

Wie kann man Software besser entwickeln?

- Ingenieursmäßige Herangehensweise
  - Verwendung bekannter Prinzipien und Methoden
  - Systematische Vorgehensweise
- Verwendung von:
  - Abstraktion, Modelle, Notation, Simulation
  - Wiederverwendung: Muster, Komponenten, Framework
- Organisation
  - Arbeitsteilung, Integration, Planung
- Verwendung von Werkzeugen
  - IDE (Integrated Development Environment)
  - Versionierung, Bugtracker, Modellierungswerkzeug

## Modellierungskonzepte

Modell: ist eine Abstraktion eines Systems mit der Zielsetzung, das Nachdenken über ein System zu vereinfachen, indem irrelevante Details ausgelassen werden

- erstellen einer Abstraktion
- abbilden signifikanter Eigenschaften
- Deskriptiv/präskriptiv (real oder geplant)
- Sichtweise auf ein System (Struktur, Verhalten, Zustand,...)
- setzt Verstehen voraus
- ist nicht automatisierbar

## Objektorientierung

- Grundprinzip: Teile und Herrsche
- ein System besteht aus vielen Objekten, mit:
  - definiertes Verhalten
    - \* Menge genau definierter Operationen
    - \* Operation beim Empfang einer Nachricht ausgeführt
  - inneren Zustand
    - \* Zustand des Objekts ist Privatsache
    - \* Resultat hängt vom aktuellen Zustand ab
  - eindeutige Identität
    - \* Identität ist unabhängig von anderen Eigenschaften
    - \* verschiedene Objekte mit identischem Verhalten im gleichen System möglich
- Klasse
  - Gleichartige Objekte mit ggf. verschiedenen Zuständen
  - Verhaltensschema - Operationen
  - Innere Struktur - Attribute
- Vorteile
  - Zuständigkeitsbereiche** Daten, Operationen und Zustand: lokal und gekapselt
  - Klare Schnittstellen** Definiertes Objektverhalten, Nachrichten
  - Hierarchie** Vererbung und Polymorphie (Spezialisierung), Klassenschachtelung
  - Baukastenprinzip** Benutzung vorgefertigter Klassenbibliotheken, Anpassung durch Spezialisierung

## Unified Modeling Language

- Grafisches Beschreibungsmittel für Aspekte des Softwareentwurfs diskreter Systeme
- Kombination von Struktur-, Verhaltens-, Interaktions-, und Verteilungsmodellen
- Für Analyse, Entwurf, Implementierung und Test einsetzbar
- Gute Werkzeugunterstützung für Editieren, Versionierung, Codegenerierung
- Erweiterbarkeit der UML mit Stereotypen und Tags

Nachteile UML

- UML ist in vielen Facetten nicht präzise festgelegt
- Werkzeuge für Transformation, Analyse etc. fehlen noch
- UML ist keine „kleine Sprache“: Lernaufwand notwendig
- Komponenten sind nicht adäquat darstellbar

## Klassische Modelle

- Funktionen
  - Funktionsbaum
  - Blockschaltbild
- Daten
  - Data Dictionary (Daten mit Strukturinformationen)
  - Entity Relationship Diagram
- Systemumgebung (Datenflussdiagramm)
- Algorithmen
  - Entscheidungstabelle
  - Pseudocode
  - Programmablaufplan
  - Struktogramm
- Dynamisches Verhalten (diskrete Zustände und atomare zustandsübergänge)
  - Zustandsautomat
  - Flow-Chart
  - Ereignisgesteuerte Prozesskette (EPK)
  - Petri-Netz (ggf. mit Zeitmodell)
- Objektorientierte Modelle
  - Klassendiagramme
  - UML
- Struktur-Diagramme
  - Klassen-, Objekt-, Komponenten-, Kompositions-Struktur-,
  - Paket- und Verteilungsdiagramm
- Verhaltens-Diagramme
  - Use-Case-, Aktivitäts- und Zustandsdiagramms
  - Interaktionsdiagramme: Sequenz-, Kommunikations-, Timing- und Interaktionsübersichts-Diagramm

## Use-Case-Diagramm

- Beschreiben Systemfunktion aus Benutzersicht (Was, nicht Wie)
- Erste Anforderungsspezifikation (requirements)
- Planbare Einheiten als Inkremente für die Entwicklung
- Keine Modellierung eines Ablaufs!
- Erstellen von Testfällen (test case generation)

## Klassendiagramm

- Modellierung der Struktur (Aufbau) eines Systems
- Modellierung von statischen Aspekten und Struktur von Daten
- Klasse im Mittelpunkt (Attribute, Operationen, Beziehung)

## Objektdiagramm

- Struktur des Systems zur Laufzeit zu einem Zeitpunkt
- detaillierte Sicht auf einen Aspekt
- Keine Exemplare von Operationen
- Kein Verlauf der Wertebelegung über die Zeit

## Paketdiagramm

- Gliederung des Systems in Teile (Pakete)
- Zuordnung von Elementen zu einem Paket
- Bildung von Hierarchien (Enthält-Beziehung)
- Abhängigkeiten zwischen den Paketen
- Zum Grobentwurf von Systemen

## Komponentendiagramm

- Strukturierung des Systems durch Komponenten
- Komponente: Modulare, austauschbare Einheit (Substitution)
- Modellierung der Abhängigkeiten zwischen Komponenten
- Modellierung der inneren Struktur von Komponenten
- Definition von Schnittstellen

## Kompositionsstrukturdiagramm

- Teile-Ganzes-Strukturen → Kompositionsstruktur
- Strukturell statische Kompositionsstrukturen
- Strukturell dynamische Kompositionsstrukturen
- Spezialisierte Kompositionsbeziehung → erweiterte Semantik

## Aktivitätsdiagramm

- Modellierung von Kontrollflüssen, Datenflüssen, Parallelem Verhalten
- Geschäftsprozessmodellierung möglich
- Abstrakte und detaillierte Verhaltensbeschreibung möglich
- Grundlage zur Codegenerierung
- Zur Verfeinerung von
  - Use-Cases
  - Operationen / Interaktionen

## Interaktionsdiagramme

- Modellierung von Kommunikation & Operationen
- Sehr detaillierte Diagramme
- Meist nicht zur vollständigen Beschreibung eines Systems
- Betrachtung eines wichtigen Teilaspekts
- Grundlage zur Codegenerierung
- Ereignis des Sendens bzw. Empfangens von Nachrichten
- Typen:
  - Operationsaufruf (synchron / asynchron)
  - Antwort Nachricht
  - Signal (asynchron), Create-/ Delete Message

## Sequenzdiagramm

- Genaue zeitliche Abfolge von Nachrichten
- Umfangreichstes Interaktionsdiagramm
- Kontrollelemente möglich (Schleifen, Verzweigungen)
- Weitere Elemente des Sequenzdiagramms
  - Nachrichten ohne Sender z.B. am Beginn einer Interaktion
  - Verlorene Nachrichten (ohne Empfänger) z. B. am Ende einer Interaktion
  - Erzeugen von Lebenslinien
  - Zerstören von Lebenslinien

## Kommunikationsdiagramm

- Kommunikationsbeziehungen der Kommunikationspartner stehen im Vordergrund
- Welche Komponenten arbeiten wie zusammen, um eine Funktion zu erfüllen

## Timing-Diagramm

- Genaue zeitliche Darstellung von Zustandsübergängen
- Kommunikation abhängiger Zustandsautomaten
- Modellierung einzelner Interaktion
- Zeitlicher Verlauf senkrecht
- Kommunikationspartner waagerecht (unsortiert)
- Ereignisspezifikation
- Trace: Folge von Sende- und Empfangsereignissen

## Zustandsdiagramm

- Modellierung des (vollständigen) Verhaltens
  - Zustände von Klassen / Objekten / Komponenten
  - Übergänge zwischen den Zuständen
  - Ereignisse, die Zustandswechsel auslösen
- Modellierung von endlichen Automaten (Zustandsmaschinen)
  - Deterministische
  - Nichtdeterministische
- Modellierung von verteilten Systemen / parallelem Verhalten
- Grundlage zur Codegenerierung

# Analyse

Funktionale Anforderungen - Was soll es tun?

- Was leistet das System
- Welche Funktionen bietet es
- Wie interagiert es mit der Umgebung
- Anforderungen an: Verhalten, Struktur

Nichtfunktionale Anforderungen - Wie?

- Quality of Service/Qualitätsanforderungen
- Hängen oft von Verhalten ab: komplex und nicht direkt sichtbar
- Nicht direkt „by construction“ zu realisieren
- Untersuchung der Projektrisiken bereits in der Analysephase
- Modellbasierter Entwurf des Systems und seiner Umwelt
- Arten - FURPS (ISO 9126):

**Functionality** (Funktionalität) Angemessen, Genauigkeit, Sicherheit  
**Usability** (Benutzbarkeit) Verständlichkeit, Erlernbarkeit, Bedienbarkeit, Attraktivität  
**Reliability** (Zuverlässigkeit) Reife (Fehler-Anzahl), Fehlertoleranz, Wiederherstellbarkeit  
**Performance** (Effizienz/Portability) Zeitverhalten, Verbrauchsverhalten, Wirtschaftlichkeit, Koexistenz  
**Supportability** (Änderbarkeit/Wartbarkeit) Analysierbarkeit, Änder- und Erweiterbarkeit, Stabilität, Testbarkeit

- weitere
  - Konformität zu Konventionen und Bestimmungen
  - Interoperabilität zu anderen Systemen
  - Implementierungsanforderungen
  - Schnittstellenanforderungen
  - Skalierbarkeit (Änderungen des Problemumfangs)
  - Betriebliche und rechtliche Rahmenbedingungen
  - Liefer- und Verpackungsanforderungen

Randbedingungen

- Werden nicht umgesetzt
- Schränken Lösungsraum ein
- Kosten, Durchlaufzeit, Technische Randbedingungen

Geforderte (Meta-)Eigenschaften

**Vollständig** alle Szenarien sind beschrieben  
**Konsistent** keine Widersprüche  
**Eindeutig** nur eine Interpretation möglich  
**Korrekt** genaue und richtige Darstellung  
**Realistisch** unter geg. Einschränkungen implementierbar  
**Überprüfbar** durch Tests am Endprodukt nachweisbar  
**Rückverfolgbar** Auswirkungen bis Implementierung nachvollziehbar  
**Klassifizierbar** Risiko, Priorität, Dringlichkeit, Nutzen ...  
**Validierung** mit dem Kunden

## Ermitteln von Anforderungen

- Ausgangspunkt
  - Projektidee, schriftliche Skizze
  - Kurz und knapp
  - Stichpunkte der wichtigsten Funktionen
  - Lastenheft (falls schon existiert)
- Beteiligte Rollen
  - Endbenutzer** Aufnahme Ist-Zustand, Domänenwissen, Anforderungen
  - Kunde** Definiert Ziel des Systems, Vertragsverhandlung
  - Konfigurationsmanager** Revisionsgeschichte der Dokumente, Nachvollziehbarkeit
  - Architekt** Integration von Anwendungsfall- und Objektmodellen
  - Analytiker** Modelliert das System und erstellt Anwendungsfälle
  - Redakteur**
  - Prüfer**

- Wie ermittelt man Anforderungen?
  - Kommunikation mit Kunden
  - Geschäftsprozess: fachlicher Ablauf, der Wert oder Kosten verursacht
    - Akteur** Benutzer, Schnittstelle nach außen
    - Szenario** Interaktion mit System als Ablauf
    - Anwendungsfall** Automatisierter Arbeitsschritt, vom System ausgeführt
    - Interviews mit Fachanwendern** Mitschrift, später strukturierter Text und Tabelle
    - Strukturierte Spezifikation** Vorlagen, Formulare, Reduzierung sprachlicher Mehrdeutigkeiten
  - Anwendungsfalldiagramm (Use-Case-Diagramm)
    - \* Arbeitsschritt eines Geschäftsprozesses
    - \* Anforderungen an das System modellieren
    - \* Systemgrenzen / Systemkontext festlegen
    - \* Systembeteiligte modellieren
    - \* Planbare Einheiten als Schritte für die Entwicklung
    - \* Verwendung bereits ab Projektbeginn
  - Umgang mit Szenarien und Anwendungsfällen
    - \* Systemgrenze definieren
    - \* Beschreibungen verfeinern
    - \* Änderungen mit Kunden abstimmen
    - \* Prototypen nur zur visuellen Unterstützung
- Typische Probleme
  - Kommunikations- und Verständnisprobleme
  - Viele verschiedene Beteiligte
  - Widersprüchliche Anforderungen, verschiedene Interessen
  - Nicht-technische organisatorische, historische oder rechtliche Rahmenbedingungen
  - Anforderungen ändern sich während der Entwicklung
- Tätigkeiten der Anforderungsanalyse
  - Anforderungen strukturieren
    - \* Unterteilung (Funktional, Nichtfunktional)
    - \* Hierarchische Zerlegung (Unterteilen, Verfeinern)
    - \* Ordnung festlegen, eindeutig Nummerieren
    - \* Beziehungen festhalten
    - \* Verwendung von Werkzeugen
  - Eigenschaften der Anforderungen bestimmen
  - Anforderungen priorisieren
    - \* Muss-, Kann-, Optional, Nicht
    - \* Ad-hoc: Stakeholder priorisiert Anforderungen
    - \* Priorisierungsmatrix / Kosten-Wert-Analyse
    - \* Priorität berechnen  $\text{Prioritäten} = \frac{\text{Nutzen} - \text{Nachteil}}{\text{Kosten} + \text{Risiko}}$
  - Kano-Klassifikation
    - \* Basiseigenschaften: Werden vorausgesetzt
    - \* Leistungseigenschaften: Sonderwünsche
    - \* Begeisterungseigenschaften: Wird nicht erwartet
    - \* Abfragen per Fragenkatalog
  - Reihenfolge festlegen
  - Anforderungen dokumentieren
  - Anforderungen modellieren
  - Anforderungen auf inhaltliche Qualität prüfen
  - Auf Übereinstimmung mit den Zielen prüfen

## Objektorientierte Analyse und Modellierung

- Systemmodell erstellen, funktionale Spezifikation (Was, nicht wie)
- Analysemodell
  - Korrekt, vollständig, konsistent und nachprüfbar
  - Strukturiert und Verhalten
- Eingangsdokumente (Lastenheft, Anforderungsspezifikation)
- Typische Ergebnisse
  - Funktionales Modell (Anwendungsfälle)
  - Dynamisches Modell - Systemverhalten (Zustandsdiagramme)
  - Vor- und Nachbedingungen von Systemoperationen

