Universität Zürich[UZH]

Institut für Informatik

Martin Glinz Thomas Fritz

Software Engineering

Kapitel 5

Entwurf von Software

5.1 Grundlagen und Prinzipien

5.2 Architekturentwurf

5.3 Modularer Entwurf

5.4 Entwurfsmuster (design patterns)

5.5 Vertragsorientierter Entwurf

Motivation

- Kleinsoftware – kein systematischer Entwurf notwendig
- «Richtige» Software – braucht systematischen, strukturierten Aufbau
- Konzeptfehler sind teuer
- ⇨ Entwurf zwingend
  - Lösung verstehen und strukturieren
  - Entwicklungsaufwand verteilen auf mehrere Personen
  - Lösung einbetten in vorhandene Systemlandschaft
  - Lösung geographisch verteilen
  - Grundstein für leicht pflegbares System

Der Entwurfsprozess

Erster Schritt der Lösung

Architekturentwurf: Komponenten, Schnittstellen, Interaktionen

Detailentwurf: Algorithmen und interne Datenstrukturen

- Anforderungsspezifikation als Vorgabe notwendig
- Zeitliche und hierarchische Verzahnung von Anforderungsspezifikation und Architekturentwurf möglich
- Ergebnisse immer in separaten Dokumenten

Definitionen und Begriffe

Architektur (architecture) – Die Organisationsstruktur eines Systems oder einer Komponente. Legt die wesentlichen Komponenten der Lönsung und die Interaktionen zwischen diesen Komponenten fest.

Entwurf (design) – 1. Der Prozess des Definierens von Architektur, Komponenten, Schnittstellen und anderen Charakteristika eines Systems oder einer Komponente. 2. Das Ergebnis des Prozesses gemäß 1.

Architekturentwurf (architectural design) – Erstellung und Dokumentation der Architektur eines Systems.

Lösungskonzept (Software-Architektur, Systemarchitektur, software architecture, system architecture) – das Dokument, welches das Konzept der Lösung, d.h. die Architektur des zu erstellenden Systems dokumentiert.

Entwurfsqualität

Merkmale guter Entwürfe:

- Effektiv: Erfüllt die Vorgaben und löst das Problem des Auftraggebers
- Wirtschaftlich: resultierende Software kostengünstig in Entwicklung, Betrieb und Pflege
- Softwaretechnisch gut: Leicht verständlich, fehlerarm, robust, zuverlässig änderungsfreundlich

Entwurfsprinzipien: Struktur und Abstraktion

Struktur: Gliedern der Lösung in Komponenten und Interaktionen

Abstraktion: Verstehen durch systematisches Vergröbern/Verfeinern

Kapselnde Dekomposition als zentrales Abstraktionsmittel:

Ein System so in Teile zerlegen, dass

- jeder Teil mit möglichst wenig Kenntnissen des Ganzen und der übrigen Teile verstanden werden kann
- das Ganze ohne Detailkenntnisse über die Teile verstanden werden kann

