

---

# Komponentenprogrammierung

- Software aus bestehenden Komponenten zusammensetzen
- gewinnt zunehmend an Bedeutung
- alte Idee, bisher wenig erfolgreich, warum jetzt?
  - mehr Erfahrung, aktives Forschungsthema
  - in Teilbereichen starke Unterstützung durch Industrie
  - Technologie breiter verfügbar (Generizität, zahlreiche Komponentenplattformen)
  - Ausbildung

---

# Komponente = Objekt?

- grundlegende Sichtweise: Komponente = Modul = Objekt
- praktische Unterschiede:
  - Komponenten sollen leicht von einer Umgebung zur anderen portierbar sein
  - „Require Interface“ zusätzlich zu „Deliver Interface“
  - fixe Annahmen über Umgebung (Komponentenmodell)
  - Komponenten viel komplexer als einfache Objekte
- Komponente  $\neq$  Klasse  
Komponentenbibliothek  $\neq$  Klassenbibliothek

---

# Probleme mit Komponenten

- viele Komponenten „low level“
  - Komponentenprogrammierung bringt wenig
- anwendungsorientierte große Komponenten gewünscht, aber wie oft verwendet? — nur für bestimmte Applikation (oder sogar nur bestimmte Version)
- viele Probleme noch ungelöst
  - Auffinden, Verteilung, Geschäftsmodelle
  - Beschreibung der Eigenschaften
  - Vertrauen in fremdentwickelte Komponenten
  - Kompatibilitätsprüfung (einfache Typen unzureichend)
  - hot swappable components (Zustand beachten)

---

# Forderungen an gute Bibliotheken

- systematische Taxonomie für Bibliothek
  - sonst Erlernen der Bibliothek zu teuer
  - Verhalten besser einschätzbar (Zusicherungen, Namen)
- so generisch wie möglich
  - Anwendbar in vielen Bereichen
- so effizient wie möglich
  - sonst nicht verwendet
- Bibliotheken müssen verständlich sein
  - sonst nicht verwendet

---

# Persistente Objekte (einfache Form)

Objekte persistent, wenn in einer Datei/Datenbank gespeichert und länger als Programm existent

**Einfachste Form:** Objekt zu (char \*) konvertiert, als String gespeichert, beim Einlesen zurückkonvertiert

- einfach, aber funktioniert nicht mit Zeigern  
→ vorher auflösen
- nicht portabel, da Speicherformat von Hardware, Compiler, Programmversion (Bibliotheksversion) abhängig
- fehleranfällig weil oft nicht im Detail verstanden

Persistenzmechanismen von Standardbibliotheken kaum besser

---

# Persistente Objekte

meist sinnvoll, externe von interner Darstellung zu trennen:

- internes Format leichter änderbar
- leichter portierbar
- Zeiger (Referenzen) explizit darstellbar

Funktionen zur Konvertierung zwischen interner und externer Darstellung meist kontextabhängig

gemeinsame Oberklasse aller Objekte, die persistent gemacht werden können, nur begrenzt sinnvoll

---

# Konzept versus Datenstruktur

Beispiel: Implementierung einer Liste

Variante A: Liste als Konzept; Listenelemente privat

Variante B: Klasse beschreibt Listenelement (Datenstruktur)

- B einfacher zu implementieren
- A empfohlen, da schwächere Objektkopplung (B führt leicht zu zusätzlichen Parametern)
- Probleme bei der Verwendung von B werden normalerweise früh erkannt, und Refaktorisierung führt zu A

→ Denken in Konzepten

---

# Globale und statische Variablen

- globale Variablen bewirken, dass Änderungen beliebige andere Programmteile beeinflussen können
- statische Variablen von vielen Objekten manipulierbar
- statische Variablen in „templates“ fehleranfällig (homogene versus viele Arten heterogener Generizität)
- Konstante bilden oft Ausnahme (Vermeidung starker Objektkopplung würde Änderbarkeit einschränken)
- statische Variablen sinnvoll, wenn tatsächlich als Variablen der Klassen selbst gesehen



---

# Zugriff auf Variablen

- public Variablen machen Implementierungsdetails sichtbar
  - nachträgliche Codeänderungen schwierig
  - public Variablen vermeiden
- Variablen möglichst nur über „this“ schreiben
- Vermeidung von public Variablen ist durch „get“- und „set“-Methoden immer möglich
  - Änderungen leichter als bei Variablen, da Methoden in Unterklassen überschreibbar
  - starke Objektkopplung bleibt
  - besseres Design überlegen

---

# Typumwandlung

- „static cast“ besonders gefährlich, da unüberprüfte Annahmen fix verdrahtet
- „dynamic cast“ (überprüfte Typumwandlung) auch gefährlich, da falsche Annahmen spät erkannt
- Typumwandlungen setzen fixe, bekannte Typhierarchie voraus, was nachträgliche Änderungen erschwert

---

# Bedingte Anweisungen, Typvergleiche

- scheinen harmlos weil häufig verwendet
- Typvergleiche setzen fixe Typhierarchie voraus
  - nachträgliche Änderungen erschwert (nicht lokal)
- case-Statements setzen fixe Menge von Fallunterscheidungen voraus – Änderungen der Menge nicht lokal
- Bedingte Anweisungen und Typvergleiche durch dynamisches Binden fast immer ersetzbar
  - Änderungen lokal und überprüft

---

# Binäre Methoden – Lösungsvergleiche

- dynamischer Typvergleich  
→ akzeptabel, da nur eigener Typ vorkommt
- Kombination Überladen und dynamischer Typvergleich  
→ effizient wenn Typ statisch bekannt
- Multimethoden bzw. Visitor Pattern → meist overkill
- einfache Generizität  
→ selten eine Lösung, da Typ statisch unbekannt
- gebundene Generizität  
→ Lösung, wenn von Sprache und Klassen unterstützt  
(aber Probleme bei tiefen Hierarchien)

---

# Arten gebundener Generizität

- F-gebundene Generizität:  $S \leq T\langle S \rangle$

Beispiel:  $\text{Integer} \leq \text{Comparable}\langle \text{Integer} \rangle$

kann mit binären Methoden umgehen

funktioniert nur eingeschränkt über mehrere Ebenen

- higher-order subtyping (matching):

Def.:  $S <\# T$  wenn  $S\langle U \rangle \leq T\langle U \rangle$  für alle  $U$

Matching unterstützt binäre Methoden

als Schranke bei Generizität typsicher

allgemein muss  $U$  für statische Typsicherheit bekannt sein

---

# Wann Generizität, wann Subtyping

- Mechanismen nicht immer austauschbar. Daher:  
Subtyping für heterogene Datenstrukturen  
Generizität zur Vermeidung kovarianter Probleme
- Erwartete Änderungen abfangen:  
Generizität wenn sich Parametertypen ändern  
Subtyping für neue Versionen mit gemeinsamem Obertyp
- Subtyping für Kompatibilität zu gegebener Klasse
- Subtyping um case-Statements, if-Statements und dynamische Typvergleiche zu vermeiden
- kombinieren, oder natürlicheren Mechanismus verwenden

---

# Rückgabewerte von Funktionen

- Ergebnistyp „void“ wenn kein Rückgabewert nötig  
→ unbedeutend bessere Fehlererkennung
- „this“ als Rückgabewert wenn nichts anderes nötig  
→ manchmal schönere Syntax

Beispiel: `Collection x = f(); x.add(y); x.add(z); g(x)`  
ersetzt durch  
`g(f().add(y).add(z));`