- Pflichtenheft
- Verdeutlicht iterativen Ablauf
- Unterteilung des Analysemodells
  - Funktionales Modell (Anwendungsfälle)
  - Objektmodell (Klassen und Objektdiagramme)
  - Dynamisches Modell (Zustands- und Sequenzdiagramme)
  - Unterscheidung der Objekttypen
- Objektarten im Systemmodell
  - Entitätsobjekte** vom System verwaltete Informationen
  - Grenzobjekte** Interaktion zwischen System und Akteuren
  - Steuerungsobjekte** Durchführung der Anwendungsfälle
- Identifizierung von Entitätsobjekten
  - Begriffe, die klargestellt werden müssen
  - Wiederkehrende Substantive in Anwendungsfällen
  - Reale Objekte, die das System kennen muss
  - Reale Prozesse, die das System verfolgen muss
  - Anwendungsfälle
  - Datenquellen und -senken
- Identifizierung von Grenzobjekten
  - Elemente der Benutzungsschnittstelle
  - Formulare für Eingaben
  - Nachrichten, Rückmeldungen
  - Endgeräte
- Identifizierung von Steuerungsobjekten
- Abläufe der Anwendungsfälle modellieren
- Use Case durch Interaktion verfeinern
  - datengetriebener Ablauf mit Verzweigungen** Aktivitätsdiagramm
  - Interaktion zwischen den Objekten wichtig** Kommunikations-, Aktivitäts-, Sequenzdiagramm
  - zeitliche Abfolge steht im Mittelpunkt** Sequenzdiagramm
  - Zustandswechsel / zeitliche Abfolge von Zuständen** Zustands-/Timing-Diagramm
  - komplexe Abläufe mit Verzweigungen und Parallelitäten** Interaktionsübersichtsdiagramm
  - komplexer strukturierter Ablauf** Kollaboration aus Kompositionsstrukturdiagramm
- Verfeinerung der Aktionen durch Aktivitäten
- Aktion durch Interaktionen verfeinern (Detaillierte Diagramme)
- Verfeinerung der Aktionen durch StateChart
- Objekte zusammenstellen und klassifizieren
  - Toolunterstützung (Möglichkeiten stark toolabhängig)
  - Objekte Ergebnis der Verhaltensmodellierung
  - Ergebnis Verhaltensdiagramm: Operationen der Klassen
  - Klassen generalisieren / spezialisieren → Klassenhierarchie
- Übergang zum Entwurf: Klassenstruktur festlegen
- Spezifikation von Benutzungsschnittstellen
  - Skizzieren, Prototyp generieren, Spezialwerkzeuge
  - Klassen und Operationen in Funktionen
  - Gestaltung MMI, style guides, Standards

## Dokumentation von Anforderungen

- Lastenheft
  - Forderungen eines Auftraggebers (AG) an Lieferungen und Leistungen eines Auftragnehmers (AN)
  - Muss-Kriterien, Kann-Kriterien, Abgrenzungskriterien
- Pflichtenheft: Entwurf aus AN-Sicht, Umsetzung des Lastenhefts
- Anforderungsspezifikation
  - Zielsetzung**
  - Allgemeine Beschreibung** Umgebung, Funktion, Benutzer
  - Spezifische funktionale Anforderungen** quantitativ + identifizierbar
  - Spezifische nicht-funktionale Anforderungen** Entwicklungs- und Produkt-Standards
  - Qualitäts-Zielbestimmung**
  - Zu erwartende Evolution** des Systems, Versionen
  - Abkürzungsverzeichnis** Glossar, Index, Referenzen

# Grobentwurf Einführung

## Systementwurf - Aufgabe

- Sicht des geplanten Systems von innen (Entwickler)
- Wie sollen vereinbartes Verhalten und Funktionen (Analysemodell) intern realisiert werden?
- Von Spezifikation von Anforderungen und Funktionen → Vorbereitung der Implementierung
- Formal: Transformation des Analysemodells in ein Systementwurfsmodell
- System(grob)entwurf, Feinentwurf/Objektentwurf

## Teile und herrsche

- Grobentwurf
  - Entwurfsziele identifizieren
  - Grobe Systemstruktur festlegen (Architektur)
  - Zerlegung in Subsysteme, Spezifikation
    - \* Schichten, Pakete, Komponenten
  - Bewerten der Zerlegung anhand der Entwurfsziele
  - Schnittstellen festlegen
- Feinentwurf
  - Subsysteme im Detail entwerfen
    - \* Strukturierung der Komponenten
    - \* Klassen, Objekte, Funktionen, Datenstrukturen
    - \* Verhalten, Algorithmen - Teillösungen

## Systemzerlegung

### Vorgehen

- Zerlegung eines Systems in Subsysteme
- Betrachtung der Lösungsdomäne!
- Subsysteme weiter zerlegen bis Komplexität ausreichend klein ist z.B. für Arbeitspakete

### Was macht ein Subsystem aus?

- Schnittstellen, Funktionen, „Verantwortung“
- Was bietet es an?
- Was benutzt es?
- Was tut es intern?

### Operation

- Name und Parameter
- Funktion, Prozedur, Methode, Eintrittspunkt ...

Dienst: Satz von Operationen, die bereitgestellt werden  
Abhängigkeiten von Subsystemen

- Subsysteme untereinander: Kopplung (coupling)
- Maß für die Abhängigkeit von Subsystemen

### Möglichst lose Kopplung

- Änderungen in einem beteiligten Subsystem haben geringe Auswirkungen (Stabilität)
- Erleichtert Wartbarkeit und Arbeitsteilung

### Mittel zur Verringerung der Kopplung

- Zusätzliche Unterteilung in Subsysteme
- Aber: dann größere Komplexität!

### Abhängigkeiten von Subsystemen

- Elemente eines Subsystems: Kohäsion (cohesion)
- Maß für Zusammengehörigkeit der Elemente
- Möglichst hohe Kohäsion
  - Enge Beziehung oder ähnliche Aufgaben der Elemente

- Erleichtert Verständnis, Wartung und Anpassung
- Mittel zum Erreichen hoher Kohäsion
  - Datenkapselung, Objektorientierung
  - Benutzung geeigneter Patterns (Kapitel 5)

### Metriken für modulare Entwürfe

- Fan-in / fan-out-Metrik [S.Henry, D. Kafura 1981]:
  - Fan-in: Anzahl der Stellen, wo Kontrollfluss auf das betrachtete Modul M übergeht (Aufrufe von Funktionen / Prozeduren in M) + Anzahl globaler Variablen, die in M zugänglich sind
  - Fan-out: Anzahl von Stellen, an denen M andere Module aufruft + Anzahl der globalen Variablen, die von M verändert werden
- Heuristik Kopplung / Kohäsion
  - Hoher Fan-out bedeutet hohe Kopplung, minimieren
  - Hoher Fan-in kann auf geringe Kohäsion von M hindeuten

Komplexität beherrschen: ”Wenn Du es nicht in fünf Minuten erklären kannst, hast Du es entweder selbst nicht verstanden oder es funktioniert nicht.”[Rechtin, Maier: The Art of Systems Architecting 2000]  
Vorgehen: Heuristiken und Erfahrungen

- „Erfahrung ist die härteste Lehrerin. Sie gibt Dir zuerst den Test und anschließend den Unterricht.“ [Ruth 1993]
- „Ein Architekt der zu Beginn seiner Arbeit vollständige und konsistente Anforderungen benötigt, mag ein brillanter Entwickler sein - aber er ist kein Architekt“ [Rechtin 2000]
- „Das Leben von Software-Architekten besteht aus einer langen und schnellen Abfolge suboptimaler Entwurfs-entscheidungen, die teilweise im Dunkeln getroffen werden.“ [Kruchten2001]

### Wie organisiert man Subsysteme?

- Innerhalb einer Verfeinerungsstufe: fachlich orientierte Zerlegung
- Mehrfache Zerlegung: Hierarchie-Graph der Verfeinerung

### Schicht

- Gruppe von Subsystemen in der Zerlegungshierarchie
- Verwandte Dienste
- Ähnlicher Abstraktionsgrad
- Abhängigkeit nur von darunter liegenden!
- Geschlossene Schichtenarchitektur
  - Beispiel: OSI-Modell für Kommunikationssysteme
- Offene Schichtenarchitektur
  - Beispiel: Java Swing auf X11-Plattform

### Prinzipien des OO-Entwurfs

- So-einfach-wie-möglich-Prinzip (KISS)
- Fehler berücksichtigen (Strukturierung, Kapselung, Modularisierung, Wiederverwendung)
- Entwerfen nach Verantwortlichkeiten
- Hohe Kohäsion / Geringe Kopplung
- Zyklische Abhängigkeiten vermeiden
- Auf Schnittstellen konzentrieren
  - Abhängigkeiten nur von Schnittstellen
  - Abtrennung von Schnittstellen (eher viele kleine als eine große)
  - Umkehr der Abhängigkeiten (dependency inversion-Prinzip)
- Offen / Geschlossen Prinzip

### Zyklische Abhängigkeiten vermeiden

- Änderungen wirken sich auf beide Komponenten aus
- Probleme beim Löschen und Initialisieren

- Auflösen durch
  - Gemeinsame Klassen in separates Paket
  - Gemeinsame Schnittstellen definieren

### Symptome schlechten Designs

- Starrheit
  - Einfache Änderungen schwierig realisierbar
  - Einfache Änderungen führen zur Modifikation einer Vielzahl von Komponenten
- Zerbrechlichkeit
  - Änderungen an einer Stelle führen zu Fehlern an völlig anderer Stelle
- Schlechte Wiederverwendbarkeit
  - Komponenten können aufgrund spezieller Anhängigkeiten kaum wiederverwendet werden

### Wann ist ein Entwurf „gut“?

- Korrekt
  - Erfüllung der Anforderungen
  - Wiedergabe aller Funktionen des Systemmodells
  - Sicherstellung der nichtfunktionalen Anforderungen
- Verständlich und präzise, gut dokumentiert
- Anpassbar
- Hohe Kohäsion innerhalb der Komponenten
- Schwache Kopplung zwischen den Komponenten
- Wiederverwendung
- Kriterien gelten auf allen Ebenen des Entwurfs! (Architektur, Subsysteme, Komponenten)

## Architekturmodelle

- Modellierung mit UML
  - Bisher: logische Sicht
  - Technisch: Organisation in Paketen, Namensraum, Import
- Paketdiagramm
  - Gliederung (Strukturierung) des Systems in Teile
  - Zuordnung von Elementen zu einem Paket
  - Hierarchien und Abhängigkeiten zwischen den Paketen
  - Anwendung: Definition von Schichten
- Enthält-Beziehung
  - Definiert, in welchem Paket ein Element enthalten ist
  - Ermöglicht qualifizierten Zugriff auf enthaltene Elemente
  - Löschen des Pakets bewirkt Löschen beinhalteter Elemente
  - Definition von Sichtbarkeit / Zugriffsrechte
    - \* Auswirkung auf weitere Enthält-Beziehung
    - \* '+' - public (default)
    - \* '-' - private
- Paket- / Element-Import
  - Unqualifizierter Zugriff auf Elemente eines anderen Namensraums (Paketes)
- Komponentendiagramm
  - Komponente - modulare, austauschbare Einheit
  - Strukturierung des Systems durch Komponenten
  - Modellierung der
    - \* Abhängigkeiten zwischen Komponenten
    - \* inneren Struktur von Komponenten
  - Definition von Schnittstellen
  - Verwendung von Elementen aus Klassen- und Objektdiagramm
  - Stärkere dynamische Sicht → kein Verhalten
  - Komponente `component`
    - \* Kapselt Funktionalitäten (Physisch gruppierte Klassen)
    - \* „Spezialisierte“ Klasse (Vererbung, Exemplare möglich)
    - \* Stellt Funktionalitäten über Schnittstellen bereit

- \* Definiert benötigte Schnittstellen
- \* Enthält Klassen oder weitere Komponenten
- \* Modulares Element: Substitution (Austauschbarkeit) steht im Vordergrund
- Black-Box-Darstellung
  - \* Zur Verfügung gestellte Funktionalität ‘`provided interfaces`’
  - \* Benötigte Funktionalität ‘`required interfaces`’,
- White-Box-Darstellung
  - \* Interner Aufbau der Komponente ‘`realization`’
  - \* Artefakte ‘`artifacts`’, Realisierende physische Einheit (z.B.: .dll)

#### Schnittstellen / Interfaces

- Definition Diagrammunabhängig
  - Meist Klassendiagramm
- Ähnlich Semantik einer Klasse
  - Nur public-Attribute und Operationen
- Definiert Verpflichtung zur Implementierung von
  - Operationen
  - Merkmale → Attribute dürfen definiert werden
  - Verpflichtungen (z.B.: Vor- / Nachbedingungen)
- Meist abstrakte Klassen mit abstrakten Operationen
- Abstrakt - muss überschrieben werden
- Notation
  - Stereotyp: `interface`
  - Meist kursiv geschrieben, da abstrakte Klasse

#### Schnittstellenrealisierung, Implementierungsbeziehung

- Schnittstellen werden realisiert, nicht instanziiert
- Schnittstellenkonform
  - Klasse realisiert alle Attribute und Operationen
- Schnittstelle kann von anderen Schnittstellen erben
- Keine Schnittstellenrealisierung zwischen zwei Interface-Klassen → Generalisierung verwenden
- Darstellung
  - Gestrichelte Linie mit nicht gefülltem Dreieck an der Seite der Superklasse
  - Alternativ: Lollipop-Darstellung

### Softwarearchitekturmuster

- Wiederverwendung auf sehr hoher Abstraktionsstufe
- Falls geplante Anwendung passt, anwenden!

#### Schichten-Architektur (layers)

- Problem
  - Komplexität: Strukturierung des Systems, unterschiedliche Abstraktionsebenen
  - Änderungen sollen möglichst lokal bleiben
  - Teilsysteme sollen austauschbar, wiederverwendbar und getrennt entwickelbar sein
  - Schnittstellen sollen stabil sein
- Lösung
  - Zuordnung von Subsystemen zu horizontalen Schichten gleicher Abstraktionsebene
  - Komponenten einer Schicht bieten Dienste der darüber liegenden Schicht an

#### Client-Server (Klient/Anbieter)

- Client (front-end)
  - Benutzungsschnittstelle
  - Einbindung in Geschäftsprozesse

- Entkoppelt von Netztechnologie und Datenhaltung
- Server (back-end)
  - Datenhaltung, evtl. Fachlogik
- Genauer: Two-tier client/server architecture
- Asynchroner Kontrollfluss
- Aufteilung Funktionen Client / Server
  - Mehr Funktionen im Server:
    - \* zentrale Verwaltung, Wartungsaufwand geringer, Portabilität, einfache Client-Hardware (Net PC)
    - \* „Thin Client“ - nur GUI
  - Mehr Funktionen im Client: Flaschenhals Server wird entlastet, individuellere Client-Funktionen
    - \* „Fat Client“ - Teil der Anwendung im Client
  - Entscheidet mit über Umsetzung (Java Script, ...)

