OOP

Paradigmen der Programmierung

Paradigma := bestimmte Denkweise, fundamentales Programmierstil

Einteilung nach:

- Berechnungsmodell, Programmstrukturen
- Programmorganisation
  - -> Modularisierung: Modul (Übersetzungseinheit), Objekt, Klasse, Komponenten
  - -> Parameterisierung: Flexibilität; Sicherheit, Generizität, Instanziierung, Implizit
  - -> Erweiterbarkeit: Signaturen, Abstraktion realer Welt, Zusicherungen, Ausprägbare Protokolle
- Typisierung
  - -> Typkonzepte, Verständlichkeit, Praxisbezug
    - dynamische, statische Typprüfungen
  - -> Abstraktion: strukturelle, nominale Typen -> Untertypen
  - -> Gestaltungsspielraum: Rekursive Typen, Typvarianz, Propagieren von Eigenschaft.

Objektorientierte Programmierung

Objekt: Identität, Zustand, Verhalten
-> Schnittstelle, Implementierung

Polymorphismus: universell (Generizität, Untertypen) ad-hoc (Überladen, Typumwandlung): Variable oder Methode gleichzeitig mehrere Typen

Dynamischer, Statischer, Statischer Typ
supremum / maximum finden

Vererbung: Überschreiben, Erweitern

Fabrikmuster: Zerlegen des Programms in zusammengehörige Teile

Verantwortlichkeiten Klasse: was ich weiß, was ich mache, wen ich kenne

Klassenzusammenhalt, Objekt-Kopplung

Untertypen und Vererbung

U ist Untertyp von T, wenn U überall verwendet werden kann, wo T verwendet wird

Strukturelle Untertypbeziehung: reflexiv, transitiv, anti-symmetrisch
lesen: kovariant, schreiben: kontravariant, lesen+schreiben: invariant
-> zyklische Schnittstelle

Problem: binäre Methoden: this und Eingabeparameter selber Typ
-> Typklassen, Casts

Java: Alles außer Ausgabeparam. invariant: statisch binden, überladen

-> Codeduplizierung
-> Änderungen lokal
-> Untertypen von statischen Obertypen

Client-Server Beziehungen: austauschbar, wenn als Server selbe Dienste

-> Vorbedingung: kontravariant (schwächer)
-> Nachbedingung: kovariant (stärker)
-> Invariante: (stärker, wenn keine Änderung von außen)
-> History Constraint: Server-konstruktiv: zB kann nur später werden (ähnlich invariante)
Client-konstruktiv: Aufrufreihenfolge (ähnlich Vorbedingung)

-> Nominaler Typ: Namen, Signatur (Struktur), Zusicherungen

Obertypen: abstrakte Klassen, Interfaces empfohlen

Beziehungen zwischen Klassen: Vererbung, Untertyp, "is-a"

Generizität

Einfache Generizität: Typparameter

Gebundene Generizität: Schranken: extends
↳ F-gebundene Generizität: rekursive Schranken

Typparameter

↳ keine impliziten Untertypen Comp<A>, Comp<B> ← Schreibzugriff

↳ gebundene Wildcards: List<? extends Rha>

-> extends: schreiben verhindert
-> super: lesen verhindert
-> <A extends Comp<? super A>>

lesen / schreiben direkt nicht mit jeder Klasse von

Einschränkungen in Java: neue A() unzulässig, Typüberprüf. zur Laufzeit

Arten: Übersetzung

-> homogen: generische Klasse -> eine übersetzte Klasse
Erasure durch untere Schranke -> casten

-> heterogen: jede Verwendung -> eigene Übersetzung
keine Typumwandlung, auch primitive Typen
keine Schranken für Methodenaufrufe
-> Performanter, aber implizite Schranken durch Code

Typabfragen und Umwandlung

-> statische Typsicherheit abgeschwächt
-> erhöhte Komplexität, Wartungsaufwand
-> mehr Möglichkeiten vermieden

-> Einsatz bei Homogener Übersetzung von Generizität
↳ Sichere Form

Sichere Typumwandlungen: in Obertyp
dessen dynamische Typabfrage (was wenig kein Zweig zutrifft)
Generizität nachbauen (was wenig keine Generizität)

