Softwaretechnik

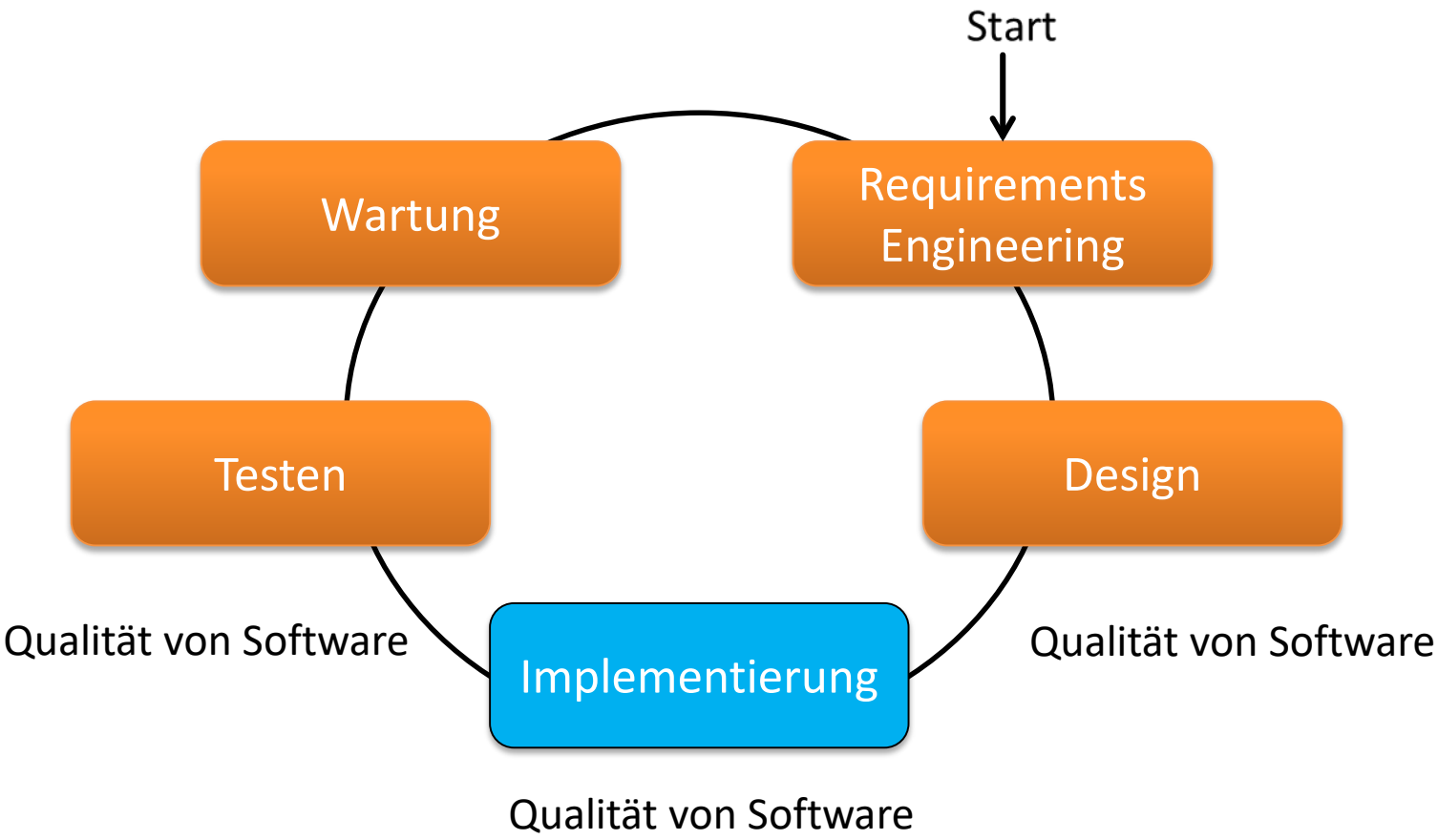
Qualität von Software und Design

SOFTWARE SYSTEME

UNIVERSITÄT LEIPZIG

Prof. Dr.-Ing. Norbert Siegmund
Software Systems

Einordnung


Start
Requirements Engineering
Design
Implementierung
Testen
Wartung
Qualität von Software
Qualität von Software
Qualität von Software

Lernziele

- Wichtige Kriterien für das Design von Software kennenlernen
- In der Lage sein, gutes von schlechten Design zu unterscheiden
- Prinzipien von guter Softwarequalität kennen und anwenden können
- Qualität von Software-Design bewerten können

Was bedeutet gutes Software Design?

- Für jedes Verhalten gibt es unendlich viele Programme
  - Wie unterscheiden sich diese Varianten?
  - Welche Variante hat die beste Qualität?

- Wie soll Variante entworfen werden, damit sie gewünschte Eigenschaften von Qualität besitzt?

Qualität von Software-Design

Was ist Qualität?