DAS Abstraktionsmittel für das Verstehen komplexer Systeme

Entwurfsprinzipien: Modularität

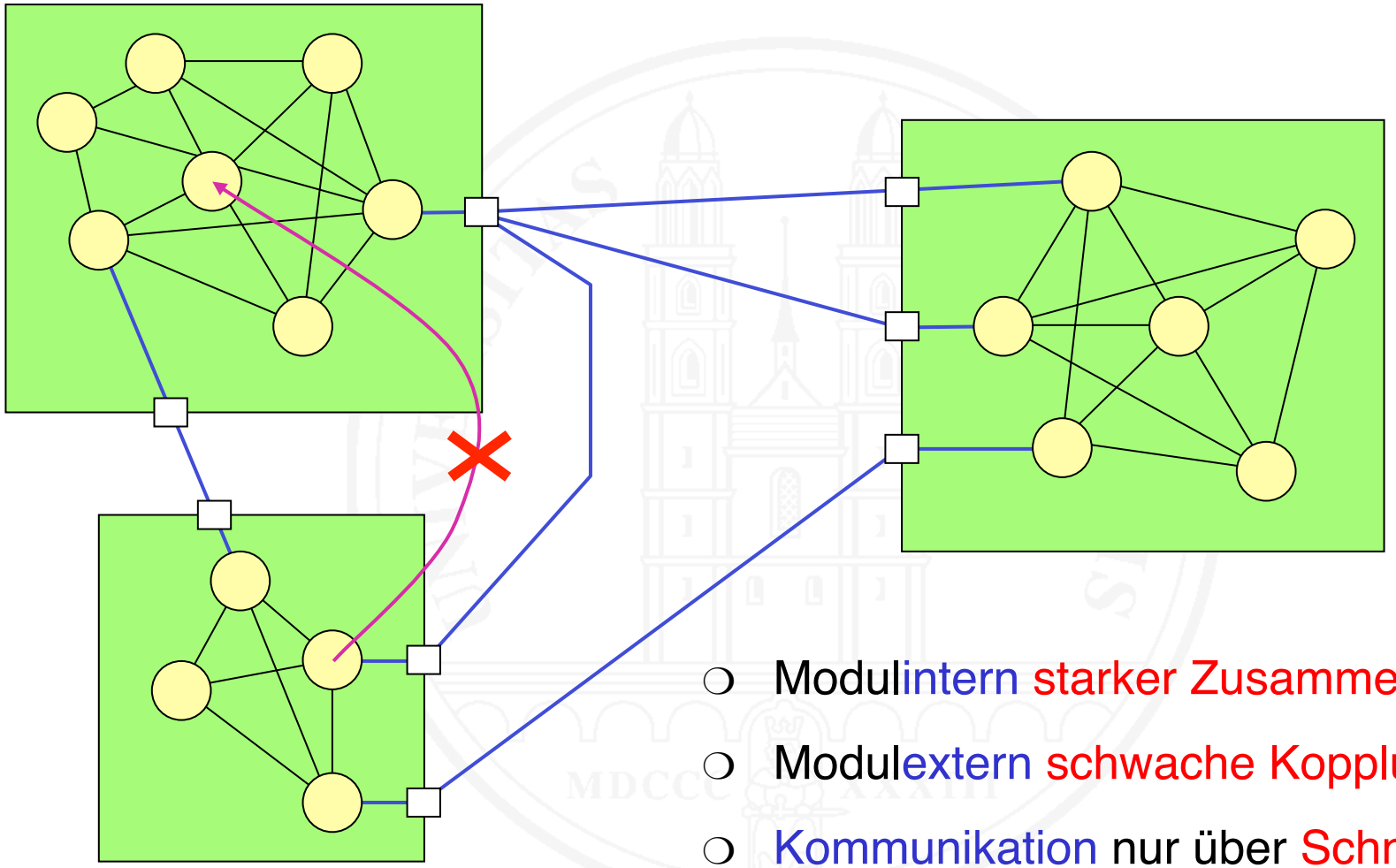
Modularisierung ist eine Hauptaufgabe des Software-Entwurfs

Modul (module) – Ein benannter, klar abgegrenzter Teil eines Systems.

Gute Module haben folgende Eigenschaften:

- In sich geschlossene Einheit
- Ohne Kenntnisse über inneren Aufbau verwendbar
- Kommunikation mit Umgebung ausschließlich über Schnittstellen
- Im Inneren rückwirkungsfrei änderbar
- Korrektheit ohne Kenntnis der Einbettung ins Gesamtsystem prüfbar
- Erlaubt Komposition und Dekomposition

Das Prinzip einer modularen Struktur



- Modulintern starker Zusammenhang
- Modulextern schwache Kopplung
- Kommunikation nur über Schnittstellen
- Modulinneres von außen nicht sichtbar

Mini-Übung 5.1

Eine Anlage füllt eine Flüssigkeit in Flaschen ab. Sie besteht aus einem Tank, zwei Förderbändern für das Zuführen und Wegführen der Flaschen und einer Abfüllstation mit Waage.

Die Software für die Steuerung dieser Anlage sei wie folgt modularisiert:

- Tank (Steuerung des Tank-Einlassventils, Feststellen des Füllstands)
- Abfüllung (Steuerung des Tank-Abfüllventils, Ablesen der Waage, Zuführen/Wegführen von Flaschen zur Abfüllstation)
- Band (Steuerung der Förderbänder)
- Init (Initialisierung der gesamten Steuerung)

Beurteilen Sie die Qualität dieser Modularisierung. Wo sehen Sie Probleme?

Geheimnisprinzip (Information hiding)

Geheimnisprinzip (information hiding) – Kriterium zur Gliederung eines Gebildes in Komponenten, so dass

- jede Komponente eine Leistung (oder eine Gruppe logisch eng zusammenhängender Leistungen) vollständig erbringt,
- außerhalb der Komponente nur bekannt ist, was die Komponente leistet,
- nach außen verborgen wird, wie sie ihre Leistungen erbringt.

