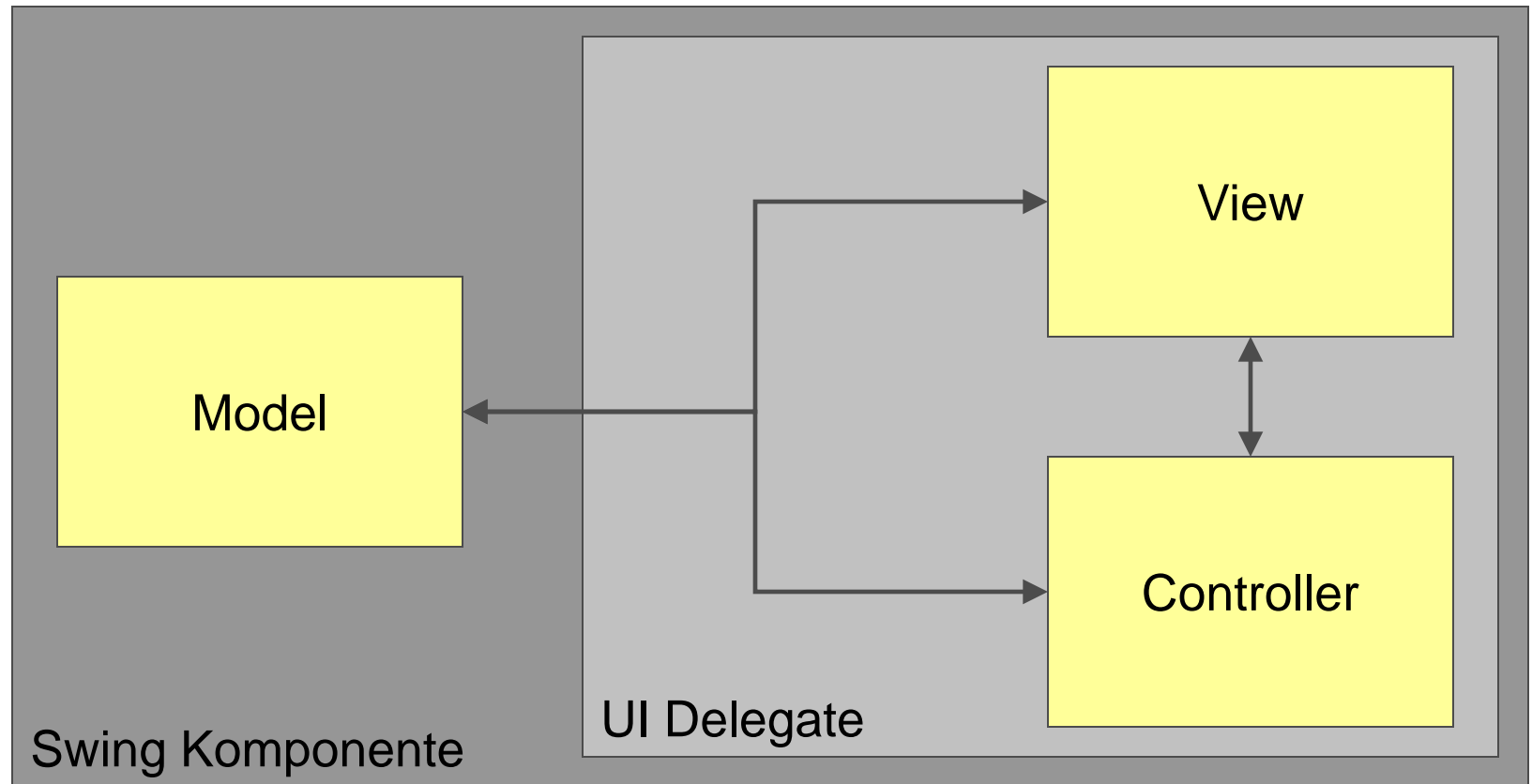
MVC Paradigma



MVC mit Swing

- Swing verwendet eine vereinfachte MVC Implementierung
 - model-delegate (Modell-Delegierer)
 - ein View und ein Controller werden als gemeinsamer UI delegate implementiert
- Vorteil: einfachere Darstellungsmöglichkeiten von Interaktionselementen ohne Seiteneffekte für die Zustandsdaten
 - wird z.B. von PLAF verwendet

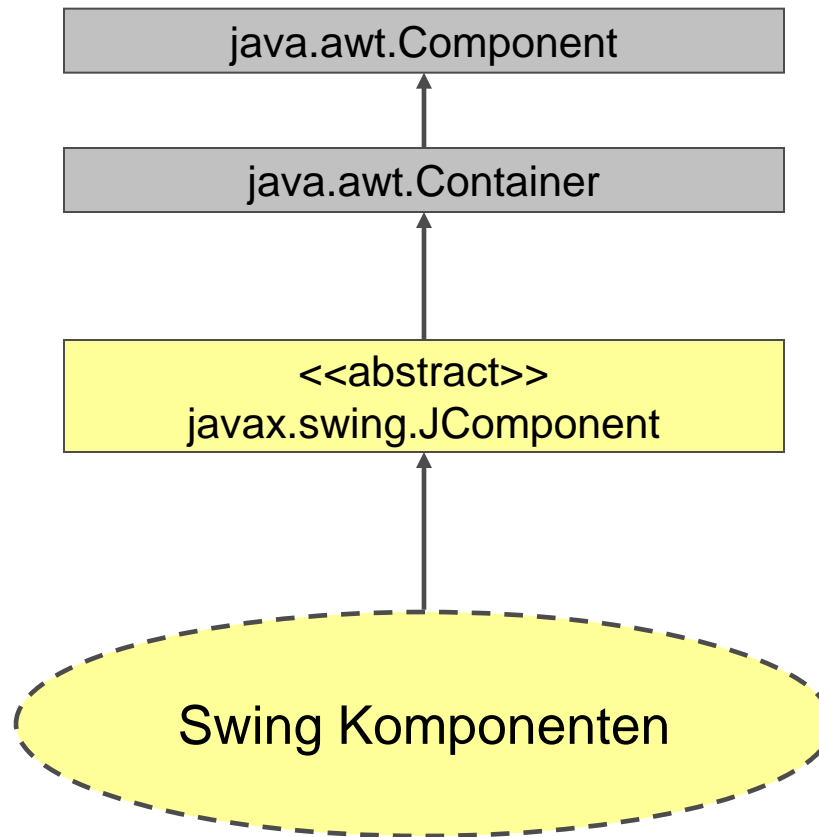
MVC mit Swing



JComponent

- abstrakte Oberklasse aller Swing Komponenten
 - Rahmenfunktionen für alle konkreten Implementierungen
- direkte Unterklasse von `java.awt.Container`
- können anderer Swing oder AWT Komponenten enthalten
 - parent
 - children

JComponent



JComponent

- wichtigste Methoden in JComponent
 - add
 - Hinzufügen von Komponenten
 - remove
 - Entfernen von Komponenten
 - pack
 - Optimieren der Anordnung
 - validate
 - Neuberechnung der optimalen Komponentengrößen
 - invalidate
 - Invalidierung der aktuellen Größenberechnung

JComponent

- setVisible
 - Darstellen der GUI
- show
 - veraltet, heute wird setVisible verwendet
- paint
 - interne Implementierung der Darstellung
- repaint
 - von außen steuerbares Neuzeichnen

JComponent: add, remove

- add
 - Hinzufügen einer Komponente in den Behälter
 - optional mit Angabe von Bedingungen
 - optional mit Angabe eines Index
- remove
 - Entfernen einer Komponente aus einem Behälter
 - Angabe des Index
 - Angabe der Komponente
 - Löschen aller Komponenten: Behälter wird vollständig geleert.

JComponent: pack

- Methode geerbt aus java.awt.Window
- Fenster wird auf die Größe gebracht, die zur Darstellung der enthaltenen Komponenten erforderlich ist
 - erforderliche Größe hängt direkt von der aktuellen Darstellung (LayoutManager) ab
- Aufruf, nachdem Behältern oberster Ebene Komponenten hinzugefügt wurden
 - JApplet
 - JFrame
 - JDialog
 - JWindow

JComponent: invalidate

- invalidate signalisiert einem Behälter, dass seine enthaltenen Komponenten neu gestaltet werden müssen
 - z.B. durch Änderung von Eigenschaften
 - wird häufig automatisch von den Komponenten ausgelöst
 - nicht alle Eigenschaftsänderungen erfordern den Aufruf von invalidate

JComponent: validate

- validate gestaltet den Behälter und die darin enthaltenen Komponenten neu und löst Neuzeichnen aus
 - wichtig bei Komponenten, die in einen bereits angezeigten Behälter hinzugefügt werden
 - oder aus einem angezeigten Behälter entfernt wurden

JComponent: setVisible, show

- setVisible(boolean visible) zeigt einen Behälter an
 - visible=true: anzeigen
 - visible=false: nicht anzeigen
- ersetzt die veraltete Methode show(boolean visible) aus AWT
 - konform mit JavaBeans: nur set/get Methoden

JComponent: paint

- paint wird von java.awt.Component vererbt
 - aus Kompatibilitätsgründen: Nachricht wird an UI-delegate weitergeleitet
- Behälter hat 3 Aufgaben
 - sich selbst zeichnen
 - paintComponent
 - seine Ränder zeichnen
 - paintBorder
 - Nachricht an enthaltene Komponenten weiterreichen
 - paintChildren

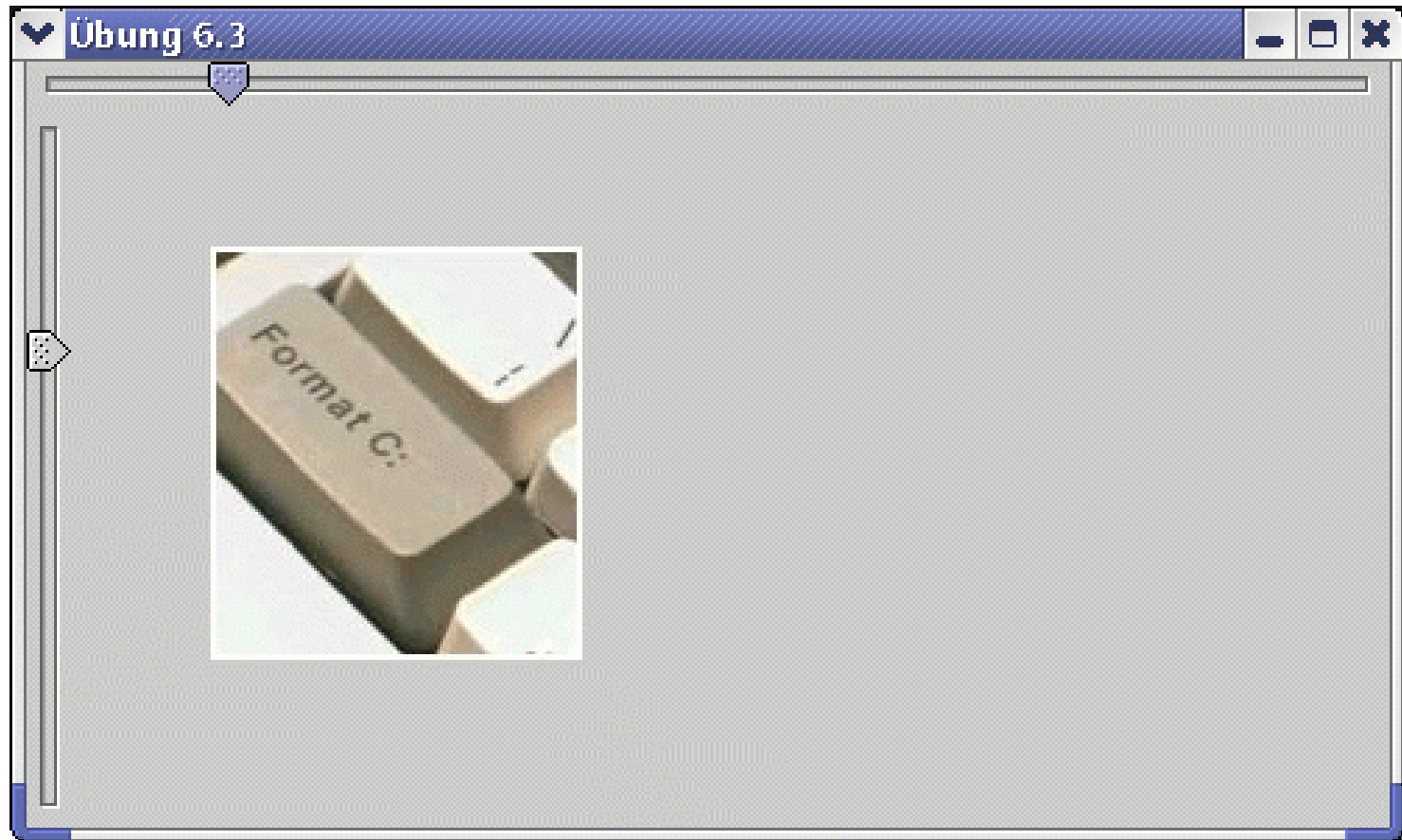
JComponent: paint

- paint nur dann überschreiben, wenn super.paint() aufgerufen wird!
 - wenn alternative Möglichkeit für Neuzeichnen benötigt wird
 - Komplexität der Behälterstruktur erfordert Aufruf der Supermethode

