KV Software Engineering
Prof. Dr. Martin Glinz

Kapitel 3

Systematisches Entwerfen

Universität Zürich
Institut für Informatik

© 2004, 2005 by Martin Glinz. Alle Rechte vorbehalten. Reproduktion, Speicherung und Wiedergabe jeglicher Art sind ausschließlich für den persönlichen, nicht kommerziellen Gebrauch gestattet, wobei bei auszugsweiser Wiedergabe die Quelle und das Copyright zu nennen sind. Die Verwendung für Unterrichtszwecke ist nicht gestattet.

3.1 Modulkonzepte und Schnittstellen

3.2 Vertragsorientierter Entwurf

3.3 Zusammenarbeit

3.4 Spezialisierung von Schnittstellen; Schnittstellenvererbung

3.1.1 Module und Dekomposition

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

Eigenschaften einer guten Modularisierung:

- Kapselnde Dekomposition
- Verwendbarkeit
- Geschlossenheit und Lokalität

Beispiele für Module (auf verschiedenen Stufen)

- Prozedur/Methode, abstrakter Datentyp, Klasse, Komponente

Kapselnde Dekomposition

Ein System so in Teile (Module) 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

© 2004 by Martin Glinz

Verwendbarkeit

- Ein Modul bildet eine Einheit bezüglich Verwendung bzw. Wiederverwendung
- Die Verwendung eines Moduls erfordert keine Kenntnisse über seinen inneren Aufbau
- Ein Modul beschreibt sein Leistungsangebot für Dritte in Form einer Schnittstelle
- Module können durch Dritte zur Komposition von Systemen verwendet werden.

Geschlossenheit und Lokalität

- Jedes Modul ist eine in sich geschlossene Einheit
- Änderungen im Inneren eines Moduls, welche seine Schnittstelle unverändert lassen, haben keine Rückwirkungen auf das übrige System
- Die Korrektheit eines Moduls ist ohne Kenntnis seiner Einbettung ins Gesamtsystem prüfbar.

© 2004 by Martin Glinz

Mini-Übung 3.1

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?

3.1.2 Modularisierungsarten

- Strukturorientierte Modularisierung
  - Modularisierungskriterien: Namensraum, Übersetzungseinheit
  - Beispiele: Subroutinen in FORTRAN, Programme in COBOL
  - Güte der Modularisierung: zufällig

- Funktionsorientierte Modularisierung
  - Modularisierungskriterium: Jeder Modul berechnet eine Funktion
  - Beispiel: Structured Design (Stevens, Myers, Constantine 1974, Page-Jones 1988)
  - Güte der Modularisierung: gut für rein funktionale, zustandsfreie Probleme, sonst schlecht

© 2004 by Martin Glinz

Modularisierungsarten – 2

- Datenorientierte Modularisierung
  - Modularisierungskriterium: Modul fasst eine Datenstruktur und alle darauf möglichen Operationen zusammen
  - Beispiel: Abstrakter Datentyp (ADT)
  - Güte der Modularisierung: gut
  - Problem: ADT sind streng disjunkt; Gemeinsamkeiten im Leistungsangebot verschiedener ADT können nicht zusammengefasst werden

© 2004 by Martin Glinz

Modularisierungsarten – 3

- Objektorientierte Modularisierung
  - Modularisierungskriterium: Modul repräsentiert Objekt des Problembereichs oder benötigtes Informatik-Element
  - Beispiel: Klassen im objektorientierten Entwurf
  - Güte der Modularisierung: gut, wenn Klassen als ADT konzipiert werden. Mäßig bis schlecht, wenn Klassen offen konzipiert werden
  - Vorteil: Extrem flexibel und ausdrucksmächtig

Modularisierungsarten – 4

- Komponentenorientierte Modularisierung
  - Modularisierungskriterium: Jeder Modul ist eine stark gekapselte Menge zusammengehöriger Elemente, die eine gemeinsame Aufgabe lösen und als Einheit von Dritten verwendet werden
  - Beispiel: Werkzeugsatz zur Bearbeitung von Verbunddokumenten
  - Güte der Modularisierung: sehr gut
  - Vorteil: Als in sich geschlossene Einheit verwendbar
  - Problem: Weniger flexibel als Klassen, Verwendung in unbekanntem Kontext stellt sehr hohe Ansprüche an die Qualität der Schnittstellen wie der Implementierung

© 2004 by Martin Glinz

3.1.3 Schnittstelle und Implementierung

- Verwendbarkeit
  - Die Schnittstelle eines Moduls muss nach außen sichtbar und dokumentiert sein
- Geschlossenheit
  - Der Modul ist ausschließlich über die Schnittstelle zugänglich
  - Die Implementierung des Moduls ist nach außen verborgen

➪ Idealerweise sind Schnittstelle und Implementierung getrennt

- Prozeduren/Methoden: keine oder schwache Trennung
- Klassen in objektorientierten Programmiersprachen: dito

© 2004 by Martin Glinz

Trennung von Schnittstelle und Implementierung

- Beispiel Modula-2:
- DEFINITON MODULE (Schnittstelle)
- IMPLEMENTATION MODULE (Implementierung)
  - Syntaktisch getrennt
  - Aber noch eng gekoppelt: Zu jeder Schnittstelle genau eine Implementierung gleichen Namens

- Beispiel Java:
- **interface** abc ... (Schnittstelle)
- **class** xyz **implements** abc ... (Implementierung)
  - Schwach gekoppelt: mehrere Implementierungen zu einer Schnittstelle möglich
  - Eine Klasse kann gleichzeitig mehrere Schnittstellen implementieren

Spezifikation der Leistungen eines Moduls

Notwendiges Minimum: Namen/Signaturen der verwendbaren Operationen, Typen, Konstanten und ggf. Variablen, zum Beispiel (Wirth 1985):

```
DEFINITION MODULE InOut;
 ...
 CONST EOL = 15C;
 VAR Done: BOOLEAN;
 ...
 PROCEDURE ReadString (VAR s: ARRAY OF CHAR);
 PROCEDURE ReadInt (VAR x: INTEGER);
 ...
```

Besser: Zusätzlicher, erläuternder Kommentar:

```
PROCEDURE ReadString (VAR s: ARRAY OF CHAR);
(* Reads a text that is being typed on the keyboard into s *)
```

Spezifikation der Leistungen eines Moduls – 2

Noch besser: Rigorose, teilformale oder formale Spezifikation der Schnittstelle

```
PROCEDURE ReadString (VAR s: ARRAY OF CHAR);
(* PRE: -
   POST: Let str be the string typed on the keyboard,
          terminated by a character <= " " (blank)
         and l = HIGH(s) + 1
         IF length (str) <= l
           THEN s = str (excluding the terminating character)
           ELSE s contains the first l characters of str;
                The remaining characters are ignored: they
                are neither read nor displayed
          END IF.
*)
```