[Parnas 1972]

- ➪ Fundamentales Prinzip zur Beherrschung komplexer Systeme
- ➪ Auch im täglichen Leben fortwährend benutzt
- ➪ Liefert gute Modularisierungen

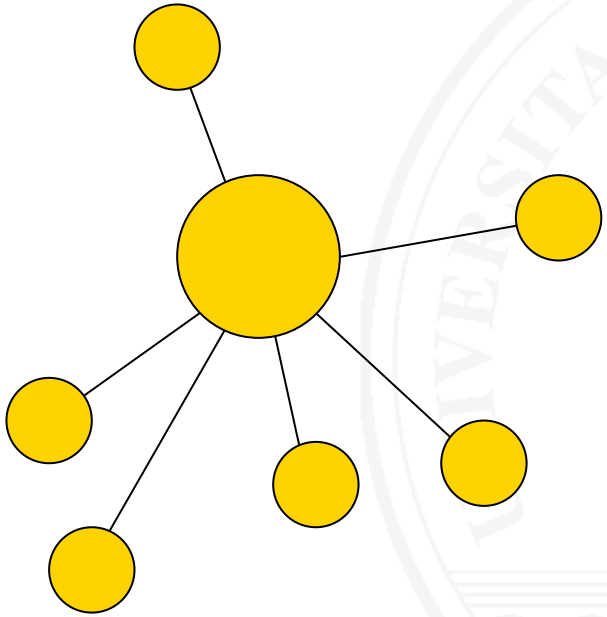
Entwurfsprinzipien: Schnittstellen und Verträge

Schnittstelle (interface) – Verbindungsglied zwischen einem Modul und der Außenwelt zwecks Austausch von Information

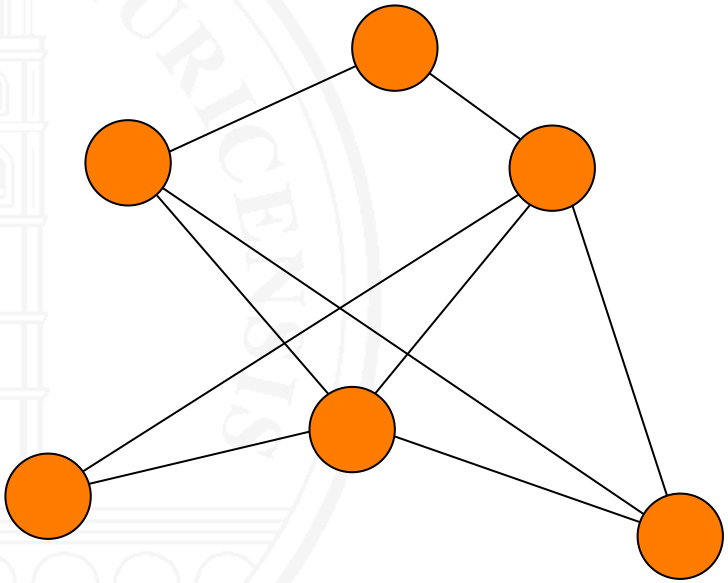
- Leistungen, die ein Modul zur Nutzung anbietet
- Bedarf eines Moduls an Leistungen Dritter
- Beschreibung in Form eines Vertrags (contract) zwischen Anbieter und Verwender: Rechte und Pflichten
- Details siehe Kapitel 5.5: Vertragsorientierter Entwurf

Entwurfsprinzipien: Nebenläufigkeit (1)

Problem: Gleichzeitige, koordinierte Bearbeitung mehrerer Aufgaben



Mehrere Verwender nutzen parallel oder zeitlich verzahnt gemeinsame Dienstleistungen



Unabhängig arbeitende Agenten kooperieren zwecks Erbringung von Leistungen

Entwurfsprinzipien: Nebenläufigkeit (2)

- Nebenläufigkeit wird durch Prozesse realisiert

Prozess (process) – Eine durch ein Programm gegebene Folge von Aktionen, die sich in Bearbeitung befindet.

Nebenläufigkeit (concurrency) – Die parallele oder zeitlich verzahnte Bearbeitung mehrerer Aufgaben.

- Entwurfsprobleme
  - Welcher Prozess bearbeitet welche Aufgaben?
  - Wann und wie tauschen Prozesse welche Information aus?
  - Wann und wie synchronisieren Prozesse ihren Arbeitsfortschritt?

