

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, Entity Relationship Diagram  
**Systemumgebung** Datenflussdiagramm  
**Algorithmen** Entscheidungstabelle, Pseudocode, Programmablaufplan  
**Dynamisches Verhalten** diskrete Zustände und atomare zustandübergänge; Zustandsautomat, Flow-Chart  
**Objektorientierte Modelle** Klassendiagramme, UML  
**Struktur** Klassen-, Objekt-, Komponenten-, Kompositions-Strukturdiagramm  
**Verhalten** Use-Case-, Aktivitäts- und Zustandsdiagramms  
**Interaktion** Sequenz-, Kommunikations-, Timing-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
- Löschen des Pakets bewirkt Löschen beinhalteneter Elemente
- Definition von Sichtbarkeit / Zugriffsrechte
  - '+' - public (default)
  - '-' - private

## 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
- Black-Box-Darstellung
  - Zur Verfügung gestellte Funktionalität
  - Benötigte Funktionalität
- White-Box-Darstellung
  - Interner Aufbau der Komponente
  - Artefakte, Realisierende physische Einheit

## 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

## 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)
  - Erzeugen/Zerstören von Lebenslinien

## Kommunikationsdiagramm

- Kommunikationsbeziehungen der Kommunikationspartner stehen
- Welche Komponenten arbeiten zusammen → Funktion erfüllen

## Timing-Diagramm

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

## Zustandsdiagramm

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

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)
- Kommunikationsverbindung und Abhängigkeiten zwischen Knoten
- Relativ spät im Projekt Installation / Wartung des Systems

## 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  $Prioritäten = \frac{\text{Nutzen} - \text{Nachteil}}{\text{Kosten} + \text{Risiko}}$
- 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
  - Struktur 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 Kompositionstrukturdiagramm

- 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

- Entwurfsziele identifizieren
- Grobe Systemstruktur festlegen (Architektur)
- Zerlegung in Subsysteme, Spezifikation
- Bewerten der Zerlegung anhand der Entwurfsziele
- Schnittstellen festlegen

## Systemzerlegung

Kopplung/Abhängigkeiten von Subsystemen

- Möglichst lose Kopplung
  - Änderungen haben geringe Auswirkungen
  - Erleichtert Wartbarkeit und Arbeitsteilung
- Mittel zur Verringerung
  - Zusätzliche Unterteilung in Subsysteme
  - Aber: dann größere Komplexität!

### Kopplungsarten

**Daten** vermeiden oder zentralisieren  
**Schnittstellen** gegenseitiges Aufrufen  
**Struktur** gemeinsame Elemente vermeiden über Paketgrenzen

Kohäsion (cohesion)

- Maß für Zusammengehörigkeit der Elemente
- hohe Kohäsion erleichtert Verständnis, Wartung und Anpassung
- Mittel zum Erreichen hoher Kohäsion: Datenkapselung

**Fan-in** Anzahl der Stellen, wo Kontrollfluss auf das betrachtete Modul M übergeht + 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

Prinzipien des OO-Entwurfs

- Keep it simple stupid (KISS)
- Entwerfen nach Verantwortlichkeiten
- Hohe Kohäsion / Geringe Kopplung
- Zyklische Abhängigkeiten vermeiden
- Auf Schnittstellen konzentrieren
  - Abhängigkeiten nur von Schnittstellen
  - Abtrennung von Schnitts. (viele kleine statt wenig große)
  - Umkehr der Abhängigkeiten (dependency inversion)
- Offen / Geschlossen Prinzip

Symptome schlechten Designs

- Starrheit (Änderungen schwer machbar)
- Zerbrechlichkeit (Änderungen werfen Fehler)
- Schlechte Wiederverwendbarkeit

Wann ist ein Entwurf „gut“?

- Korrekt (Anforderungen erfüllen)
- Verständlich und präzise, gut dokumentiert
- Anpassbar
- Hohe Kohäsion innerhalb der Komponenten
- Schwache Kopplung zwischen den Komponenten
- Wiederverwendung
- Kriterien gelten auf allen Ebenen des Entwurfs!

