# 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 + #selectorOffest], 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 $\rightarrow$ 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

# 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

# Attribute in C#

- Attribut = Objekt, das mit Programmelementen verknüpfte Daten darstellt
- verknüpfbar mit Klassen, Methoden, Parametern, etc.:

```
[Serializable]
class MySerializableClass { ... }
```

- intrinsic (in .NET integriert) versus benutzerdefiniert
- Beispiel für Verwendung eines benutzerdef. Attributs:

```
[BugFixAttribute("Stefan", "23.3.2007", Comment="OK")]
class MyTestClass { ... }
```

- über Reflection zur Laufzeit zugänglich

# Beispiel für Attributdefinition

```
[AttributeUsage(AttributeTargets.Class,AllowMultiple=true)]
public class BugFixAttribute : System.Attribute {
  public BugFixAttribute (string programmer, string date) {
    this.programmer = programmer;  this.date = date);
  }
  public string Comment {
    get { return comment; }
    set { comment = value; }
  }
  public string Programmer { get { return programmer; } }
  public string Date { get { return date; } }
  private string programmer, date, comment;
}
```

# Annotationen in Java

- entsprechen ungefähr Attributen von C# – Beispiel:

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface BugFix {
  String prog(); String date(); String comment();
}
...
@BugFix(prog="Stefan", date="23.3.2007", comment="OK")
class MyTestClass { ... }
```

- SOURCE Annotationen ersetzen javadoc Annotationen

# Dynamische Erweiterung von Klassen

- dynamische Sprachen unterstützen Klassenerweiterung zur Laufzeit, statische Sprachen nicht direkt – Auswirkungen?
- Decorator erlaubt dynamische Erweiterung
- durch „final“ nur ohne Ersetzbarkeit verhinderbar

  $\rightarrow$ Erweiterbarkeit auch in statischen Sprachen Normalfall
- direkte Klassenänderung in dyn. Sprachen etwas effizienter als Decorator (Laufzeit + Programmieraufwand)
- statisches Konzept effizienter wenn keine Klassenänderung nötig (auch in dynamischen Sprachen)