- Interne Qualität
  - Erweiterbarkeit, Wartbarkeit, Verständlichkeit, Lesbarkeit
  - Robust gegenüber Änderungen
  - Coupling und Cohesion (was ist das? Wird im Folgenden erklärt)
  - Wiederverwendbarkeit
  - → Zusammengefasst typischerweise beschrieben als Modularität

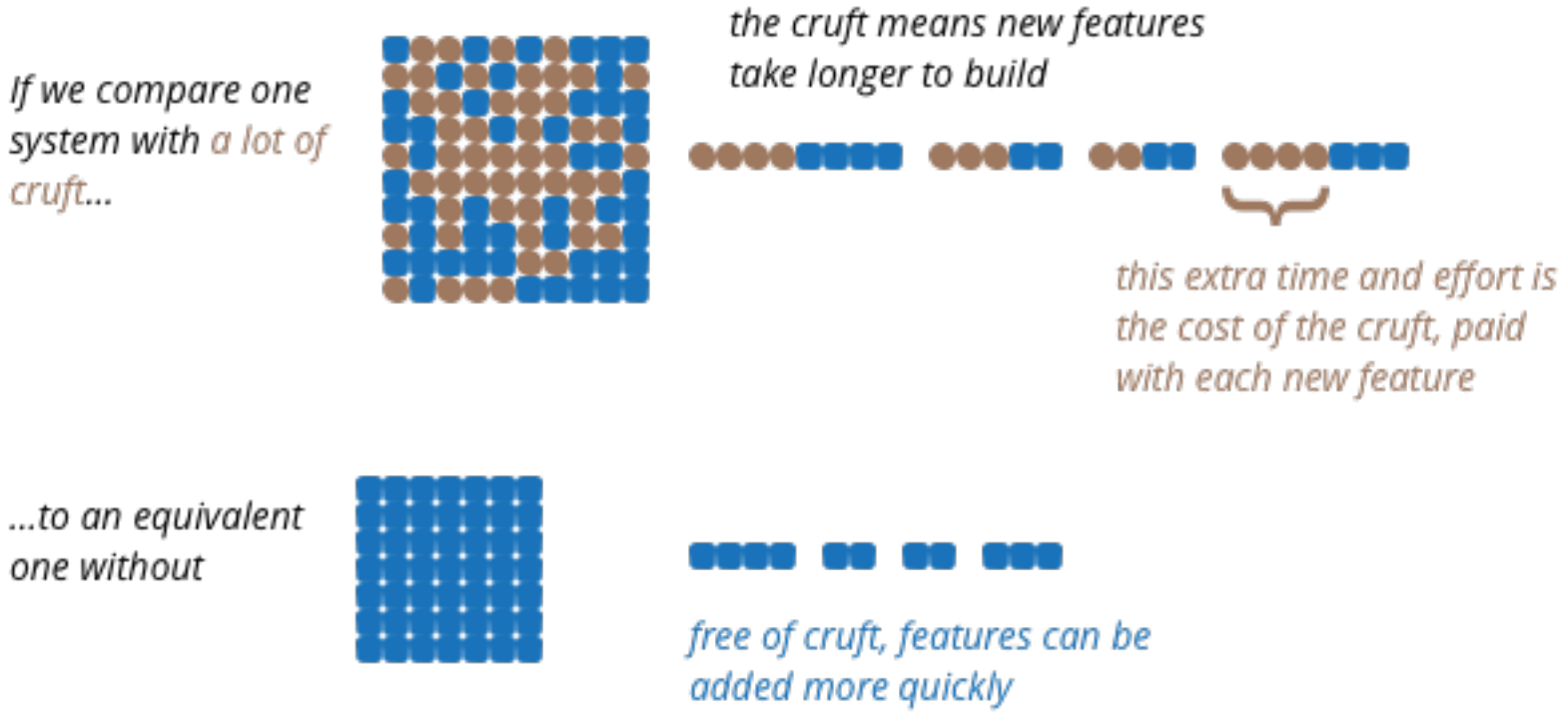
- Externe Qualität
  - Korrektheit: Erfüllung der Anforderungen
  - Einfachheit in der Benutzung
  - Ressourcenverbrauch
  - Legale und politische Beschränkungen

Ist gute Qualität teuer?

- Gibt es einen Trade-off zwischen SW Qualität und Funktionalität?