## Architekturmodelle

- 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
- Klasse realisiert alle Attribute und Operationen
- Schnittstellen werden realisiert, nicht instanziiert
- Schnittstelle kann von anderen Schnittstellen erben
- Keine Schnittstellenrealisierung zwischen zwei Interface-Klassen → Generalisierung verwenden
- Wiederverwendung auf sehr hoher Abstraktionsstufe

Schichten-Architektur (layers)

- Zuordnung von Subsystemen zu horizontalen Schichten gleicher Abstraktionsebene
- Komponenten einer Schicht bieten Dienste der darüber liegenden Schicht an

Client-Server (Klient/Anbieter) Two-tier

- Client (front-end): Benutzungsschnittstelle
- Server (back-end): Datenhaltung, evtl. Fachlogik
- Asynchroner Kontrollfluss
- Aufteilung Funktionen Client / Server
- Vorteile
  - 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

Pipes and Filters

- Datenstrom- oder Kontrollflussorientiertes System
- Kombination der Berechnungskomponenten nur vom Typ der Ein- und Ausgabedaten abhängig
- Verwendung von globalen Steuerungskontrollstrukturen
- Vorteile
  - Hohe Flexibilität gegenüber Änderungen & Erweiterungen
  - Hoher Wiederverwendungsgrad der Komponenten
  - Unabhängige Entwicklung der Komponenten
  - Leichte Parallelisierung der Berechnungen möglich
- Nachteile
  - Schwierige Fehlerbehandlung, kein expliziter Kontrollfluss
  - Fehler durch inkompatible Datentypfehler erst zur Laufzeit
  - Häufig zusätzliche Datenkonvertierungen notwendig

Plug-In Architektur (Microkernel)

- Stabile, schlanker verbreitete Standard-Anwendung (Kern)
- Funktionalität durch Komponenten leicht erweiterbar sein
- Dritte sollen Komponenten leicht erstellen können
- Plugin-Manager verwaltet Komponenten: Laden, Entladen
  - Komponente mit Standard-Schnittstelle
  - Erweitert Funktionalität (extension point)
- Vorteile
  - Robustes Verhalten

- Trennung der Zuständigkeiten
- Erweiterbar, Austauschbar, Wiederverwendbar
- Geringe Kopplung zu den Komponenten
- Leichte Aufteilung der Entwicklung der Arbeitspakete

### Nachteile

- Höherer initialer Aufwand
- Verwaltungsoverhead zur Laufzeit
- Versionsverwaltung der Komponenten nötig
- Abhängigkeiten unter den Komponenten schwierig realisierbar

Model-View-Controller (MVC)

- Erlauben Austausch von Anzeige- und Speichersystem
- Model** verwaltet Domänenwissen, Daten und Zustand; Datenbank
- View** Darstellung, Anzeige, GUI
- Controller** Steuerung der Interaktion, Nutzerbefehle
- Geeignet für interaktive Systeme
- Daten enthält Kernfunktionalität, kapselt und speichert Daten
- Sichten stellt die Daten in unterschiedlicher Art dar
- Steuerung realisiert die Interaktion mit dem Benutzer, übernimmt die Eingaben vom View und ändert die Daten im Modell

## Frameworks

- Menge von zusammengehörigen Klassen, für eine Problemfamilie
- Wiederverwendung von Code, Architektur, Entwurfsprinzipien
- Ähnliche Benutzungsschnittstelle
- Klassifikation I
  - Anwendungs-Framework (application framework)
  - Bereichsspezifisches Framework (domain framework)
  - Infrastrukturgerüst (support framework)
- Klassifikation II
  - **Offene Programmgerüste** (white box) Erweiterbarkeit durch Vererbung und dynamische Bindung
  - **Geschlossene Programmgerüste** (black box) Erweiterbarkeit durch Definition von Schnittstellen für Module
- Webframeworks

