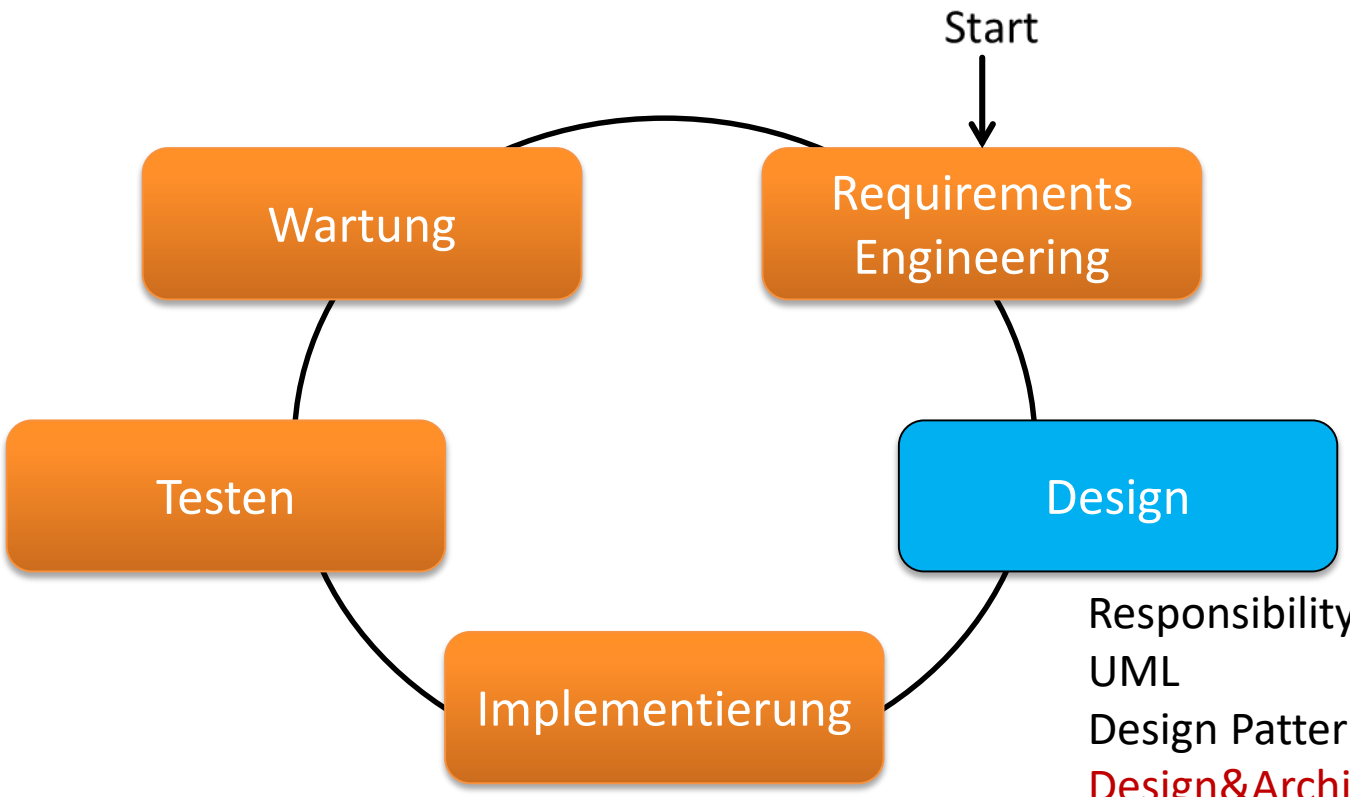
Design

Authors of slides:
Norbert Siegmund
Janet Siegmund
Oscar Nierstrasz
Sven Apel

Einordnung



Responsibility-Driven
UML
Design Patterns
Design&Architecture

Lernziele

- Wichtige Kriterien für das Design von Software kennenlernen
- Qualität von Software-Design bewerten können

Warum Design von Software?