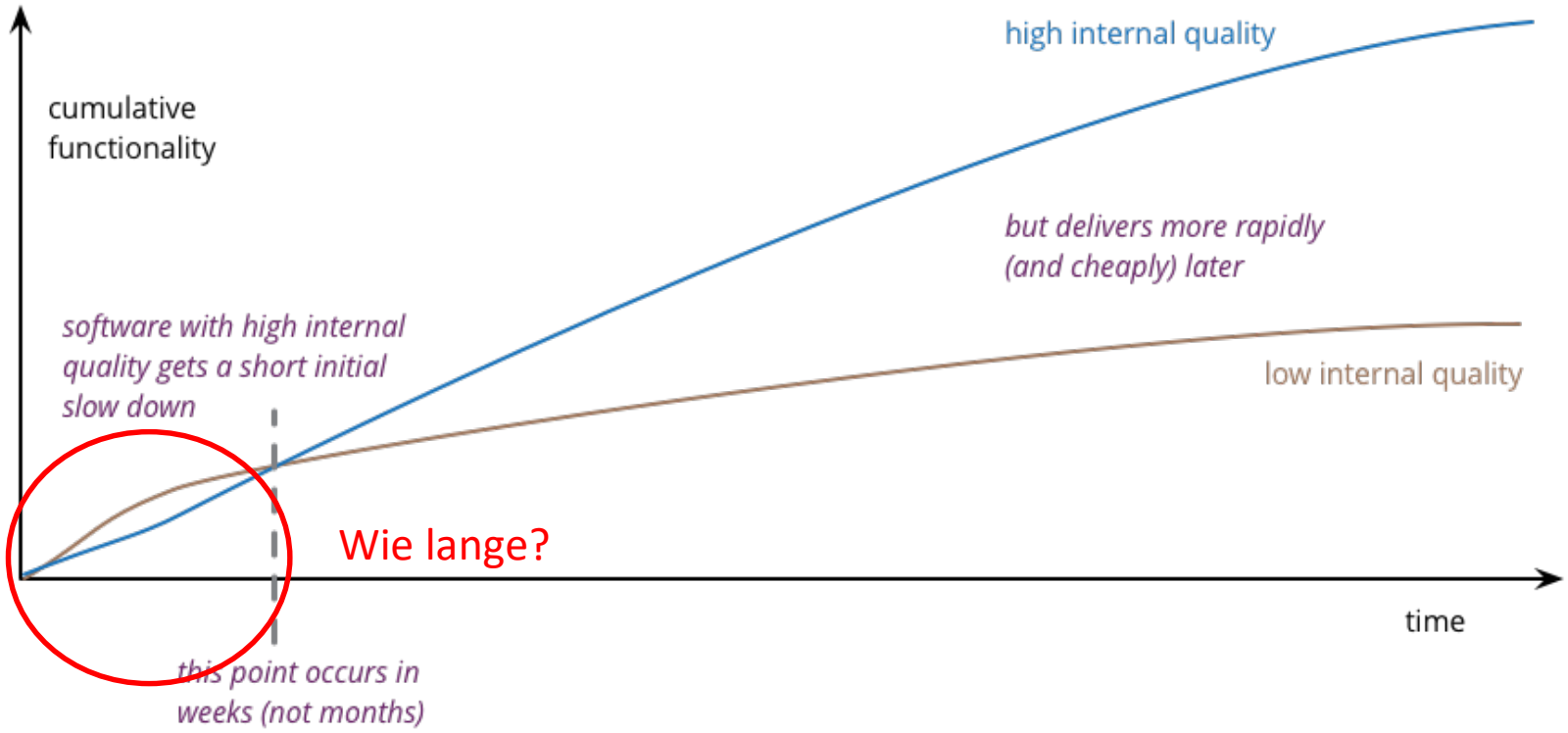
SW Qualität = Günstigere Entwicklung

Schlechte interne Qualität ~ hohe technische Schulden -> langsamere Weiterentwicklung



https://martinfowler.com/articles/is-quality-worth-cost.html

SW Qualität zahl sich aus



https://martinfowler.com/articles/is-quality-worth-cost.html

Bisher: Design

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

Jetzt: Wie gutes Design?

- Wie genau sollte das Design durchgeführt werden, um eine hohe interne Qualität zu erreichen?
  - Welche Methode kommt wohin? Was hat das für Auswirkungen?
  - Wie sollen die Objekte interagieren? Was hat das für Auswirkungen?

  - Wichtige, kritische, nicht-triviale Fragestellung!

Fünf Kriterien für gutes Design

Ziel: Modularität

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

- Dabei helfen 5 Kriterien und 5 Regeln

Exkurs: Coupling / Kopplung

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

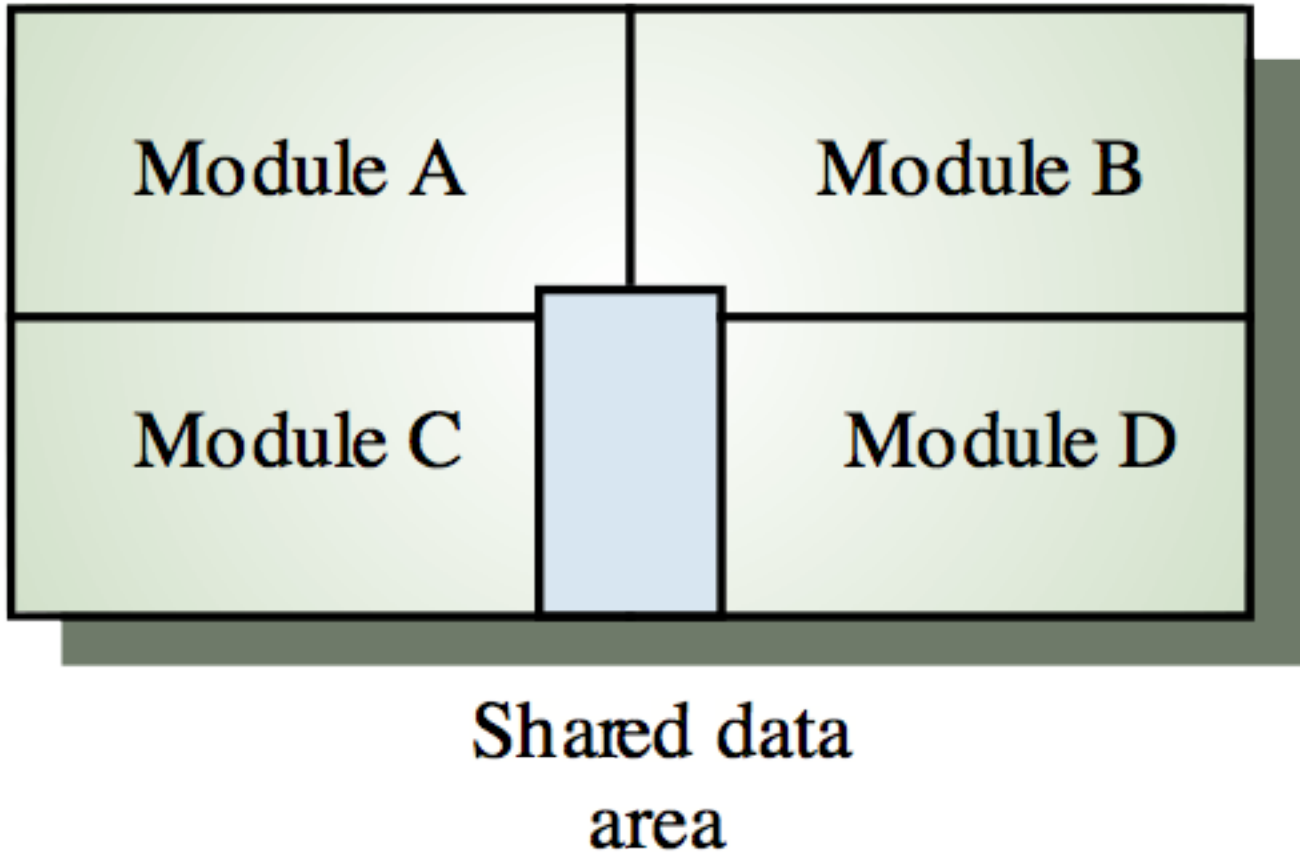
- Coupling ist eng (tight) zwischen Komponenten, wenn sie stark voneinander abhängig sind (z.B., wenn sehr viel Kommunikationen zw. beiden stattfindet).

