

---

# Implementierung Einfachvererbung

- jedes Objekt enthält Zeiger auf dynamischen Typ
- Typ enthält Array von Zeigern auf Methoden (VFT)
- Methode intern als Array-Index dargestellt,  
Variable als Offset von Objektadresse
- dyn. Binden: Methode in Array-Eintrag des dyn. Typs  
Variablenzugriff über Offset (dyn. Typ nicht nötig)
- Einfachvererbung: Indexe und Offsets bleiben erhalten  
zusätzliche Einträge und Offsets hinten angehängt  
ererbte Methode = kopierter Array-Eintrag  
überschriebene Methode = überschriebener Array-Eintrag

---

# Implementierung Interfaces

- einfache Impl. für Mehrfachvererbung ungeeignet (unterschiedliche geerbte Array-Indexe für selbe Methode)
- Array-Eintrag für jedes implementierte Interface (zeigt auf Interface, sonst wie Methodenzeiger)
- Interface enthält Array von Methodenzeigern (und Variablenoffsets wenn unterstützt)
- eine Indirektion mehr, Variablenzugriffe teurer
- Einfachvererbung von Interfaces durch hinten anhängen, Mehrfachvererbung durch zusätzliche Interfaces in Array
- manchmal Interface-Array direkt in Objekt

---

# Implementierung Mehrfachvererbung 1

- im Prinzip Techniken für Interfaces verwendbar  
wenn nötig mehrere Arrays (VFT = virtual function table)
- Objektlayout ererbter Klassen in C++ direkt übernommen  
(für up-cast auf Objekten notwendig)
- up-cast ändert Zeiger um Konstante „delta“ auf Anfang  
des Teilobjekts
- Anfang jedes Teilobjekts zeigt auf eigenen VFT
- bei Vergleichen auf Identität „delta“ berücksichtigen

---

# Implementierung Mehrfachvererbung 2

- Methodenaufruf bei Mehrfachvererbung (allgemein):
  1. `load [objectReg + #VFToffset], tableReg`
  2. `load [tableReg + #deltaOffset], deltaReg`
  3. `load [tableReg + #selectorOffset], methodReg`
  4. `add objectReg, deltaReg, objectReg`
  5. `call methodReg`
- „delta“ meist 0 → erlaubt Optimierung mit „thunks“:
  - Befehle 2 und 4 weglassen
  - Sprung auf „thunk“ (= überschriebene Methode):

```
thunk: add objectReg, #delta, objectReg
      jump #method
```

---

# Optimierungsmöglichkeiten

- wenn Methode statisch bekannt:
  - direkter Sprung statt über VFT
  - „inlining“ und weitere Optimierungen möglich
- Methode eher statisch bekannt wenn Compiler komplette Typhierarchie kennt (Eiffel)
- teilweises „inlining“ (Spezialisierung) durch „thunks“
- durch Wachsen in beide Richtungen und bei statisch bekannter Typhierarchie „thunks“ fast immer vermeidbar
- dynamisches Caching: zuerst auf selbe Methode wie bei letztem Aufruf springen, dann schauen ob Typ passt und nötigenfalls weiterspringen

---

# Ist Code-Mehrfachvererbung nötig?

- technische Antwort:  
Vererbung ist überhaupt nicht nötig, nur Subtyping
- Grund: Code-Vererbung lässt sich z.B. über Decorator-Pattern fast immer simulieren
- aber Code-Vererbung ist praktisch und sinnvoll
- einfache Code- und mehrfache Interface-Vererbung fordert frühzeitige Entscheidung, ob Interface oder Klasse
- Mehrfachvererbung auch sinnvoll wenn nur selten benötigt
- mehr Erfahrung → Gefahr falscher Anwendung geringer

---

## Code-Mehrfachvererbung: Beispiele

- eine Oberklasse implementiert eine Template Method, eine andere implementiert eine Operation der Template Method
- eine Oberklasse implementiert nur inhaltlichen Code, eine andere nur Synchronisationscode
- beide Beispiele sind „mixins“ im weiteren Sinne, wobei eine Unterklasse bestimmt, aus welchen vorgefertigten austauschbaren Teilen sie besteht
- alternative Lösungen sind jeweils möglich, aber teilweise schwer realisierbar wenn Oberklassen-Code nicht sichtbar ist oder nicht geändert werden soll