Entwurfsprinzipien: Nutzung von Vorhandenem

- Wo möglich: nicht neu entwickeln
  - ➪ Wiederverwenden, Beschaffen
- Zu untersuchen:
  - Vollständige Beschaffung (Standardsoftware, konfigurierbare Bausteine)
  - Beschaffung abgeschlossener Teilsysteme (zum Beispiel Datenbanksystem)
  - Realisierung durch Einbettung in einen existierenden Software-Rahmen (framework)
  - Nutzung einzelner Komponenten (Programm- /Klassenbibliotheken)
  - Wiederverwendung von Architektur- und Entwurfsideen: Entwurfsmuster (design patterns)

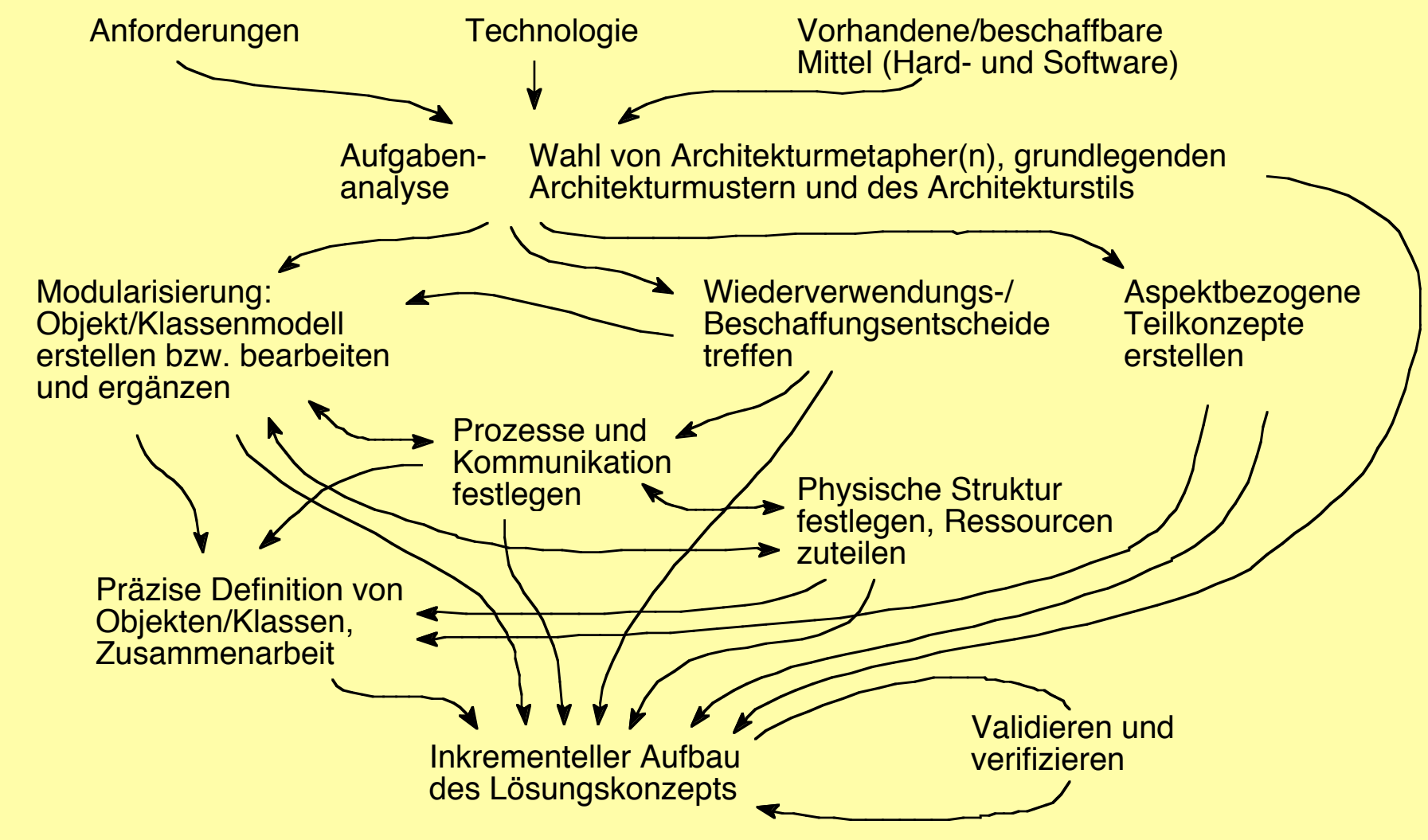
Entwurfsprinzipien: Ästhetik

- Wahl und konsequente Verwendung eines Architekturstils
- Klar erkennbaren, gestalteten Strukturen
- Wenig Gewordenes
- Kein Gewursteltes
- Der Struktur des Problems angemessene Struktur der Architektur
- Wahl der einfachsten und klarsten Lösung aus der Menge der möglichen Lösungen.