- Coupling ist lose (loose), wenn es nur wenige Abhängigkeiten zwischen Komponenten gibt.

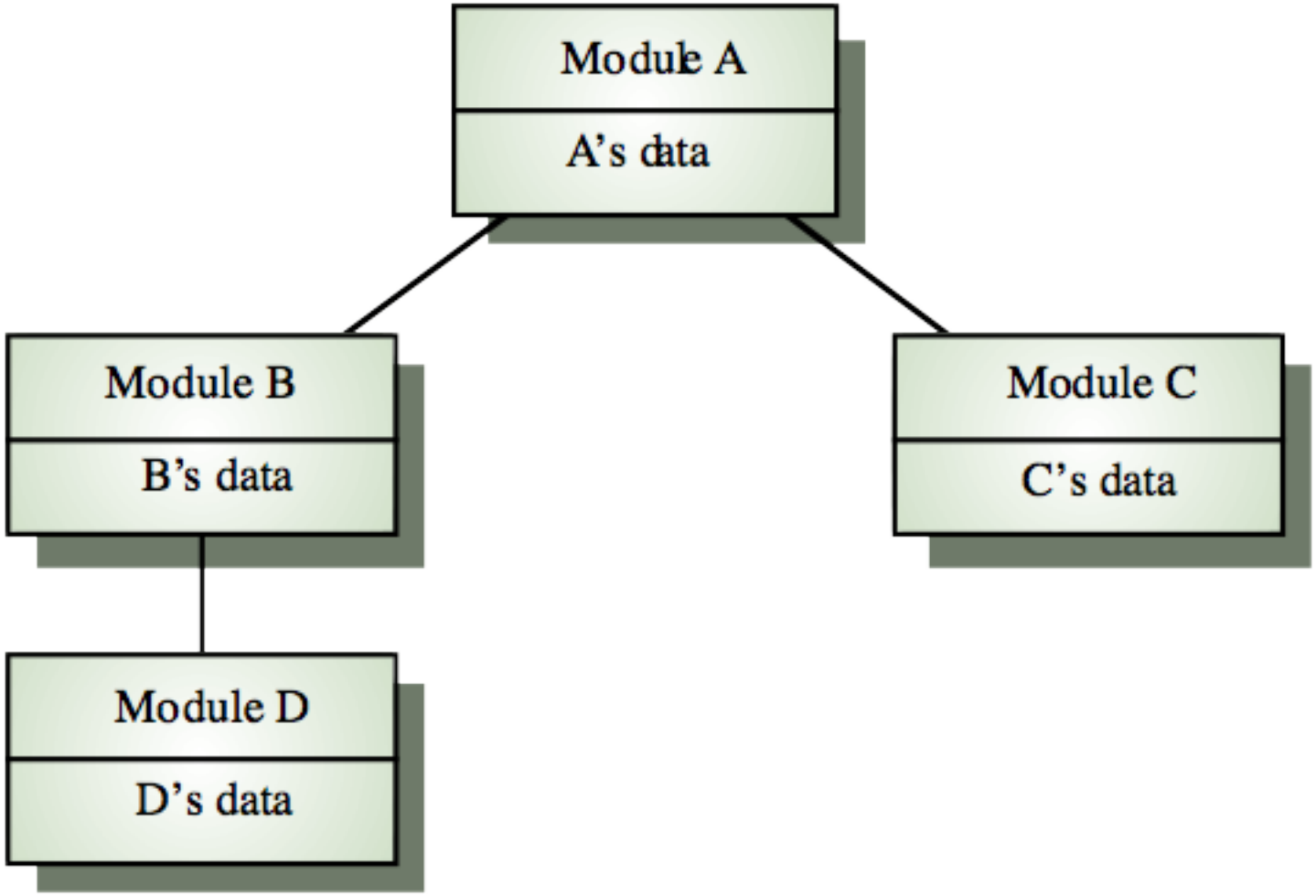
Welche Eigenschaft führt zu einer besseren Qualität?

- **Loose coupling** *verbessert Wartbarkeit* und Adaptabilität, da *Änderungen in einer Komponente sich weniger wahrscheinlich auf anderen Komponenten auswirkt.*

Tight Coupling



Loose Coupling



Exkurs: Cohesion / Kohäsion

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

- Cohesion ist schwach, wenn Elemente nur wegen ihrer Ähnlichkeit ihrer Funktionen zusammengefasst sind (z.B., **`java.lang.Math`**).
- Cohesion ist stark, wenn alle Teile für eine Funktionalität tatsächlich benötigt werden (z.B. **`java.lang.String`**).

Welche Eigenschaft führt zu einer besseren Qualität?