Three-Tier / Four-Tier Architecture: Client/Server mit weiterer Aufteilung ähnlich Repository  
Bewertung Client-Server

- Vorteile
  - Leicht verständlich
  - Änderungen bleiben lokal
  - Geringere Kopplung zwischen den Schichten
  - Schichten austauschbar und wiederverwendbar
  - Getrennte Entwicklung der Schichten möglich
  - Vorhandene / stabilere Schnittstellen
- Nachteile
  - Geringere Performance
  - Zusätzlicher Verwaltungs- oder Datenoverhead
  - Manche Änderungen führen zu Änderungen in allen Schichten (z.B. neues Datenfeld)

#### Pipes and Filters

- Datenstrom- oder Kontrollflussorientiertes System
- Lose verbundene Berechnungskomponenten
- Kombination der Berechnungskomponenten nur vom Typ der Ein- und Ausgabedaten abhängig
- Leicht erweiterbar System gewünscht
- Parallele Verarbeitung vorteilhaft
- Verwendung von globalen Steuerungskontrollstrukturen (Parallelisierung, Verzweigung, Schleifen) gewünscht
- Vorteile
  - Stark entkoppelte Komponenten
  - Hohe Flexibilität gegenüber Änderungen & Erweiterungen
  - Hoher Wiederverwendungsgrad der Komponenten
  - Unabhängige Entwicklung der Komponenten
  - Leichte Parallelisierung der Berechnungen möglich
  - Überprüfung der Datenkompatibilität dynamisch / statisch
- Nachteile
  - Schwierige Fehlerbehandlung, kein expliziter Kontrollfluss
  - Fehler durch inkompatible Datentypfehler erst zur Laufzeit
  - Häufig zusätzliche Datenkonvertierungen notwendig

#### Plug-In Architektur (Microkernel)

- Zielstellung
  - Stabile, verbreitete Standard-Anwendung (Kern)
  - Funktionalität soll durch Komponenten leicht erweiterbar sein
  - Dritte sollen Komponenten leicht erstellen können
- Lösung
  - Möglichst schlanker zentraler Kern
  - Plugin-Manager verwaltet Komponenten: Laden, Entladen, Zugriffskontrolle, Konfiguration
- Plugin

- Komponente mit Standard-Schnittstelle
- Erweitert Funktionalität (extension point)

- Vorteile
  - Robustes Verhalten
  - Trennung der Zuständigkeiten
  - Erweiterbar, Austauschbar, Wiederverwendbar
  - Geringe Kopplung zu den Komponenten
  - Anpassung an eigene Bedürfnisse möglich
  - Leichte Aufteilung der Entwicklung der Arbeitspakete
- Nachteile
  - Höherer initialer Aufwand
  - Verwaltungsoverhead zur Laufzeit
  - Versionsverwaltung der Komponenten nötig
  - Abhängigkeiten unter den Komponenten schwierig realisierbar
  - Geschickte Definition der Extension Points nötig

#### Repository (Depot, blackboard)

- Zentrale Datenhaltung
  - Datenbankmanagementsystem, Dateisystem
- Anwendungen tauschen Daten nur über Repository aus
- Kontrollfluss z.B. über Signale oder Semaphore
- Gut für datenintensive Verarbeitungsaufgaben geeignet

#### Peer-to-peer

- Gleichberechtigte Partner, “Föderation”
- Verteilte kommunizierende Subsysteme
- Orts- und Umgebungsunabhängigkeit

#### Model-View-Controller (MVC)

- Modell / Sicht / Steuerung
- Trennung verschiedener Aufgabengebiete:
  - Modell: verwaltet Domänenwissen, Daten und Zustand; häufig Datenbank
  - View: Darstellung, Anzeige, GUI
  - Controller: Steuerung der Interaktion, Nutzerbefehle
- Erlauben Austausch von Anzeige- und Speichersystem
- Kontrollfluss
  - Controller steuert
  - View wird über Datenänderungen benachrichtigt (callback)
- Geeignet für interaktive Systeme
- Problem
  - Lose Kopplung zwischen verschiedenen Komponenten
  - Daten werden in verschiedenen Sichten dargestellt
  - Realisierung von GUI's
- Lösung durch drei Komponenten
  - Daten (Modell) enthält die Kernfunktionalität / Durchführung der Geschäftsprozesse, kapselt und Speichert die Daten
  - Sichten bzw. Dialoge (View) stellt die Daten für den Anwender in unterschiedlicher Art dar
  - Logik bzw. Steuerung (Controller) Realisiert die Interaktion mit dem Benutzer, übernimmt die Eingaben vom View und ändert die Daten im Modell, legt die Darstellungsart der Sichten fest
- Vorteile
  - Unabhängige Entwicklung der Komponenten
  - Änderung der Oberfläche ohne Änderung des Modells
  - Unterschiedliche Oberflächen für das selbe Modell
- Nachteile
  - Performance
  - Erhöhter initialer Entwicklungsaufwand

## Frameworks

Was ist ein Framework?

- A framework is a set of prefabricated software building blocks that programmers can use, extend, or customize for specific computing solutions [Taligent]
- Ein framework (Rahmenwerk, Anwendungsgerüst) ist eine Menge von zusammengehörigen Klassen, die einen abstrakten Entwurf für eine Problemfamilie darstellen [nach Pomberger/Blaschek]

Ziele

- Wiederverwendung von Code, Architektur, Entwurfsprinzipien und Verhaltensschema
- Ähnliche Benutzungsschnittstelle

Klassifikation I

- Anwendungs-Framework (application framework)
  - Gibt Systemarchitektur für typische Anwendungsstruktur vor
  - GUI-Framework: Motif, Qt, Swing, ...
- Bereichsspezifisches Framework (domain framework)
  - Expertenwissen für Anwendungsbereich
  - für typische Anwendungen u.a. in den Bereichen Luftfahrt, Produktion, Finanzwesen, Automotive, ...
  - Beispiel: AUTOSAR
- Infrastrukturgerüst (support framework)
  - Gerätetreiber, Anpassung an Hardware
  - Middleware: DCOM, Java RMI, CORBA, WebSphere, ...

Klassifikation II

- Offene Programmgerüste (white box)
  - Erweiterbarkeit durch Vererbung und dynamische Bindung
  - Funktionen konkretisieren durch Ableitung von Basisklassen des Programmgerüsts und Überschreiben vordefinierter Methoden
- Geschlossene Programmgerüste (black box)
  - Erweiterbarkeit durch Definition von Schnittstellen für Module, die für eine konkrete Anwendung in das Gerüst eingesetzt werden können
  - Wiederverwendung durch Komponenten, die sich an Schnittstellen halten; Aufruf über Delegation

Webframeworks - Angular JS

- Clientseitiges Webframework von Google
- Frei verwendbar (Open Source)
- Erstellung von Single-Page-Webanwendungen
- Model View Prinzip
- Vorteile
  - Weitergabe von Expertenwissen
  - Durchdachtes Design: langfristige Aufwandsersparnis
  - Wartungsaufwand reduziert, systematische Tests möglich
  - Prinzipiell sehr hohe Produktivität möglich
  - Erleichtert Integration und Konsistenz verwandter Anforderungen
- Nachteile
  - Erstellung und Einarbeitung aufwändig
  - Zusätzlicher Dokumentations- und Wartungsaufwand
  - Fehlersuche erschwert durch Overhead des Frameworks
  - Kombination verschiedener Frameworks sehr schwierig

## Systemarchitektur und Verteilung

Systemarchitektur

- Aufbau und Elemente der Ablaufumgebung, Hardware
- Häufig enger Zusammenhang mit Softwarearchitektur
  - Architekturmuster
  - Ablaufmuster
- Besonders bei eingebetteten Systemen
- Systemarchitektur hat Einfluss auf Softwarearchitektur
  - Grenzobjekte, Schnittstellen, ...
- Gute Systemarchitektur?
  - Nichtfunktionale Anforderungen
  - Modellierung und Simulation, Lastmessungen

Typische Strukturen

- Zentralrechner (mainframe) mit Terminals
- Server und einfache Stationen
- PCs und Server
- Kommunikationsverbindungen, Sensoren, Speicher, ...
- Blockdiagramm
  - Klassisches Beschreibungsmittel für Systemaufbau
  - Nicht Teil von UML
- Konfigurationsdiagramm
  - meistverbreitetes Hilfsmittel zur Beschreibung der physikalischen Verteilung von System-Komponenten
  - Nicht Teil von UML
- Verteilungsdiagramm (UML deployment diagram)
  - Darstellung der Hardwaretopologie
  - Zuordnung von Artefakten zu Hardwareeinheiten (Knoten)
    - \* Verteilung von Systembestandteilen auf Hardware
  - Kommunikationsverbindung und Abhängigkeiten zwischen Knoten
  - Relativ spät im Projekt Installation / Wartung des Systems

## Globaler Kontrollfluss

Globaler Kontrollfluss

- Ablaufsicht der Architektur
  - Definition nebenläufiger Systemeinheiten (z.B. Prozesse)
  - Steuerung der Abfolge von Einzelfunktionen
  - Synchronisation und Koordination
  - Reaktion auf externe Ereignisse
  - Darstellung z.B. durch Sequenzdiagramme
- Nebenläufigkeit auf Architekturebene
  - Threads , Prozesse, verteiltes System
  - Asynchroner Nachrichtenaustausch
- Einfluss auf Architektur / abhängig von Architektur!
- Ablaufmuster
  - Zentral
    - \* Call/Return (prozedural, synchron)
    - \* Master/Slave (nebenläufig mit zentraler Steuerung)
  - Dezentral
    - \* Ereignisgesteuert (event-driven)
    - \* interrupts
    - \* publish-subscribe (ähnlich observer)
    - \* (selective) broadcast
    - \* Datenflussgesteuert (data flow architecture)

## Sonstiges

Ablauf des OO-Systementwurfs [B. Oesterreich]

- Schichtenmodell definieren
- Verteilungsmodell definieren
- Fachliches Subsystemmodell definieren
- Ablaufverantwortlichkeiten definieren
- Komponentenspezifisches Klassenmodell entwickeln
- Komponentenschnittstelle entwerfen
- Zustandsmodelle weiterentwickeln
- Objektfluss modellieren
- Interaktionsmodelle entwickeln, Attribute definieren
- Dialoge spezifizieren
- Design-Diskurs
- Testgetriebene Entwicklung

Weitere Aufgaben beim Grobentwurf

- Entwurf einer persistenten Datenverwaltung
  - Dateisystem, Datenbank
- Sicherheit
  - Zugriffskontrolle
  - Fehlertoleranz (Daten und Hardware)
  - Protokollfunktionen
- Kontrollfluss
  - Ausnahmen
  - Starten, Initialisieren und Beenden der Anwendung
  - „Randanwendungsfälle“

Notwendigkeit der Architekturdokumentation

- Quellcode aufgrund niedrigen Abstraktionsniveaus ungünstig für Dokumentation
- Überblick und Arbeitsteilung
- Lebensdauer von Systemen länger als geplant
- Fehler und Probleme leichter finden und beseitigen
- Neue Anforderungen mit angemessenem Aufwand erfüllen
- Vereinfachung der Wartung, Pflege, Erweiterung, Wiederverwendung

## Dokumentation

- Grundprinzipien
  - Verständlich aus Sicht des Lesers formulieren (Glossar)
  - Das Warum beschreiben (Entwurfsentscheidungen)
  - Annahmen, Voraussetzungen, Randbedingungen dokumentieren
  - Wiederholungen vermeiden
  - Notation erklären oder Standards verwenden (UML)
    - \* Legende hinzufügen
  - Auf Zweckdienlichkeit prüfen, Reviews durchführen (Inhalt, Qualität)
  - Verschiedene Sichten für verschiedene Zielgruppen

# Feinentwurf

Schließen der Lücke zwischen Grobentwurf und Implementierung

- Identifizieren und Entwerfen von Klassen der Lösungsdomäne
- Identifikation und Verwendung von Entwurfsmustern
- Detaillierte Beschreibung der Klassen
- Beschreibung von Schnittstellen
- Iterativer Prozess!
  - Verbesserung des Entwurfs - Refactoring
  - Optimieren des Entwurfsmodells zur Erfüllung nichtfunktionaler Anforderungen

Objektorientierter Feinentwurf

- Ausgangspunkt
  - Grobdefinition der Architektur, Zerlegung in Subsysteme (evtl. unter Verwendung von Standardarchitekturen)
  - Verteilungskonzept
  - Ablaufmodell
- Ergebnis
  - OO-Modell für jedes Subsystem der Architektur
  - OO-Modell für unterstützende Subsysteme unter Berücksichtigung gewählter Technologien
  - Spezifikationen der Klassen
  - Spezifikationen von externen Schnittstellen

## Klassen- und Objektentwurf

- Klassen der Lösungsdomäne
  - Klassen, die nicht durch objektorientierte Analyse der Anwendungsdomäne entstehen
- Entstehungsgründe
  - Architektur von Software und System
  - nichtfunktionale Anforderungen
  - Beispiele: Kommunikation, Fehlertoleranz, Adapter, Datenhaltung, Effizienz, Benutzerschnittstellenobjekte, Middleware, ...
  - Sichtbare (Grenz- und Steuerungsobjekte) werden schon in der Analyse identifiziert

Klassen identifizieren (responsibility-driven design (Wirfs-Brock, McKean))  $\hat{=}$  Verantwortlichkeits-Prinzip: Sichtweise: Objekte und Klassen sind nicht nur Behälter für Verhalten und Daten, sondern erfüllen in Zusammenarbeit mit anderen Objekten bestimmte Aufgaben eigenverantwortlich  
Responsibility-Driven Design - Begriffe

- Sichtweise auf Softwaresystem
- Application = set of interacting objects
- Object = implementation of role(s)
- Role = set of related responsibilities
- Responsibility = obligation to perform a task or know information
- Collaboration = interaction of objects or roles
- Contract = agreement outlining collaboration terms

Arten von Rollen

- Information holder — knows and provides information —
- Structurer — maintains relationship between objects and information about relationships —
- Service provider — performs work, offers computing services —
- Coordinator — reacts to events by delegating tasks to others —
- Controller — makes decisions and directs other's actions —
- Interfacier — transforms information and requests between system parts —

Hilfsmittel: CRC-Karten

- Candidate (or class), Responsibility, Collaboration
- Informelles Mittel zum
  - Finden,

- Beschreiben und
- iterativen Verändern von Klassen

Ein Objekt

- implementiert eine Schnittstelle und beeinflusst andere Objekte
- wird in drei Teilen entworfen
  - Öffentliche Schnittstelle
  - Art und Weise der Benutzung
  - Innere Details der Funktionsweise
- Kohärenz: zusammengehörende Verantwortlichkeiten in einer Klasse konzentrieren!