Aufgaben des Architekturentwurfs

- **Aufgabe analysieren**
  - Anforderungen verstehen
  - Vorhandene bzw. beschaffbare Technologien und Mittel analysieren
- **Architektur modellieren und dokumentieren**
  - Grundlegende Systemarchitektur festlegen: Muster, Metaphern ➪ Stil
  - Modularisieren
  - Nebenläufige Lösungen in Prozesse gliedern
  - Wiederverwendungs- und Beschaffungsentscheide treffen
  - Ressourcen zuordnen
  - Aspektbezogene Teilkonzepte für Querschnittsaufgaben erstellen
  - Lösungskonzept (als Dokument) erstellen und prüfen

Vorgehen und Zusammenhänge



Variantenbehandlung

- Ganzen Lösungsraum betrachten
- Lösungsvarianten
  - erkennen
  - verfolgen
  - abwägen
    - was kostet es, das Optimum zu verfehlen?
    - was kostet die Untersuchung?
  - entscheiden
  - dokumentieren
- Kosten
  - Nicht nur Entwicklungskosten der Variante!
  - auch Betriebskosten, Pflegekosten, Folgekosten anderswo
- Beschaffungsvariante immer betrachten

5.1 Grundlagen und Prinzipien

5.2 Architekturentwurf

5.3 Modularer Entwurf

5.4 Entwurfsmuster (design patterns)

5.5 Vertragsorientierter Entwurf

Siehe separate Folien von Thomas Fritz: Software Design

5.1 Grundlagen und Prinzipien

5.2 Architekturentwurf

5.3 Modularer Entwurf

5.4 Entwurfsmuster (design patterns)

5.5 Vertragsorientierter Entwurf

Siehe separate Folien von Thomas Fritz: Modular Design

5.1 Grundlagen und Prinzipien

5.2 Architekturentwurf

5.3 Modularer Entwurf

5.4 Entwurfsmuster (design patterns)

5.5 Vertragsorientierter Entwurf

Siehe separate Folien von Thomas Fritz: Design Patterns

5.1 Grundlagen und Prinzipien

5.2 Architekturentwurf

5.3 Modularer Entwurf

5.4 Entwurfsmuster (design patterns)

5.5 Vertragsorientierter Entwurf

Schnittstellendefinition mit Verträgen

- Schnittstelle ist Vertrag zwischen Modul und Modulverwender
- Beschreibung des Vertrags mit Zusicherungen (assertions)
- Vier Arten von Zusicherungen
  - Voraussetzungen (preconditions, requirement, Schlüsselworte: pre, require)
  - Ergebniszusicherungen (postconditions, Schlüsselworte: post, ensure)
  - Invarianten (invariants)
  - Verpflichtungen (obligations)
- "Design by Contract" (Meyer 1988, 1992)

Vertragserfüllung

Vertragserfüllung bedeutet:

- Verwender muss
  - Voraussetzungen erfüllen
  - Übernommene Verpflichtungen einhalten
- Modul muss
  - Ergebniszusicherungen erfüllen
  - Invarianten garantieren
  - aber unter der Annahme der Vertragstreue des Modulverwenders!

Schnittstellendefinition mit Verträgen – Eigenschaften

- Leichter lesbar als algebraische Spezifikation
- Präziser und eindeutiger als einfacher Kommentartext
- Voraussetzungen und Resultate klar formulierbar
- Benötigt in der Regel Zustandsvariablen
  - Gefahr implementierungsabhängiger Spezifikationen
  - ⇨ Nur solche Zustandsvariablen verwenden, welche eine Entsprechung im Problembereich / der Anwendungsdomäne haben

Sprache für die Formulierung von Verträgen

- Natürliche Sprache ist nur beschränkt geeignet
  - mehrdeutig
  - unpräzise
- Rein formale Sprache häufig zu wenig verständlich
- Besser: Teilformale, deklarative Sprache, basierend auf
  - Prädikaten – soweit möglich
  - Fallunterscheidungen
  - Natürlicher Sprache – wo nötig
  - (möglichst wenig) Zustandsvariablen