Algebraische Spezifikation einer Schnittstelle

Beispiel: Formale algebraische Spezifikation der Schnittstelle für ein einfaches Konto in Java

```
interface EinfachesKonto
{
    public void Einzahlen (int betrag);
    public void Abheben (int betrag);
    public int Kontostand ();

    // Axiome:
    // ∀ k ∈ EinfachesKonto, b ∈ int
    // (1)   new().Kontostand() = 0
    // (2)   b ≥ 0 →k.Einzahlen(b).Abheben(b) = k
    // (3)   b ≥ 0 →k.Einzahlen(b).Kontostand() = k.Kontostand() + b
    // (4)   b ≥ 0 →k.Abheben(b).Kontostand() = k.Kontostand() - b
}
```

Signaturen: Syntax

Axiome: Semantik

© 2004 by Martin Glinz

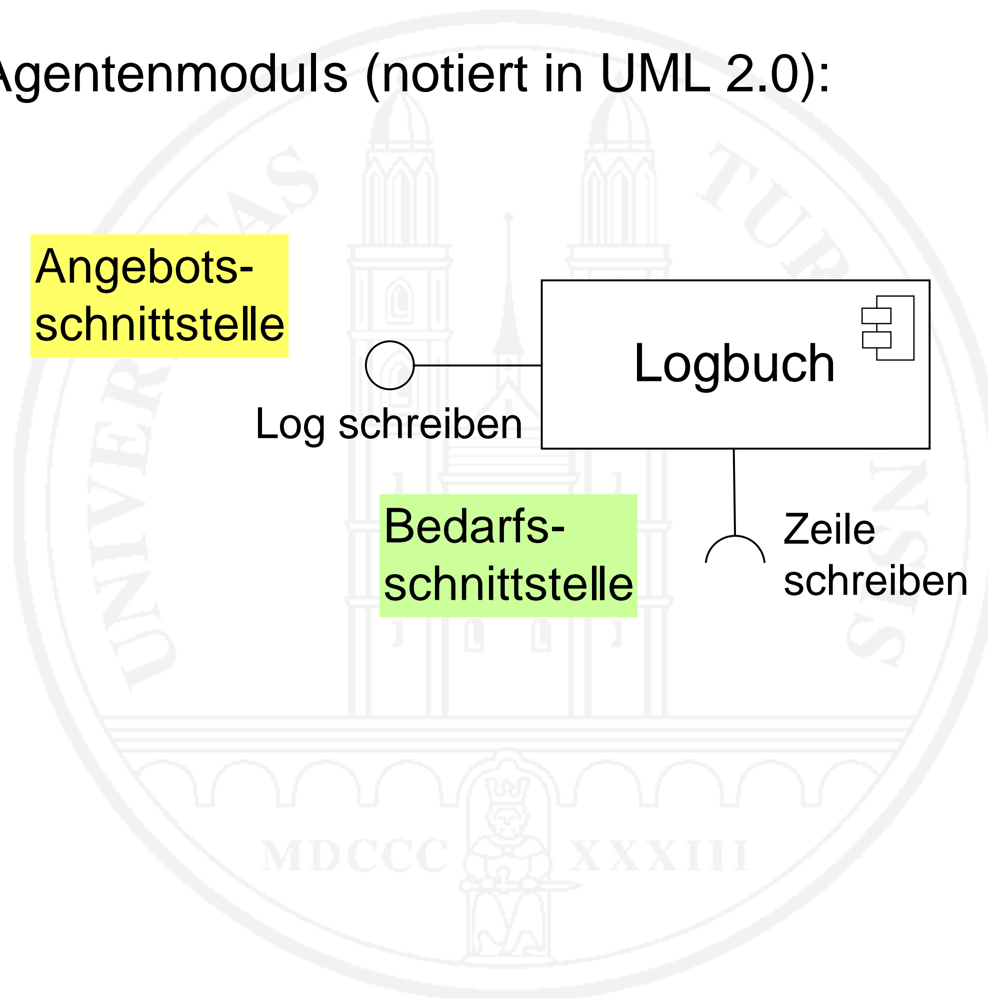
3.1.4 Angebots- und Bedarfsschnittstellen

Zwei Arten von Modulen:

- Dienstleistungsmodul
  - Stellt Leistungen für Dritte bereit
  - Leistungen in einer Angebotsschnittstelle definiert
  - Jede Implementierung der Komponente erbringt die angebotenen Leistungen vollständig selbst
  - Ausnahme: nutzt gegebenenfalls Leistungen des Betriebssystems
- Agentenmodul
  - Stellt Leistungen für Dritte bereit
  - Benötigt zur Erbringung dieser Leistungen die Leistungen von Drittkomponenten
  - Angebots- und Bedarfsschnittstellen erforderlich

Angebots- und Bedarfsschnittstellen – 2

Beispiel eines Agentenmoduls (notiert in UML 2.0):



Merkpunkte Modulkonzepte

- Eine gute Modularisierung
  - kapselt Information
  - bildet Einheiten der Wiederverwendung, die ohne Kenntnis ihres inneren Aufbaus verwendbar sind
  - bildet in sich geschlossene Module und lokalisiert Effekte in den Modulen
- Die Verwendbarkeit solcher Module bedingt die Definition von Schnittstellen
- Das Leistungsangebot eines Moduls
  - muss in seiner Schnittstelle definiert sein
  - muss ohne Kenntnis des inneren Aufbaus verstehbar sein
- Agentenmodule haben Angebots- und Bedarfsschnittstellen

 © 2004 by Martin Glinz

3.1 Modulkonzepte und Schnittstellen

3.2 Vertragsorientierter Entwurf

3.3 Zusammenarbeit

3.4 Spezialisierung von Schnittstellen; Schnittstellenvererbung

3.2.1 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)

© 2004 by Martin Glinz

Vertragserfüllung

Vertragserfüllung bedeutet:

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

© 2004 by Martin Glinz

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

© 2004 by Martin Glinz

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 Zustandsvariable bei Prüfung der Voraussetzung

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 3.2

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, und zwar

a) mit vertragsorientiertem Entwurf

b) mit algebraischer Spezifikation

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

© 2004 by Martin Glinz

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
}
```


© 2004, 2005 by Martin Glinz

Mini-Übung 3.3

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)

3.2.2 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

© 2004 by Martin Glinz

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

© 2004 by Martin Glinz

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 3.4

Verträge der folgenden beiden Methoden schlecht bzw. gefährlich sind.
Entwerfen Sie gute Verträge für diese Methoden.

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 1.4 bietet neu einen primitiven, auf Ausnahmebehandlung basierenden Zusicherungsmechanismus als Bestandteil der Sprache

Dynamische Prüfung in Eiffel

- Hochzählоperation 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
```