---

# Implementierung Exception Handling

- Compiler erzeugt Tabelle, die Adressbereichen (Blöcken) Exception-Handler zuordnet (für jeden Exception-Typ)
- nach Auftritt wird Handler für aktuellen Instruction Pointer (IP) gesucht und (wenn vorhanden) darauf gesprungen
- existiert kein passender Handler, wird aus aktueller Routine zurückgekehrt und mit neuem IP (Aufrufstelle) gesucht
- für Code nach Rückkehr aus „main“ existiert Handler
- kein Overhead ohne Exception, Handler-Suche billig, Objekterzeugung manchmal teuer
- bei Retry-Semantik ruft Handler ursprünglichen Code auf

---

# 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, Refaktorisierung führt zu A
- falsche Denkansätze rasch erkannt und beseitigt wenn Architekt = Programmierer, sonst problematisch

---

# Globale und statische Variablen

- globale Variablen böse, statische Variablen fast so böse
- statische Variablen in „templates“ fehleranfällig  
(homogene versus viele Arten heterogener Generizität)
- Konstante (statische Variablen) bilden oft Ausnahme:  
Vermeidung starker Kopplung schränkt Änderbarkeit ein
- statische Variablen sinnvoll, wenn tatsächlich als Variablen  
der Klassen selbst gesehen
- Intention unklar wenn Architekt  $\neq$  Programmierer

---

# Zugriff auf Variablen

- public Variablen machen Codeänderungen schwierig  
→ public Variablen vermeiden, sowieso klar
- Invarianten auf von außen geschr. Variablen unhandlich  
→ Variablen nur über „this“ schreiben, nicht jedem klar
- Vermeidung von public Variablen durch „get“ und „set“  
→ Änderungen leichter da Methoden überschreibbar  
→ klare Vor- und Nachbedingungen  
→ starke Objektkopplung bleibt  
→ besseres Design überlegen

---

# 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
- 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));`

---

# Aspekte

**Komponente:** alles was sauber in Prozedur/Objekt gekapselt werden kann

**Aspekt:** alles andere —

Eigenschaften, die die Effizienz oder Semantik einer Komponente auf systematische Weise beeinflussen, wie z.B.

- Muster der Speicherzugriffe
- Synchronisation bei Nebenläufigkeit
- Netzwerkbelastung
- Verschmelzung von Schleifen
- Speicherung von Zwischenergebnissen

---

# Ebenen der Interprozesskommunikation

- Transaktionen
- atomare Aktionen
- serialisierbare Kommunikation
- „mutual exclusion“
- einfacher Input/Output – unsynchronisiert

wenn mehr als „mutual exclusion“ gefordert, muss sich Programmierer selbst darum kümmern

---

# Aspekte der Kommunikation

Für die Ebenen der Interprozesskommunikation und der Sicherheit vor unerwünschten Zugriffen gelten:

- hohe Ebene: teuer und ineffizient
- niedrige Ebene: fehleranfällig, oft inakzeptabel
- Optimum hängt von der Anwendung ab
- Programmierer/Anwender sollen wählen können

---

# Reflektive Programmierung

**Reflektion:** Sprache wird dynamisch analysiert oder verändert

- lässt dem Programmierer viele Freiheiten
- sehr fehleranfällig
- Basis für aspektorientierte Programmierung

**Änderungen** der Semantik eines Programms möglich durch

- Änderungen des Programms
- Änderungen der Sprachen bzw. Standardbibliotheken
- Parameterisierte Sprachen bzw. Bibliotheken

---

## OO Arten der Reflektion

- interne Klassenstruktur sichtbar machen (Metadaten)
- Klassen zur Laufzeit hinzufügen, Typinfo. verwenden
- Klassen dynamisch erzeugen bzw. verändern
- „message passing mechanism“ oder andere Sprachkonzepte der untersten Stufe sichtbar machen und verändern

---

# Alternativen zur Reflektion

**Precompiler** vor Programmübersetzung ausführen:

- effizient zur Laufzeit
- einfache Änderungen mit kleinem Aufwand
- größere Analysen können teuer sein
- Laufzeitinformation nicht verfügbar

**Programmiersprache** ändern oder erweitern

- nur für große Änderungen sinnvoll
- interne Informationen im Compiler wiederverwendbar
- nicht portabel (mit Ausnahmen)
- einfach, wenn Erweiterung nur mit Bibliotheken