Beispiel: Ein einfaches Konto

```
interface EinfachesKonto
{
    // public EinfachesKonto ();
    // PRE –
    // POST int saldo = 0
    public void Einzahlen (int betrag);
    // PRE betrag ≥ 0
    // POST saldo = saldo@PRE + betrag
    public void Abheben (int betrag);
    // PRE betrag ≥ 0
    // POST saldo = saldo@PRE - betrag
    public int Kontostand ();
    // PRE –
    // POST result = saldo and saldo = saldo@PRE
}
```

Syntaktisch Kommentar, da es in Java-Schnittstellen keine Konstruktoren gibt

Erforderlich, damit der Anfangszustand von saldo spezifizierbar ist

Wert einer Zustands-variable bei Prüfung der Voraussetzung

Mathematische Gleich-heit, *keine* Zuweisung

Voraussetzungen und Ergebniszusicherungen

- Spezifizieren die Wirkung einer Operation / Methode einer Schnittstelle
- Voraussetzungen
  - Müssen zum Zeitpunkt des Aufrufs durch den Aufrufer erfüllt sein
  - Werden von der Implementierung der Schnittstelle nicht geprüft
- Ergebniszusicherungen
  - Beschreiben die Effekte der Operation
  - Müssen von jeder Implementierung der Schnittstelle erfüllt werden
  - Aber unter der Annahme, dass die Voraussetzungen erfüllt sind

Mini-Übung 5.2

Benötigt wird ein dreistelliger Dezimalzähler, der von 0 bis 999 hochzählt und dann wieder bei Null beginnt. Es werden drei Operationen benötigt: Reset, Increment und Display

Entwerfen Sie eine Schnittstelle Zähler mit diesen drei Operationen mit vertragsorientiertem Entwurf

Invarianten

- Objekte haben Eigenschaften, die nicht verändert werden dürfen
  - Beispiel: ein Quadrat hat vier gleiche Seiten und ist rechtwinklig
- Wenn eine Methode eine Zustandsvariable nicht verändert, so muss dies explizit zugesichert werden
  - Beispiel: POST result = saldo **and** saldo = saldo@PRE
- Operationen / Methoden hängen zusammen
  - Beispiel: (konto.Einzahlen(n)).Abheben(n) = konto
- Invarianten lösen diese Probleme
  - Beschreiben Eigenschaften der Schnittstelle, die unter allen Operationen invariant sind
  - Entlasten die Ergebniszusicherungen der Operationen
  - Spezifizieren Zusammenhänge zwischen Operationen

Beispiel einer Invariante

```
interface EinfachesKonto
{
// INVARIANT with e = Summe aller mit Einzahlen eingezahlten Beträge,
//                a = Summe  aller mit Abheben abgehobenen Beträge
//                holds saldo = e - a
...
```

- Garantiert, dass der Saldo nur durch Einzahlen und Abheben verändert wird
- Ermöglicht, die Bedingung saldo = saldo@PRE in der Ergebnis-zusicherung von Kontostand wegzulassen

➪ Eine Invariante bezieht sich immer auf die ganze Schnittstelle, nicht auf eine einzelne Operation / Methode

Verpflichtungen

- „Wer A sagt, muss auch B sagen“ (Volksweisheit)
- Mit dem Aufruf einer Operation / Methode übernimmt der Aufrufer häufig Pflichten, zum Beispiel
  - Aufräum- oder Terminierungsoperationen aufzurufen
  - Ein Protokoll von Aufrufen einzuhalten
- Verpflichtungen
  - Spezifizieren Pflichten, die der Aufrufer mit dem Aufruf einer Operationen übernimmt
  - Brauchen in der Darstellung oft temporale Logik

Beispiel: Einfaches Sperrprotokoll

```
// Wer eine Sperre setzt, muss sie später auch wieder freigeben
interface EinfacheSperre
{
  //public EinfacheSperre ();
  //PRE –
  //POST boolean gesperrt = false
  public boolean Sperren ();
  //PRE –
  //POST if gesperrt@PRE then result = false
  //              else gesperrt and result = true endif
  //OBLIGATION sometimes Freigeben()
  public void Freigeben ();
  //PRE –
  //POST gesperrt = false
}
```

Mini-Übung 5.3

Beurteilen Sie die Qualität der folgenden beiden Entwurfsfragmente:

double Sqrt (**double** x);
// PRE –
// POST **if** $x \geq 0$ **then for all** $\varepsilon < 10^{-12}$ **and** $\varepsilon < 10^{-6}x$ $|(Sqrt(x))^2 - x| < \varepsilon$
// **else** Fehlermeldung "x ist negativ" **endif**

double Kalibrieren (**double** a, **double** b);
// Die Korrekturfaktoren a und b werden vom Benutzer eingegeben
// PRE $a \neq 0$
// POST **result** = (this/a)*Fkorr(b)

Was, wann und wo prüfen?