JComponent: repaint

- Swing verwendet einen Repaint Manager zur Verwaltung von Zeichnungsbereichen
 - repaint zeichnet genau den Zeichenbereich der Komponente neu
 - repaint(x,y,w,h) zeichnet einen Teilbereich der Komponente neu
- Neuzeichnen immer über repaint auslösen
 - benutzt den Repaint Manager
 - javax.swing.RepaintManager
- Repaint Manager optimiert die Neuzeichnung der GUI

Übung: freies Layout



Übung: freies Layout

- Funktionsbeschreibung:
 - das angezeigte Bild lässt sich mit den Schieberegler in X- und Y-Richtung bewegen
 - die Bereiche des Reglers werden so gewählt, daß das Bild immer vollständig sichtbar bleibt
 - Klicken auf den Fensterschließer beendet das Programm
 - Die Anzeige des Bildes startet an Position (50,50);

Übung: freies Layout

- Hinweise
 - verwenden Sie BorderLayout mit einem JPanel in der Mitte
 - nutzen Sie JLabel kombiniert mit ImageIcon zur Anzeige des Bildes
 - wählen Sie für das JPanel XY-Layout mit setLayout(null)
 - Komponenten lassen sich dann mit setLocation(x,y) direkt positionieren
 - das Bild sollte nicht größer sein als 100x100 Pixel
 - Lesen die die API Dokumentation
 - JComponent, JFrame, JPanel, ImageIcon, JSlider, ChangeEvent

Tabellen

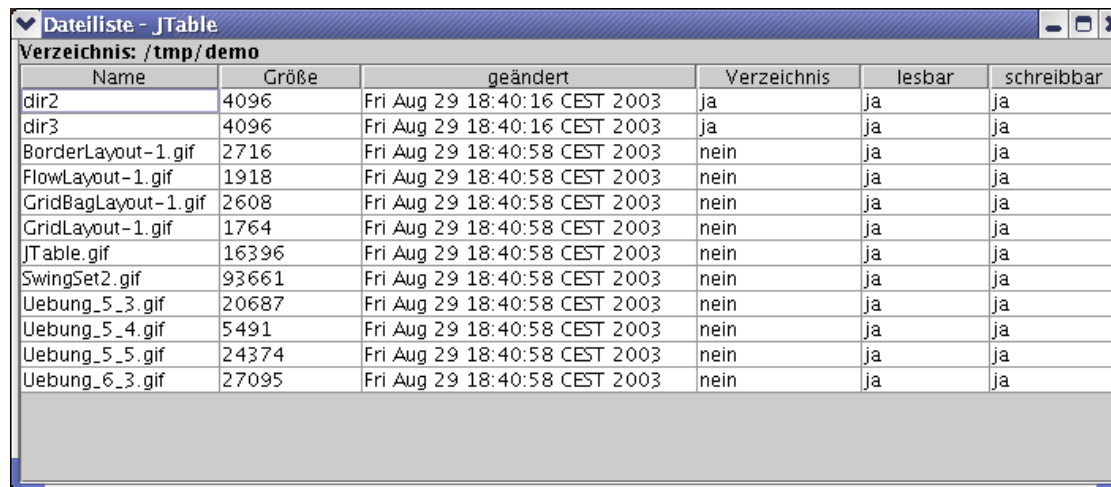
- Tabellen sind häufigst verwendete Form, Daten darzustellen
- Swing bietet mit JTable eine sehr flexible Implementierung für Tabellen
- Tabellen mit beliebiger Konfiguration werden als eine Komponente angesehen
 - JTable ist direkte Unterklasse von JComponent
- Tabellenfelder werden mit zweidimensionalem Array von Objekten gefüllt
 - jeder beliebige Datentyp darstellbar
 - Wrapper-Klassen für primitive Datentypen

Tabellen

- Kopfzeilen werden als Array von String-Objekten beschrieben
- JTable übernimmt die Darstellung der Tabelle
 - konfigurierbar durch Randbedingungen
- Auswählbarkeit von Zellen, Zeilen und Spalten kann präzise konfiguriert werden
- Tabellen werden nicht zellenweise, sondern spaltenweise verstanden
 - Tabellenmodell (table model)
 - Zustand jeder Zelle
 - Spaltenmodell (column model)
 - Zustand jeder Spalte

Beispiel: Inhaltsverzeichnis anzeigen

- eine einfache Tabelle
 - zeigt Einträge in einem Verzeichnis an
 - Tabelle passt sich an die Größe des Fensters an
 - bei Bedarf werden horizontale Schieber erzeugt



The screenshot shows a Java Swing window titled "Dateiliste - JTable". Inside the window is a table displaying a directory listing for the path "/tmp/demo". The table has six columns: "Name", "Größe", "geändert", "Verzeichnis", "lesbar", and "schreibbar". The rows list various files and directories, including "dir2", "dir3", "BorderLayout-1.gif", "FlowLayout-1.gif", "GridBagLayout-1.gif", "GridLayout-1.gif", "JTable.gif", "SwingSet2.gif", "Uebung_5_3.gif", "Uebung_5_4.gif", "Uebung_5_5.gif", and "Uebung_6_3.gif". Each row shows the file name, its size in bytes, the last modified date and time, and permissions for reading and writing.

Name	Größe	geändert	Verzeichnis	lesbar	schreibbar
dir2	4096	Fri Aug 29 18:40:16 CEST 2003	ja	ja	ja
dir3	4096	Fri Aug 29 18:40:16 CEST 2003	ja	ja	ja
BorderLayout-1.gif	2716	Fri Aug 29 18:40:58 CEST 2003	nein	ja	ja
FlowLayout-1.gif	1918	Fri Aug 29 18:40:58 CEST 2003	nein	ja	ja
GridBagLayout-1.gif	2608	Fri Aug 29 18:40:58 CEST 2003	nein	ja	ja
GridLayout-1.gif	1764	Fri Aug 29 18:40:58 CEST 2003	nein	ja	ja
JTable.gif	16396	Fri Aug 29 18:40:58 CEST 2003	nein	ja	ja
SwingSet2.gif	93661	Fri Aug 29 18:40:58 CEST 2003	nein	ja	ja
Uebung_5_3.gif	20687	Fri Aug 29 18:40:58 CEST 2003	nein	ja	ja
Uebung_5_4.gif	5491	Fri Aug 29 18:40:58 CEST 2003	nein	ja	ja
Uebung_5_5.gif	24374	Fri Aug 29 18:40:58 CEST 2003	nein	ja	ja
Uebung_6_3.gif	27095	Fri Aug 29 18:40:58 CEST 2003	nein	ja	ja

Beispiel:

Inhaltsverzeichnis anzeigen

```
import java.awt.*;  
import javax.swing.*;  
import java.io.*;  
import java.util.*;
```

```
public class DirectoryTable extends javax.swing.JFrame {  
    String[] header = new String[] { "Name", "Größe",  
        "geändert", "Verzeichnis", "lesbar", "schreibbar" };  
    String dirName="/tmp/demo";  
    public static void main(String[] args) {  
        DirectoryTable dt = new DirectoryTable();  
    }  
}
```

Beispiel:

Inhaltsverzeichnis anzeigen

```
public DirectoryTable() {
    super("Dateiliste - JTable");
    setDefaultCloseOperation(EXIT_ON_CLOSE);
    setSize(500,300);
    File tmp = new File(dirName);
    Object[][] directory = createDirectoryStatus(tmp);
    JTable table = new JTable(directory,header);
    table.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
    table.setCellSelectionEnabled(true);
    JLabel label = new JLabel("Verzeichnis: " + dirName);
    getContentPane().add(label,BorderLayout.NORTH);
    JScrollPane pane = new JScrollPane(table);
    getContentPane().add(pane,BorderLayout.CENTER);
    setVisible(true);
}
```


Beispiel:

Inhaltsverzeichnis anzeigen

```
private Object[][] createDirectoryStatus(File dir) {
    String[] fileList = dir.list();
    Object[][] dirStatus =
        new Object[fileList.length][header.length];
    File f;
    for (int i=0;i<fileList.length;i++) {
        f = new File(dirName,fileList[i]);
        dirStatus[i][0] = f.getName();
        dirStatus[i][1] = new Long(f.length());
        dirStatus[i][2] = new Date(f.lastModified());
        dirStatus[i][3] = f.isDirectory()?"ja":"nein";
        dirStatus[i][4] = f.canRead()?"ja":"nein";
        dirStatus[i][5] = f.canWrite()?"ja":"nein";
    } return dirStatus; }}
```

Tabelleneigenschaften: Zellen und Spalten

- Properties haben set/get/is Methoden
- `autoCreateColumnsFrom Model`
 - boolean, {set,get}, false
 - entscheidet, ob Spaltenmodell automatisch mit Daten des Tabellenmodells geladen werden
 - true, wenn kein explizites Spaltenmodell angegeben (`setColumnModel`) wurde
- `autoResizeMode`
 - int, {set, get}, `AUTO_RESIZE_ALL_COLUMNS`
 - definiert, wie Tabelle auf Größenänderung reagiert
 - automatische Größenänderung ausschalten, wenn Schieber verwendet werden sollen

Tabelleneigenschaften: Zellen und Spalten

- `columnCount`, `rowCount`
 - `int`, `{get}`, 0
 - Zahl der Spalten/Zeilen erfragen
- `columnModel`
 - `TableColumnModel`, `{set, get}`, `DefaultTableColumnModel()`
 - Referenz auf das verwendete Spaltenmodell
 - zur Laufzeit austauschbar
- `model`
 - `TableModel`, `{set, get}`, `DefaultTableModel()`
 - Referenz auf das verwendete Tabellenmodell
 - zur Laufzeit austauschbar

Tabelleneigenschaften: Zellen und Spalten

- rowHeight
 - int, {set, get}, 16
 - Pixelwert für die Höhe der Zellen
 - Wert muss größer 1 sein
 - IllegalArgumentException

Tabelleneigenschaften: Selektionsmodell

- selectionModel
 - ListSelectionModel, {set, get}, DefaultListSelectionModel()
 - Referenz zum aktuellen Selektionsmodell für Tabellenzeilen
 - Spaltenselektion geschieht über TableColumnModel
- cellSelectionEnabled, columnSelctionAllowed, rowSelectionAllowed
 - boolean, {set, get}, {false, false, true}
 - Selektionsmöglichkeiten für Zeilen bzw. Spalten

Tabelleneigenschaften: Selektionsmodell

- selectedColumn, selectedRow
 - int, {get}, -1
 - erste Selektion der Zeile/Spalte des jeweiligen Modells
- selectedColumns, selectedRows
 - int[], {get}, int[0]
 - Indizes der selektierten Zeilen/Spalten
- selectColumnCount, selectedRowCount
 - int, {get}, 0
 - Zahl der selektierten Zeilen/Spalten

weitere Tabelleneigenschaften

- weitere Properties
 - Hintergründe
 - Farben
 - Ränder
 - Gitter
- UI
 - TableUI, {set, get}, [durch PLAF definiert]
 - UI-delegate für Tabelle
- cellEditor
 - TableCellEditor, {set, get}, null

Tabellen: weitere Methoden

- `add{Column|Row}SelectionInterval(int index0, int index1)`
 - erweitert die Selektion um Intervall `[index0,index1]`
- `clearSelection()`
 - löscht Selektionen in allen Reihen und Spalten
- `int {column|row}AtPoint(Point p)`
 - liefert Zeilen- bzw. Spaltenindex einer XY Koordinate
 - Punkt aus dem Zeichenbereich der JTable!
 - (0,0) ist linke obere Ecke der Tabelle

Tabellen: weitere Methoden

- `boolean editCellAt(int row, int column, EventObject event)`
 - startet Editieren einer Zelle
 - bei Bedarf ein Ereignis erzeugen und an Editor übergeben
- `moveColumn{int column, int targetColumn)`
 - verschiebt Spalten der Tabelle
 - kann vom Anwender bei entsprechendem LayoutManager der Tabelle vorgenommen werden

Tabellen: weitere Methoden

- `remove{Column|Row}SelectionInterval(int index0, int index1)`
 - entfernt Selektion einer Zeile bzw. Spalte im Intervall `[index0,index1]`
- `selectAll()`
 - selektiert alle Zeilen und Spalten der Tabelle

Tabellen: wichtige Klassen/Schnittstellen

- TableColumn {class}
 - Aufbau von einzelnen Zellen
 - Zugriff auf alle notwendigen Zelleigenschaften
- TableColumnModel {interface}
 - nicht verwechseln mit TableColumn!
 - verwaltet Selektionen in der Spalte und Zellabstände
 - Zugriff auf Tabellenspalte über Collection
- DefaultTableColumnModel {class}
 - Standardimplementierung des TableColumnModel in JTable

Tabellen: wichtige Klassen/Schnittstellen

- TableColumnModelEvent {class}
 - enthält Informationen, welche Spalten von dem Ereignis betroffen sind
- TableColumnModelListener {interface}
 - Hörer-Implementierung für Ereignisse vom Typ TableColumnModelEvent
- TableModel {interface}
 - Basis für alle Tabellenmodelle
 - Informationen über die Tabelle
- AbstractTableModel {abstract class}
 - abstrakte Implementierung des Tabellenmodells
 - Tabellenmodelle damit leicht erzeugbar!