- Für jedes Verhalten gibt es unendlich viele Programme
  - Wie unterscheiden sich diese Varianten?
  - Welche Variante ist die beste?
- Wie soll Variante designed werden, damit sie gewünschte Eigenschaften hat?
- Kosten für Fehler werden größer, je später Fehler bzw. Schwächen entdeckt werden; darum gute Modellierung!

Qualität von Software-Design

Was ist Qualität?

- Google doc
- Interne Qualität
  - Erweiterbarkeit, Wartbarkeit, Verständlichkeit, Lesbarkeit
  - Robust gegenüber Änderungen
  - Coupling und Cohesion
  - Wiederverwendbarkeit

  → Typischerweise beschrieben als Modularität
- Externe Qualität
  - Korrektheit: Erfüllung der Anforderungen
  - Einfachheit in der Benutzung
  - Ressourcenverbrauch
  - Legale und politische Beschränkungen

Design

- Nach der Modellierung werden Methoden definiert und zu Klassen zugeordnet sowie die Kommunikation zwischen den Objekten festgelegt, um die spezifizierten Anforderungen zu erfüllen.

- Wie genau?
  - Welche Methode kommt wohin?
  - Wie sollen die Objekte interagieren?
  - Wichtige, kritische, nicht-triviale Fragestellung!

Kriterien für gutes Design

Modularität

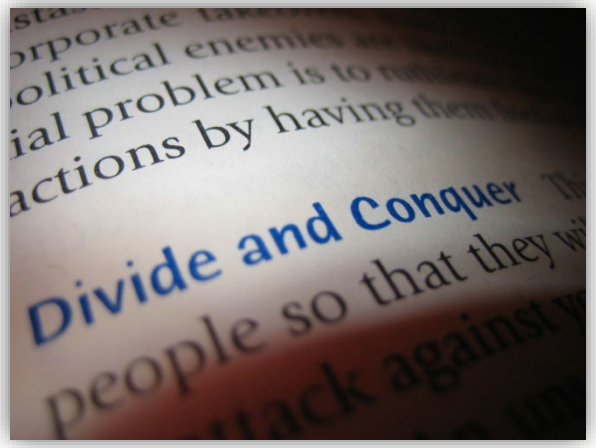
- Beschreibt, in wie weit man ein Softwaresystem aus autonomen Elementen bauen kann, die mit einer kohärenten, einfachen Struktur miteinander verbunden sind

Fünf Kriterien

- Modular Decomposability
- Modular Composability
- Modular Understandability
- Modular Continuity
- Modular Protection

Fünf Kriterien: Modular Decomposability

- Problem wird in wenige kleinere, weniger komplexe Sub-Probleme zerlegt
- Sub-Probleme sind durch einfache Struktur verbunden
- Sub-Probleme sind unabhängig genug, dass sie einzeln bearbeitet werden können
- Voraussetzung: Modular decomposability setzt voraus: Teilung von Arbeit möglich



Fünf Kriterien: Modular Composability

- Gegenstück zu modular decomposability
- Softwarekomponenten können beliebig kombiniert werden
- Möglicherweise auch in anderer Domäne

- Beispiel: Tischreservierung aus NoMoreWaiting kann auch für das Vormerken von Büchern benutzt werden (gutes Design!)

Fünf Kriterien: Modular Understandability

- Entwickler kann jedes einzelne Modul verstehen, ohne die anderen zu kennen bzw. möglichst wenige andere kennen zu müssen

- Wichtig für Wartung
- Gilt für alle Softwareartefakte, nicht nur Quelltext
- Gegenbeispiel: Sequentielle Abhängigkeit zwischen Modulen

Fünf Kriterien: Modular Continuity

- Kleine Änderung der Problemspezifikation führt zu Änderung in nur einem Modul bzw. möglichst wenigen Modulen