## Entwurfsprinzipien

- Kapselung
  - Probleme: Zugriff auf private oder ebenen-fremde Attribute
  - Verwenden von get- und set-Operationen
  - Zusicherungen einhalten
  - Zugriffe zentralisieren
  - Verbalisierung
  - Zugriffsbeschränkung
- Zerlegung
  - Teile und Herrsche
  - Zerlegen in Komponenten
  - Verantwortlichkeitsprinzip: Komponente ist klar für eine Aufgabe verantwortlich
  - Eigenschaften und Schnittstellen im Klassendiagramm
  - Beziehungen zwischen Klassen: Assoziationen
  - Aggregation
    - \* „besteht aus“, „ist Teil von“ oder „Ganzes-/Teile-Beziehung“
    - \* Schwache Bindung der Teile mit dem Ganzen
    - \* Notation: ungefüllte Raute am Ganzen
  - Komposition
    - \* Wie Aggregation, jedoch stärkere Bindung
    - \* Teil nur einem Ganzen zugeordnet
    - \* Nur Multiplizität von 1 oder 0..1 möglich!
    - \* Gefüllte Raute am Ganzen
  - Polymorphie
    - \* Reaktion auf eine Nachricht abhängig vom Typ des Objektes
    - \* Variablen können Objekte verschiedener Klassen aufnehmen (Voraussetzung: Typ der Variablen ist eine gemeinsame Basisklasse der (davon) abgeleiteten Klasse(n) der Objekte)
    - \* Überladen von Operationen
    - \* gleicher Operationsname, unterschiedliche Signatur
    - \* abstrakte Operationen: Virtuelle Operationen ohne Implementierung
    - \* abstrakte Klasse: Klasse mit abstrakten Operationen
    - \* Folgen:
      - \* von abstrakten Klassen können keine Objekte angelegt werden (Implementierung fehlt)
      - \* Abgeleitete Klassen müssen Operation implementieren, damit Objekte angelegt werden können

Vererbung im Entwurf

- In der Analyse: Klassifikation von Objekten, Taxonomie, Spezialisierung/Verallgemeinerung, Organisation von Klassen in Hierarchien
- Verringerung von Redundanz und damit Inkonsistenzen
  - Funktionalität nur einmal implementieren!
  - Spezifikations-Wiederverwendung
  - Implementierungs-Wiederverwendung
- Verbesserung der Erweiterbarkeit
  - Abstrakte Schnittstellen einsetzen!

Vererbung oder Assoziation

- Schlüsselwort Vererbung: ist ein
- Schlüsselwort Assoziation: besteht aus, ist Teil, hat,...
- Vererbung: Unterscheidungsmerkmal definierbar (Diskriminator)
- Vermeide Vererbung, wenn es Alternativen gibt
- Mehrfachvererbung

- Problem: Unabhängige Aspekte der Vererbungshierarchie
- Vermeidung: abstrakte Klassen oder Komposition

Abstrakte Klassen

- Nur Unterklassen, keine Instanzen
- Attribute in Unterklassen füllen
- Notation: Kursiv oder Stereotyp `||abstract|`

Offen / Geschlossen-Prinzip [Meyer 1988]

- Erweiterbarkeit eines Entwurfs
- Offen für Erweiterungen,
  - z.B. durch Vererbung / Polymorphie
  - Virtuelle Operationen verwenden
  - Verändert vorhandenes Verhalten nicht
  - Erweiterung um zusätzliche Funktionen oder Daten
- Geschlossen für Änderungen
  - private Attribute
  - Möglichst protected Operationen
- Beschränkung der Erweiterbarkeit
  - Keine Einschränkungen der Funktionalität der Basisklasse!

Liskovsches Ersetzungsprinzip

- Wenn S eine Unterklasse von T ist, dann können Objekte des Typs T in einem Programm durch Objekte des Typs S ersetzt werden, ohne die Funktion des Programms zu verändern. [Barbara Liskov 1987]
- Engere Definition als „ist-ein“-Beziehung
- Kein unerwartetes Verhalten eines Objektes eines Subtyps
- Methoden, die Objekte der Basisklasse erwarten, müssen auch mit Objekten der abgeleiteten Klasse funktionieren
- Zusicherungen der Basisklasse müssen von der abgeleiteten Klasse erfüllt werden!

Gesetz von Demeter (LoD)

- Gesetz von „schüchternen“ Objekten
- Objekte sollen nur mit Objekten in ihrer unmittelbaren Umgebung kommunizieren
- Aus einer Methode M dürfen (sollten) nur Nachrichten an Objekte gesendet werden, die ...
  - unmittelbarer Bestandteil des Objekts von M sind (super)
  - M als Argument übergeben wurden
  - direkt in M erzeugt wurden
  - (oder sich in globalen Variablen befinden)

- Als Metrik überprüfbar

Ein Objekt sollte

- Nur Methoden aufrufen, die zur eigenen Klasse gehören
- Nur Methoden von Objekten aufrufen, die:
  - Von Attributen referenziert werden
  - Als Parameter übergeben wurden
  - Selbst erzeugt wurden

# Entwurfsmodelle

## Klassendiagramm

- Eigenschaften
  - Modellierung der statischen Struktur (Aufbau)
  - Modellierung der Struktur von Daten
  - Klasse im Mittelpunkt (Aufbau, Beziehungen zueinander)
  - Wichtigstes und bekanntestes Diagramm der UML!
- Elemente des Klassendiagramms
  - Klasse (Attribute, Operationen)
  - Vererbung / Realisierung
  - Assoziationen
  - Beziehungen / Abhängigkeiten
- Attribute
  - Klassenattribut: "X\$static - statisch, nur einmal pro Klasse vorhanden
  - Sichtbarkeit
  - "+" public - im Namensraum sichtbar
  - "#" protected - nur in abgeleiteten Klassen sichtbar
  - "-" package - im Paket sichtbar
  - "private - nur in der Klasse selbst sichtbar
  - Ableitung / "derived - abgeleitetes Attribut
- Weitere Eigenschaften
  - readOnly - nach Initialisierung nicht änderbar
  - composite - Aggregation: Composition
  - redefines X - überschreibe Attr. der Oberklasse
  - subsets X - Teilmenge
  - union - Attribut ist Vereinigung der subsets
  - unique - Elemente eindeutig (Schlüsselattribut)
  - ordered - Elemente sind geordnet (unordered)
  - sequence - Speicherung der Elemente als Liste
  - bag - Elemente sind Multimenge
- Parameterlisten
  - in: Eingangsparameter
  - out: Ausgangsparameter
  - inout: Eingangs- und Ausgangsparameter
  - return: Rückgabewert
- Beziehungen
  - navigierbar/unspezifiziert/nicht-navigierbar
  - ungerichtete/gerichtete Relation/assoziation

## Aktive Klassen

- Reagieren nicht nur, sondern werden von sich aus aktiv
- Z.B. Steuerobjekte
- Als Thread oder Prozess realisiert

## Schnittstellen

- Vereinbarung über Art des Aufrufs
  - Homogenität gleicher Funktionen
  - Enthält:
  - Spezifikation von Operationen
  - keine Implementierung ( Java, nicht UML!)
  - keine Attribute
  - In Java außerdem anstelle von Mehrfachvererbung
- Schnittstellen in UML
  - Funktion ähnlich abstrakter Klasse
  - Meist für technische Aspekte
  - Notation: Stereotyp <code>jjinterface</code> oder grafisch (lollipop notation)
- Verträge („design by contract“)
  - Schnittstelle sagt bisher nichts über Effekt der Klasse aus
  - Vollständige Beschreibung wäre Programm?
  - Vereinfachte Beschreibung für Abfolgen:
  - Vorbedingung: Prädikat, das vor Aufruf gelten muss <code>jjprecondition</code>
  - Nachbedingung: Prädikat, das nach Aufruf gelten muss <code>jjpostcondition</code>
  - Invariante: Prädikat, das immer gilt <code>jjinvariant</code>
  - Jeweils Einschränkungen!

## Protokollrollen - Dynamisches Verhalten von Schnittstellen

- Ohne Sicht auf innere Implementierung (anders als beim Objektlebenszyklus)
- Protokoll = Kollaboration von Protokollrollen (protocol, protocol role)
- Modell: Zustandsautomat
  - Genauer: Spezialisierung
  - Beschreibung der Synchronisation von Objekten

## Entwurfsmuster

- Warum Wiederverwendung?
  - Geringerer Aufwand
  - Das Rad nicht noch einmal neu erfinden
  - Verwenden üblicher, aus Erfahrung gewachsener Strukturen
- ... und warum nicht?
  - Aufwand für Anpassung kann hoch sein!
  - Einarbeiten in teilweise komplexe Schnittstellen
  - Abhängigkeit von externen Komponenten, Zwang zu späterer Portierung

Was ist ein Entwurfsmuster? Eine schematische Lösung für eine Klasse verwandter Probleme (Höhere Ebene: Architekturmuster)

- Wie helfen Muster im Entwurf?
  - Identifizieren von Klassen (Anwendungs- und Lösungsdomäne)
  - Regeln sind abstrakt oder an realen Objekten orientiert
  - Muster: Arten von Rollen bzw. Lösungshinweise für typische Strukturierungsaufgaben
  - Änderbarkeit und Lesbarkeit des Entwurfs verbessern
- Arten von Entwurfsmustern
  - Erzeugungsmuster
  - Strukturmuster
  - Verhaltensmuster
- Erzeugungsmuster
  - Factory Method, Fabrikmethode:
  - Implementierungsvarianten; Erzeugung von Objekten wird an Unterklassen delegiert
  - Abstract Factory, Abstrakte Fabrik:
  - Schnittstelle zur Erzeugung von Familien verwandter Objekte
  - Prototype, Prototyp:
  - Objekterzeugung durch Vorlage und Kopie
  - Builder, Erbauer:
  - Trennung von Erzeugung und Repräsentation komplexer Objekte, für Erzeugung unterschiedlicher Repräsentationen
  - Singleton:
  - Sicherstellung, dass nur ein Objekt einer Klasse erzeugt wird, die einen globalen Zugriff bietet

## Strukturmuster

- Adapter
  - Anpassung der (inkompatiblen) Schnittstelle einer Klasse oder eines Objekts an eine erwartete Schnittstelle
- Bridge, Brücke
  - Abstraktion (Schnittstelle) von Implementierung entkoppeln, um beide unabhängig zu ändern; Impl.-Klasse nur als Verweis
- Decorator, Dekorierer
  - Objekt dynamisch um Zuständigkeiten erweitern (Alternative zur Bildung von Unterklassen)

- Facade, Fassade
  - Einheitliche Schnittstelle zu einer Schnittstellenmenge, vereinfacht Zugriff
- Flyweight, Fliegengewicht
  - Gemeinsame Nutzung kleiner Objekte zur effizienten Verwendung großer Mengen davon (Speicheraufwand)
- Composite, Verbund, Kompositum
  - Zusammenfügen verschiedener Objekte zur Repräsentation von Teil-Ganzes-Beziehungen; Objekte und Kompositionen können einheitlich behandelt werden, Baumstruktur
- Proxy, Stellvertreter
  - Kontrollierter Zugriff auf Objekt durch vorgeschaltetes Stellvertreterobjekt
  - Gründe: Schutz, entfernter Zugriff (remote proxy), smart pointer, Erzeugung on demand

## Adapter

- Vorteile
  - Kommunikation unabhängiger Softwarekomponenten
  - Einfache Erweiterung um zusätzliche Funktionalität
  - Austausch der Komponente durch Änderung des Adapters leicht möglich
- Nachteile
  - Zusätzlicher Adaptierungsschritt benötigt Zeit
  - Schlechte Wiederverwendbarkeit der Adapter
- Bekannte Verwendung, Spezialfälle
  - Facade: Adapter eines Teilsystems
  - Proxy: erweitert die Funktionalität bei gleicher Schnittstelle
  - Brücke: keine Anpassung, sondern vorherige Strukturierung

## Verhaltensmuster

- Command, Befehl
  - Befehl / Operation als Objekt kapseln (Parameterübergabe, Operations-Warteschlangen, logging, Rückgängig machen)
- Observer, Beobachter
  - 1-zu-n-Beziehung zwischen Objekten, so dass die Änderung des zentralen Objekts zu einer Benachrichtigung und Aktualisierung der n (abhängigen) Zustände führt
- Visitor, Besucher
  - Beschreibung und Kapselung einer zu definierenden Operation, die auf einer Objektmenge ausgeführt wird
- Interpreter
  - Repräsentation der Grammatik einer Sprache sowie Interpreter zur Analyse von Sätzen der Sprache
- Iterator
  - Sequentieller Zugriff auf die Elemente einer Sammlung ohne Kenntnis der Implementierung der Sammlung
- Memento
  - Internen Zustand eines Objekts erfassen und speichern, um Objektzustand wiederherstellen zu können
- Template Method, Schablonenmethode
  - Beschreibung des Skeletts eines Algorithmus mit Delegation der Einzelschritte an Unterklassen; Teilschritte können von Unterklassen geändert werden
- Strategy, Strategie
  - Ermöglicht Austausch verschiedener Implementierungen einer Aufgabe ohne Beeinflussung der sie benutzenden Objekte
- Mediator, Vermittler

- Objekt, welches das Zusammenspiel einer lose gekoppelten Objektmenge in sich kapselt. Vermeidet direkten Bezug der Objekte untereinander und ermöglicht unabhängige Änderung des Zusammenspiels
- State, Zustand
  - Ermöglicht Objekt, sein Verhalten abhängig von seinem inneren Zustand zu ändern, als ob es die Klasse wechselt
- Chain of Responsibility, Zuständigkeitskette
  - Vermeidet direkte Kopplung von Auslöser und Empfänger einer Anfrage bzw. Operation. Mehrere Objekte werden nacheinander benachrichtigt, bis die Anfrage erledigt ist

#### Bewertung Observer

- Vorteile
  - Entkopplung von Komponenten und Schichten möglich
  - Broadcast und selective Broadcast möglich
- Nachteile
  - Bei vielen Beobachtern: Benachrichtigung aufwendig
  - Unerwartete Änderung, Änderungskaskaden und Rekursion
  - Abmelden der Beobachter vor dem Löschen
- Bekannte Verwendung, Spezialfälle
  - Verwendung im Model-View-Controller Muster
  - Qt: Signal / Slot-Prinzip ähnlich

#### Anwendung von Entwurfsmustern

- Untersuche Anwendbarkeit und Konsequenzen
- Analysiere Struktur, Teilnehmer und Kollaborationen
- Wähle aus dem Anwendungskontext Namen für Teilnehmer
- Spezifiziere die teilnehmenden Klassen
  - Deklariere Schnittstellen, Vererbung und Variablen
  - Identifiziere existierende Entwurfsklassen, die durch das Muster beeinflusst werden
- Wähle anwendungsspezifische Namen für Operationen
- Implementiere Operationen entsprechend den Verantwortlichkeiten und Kollaborationen des Musters

## Klassenbibliotheken und Komponenten

#### Klassenbibliotheken

- Zusammenfassung von Modulen, Klassen, etc.
- Mit einem bestimmten (abstrakten) Zweck
  - Abstrakte Datenverwaltung, Templates
  - Grundlegende System-Aufgaben
  - Untere Kapselungs-Schicht des Laufzeitsystems oder der Programmierumgebung
  - Numerische Routinen, Simulation, ...
- Wird in Anwendung eingebunden (importiert), API
  - Objekte instanziiieren oder Klassen ableiten
- Meist passiv: Kontrollfluss wird von Anwendung gesteuert
- Beispiele: stdlib, MFC, GNU scientific library, Java 3D, IPP

#### Komponentenbasierte Entwicklung

- Bausteinorientierte Programmierung (component-ware)
- Softwareentwicklung: Konstruktion aus vorgegebenen Bausteinen
- Entsprechung für Wiederverwendung: Generische Bausteine (components)
  - Anpassbar, zusammensetzbar
- Werkzeuggestützte bzw. grafische Kompositionsmechanismen
- Beispiele: Java Beans, Enterprise Java Beans (EJBs), Microsoft COM+
- Komponenten-Entwicklung oft auch projektspezifisch
- Warum Komponenten

- Monolithische, proprietäre Software führt zunehmend zu Problemen
- Zunehmend verteilte Anwendungen mit offener Struktur und Internet-Anbindung
- Zusammensetzen der Funktionalität aus standardisierten Elementen, die über offene Schnittstellen kommunizieren
- Komponenten sollen Flexibilität bei sich ändernden Anforderungen erhöhen
- Weg aus der „Software-Krise“?