## Systemarchitektur und Verteilung

- Aufbau und Elemente der Ablaufumgebung, Hardware
- Häufig enger Zusammenhang mit Softwarearchitektur
- Besonders bei eingebetteten Systemen
- Systemarchitektur hat Einfluss auf Softwarearchitektur

## Globaler Kontrollfluss

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

## Weitere Aufgaben beim Grobentwurf

- Entwurf einer persistenten Datenverwaltung (Datenbank)
- Sicherheit
  - Zugriffskontrolle
  - Fehlertoleranz (Daten und Hardware)
  - Protokollfunktionen
- Kontrollfluss
- Dokumentation
  - Verständlich aus Sicht des Lesers formulieren (Glossar)
  - Das Warum beschreiben (Entwurfsentscheidungen)

- Annahmen, Voraussetzungen, Randbedingungen
- Notation erklären oder Standards verwenden (UML)
- Auf Zweckdienlichkeit prüfen, Reviews durchführen
- Verschiedene Sichten für verschiedene Zielgruppen

### Schicht

- Gruppe von Subsystemen in der Zerlegungshierarchie
- Verwandte Dienste
- Ähnlicher Abstraktionsgrad
- Abhängigkeit nur von darunter liegenden!

Repository (Depot, blackboard)

- Zentrale Datenhaltung
- 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

## 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!

Objektorientierter Feinentwurf

- Grobdefinition der Architektur, Zerlegung in Subsysteme (evtl. unter Verwendung von Standardarchitekturen)
- 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
  - Sichtbare Grenz- und Steuerungsobjekte werden schon in der Analyse identifiziert

Responsibility-Driven Design

**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: zusammengehörnde Verantwortlichkeiten in einer Klasse konzentrieren!

## Entwurfsprinzipien

- Kapselung
  - 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
  - Komposition
    - \* Wie Aggregation, jedoch stärkere Bindung
    - \* Teil nur einem Ganzen zugeordnet
    - \* Nur Multiplizität von 1 oder 0..1 möglich!
  - Polymorphie
    - \* Reaktion auf eine Nachricht abhängig vom Typ des Objektes
    - \* Variablen können Objekte verschiedener Klassen aufnehmen
    - \* 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
      - \* Abgeleitete Klassen müssen Operation implementieren, damit Objekte angelegt werden können

Vererbung im Entwurf

- 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

Abstrakte Klassen

- Nur Unterklassen, keine Instanzen
- Attribute in Unterklassen füllen

Offen / Geschlossen-Prinzip

- Erweiterbarkeit eines Entwurfs
- Offen für Erweiterungen,
  - Virtuelle Operationen verwenden
  - Verändert vorhandenes Verhalten nicht
  - Erweiterung um zusätzliche Funktionen oder Daten
- Geschlossen für Änderungen (private)
- 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
- 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)

- 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)

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

## Entwurfsmodell Klassendiagramm

- Elemente des Klassendiagramms
  - Klasse (Attribute, Operationen)
  - Vererbung / Realisierung
  - Assoziationen
  - Beziehungen / Abhängigkeiten
- Attribute
  - Klassenattribut: "X"batic - 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

## Schnittstellen

- Vereinbarung über Art des Aufrufs
  - Homogenität gleicher Funktionen
  - Spezifikation von Operationen
  - keine Implementierung, keine Attribute
- Schnittstellen in UML
  - Funktion ähnlich abstrakter Klasse
  - Meist für technische Aspekte
- Verträge („design by contract“)
  - Vorbedingung: Prädikat, das vor Aufruf gelten muss
  - Nachbedingung: Prädikat, das nach Aufruf gelten muss
  - Invariante: Prädikat, das immer gilt

Protokollrollen - Dynamisches Verhalten von Schnittstellen

- Ohne Sicht auf innere Implementierung
- Protokoll = Kollaboration von Protokollrollen
- Modell: Zustandsautomat
- Beschreibung der Synchronisation von Objekten