© 2004 by Martin Glinz

Dynamische Prüfung in Java 1.4 – 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 1.4 – 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 1.4 – 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)
```

3.2.3 Spezifikation von 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

© 2004 by Martin Glinz

Merkpunkte Vertragsorientierter Entwurf

- Vertrag zwischen Anbieter und Verwender
- Bei Angebotsschnittstellen ist der Modul der Anbieter
- Vertragselemente (bei Angebotsschnittstellen):
  - Voraussetzungen (durch Verwender zu erfüllen)
  - Ergebniszusicherungen (durch Modul zu garantieren unter der Annahme der Vertragstreue des Verwenders)
  - Invarianten (Eigenschaften, die unverändert bleiben müssen)
  - Verpflichtungen (die bei Verwendung übernommen werden)
- Wo und wie geprüft wird, ist eine wesentliche Entwurfsentscheidung
- Formale Verträge sind dynamisch prüfbar
- Bedarfsschnittstellen werden invers zu Angebotsschnittstellen definiert

3.1 Modulkonzepte und Schnittstellen

3.2 Vertragsorientierter Entwurf

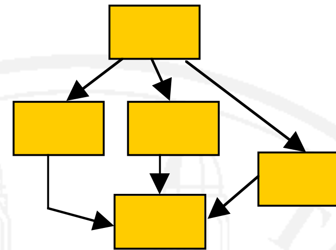
3.3 Zusammenarbeit

3.4 Spezialisierung von Schnittstellen; Schnittstellenvererbung

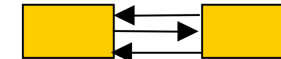
MDCCC XXXIII
UNIVERSITAS TURICENSIS

3.3.1 Formen der Zusammenarbeit

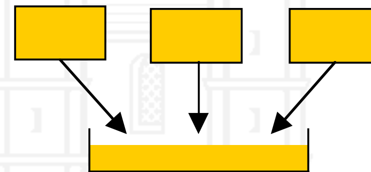
- Leistungserbringung



- Informationsaustausch



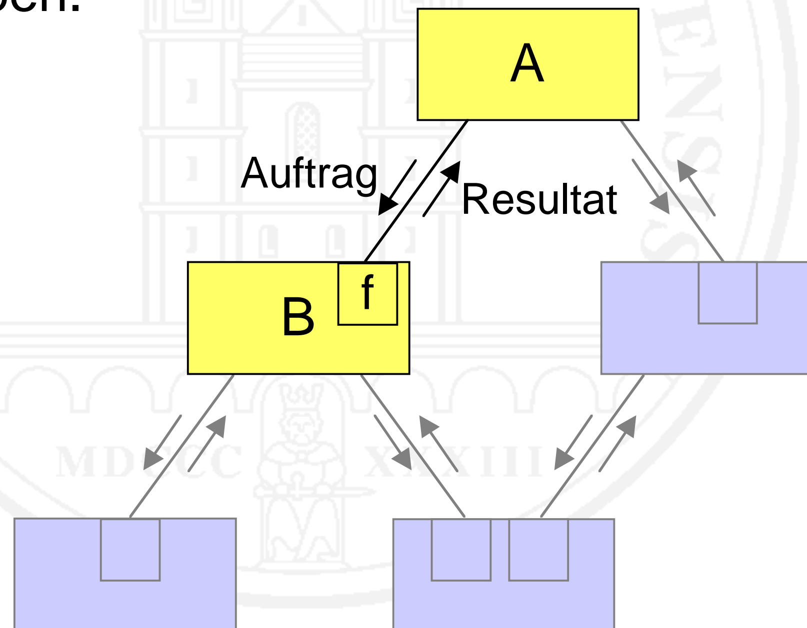
- Informationsteilhabe



- Die Art der Zusammenarbeit definiert wesentlich den Entwurfsstil
- Die Zusammenarbeit muss dokumentiert werden

Leistungserbringung – 1

- Motiv: Delegieren von Aufgaben
- Situation 1
  - A will eine benötigte Funktion f durch B ausführen lassen.
  - Mit Ausnahme des Funktionswerts soll der Systemzustand unverändert bleiben.

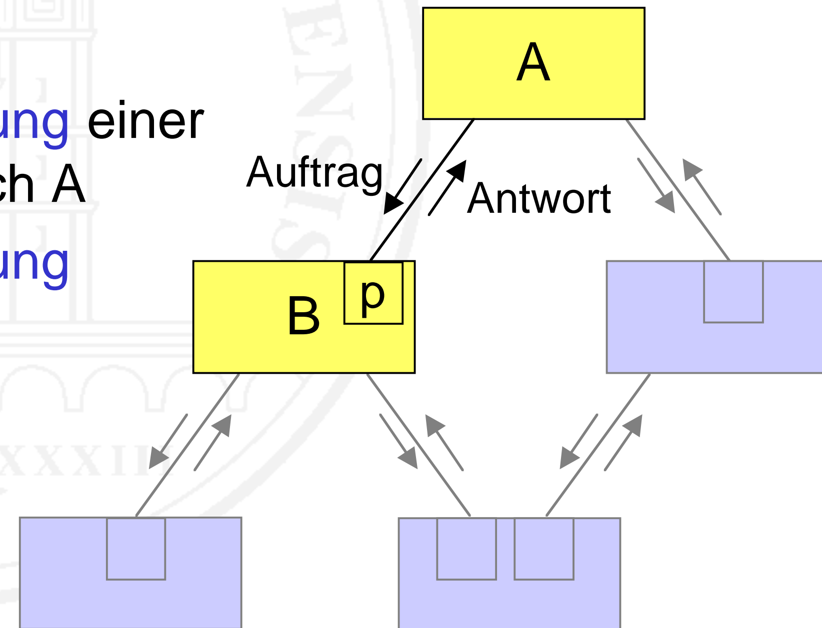


Leistungserbringung – 2

- Mittel
  - Statisch gebundener Aufruf einer Funktionsprozedur oder Methode f in B durch A
  - Dynamisch gebundene Anwendung einer Methode f (mit Rückgabewert) auf das Objekt B durch A
  - Dynamisch gebundene Anwendung einer Methode f aus einer Oberklasse von B auf das Objekt B durch A : super.f
- Bemerkungen
  - Ausführung von f darf den Systemzustand nicht verändern mit Ausnahme des Funktionswerts
    - ➪ Die Verwendung einer Funktion ist nebenwirkungsfrei
  - Als Parameter und Resultat werden in der Regel Daten oder Objekte übergeben

Leistungserbringung – 3

- Situation 2
  A will eine benötigte Operation durch B ausführen lassen. Die Ausführung kann (oder soll) den Systemzustand verändern.
- Mittel
  - Statisch gebundener Aufruf einer Prozedur oder Methode p in B durch A
  - Dynamisch gebundene Anwendung einer Methode p auf das Objekt B durch A