#### • Eigenschaften von Komponenten

- müssen von ihrer Umgebung und anderen Komponenten unabhängig und getrennt sein
- Kontextabhängigkeiten: benötigte Komponenten-Infrastruktur und Systemressourcen
- Kapseln ihre angebotenen Funktionen
- Werden immer als ganze Einheit eingesetzt; alle Bestandteile sind enthalten (Archiv-Datei)
- Sind nicht von Kopien ihrer selbst unterscheidbar
- Klare Spezifikation der Schnittstelle nötig; explizit definierte Interaktionen mit Komponenten und Umgebung
- Komposition durch Dritte: Endbenutzer, Komponenten-Hersteller und Komponenten-Integrator; meist nur kompilierter Code verfügbar

#### Komponenten für Client/Server-Architekturen

#### • Wichtige Aspekte

- Transaktionen
- Sicherheit
- Ressourcenverwaltung
- Persistenz

#### • Komponentenkonzept für Server-Komponenten

- meist unsichtbare Komponenten
- standardisierte Realisierung der wichtigen Eigenschaften für Client/Server-Anwendungen
- Realisierung: Enterprise Java Beans (EJBs) innerhalb eines Java Enterprise Edition Servers

## Dokumentation

#### Dokumentation des Feinentwurfs

#### • Möglichkeiten

- Eigenständiges Dokument
- Erweiterung des Lastenhefts / Grobkonzepts
- Eingebettet in den Quellcode (Werkzeug, z.B. Javadoc)

#### • Inhalt

- Ähnlich Grobkonzept
- Zusätzlich detaillierte Modelle
- Abwägungen des Objektentwurfs
- Klassenschnittstellen



# Implementierung

## Aufgaben der Implementierung

- Aus Spezifikationen Programm(code) erzeugen
- Aufbauend auf Ergebnissen des Feinentwurfs
  - Algorithmen konzipieren
  - Datenstrukturen realisieren
  - Umsetzen in konkreter Programmiersprache
  - Dokumentation
  - Untersuchung des Zeit- und Speicherbedarfs
  - Test und Verifikation
- „Programmieren im Kleinen“

## Konventionen und Werkzeuge

### Konventionen beim Programmieren

- (Coding Rules, -conventions, -standards)
- Regeln für verständliche Programme
  - „wie“ sollte Quellcode formal und strukturell gestaltet sein
  - Bezeichner, Einrückungen, Dokumentation, Dateien, ...
  - Strukturierung: Block, Methode, Klasse, Package
- Firmenspezifische Regeln
  - Festlegung Entwurfsprinzipien (z.B. keine Mehrfachvererbung)

### Namenskonventionen

- Klasse
  - (mit) Substantiv, „UpperCamelCase“
  - Beispiele: Account, StandardTemplate
- Methode
  - (mit) Verb, Imperativ (Aufforderung), „lowerCamelCase“
  - Beispiele: checkAvailability(), getDate()
- Attribut, Variable
  - (mit) Substantiv, „lowerCamelCase“
  - Beispiele: anzahlAutos, fensterBreite
- Konstante
  - Nur Großbuchstaben, Worte mit „\_busammengesetzt
  - Standardpräfixe: „MIN.“, „MAX.“, „DEFAULT.“, ...
  - Beispiele: NORTH, BLUE, MIN\_WIDTH, DEFAULT\_SIZE

### Formatierungs-Richtlinien

- Entsprechend Schachtelungstiefe einrücken, aber nicht zu weit
- Einheitliche Verwendung von Leerzeilen und Leerzeichen
- Einheitliche Dateistruktur verwenden
  - Eine .java-Datei pro Klasse
  - Ein Verzeichnis für jedes package
- Werkzeuge: source beautifier, oft in IDEs enthalten
- Editor: syntax highlighting
- Navigationswerkzeuge
  - Auf- und Zuklappen, Inhaltsverzeichnis, tagging
  - doxygen, Eclipse etc.

### Änderungsfreundlicher Code

- Wahl von Variablen, Konstanten und Typen orientiert an der fachlichen Aufgabe, nicht an der Implementierung:
  - ‘typedef char name [NAME\_LENGTH]’
  - ‘typedef char firstName [FIRST\_NAME\_LENGTH]’
- Symbolische Konstanten statt literaler Werte verwenden, wenn spätere Änderung denkbar
- Algorithmen, Formeln, Standardkonzepte in Methoden/Prozeduren kapseln

- Übersichtlichkeit: Zusammenhängende Einheit nicht größer als Editorfenster (40-60 Zeilen, 70 Zeichen breit)
- Strukturierte Programmierung (Regeln je nach Schärfe)
  - Kein goto verwenden (in anderen Sprachen als Java)
  - switch nur mit break-Anweisung nach jedem Fall
  - break nur in switch-Anweisungen verwenden
  - continue nicht verwenden (Effekt ähnlich goto)
  - return nur am Ende zur Rückgabe des Werts
- Übersichtliche Ausdrücke
  - Seiteneffektfreie Ausdrücke, schlecht: y += 12\*x++;
- Variablen möglichst lokal und immer private deklarieren
- Wiederverwendung ”äußerer”Namen vermeiden

### Werkzeuge

- Integrated Development Environments (Eclipse, KDevelop)
- Compiler, Linker; Build / Make; Versionskontrolle (git, svn)

## Code-Qualität

### Portierbarer Code

- Code, den man ohne Änderungen in ein anderes System (Compiler, Betriebssystem, Rechner) übertragen kann
  - Kein implementierungsabhängiges Verhalten!
- ANSI C++ Standard ist nicht vollständig definiert
  - Ist das Verhalten nicht festgelegt, unterscheidet der ANSI C++ Standard zwischen:
  - Implementierungsabhängigem, unspezifiziertem oder undefiniertem Verhalten
  - Code, welcher auf implementierungsabhängigem, unspezifiziertem oder undefiniertem Verhalten basiert, ist
  - Nicht portabel und somit häufig verboten
  - Wird unter Umständen ungewollt wegoptimiert

### Implementierungsabhängiges Verhalten

- Compiler übersetzen bestimmte Sprachkonstrukte unterschiedlich, Ergebnis unterscheidet sich
- Voraussetzung:
- Verhalten ist konsistent festgelegt und dokumentiert
- Kompilierung von standardkonformem Code ist erfolgreich
- Beispiel: Speichergröße von Integer-Typen
- char kann signed oder unsigned sein: Nicht damit rechnen!
- 32 Bit System ist wie erwartet
- 16 Bit System: Multiplikation wird mit int durchgeführt → Überlauf → undefiniertes Verhalten
- Unspezifiziertes Verhalten:
- Wie implementierungsabhängiges Verhalten
- Compiler muss sich für ein bestimmtes Verhalten entscheiden
- Muss nicht dokumentiert sein
- Beispiel: Evaluierungsreihenfolge von Funktionsargumenten ‘tuWas(zuerstDas(),oderDochLieberDas());’
- Undefiniertes Verhalten:
- Keinerlei Vorgaben
- Compiler muss mögliches Problem nicht melden
- Keine Voraussage welches Resultat eintritt
- Bereits die Kompilierung kann fehlschlagen
- Oder das laufende Programm kann falsche Resultate liefern.
- Effekt: „Bei mir läuft es aber!?“
- „undefiniertes Verhalten nutzen grenzt an Sabotage!“

### Sicherer Code mit const

- Const Variable - Konstante
  - Stellt sicher, dass sich der Wert nicht verändert
- Const Parameter
  - Übergabeparameter ändert sich nicht innerhalb der Operation

- Z.B. bei Übergabe komplexer Daten als Referenz bzw. Zeiger ‘long calcMeanValue(const image &i)...‘

### • Const Operationen

- Sicherstellen, dass Operation das Exemplar nicht ändert
- Aufruf der const Operation bei const Variablen möglich

### • Verwende const wenn möglich

## Dokumentation

### • Selbstdokumentierende Programme?

- 2001 Int. Obfuscated C Code Contest Winner, Short Program

### Integrierte Dokumentation

- Verständlichkeit, Wartbarkeit - auch für Programmierer!
- Code selbst sollte möglichst verständlich sein
- Dokumentation in Programm schreiben und aktualisieren
- Beschreibung der Bedeutung des Codes!
- Als Konventionen festschreiben
- Programmvorspann
- Kurzbeschreibung Datei / Klasse / Funktion ...
- Verwaltungsinformationen
  - Autor, Datum, Version, Projekt, ToDo, FixMe, ...
  - Zustand: geplant, in Bearbeitung, vorgelegt, akzeptiert

### • Laufende Kommentare im Quellcode

### Programmierer-Dokumentation

- Als eigenes Dokument elektronisch oder gedruckt
- Einstieg in Programmverständnis (z.B. Bachelor-Arbeit)
- Konsistenz mit Quelltext? Verweise?
- Technische Unterstützung: JavaDoc (Java), doxygen (C++)
- Ergänzt Java-Programm: Dokumentation HTML, PDF,

### Benutzerdokumentation

- Benutzer-Handbuch, Online-Dokumentation
- Unterstützung ohne Support?
- Vollständige und fehlerfreie Beschreibung der Benutzung
  - Beispiele, screen shots

### • Arten: Tutorial, Beschreibung,

### Benutzer-Unterstützungssysteme

- Integrierte Hilfe (Suchfunktion, balloon help / tool tips)
- Assistenz-System (Zustandsabhängige Anleitung)
- Tutor-System zum Erlernen
- Bug-Listen, Mailinglisten, Diskussionsforen

## Codegenerierung

Bezug zwischen Modell und Programmcode

- Vorwärtsmodellierung: Modell - Code
- Rückwärtsmodellierung: Code - Modell
  - Außerdem: Modelltransformation, Refaktorisierung
- Idealfall: Automatische Übersetzung durch SW-Werkzeug (in beiden Richtungen)
  - „Modellbasierte Entwicklung“
- Statisch: Beispiel Klassendiagramm - Quelltext der Klassen mit allen Vererbungsbeziehungen, Attributen und Methodensignaturen (Klassen-Stümpfe mit leeren Methodenrümpfen zum Ausfüllen)
- Dynamisch: Beispiel Zustandsdiagramm - Quelltext der Zustandssteuerung einer Klasse

Weitere statische Transformationen

- Abbildung von Assoziationen auf Sammlungen
- Abbildung von Verträgen auf Ausnahmen
- Abbildung von Objektmodellen auf Datenbankschemata
- Abbildung von Entwurfsmustern auf Codefragmente

Optimierung des Entwurfsmodells

- Grund: nichtfunktionale Eigenschaften
- Zugriffspfade
- Klassen in Attribute umwandeln
- Verzögerung von Berechnungen
- Zwischenspeicherung aufwändiger Ergebnisse

Codegenerierung aus StateCharts

- Einfachste Möglichkeit: Switch (Case) Statement
- Zustände werden durch Datenwerte repräsentiert
  - Aktueller Zustand: einzelne skalare Variable
- Jedes Ereignis wird durch Methode implementiert
- Ausgehend von aktivem Zustand wird bei Eintreffen eines Ereignisses der entsprechende Programmcode ausgeführt
- Abhängig von Zustandsvariable wird Aktion ausgeführt und der Folgezustand eingestellt
- Wird in einer Klasse realisiert
- Sinnvoll für einfache, „flache“ Modelle
  - Sonst Logik für Hierarchie nötig

Anpassung der Generierung

- Verschiedene Zielsprachen (Java, C++, ...)
- Model2Text-Transformationen
  - Verschiedene Generatoren, z.B. Eclipse Modelling Project
- Generierung aus dem Modellierungswerkzeug
  - Parametrisierung der Codegenerierung
  - Generierungsvorlagen

Weitere Werkzeuge

- Compiler-Compiler: Syntaxbeschreibung wird in lexikalische Analyse (tokenizer) und Syntaxanalyse-Programm transformiert (lex & yacc / flex & bison / antlr)
- Codegenerierung für grafische Benutzungsoberflächen aus grafischer Beschreibung: GUI toolkits
- XML-Parser
  - XSLT, DOM, SAX, ...