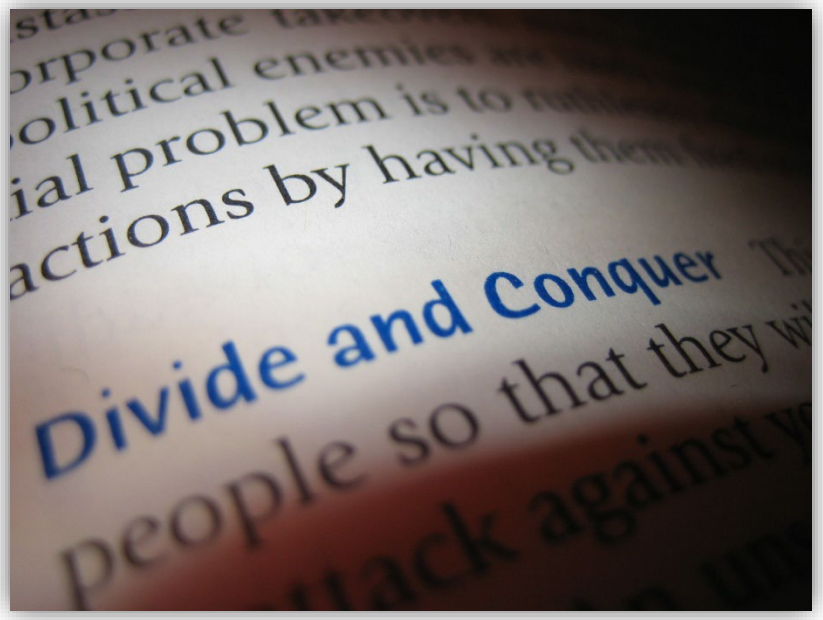
- Starke cohesion *verbessert Wartbarkeit* and Adaptivität durch *die Einschränkung des Ausmaßes von Änderungen* auf eine kleine Anzahl von Komponenten.

Fünf Kriterien für gutes Design

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

Fünf Kriterien: Modular Decomposability

- Problem kann in wenige kleinere, weniger komplexe Sub-Probleme zerlegt werden
- Sub-Probleme sind durch einfache Struktur verbunden
- Sub-Probleme sind unabhängig genug, dass sie einzeln bearbeitet werden können



Fünf Kriterien: Modular Decomposability

- Modular decomposability setzt voraus: Teilung von Arbeit möglich
- Beispiel: Top-Down Design
- Gegenbeispiel: Ein globales Initialisierungsmodul, eine große Main-Methode

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

Regel 1: Direct Mapping

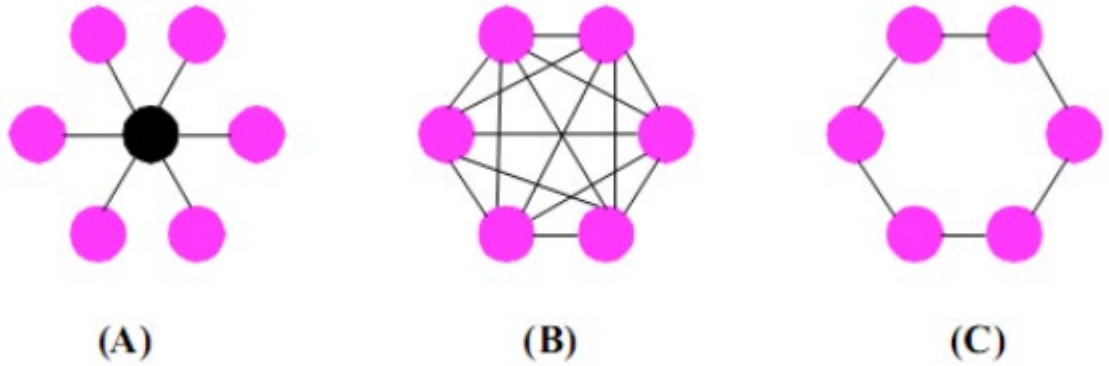
- Modulare Struktur des Softwaresystems sollte modularer Struktur des Modells der Problemdomäne entsprechen

- Folgt aus continuity und decomposability
- A.k.a. “low representational gap”[C. Larman]

Regel 2: Few Interfaces

- Jedes Modul sollte mit möglichst wenigen anderen Modulen kommunizieren
- Folgt aus continuity and protection
- Struktur mit möglichst wenigen Verbindungen

Types of module interconnection structures



Regel 3: 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

Regel 4: Explicit Interfaces

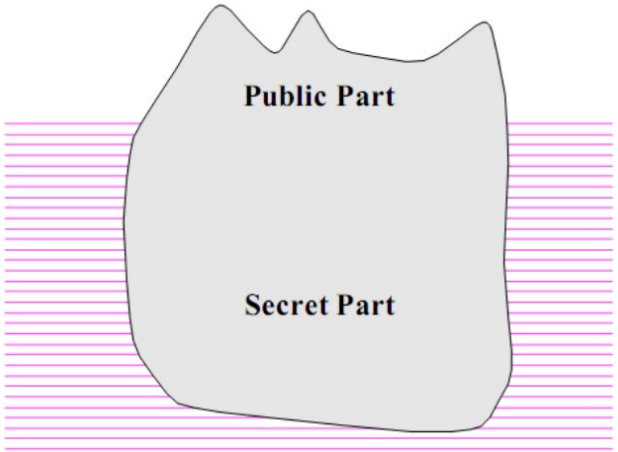
- Wenn zwei Module miteinander kommunizieren, muss das aus dem Interface von mindestens einem hervorgehen

- 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?