  - Dynamisch gebundene Anwendung einer Methode p aus einer Oberklasse von B auf das Objekt B durch A : super.f



 © 2004 by Martin Glinz

Leistungserbringung – 4

- Bemerkungen
  - Direkt veränderbar sind:
    - Ausgabeparameter von p
    - Zustand des Moduls, der p enthält, bzw. des Objekts, auf das p angewendet wird.
  - Indirekt veränderbar sind
    - Alle Zustände von Elementen, die von Operationen veränderbar sind, an die p (direkt oder transitiv) Arbeit delegiert.
  - Beliebige Nebenwirkungen möglich
  - Als Parameter können Daten, Operationen und Objekte übergeben werden
  - Rückruf / Delegation durch Übergabe von Operationen und Objekten möglich

Informationsaustausch

- Motiv: Organisation der Zusammenarbeit zwischen Komponenten nach dem Prinzip
  - der Wertschöpfungskette oder der Fließbandarbeit (Bringprinzip)
  - einer Kette von Einkäufern (Holprinzip)
  - von Lieferverträgen (Abonnementsprinzip)
- Information: Daten, Operationen oder Objekte

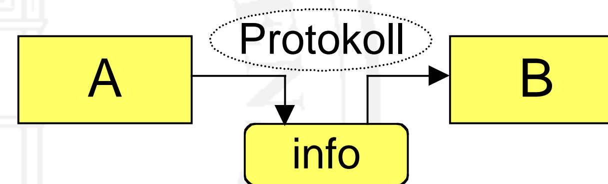
Mini-Übung 3.5

Nach welchem Prinzip arbeiten Pipes in UNIX?

Informationsaustausch: Bringprinzip – 1

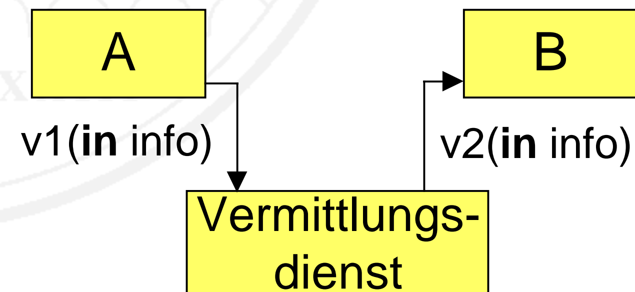
- Situation 1
  A will Informationen an B weiterreichen (Bringprinzip)
- Mittel
  - Aufruf mit Parameterübergabe

  - Zugriff auf globale Variablen



  - Direktmanipulation von Attributen

  - Verwendung eines Vermittlungsdienstes



© 2004 by Martin Glinz

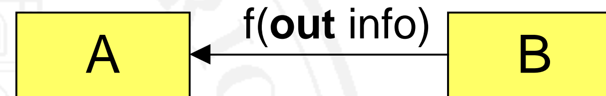
Informationsaustausch: Bringprinzip – 2

- Bemerkungen
  - Übergabe mit Wertparameter: nebenwirkungsfrei, schwache Kopplung
  - Übergabe von Operationen und Objekten: mächtiger und flexibler
    Aber: Nebenwirkungen und Rückwirkungen auf A möglich, stärkere Kopplung
  - Globale (oder teilglobale) Variablen: fast immer Nebenwirkungen, Synchronisation erforderlich, starke Kopplung
  - Direktmanipulation: sehr starke Kopplung ➪ vermeiden
  - Vermittlungsdienst: entkoppelt A und B, ermöglicht geografische Verteilung
    Aber: Funktionen und Objekte nur eingeschränkt übertragbar

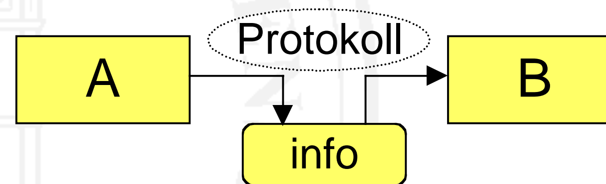
 © 2004 by Martin Glinz

Informationsaustausch: Holprinzip – 1

- Situation 2
  A will Informationen von B erhalten (Holprinzip)
- Mittel
  - Aufruf mit Parameterübergabe

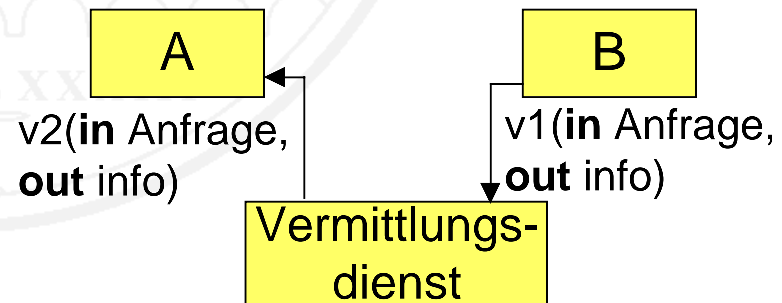


  - Zugriff auf globale Variablen



  - Direktmanipulation von Attributen

  - Verwendung eines Vermittlungsdienstes

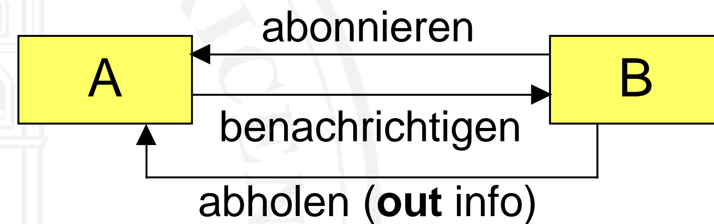


Informationsaustausch: Holprinzip – 2

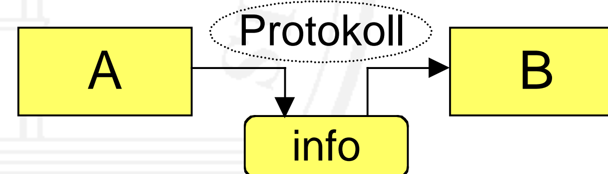
- Bemerkungen
  - Übernahme als Referenzparameter: nebenwirkungsfrei und schwach koppelnd, wenn Daten nicht verändert werden
    Sonst: massive Nebenwirkungen möglich
  - Übernahme von Operationen und Objekten: mächtiger und flexibler
    Aber: Nebenwirkungen und Rückwirkungen auf A möglich, stärkere Kopplung
