Funktionale OOP

- Mögliche Sichtweise: OO = Organisationsparadigma, Berechnungsmodell kann imperativ, funktional, etc. sein
- Widersprche: Objekt hat Zustand, Verhalten, Identität; Referentielle Transparenz = kein Zustand, keine Identität
- Praktisch: Ein- und Ausgabe nicht referentiell transparent
  $\rightarrow$ auch in FP gibt es Zustände und Identität
  $\rightarrow$ aber in FP gibt es auch referentiell transparente Teile
- Tatsächlich: FP+OOP = FP-Teil getrennt von OOP-Teil
- Pattern Matching (als FP-Konzept) mit OOP verträglich
- Generizität geht in Richtung referentieller Transparenz

Delegate in C#

```
using System;
delegate void Sample(string message);
class MainClass {
    static void Method(string message)
        { Console.WriteLine(message); }
    static void Main() {
        // Instantiate delegate with named method:
        Sample d1 = Method;
        // Instantiate delegate with anonymous method:
        Sample d2 = delegate(string message)
                    { Console.WriteLine(message); };
        d1("Hello");
        d2(" World");
    }
}
```

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

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

Trait

- ein Trait stellt implementierte Methoden bereit
- ein Trait verlangt Methoden
- Traits haben und benutzen keine Instanzvariablen
- Reihenfolge des Zusammenfügens von Traits irrelevant
- Verwendung von Traits ist so als ob Methoden lokal in Klasse (oder Trait der Traits verwendet) definiert wären
- Sprachen: Squeak, Scala, Perl 6 (roles);
  module mixins in Ruby ähnlich

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

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)

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

Garbage Collection (GC)

- Effizient und vereinfacht Programmierung wesentlich
- Gute Denkansätze: Objekte leben konzeptuell ewig
  unnötige Referenzen beseitigen
  Ressourcen 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

Sichtbarkeit und Sicherheit

- Sichtbarkeit auf Klassenebene (Java, C#, C++, . . .)
  $\rightarrow$ Konzentration auf Verantwortlichen für Klasse
- Sichtbarkeit auf Objektebene (Variablen in Eiffel)
  $\rightarrow$ Konzentration auf Zusicherungen und Ersetzbarkeit
- Sichtbarkeit als Sicherheitskonzept (theoretisch)
  $\rightarrow$ Objekte nur in einem bestimmten Bereich sichtbar
  - Uniqueness (nur eine Referenz auf Objekt)
  - Ownership (nur von einem Objekt referenziert)
  - Confinement (nur innerhalb einer Zone referenziert)

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)

Type Erasure

- entsteht bei homogener Übersetzung von Generizität:
  — Typparameter durch Schranke oder `Object` ersetzt
  — spitze Klammern samt Inhalt weggelassen
  — (type cast nötig wenn Ergebnistyp Typparameter war)

- `List<A>` und `List<B>` nur kompatibel wenn `A` = `B`

- bei direkter Verwendung von type erasure (`List` statt `List<Object>`) keine Überprüfung der Typkompatibilität

- `List` mit `List<A>` kompatibel obwohl nicht typsicher

- Warnung nur wenn Compiler unsichere type casts einfügt

Type Erasure aus Programmierersicht

- in Java Typparameterersetzungen nur statisch

  z.B. `Object x; ... x instanceof List<A> ...` nicht prüfbar

  → Verwendung von `List` statt `List<A>` einziger Ausweg

- wildcard types bieten zusätzliche statische Möglichkeiten

  — `List<? extends A>` erlaubt sichere Lesezugriffe

  — `List<? super A>` erlaubt sichere Schreibzugriffe

- in C# Typparameterersetzungen zur Laufzeit bekannt

  → minimal höherer Aufwand für Prüfungen zur Laufzeit

  → type erasure und wildcard types nicht benötigt

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