Tabellen: wichtige Klassen/Schnittstellen

- DefaultTableModel {class}
 - Standardimplementierung des Tabellenmodells
 - erweitert AbstractTableModel
- TableModelEvent {class}
 - Ereignis zur Signalisierung von Änderungen im Tabellenmodell
- TableModelListener {interface}
 - Hörer-Implementierung für Ereignisses des Tabellenmodells
- JTableHeader {class}
 - Implementierung für Tabellenüberschriften

Tabellen:

wichtige Klassen/Schnittstellen

- TableCellRenderer {interface}
 - Schnittstelle für Darstellung von Zellen
- DefaultTableCellRenderer {class}
 - Standardimplementierung für Zellendarstellung
 - nutzt JLabel zur Darstellung von
 - Text, Zahlen, Icons, Objekte (mit toString() Methode)
- CellRendererPane {class}
 - effizientes Zeichnen/Neuzeichnen von Zellen
 - nutzt NICHT repaint
- CellEditor {interface}
 - definiert Basisfunktionen eines Zelleditors

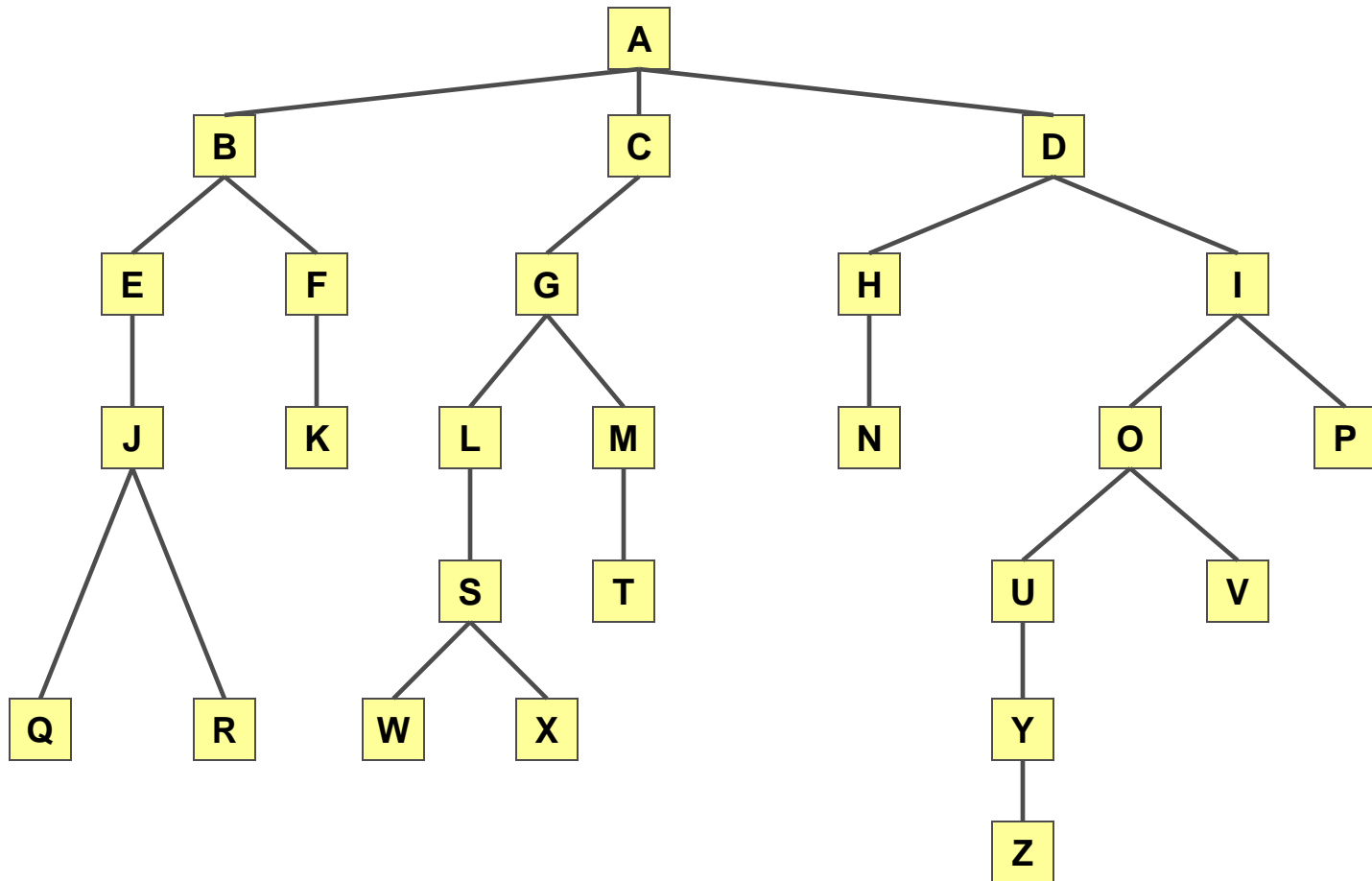
Tabellen: wichtige Klassen/Schnittstellen

- TableCellEditor {interface}
 - Schnittstelle für Zugriff auf Zelleneditor
- CellEditorListener
 - Hörer-Implementierung für Ereignisse des Zelleditors
 - hört auf Ereignisse vom Typ ChangeEvent
- DefaultCellEditor
 - Standardimplementierung eines Zelleditors

Bäume

- Bäume stellen hierarchische Informationen dar
 - z.B. Dateisystem, Kommandostrukturen
- Swing implementiert Bäume in `javax.swing.JTree`
- JTree enthält verschiedene Modelle
 - Darstellung von Knoten
 - Auswahl von Knoten
 - Modifikation des Baums
- Die Implementierung von Bäumen ähnelt der Implementierung von Tabellen.

Bäume: ein Beispiel



Bäume: Terminologie

- Knoten (node)
 - Element im Baum
- Wurzel (root)
 - Ursprung des Baums
 - hat keinen Vorgänger
 - ist genau einmal im Baum enthalten
 - Beispiel: A ist die Wurzel des Baums
- Kind (child)
 - Knoten, die an einen anderen Knoten geknüpft sind
 - Beispiel: M ist Kind von G

Bäume: Terminologie

- Vater (parent)
 - Knoten, der oberhalb seiner verknüpften Knoten steht
 - Beispiel: C ist Vater von G
- Geschwister (sibling)
 - jedes beliebige Kind eines Knotens
 - Beispiel: A ist sein eigenes Geschwister, Geschwister von B sind B, C und D
- Nachkomme (descendant)
 - jedes Kind, Kind eines Kindes, etc.
 - ein Knoten ist definiert als sein eigener Nachkomme
 - Beispiel: L, S, W und X sind Nachkommen von L

Bäume: Terminologie

- Vorfahr (ancestor)
 - jeder Vater, Vater eines Vaters, etc.
 - ein Vater ist definiert als sein eigener Vorfahr
 - Beispiel: A, D, I und P sind Vorfahren von P
- Niveau (level)
 - Zahl der Nachkommen von der Wurzel bis zu einem Knoten
 - das größte Niveau heißt Tiefe, bzw. Höhe des Baums
 - Beispiel: A hat Niveau 0, T hat Niveau 4

Bäume: Terminologie

- Pfad (path)
 - verknüpfte Menge von Knoten absteigenden Niveaus
 - häufig von der Wurzel aus betrachtet
 - {A, B, E, J, Q} ist ein Pfad
- Reihe (row)
 - graphischer Abstand eines Knotens zur Wurzel
 - entspricht nicht dem Niveau!
 - hängt von der Implementierung des Baums ab!
- eingeklappt (collapsed)
 - ein Knoten ist eingeklappt, wenn keines seiner Kinder dargestellt ist

Bäume: Terminologie

- ausgeklappt (expanded)
 - ein Knoten ist ausgeklappt, wenn seine Kinder dargestellt sind
- sichtbar (visible)
 - ein Knoten ist sichtbar, wenn er (ohne einen Vater auszuklappen) dargestellt ist
 - dargestellte und im Fenster weggerollte Knoten werden ebenfalls als sichtbar definiert

Bäume: die Klasse JTree

- direkte Unterklasse von javax.swing.JComponent
- JTree erzeugt einen Baum aus verschiedenen Objekten, inklusive dem TreeModel
 - Erscheinungsbild des Baums wird über Properties {get/set/is}XXX konfiguriert, .z.B.
 - isExpanded
 - {set/get}Model
 - isEditable
- JTree kann viele Arten von Ereignissen erzeugen

Bäume:

Methoden der Klasse JTree

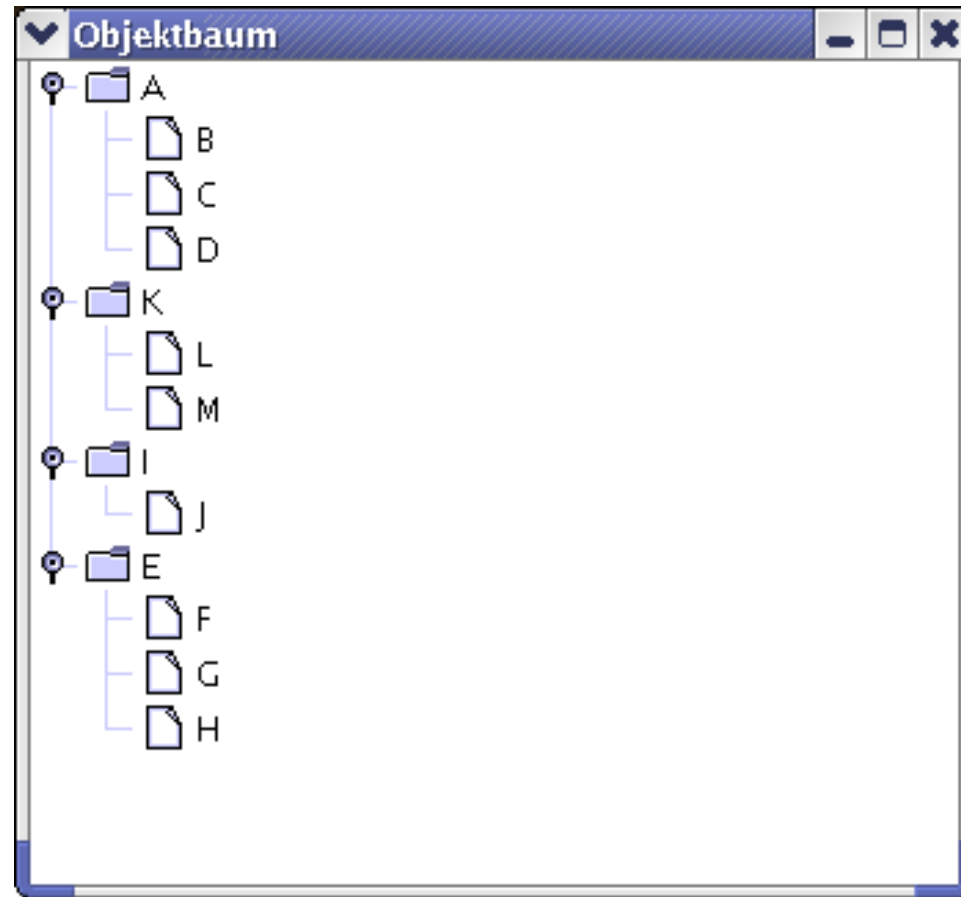
- Auswahl
 - Selection methods
 - Zugriff auf die Knotenauswahl des Baums
 - Verhalten kann durch ein Selektionsmodell verfeinert werden
- Einklappen/Ausklappen
 - collapse/expand methods
 - Zustand des Baums bezüglich Einklappen/Ausklappen von Knoten