## Entwurfsmuster

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** Implementierungsvarianten; Erzeugung von Objekten wird an Unterklassen delegiert
  - Abstract Factory** Schnittstelle zur Erzeugung von Familien verwandter Objekte
  - Prototype** Objekterzeugung durch Vorlage und Kopie
  - Builder** 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** Abstraktion (Schnittstelle) von Implementierung entkoppeln, um beide unabhängig zu ändern; Impl.-Klasse nur als Verweis
- Decorator** Objekt dynamisch um Zuständigkeiten erweitern (Alternative zur Bildung von Unterklassen)
- Facade** Einheitliche Schnittstelle zu einer Schnittstellenmenge, vereinfacht Zugriff

**Flyweight** Gemeinsame Nutzung kleiner Objekte zur effizienten Verwendung großer Mengen davon (Speicheraufwand)

**Composite** Zusammenfügen verschiedener Objekte zur Repräsentation von Teil-Ganzes-Beziehungen; Objekte und Kompositionen können einheitlich behandelt werden, Baumstruktur

**Proxy** kontrollierter Zugriff auf Objekt durch vorgeschaltetes Stellvertreterobjekt

Verhaltensmuster

**Command** Befehl / Operation als Objekt kapseln (Parameterübergabe, Operations-Warteschlangen, logging, Rückgängig machen)

**Observer** 1-zu-n-Beziehung zwischen Objekten, so dass die Änderung des zentralen Objekts zu einer Benachrichtigung und Aktualisierung der n (abhängigen) Zustände führt

**Visitor** 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** Beschreibung des Skeletts eines Algorithmus mit Delegation der Einzelschritte an Unterklassen; Teilschritte können von Unterklassen geändert werden

**Strategy** Ermöglicht Austausch verschiedener Implementierungen einer Aufgabe ohne Beeinflussung der sie benutzenden Objekte

**Mediator** Objekt, welches das Zusammenspiel einer lose gekoppelten Objektmenge in sich kapselt. Vermeidet direkten Bezug der Objekte untereinander und ermöglicht unabhängige Änderung des Zusammenspiels

**State** Ermöglicht Objekt, sein Verhalten abhängig von seinem inneren Zustand zu ändern, als ob es die Klasse wechselt

**Chain of Responsibility** Vermeidet direkte Kopplung von Auslöser und Empfänger einer Anfrage bzw. Operation. Mehrere Objekte werden nacheinander benachrichtigt, bis die Anfrage erledigt ist

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

- 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

Komponentenbasierte Entwicklung

- Bausteinorientierte Programmierung (component-ware)
- Softwareentwicklung: Konstruktion aus vorgegebenen Bausteinen
- Entsprechung für Wiederverwendung: Generische Bausteine (components)
- Werkzeuggestützte bzw. grafische Kompositionsmechanismen
- 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
- 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
  - Sind nicht von Kopien ihrer selbst unterscheidbar
  - Klare Spezifikation der Schnittstelle nötig; explizit definierte Interaktionen mit Komponenten und Umgebung

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

- Möglichkeiten
  - Eigenständiges Dokument
  - Erweiterung des Lastenhefts / Grobkonzepts
  - Eingebettet in den Quellcode
- Inhalt
  - Ähnlich Grobkonzept
  - Zusätzlich detaillierte Modelle
  - Abwägungen des Objektentwurfs
  - Klassenschnittstellen

## 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

## Konventionen und Werkzeuge

**Klasse** (mit) Substantiv, „UpperCamelCase“

**Methode** (mit) Verb, Imperativ (Aufforderung), „lowerCamelCase“

**Attribut, Variable** (mit) Substantiv, „lowerCamelCase“

**Konstante** Nur Großbuchstaben, Worte mit „\_bunammengesetzt“

Formatierungs-Richtlinien

- Entsprechend Schachtelungstiefe einrücken, aber nicht zu weit
- Einheitliche Verwendung von Leerzeilen und Leerzeichen
- Einheitliche Dateistruktur verwenden
- Werkzeuge: source beautifier, oft in IDEs enthalten
- Editor: syntax highlighting