  - Globale (oder teilglobale) Variablen: fast immer Nebenwirkungen, Synchronisation erforderlich, starke Kopplung
  - Direktmanipulation: starke Kopplung ➪ nur verwenden, wenn Änderungen in der Struktur der gelesenen Daten wenig wahrscheinlich
  - Vermittlungsdienst: entkoppelt A und B, ermöglicht geografische Verteilung
    Aber: Funktionen und Objekte nur eingeschränkt übertragbar

Informationsaustausch: Abonnementsprinzip – 1

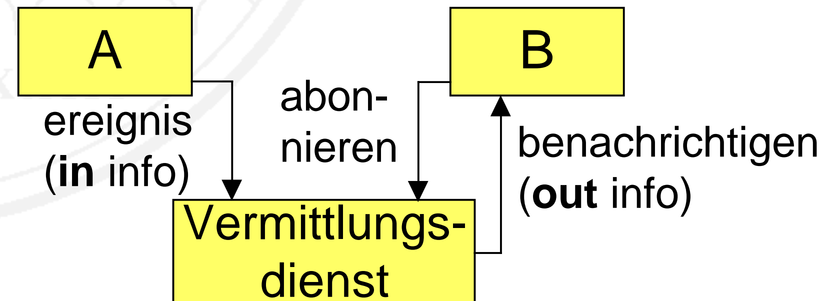
- Situation 3
  - B abonniert Informationen bei A
  - A benachrichtigt B, worauf B abholt (Abonnementsprinzip)
- Mittel
  - Abonnieren-Benachrichtigen-Holen (Beobachtermuster)



  - Zugriff auf globale Variablen



  - Verwendung eines Vermittlungsdienstes



Informationsaustausch: Abonnementsprinzip – 2

- Bemerkungen
  - Dient zur Trennung eng kooperierender Aufgaben in separate Module (Entkopplung)
  - Kopplung zwischen A und B wird von stark auf mittel reduziert.
  - Verwendung globaler oder teilglobaler Variablen zur Realisierung eines Abonnementsprinzips ist aufwendig und fehlerträchtig
    ➪ vermeiden
  - Verwendung eines Vermittlungsdienstes entkoppelt A und B, ermöglicht geografische Verteilung
    Aber: Funktionen und Objekte nur eingeschränkt übertragbar

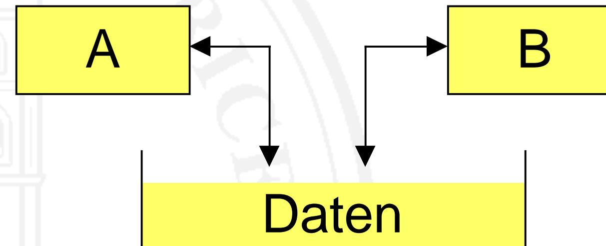
Informationsteilhabe

- Motiv: Komponenten sind gleichberechtigte Teilhaber an einer Menge von Information
  - Offene, direkte Teilhabe: gemeinsame Speicher
  - Gekapselte, direkte Teilhabe: Datenabstraktionen
  - Teilhabe über einen gemeinsamen Datenverwalter: Informationsdepot

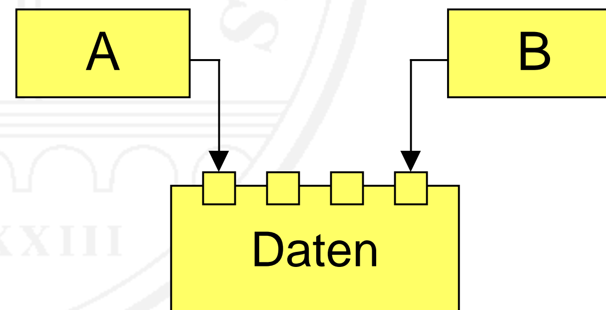
© 2004 by Martin Glinz

Informationsteilhabe: gemeinsamer Speicher – 1

- Situation 1
  A und B nutzen einen gemeinsamen Speicherbereich
- Mittel:
  - Direktzugriff auf gemeinsamen Speicher



  - Gemeinsam genutzte Datenabstraktion



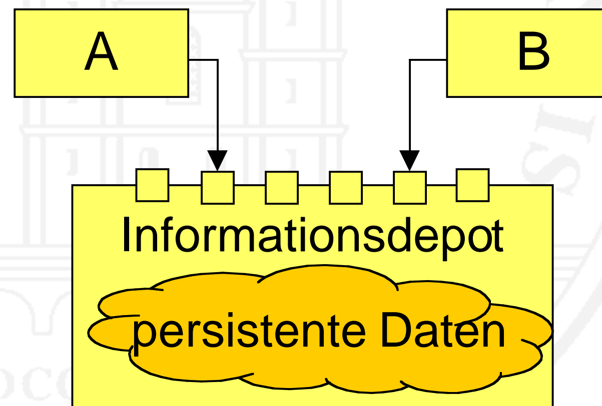
Informationsteilhabe: gemeinsamer Speicher – 2

- Bemerkungen
  - Direktzugriff ist einfach und schnell
    Aber: erfordert Sichtbarkeit der Datenstrukturen und explizite Synchronisation ➪ starke Kopplung
  - Gemeinsam genutzte Datenabstraktion verbirgt Datenstrukturen und Synchronisation

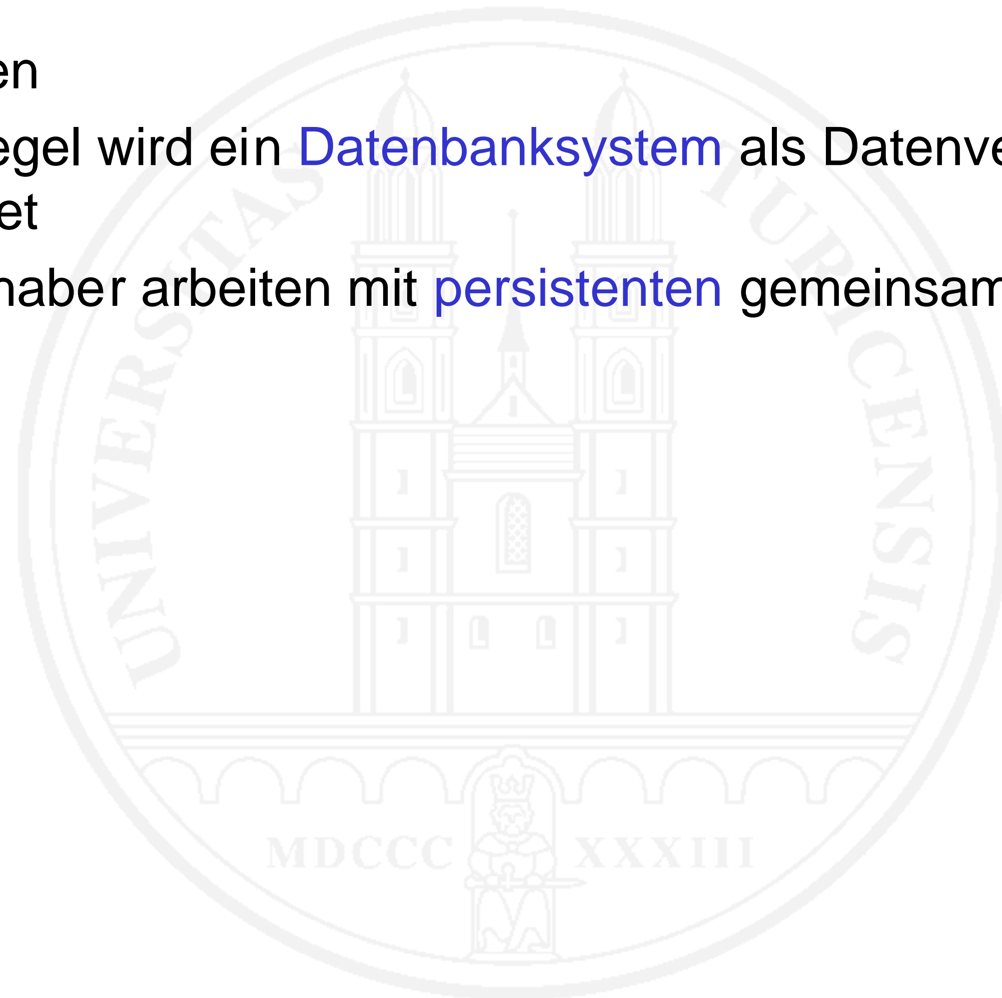
© 2004 by Martin Glinz

Informationsteilhabe: Informationsdepot – 1

- Situation 2
  Mehrere Teilhaber betreiben ein gemeinsames Informationsdepot (Repository)
- Mittel:
  - Verwaltung und Zugriff durch einen Datenverwaltungsdienst



Informationsteilhabe: Informationsdepot – 2



- Bemerkungen
  - In der Regel wird ein Datenbanksystem als Datenverwaltungsdienst verwendet
  - Die Teilhaber arbeiten mit persistenten gemeinsamen Objekten

Mini-Übung 3.6