- Beispiel 1: Symbolische Konstanten (im Gegensatz zu Magic Numbers)
- Beispiel 2: Datendarstellung hinter Interface kapseln
- Gegenbeispiel: Magic Numbers, (zu viele) globale Variablen

Fünf Kriterien: Modular Protection

- Abnormales Programmverhalten in einem Modul bleibt in diesem Modul bzw. wird zu möglichst wenigen Modulen propagiert

- Motivation: Große Software wird immer Fehler enthalten
- Beispiel: Defensives Programmieren
- Gegenbeispiel: Nullpointer in einem Modul führt zu Fehler in anderem Modul

Fünf Regeln für gutes Design

Fünf Regeln

- Fünf Regeln für qualitativ hochwertiges Software-Design:
  - Direct Mapping
  - Few Interfaces
  - Small Interfaces
  - Explicit Interfaces
  - Information Hiding

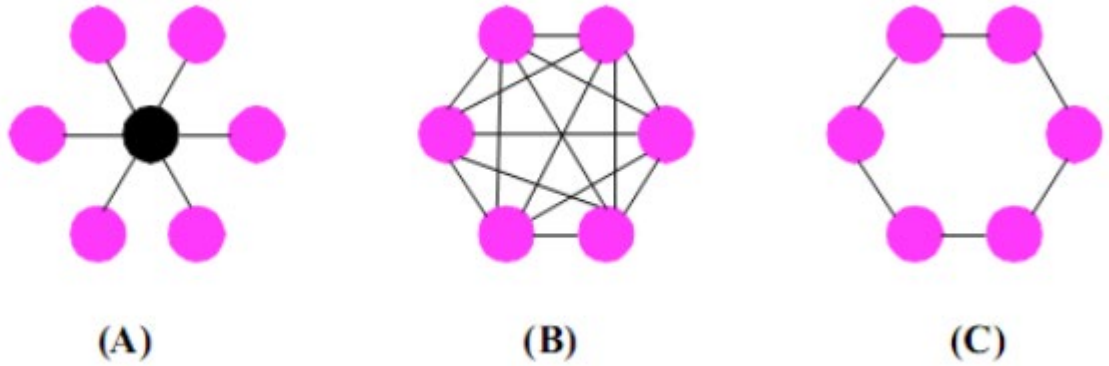
Fünf Regeln: Direct Mapping

- Modulare Struktur des Softwaresystems sollte modularer Struktur des Modells der Problemdomäne entsprechen
- A.k.a. “low representational gap”[C. Larman]

Fünf Regeln: Few Interfaces

- Jedes Modul sollte mit möglichst wenigen anderen Modulen kommunizieren
- Struktur mit möglichst wenigen Verbindungen

Types of module interconnection structures



Fünf Regeln: Small Interfaces

- Wenn zwei Module kommunizieren, sollten sie so wenig Informationen wie möglich austauschen

- Folgt aus continuity und protection, notwendig für composability
- Gegenbeispiel: Big Interfaces ☺
  - Wenn ich jede Methode ins Interface aufnehme, muss jede Klasse, die das Interface implementiert, alle Methoden realisieren -> Vorteil geht verloren
  - Man weiß nicht, welche Methoden entscheidend sind
  - Zu viele Einfallstore für Fehler

Fünf Regeln: Explicit Interfaces

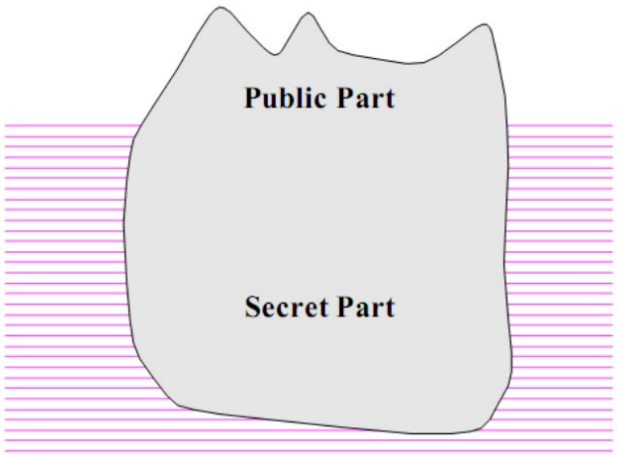
- Wenn zwei Module miteinander kommunizieren, muss das aus dem Interface von mindestens einem hervorgehen