Änderungsfreundlicher Code

- Wahl von Variablen, Konstanten und Typen orientiert an der fachlichen Aufgabe, nicht an der Implementierung
- Symbolische Konstanten statt literaler Werte verwenden, wenn spätere Änderung denkbar
- Algorithmen, Formeln, Standardkonzepte in Methoden/Prozeduren kapseln
- Übersichtlichkeit: Zusammenhängende Einheit nicht größer als Editorfenster
- 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
- Variablen möglichst lokal und immer private deklarieren
- Wiederverwendung „äußerer“ Namen vermeiden

## Code-Qualität

- Portierbarer Code: ohne Änderungen in anderes System (Compiler, Betriebssystem, Rechner) übertragen kann
- Implementierungsabhängiges Verhalten
  - Compiler übersetzen Sprachkonstrukte unterschiedlich
  - Verhalten ist konsistent festgelegt und dokumentiert
  - Kompilierung von standardkonformem Code ist erfolgreich
  - char kann signed oder unsigned sein: Nicht damit rechnen!
- Sicherer Code mit const

**Const Variable** Stellt sicher, dass sich der Wert nicht verändert

**Const Parameter** Übergabeparameter ändert sich nicht

**Const Operationen** Sicherstellen, dass Operation das Exemplar nicht ändert

## Dokumentation

**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
- 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++)
- Ergänztes Java-Programm: Dokumentation HTML, PDF,

**Benutzer** Dokumentation

- Benutzer-Handbuch, Online-Dokumentation
- Unterstützung ohne Support?
- Vollständige und fehlerfreie Beschreibung der Benutzung
- Arten: Tutorial, Beschreibung

## Codegenerierung

- Vorwärtsmodellierung: Modell - Code
- Rückwärtsmodellierung: Code - Modell
- Modelltransformation, Refaktorisierung
- Idealfall: Automatische Übersetzung durch SW-Werkzeug
- Statisch: Quelltext der Klassen mit Beziehungen, Attributen...
- Dynamisch: Quelltext der Zustandssteuerung einer Klasse

Optimierung des Entwurfsmodells

- Zugriffspfade
- Klassen in Attribute umwandeln
- Verzögerung von Berechnungen
- Zwischenspeicherung aufwändiger Ergebnisse

Codegenerierung aus StateCharts

- Einfachste Möglichkeit: Switch (Case) Statement
- Zustände werden durch Datenwerte repräsentiert
- 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

Anpassung der Generierung

- Verschiedene Zielsprachen (Java, C++, ...)
- Model2Text-Transformationen
- Generierung aus dem Modellierungswerkzeug

## 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
- Determiniertheit nebenläufiger Programme: Synchronisation

## 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)
- Technische Unterstützung (Versionierung, Dokumentation, Testen)

Begriffe

**Zuverlässigkeit** Maß für Übereinstimmung des Systemverhaltens mit Spezifikation

**Unzuverlässigkeit** Fehler (bug, fault), fehlerhafter Programmcode

**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

## Verifikation

- Mathematisch formaler Beweis genügt
- Vorteil: wenn anwendbar, dann vollständiger Beweis
- Problem: für viele (realistisch große) Fälle nicht anwendbar
- Theoretische Informatik: Berechenbarkeitstheorie, Semantik

Testen

- Systematischer Versuch, Defekte in der Software zu finden
- Ingenieurtechnik zur Erhöhung des Vertrauens
- Aufgabe: Unterschiede zwischen Modell und System finden
- Destruktiv im Gegensatz zu sonstigen SWE-Aufgaben

Testplanung

- Testen ist aufwändig, deshalb ist gute Planung nötig!
- Testplanung sollte bereits mit der Anforderungsanalyse beginnen
- Typische Bestandteile einer Test-Spezifikation (Testdrehbuch)
  - Phasenmodell des Testprozesses
  - Zusammenhang zur Anforderungsspezifikation
  - Zu testende Produkte
  - Zeitplan für die Tests
  - Abhängigkeiten der Testphasen
  - Aufzeichnung der Testergebnisse
  - Hardware- und Softwareanforderungen