Die Authentifizierung eines Benutzers soll über eine persönliche Benutzerkarte mit PIN-Code erfolgen

Entwerfen Sie eine Modularisierung dieser Authentifizierung

a) nach dem Prinzip des Delegierens von Aufgaben

b) nach dem Prinzip der Wertschöpfungskette (Bringprinzip)

3.3.2 Zusammenarbeit und Entwurfsstil

- Die meisten Entwurfs- bzw. Architekturstile verwenden eine charakteristische Form der Zusammenarbeit

- Zusammenarbeit bei funktionsorientiertem Stil
  - Leistungserbringung mit statisch gebundenen Funktionen und Prozeduren
  - Azyklische Aufrufhierarchie
  - Übergabe von Daten als Parameter
  - Sekundär: Zusammenarbeit durch Informationsteilhabe mit direktem Lesen/Schreiben gemeinsamer Speicherbereiche

Zusammenarbeit und Entwurfsstil – 2

- Zusammenarbeit bei datenorientiertem Stil
  - Leistungserbringung mit statisch gebundenen Funktionen und Prozeduren
  - Azyklische Aufrufhierarchie
  - Übergabe von Daten als Parameter
  - Informationsteilhabe über gemeinsame Speicher (gekapselt in abstrakten Datentypen) oder über gemeinsames Informationsdepot

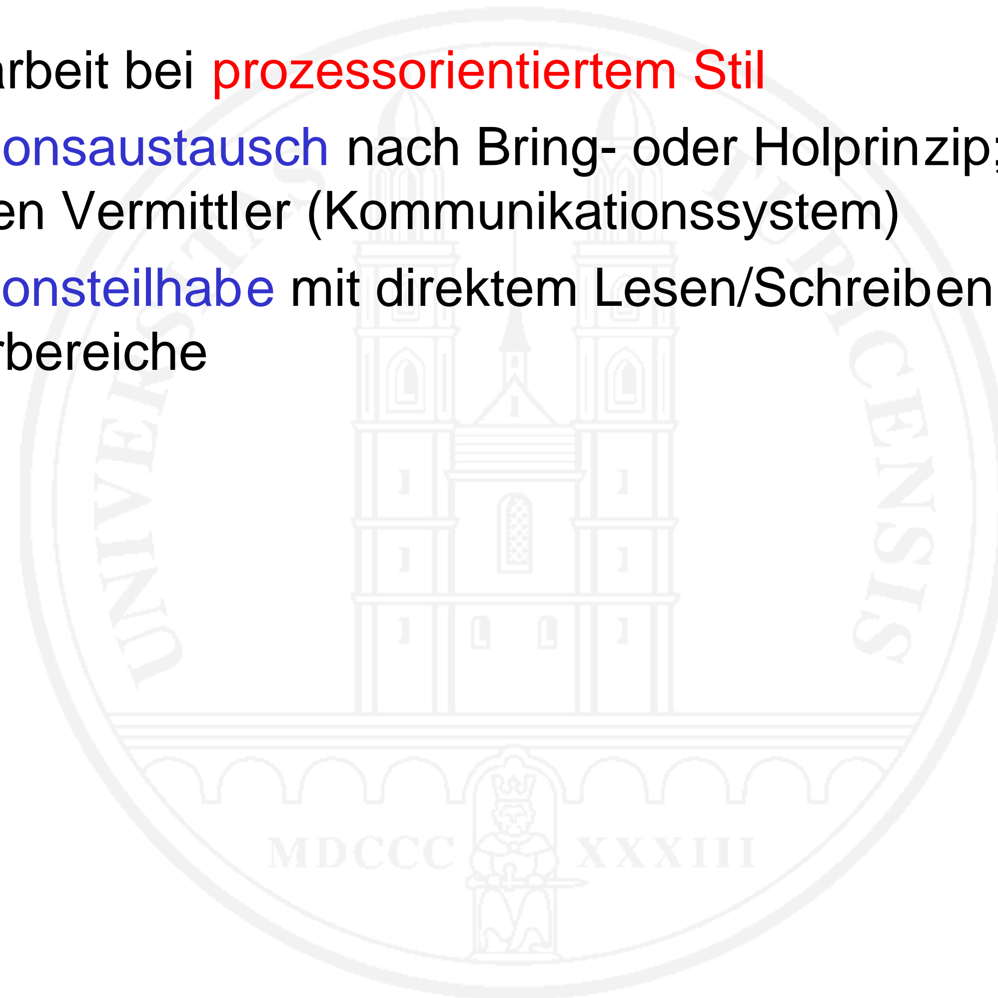
Zusammenarbeit und Entwurfsstil – 3

- Zusammenarbeit bei objektorientiertem Stil
  - Alle Zusammenarbeitsformen möglich
  - Leistungserbringung mit statisch oder dynamisch gebundenen Methoden
  - Übergabe von Daten und Objekten
  - Meistens nicht azyklisch
  - Informationsaustausch in allen Formen; häufig nach dem Abonnementsprinzip
  - Informationsteilhabe über gemeinsame Speicher (gekapselt in Klassen) oder über gemeinsames Informationsdepot
  - Bestimmte Entwurfsmuster (Gamma et al. 1995) verwenden spezifische Arten der Zusammenarbeit, z.B. Abonnementsprinzip im Beobachtermuster

© 2004 by Martin Glinz

Zusammenarbeit und Entwurfsstil – 4

- Zusammenarbeit bei prozessorientiertem Stil
  - Informationsaustausch nach Bring- oder Holprinzip; in der Regel über einen Vermittler (Kommunikationssystem)
  - Informationsteilhabe mit direktem Lesen/Schreiben gemeinsamer Speicherbereiche



3.3.3 Dokumentation der Zusammenarbeit

- Durch Zusammenarbeit werden aus Modulen bzw. Komponenten Systeme
- Die Festlegung und Dokumentation der Zusammenarbeit ist eine zentrale Entwurfsaufgabe
- Dokumentation der einzelnen Komponenten
  - Leistungsangebot: Angebotsschnittstelle(n)
  - Leistungsbedarf: Bedarfsschnittstelle(n)
- Dokumentation der Komposition
  - Statische Struktur typisch durch Diagramme
  - Dynamischer Ablauf wo nötig durch Zusammenarbeitsprotokolle (meistens mit Automaten / Statecharts
- Der Dokumentationsbedarf hängt vom Grad der Unabhängigkeit der Komponenten ab

© 2004 by Martin Glinz

Gemeinsam erstellte Komponenten

Bei gemeinsam erstellten und vertriebenen Komponenten

- Komponenten und Zusammenarbeit werden miteinander verzahnt entworfen
- Der Verwendungskontext jeder Komponente ist bekannt