Bäume:

Methoden der Klasse JTree

- Pfade und Reihen
 - path/row methods
 - Information über Pfade und Reihen des Baums
 - Größe
 - Tiefe
 - Sichtbarkeit
- Editieren
 - edit methods
 - Methoden aktivieren / deaktivieren Pfade des Baums für Editierung der Knoten

Beispiel: Objektbaum



Beispiel: Objektbaum

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import javax.swing.tree.*;
import java.util.*;

public class ObjectTree extends javax.swing.JFrame {

    private JTree tree;
    private String[][] data = {
        {"A"}, {"B", "C", "D"},
        {"E"}, {"F", "G", "H"},
        {"I"}, {"J"},
        {"K"}, {"L", "M"}
    };
};
```

Beispiel: Objektbaum

```
public ObjectTree() {  
    super("Objektbaum");  
    setSize(400,300);  
    init();  
    setDefaultCloseOperation(EXIT_ON_CLOSE);  
    setVisible(true);  
}  
  
private void init() {  
    Hashtable h = new Hashtable();  
    for (int i=0; i<data.length; i+=2) {  
        h.put(data[i][0], data[i+1]);  
    }  
    tree = new JTree(h);  
    getContentPane().add(tree, BorderLayout.CENTER);  
}
```

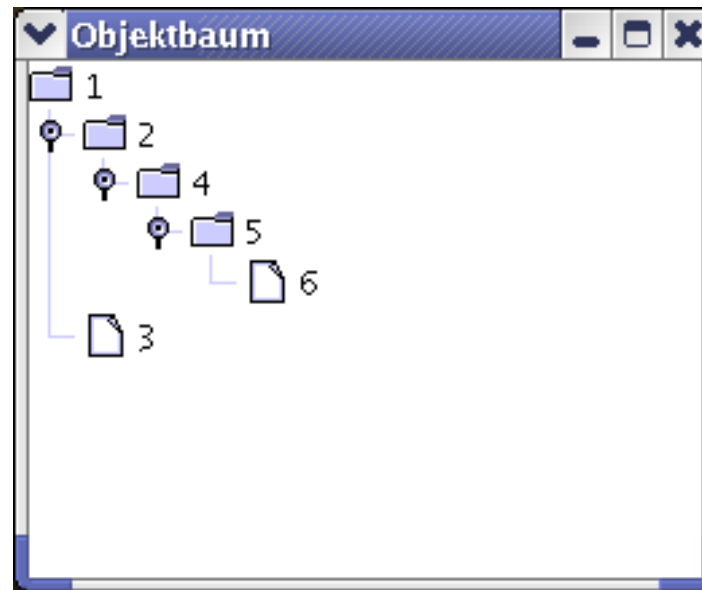
Beispiel: Objektbaum

```
public static void main(String[] args) {  
    ObjectTree t = new ObjectTree();  
}  
  