**Komponententest** Fehler in einzelnen Objekten oder Subsystemen, losgelöst vom umgebenden System

**Integrationstest** Zusammenspiel von Komponenten

**Systemtest** Vollständiges System; Szenarios

**Strukturtest** innere Zustände, Interaktionen

**Funktionstest** Anforderungen aus Lastenheft

**Leistungstest** nichtfunktionale Anforderungen

**Benutzbarkeitstest** Fehler in der Benutzungsschnittstelle, Verständlichkeit, Akzeptanz bei Anwendern

**Akzeptanztest, Installationstest** : Kunde, Abnahme

- Da Tests bereits frühzeitig stattfinden sollten, ist Umgebung meist nicht vollständig implementiert

- Teststumpf simuliert aufgerufene Komponenten
- Testtreiber simuliert aufrufende Komponenten
- Vorgehensweisen: Bottom-up, Top-down, Sandwich, Schichtenweise

## 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: geänderte 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

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

- Testfallauswahl beruht auf Programmstruktur
- Wie erreicht man möglichst vollständige Abdeckung?
- Kontrollflussorientiert (Anweisungsüberdeckung anhand Quellcode)
- Datenflussorientiert (Abarbeitungspfade von Definition)
- 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
  - Bereitstellung von Testdaten 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
  - Fehler im Fehlermanagement eintragen (Bug report)

## 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 (Erweiterung)
  - task** Allgemeine Aufgabe
- Priorität festlegen
  - blocker** Schwerwiegender Fehler, führt zur Arbeitsunfähigkeit anderer
  - critical** Schwerer Fehler, grundlegende Funktionalität des Systems nicht gegeben
  - major** normaler Fehler. System kann weiterhin verwendet werden
  - minor** Kleiner Fehler, behindert nicht weiter die Funktionsfähigkeit des Systems
  - trivial** Schönheitsfehler, stört während des Betrieb des Systems nicht.

## Testabschluss

- Zusammenfassen der Tests dokumentieren und kommunizieren
- Entscheidungen herbeiführen
  - Ziele erreicht - nächste Schritte (Auslieferung)
  - Tests vorzeitig beenden oder unterbrechen
  - Gründe dokumentieren
- Unterlagen archivieren

## Testautomatisierung

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

## 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

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

## Softwareverteilung (deployment)

- Prozess zur Installation von Software auf 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!
- 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** Konfiguration  
**Software-Update** Aktualisierung der Software, Daten oder Konfiguration

**Hotfixes/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

## Software-Rollout

- Vorgang des Veröffentlichens und Verteilens
- Rollout abhängig von verschiedenen Kriterien
- Rollout-Varianten
  - Zentral / Dezentral
  - Manuell (Benutzer löst Installation aus)
  - Automatisiert (ohne Benutzerinteraktion)

## Verteilungsarten

**Vollständige Verteilung (big bang)** Alle Installationen werden mit einem Mal installiert (Sehr hohes Risiko)

**Rollback** Wiederherstellung des Ursprungszustands

**Pilot** Einführung für wenige ausgewählte Installationen (Sehr hohe Sicherheit)

**Iterativ (schrittweise)** mit definierten Anzahl von Installationen bessere Reproduzierbarkeit



## Vorgehensmodelle

- Methode zur Erstellung von Softwaresystemen
- Systematisch, rational und schrittweise zur Lösung
- Softwareentwicklungsprozess übersichtlich, plan- und strukturierbar
- Zerlegung des Softwareentwicklungsprozesses in Einheiten
- Unternehmensspezifisch, anpassbar, erweiterbar

### Phasen

- Überwiegend zeitliche Zerlegung
- Zeitlich begrenzte Phasen
- Auch inhaltlich und organisatorisch begrenzte Phasen möglich
- Teilprozesse / Aktivitäten

### Aufgabe

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

### Arbeitsergebnis (oder Artefakt)

- Dokument, Modell, System, Programmcode, Lastenheft,...
- 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** Softwaremetriken, Statische + dynamische Analyse (Bestimmung von Schwachstellen)
- **Konfigurationsmanagement** Versionsverwaltung, Änderungsmanagement
- **Dokumentation**