- Vertragsorientierter Entwurf: Voraussetzungen werden nicht geprüft

  Metapher: Vertragstreue Partner
- Vorteil: klare Verantwortlichkeiten, schlanker Code
- Problem: Woher weiß ich, ob mein Partner vertragstreu ist?

  ➪ Gegebenenfalls Zusicherungen dynamisch prüfen
- Defensives Programmieren: Prüfe, was immer du kannst

  Metapher: “Designed for the unexpected”
- Vorteil: Mehr Sicherheit
- Problem: redundante Mehrfachprüfungen
  - blähen den Code auf
  - behindern die Lesbarkeit des Codes

Gefährlich: Implizite Voraussetzungen

- Eine Operation / Methode macht faktisch Voraussetzungen
- Die Voraussetzungen werden weder geprüft noch sind sie dokumentiert
- Der Aufrufer muss entweder die Implementierung kennen oder durch Experimente herausfinden
- Standardvorgehen bei C-Bibliotheken
- Bevorzugte Angriffsstelle für Hacker (Pufferüberlauf-Angriffe)

Prüfregeln für vertragsorientierten Entwurf

- Voraussetzen immer dann, wenn dem Aufrufer die Erfüllung der Voraussetzungen zugemutet werden kann
- Prüfen immer dann, wenn mit Falscheingaben gerechnet werden muss (zum Beispiel bei Benutzereingaben)
- Prüfen nur, wenn eine sinnvolle Behandlung von Fehlern möglich ist
- Bewusste Entwurfsentscheidungen treffen ➔ Nicht dem Geschmack der Programmierer überlassen

Prüfen der Ergebnisse

- Voraussetzungen werden nicht geprüft
- Leistungserbringer prüft Ergebnisse und erzeugt bei falschen oder unzulässigen Ergebnissen Fehlerbedingungen
- Leistungserbringer behandelt den Fehler nicht, sondern gibt nur Fehlerbedingungen an Aufrufer zurück
- Aufrufer interpretiert Fehlerbedingungen und handelt danach
- Nachteil: Umständlich, erschwert Lesbarkeit des Codes des Aufrufers
- Vorteil: Der Aufrufer kennt den Kontext besser:
  bessere Fehlermeldung möglich
- Aber: wenn schon, dann besser mit Ausnahmebehandlung lösen (siehe unten)

Beispiel für Ergebnisprüfung

```
public abstract class EinfachesKonto
{
    public boolean ok;   // Falsch nach Aufrufen mit ungültigem Ergebnis
    ...
    public abstract void Einzahlen (int betrag);
    // PRE –
    // POST   if betrag ≥ 0 then (saldo = saldo@PRE + betrag) and ok
    //              else (saldo = saldo@PRE) and not ok endif
    ...
}
```

Für den Aufrufer bedeutet das Konstruktionen der Art:

```
...
k.Einzahlen (betrag);
if (!k.ok) ...        // Fehler behandeln
```

Ausnahmebehandlung

- Voraussetzungen werden nicht geprüft
- Leistungserbringer prüft Ergebnisse und erzeugt bei falschen oder unzulässigen Ergebnissen Ausnahmen (exceptions)
- Laufzeitsystem übergibt Steuerung an Ausnahmebehandler des Aufrufers
- Falls kein Behandler vorhanden, wird Ausnahme in der Aufrufhierarchie hochgereicht
- Behandler behandelt Ausnahmen
  - Fehlermeldungen
  - Abbruch oder geordnete Rückkehr in den Programmablauf
- Nachteil: Nicht in allen Programmiersprachen verfügbar
- Vorteil: Code für Normal- und Ausnahmesituationen sauber trennbar
  Keine Variablen zur Weitergabe von Prüfergebnissen nötig

Ausnahmebehandlung, Beispiel

```
public void Einzahlen (int betrag) throws BetragNegativ;

    // PRE –
    // POST  if betrag ≥ 0 then (saldo = saldo@PRE + betrag)
    //       else (saldo = saldo@PRE) and exception BetragNegativ
    //       endif
```

