Vererbung versus Delegation

```
class A { public void x() { z(); }
          protected void z() { /* A-Code */ ... }  }
class B extends A {
          protected void z() { /* B-Code */ ... }
          public void y() { delegate.x(); }
          private A delegate = new A();            }
```

- Vererbung: `new B().x()` $\longrightarrow$ B-Code
  Delegation: `new B().y()` $\longrightarrow$ A-Code
- Oft kein Überschreiben von Methoden wie `z()` $\Rightarrow$ egal
- Delegation sogar häufiger benötigt als Vererbung
- Unterschied manchmal wichtig da Verwechslung gefährlich

Mixins

- Begriff unklar, etwas unterschiedliche Bedeutungen
- Mehrfach-Code+Interface-Vererbung (oder ähnlich dazu) mit konfigurierbarem geerbten Code (statisch/dynamisch)
- Anwendungsbeispiel: Template Method geerbt und Operationen (der Template Method) durch Mixins hinzugefügt
- Unterschiede zwischen Mehrfachvererbung und Delegation für Mixins in Praxis oft nicht entscheidend (asymmetrisch)
- Oberklassen-Typparameter (C++) u. Delegation geeignet
- Mixin-Oberklassen oft nur zwecks Delegation instanzierbar

Lambda-Ausdrücke in OOP

- *Delegate* bzw. *Closure* kapselt eine Funktion in ein Objekt; im Gegensatz zu Funktionszeigern ist Umgebung enthalten (anonymisiert, nur über gekapselte Funktion zugreifbar)
- *Lambda-Ausdruck* = direkt erzeugtes Delegate/Closure
- Einfache Schreibweise (syntactic suggar + Intelligenz)
- Umgebung wird implizit aufgebaut (ähnlich innerer Klasse)
- Typinferenz für Parameter (da kein dynamisches Binden)
- Eher bei applikativem Programmierstil große Bedeutung

Dependency Injection (DI)

- Ziele: Abhängigkeit zwischen Client und Server verringern, *Boilerplate Code* vermeiden, einfache Konfigurierbarkeit
- Klassisch: *Inversion of Control* (z.B. Factory Method)
- M. Fowler: DI = Service durch *Mini-Container* bestimmt
  - Interface Injection (von Client implementiert)
  - Setter Injection (Mini-Container z.B. von Client befüllt)
  - Constructer Injection (bei Erzeugung gesetzt)
- Problem: Flexibilität durch Unübersichtlichkeit erkauft

Laden von Klassen zu Laufzeit

- Patchen von Systemen durch Austausch einzelner Module (Versioning, seltener Konfiguration)
- Prakischer Nutzen:
  kein Linker und dadurch einfacherer Übersetzungszyklus
- Suche nach anderen Anwendungen hat nichts ergeben
- Möglichkeiten von Java auf Paket-Ebene kaum genutzt (nur auf Dateisystem-Ebene nutzbar)
- Aktuellere Techniken für Componenten und Frameworks kommen ohne Patches auf Dateisystem-Ebene aus

Plugin Frameworks

- Zentrale Fragen: Wie findet man die Plugins?
  Wie (de)aktiviert man Plugins?
  Wo kann/soll Plugin eingreifen?
- Framework-spezifisches Interface für Plugins
- Konfigurationsdatei (meist XML) speichert Plugin-Liste
- An vorgegebenen Punkten Aufruf von Plugin-Methoden (über Interface) für jedes Plugin in Collection
- Oft hierarchisch (Plugins in Plugins), aber nicht zyklisch
- Zahlreiche Frameworks, die Organisation übernehmen (z.B. OSGi, JPF, Boost Extension, Qt, FxEngine, Zope)

Dynamische Sprachkonzepte

- Dynamische Vererbung u. Delegation (= Methodensuche)
- Dynamische Programmänderungen (Oberklasse ändern, Methoden und Attribute hinzufügen bzw. entfernen)
- Keine deklarierten Typen, wohl aber dynamische Typen
- Typprüfung implizit (keine passende Methode gefunden)
- Reaktion auf unbekannte Methoden programmierbar
- Programmiertechniken von extensiven Zusicherungen bis „Duck Typing“ (strukturelle Typen, Client verantwortlich)