}
```

Übung: Bäume

- Betrachten Sie das Beispiel Objektbaum.
 - Wie funktioniert die Methode `init()`?
 - Wie müssen Sie `init()` modifizieren, um Bäume größerer Tiefe zu erzeugen?
- Implementieren Sie einen Baum der Tiefe 4 wie vorgegeben.
- Erarbeiten Sie die Lösung mittels Java API Dokumentation.

Übung: Bäume



Bäume:

die Schnittstelle TreeModel

- TreeModel Implementierung definiert das Grundverhalten des Baums
 - enthält Zeiger auf die Wurzel
 - einziger Knoten ohne Vorgänger
 - wertet Ereignisse vom Typ `javax.swing.event.TreeModelEvent` aus
- Standardimplementierung des TreeModel:
`javax.swing.DefaultTreeModel`
 - implementiert TreeModel und weitere Methoden
 - erweiterbar für Spezialisierung
 - z.B. Binärbäume

Bäume:

die Schnittstelle TreeModel

- `Object getChild(Object parent, int index)`
 - liefert den Nachfolger eines Knotens parent an gegebener Position index
- `int getChildCount(Object parent)`
 - liefert die Zahl der Nachfolger eines Knotens parent
 - 0 für Blätter
- `int getIndexOfChild(Object parent, Object child)`
 - liefert den Index eines Nachfolgers child für einen Knoten parent

Bäume: die Schnittstelle TreeModel

- boolean isLeaf(Object node)
 - testet, ob Knoten ein Blatt ist
 - unterscheidet nicht zwischen echten Blättern und momentanen Blättern!
 - echte Blätter könne nie einen Nachfolger besitzen
- void valueFromPathChanged(TreePath path, Object newValue)
 - benachrichtigt einen Pfad über die Änderung eines benutzerdefinierten Knotens newValue

Bäume:

die Schnittstelle TreeNode

- Basismethoden für Knoten
- definiert Eigenschaften des Knotens mit Properties
 - allowsChildren, {get}, boolean
 - childAt, {get}, TreeNode
 - childCount, {get}, int
 - leaf, {is}, boolean
 - parent, {get}, TreeNode

Bäume:

die Schnittstelle MutableTreeNode

- spezialisiert TreeNode
- erweitert die Eigenschaften des Knotens um Methoden zur Manipulation
- neue Properties
 - parent, {get,set}, MutableTreeNode
 - userObject, {get}, Object

Bäume:

die Klasse DefaultMutableTreeNode

- implementiert MutableTreeNode
- bietet zusätzlich Methoden zur Strukturierung von Knoten
 - Knoten hinzufügen/verknüpfen
 - Durchlaufen des Baums
 - Tiefenberechnung
 - Extrahierung von Pfaden

Bäume: weitere Klassen und Schnittstellen

- Klasse `TreePath`
- Schnittstelle `TreeSelectionModel`
- Schnittstelle `RowMapper`
- Klasse `DefaultTreeSelectionModel`
- Klasse `TreeModelEvent`
- Schnittstelle `TreeModelListener`
- Klasse `TreeSelectionEvent`
- Schnittstelle `TreeSelectionListener`
- Klasse `TreeExpansionEvent`

Bäume: weitere Klassen und Schnittstellen

- Schnittstelle `TreeExpansionListener`
- Klasse `DefaultTreeCellRenderer`
- Schnittstelle `TreeCellRenderer`
- Klasse `DefaultTreeCellEditor`
- Schnittstelle `TreeCellEditor`

weitere wichtige Swing API

- UNDO
 - Rückgängigmachen/Wiederherstellen von Modifikationen
 - zur Zeit beschränkt auf Textkomponenten in `javax.swing.text`
- PLAF (Pluggable-Look-And-Feel)
 - beliebig definierbares und austauschbares Aussehen von Swing Fenstern
- DRAG-N-DROP
 - ziehen und ablegen ob Objekten in Java-Anwendungen
 - Auslösen der erforderlichen Programmaktion
 - z.B. Datei kopieren
- Interne Fenster
 - eigene Fensterverwaltung in Swing GUI

Eigenschaften einer Entwicklungsumgebung

- Integrierte Entwicklungsumgebungen bieten diverse Programmierwerkzeuge
 - Projektverwaltung
 - Quelltexteditor
 - Syntaxmarkierung
 - Datei-/Verzeichnisverwaltung
 - Versionsverwaltung
 - Übersetzung (compiling)
 - Fehlersuche (debugging)
 - Programmablauf (running)
 - GUI Editor
- integrated development environment (IDE)
 - auch development platform

Ziel einer IDE: Programmierung vereinfachen

- Programmentwicklung soll vereinfacht werden
 - Unterstützung bei der Auswahl von Klassen und Methoden
 - automatische Ergänzung im Quelltext
 - komfortable Tests
 - Ablaufanalyse
 - Haltepunkte
 - WYSIWYG GUI Entwicklung
 - what you see is what you get
 - Unterstützung von Projektgruppen
 - Versionsverwaltung

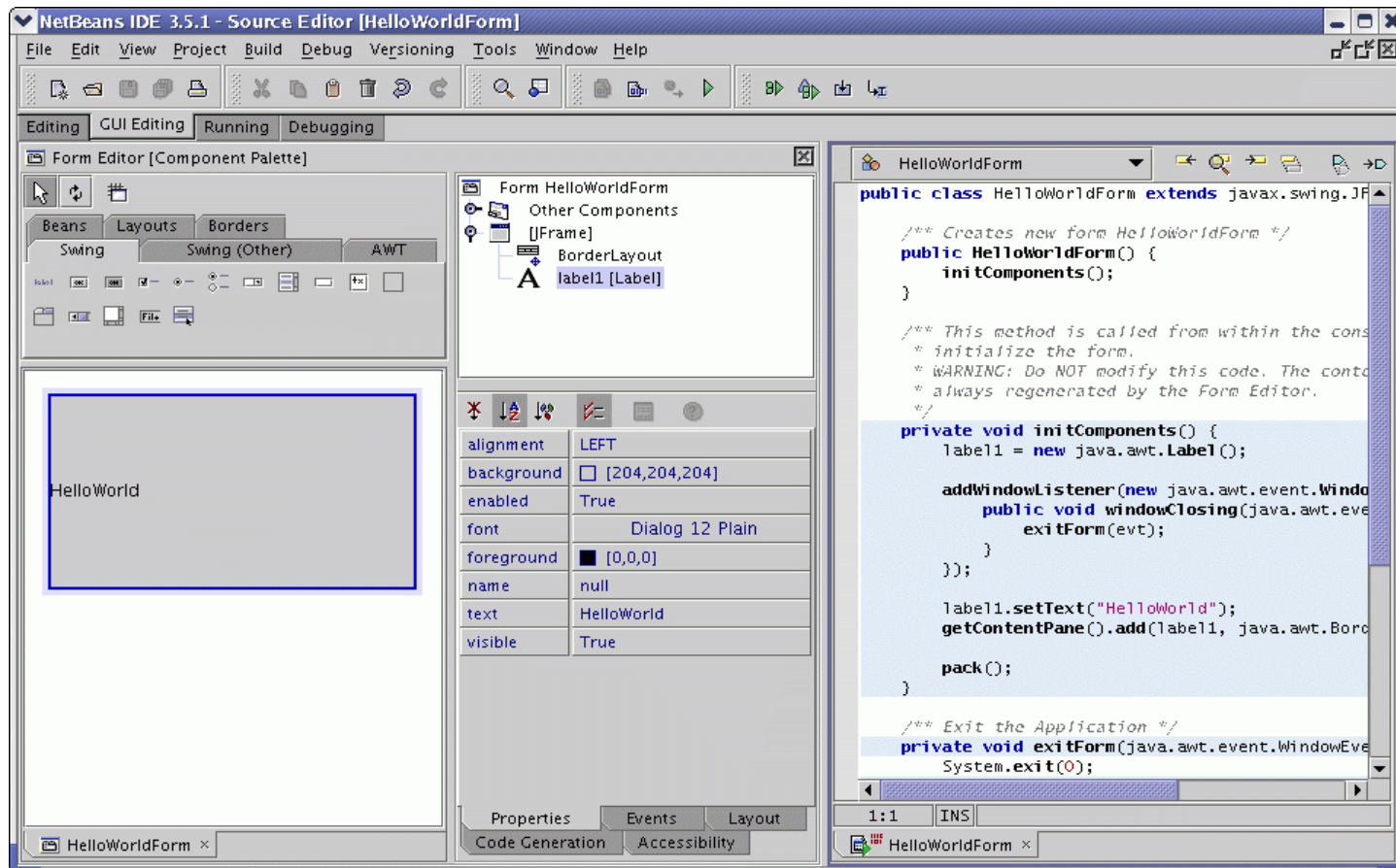
Nachteile

- IDE schränken häufig die Freiheit der Entwickler unnötig ein
 - z.B. Implementierung von Zuhörern gegen Verwendung von Adaptern
- Entwicklungsprojekte sind nicht standardisiert
- Programmentwicklungen sind nicht zwischen IDE austauschbar
- IDE haben selbstdefinierte GUI-Editor Formate

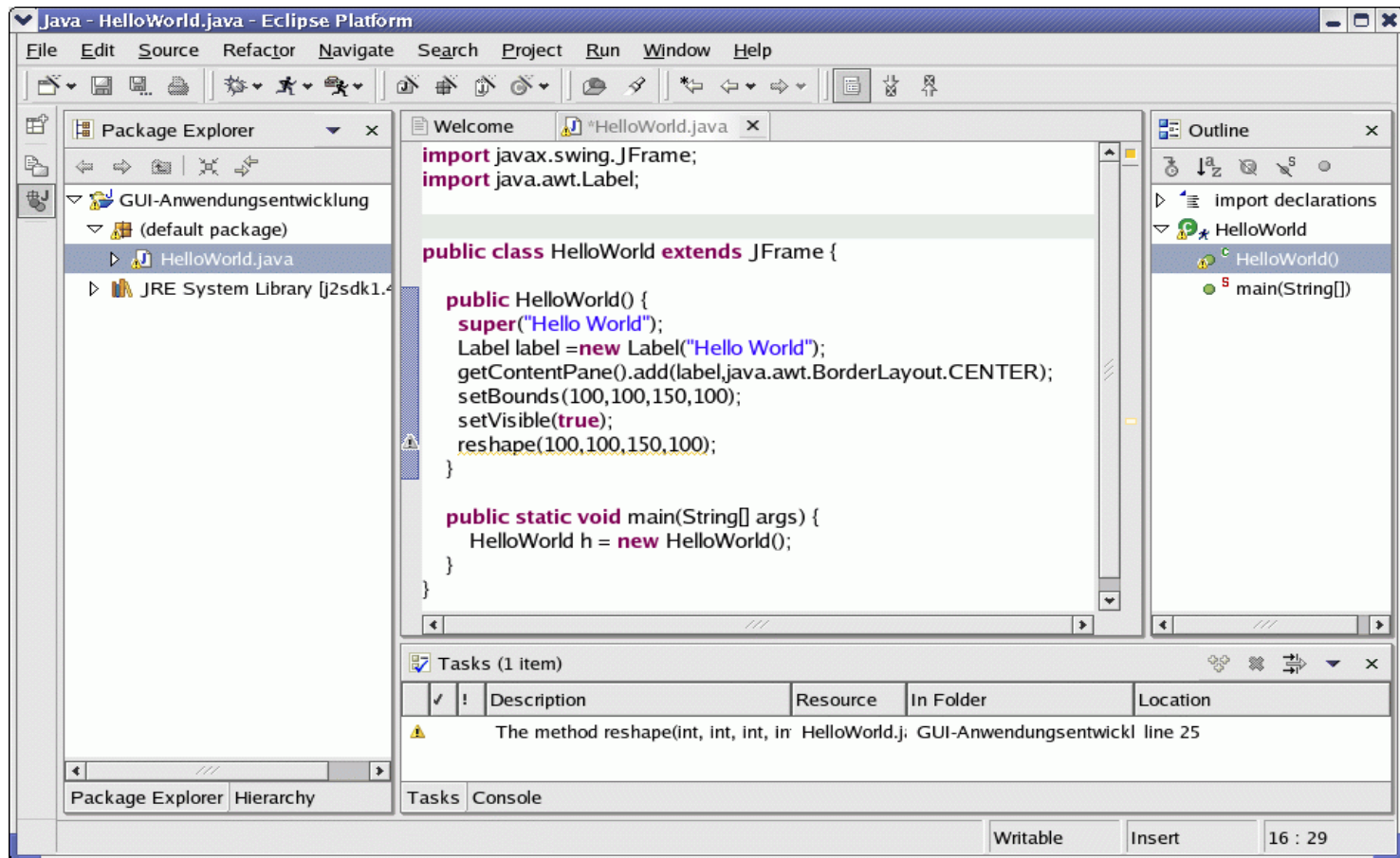
GUI Editor: NetBeans Form

- OpenSource Projekt
 - www.netbeans.org
 - Basis für Sun ONE Studio von Sun Microsystems
 - vollständige Java IDE
- bietet grafische Darstellung und äquivalenten Quelltext
 - Änderungen werden im jeweiligen Äquivalent automatisch eingefügt
 - WYSIWYG Editor
 - definiert strenge Regeln, nach denen Quelltext modifiziert werden darf

GUI Editor: NetBeans Form



IDE ohne GUI-Editor: Eclipse



eigene Swing Komponenten

- eigene Komponenten in eine GUI Bibliothek integrieren
 - eigene Lösungen ohne Standardisierung sind nicht wiederverwendbar!
 - also: Standard verwenden
- Analyse von Klassen: Reflection API
- JavaBeans Konvention
 - Serialisierbarkeit
 - definierte Eigenschaften (properties)
 - {set|get|is} Methoden

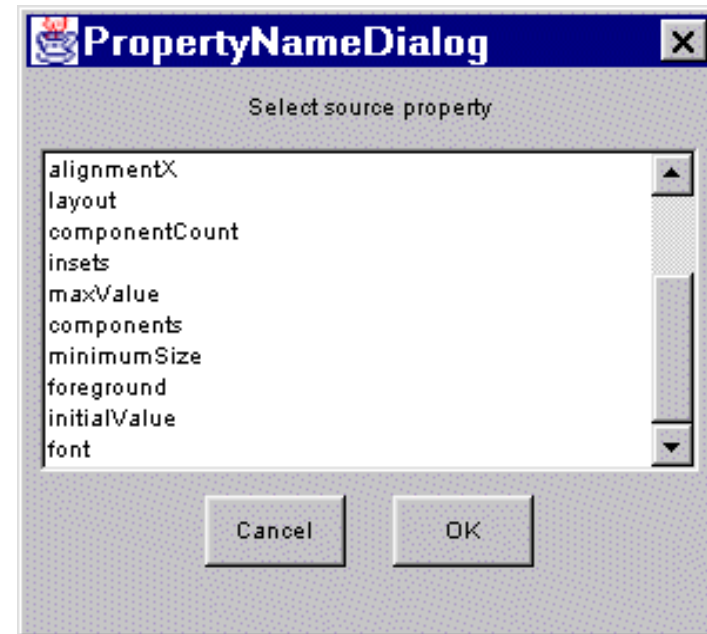
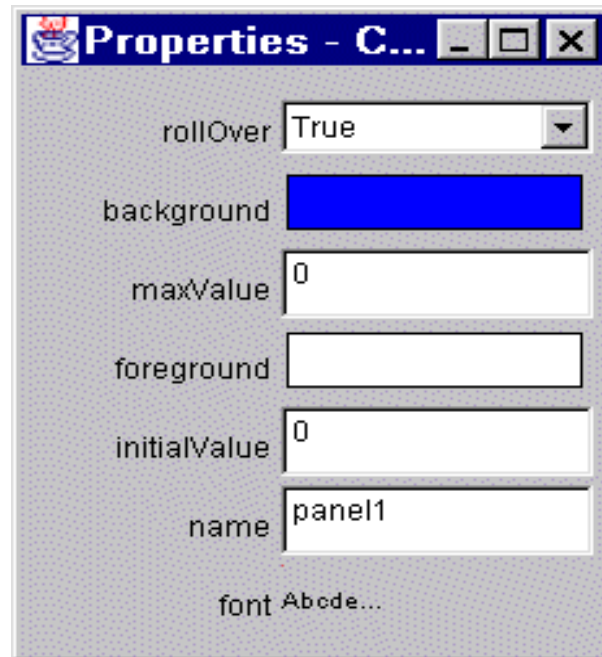
Zusatzinformationen für JavaBeans: BeanInfo Klassen

- GUI Komponenten sind JavaBeans
- BeanInfo Klassen beschreiben eigene GUI-Komponenten
 - Properties
 - Semantik, Wertebereiche
- Reflection API erlaubt eine vollautomatische Analyse von eigenen Komponenten
 - automatischer Aufbau eines Komponenten-Editors
 - Integration in vorhandene IDE
 - siehe: www.netbeans.org Dokumentation
- kein Unterschied in der IDE zwischen fremden und eigenen GUI-Komponenten

Zusatzinformationen für JavaBeans: BeanInfo Klassen

- JavaBeans API Paket: java.beans
 - in J2SDK 1.4 erweitert
- Schnittstelle java.beans.BeanInfo
 - Schnittstelle zur Beschreibung von Methoden, Eigenschaften, Ereignisse, etc. einer Bean
- Klasse java.beans.SimpleBeanInfo
 - leere Standardimplementierung als Basis für Spezialisierungen
- Klasse java.beans.Introspector
 - Werkzeug zur Analyse von JavaBeans
 - wird von den IDE zur Integration verwendet
 - Basis zur Erstellung eines Editors

Beispiel: automatisch erzeugte Editoren



Selbstkontrolle

- Worin unterscheiden sich AWT und Swing?
- Warum ist das Tabellenmodell spaltenorientiert?
- Wie können Sie ein Bild in einer Tabellenzelle anzeigen?
- Können Sie einen Schieberegler (engl. slider) als CellEditor für ein einziges Tabellenfeld einfügen? Wenn ja, wie?

Selbstkontrolle

- Wie können Sie eine zweite Darstellung (siehe MVC) mit einem Baum koppeln, so dass Änderungen im Baum sowohl in das Modell als auch in die zweite Ansicht übernommen werden?
- Warum verhindern GUI Editoren üblicherweise den Wechsel von IDEs?
- Warum reicht die Einhaltung des JavaBeans Standard aus, um eigene Swing Komponenten in WYSIWYG Editoren zu integrieren?

Kontakt

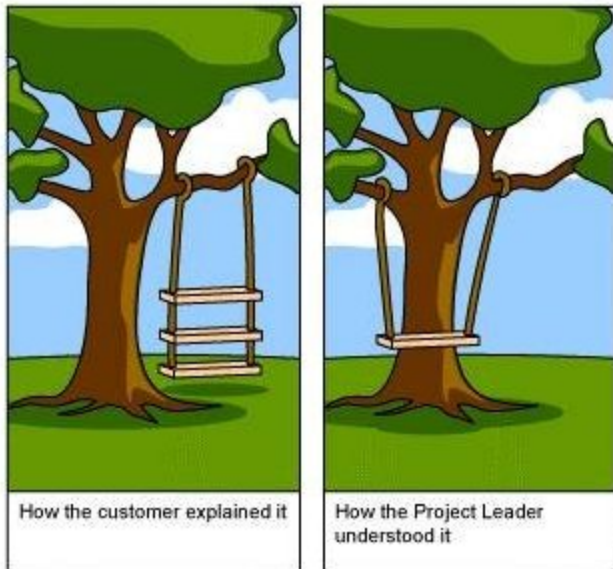
Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Software Engineering 1

Grafische Benutzerschnittstellen mit dem Standard Widget Toolkit (SWT)



Prof. Dr. Andreas Judt

Warum noch eine GUI Bibliothek?

- GUI Bibliothek für Java mit enger Anlehnung an das Fenstersystem des Betriebssystems
 - portabel
 - schließt die Lücke zu hochwertigen GUI-Anwendungen mit look-and-feel der darunter liegenden Betriebssysteme
- SWT will die Funktionalitäten der gängigen Fenstersysteme vereinigen
 - SWT emuliert fehlenden Funktionalität
- Fazit: die SWT Programmbibliothek ist plattformabhängig, bietet aber viele Vorteile in der Programmentwicklung

SWT und Eclipse

- SWT gehört zur Entwicklungsumgebung Eclipse
 - www.eclipse.org
 - OpenSource Projekt
 - Initiative von IBM
 - Eclipse ist beliebig erweiterbar und anpassbar
- SWT Bibliothek ist heute OpenSource
 - musste früher lizenziert werden
 - Quellen: www.eclipse.org/swt

SWT gegen AWT

- AWT nutzt Interaktionselemente (widgets), die auf allen gängigen Fenstersystemen implementiert sind
 - nur einfache GUI implementierbar
 - Strategie des kleinsten gemeinsamen Nenners
- Höherwertige Funktionalitäten der Fenstersysteme nicht zugreifbar
 - z.B. Bäume, Tabellen
 - kleinster gemeinsamer Nenner
- Aussehen und Größe der Interaktionselemente nicht präzise bestimmbar

SWT gegen Swing

- eigene Implementierung höherwertiger Interaktionselemente und Funktionalitäten
 - Emulation des Fenstersystems mit PLAF
 - Swing Anwendung können wie die des Fenstersystems aussehen, sich jedoch unterschiedlich verhalten
- PLAF Einstellungen von Swing sind unabhängig vom verwendeten Fenstersystem
 - PLAF basiert auf Emulation
 - flexibler als SWT
 - Emulation mit PLAF nicht perfekt, Anwender bemerken Unterschiede

SWT Philosophie: Funktionalitäten vereinigen

- Interaktionselemente werden von SWT emuliert, wenn sie nicht im Fenstersystem verfügbar sind
 - z.B. Bäume unter OSF/Motif 1.2
- selten verwendete Interaktionselemente werden in SWT nicht unterstützt
 - z.B. Kalender
- betriebssystemspezifische Funktionalitäten werden nicht unterstützt
 - z.B. ActiveX

Kochrezept: SWT GUI

- Display Objekt erzeugen
 - definiert den Startpunkt der Benutzerschnittstelle
- Shell Objekte erzeugen
 - Hauptfenster des Programms
- Interaktionselemente erzeugen, die in den Fenstern (Shell) angezeigt werden
- Größe und Zustände der Interaktionselemente definieren
 - auch Zuhörer
- Fenster öffnen

Kochrezept: SWT GUI

- Fenster (Shell) öffnen
- Erzeugen einer Schleife, die Ereignisse verarbeitet
- bricht ab, wenn der Benutzer das Programm beendet
- z.B. durch Schließen des Fensters
- Fensterstruktur freigeben (dispose)

Beispiel: Hello World