Regel 5: 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
- Impliziert durch continuity



SOLID-Prinzipien für gutes objekt-orientiertes Design

SOLID: Prinzipien für OO Design

Ziel: Software verständlicher, flexibler und wartbarer zu machen

Single-Responsibility Principle
Open-Closed Principle
Liskov Substitution Principle
Interface Segregation Principle
Dependency Inversion Principle

Prinzipien stellen Basis eines jeden OO-SW Designs dar

Single-Responsibility Principle

- Jedes Modul, Klasse, Funktion eines Programms soll Verantwortung über eine bestimmte Teilfunktionalität haben
- Es sollte nur diese *eine* Verantwortung haben (siehe Responsibility-Driven Design)

- Effekt:
  - Robust gegenüber zukünftigen Änderungen
  - Änderungen sind lokalisiert an einen Ort
  - Einfaches Verständnis über die Verantwortung, wenn limitiert auf eins

Open-Closed Principle

- Software Entitäten (Module, Klassen, Funktionen, etc.) sollten offen für Erweiterungen, aber geschlossen für Modifikationen sein
- Funktionalität kann erweitert werden ohne existierenden Code zu modifizieren

- Effekte:
  - Klasse kann z.B. durch Vererbung erweitert werden ohne vorhandenen Code zu modifizieren
  - Klasse kann von anderen benutzt werden, ohne sie verändern zu müssen (Information Hiding)

Liskov Substitution Principle

- Jedes Objekt S, welches ein Subtyp (Kindklasse) eines Typs T ist, sollte anstelle eines Objekts von Typ T benutzen werden können, ohne das sich das Verhalten des Programmes (Korrektheit, Performance, etc.) verändert
- Strong behavioral subtyping genannt

- Effekte:
  - Garantiert semantische Interoperabilität von Typen in einer Hierarchie

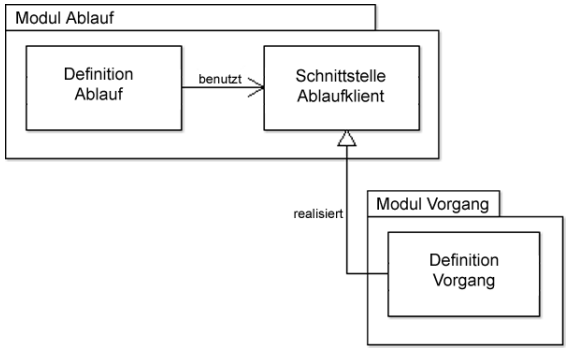
Interface Segregation Principle

- Kein Klient sollte nicht dazu gezwungen werden, abhängig von Methoden zu sein, die man nicht nutzt
- Große Interfaces in mehrere kleinere aufteilen, so dass Klienten nur tatsächlich benutzt Teile implementieren müssen

- Effekte:
  - Komponenten werden in kleinere, leichter zu wartende Systeme zerlegt
  - SW ist einfacher zu refaktorisieren durch kleinere Interfaces
  - Klassen müssen nur die für sie relevanten Methoden kennen
  - Führt zu höherer Kohäsion (siehe GRASP)

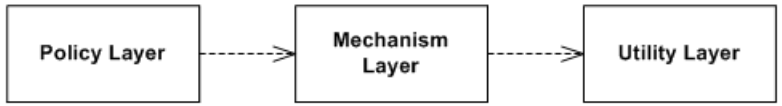
Dependency Inversion Principle (DIP)

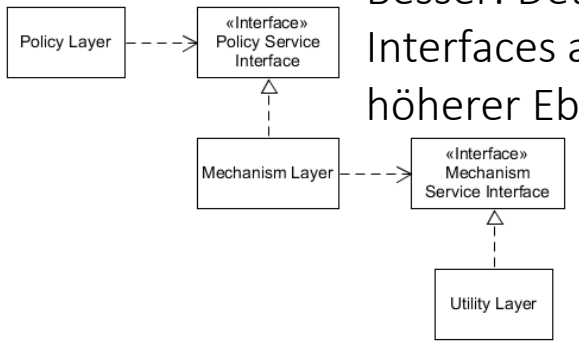
- Module höherer Ebenen sollten nicht von Modulen von unteren Ebenen abhängig sein, stattdessen nur von Abstraktionen
- Abstraktionen sollten nicht von Details abhängen, sondern anders herum



- Effekte:
  - Führt zu verringerte Kopplung von Komponenten
  - Verhindert zyklische Abhängigkeiten zwischen Komponenten

Höhere Ebene ist abhängig von Detailebene





Besser: Details sind durch Interfaces abgekoppelt von höherer Ebene

Beispiel: DIP

```
public class Lampe {
   private boolean leuchtet;

   public void anschalten() {
      leuchtet = true;
   }

   public void ausschalten() {
      leuchtet = false;
   }
}
```

Schalter steuert Lampe, daher höheres Modul, aber Schalter ist abhängig von Lampe; falsche Abhängigkeitsrichtung

```
public class Schalter {
   private Lampe lampe;
   private boolean gedrueckt;

   public Schalter(Lampe lampe) {
      this.lampe = lampe;
   }
   public void drueckeSchalter() {
      gedrueckt = !gedrueckt;
      if(gedrueckt) {
         lampe.anschalten();
      } else {
         lampe.ausschalten();
      }
   }
}
```

DIP folgend:

```
public interface SchalterClient {
   public void anschalten();
   public void ausschalten();
}
```

Entkopplung der Abhängigkeiten durch Interface

```
public class Lampe implements SchalterClient {
   private boolean leuchtet;

   public void anschalten() {
      leuchtet = true;
   }

   public void ausschalten() {
      leuchtet = false;
   }
}
```

Lampe wird nun abhängig von Abstraktion (Interface); Richtung low level to high level

Schalter kann nun beliebige Entitäten steuern

```
public class Schalter {
   private SchalterClient client;
   private boolean gedrueckt;

   public Schalter(SchalterClient client) {
      this.client = client;
   }

   public void drueckeSchalter() {
      gedrueckt = !gedrueckt;
      if(gedrueckt) {
         client.anschalten();
      } else {
         client.ausschalten();
      }
   }
}
```