Mini-Übung 5.4

In Mini-Übung 5.3 haben wir diskutiert, warum die Verträge der folgenden beiden Methoden schlecht bzw. gefährlich sind. Entwerfen Sie bessere Verträge.

double Sqrt (**double** x);
// PRE –
// POST **if** $x \geq 0$ **then for all** $\varepsilon < 10^{-12}$ **and** $\varepsilon < 10^{-6}x$ $|(Sqrt(x))^2 - x| < \varepsilon$
// **else** Fehlermeldung "x ist negativ" **endif**

double Kalibrieren (**double** a, **double** b);
// Die Korrekturfaktoren a und b werden vom Benutzer eingegeben
// PRE $a \neq 0$
// POST **result** = (this/a)*Fkorr(b)

Dynamische Prüfung von Zusicherungen

- Geeignet formulierte Zusicherungen in Programmen sind maschinell prüfbar
- Mächtiges Mittel zur dynamischen Prüfung von Programmen
  - Als eigenständiges Prüfverfahren
  - Zur Lokalisierung von Defekten bei der Behebung von beim Testen festgestellten Fehlern (Debugging)
- In manchen Programmiersprachen (z. B. Eiffel) direkt programmierbar
- Sonst mit Hilfskonstrukten zu programmieren, z. B. in Java bis Version 1.3 mit Ausnahmen
- Java bietet ab Version 1.4 einen primitiven, auf Ausnahmebehandlung basierenden Zusicherungsmechanismus als Bestandteil der Sprache

Dynamische Prüfung in Eiffel

- Hochzähloperation für einen Zähler mit oberer Grenze

```
-- upper: Obere Grenze
-- count: aktueller Zählerwert

add(n: INTEGER) is
  require
    (n > 0) and (count + n <= upper)
  do
    count := count + n
  ensure
    count = old count + n
  end -- add
```

Dynamische Prüfung in Java – 1

```
class BoundedCounter {
private int count, lower, upper;
…                                   // add constructor method here
public void add(int n) {
   // assert precondition
   assert (n > 0) && (count + n <= upper) :
       "precondition violated:  n: " + n + " count: " + count
       + " upper: " + upper;

   // Inner class that saves state to verify postcondition
   class DataCopy {
       private int countCopy;
       DataCopy(int value) {countCopy = value; }
       int countAtPRE() { return countCopy; }
   }
   DataCopy copy = new DataCopy(count);
   // Creates an object that saves the value of count@PRE
```

Dynamische Prüfung in Java – 2

```
   // productive code
   count = count + n;

   // assert postcondition
   assert count == copy.countAtPRE() + n :
       "postcondition violated:  count: " + count +
       " count@PRE: "  + copy.countAtPRE() + " n: " + n;
 }
…

// assert invariant
assert count >= lower && count <= upper :
       "invariat violated:  count: " + count + " lower: " +
        lower + " upper: " + upper;
}
```

Dynamische Prüfung in Java – 3

```
//Main program for testing
public static void main (String[] args) {
   BoundedCounter bc = new BoundedCounter(0, 0, 100);
   bc.add(25);
   bc.add(50);
   bc.add(33);
}
```

Ausführungsprotokoll:

```
$ javac -source 1.4 BounderCounter.java
$ java BoundedCounter
$ java -ea BoundedCounter
Exception in thread "main" java.lang.AssertionError: precondition violated:
    n: 33 count: 75 upper: 100
       at BoundedCounter.add(BounderCounter.java:20)
       at BoundedCounter.main(BounderCounter.java:53)
```

Verträge für Bedarfsschnittstellen

Vertrag ist invers zu Verträgen für Angebotsschnittstellen

- Für jede benötigte Operation sind zu spezifizieren
  - Die Voraussetzungen, welche die benötigte externe Operation höchstens machen darf
  - Die Ergebniszusicherungen, welche die benötigte Operation mindestens machen muss
- Notwendige Invarianten: Eigenschaften, welche jede Implementierung der benötigten Komponente unverändert lassen muss

Literatur

Siehe Literaturverweise im Kapitel 6 des Skripts. Weitere Literatur:

S. Dustdar, H. Gall, M. Hauswirth (2003). *Software-Architekturen für Verteilte Systeme: Prinzipien, Bausteine und Standardarchitekturen für moderne Software*. Berlin: Springer.

C. Szyperski (1998). *Component Software – Beyond Object-Oriented Programming*. Harlow, England etc.: Addison-Wesley.