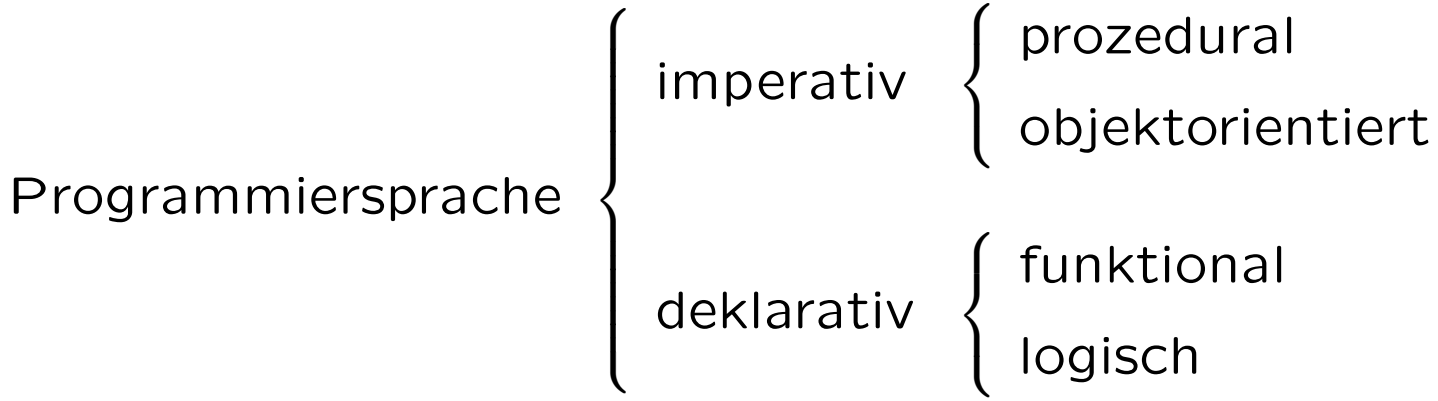
Programmierparadigmen

- Paradigma = *Denkweise* oder *Art der Weltanschauung*
- klassische Einteilung:



- Zusammenhänge tatsächlich wesentlich komplexer

Berechnungsmodell

- formaler Hintergrund

  (Funktionen, diverse Logiken, Algebren, Automaten, etc.)

- praktische Realisierung entscheidend
  - Kombinierbarkeit
  - Konsistenz
  - Abstraktion
  - Systemnähe
  - Unterstützung und Beharrungsvermögen

Struktur im Kleinen

- widersprüchlich: flexibel, sicher, verständlich

  $\rightarrow$ breites Spektrum an Sprachen, Evolution nichtlinear

- strukturierte Programmierung:

  Sequenz, Auswahl, Wiederholung

- Umgang mit Querverbindungen (durch Seiteneffekte)

  funktional $\rightarrow$ Seiteneffekte verbieten

  objektorientiert $\rightarrow$ Querverbindungen sichtbar machen

- First-Class-Entities

  $\rightarrow$ Aufwand lohnt sich nur für wichtigste Konzepte

Modularisierungseinheiten

- Modul (Übersetzungseinheit, zyklenfrei)
- Objekt (zur Laufzeit)
- Klasse (Modul und Muster für Objekterzeugung)
- Komponente (komplexere Initialisierung)
- Namensraum

Parametrisierung

- Befüllen der Löcher zur Laufzeit
  - Konstruktor
  - Initialisierungsmethode
  - zentrale Ablage
- Generizität (Überseztungszeit)
- Annotationen (Parametrisierung optional)
- Aspekte (Modifikation des Programms)

Ersetzbarkeit

A durch B ersetzbar wenn B überall verwendbar wo A erwartet

Schnittstellen von A und B folgendermaßen spezifizierbar:

- Signatur (einfach, wenig aussagekräftig)
- Abstraktion (intuitiv, nicht immer zuverlässig)
- Zusicherungen (manchmal komplex, recht zuverlässig)
- überprüfbare Protokolle (sehr komplex, nicht verfügbar)