## Sequenzielle Modelle

- Wasserfallmodell
  - Abhängigkeiten zwischen Teilergebnissen
  - Ursprung in System- Hardwareentwicklung
  - Sequenzielles Phasenmodell (Abschluss der Phasen)
  - Stark Dokumentengetrieben (Ergebnisse der Phasen)
  - Unterteilung in abgeschlossene Phasen:
    - \* Analyse (Planung)
    - \* Design/Entwurf
    - \* Implementierung
    - \* Test & Integration
    - \* Einführung, Betrieb & Wartung
  - Einfach und verständlich, bekannt und verbreitet
  - Starke Abhängigkeiten zwischen Teilergebnissen
  - Ungeeignet, falls Anforderungen zu Beginn unklar
  - Unflexibel gegenüber Änderungen
- 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
- Evolutionär
  - Anforderungen entwickeln sich im Projekt
  - Ausgelieferte Versionen ergeben neue Anforderungen
- V-Modell [Boehm]
  - Explizite Adressierung der Qualitätssicherung
  - Entwicklung des V-Modells in Deutschland
  - V-Modell XT (extreme tailoring)

## Iterative Modelle

- Spezifikation, Entwurf und Implementierung müssen immer wieder verändert und angepasst werden
- Häufiges Integrieren, Validieren und Testen
- 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
- Spirale = iterativer Zyklus durch dieselben Schritte
- Ziele jedes Durchlaufs aus alten Ergebnissen ableiten
- Regelmäßige Überprüfung des Prozessmodells
- Hoher Management-Overhead, große Projekte

### Unified Process

- Vorgehensmodelle zur objektorientierten Softwareentwicklung
- 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!
- Variation
  - Inkrementell, iterativ, evolutionär
  - Anwendunggetrieben
  - Architekturzentriert
  - Risikoorientiert
  - Ereignisorientiert
- Passend für objektorientiertes Paradigmas und UML
- Tool-Unterstützung
- Reaktion auf Änderungen möglich
- Hoher Bürokratischer Aufwand → Tailoring notwendig!
- Relativ hohe „Lernkurve“ aller Beteiligten
- Keine schnelle Reaktion auf Änderungen
- Keine integrierte Qualitätssicherung

## Agile Methoden

### Das Agile Manifest (2001)

- Individuen und Interaktionen > als Prozesse und Tools
- Funktionierende Software > als übermäßige Dokumentation
- Stetige Zusammenarbeit mit Kunden > als Vertragsverhandlung
- Mut und Offenheit für Änderungen > 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
- 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)

- Evolutionäre Entwicklung in kleinen Schritten
- 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

## Scrum

- Iterativ / Inkrementell, Evolutionär
- stark Anwendungs- und Ereignisorientiert
- schnelle Entwicklungsiterationen
- Sprint: eine schnelle Iteration/Umsetzung Features (30 Tage)
- 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** Team organisiert sich und die Aufgaben selbst, gemeinsames Ziel verfolgen, selbstloses Handeln, Teamentscheidungen vertreten
- Sprint Backlog
  - Für die aktuelle Iteration ausgewählte Aufgaben ( $\leq 2$  Tage)
  - Team-Mitglieder wählen Tasks aus - keine Zuweisung
  - Restaufwand wird täglich aktualisiert - Burndown Chart
  - Team-Mitglied kann Tasks hinzufügen, löschen, ändern
- Daily Meeting (15 Minuten)
  - 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?
  - Nach jedem Sprint
  - Diskussion der identifizierten Probleme
  - Identifikation von wenigen „Action Items“
- Burndown Chart: Darstellung der offenen und erledigten Aufwände / Tasks

## 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 =  $1 + 0.01 \times \text{Summe Einflussfaktoren}$
- 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 multiplier  $EM : n = 6..16$ , 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
- Liniorganisation:
  - 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