GRASP – Pattern

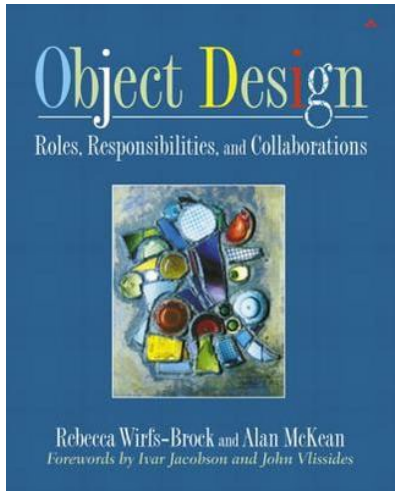
General Responsibility Assignment Software Patterns

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)

http://www.wirfs-brock.com/



Responsibilities (Recap)

- Bezogen darauf, wie sich Objekt verhalten soll
- Zwei Arten
  - Wissen
    - Wissen über private Daten
    - Wissen über Objekte, die sich auf einander beziehen
    - Wissen, was es ableiten oder berechnen kann
  - Tätigkeit
    - Etwas selbst tun, z.B. Objekt erstellen oder berechnen
    - Aktionen in anderen Objekten initiieren
    - Kontrolle und Koordination von Aktivitäten anderer Objekte

GRASP-Pattern

- Information Expert
- Creator
- Controller
- Low Coupling
- High Cohesion
- Indirection
- Polymorphism
- Pure Fabrication
- Protected Variations
- Pure Fabrication
- Protected Variations

GRASP-Prinzip: Information Expert

- Problem: Wer soll die Responsibilities bekommen?
- Lösung: Das Objekt, das die meisten Informationen (Daten) hat, diese Responsibilty zu erfüllen

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

GRASP-Prinzip: Creator

- Problem: Wer erstellt Instanzen eines Objekts?

- Lösung: Creator braucht erstelltes Objekt häufig in seinem Lebenszyklus
- Objekt B bekommt Verantwortung, Objekt A zu erstellen, wenn:
  - B A-Objekte aggregiert
  - B A-Objekte enthält
  - B Instanzen von A-Objekten loggt
  - B Initialisierungsdaten von A hat

GRASP-Prinzip: Controller

- Problem: Welches Objekt nach dem UI koordiniert die Business-Logik / Programmablauf?
- Lösung: Verantwortung bekommt ein Objekt mit folgenden Möglichkeiten:
  - Fassade Controller
    - Objekt repräsentiert das Gesamtsystem,
    - Ist ein Root-objekt,
    - Repräsentiert das Gerät, worauf die SW läuft
  - Use Case Controller
    - Alle Ereignisse eines Use Cases werden in dieser Klasse koordiniert

- Controller in MVC?
  - Nein, sollte nur delegieren und nicht koordinieren
  - Service im Modell sollte koordinieren

GRASP-Prinzip: Loose Coupling

- Problem: Wie kann ich die Auswirkungen von Änderungen und Abhängigkeiten reduzieren und Wiederverwendung erhöhen?

- Lösung: Verantwortung so zuteilen, dass Kopplung verringert wird
  - Siehe SOLID und Interfaces

GRASP-Prinzip: High Cohesion

- Problem: Wie können Objekte fokussiert, verständlich und wartbar gehalten werden?

- Lösung: Verantwortung so zuteilen, dass Kohäsion hoch ist
  - Nicht-verwandte Daten und Funktionen aus Klassen entfernen
  - Nur auf tatsächliche Verantwortung konzentrieren
  - 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: Indirection

- Problem: Wo Verantwortung zuweisen, um direkte Kopplung zweier Klassen zu verhindern?

- Lösung: Verantwortung auf eine Zwischenklasse zuweisen, so dass direkte Kopplung verhindert wird
  - Siehe Mediator Design Pattern
  - Reduziert jedoch Lesbarkeit und Verständlichkeit

GRASP-Prinzip: Polymorphism

- Problem: Wie kann ich Alternativen basierend auf Typen händeln?

- Lösung: Wenn Alternativen variieren durch den Typen, weise Verantwortlichkeiten abhängig von Typ zu
  - Siehe Strategy Design Pattern (initialisiere im Konstruktor ein Objekt vom Typ, dessen Verantwortlichkeit gerade benötigt wird)
  - Ermöglicht unterschiedliche Implementierungen für denselben Input

GRASP-Prinzip: Pure Fabrication

- Problem: Welches Objekt bekommt Verantwortlichkeiten, wenn ich dadurch die anderen Prinzipien (High Cohesion, Low Coupling) verletzen würde?

- Lösung: Weise ein kohäsive Menge an Verantwortlichkeiten zu einer künstlichen oder unterstützenden Klasse (z.B. Utility Class) zu, welche kein Domänenkonzept repräsentiert
  - Einführung von Utility- / Supportklassen

GRASP-Prinzip: Protected Variations

- Problem: Wie designe ich Klassen, Systeme und Subsystem so, dass Variationen und Änderungen keinen ungewünschten Effekt auf andere Elemente hat?

- Lösung: Identifiziere Stellen dieser Variationen und weise Verantwortlichkeiten zu, um ein stabiles Interface herum zu schaffen
  - Encapsulation durch Interfaces und Information Hiding
  - Modularity erhöhen
  - Design Patterns anwenden
  - Virtualisierung und Containerisierung für (Sub)Systeme
  - Ereignisbasierte Kommunikation und Architekturen

Software Qualität

Aufgabe

- Was ist Software Qualität?
- Wie würden Sie Software Qualität messen?