Dynamische Änderbarkeit von Klassen

- Decorator erlaubt dynamische Änderung von Konzepten
  $\rightarrow$ Änderbarkeit auch in statischen Sprachen Normalfall
- Direkte Klassenänderung in dyn. Sprachen effizienter als Decorator (Laufzeit, Programmieraufwand, Lesbarkeit)
- Statische Sprachen effizienter wenn Änderungen selten
- Aber: nur vorhergesehene Änderungen durch Decorator
  $\rightarrow$ nicht für Versioning geeignet
- Änderungen in dynamischen Sprachen sofort wirksam
  (von Programmierern geschätzt oder gehasst)

Statische versus dynamische Typen

- Einiges geht nicht statisch (z.B. Array-Index prüfen, binäre Methoden mit Ersetzbarkeit)
- Dynamische Konzepte auch in statischen Sprachen sinnvoll (dynamische Typabfragen, Zugriff auf Typparameter)
- Anhänger dynamischer Sprachen überlegen selten, welche Information statisch ist $\rightarrow$ alles bleibt dynamisch
- Andere wollen mit viel Aufwand alles statisch machen
- Starke Typsysteme fast immer halbentscheidbar
  $\rightarrow$ Typkompatibilität erfordert bestimmte Typstruktur
  $\rightarrow$ Zusatzaufwand um bestimmte Struktur herzustellen

Primitive Datentypen

- Spezialbehandlung kleiner Zahlen in jeder Sprache
- Gründe: Effizienz, Semantik von Literalen
- „Integer“ in Ruby normale Klasse; Auswirkungen gering:
  - Zahlen gemischt mit anderen Objekten in Aggregaten (durch gemeinsame Oberklasse möglich)
  - homogene Übersetzung der Generizität einfacher
  - Erweitern von „Integer“ praktisch kaum sinnvoll
- Effizientes „Rechnen“ in statischen Sprachen wichtiger
- Kaum Unterschiede zw. rein objektorientierten und heterogenen Sprachen, aber oft andere praktische Verwendung

Serialisierung in Java

- Wenn `x` Instanz von *Serializable*, automatisch durch `s.writeObject(x)` wobei `s` vom Typ `ObjectOutputStream`
  Lesen: `x=s.readObject()` mit `s` vom Typ `ObjectInputStream`

- Serialisierbare Klasse kann/soll enthalten:
  `static final long serialVersionUID=...` (Version passt?)
  `private void writeObject(ObjectOutputStream s)`
  `private void readObject(ObjectInputStream s)`
  `private void readObjectNoData()` (wenn Daten fehlen)

- Normalerweise rekursive Aufrufe für referenzierte Objekte

- Volle Kontrolle durch Implementierung von *Externalizable*

Garbage Collection (GC)

- Effizient und vereinfacht Programmierung wesentlich
- Gute Denkansätze: Objekte leben konzeptuell ewig
  unnötige Referenzen beseitigen
  Resourcen von Speicher entkoppeln
- Ansätze für Speicherverwaltung funktionieren bei GC kaum (Kontrolle über Speicher, Destruktoren räumen auf)
- Keine Garantie für genug Speicher, daher manchmal trotz Echtzeitfähigkeit wegen dyn. Speicherverwalt. ungeeignet
- Programme sollen auch bei ausgeschalteter GC laufen

Trennung zwischen GUI und Logik

- Fast automatisch durch 3-schichtige Architektur im Web: Präsentation, Anwendungslogik, Datenhaltung
- Entsprechende Dreiteilung auch in Anwendungslogik: Steuerung, Verarbeitung, Datenzugriff
- MVC: Model=Verarbeitung+Datenzugriff+Datenhaltung, View=Präsentation, Controller=Steuerung
- Schwer wartbar wenn Controller=Steuerung+Verarbeitung (unterschiedliche Bereiche der Anwendungslogik gemischt)
- Gute Architektur = unterschiedliche Bereiche der Anwendungslogik getrennt halten (anwendungsspezifisch)

Zukunft der OOP

- OOP allgegenwärtig $\Rightarrow$ Innovations-Charakter verloren
- erleben heute Aufsplittung in viele Rand- u. Detailthemen
- wichtige Bereiche in absehbarer Zukunft: Parallelität, Verteiltheit, Datenintegration, vielschichtige Architekturen, Cloude-Computing, komplexe Schnittstellenprotokolle
- in vielen dieser Bereiche derzeit mehr Fragen als Antworten
- Sprachunterstützung zu erwarten, sobald die wichtigsten Fragen beantwortet sind