## Implementierung aktiver Objekte

Realisierung aktiver Entwurfsobjekte

- Reagieren nicht nur (Methodenaufruf), sondern implementieren eigenes Verhalten
- Aktive Klassen, z.B. Steuerobjekte

Arten von Programmabarbeitung

- Sequentiell: es gibt immer genau einen nächsten Schritt, alle Schritte werden nacheinander ausgeführt
- Parallel: Spezielle Hardware bzw. Mehrkernprozessor, mehrere Befehlsfolgen werden echt parallel bearbeitet
- Quasi-parallel: Ein Prozessor arbeitet mehrere Befehlsfolgen in freier Einteilung ab
- Nebenläufig: Oberbegriff für Parallel und Quasi-parallel
  - concurrent

Vorteile

- Höhere Geschwindigkeit
- Kein aktives Warten auf Ereignisse
- Getrennte Implementierung unabhängiger Aspekte

Ergebnisse eines Programms

- Ein Programm, dessen Ablauf eindeutig vorherbestimmt ist, nennt man deterministisch (deterministic)
- Ein Programm, das bei gleichen Eingaben gleiche Ausgaben produziert, heißt determiniert (determined)
- Programme in üblichen Programmiersprachen sind sequentiell, deterministisch und determiniert
- Grund: Herkömmliche Programmiersprachen sind durch das von-Neumann-Modell geprägt
- Determinismus nicht notwendig für Determiniertheit!
  - Determiniertheit nebenläufiger Programme: Synchronisation
  - Vermeidung von Schreib/Schreib und Schreib/Lese-Konflikten

Java Threads

- Verwaltung durch die Java Virtuelle Maschine (JVM)
- Realisierung der Threads ist je nach Implementierung der JVM unterschiedlich
  - Abbildung auf Betriebssystem-Threads (z.B. unter Windows weitverbreitet)
  - Realisierung durch die JVM (z.B. unter Unix und in Java-fähigen Browsern)
  - Nachteile: Keine Ausnutzung von Multiprozessorssystemen durch die VM; Zuteilungsstrategie für Threads ist in derzeitigen Implementierungen unterschiedlich
- Threads arbeiten immer im Adressraum der JVM (eigener Prozess) und sind außerhalb dieser nicht sichtbar

Erzeugung eines Threads

- Unterklasse der Basisklasse „Thread“ bilden ‘class MyThread extends Thread’
- Problem: keine Mehrfachvererbung, daher Alternative nötig (Beispiel: Applet):
  - Schnittstelle „Runnable“ implementieren
  - ‘class MyThread implements Runnable’
- Die vordefinierte Schnittstelle Runnable ist definiert als

Starten eines Threads

- Eine Klasse, die Runnable implementiert, muss wie Unterklassen von Thread immer eine run()-Methode definieren
- Seiteneffekt der Runnable-Schnittstelle

- Instanzen der Klasse werden nebenläufig zu den anderen laufenden Threads ausgeführt
- Ausführung beginnt mit der Methode run ()

• Ablauf

- Thread-Objekt erzeugen
- Thread starten mit t.start()
- start() ruft implizit run() auf

Synchronisation von Threads

- Gezielte Einschränkung der Nebenläufigkeit
- Gründe
  - Zugriffsbeschränkung, gegenseitiger Ausschluss
  - Abhängigkeiten, einseitige Synchronisation
- Methoden: Semaphore, Monitore, Schlossvariablen, ...

Java: Monitore

- Zugriffsoperationen werden in Klassen zusammengefasst
- Gegenseitiger Ausschluss: Spezifikation der betroffenen Zugriffsoperation als synchronized

## Verifikation und Testen

Wie erreicht man qualitativ hochwertige Software?

- Wissen, Erfahrung und Methodenkompetenz der Programmierer
- Projektstruktur, klare Verantwortlichkeiten
- Kosten- und Zeitdruck? Änderungen?
- Programmier- und Testmethoden
  - pair programming, code reading etc.
  - Qualitätsverantwortlicher, automatisiertes Testen
- Technische Unterstützung
  - Z.B. Versionierung, Dokumentation, Testen, Entwicklungsumgebung

Begriffe

- Zuverlässigkeit: Maß für Übereinstimmung des Systemverhaltens mit Spezifikation
- Grund für Unzuverlässigkeit:
  - Fehler (bug, fault): fehlerhafter Programmcode o.ä.
  - Der Begriff „Bug“:
    - \* Schon vor Computern als Begriff für Fehler benutzt
    - \* Motte im Relais des Computers Mark II Aiken (1947)
- Fehlerhafter Zustand (error): Fehler hat zur Laufzeit zu einem internen fehlerhaften Zustand geführt, der möglicherweise zu einem Ausfall führt
- Störfall, Ausfall (failure): Abweichung vom spezifizierten Verhalten, meist mit negativen Folgen

Vergleich System / Systemmodell

- Anspruch guter Software: System entspricht Systemmodell (Korrektheit)
- Problem: System nicht vollständig automatisch erzeugbar!
- Auswege
  - Fehlervermeidung (Inspektion, pair programming, ...)
  - Nachweis, dass System dem Modell entspricht - Verifikation
  - Überprüfen, ob System dem Modell entspricht - Testen
  - Fehlertoleranz (durch Redundanz)

Verifikation

- Mathematisch formaler Beweis, dass ein Programm einer Spezifikation genügt
- Vorteil: wenn anwendbar, dann vollständiger Beweis
- Problem: für viele (realistisch große) Fälle nicht anwendbar

- Zu aufwändig
- Umgebung muss ebenfalls verifiziert werden
- Auch in der Theorie nicht immer entscheidbar: Halteproblem, Gödelscher Unvollständigkeitssatz
- Theoretische Informatik: Berechenbarkeitstheorie, formale Semantik; aktives Forschungsgebiet
  - model checking

## Testen

- Systematischer Versuch, Defekte in der Software zu finden
- Ingenieurtechnik zur Erhöhung des Vertrauens in Softwaresysteme, aber: unvollständig!
  - Kann nur die Anwesenheit von Fehlern nachweisen, aber nicht Korrektheit (Abwesenheit von Fehlern)!
- Aufgabe: Unterschiede zwischen Modell und System finden
- Destruktiv im Gegensatz zu sonstigen SWE-Aufgaben
  - Daher sollten nicht (nur) Entwickler selbst testen

## Testplanung

- Testen ist aufwändig, deshalb ist gute Planung nötig!
- Testplanung sollte bereits mit der Anforderungsanalyse beginnen und im Entwurf verfeinert werden (V-Modell, Test-First-Ansatz)!
- Typische Bestandteile einer Test-Spezifikation (Testdrehbuch)
  - Phasenmodell des Testprozesses
  - Zusammenhang zur Anforderungsspezifikation, z.B. dort festgelegte Qualitätsziele
  - Zu testende Produkte
  - Zeitplan für die Tests
  - Abhängigkeiten der Testphasen
  - Aufzeichnung der Testergebnisse
  - Hardware
  - und Softwareanforderungen

## Arten von Tests

- Komponententest: Fehler in einzelnen Objekten oder Subsystemen, losgelöst vom umgebenden System
  - Umgebung muss nachgebildet werden
- Integrationstest: Zusammenspiel von Komponenten
  - Vollständiges System: Systemtest; Szenarios
- Strukturtest: innere Zustände, Interaktionen
- Funktionstest: Anforderungen aus Lastenheft
- Leistungstest: nichtfunktionale Anforderungen
- Benutzbarkeitstest: Fehler in der Benutzungsschnittstelle, Verständlichkeit, Akzeptanz bei Anwendern
  - Prototypen
- Akzeptanztest, Installationstest: Kunde, Abnahme

## Komponententests

- Überprüft Verhalten einer Systemkomponenten im Vergleich zur Spezifikation
- Da Tests bereits frühzeitig stattfinden sollten, ist Umgebung meist nicht vollständig implementiert
  - Teststumpf (stub, dummy) simuliert aufgerufene Komponenten
  - Testtreiber simuliert aufrufende Komponenten
- Vorgehensweisen
  - Bottom-up
  - Top-down
  - Sandwich
  - Schichtenweises Testen

## Systematisches Testen

- Testfall
  - Beschreibung, Name
  - Zu testende Komponente, Testgegenstand (Pfad, Aufrufart)
  - Eingabedaten (Testdaten)
  - Erwartete Ergebnisse („Orakel“)
  - Protokoll (erzeugte Ausgaben)
  - Bewertung des Ergebnisses
- Weitere Begriffe
  - Regressionstest: erneute Durchführung eines Tests anhand einer geänderten Version des Testgegenstands
  - Alphatest: Test eines Prototypen durch Benutzer
  - Betatest: Test der vollständigen Software durch Benutzer

## Funktionaler Test (black box test)

- Testfallauswahl beruht auf Spezifikation
- Ohne Wissen über inneren Aufbau
- E/A-Zusammenhang

## Äquivalenzklassen im funktionalen Test

- Problem: alle Kombinationsmöglichkeiten der Eingangsdaten sind zu umfangreich für vollständigen Test
- Mögliche Einschränkung: Bildung von Äquivalenzklassen der Eingangsdaten, für die ähnliches Verhalten erwartet wird
- Basierend auf Anwendungsdomäne
- Äquivalenzklasse = Teilmenge der möglichen Datenwerte der Eingabeparameter
- Test je eines Repräsentanten jeder Äquivalenzklasse
- Finden von Äquivalenzklassen
  - Zulässige / unzulässige Teilbereiche der Datenwerte
  - Unterteilung der Bereiche nach erwarteten Ausgabewerten

## Grenztests

- Ergänzung von Äquivalenztests: Spezialfälle
- Rand der Äquivalenzklasse
- Außerdem: Sonderfälle, erwartete Problemfälle (technisch)

## Strukturtest (white box test, glass box test)

- Testfallauswahl beruht auf Programmstruktur
- Wie erreicht man möglichst vollständige Abdeckung?
- Kontrollflussorientiert
  - Anweisungsüberdeckung anhand Quellcode
  - Zweigüberdeckung und
  - Pfadüberdeckung anhand des Flussgraphen reduzierte Variante: bounded interior Pfadtest
- Datenflussorientiert
  - defines / uses-Verfahren: Abarbeitungspfade von Definition zu jeder Verwendung von Variable oder Objekt durchlaufen
- Zustandsorientiert

## Testaktivitäten und Werkzeuge

Wann wird getestet?

- Während der Implementierung!
  - Auch wenn Schreiben von Tests scheinbar unproduktiv ist
  - Tests sind unbeliebt, da Probleme aufgezeigt werden
  - Aber: spätes Testen erhöht Aufwand!
- Inkrementell, d.h. für jede überschaubare Änderung
- Spezielle Vorgehensweise: test first-Ansatz

Wie wird getestet? Geplant, systematisch, möglichst automatisiert, dokumentiert, wiederholbar  
Testplanung

- Iterative Erstellung eines Testplans / Testdrehbuchs
- Aktivitäten - Festlegen von
  - Teststrategie (Testumfang, Testabdeckung, Risikoabschätzung)
  - Testziele und Kriterien für Testbeginn, -ende und -abbruch
  - Vorgehensweise (Testarten)
  - Testumgebung (Beschreibung)
  - Hilfsmittel und Werkzeuge zum Testen
  - Testspezifikation
  - Testorganisation (Termine, Rollen, Ressourcen)
  - Fehlermanagement (Werkzeugunterstützung)
- Ergebnis: Inhalt des Testplans

## Testspezifikation

- Auswahl der zu testenden Testfälle
- Definition einzelner Testfälle
  - Grundlage: Anwendungsfälle, Anforderungen, Fehlermeldungen (Zuordnung notwendig)
- Aktivitäten
  - Testfallfindung und Testfalloptimierung
  - Testfälle beschreiben (was genau ist zu testen)
  - Randbedingungen finden (Abhängigkeiten zu anderen Testfällen)
  - Festlegen und Erstellen der Eingabedaten
  - Festlegungen zum Testablauf und zur Testreihenfolge
  - Festlegen Soll-Ergebnis
  - Festlegung der Bedingung(en) für 'Test erfüllt'; ...

## Testvorbereitung

- Tätigkeiten im Vorfeld des Tests
- Aktivitäten
  - Vorbereitung der Tests entsprechend Testplan
  - Bereitstellen der Dokumente (Testspezifikation)
  - Verfügbar machen von Werkzeugen (Fehlermanagement)
  - Aufbauen der Testumgebung(en)
  - Integration der Entwicklungsergebnisse in die Testumgebung (Software installieren, konfigurieren, ...)
  - Bereitstellung von Testdaten bzw. Eingabedaten in die Testumgebung
  - Benutzer und Benutzerrechte anlegen

## Testdurchführung

- Durchführung der spezifizierten Tests
  - Manuell
  - Automatisiert
- Aktivitäten
  - Auswählen der zu testenden Testfälle
  - Bereitstellen der Testdaten
  - Starten, Überwachen, Abbrechen, Beenden
  - Erfassung des Ist-Ergebnisses zur Auswertung
  - Besondere Vorkommnisse dokumentieren
  - Umgebungsinformationen für den Testlauf archivieren, ...

## Testauswertung

- Überprüfung der Ergebnisse
- Vergleich Ist-Ergebnis mit Soll-Ergebnis
- Entscheidung Testergebnis (ok / Fehler)
- Bei Fehler:
  - Klassifizierung (z. B.: Fehlerursache, Fehlerschwere etc.)
  - Fehler reproduzieren!
  - Angemessene Fehlerbeschreibung und –erläuterung: Nur ausreichend dokumentierte Fehler sind gültig und können bearbeitet werden!
  - Fehler im Fehlermanagement eintragen (Bug report)
  - Testfall bleibt offen

## Bugbeschreibung

- Möglichst genaue Beschreibung des Fehlers

- Ziel ist die Reproduzierbarkeit des Fehlers
- Zuordnung zu Projektplan: Meilenstein, Version
- Fehlerklassifikation
  - defect: Fehler in einer bestehenden Funktionalität
  - enhancement / feature: Funktionale Anforderung oder Erweiterung der bestehenden Funktionalität
  - task: Allgemeine Aufgabe
- Priorität festlegen
  - Unterschiedliche Stufen
  - Festlegung innerhalb eines Unternehmens / Projektes
- Prioritäten von Fehlern (bugs)

#### Testabschluss

- Zusammenfassen der Tests
- Gesamtstatus dokumentieren und kommunizieren
- Entscheidungen herbeiführen z.B.: Auslieferung?
  - Ziele erreicht - nächste Schritte (Auslieferung)
  - Tests vorzeitig beenden oder unterbrechen
  - Gründe dokumentieren
  - Vollständiger Nachtest oder Teilttest möglich?
- Unterlagen archivieren

#### Testautomatisierung

- Automatische Code-Validierung
  - Statisch: lint (C), Compiler
  - Dynamisch: run-time checking, memory leaks etc.
- Beispiel: Test-Framework JUnit

#### Behebung funktionaler Fehler

- Log-Ausschriften bzw. Signale
- Debugger: Vorwärts / Rückwärts
- Haltepunkte setzen: Bedingte Haltepunkte für Schleifen
- Darstellung der Variablenbelegung, Werte setzen
- Analyse des Aufruf-Stacks

#### Behebung nichtfunktionaler Fehler

- Geschwindigkeit: profiling z.B. mit Eclipse TPTP
- Aufrufe, Zeitverbrauch in Methoden usw.

#### Memory Leaks, JProbe

- Runtime heap summary: Welche Objekte verbrauchen Speicher?
- Reference graph: Wer referenziert diese Objekte, so dass sie nicht per garbage collection gelöscht werden?