Typisierung

- statische versus dynamische Typprüfungen
  = Einschränkungen versus Freiheit
  = diktierte Disziplin versus freiwillige Disziplin
- explizit deklarierte Typen verbessern Lesbarkeit
  $\rightarrow$ Sicherheit (durch Lesbarkeit, nicht Typprüfungen)
- deklarierte Typen $\rightarrow$ frühe Entscheidungen $\rightarrow$ Effizienz
- deklarierte Typen $\rightarrow$ deklarierte Abhängigkeiten

Abstraktion und Typen

- *struktureller Typ* $\rightarrow$ nur Signatur
- *nominaler Typ* $\rightarrow$ Typname ermöglicht Abstraktion
- *abstrakter Datentyp (ADT)*
  = nominale Schnittstelle einer Modularisierungseinheit

$$\left.\begin{array}{r}\text{Objekt aus realer Welt}\\ \text{Softwareobjekt}\end{array}\right\}\text{gemeinsamer Name}$$

- ADT $\rightarrow$ Denken in Konzepten

Untertypen

Ein Typ U ist Untertyp eines Typs T wenn jede Instanz von U überall verwendbar ist wo eine Instanz von T erwartet wird.

- für strukturelle Typen eindeutig

  $\rightarrow$ ohne Deklarationen vom Compiler prüfbar

- für ADT in der Verantwortung der Programmierer

  $\rightarrow$ explizite Deklaration von Untertypbeziehungen

- Zusicherungen als Ergänzung

  $\rightarrow$ müssen in Untertypbeziehungen berücksichtigt werden

Gestaltungsspielraum für Typen

Typen zeigen Grenzen auf:

- kovariante Probleme bei Untertypen
- rekursive Datentypen erfordern induktive Konstruktion
- Typinferenz mächtig, aber nicht mit Untertypen

Typen bieten ungeahnte Möglichkeiten:

- praktisch alle Eigenschaften statisch propagierbar
- aber derzeit unzureichend unterstützt

Objekt

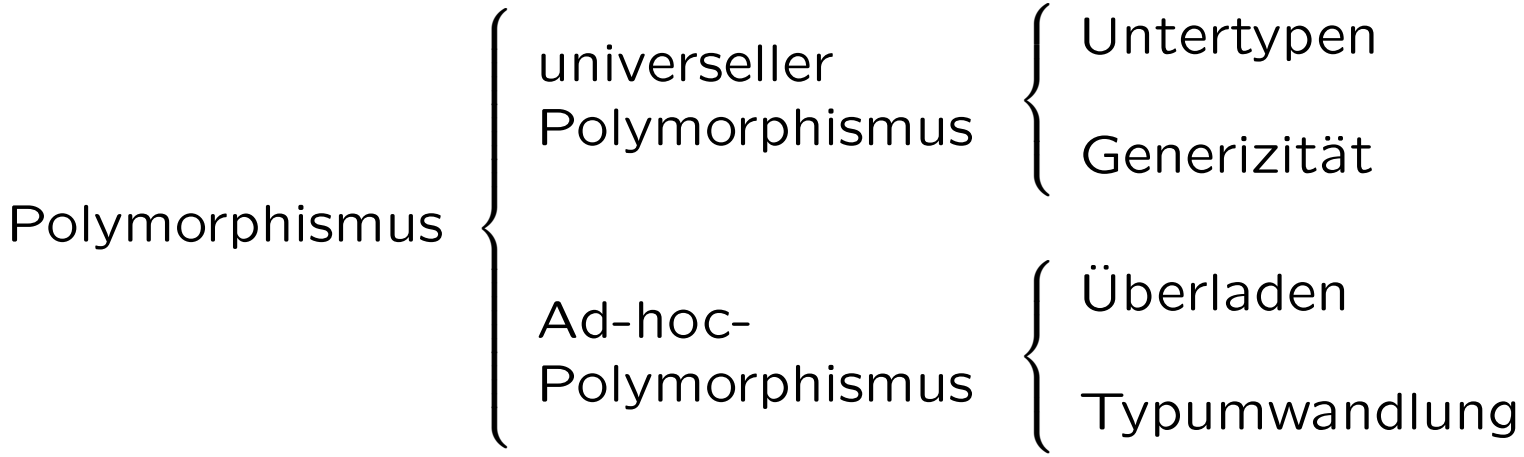
- kapselt Variablen und Methoden
- Eigenschaften:
  - Identität (identisch = mehrere Namen für ein Objekt)
  - Zustand (gleich $\neq$ identisch)
  - Verhalten (Reaktion auf Nachrichten)
- Softwareobjekt (ADT) simuliert, abstrahiert reales Objekt