- Entwerfende stimmen Schnittstellen und Zusammenarbeitsbedürfnisse aufeinander ab
  - Angebotsschnittstellen häufig nur syntaktisch definiert
  - Bedarfsschnittstellen in der Regel nicht explizit definiert, ggf. Namen der benötigten Komponenten aufgelistet
  - Dokumentation der Zusammenarbeit durch Kompositions-diagramm(e)

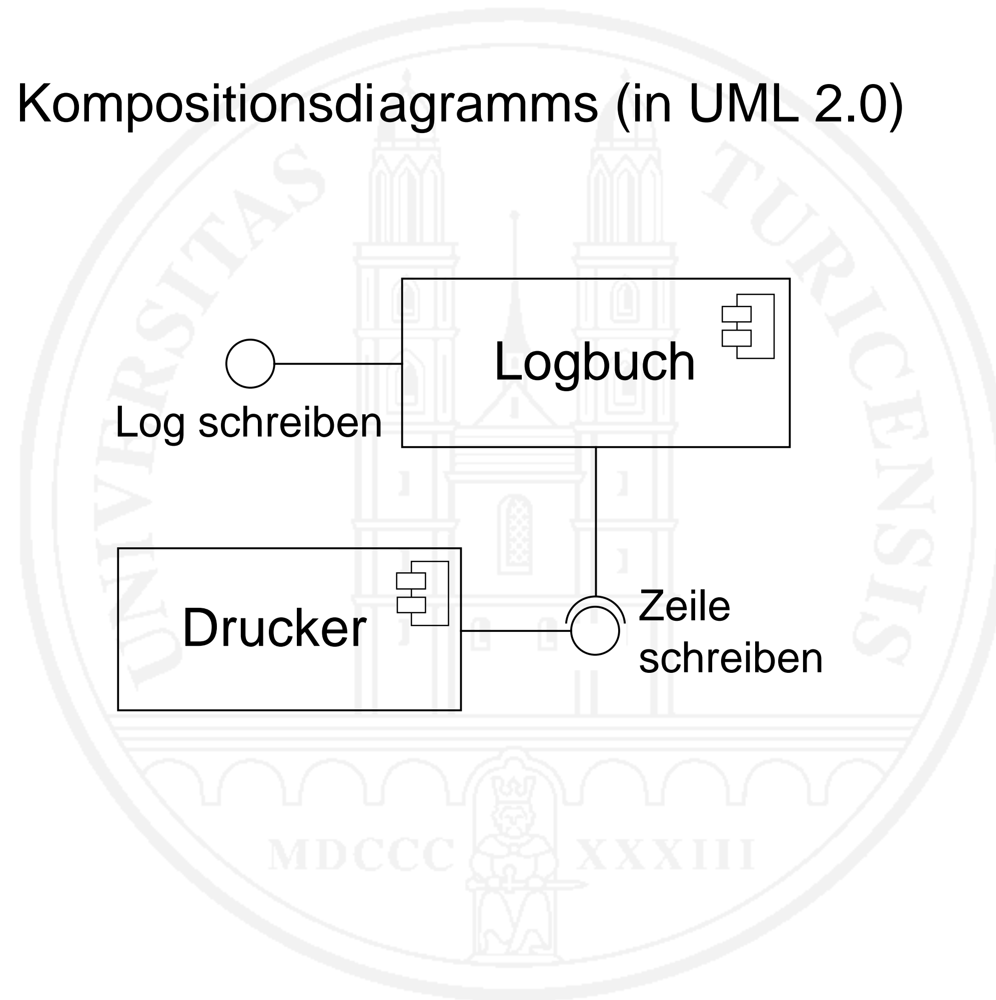
Separat erstellte Komponenten

- Bei separat erstellten und vertriebenen Komponenten
- Jede Komponente steht für sich und kennt bei Erstellung ihren Verwendungskontext nicht
- Präzise Definition der Angebotsschnittstelle(n) notwendig, damit die Komponente verwendbar ist
- Präzise Definition der Bedarfsschnittstelle(n) notwendig, damit die Komposition von Komponenten möglich ist
- Dokumentation der Komposition durch Kompositionsdiagramme

© 2004 by Martin Glinz

Kompositionsdiagramm

Beispiel eines Kompositionsdiagramms (in UML 2.0)



Merkpunkte Zusammenarbeit

- Charakteristische Arten der Zusammenarbeit :
  - Leistungserbringung: Prinzip des Delegierens von Aufgaben
  - Informationsaustausch:
    - Bringprinzip (Wertschöpfungskette / Fließbandarbeit)
    - Holprinzip (Einkäuferkette)
    - Abonnementsprinzip (Lieferverträge)
  - Informationsteilhabe
    - gemeinsamer Speicher
    - Informationsdepot
- Die Art der Zusammenarbeit ist charakteristisch für den verwendeten Entwurfsstil
- Zusammenarbeit und Komposition von Modulen müssen dokumentiert werden

3.1 Modulkonzepte und Schnittstellen

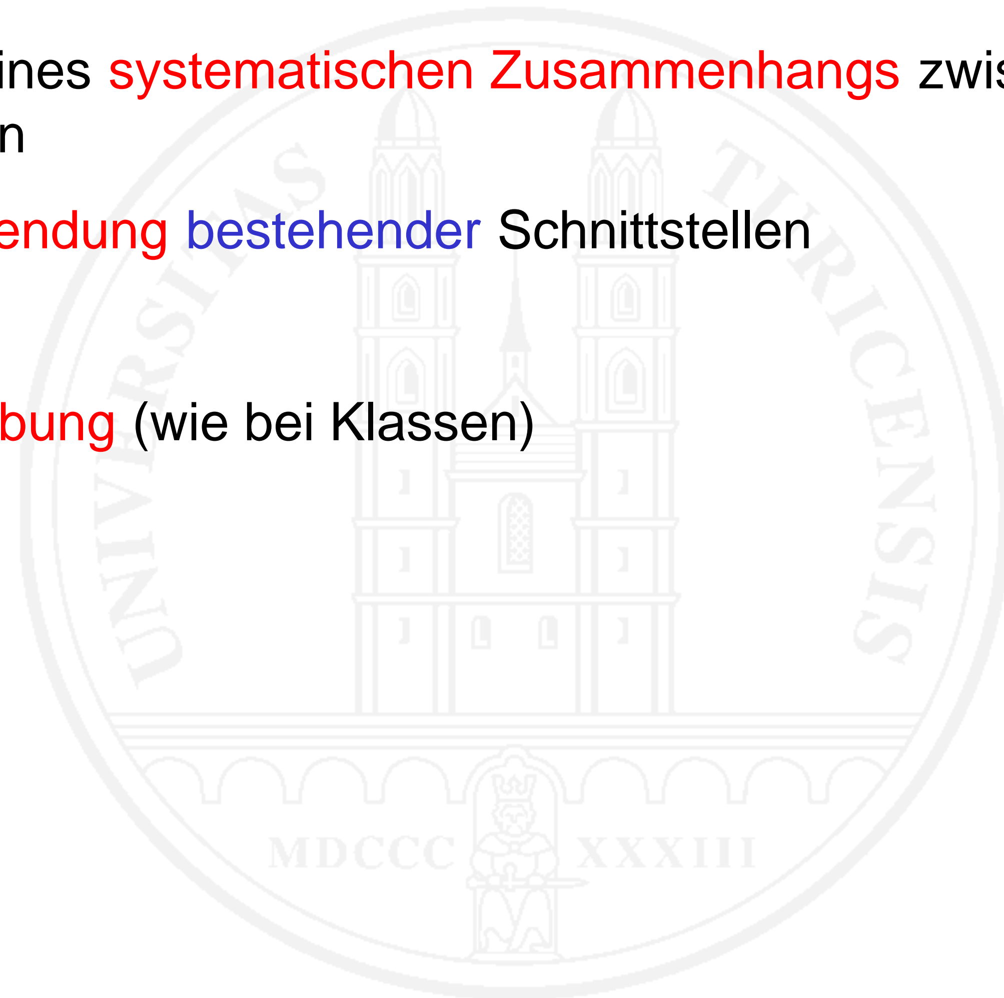
3.2 Vertragsorientierter Entwurf

3.3 Zusammenarbeit

3.4 Spezialisierung von Schnittstellen; Schnittstellenvererbung

© 2004 by Martin Glinz

Das Problem



- Schaffung eines systematischen Zusammenhangs zwischen ähnlichen Schnittstellen
- Wiederverwendung bestehender Schnittstellen
- Mittel: Vererbung (wie bei Klassen)

© 2004 by Martin Glinz

Beispiele – 1

- Von der Schnittstelle EinfachesKonto wird eine Schnittstelle Spielkasino abgeleitet, weil die Definition der Ein- und Auszahloperationen teilweise wiederverwendet werden kann
  - Die Schnittstellen haben keinen systematischen Zusammenhang
  - Finger weg von dieser Art von Wiederverwendung
  - Zu Grunde liegendes Prinzip: Steinbruch
- Von der Schnittstelle EinfachesKonto wird eine Schnittstelle Sparkonto abgeleitet.
  - Sparkonto hat «Saldo ≥ 0» als zusätzliche Invariante
  - Sparkonto ist ein Spezialfall von EinfachesKonto
  - Zu Grunde liegendes Prinzip: Spezialisierung

Beispiele – 2

- Aber: Korrekte Implementierungen von Sparkonto sind keine korrekten Implementierungen von EinfachesKonto
- Die Implementierungen sind nicht substituierbar

Warum ist das so?