```
import org.eclipse.swt.*;
import org.eclipse.swt.widgets.*;
public class SWTHelloWorld {
    public static void main(String[] args) {
        Display display = new Display();
        Shell shell = new Shell();
        Label label = new Label(shell,SWT.CENTER);
        label.setText("Hello World");
        label.setBounds(shell.getClientArea());
        shell.open();
        while(!shell.isDisposed()) {
            if (!display.readAndDispatch()) display.sleep();
        }
        display.dispose();
    }
}
```

Beispiel: Hello World

- SWT-Anwendungen starten:
 - swt.jar in CLASSPATH aufnehmen
 - Eclipse Laufzeitbibliotheken einbinden
 - z.B. Linux: LD_LIBRARY_PATH
 - Dokumentation von Eclipse



SWT Klassen: Display

- `org.eclipse.swt.widgets.Display`
- repräsentiert die Schnittstelle zwischen SWT und dem darunter liegenden Fenstersystem
 - verwaltet die Kommunikation zwischen der eigenen Ereigniswarteschlange und der Ereigniswarteschlange des Fenstersystems
- SWT Anwendung besitzen üblicherweise die im Beispiel gezeigte Struktur
 - Spezialfall: Mehrprozessanwendungen

SWT Klassen: Shell

- `org.eclipse.swt.widgets.Shell`
- Shell ist ein Fenster, das vom Fenstersystem verwaltet wird
 - Fenster oberster Ebene sind Display Objekten zugeordnet
 - Fenster unterer Ebenen sind anderen Fenstern (Shell Objekten) zugeordnet
 - z.B. Dialogfenster
- Interaktionselemente werden dem Shell Objekt zugeordnet
 - es gibt auch zusammengesetzte Interaktionselemente wie in Swing

SWT Komponentenbaum

- SWT GUI sind als Baum von Interaktionselementen zusammengesetzt
 - analog zu AWT und Swing
 - Unterschied: Wurzelklasse Display ohne grafische Darstellung

Lebenszyklus: Erzeugung

- Erzeugen eines Interaktionselements löst die Erzeugung im darunter liegenden Fenstersystem aus
 - Daten des Interaktionselements werden auf der Ebene des Fenstersystems gespeichert
- Beziehung zur Vaterkomponente wird im Konstruktor erzwungen
- Basisinitialisierung wird bei der Konstruktion definiert
 - Styles, über style bits definiert
 - gilt nicht für alle Eigenschaften
 - als ein int, logische Verknüpfung mit OR

Lebenszyklus: Vernichtung

- Interaktionselemente müssen explizit vernichtet werden
 - Freigabe von Systemressourcen
 - eigentlich untypisch für Java!
 - Methode dispose in jedem Interaktionselement implementiert
 - Vernichtung (dispose) wird rekursiv im Komponentenbaum ausgelöst

Interaktionselemente

- SWT definiert Interaktionselemente als controls bzw. widgets
 - controls sind konkrete Implementierung
 - widgets können abstrakt sein
- Beispiele für Controls (`org.eclipse.swt.widgets`)
 - Button, Combo, Label, List, Menu, ProgressBar, ScrollBar, Shell, TabFolder, Slider, Table, Text, ToolBar, Tree

Ereignisse

- Ereignisse funktionieren analog zu AWT und Swing
 - entsprechende Zuhörer- und Ereignisklassen
 - Methoden zur Verknüpfung von Zuhörern
 - `org.eclipse.swt.events`
- SWT kennt zusätzlich untypisierte Ereignisse
 - Ereignisse werden über eine `int`-Konstante definiert
 - untypisierte Ereignisse können mit Zuhörern verknüpft werden
 - sollten nicht verwendet werden
 - werden intern in SWT verwendet

Gruppierungen

- SWT definiert verschiedene Gruppierungen (layout)
 - Oberklasse `org.eclipse.swt.layout`
- FillLayout
 - einfachste Gruppierung
 - ordnet alle Interaktionselemente in einer Zeile oder einer Spalte
 - alle Interaktionselemente erhalten die gleiche Größe

Gruppierungen

- RowLayout
 - erweitert FillLayout um Umbruch der Interaktionselemente
 - Umbruchkriterien werden konfiguriert
- GridLayout
 - komplexe und flexible Gruppierung
 - ordnet Interaktionselemente in einem Gitter an
 - Flexibilität vergleichbar mit AWT GridBagLayout
- benutzerdefinierte Gruppierungen

Selbstkontrolle

- Warum sehen Swing GUI nicht so aus wie native Anwendungen?
- Gibt es optische Unterschiede zwischen AWT und SWT?
- Warum genügt bei SWT GUI nicht das Einbinden von swt.jar in den Klassenpfad?
- Warum gibt es eine unsichtbare Wurzel Display im Komponentenbaum?
- Warum benötigt SWT eine Methode dispose?
- Wozu dient die Endlosschleife nach Erzeugung einer SWT GUI?

Kontakt

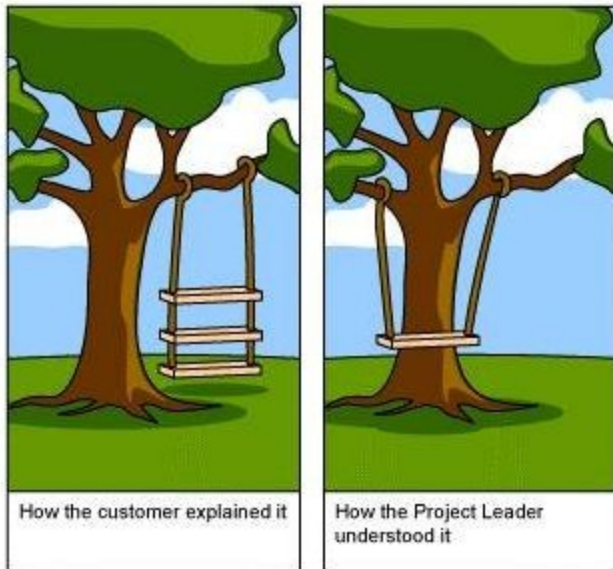
Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Software Engineering 1

Testen



Prof. Dr. Andreas Judt

Was ist Testen?

- Testen ist der Prozess, ein Programm mit der Absicht auszuführen, Fehler zu finden.
- Idee
 - Wurden alle beim Testen gefundenen Fehler behoben, so steigt die Wahrscheinlichkeit, dass im Betrieb keine Fehler mehr auftreten.

Reicht Testen aus?

- Beweist Testen die Fehlerfreiheit? NEIN
 - ein vollständiger Test umfasst alle möglichen Kombinationen von Dateneingaben.
 - vollständiger Test zeigt die Fehlerfreiheit für jeden theoretisch erreichbaren Systemzustand.
 - bei Zahlen noch überschaubar, bei anderen Datentypen praktisch nicht mehr
 - selbst bei wenigen Dateneingaben kann ein vollständiger Test bei heutiger Rechenleistung mehrere hundert Jahre dauern!

Ansätze zum Nachweis von Fehlerfreiheit

- Formaler Beweis
 - weist die mathematische Korrektheit nach
 - Verfahren aus der formalen Verifikation
 - vollständiger Test
 - heutige Verfahren funktionieren nur in trivialen Fällen oder sind aufgrund der Rechenleistung und –Zeit nicht praktisch einsetzbar.

Ansätze zum Nachweis von Fehlerfreiheit

- Modellprüfung (engl. model checking)
 - liefert Hinweise auf Fehler aus einem funktional reduzierten Modell
 - Tritt im Modell ein Fehler auf, deutet das auf einen möglichen Fehler im Ursprungssystem hin
 - tritt im Modell kein Fehler auf, dann ist auch das Ursprungssystem an der Stelle fehlerfrei
 - die Schwierigkeit liegt in der formalen Reduktion zum Modell
 - erfolgreicher Ansatz: Abbildung einer Programmiersprache auf eine Modellsprache; erzeugen eines Zustandsautomaten
 - Modellprüfung betrachtet einzelne Aspekte und erfordert sehr hohe Rechenleistungen: wird in der Praxis teilweise eingesetzt

Testvorgaben

- Testen setzt voraus, dass die erwarteten Ergebnisse bekannt sind.
 - Testen gegen eine Spezifikation
 - Regressionstest
 - Testen gegen bereits vorhandene Testergebnisse
 - z.B. nach Behebung von Fehlern
- Wichtig:
 - Testen ohne Vorbereitung und Dokumentation ist sinnlos!
 - Testen kann nicht alle Vorgaben abdecken, speziell nicht-funktionale Anforderungen

Behandlung von Fehlern

- gefundene Fehler beschreiben Symptome
 - z.B. Festplatte läuft voll
- Rückkopplung zum Programmierer ist schwierig!
 - Programmierer fühlen sich oft nicht zuständig
 - die Fehler erzeugenden Systemteile sind vom Tester nur schwer zu identifizieren
- Fazit:
 - die Rückkopplung an einen Entwickler oder ein Team ist auf der Basis eines gefundenen Fehlers sehr schwierig

falsche Testsystematik

- Ablauftest: Entwickler testet
 - Entwickler übersetzt und startet sein Programm
 - bei erkannten Fehlern oder Abweichungen werden die Defekte gesucht und behoben (engl. debugging)
 - Test ist beendet, wenn die Ergebnisse vernünftig aussehen
- Wegwerf-Test: jemand testet ohne System
 - jemand testet mit selbst vorgegebenen Eingabedaten
 - erkannte Fehler werden behoben
 - der Test endet, wenn der Tester meint, dass ausreichend getestet wurde

Systematischer Test

- Systematischer Test: Spezialisten testen
 - Test geplant, Testvorschrift wurde erstellt
 - Programm wird nach Testvorschrift ausgeführt
 - Ist und Soll werden verglichen
 - Fehlersuche und Fehlerbehebung erfolgen separat
 - nicht bestandene Tests werden wiederholt
 - Testergebnisse werden dokumentiert
 - Test endet, wenn das Testziel erreicht wurde

Systematischer Test

- Testarten
 - Komponententest = Modultest (engl. unit test)
 - Integrationstest (engl. integration test)
 - Systemtest (engl. system test)
 - Abnahmetest = Akzeptanztest (engl. acceptance test)
 - zeigt, dass die gestellten Anforderung erfüllt sind

Testablauf

- Planung
 - Teststrategie: was, wann, wie, wie lange
 - Einbetten des Testens in die Entwicklungsplanung
 - welche Dokumente sind zu erstellen
 - Termine und Kosten für Vorbereitung, Durchführung, Auswertung
 - Festlegung der Tester
- Vorbereitung
 - Auswahl der Testfälle
 - Bereitstellung der Testumgebung
 - Erstellung der Testvorschrift

Testablauf

- Durchführung
 - Testumgebung einrichten
 - Testfälle nach Testvorschrift ausführen
 - Testergebnisse dokumentieren
 - keine Änderungen vornehmen!
- Auswertung
 - Testergebnisse zusammenstellen
- Fehlerbehebung (gehört nicht zum Test)
 - gefundene Fehler (eigentlich Symptome) analysieren