Klasse

- beschreibt Struktur der Objekte
- Konstruktoren zur Initialisierung aller Objekte
- gleiche Klasse $\rightarrow$ selbe Schnittstellen und selbes Verhalten
- jedes Objekt ist Instanz genau einer Klasse

Polymorphismus

Objekt hat gleichzeitig mehrere Typen ($\approx$ Schnittstellen)



Untertypen und Vererbung

- Untertypen bedingen Ersetzbarkeit

  $\longrightarrow$ Variable hat *deklarierten* und *dynamischen Typ*

  $\longrightarrow$ dynamisches Binden

- Vererbung = Übernehmen von Code aus Oberklasse

- Praxis: Vererbung als Hilfsmittel für Untertypen

Erfolg in der OO-Programmierung

- gezielter Einsatz von *Erfahrung* erhöht Erfolgsaussichten
- OOP = viele Möglichkeiten zur *Faktorisierung*
  $\rightarrow$ erleichtert gezielten Einsatz von Erfahrung
  $\rightarrow$ ermöglicht Wiederverwendung durch Ersetzbarkeit
  $\rightarrow$ überfordert Anfänger
- OOP gut für große, langlebige Programme
- OOP schlecht für komplexe Algorithmen

Verantwortlichkeiten einer Klasse

- definiert durch drei w-Ausdrücke, Ich ist Objekt der Klasse:
  - was ich weiß (Zustand der Objekte)
  - was ich mache (Verhalten der Objekte)
  - wen ich kenne (sichtbare Objekte, Klassen)
- wer Klasse entwickelt ist zuständig für Änderungen in den Verantwortlichkeiten der Klasse

Klassen-Zusammenhalt

- Klassen-Zusammenhalt (class coherence) = Grad der Beziehungen zwischen den Verantwortlichkeiten der Klasse
- hoch, wenn
  - Variablen und Methoden eng zusammenarbeiten
  - und durch Klassenname gut beschrieben
- Klassen-Zusammenhalt soll hoch sein

  $\rightarrow$ Hinweis auf gute Faktorisierung

  $\rightarrow$ verringert Wahrscheinlichkeit für nötige Änderungen

Objekt-Kopplung

- Objekt-Kopplung (object coupling)
  = Abhängigkeit der Objekte voneinander

- stark, wenn
  - viele sichtbare Methoden und Variablen
  - viele Nachrichten im laufenden System
  - viele Parameter in Methoden

- Objekt-Kopplung soll schwach sein
  → Hinweis auf gute Kapselung
  → weniger unnötige Beeinflussung bei Änderungen

Frühe Bewertung von Alternativen

Klassenzusammenhalt und Objektkopplung

- hängen oft zusammen (beides gut oder beides schlecht)
- sind bereits in früher Entwicklungsphase abschätzbar
- helfen bei der Bewertung von Alternativen

Refaktorisierung

- Refaktorisierung = Änderung der Programmstruktur (bei gleicher Funktionalität)
- in Anfangsphase billig
- wenige gezielte Refaktorisierungen $\rightarrow$ stabile Faktorisierung