- Von der Schnittstelle EinfachesKonto wird eine Schnittstelle KontoMitVerbuchung abgeleitet, welche bei jeder Mutation von Saldo zusätzlich die Verbuchung dieser Mutation zusichert
  - KontoMitVerbuchung ist eine echte Subschnittstelle von EinfachesKonto: Jede korrekte Implementierung von KontoMitVerbuchung ist gleichzeitig auch eine korrekte Implementierung von EinfachesKonto
  - Zu Grunde liegendes Prinzip: Substituierbarkeit

Arten der Vererbung

Das Prinzip der Vererbung kann in gleicher Weise wie auf Klassen auch auf Schnittstellen angewendet werden:

- **Steinbruch**: Die Vererbung dient nur dazu, Schreibaufwand zu sparen, indem Teile einer bestehenden Schnittstelle übernommen werden. Im übrigen wird beliebig ergänzt und abgeändert.
- **Spezialisierung**: Sei S' eine Subschnittstelle von S. Die von S' offerierten Leistungen sind ein Spezialfall der von S offerierten Leistungen.
- **Substituierbarkeit**: Sei S' eine Subschnittstelle von S. Jede korrekte Implementierung von S' ist gleichzeitig auch eine korrekte Implementierung von S. Dementsprechend ist jede Implementierung von S durch eine beliebige (korrekte) Implementierung von S' ersetzbar.

© 2004 by Martin Glinz

Merkpunkte Schnittstellenvererbung

- Schnittstellenvererbung ermöglicht
  - systematische Darstellung des Zusammenhangs zwischen ähnlichen Schnittstellen (mit Überlappungen im Leistungsangebot)
  - Wiederverwendung von Schnittstellen
- Drei Arten der Vererbung
  - Steinbruch: pure, nicht reflektierte Wiederverwendung ➪ vermeiden
  - Spezialisierung: Systematischer Zusammenhang, aber keine Substituierbarkeit
  - Substituierbarkeit: Nebenwirkungsfreie Verwendung von Subschnittstellen

Literatur

Gamma, E., R. Helm, R. Johnson, J. Vlissides (1995). *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, Mass. etc.: Addison -Wesley.

Glinz, M. (2003). *Software Engineering I*. Vorlesungsskript. Universität Zürich.

Meyer, B. (1988). *Object-Oriented Software Construction*. Englewood Cliffs, N.J.: Prentice Hall. [auf Deutsch: *Objektorientierte Softwareentwicklung*, München: Hanser, 1990]

Meyer, B. (1992). Applying "Design by Contract". *IEEE Computer* **25**, 10 (Oct. 1992). 40-51.

Meyer, B. (1997). *Object-Oriented Software Construction*. Englewood Cliffs, N.J.: Prentice Hall. [Neuauflage von Meyer (1988)]

Page-Jones, M. (1988). *The Practical Guide to Structured Systems Design*. Englewood Cliffs, N.J.: Prentice Hall.

Parnas, D.L. (1972). On the Criteria To Be Used in Decomposing Systems into Modules. *Communications of the ACM* **15**, 12 (Dec. 1972). 1053-1058.

Stevens, W.G., G. Myers, L. Constantine (1974). Structured Design. *IBM Systems Journal* **13**, 2. 115-139.

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

Wirth, N. (1985). *Programming in Modula-2*. 3rd edition. Berlin, etc.: Springer.

 © 2004 by Martin Glinz