 - Fehlerursache finden und Fehler beheben

Auswahl von Testfällen

- zentrale Aufgabe des Testers
- Anforderungen an Testfälle
 - repräsentativ
 - fehlersensitiv
 - redundanzarm
 - ökonomisch
- Ziel:
 - mit möglichst wenigen Testfällen möglichst viele Fehler finden

Klassifizierung

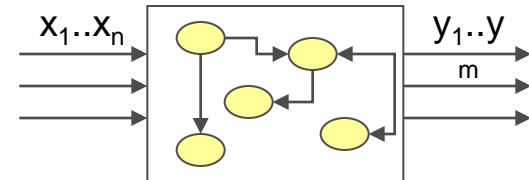
- funktionsorientierter Test
= black box test

- Auswahl der Testfälle basiert auf der Spezifikation
- Programmstruktur muss nicht bekannt sein



- strukturorientierter Test
= white box test
= glass box test

- Auswahl der Testfälle basiert auf der Programmstruktur
- Spezifikation liefert die erwarteten Testergebnisse



White Box: Eigenschaften

- Überprüfung von
 - Programmablauf
 - Testfälle werden so gewählt, dass das Programm systematisch durchlaufen wird.
 - Datenfluss im Programm
- In der Regel nur für Modultest und teilweise für Integrationstest geeignet

White Box: Testziele

- Anweisungsüberdeckung
 - Jede Anweisung des Programms wird mindestens einmal ausgeführt
- Zweigüberdeckung
 - jeder Programmzweig wird mindestens einmal durchlaufen
- Pfadüberdeckung
 - jeder Programmpfad wird mindestens einmal durchlaufen

Programmzweig vs. Programmpfad

- Bestimmung der Programmzweige
 - Betrachtung von Verzweigungen und Schleifen
 - IF und Schleifen: je 2 Zweige
 - SWITCH / CASE: so viele Zweige wie Fälle
- Bestimmung aller Programmpfade
 - alle Kombinationen aller Programmzweige bis zum maximalem Durchlauf aller Schleifen

White Box: Testgüte

- Die Testgüte hängt von der gewählten Überdeckung und dem erreichten Überdeckungsgrad ab.

$$\text{Überdeckungsgrad} = \frac{\# \text{ überdeckte Elemente}}{\# \text{ vorhandene Elemente}}$$

- Bewertung der Überdeckungen
 - Anweisungsüberdeckung schwach: fehlende Anweisungen werden nicht entdeckt
 - Pfadüberdeckung ist aufgrund der Komplexität nicht testbar.
 - Zweigüberdeckung ist praxistauglich jedoch werden falsch formulierte Bedingungen (legen die Verzweigung fest) nicht entdeckt.

Black Box: Testziele

- Funktionsüberdeckung
 - jede spezifizierte Funktion wird mindestens einmal aktiviert
- Ausgabeüberdeckung
 - jede spezifizierte Ausgabe wird mindestens einmal erzeugt
- Ausnahmeüberdeckung
 - jede spezifizierte Ausnahme bzw. Fehler wird mindestens einmal erzeugt
- Attributüberdeckung
 - alle geforderten Attribute getestet (soweit technisch möglich)
 - wichtiger Test: Erfüllung der Leistungsanforderungen
 - unter Normalbedingungen
 - unter möglichst ungünstigen Bedingungen (Belastungstest)

Black Box: Techniken zur effizienten Auswahl von Testfällen

- Äquivalenzklassenbildung
 - gleichartige Eingaben zusammenfassen (Äquivalenzrelation)
 - einen einzigen Repräsentanten testen
- Grenzwertanalyse
 - Fehler liegen oft an Wertebereichsgrenzen
 - Testfälle für Grenzen auswählen
- Ursache-Wirkung Graphen
 - systematisches Bestimmen von Eingabedaten, die ein gewünschtes Ergebnis liefern

Black Box: Techniken zur effizienten Auswahl von Testfällen

- statistisches Testen (engl. random testing)
 - Eingabedaten in großer Menge zufällig auswählen
 - Gesetz der großen Zahlen liefert Aussage über Zuverlässigkeit
 - automatische Prüfung der erwarteten Ergebnisse (mit sog. Orakel)
- Fehler erraten (engl. error guessing)
 - Auswahl der Testfälle erfolgt intuitiv: Qualität der Testergebnisse sehr stark von der Erfahrung des Testers abhängig
 - kann ergänzend zu anderen Techniken verwendet werden

Testplanung

- Qualitätsprüfung planen
 - was, wann nach welchen Strategien prüfen
- Qualität beim Testen
 - Testverfahren auswählen
 - Testdokumente spezifizieren
 - Zeitpunkte und Personen für die Tests festlegen

Testdokumentation

- Testvorschrift
 - wichtigstes Dokument für Vorbereitung und Durchführung
- Testvorschrift + Testergebnis = Testprotokoll
- Testzusammenfassung
 - Nachweis über Durchführung und Gesamtergebnis
- Normen für Testplanung und Dokumentation
 - IEEE 1987, 1988, 1998a, 1998b
 - sollten für kritische Entwicklungsprojekte verwendet werden

Vorschlag für eine Testvorschrift

- 1. Einleitung
- 1.1. Zweck
 - Art und Zweck des beschriebenen Tests
- 1.2. Testumfang
 - Teile des Systems, die hier getestet werden
- 1.3. Referenzierte Unterlagen
 - Verzeichnis der Dokumente, die hier verwendet wurden
- 2. Testumgebung
- 2.1. Überblick
 - Testgliederung, Testgüte, Annahmen, Hinweise
- 2.2. Testmittel
 - verwendete Software, Hardware, Betriebssystem, Werkzeuge

Vorschlag für eine Testvorschrift

- 2.3. Testdaten, Testdatenbank
 - Quelle für Testdaten oder Ort der bereitzustellenden Testdaten
- 2.4. Personalbedarf
 - Personen, die für die Durchführung benötigt werden
- 3. Annahmekriterien
 - Kriterien für
 - erfolgreichen Abschluss des Tests
 - Abbruch des Tests
 - Unterbrechung und Wiederaufnahme

Vorschlag für eine Testvorschrift

- 4. Testfälle
 - Testfall-Nummer
 - Eingabe
 - erwartetes Ergebnis
 - Bewertung des tatsächlichen Testergebnis

Programmierte Testvorschriften

- objektorientiert programmierte Testfälle
 - jeder Testfall ist ein Objekt
 - jedes Testfall-Objekt enthält die Testdaten und die erwarteten Ergebnisse
 - Testfall-Objekt ruft die zu testende Software auf
 - Testfall-Objekt bewertet das tatsächliche Ergebnis
- Einbettung in Laufzeitsystem für teilautomatisierte Tests
 - Beispiel für Java: JUnit
- geeignet für Komponenten- und Integrationstest
- nützlich als kontinuierlicher Regressionstest bei inkrementeller Entwicklung

Testen nicht funktionaler Anforderungen: eine Aufstellung

- Testen von Leistungsanforderungen
 - Leistungstest: Zeiten, Mengen, Raten, Intervalle
 - Lasttest: Verhalten am Grenzbereich der Systemlast
 - Stresstest: Verhalten bei Überlast
 - Ressourcenverbrauch
- Testen von Qualitätsanforderungen
 - in der Praxis kaum machbar
- in der Praxis testbar:
 - Zuverlässigkeit
 - Benutzbarkeit
 - Sicherheit

Testen von Benutzerschnittstellen

- Funktionalität
 - alle Funktionen über Dialoge zugänglich?
- Benutzbarkeit
 - Bedienbarkeit, Erlernbarkeit, Anpassbarkeit
- Nützlichkeit
 - Erleichtert die Benutzerschnittstelle die Arbeit wirklich?
- Dialogstruktur
 - Vollständigkeit, Konsistenz, Redundanz
- Antwortzeiten
 - Wartezeiten den Benutzers nach Interaktion

Kriterien für den Testabschluss

- wenn mit den Testvorschriften keine Fehler mehr gefunden werden
- wenn die Kosten für die Prüfung eine vorher festgelegte Grenze überschreiten
- wenn bei der Durchführung einer vorher festgelegten Teilmenge von Testfällen keine Fehler auftreten.
 - die Existenz von Fehlern bei den verbleibenden Testfällen ist dann sehr unwahrscheinlich
- wenn eine vorher festgelegte Fehlerdichte unterschritten wird

Fehlerdichte

- Anzahl der Fehler pro 1000 Codezeilen (KLOC) nach der Auslieferung
 - durchschnittlich: 5 / KLOC
 - sehr gut: 2 / KLOC
- Fazit:
 - eine sehr gute Software mit 1 Mio. Codezeilen enthält noch 2000 Fehler bei ihrer Auslieferung!
- Vergleich:
 - In Oberklasse-PKW fahren heute etwa 10 Mio. Codezeilen mit.

Selbstkontrolle

1. Was ist Testen?
2. Warum lässt sich die Fehlerfreiheit von Software in der Praxis nicht formal beweisen?
3. Erläutern Sie, wie man mit einem Testprotokoll eine fehlerhafte Programmstelle identifiziert.
4. Worin unterscheiden sich Blackbox und Whitebox Tests?
5. Warum ist es schwierig, nichtfunktionale Anforderungen zu testen?
6. Wie viele Fehler können Sie bei einem gut getesteten Betriebssystem erwarten?

Kontakt

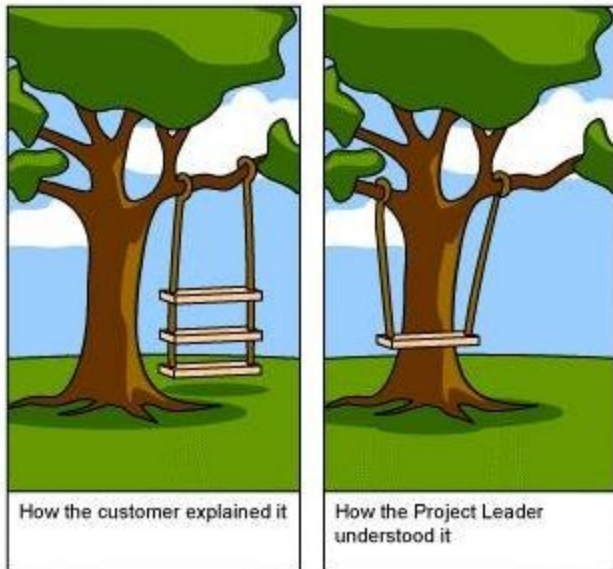
Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de

Software Engineering 1

Softwaremetriken



Prof. Dr. Andreas Judt

Warum Software messen?

- Kontrolle der Softwareentwicklung
 - während der Entwicklung
 - nach Projektabschluss
- typische Fragestellungen
 - wie weit ist die Entwicklung verglichen mit dem Zeitplan?
 - wie komplex ist die Software?
 - rechtfertigt die Komplexität den Zeitaufwand?
 - wurde die effizienteste Lösung implementiert?
- Was kann überhaupt gemessen werden?

Was ist eine Metrik?

- Metrik
 - Eine Metrik ist eine mathematische Funktion, die je zwei Elementen eines Raums einen nicht negativen skalaren Wert zuordnet, der als Abstand der beiden Elemente von einander aufgefasst werden kann.
 - in der Mathematik auch als Abstandsfunktion bezeichnet.
- Metrischer Raum
 - Ein Raum ist eine Menge, deren Elemente in geometrischer Interpretation als Punkte aufgefasst werden. Ein metrischer Raum ist ein Raum, auf dem eine Metrik definiert ist.

Mathematische Definition

- Sei X eine beliebige Menge. Eine Abbildung $d: X \times X \rightarrow \mathbb{R}$ heißt Metrik, wenn gilt