- Gegenbeispiel: Globale Variablen
  - Wer benutzt sie?
  - Welche Auswirkungen (und wo) werden durch Änderungen an der globalen Variable verursacht?
  - Wer ändert sie und wann?
  - Wer war verantwortlich für einen inkonsistenten oder fehlerhaften Zustand?

Fünf Regeln: Information Hiding

- Jedes Modul muss eine Teilmenge seiner Eigenschaften definieren, die nach außen gezeigt werden
- Alles andere wird “versteckt”

- Nicht nur Inhalt, auch Implementierung wird versteckt



GRASP – Pattern

General Responsibility Assignment Software Patterns

GRASP Pattern

- Allgemeine Lernhilfe, um
  - grundlegendes objekt-orientiertes Designen zu verstehen
  - Design-Entscheidungen methodisch, rational und erklärbar zu treffen

- Basiert of Responsibilities (Responsibility-Driven Design)

GRASP-Pattern

- Information Expert
- Creator
- Low Coupling
- High Cohesion

GRASP-Prinzip: Information Expert

- Wer soll die Responsibilities bekommen?
- Das Objekt, das die meisten Informationen hat, diese Responsibilty zu erfüllen

- Beispiel NoMoreWaiting: Wer hat die Information, ob ein Tisch frei ist oder nicht?

GRASP-Prinzip: Creator

- Wer erstellt Instanzen eines Objekts?
- Creator braucht erstelltes Objekt häufig in seinem Lebenszyklus
- Objekt B bekommt Verantwortung, Objekt A zu erstellen, wenn:
  - B A-Objekte aggregiert oder enthält
  - B Instanzen von A-Objekten loggt
  - B Initialisierungsdaten von A hat

GRASP-Prinzip: High Cohesion

Kohäsion ist ein Maß, **wie gut Teile einer Komponente "zusammen gehören".**

- Kohäsion ist schwach, wenn Elemente nur wegen ihrer Ähnlichkeit ihrer Funktionen zusammengefasst sind (z.B., **`java.lang.Math`**).
- Kohäsion ist stark, wenn alle Teile für eine Funktionalität tatsächlich benötigt werden (z.B. **`java.lang.String`**).
  - Starke Kohäsion **verbessert Wartbarkeit** and Adaptivität durch **die Einschränkung des Ausmaßes von Änderungen** auf eine kleine Anzahl von Komponenten.

Es gibt viele Definitionen und Interpretationen von cohesion.
Die meisten Versuche, dies formal zu definieren, sind inadäquat!

GRASP-Prinzip: High Cohesion

- Responsibilities so zuordnen, dass Kohäsion hoch ist
- Faustregel: Klasse mit starker Kohäsion hat meistens wenige Methoden, die verwandte Funktionalität haben, und macht nicht zu viel (d.h., ist keine Gottklasse)

GRASP-Prinzip: Loose Coupling

Kopplung ist ein Maß der **Stärke der Verbindungen** zwischen Systemkomponenten.

- Kopplung ist eng (tight) zwischen Komponenten, wenn sie stark voneinander abhängig sind (z.B., wenn sehr viel Kommunikationen zw. beiden statt findet).
- Kopplung ist lose (loose), wenn es nur wenige Abhängigkeiten zwischen Komponenten gibt.
  - Loose coupling **verbessert Wartbarkeit** und Adaptibilität, da **Änderungen in einer Komponente sich weniger wahrscheinlich auf anderen Komponenten auswirkt**.
  - Responsibility so zuteilen, dass coupling schwach ist