Software Qualität

- Korrekte Implementierung der Anforderungen
- Design-Qualität:
  - Erweiterbarkeit, Wartbarkeit, Verständlichkeit, Lesbarkeit, Wiederverwendbarkeit,…
  - Robustheit gegen Änderungen
  - Modularität, Low Coupling, High Cohesion
- Zuverlässigkeit, geringe Fehleranfälligkeit, Testbarkeit, Performance
- Usability, Effizienz, Erlernbarkeit

Software Qualität: Probleme

- Spannung zwischen Stakeholdern
  - Kunde: Effizienz, Usability
  - Entwickler: Wartbarkeit, Wiederverwendbarkeit, Verständlichkeit

- Wie misst man eigentlich Software Qualität?
  - „You can't control what you can't measure"*
  - Software-Metriken? -> direkt messbar
  - Empirische Untersuchungen? -> indirekt messbar

*Tom DeMarco, 1986

Software Metriken

- Lines of Code (LoC)
- Bugs per line of code
- Comment density
- Cyclomatic complexity (measures the number of linearly independent paths through a program's source code)
- Halstead complexity measures (derive software complexity from numbers of (distinct) operands and operators)
- Program execution time
- Test Coverage
- Number of classes and interfaces
- Abstractness = ratio of abstract classes to total number of classes
- …

Software Metriken werden selten benutzt

- Einige Firmen erstellen Messprogramme, noch weniger haben Erfolg damit
- Firmen, die Metriken benutzen, machen das oft nur, um bestimmten Qualitätsstandards zu genügen
  - Bei Interesse: N. E. Fenton, "Software Metrics: Successes, Failures & New Directions”, 1999.

- Hier kommt wieder der Unterschied zur Ingenieursdisziplin zum Tragen
  - Ingenieure sind erfolgreich, weil sie Qualität gut messen können
  - Geht eben nicht bei Softwareprodukten

Software Metriken sind selten sinnvoll

- Formal definierte Metriken sind zwar objektiv, aber was bedeuten sie?
  - Was sagt eine zyklomatische Komplexität von 12 über die Qualität des Quelltextes aus?
  - Gar nichts

- Oft unklar, ob Metrik mit irgendeinem sinnvollen Qualitätskriterium zusammenhängt
  - Wäre so, als würde man Intelligenz mit Hirnmasse oder Kopfumfang messen wollen

Software Qualität

- Wenn Metriken nicht sinnvoll sind, wie dann Software-Qualität messen?
- Analyse für jedes Softwareprojekt
  - Design-Qualität, Erweiterbarkeit
  - Vergleich zu anderen Designs, die sich als sinnvoll erwiesen haben (z.B. Design Patterns)
  - Code smells und Antipatterns suchen
  - Umfangreiche Tests und Code Reviews
  - (Formale) Verifikation
  - Nutzerstudien

Qualitätsmerkmale

- Beziehen sich auf Produkt und Prozess
  - Produkt: an Kunden geliefert, externe Qualität
  - Prozess: produziert das Produkt, interne Qualität

- Qualitätsprozess ist notwendige Bedingung für ein Qualitätsprodukt

Qualitätsmerkmale

- Korrektheit (siehe Testen-Vorlesung)
- Zuverlässigkeit (Wahrscheinlichkeit, dass System erwartungsgemäß läuft)
- Robustheit („gutes“ Verhalten in unbekannten Systemzuständen)
- Effizienz (geringer Ressourcenverbrauch)
- Usability (Maß der Bedienbarkeit, Nutzerkontrolle/-freiheit, Ähnlichkeit zur Welt)
- Wartbarkeit (wie einfach es ist das System zu ändern / anzupassen nach Release)
- Verifizierbarkeit (Einfachheit der Überprüfung gewünschter Eigenschaften)
- Verständlichkeit (Einfachheit des Verstehens für interne und externe Nutzer)
- Produktivität (wie produktiv sind Entwickler für Teile des Systems?)
- Pünktlichkeit (pünktliche Auslieferung)
- Transparenz (Einsehbarkeit des Projektzustands)

Qualität auf Implementierungsebene: Code Smells

- Jedes Symptom im Quelltext, das auf größeres Problem (z.B. Antipattern) hinweist
- Nicht jeder Code Smell ist schlechter Code; bei jedem Code Smell überprüfen, ob er wirklich behoben werden muss
- Typische Code Smells:
  - Duplizierte Code
  - Lange Methode/Klasse
  - High coupling/low cohesion
  - Schrotkugeln (shotgun surgery)

Qualität auf Implementierungsebene: Antipatterns

- Siehe Vorlesung Design Patterns
- Beispiele:
  - The Blob (Gottklasse)
  - Action at a distance: Unerwartete Interaktion zwischen eigentlich getrennten Modulen
  - Reihenfolge: Wenn Methoden in einer Klasse in bestimmter Reihenfolge aufgerufen werden müssen
  - Zirkuläre Abhängigkeit: Unnötige Abhängigkeiten zwischen Modulen
- Mehr Beispiele: http://c2.com/cgi/wiki?AntiPatternsCatalog

Was Sie mitgenommen haben sollten:

- Anwendung:
  - [Modell/Quelltext]
  - Bewerten Sie anhand der 5 Kriterien
  - Bewerten Sie anhand der 5 Regeln
  - Bewerten Sie anhand der vorgestellten Elemente des GRASP-Patterns
- Wissen:
  - Was ist modular decomposability? Erklären Sie an einem Beispiel
  - [analog die restlichen Krtiterien, Regeln, und Prinzipien]

Literatur

- Bertrand Meyer, Object-Oriented Software Construction, Prentice Hall, 1997 [Chapter 3, 4]
- *Software Engineering*, I. Sommerville, 7th Edn., 2004.
- *Objects, Components and Frameworks with UML*, D. D'Souza, A. Wills, Addison-Wesley, 1999
- *Pattern-Oriented Software Architecture — A System of Patterns*, F. Buschmann, et al., John Wiley, 1996
- *Software Architecture: Perspectives on an Emerging Discipline*, M. Shaw, D. Garlan, Prentice-Hall, 1996