 - $d(x,x) = 0$ (identische Punkte haben den Abstand 0)
 - $d(x,y) = 0 \rightarrow x=y$ (nicht-identische Punkte haben nicht den Abstand 0)
 - $d(x,y) = d(y,x)$ (Symmetrie)
 - $d(x,y) \leq d(x,z) + d(y,z)$ (Dreiecksungleichung)

Was kann man messen?

- im Software-Prozess
 - Ressourcenaufwand (Mitarbeiter, Zeit, Kosten, ...)
 - Fehler
 - Kommunikationsaufwand
- im Produkt
 - Umfang (LOC, Wiederverwendung, Anzahl von Klassen und Methoden, ...)
 - Lesbarkeit
 - Entwurfsqualität (Modularität, Abhängigkeiten, ...)
 - Produktqualität (Testabdeckung, Testergebnisse, ...)

Gütekriterien für Software-Metriken

- Objektivität
 - keine subjektiven Einflüsse des Messenden
- Zuverlässigkeit
 - Wiederholung liefert die gleichen Ergebnisse
- Normierung
 - es gibt eine Skala für Messergebnisse und eine Vergleichbarkeitsskala
- Vergleichbarkeit
 - man muss das Maß mit anderen Maßen in Relation setzen können

Gütekriterien für Software-Metriken

- Ökonomie
 - Messung hat geringe Kosten
- Nützlichkeit
 - Messung erfüllt praktische Bedürfnisse
- Validität
 - Messergebnisse ermöglichen Rückschluss auf eine Kenngröße

Umfangs-Metriken

- LOC = lines of code
 - zählt die Zeilen der Quelldateien
- NCSS = non commented source statements
 - zählt die Quellzeilen ohne Kommentare und Leerzeilen
 - $SLOC \leq NCSS$
- Umfangsmetriken eignen sich zur Messung des geistigen Eigentums, abzüglich
 - generierter Code
 - kopierter Code
- Qualitätsaussagen auf Basis von Umfangsmetriken sind unrealistisch.

Halstead-Metriken, 1972

- messen die textuelle Komplexität
- Einteilung des Programms in Operanden und Operatoren
 - Operatoren kennzeichnen Aktionen und sind typischerweise Sprachelemente des Programms, z.B. int, (,), return, +;
 - Operanden kennzeichnen Daten und sind Bezeichner oder Literale des Programms, z.B. x, y, z, 0, 1

Halstead-Metriken

- Bestimmung von Parametern
 - $N1$ = Anzahl der Operatoren
 - $n1$ = Anzahl der verschiedenen Operatoren
 - $N2$ = Anzahl der Operanden
 - $n2$ = Anzahl der verschiedenen Operanden
 - $n = n1 + n2$ = Größe des Vokabulars
 - $N = N1 + N2$ = Länge der Implementierung

Halstead-Metriken

- Schwierigkeit, ein Programm zu verstehen (engl. difficulty)
 - $D = n^{1/2} * N^{2/n}$
 - wobei $N^{2/n}$ das durchschnittliche Auftreten von Operanden ist
- Umfang des Programms (engl. volume)
 - $V = N * \log_2(n)$
 - auch als „Größe des Algorithmus in Bits“ oder Halstead-Volumen bezeichnet
- Aufwand, das gesamte Programm zu verstehen (engl. effort)
 - $E = D * V$

Halstead-Metriken: Bewertung

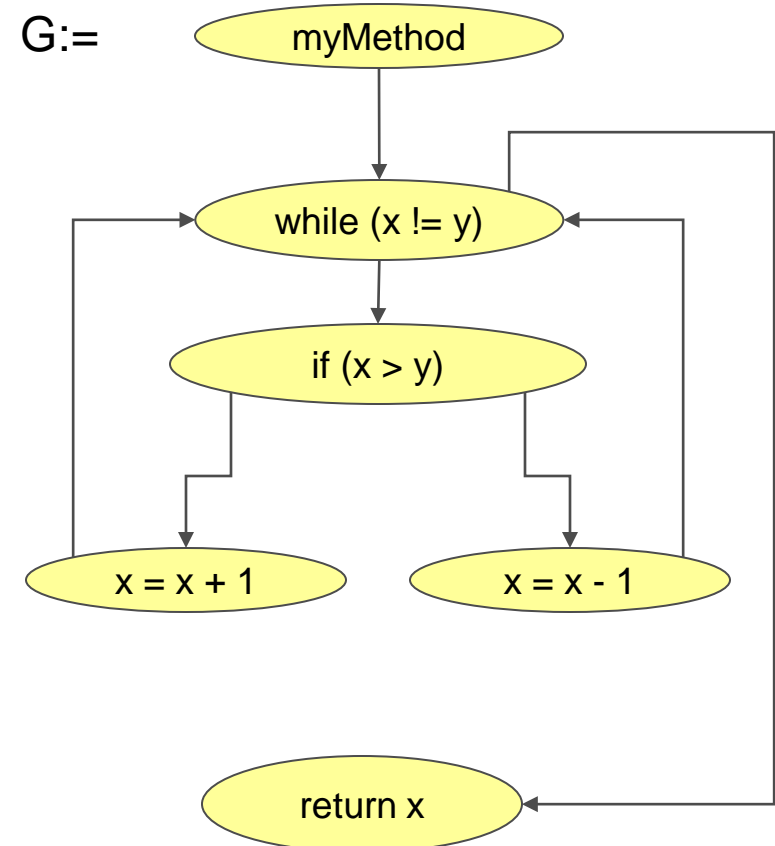
- Vorteile
 - einfach zu ermitteln und zu berechnen
 - universell einsetzbar
 - Halstead-Metriken wurden experimentell als gutes Maß für die Komplexität bestätigt
- Nachteile
 - es wird nur die textuelle Komplexität betrachtet.
 - moderne Sprachkonzepte bleiben unberücksichtigt
 - Anzahl und Aufteilung von Operatoren und Operanden ist sprachabhängig

McCabe Metriken, 1976

- Idee: strukturelle Komplexität bewerten
- Eingabe: Kontrollflussgraph G (engl. control flow graph, CFG)
- Ergebnis: zyklomatische Komplexität $V(G)$
- Bestimmung von Parametern
 - e = Zahl der Kanten (engl. edges)
 - n = Zahl der Knoten (engl. nodes)
 - p = Zahl der Zusammenhangskomponenten (engl. partial graph)
- Zyklomatische Komplexität:
 - $V(G) = e - n + 2p$

Kontrollflussgraph (CFG)

- Kanten
 $e = 7$
- Knoten
 $n = 6$
- Zusammenhangskomponenten
 $p = 1$
- Bedingungen
 $c = 2$
- $V(G) = e - n + 2p = 7 - 6 + 2 = 3$



McCabe Metriken: Bewertung

- Klassifizierung:
 - $V(G)$ in $[1..10]$:
einfaches Programm, geringes Risiko
 - $V(G)$ in $[11..20]$:
komplexeres Programm, moderates Risiko
 - $V(G)$ in $[21..50]$:
komplexes Programm, hohes Risiko
 - $V(G)$ in $[51..]$:
untestbares Programm, extrem hohes Risiko

McCabe Metriken: Bewertung

- Vorteile
 - einfach zu berechnen
 - zyklomatische Komplexität korreliert gut mit der Verständlichkeit einer Komponente
- Nachteil
 - nur Berücksichtigung des Kontrollflusses, kein Datenfluss
 - ungeeignet für objektorientierte Programme (zahlreiche triviale Aufrufe)

Hybride Metriken: Wartbarkeitsindex von Hewlett-Packard

- Idee: mehrere Metriken kombinieren
- Bestimmung von Parametern
 - V' = durchschnittliches Halstead-Volumen pro Modul
 - $V(G)'$ = durchschnittliche zyklomatische Komplexität pro Modul
 - L' = durchschnittliche LOC pro Modul
 - C' = durchschnittlicher Prozentsatz an Kommentarzeilen
- Wartbarkeitsindex MI:
 - $MI = 171 - 5,2 \cdot \ln(V') - 0,23 \cdot V(G)' - 16,2 \cdot \ln(L') + 50 \sin(\sqrt{2,4 \cdot C'})$
 - je größer MI, desto besser die Wartbarkeit
 - $MI < 30$: Restrukturierung

Metriken für objektorientierte Komponenten

- McCabe Metriken versagen bei objektorientierter Programmierung: Kontrollflusskomplexität oft gering ($V(G)=1$)
- Idee: Metriken müssen das Zusammenspiel der Klassen betrachten.
 - typisch: anhand des statischen Objektmodells
- Für dynamische Aspekte gibt es noch keine Metriken.
 - z.B. Zustandsautomaten, Sequenzdiagramme

objektorientierte Metriken: Auswahl

- DIT = depth of inheritance tree
 - Anzahl Oberklassen einer Klasse
 - DIT größer = Fehlerwahrscheinlichkeit größer
- NOC = number of children of a class
 - Anzahl direkter Unterklassen
 - NOC größer = Fehlerwahrscheinlichkeit geringer
- RFC = response of a class
 - Anzahl Funktionen, die nur direkt über Klassenmethoden aufgerufen werden
 - RFC größer = Fehlerwahrscheinlichkeit größer

objektorientierte Metriken: Auswahl

- WMC = weighted methods per class
 - Anzahl definierter Methoden
 - WMC größer = Fehlerwahrscheinlichkeit größer
- CRO = coupling between object classes
 - Anzahl Klassen auf deren Dienste die Klasse zugreift
 - CRO größer = Fehlerwahrscheinlichkeit größer
- Offene Frage Validität:

misst die Metrik etwas Sinnvolles?

Bewertung

- Das was wirklich interessant ist, kann nicht direkt gemessen werden.
- Metriken basieren oft nur auf Hypothesen.
- Metriken sind oft zu einfach.
 - management-tauglich?
- Metriken müssen in der Praxis validiert werden!

Standards für Software Metriken

- IEEE: Standard Dictionary of Measures to Produce Reliable Software, 1989
- IEEE: The Guide for the Use of Standard Dictionary of Measures to Produce Reliable Software
- IEEE Standard 1061: Software Quality Metrics
 - Funktion, die Software auf einen Zahlenwert abbildet
- US Air Force policy on software metrics

Übung: LOC Metrik

- Stellen Sie die Metrik für LOC (lines of code) auf.
- Implementieren Sie die Metrik in Java.
 - Eingabe 1: Name einer Programmdatei im Betriebssystem
 - Eingabe 2: Name einer Programmdatei im Betriebssystem
 - Ausgabe: Messwert (Wert d von LOC)
- Beweisen Sie, dass LOC eine Metrik ist.
- Bewerten Sie die LOC Metrik.

Übung: NCSS Metrik

- Stellen Sie die Metrik für NCSS (non commented source statements) auf.
- Implementieren Sie die Metrik in Java.
 - Eingabe 1: Name einer Programmdatei im Betriebssystem
 - Eingabe 2: Name einer Programmdatei im Betriebssystem
 - Ausgabe: Messwert (Wert d von NCSS)
- Beweisen Sie, dass NCSS eine Metrik ist.
- Bewerten Sie die NCSS Metrik.
- Vergleichen Sie die Metriken LOC und NCSS.

Zusammenfassung

- Software-Metriken definieren, wie eine Kenngröße in einem Software-Produkt oder einem Software-Entwicklungsprozess gemessen wird.
- Validität ist wesentlich:
 - misst die Metrik wirklich, was gemessen werden soll?
- Bei Metriken sind heute noch keine Standards etabliert.
- Messungen ersetzen nicht Test oder Verifikation.

Eine Definition...

- „Handbuch der Raumfahrttechnik“, Hanser Verlag, 3. Auflage 2008, S.364 (ausleihbar in der DHBW Bibliothek)
- „Zur Zeit gibt es ca. 400 Metriken, um die Eigenschaften von Software zu messen, aber sie sind alle sehr umstritten und kaum akzeptiert. Alles dies macht die Beurteilung von Software recht schwierig.“

=>>[im Original fortlaufender Text]

„Man weiß nur: Sie [Software] ist immateriell, abstrakt, komplex, überall, groß, teuer und dazu immer mit Fehlern behaftet.“

Selbstkontrolle

- Wie ist eine Metrik definiert?
- Warum eignen sich Umfangsmetriken kaum für die Bewertung der Softwarequalität?
- Erläutern Sie, wie Halstead die Schwierigkeit eines Programms definiert.
- Erläutern Sie, warum die Messung der zyklomatischen Komplexität eine gute Vergleichsmethode für ähnliche Programme ist.
- Begründen Sie, wie der Wartbarkeitsindex von HP entstanden ist.

Kontakt

Duale Hochschule BW Ravensburg
Campus Friedrichshafen

Prof. Dr. Andreas Judt
Informationstechnik

judt@dhbw-ravensburg.de