## Softwareverteilung

#### Softwareverteilung (deployment)

- Prozess zur Installation von Software auf
  - Anwender-PC's, Servern, Maschinen in Produktion ...
- Steuerung der Installation der Software
- Voraussetzungen für die Software schaffen
  - Schulungen planen und durchführen
- Softwareverteilung ist ein kritischer Prozess!
  - Fehler können zu vielen Störungen und Ausfällen führen
- Ziele
  - Automatische Installation, Konfiguration und Wartung einer großen Anzahl von Systemen mit geringem Aufwand
  - Erreichen eines störungsarmen und sicheren Betriebs
  - Möglichst einheitliche Softwareversionen auf allen Systemen

#### Installationsarten

- Erstinstallation
- Software-Update (Software-Aktualisierung)
  - Aktualisierung der Software, Daten oder Konfiguration
- Hotfixes und Service Packs
  - Nur bestimmte Teile der Software werden aktualisiert
  - Meist nur Fehlerbehebung, keine neuen Features
- Upgrade
  - Erweitert eine Software deutlich um neue Funktionen
- Unbeaufsichtigte (automatische) Installation
  - Installation erfolgt ohne Benutzereingriff
  - Periodisch, durch Aufruf des Anwenders, beim Programmstart
  - Einstellungen in einem Skript festgelegt oder werden als Parameter übergeben

#### Installationshilfsmittel

- Installationsprogramm (Installer)
  - Windows: Windows Installer, InstallShield
  - Linux: RPM, Port, APT
  - MAC-OS: Installer, WarpIn
- Installations-Script
- Installationsanweisung

#### Software-Rollout

- Vorgang des Veröffentlichens und Verteilens von Softwareprodukten auf entsprechende Clients
- Anzahl der Clients kann weit über 10.000 liegen!
- Rollout abhängig von verschiedenen Kriterien (Vorherige Installation, Hardwarekonfiguration, Zeit, Kunde)
- Rollout-Varianten
  - Zentral / Dezentral
  - Manuell (Benutzer löst Installation aus)
  - Automatisiert (ohne Benutzerinteraktion)

#### Vollständige Verteilung (big bang)

- Alle Installationen werden mit einem Mal installiert
- Sehr hohes Risiko
- Eventuelle Fehler führen zu vielen fehlerhaften Zuständen oder Störfällen - führt zu hohem Druck
- Eventuelle Fehler müssen schnell gelöst werden (Provisorium)
- Sehr kurze Einführungsphase
- Rollback-Mechanismus sehr empfohlen

#### Rollback

- Wiederherstellung des Ursprungszustands
- Technische Realisierung muss ermöglicht werden

#### Pilotierte Einführung

- Einführung für wenige ausgewählte Installationen
- Sehr hohe Sicherheit
- Festlegung der späteren Rollout-Schritte
- Benötigt zusätzliche Zeit
- Geringere Auftrittswahrscheinlichkeit von Fehlern

#### Schrittweise (iterative) Einführung

- Einführung erfolgt schrittweise mit einer definierten Anzahl
- von Installationen (Rampe - ramp-up)
- Höhere Fehlerwahrscheinlichkeit → Bessere Reproduzierbarkeit → schnelleres Finden von Fehlern → Erfahrungsgewinn
- Begrenztes Risiko, mittlerer Zeitaufwand

# Vorgehensmodelle

## Einführung

Wie läuft Softwareerstellung ab?

- (oder besser, wie sollte sie ablaufen?)
- Aufgaben und Phasen siehe vorangegangene Kapitel
- Wann wird was getan? Abhängigkeiten?
  - Sequentiell / nebenläufig,
- Prozessmodelle der Softwareentwicklung
  - Regelwerke, Erfahrungen, best practices für große Projekte
  - Aktives Entwicklungsgebiet
- Erweiterbar zum Software-Lebenszyklus mit Inbetriebnahme, Wartung, Außerdienststellung usw.

Softwareentwicklungsprozess/Vorgehensmodell

- Methode zur Erstellung von Softwaresystemen
- Systematisch, rational und schrittweise erfolgreicher Weg vom Problem zur Lösung
- Ziel: Softwareentwicklungsprozess übersichtlich, plan- und strukturierbar
- Zerlegung des Softwareentwicklungsprozesses in überschaubare Einheiten
- Unternehmensspezifisch, anpassbar, erweiterbar
  - Eigene Varianten, evtl. projektabhängig
  - An spezielle Bedürfnisse des Informationsbereichs angepasst
  - Kein allgemeingültiges Vorgehen
  - Einsetzbar in verschiedenartigen Projekten

Phasen

- Überwiegend zeitliche Zerlegung
- Zeitlich begrenzte Phasen
- Auch inhaltlich und organisatorisch begrenzte Phasen möglich
- Teilprozesse / Aktivitäten
  - Inhaltliche Zerlegung
  - Satz von Aufgaben
  - Verteilung der Teilprozesse / Aktivitäten auf verschiedene Phasen
  - Begleitet von unterstützenden, übergreifenden Aktivitäten

Aufgabe

- Erzeugt Arbeitsergebnis (Artefakt)
- Verbraucht Ressourcen (z.B. Arbeitskraft, Zeit, Ausrüstung)

Arbeitsergebnis (oder Artefakt)

- Dokument, Modell, System, Programmcode Lastenheft, Spezifikation, Glossar, Handbuch usw.
- Intern zu lieferndes Ergebnis

Teilnehmer und Rollen

- Verantwortungsbereich eines Teilnehmers (z.B. Kunde, Projektmanager, Entwickler, Architekt)
- Rolle bearbeitet / enthält Satz von Aufgaben

Unterstützungsprozesse / -Aktivitäten

- Projektmanagement
  - Projektplanung, -verfolgung und -steuerung
  - Risikomanagement
- Anforderungsmanagement
  - Im Gegensatz zur Anforderungsaufnahme und -analyse
- Qualitätsmanagement
  - Problem Management

- Softwaremetriken (Messung von Softwareeigenschaften)
- Statische + dynamische Analyse (Bestimmung von Schwachstellen)

- Konfigurationsmanagement
  - Versionsverwaltung, Änderungsmanagement
- Dokumentation

## Sequenzielle Modelle

- Wasserfallmodell
  - Abhängigkeiten zwischen Teilergebnissen
  - Ursprung in System- Hardwareentwicklung
    - \* Wurde für die SW-Entwicklung übernommen
    - \* Auch heute verbreitetes Vorgehen bei HW-Entwicklung
  - Sequenzielles Phasenmodell (Abschluss der Phasen)
  - Stark Dokumentengetrieben (Ergebnisse der Phasen)
  - Unterteilung in abgeschlossene Phasen:
    - \* Analyse
    - \* Design/Entwurf
    - \* Implementierung
    - \* Test & Integration
    - \* Einführung, Betrieb & Wartung
  - Alternativ:
    - \* Planung
    - \* Definition
    - \* Entwurf
    - \* Implementierung
    - \* Test & Integration
    - \* Einsatz und Wartung
  - Vorteile
    - \* Einfach und verständlich, bekannt und verbreitet
    - \* Erleichterte Planung und Steuerung
  - Nachteile
    - \* Idealisierte Annahme rein sequentiellen Ablaufs
    - \* Starke Abhängigkeiten zwischen Teilergebnissen
    - \* Ungeeignet, falls Anforderungen zu Beginn unklar
    - \* Unflexibel gegenüber Änderungen
    - \* Erst sehr spät greifbare Ergebnisse

- Erweitertes Wasserfallmodell
  - Verbesserung für Änderungen und Fehler - Rückschritte
- Alternative Arten von Phasenmodellen
  - Sequenziell
    - \* Phasen strikt nacheinander, Dokumenten-orientiert
    - \* Keine Änderungen abgeschlossener Artefakte
  - Nebenläufig
    - \* Phasen laufen teilweise parallel für bessere Zeitnutzung
    - \* Weiterhin keine Änderungen fertiger Dokumente
  - Inkrementell
    - \* Unterteilung des Produkts in Teile
    - \* Schnellere Auslieferung von Teilfunktionalität nach vollständiger Aufnahme der Anforderungen
  - Alternative: Evolutionär
    - \* Anforderungen entwickeln sich im Projekt
    - \* Ausgelieferte Versionen ergeben neue Anforderungen
  - V-Modell [Boehm]
    - \* Explizite Adressierung der Qualitätssicherung
    - \* Entwicklung des V-Modells in Deutschland
      - Leitfaden, in Bundesbehörden ab 1990er verbindlich
      - Version V-Modell 97 erweitert um Werkzeuganforderungen, Rollen und Submodelle der Beteiligten
      - Kritik: schlecht skalier
      - und anpassbar, zu unflexibel,

- bürokratisch, nicht an moderner OO-SWEntw. orientiert
- \* V-Modell XT (extreme tailoring)
  - Aktuelle Version ab 2004/05
  - Einfache projektspezifische Anpassbarkeit
  - Assistent [www.v-modell-xt.de](http://www.v-modell-xt.de) (kleines Projekt 40 Dok.!)
  - Überprüfbarer Projektfortschritt
  - AG/AN-Sichten und Schnittstellen, Ausschreibungen
  - Gesamter SW-Lebenszyklus

## Iterative Modelle

Iterativer Entwicklungsprozess

- Spezifikation, Entwurf und Implementierung müssen immer wieder verändert und angepasst werden
- Häufiges Integrieren, Validieren und Testen
- "You should use iterative development only on projects that you want to succeed." [Fowler]

Inkrementelle Entwicklung

- Wenn möglich, sollte es immer einen lauffähigen (unvollständigen) Prototypen geben
- Neue Funktionen sofort integrieren
- Neue Versionen gegenüber Anforderungen Validieren

Spiralmodell

- Risikogetrieben: Größte Projektrisiken identifizieren und als erstes bearbeiten (Prototyp?)
- Spirale = iterativer Zyklus durch dieselben Schritte
- Ziele jedes Durchlaufs aus alten Ergebnissen ableiten
- Kosten/Nutzen abwägen
- Regelmäßige Überprüfung des Prozessmodells
- Anpassbar
- Hoher Management-Overhead, große Projekte

Unified Process

- Vorgehensmodelle zur objektorientierten Softwareentwicklung
  - Ivar Jacobson, Grady Booch and James Rumbaugh: The Unified Software Development Process, (Rational/IBM), 1999
- Phasen der Entwicklung
  - Anfang, Ausarbeitung, Erstellung, Überleitung
  - Unterteilung in Iterationen
  - Definition von Meilensteinen
- Definition von Kernprozessen, die in den Phasen ablaufen
  - Geschäftsprozessmodellierung, Anforderungsanalyse, Analyse & Design, Implementierung, Test, Auslieferung
  - In unterschiedlichen Anteilen parallel ablaufend!

Prinzipieller Ablauf des Unified Process

- Haupteigenschaften des UP
  - Inkrementell, iterativ, evolutionär
  - Anwendungsgetrieben
  - Architekturzentriert
  - Risikoorientiert
  - Ereignisorientiert
  - Nutzt die UML als Notationssprache
  - Erweiterbar / Anpassbar
- Verbreitete Vertreter
  - Rational Unified Process - RUP (Rational / IBM)
  - Open Unified Process - OpenUP (Eclipse Foundation)
  - Object Engineering Process - OEP (oose GmbH)

- Vorteile
  - Vorteile der Eigenschaften (iterativ, inkrementell, anpassbar, ...)
  - Berücksichtigung des Risikos
  - Passend für objektorientiertes Paradigmas und UML
  - Tool-Unterstützung
  - Reaktion auf Änderungen möglich
  - Industriestandard
- Nachteile
  - Hoher Bürokratischer Aufwand → Tailoring notwendig!
  - Relativ hohe „Lernkurve“ aller Beteiligten
  - Keine schnelle Reaktion auf Änderungen
  - Keine integrierte Qualitätssicherung

## Agile Methoden

- Ausgangspunkt
  - Ziv’s Unsicherheitsprinzip des Software Engineering: Unsicherheit ist im Software-Entwicklungsprozess und den Produkten inhärent und unvermeidlich.”(Ziv, 1996)
  - Humphrey’s Prinzip der Anforderungsunsicherheit: In einem neuen Software System werden die Anforderungen solange nicht komplett bekannt sein, bis die Anwender damit arbeiten.”(Humphrey, 1995)
  - Wegner’s Lemma: Es ist unmöglich, ein interaktives System komplett zu spezifizieren.”(Wegner, 1995)
- Ziele
  - geringer bürokratischer Aufwand
  - Hauptziel ist die Softwareentwicklung
  - nur wenige Regeln bzw. Verhalten definiert
  - sehr flexibel gehaltenes Vorgehen
  - stark Anwendungs- und Ereignisorientiert
  - iterativ / inkrementell / evolutionär
  - sehr schnelle Entwicklungsiterationen
  - meist Architekturzentriert
  - auch testgetriebenes Vorgehen möglich
  - Berücksichtigung sozialer Aspekte
  - Softwareentwicklung: Kreative Arbeit von Kreativen

### Das Agile Manifest (2001)

- Individuen und Interaktionen bedeutender als Prozesse und Tools
- Funktionierende Software bedeutender als übermäßige Dokumentation
- Stetige Zusammenarbeit mit dem Kunden bedeutender als Vertragsverhandlung
- Mut und Offenheit für Änderungen bedeutender als Befolgen eines Plans

### Eigenschaften agiler Vorgehensmodelle

- Team ist für Ergebnis verantwortlich und organisiert sich selbst
- Kleine Teams 5-8 Personen
- Definition von Richtlinien, Werten und Prinzipien
- Beispiele für Werte
  - Kommunikation (Kommunikation statt Dokumentation)
  - Einfachheit (KISS „Keep It Simple and Stupid“)
  - Feedback
  - Mut
  - Respekt
- Beispiele für Prinzipien
  - Beidseitiger Vorteil
  - Fehlschläge hinnehmen
  - Ständige Verbesserungen
  - Ständige Lauffähigkeit des Codes
  - Kleine Schritte
  - Wiederverwendung bestehender / bewährter Lösungen
- Beispiele für Praktiken
  - Pair-Programing, Coding Rules

- Kollektives Eigentum / Gemeinsamer Codebesitz
- Testgetriebene Entwicklung
- Ständiges Refactoring
- Keine Überstunden
- Vorteile agiler Methoden
  - Geringer bürokratischer Aufwand
  - Besseres Arbeitsklima (Berücksichtigung Sozialer Aspekte)
  - Ständige Verfügbarkeit einer lauffähigen Version
  - Mögliche / nötige Einflussnahme des Kunden
  - „Freie“ Wahl der Prinzipien/Regeln
  - Vermeidung von Spezialistentum und individuellem Besitz
- Nachteile
  - Schwierigeres Projektmanagement
    - \* Chaotische Vorgehen
    - \* Schwere Planbarkeit des Ergebnisses
  - Notwendige Beteiligung des Kunden
  - Ergebnis ist schwer vorherzusagen

## eXtreme Programming (XP)

- Beck 1999, aus Kritik an „monumentalen Modellen“
- Evolutionäre Entwicklung in kleinen Schritten
  - Möglichst einfaches Design
- Konzentration auf Programmcode als Analyseergebnis, Entwurfsdokument und Dokumentation
- Weglassen von explizitem Design, ausführlicher Dokumentation und Reviews
- Code wird permanent lauffähig gehalten (täglich)
- Schnell und flexibel
- Erfordert Disziplin der Teilnehmer
- Rollen: Projektleiter, Kunde (verfügbar), Entwickler
  - Max. 5-10 Entwickler
- Kunde bestimmt Anforderung und Prioritäten
  - planning game; story cards (use cases)
- Implementierung in kleinen Schritten
  - pair programming, collective code ownership
  - Häufige Releases inkl. Integration
  - Refactoring bei Designänderungen
  - Programmier-Konventionen
- Regelmäßiges automatisiertes Testen
  - test-first Ansatz
- Morgendliches Meeting im Stehen ohne Diskussionen
- 40h-Woche
- XP
  - Sammlung von 12 ”best practices“
  - Test-getrieben
  - Flexibel, effizient
  - Kleine Teams
  - Erfordert Disziplin der Teilnehmer

## Scrum

- Haupteigenschaften
  - Iterativ / Inkrementell, Evolutionär
  - stark Anwendungs- und Ereignisorientiert
  - schnelle Entwicklungsiterationen
- Sprint
  - eine schnelle Iteration: Dauer ca. 30 Tage
  - Festlegung welche Features umgesetzt werden
- Product Backlog
  - Liste der gewünschten Features des Produkts
  - Vom Produkt-Owner priorisiert / Aufwand vom Team geschätzt

- Jeder kann Einträge beisteuern
- Rollen
  - Product Owner
    - \* Erfasst Bedürfnisse der Kunden und Stakeholder
    - \* Pflegt Backlog, definiert, priorisiert Features pro Sprint
  - Scrum Master
    - \* Berät das Team, Überprüft Einhaltung von Werten und Techniken, moderiert die Meetings
    - \* Schützt das Team vor äußeren Störungen
    - \* Repräsentiert Team gegenüber Management
  - Scrum Team (ca. 5-9 Personen)
    - \* Team organisiert sich und die Aufgaben selbst
    - \* Team bedeutet: Zielgerichtet und funktionsübergreifend arbeiten, gemeinsames Ziel verfolgen, selbstloses Handeln, Teamentscheidungen vertreten

### Sprint Backlog

- Für die aktuelle Iteration ausgewählte Aufgaben
  - Aufgabe nicht länger als 2 Tage Aufwand
- Team-Mitglieder wählen Tasks aus - keine Zuweisung
- Restaufwand wird täglich aktualisiert - Burndown Chart
- Team-Mitglied kann Tasks hinzufügen, löschen, ändern
- Darstellung an prominenter Stelle

### Daily Meeting

- ca. 15 Minuten
- Kurze Statusmeldung, Was wurde geschafft? Was ist zu tun? Was behindert den Fortschritt?
- Weiterführende Diskussionen erst im Anschluss

### Sprint Review-Meeting

- Präsentation des Erreichten (Feature Demo)
- Product Owner, Kunde usw. geben Feedback
  - Neue Anforderungen hinzufügen / Neu priorisieren
  - Qualitätsansprüche ändern

### Sprint-Retrospektive

- Rückkopplungsschleife
  - Was war gut und was hat nicht funktioniert?
  - Was kann verbessert werden?

- Nach jedem Sprint
- Diskussion der identifizierten Probleme
- Identifikation von wenigen „Action Items“

### Burndown Chart

- Darstellung der offenen und erledigten Aufwände / Tasks

## Zusammenfassung

- Software-Entwicklungsmethode
  - Elemente
    - \* Darstellung - Notation und Semantik für Modelle, Diagrammtypen, Dokumentvorlagen (Artefakte)
    - \* Vorgehensmodell - Phasen, Arbeitsschritte
    - \* Verfahren - Regeln, Anweisungen, Aktivitäten (+Rollen)
    - \* Werkzeuge
  - Industriestandards: RUP + UML
  - Öffentliche Auftraggeber: V-Modell
  - Firmenintern: eigene Varianten, evtl. projektabhängig
  - Weitere Themen
    - \* Reifegradbeurteilung CMMI, SPICE, ISO 9000
- Charakterisierung von Vorgehensmodellen
  - Sequenziell
    - \* Teilprozesse strikt nacheinander
    - \* Keine Änderungen abgeschlossener Artefakte
  - Nebenläufig
    - \* Teilprozesse laufen teilweise parallel für bessere Zeitnutzung
  - Dokumentgetrieben
    - \* Erstellung von Dokumenten (Artefakte) im Vordergrund
    - \* Festlegung der Dokumente pro Phase
- Charakterisierung von Vorgehensmodellen
  - Iterativ
    - \* Definition einer sich wiederholenden Abfolge von Teil-Prozessen bzw. Aktivitäten
    - \* Schnelles Wiederholen dieser Abfolgen
  - Inkrementell
    - \* Definition und Kontrolle des Fortschritts pro Iteration
    - \* Kleine Erweiterungen
  - Evolutionäres Vorgehen
    - \* Schnelle Prototypen
    - \* Lauffähiger Prototyp jederzeit vorhanden
    - \* Toolunterstützung (Versionierung)
- Charakterisierung von Vorgehensmodellen
  - Ereignisorientiert
    - \* Schnelle Reaktion auf Anforderungsänderungen
    - \* Keine starre Abfolge von Tätigkeiten / Prozessen
    - \* Voraussetzung: Prozesse laufen parallel ab
  - Architekturzentriert
    - \* Starke Gewichtung der Architektur
    - \* Verwendung von Modellen, Mustern und vorhandenem Wissen
  - Anwendungsgetrieben
    - \* Orientierung an den Anwendungsfällen
    - \* Umsetzen, was einem Anwendungsfall zugeordnet werden kann
    - \* Anwender steht im Mittelpunkt (User Stories)
  - Risikoorientiert
    - \* Risiko der Entwicklung wird in Planung berücksichtigt
    - \* Risiko- / Nutzen-Analyse
  - Test- / Qualitätsgetrieben
    - \* Qualität steht im Vordergrund
    - \* Test wird während oder sogar vor der Implementierungs-phase erstellt
  - Erweiterbar / Anpassbar (tailoring)
    - \* Nur Rahmen des Vorgehens festgelegt
    - \* Konkretes Vorgehen wird an die Bedürfnisse angepasst
    - \* Grundlegende Eigenschaft von Vorgehensmodellen
- Softwareprojekt im Sommersemester
  - Auswahl aus 3 Vorgehensmodellen
    - \* Klassisches Vorgehen
    - \* Unified Process
    - \* Agiles Vorgehen

# Projektmanagement

Was ist ein Projekt?

- Merkmale von Projekten
  - Zielgerichtetes Vorhaben
  - Einmaligkeit
  - Zeitliche, finanzielle und personelle Rahmenbedingungen
  - Abgrenzung zu anderen Vorhaben
  - Projektspezifische Organisation
  - Komplexität (Unterteilung in abhängige Teilaufgaben)
- Unsicherheit vor allem in den Frühphasen
- Risiko durch unbekannte Aufgabe

Was ist Projektmanagement?

- Überbegriff für planende und durchsetzende Aktivitäten zur Vorbereitung und Durchführung eines Projekts
- Management des Problemlösungsprozesses
- Aufgaben
  - Problemabgrenzung
  - Zielfestlegung, Ablaufplanung
  - Planung und Bereitstellung personeller, finanzieller und sachlicher Ressourcen
  - Führen der Projektgruppe und Koordination der Aktivitäten
  - Steuerung und Überwachung des Projektablaufes
  - Zum großen Teil Planungs- und Kommunikationsleistung

## Projektplanung

Planung des Projektablaufs

- Zunächst wieder: Teile und Herrsche!
- Projektstruktur
  - Teilung der Projektaufgabe in Arbeitspakete (work packages) und darin enthaltenen Aktivitäten (activities)
  - Einteilung anhand Produktstruktur, fachlicher Struktur oder Phasenmodell
- Überblick weiterer Planungsaufgaben
  - Bestimmen der Abhängigkeiten
  - Ermitteln der nötigen Ressourcen
  - Schätzen der Aufwände
  - Zeitplan aufstellen
  - Meilensteine definieren

Ablaufplanung

- Abhängigkeiten zwischen Vorgängen A und B
  - Ende-Anfang (Normalfolge)
    - \* B kann begonnen werden, sobald A beendet worden
  - Anfang-Anfang (Anfangsfolge)
    - \* B kann begonnen werden, sobald A begonnen worden
  - Anfang-Ende (Sprungfolge)
    - \* B kann beendet werden, sobald A begonnen worden
  - Ende-Ende (Endfolge)
    - \* B kann beendet werden, sobald A beendet worden
- Netzplantechnik
  - Planungsarten
    - \* Vorwärtsplanung (ab Startzeitpunkt)
    - \* Rückwärtsplanung (ab gewünschtem Projektende)
  - Berechnete Daten für Vorgänge
    - \* Frühester und spätester Anfangszeitpunkt (FAZ/SAZ)

- \* Frühester und spätester Endzeitpunkt (FEZ/SEZ)
- \* Pufferzeiten, Gesamtpuffer
- \* Notation unterschiedlich
  - Allgemein
    - \* Kritischer Pfad: Verzögerung der Projektdauer
    - \* Kritische Vorgänge: Teil des kritischen Pfades

Aufwandsschätzung

- Aus Erfahrungswerten systematisiert
- Versuch, wichtige Einflussfaktoren zu erfassen
  - Metriken für Spezifikationen
  - Komplexität von Teilfunktionen
  - Menge der Funktionen
  - Anpassung durch individuelle Faktoren

## Function Point Analyse

- Ursprung IBM Ende 1970er Jahre
- Funktionsumfang und Schwierigkeitsgrad von Software
- Verschiedene Verfahren
- Jeweils Anzahl x Gewicht
- Summe aller Werte = Unadjusted Function Points (UFP)
- Function Points = UFP x EG
- Einflussgrad EG = 1 + 0.01 x SummeEinflussfaktoren
- Einflussfaktoren: Einfluss auf Anwendungsentwicklung
- Berechnung der Personen-Monate aus Erfahrungen vorangegangener Projekte
  - Aufwand, der von einer Person in einem Monat unter Idealbedingungen erledigt werden kann
- Umrechnung mit Tabelle (nichtlinearer Verlauf)

## CoCoMo II

- Constructive Cost Model [Boehm2000]
- Ausgangspunkt: geschätzte Anzahl Zeilen Quellcode
  - SLOC, source lines of code (zB. aus UFP schätzen)
- Aufwand Personen-Monate (PM)  $PM = A * Size^E \prod_{i=1}^n EM$  mit  $E = B + 0,01 * \sum_{j=1}^5 SF_j$
- Faktoren  $A = 2,94$  und  $B = 0,91$  (anpassbare Koeffizienten)
- Effort multipler  $EM : n = 6..16$ , Tabelle nach Boehm
- Scale factor SF: Fünf Einflüsse auf Rentabilität der Softwareentwicklung
- Notwendige Entwicklungszeit (time to develop)  $TDEV = C * PM^F$  mit  $F = D + 0,2 * (E - B)$
- Faktoren  $C = 3,67$  und  $D = 0,28$ , anpassbar
- Ressourcenplanung
- Zeitplanung (Gantt-Chart / Balkendiagramm)

## Projektdurchführung

Projektorganisation

- Teilnehmer: Personen, Rollen, Verantwortung, Teams
- Linienorganisation:
  - hierarchisch
  - Matrixorganisation
  - Reine Projektorganisation: Mitarbeiter werden aus Organisation herausgelöst und Projektleiter unterstellt

Projektmanager - Rolle und Aufgaben

- Planung, Start, Kontrolle und Beenden des Projekts
- Schnittstelle zur Umgebung des Projekts
  - Kunden, Unterauftragnehmer, interne Kontakte, Verträge

- Team zusammenstellen und steuern
  - 5-7 Mitglieder gemischter Qualifikation
  - Team von äußeren Aufgaben abschirmen
  - Teilaufgaben definieren, vergeben und koordinieren
  - Fortschritt kontrollieren und Probleme beseitigen
- Weitere Ressourcen bereitstellen
- Notwendige Planänderungen erkennen und reagieren

Projektstart

- Nach Abschluss der Planungsphase
- Festlegung von ...
  - Arbeitsstil und interne Organisation
  - Aufgabenverteilung und Themen-Verantwortung
  - Erste Aufgaben, Verantwortliche und Termine
  - Einigung über Meilensteine und Termine
  - Art und Termine der Projekttreffen
  - Informationen und Kommunikationswege
  - Technische Infrastruktur
- Starttreffen des Projekts (kick-off meeting)

Meetings / Projekttreffen

- Regelmäßige Abstimmung der Projektteilnehmer
- Inhalt und Ablauf: geplant (Tagesordnung), Moderator
  - Aktueller Stand
  - Bericht über individuelle Aufgaben
  - Planung des nächsten Termins
- Protokoll
  - Datum, Zeit, Ort, Teilnehmer, Moderator
  - Bezeichnung
  - Tagesordnung mit einzelnen Punkten
  - Kurz und knapp, neutral bei Diskussionen

Fortschrittskontrolle

- Meilensteine: Klar definiertes Zwischenresultat zur Beurteilung des Projektfortschritts
- Besprechung in Projekttreffen
  - Besprechung des Status jedes Meilensteins / jeder Aufgabe
  - Welche Probleme sind aufgetreten / gelöst?
  - Verbleibender Aufwand - Terminverschiebung nötig?
  - Planung der nächsten Schritte (Aufgabe, Termin)

Meilenstein-Trendanalyse

- Technik zur Fortschrittskontrolle
- Überwachung des Projektfortschritts zur Erkennung von Terminverzögerungen
- Bei Verzögerungen:
  - Ressourcen erhöhen
  - Termine verschieben
  - Funktionen reduzieren

Wie viel Planung?

- Aufwand und Detaillierungsgrad der Planung an Projektgröße und „echten“ Aufwand anpassen
- Pläne müssen sich ändern können!
- Projekte sind einmalig und daher unvorhersehbar
- Adaptiv planen: nächste Aufgaben genau, spätere grob
- Einsatz von Projektmanagement-Software
- Projektende
  - Abschlusstreffen
  - Bewertung von Ergebnis und Organisation