Generizität: eingeschränkte Typumwandlung, trotzdem <> bei Übersetzung
wegoptimieren -> keine Info. über einfachere

Übersetzte Klassen: Raw Types

Kovariante Probleme (Eingangsparam): Typumwandlung / Abfrage
↳ vermeiden

Binäre Methoden: Template Method

Überladen vs. Multimethoden

Parameter werden nicht nach dynamischem Typ (deklarierter Typ) behandelt.
↳ Überladen
↳ Gegensatz: Multimethoden

-> Überladen nur so implementieren, dass dynamischer und deklarierter Typ gleich behandelt wird

↳ keine Untertypbez. in Parametern oder bei allen

Multimethoden: führen Komplexität bei Auswahl
↳ welche Methode wenn 2 gleich gut passen (2 Parameter)

Simulation von Multimethoden: Mehrfaches dynamisches Binden
↳ Visitor Pattern: Visitor Klassen, Visitor Methoden
Elementklassen

↳ Hohe Methodenanzahl: in jeder Elementklasse 1 + in jeder Visitorklasse n (anzahl Elementklassen)
↳ Elementklassen in Gruppen Einteilen

Ausnahmebehandlung

Throwable -> Error, Exception -> überprüfbare, nicht überprüfbare (Runtime)

Untertyp: Nur Ausnahmen, die auch im Obertyp geworfen werden
↳ nur in selten Fällen

-> Ausnahme sparsam, nur wenn Lesbarkeit verbessert wird
-> nicht lokale Effekte

- Kontrolliertes Wiederaufsetzen
- Unvorhergesehene Programmabbrüche
- Ausstieg aus Sprachkonstrukten
- Rückgabe alternativer Ergebniswerte

Nebenläufige Programmierung

synchronized Blöcke ∈ Locks, Monitor Konzept

wait(), notify All()

Runnable -> new Thread (r)

-> Vorgefertigte Lösungen: abstrakte synchronisierte Datentypen
Future Tasks
Executor

-> Deadlocks, Starvation

Zusicherungen: Lesbare Constraints (Class Invariant)
↳ nicht stärker synchronisiert

Fehlerbehandlung nach Objektverletzung anders als nach Nebenläufigkeit

Annotation und Reflexion

Zusätzliche Infos die bei Laufzeit, Übersetzung vorhanden sind

Definition als Interface

Element-Type, Retention Policy { source, class, Runtime
Method, Type, Parameter, Constructor

-> Ersatz für Syntax-Änderungen ∈ Hinweise für Compiler

Reflexion: Änderung des Programmverhalten bei Laufzeit
↳ Metaprogrammierung
↳ Klassenbrowser, Debugger, Testing

Aspektorientierte Programmierung

Aufteilung des Programms nach Anliegen
-> es gibt eigene Aspektorientierte Sprachen „Anliegen"

-> kapselt cross cutting concerns in verschiedenen Klassen

Aspekt: wie Klasse, erweitert Fähigkeiten

Join Point (Stelle im Programm), Pointcut (mehrere Join Points, benennt, Informationen; Parameter), Advice (Programmcode rennt um Join Point herum, after, around)

Design Pattern

... Wiederverwendung, kollektive Erfahrung

Bestandteile: Name
Problemstellung
Lösung: mehrere Varianten (anwendbar unabhängig in Library)
Konsequenzen: Vor- und Nachteile

→ Kombinierung schwierig

Verhaltensentwurfsmuster

- Visitor Pattern:

  Visitorklassen mit Visitor-Methoden
  Elementklassen: rufen Visitor-Methoden auf

  spezifische Methodenvorschrift: Elementklassen gruppieren

- Iterator:

  Ermöglicht sequentiellen Zugriff auf Aggregatselemente
  ohne Aggregatsdarstellung zu kennen

  → Einheitlicher Zugriff auf versch. Aggregate

  Aggregat → erzeugt → Iterator
  Conc. Aggr. - - - -> Concrete Aggregat

  intern: kontrollieren nächste Iteration selbst
  → funktionale Programm., virtuelle Routine

  extern: flexibler, Client kontrolliert nächste Iteration
  → vorgefertigte Einheiten

  → Aggregatsänderung kritisch → Referenzen Iterator

- Template Method

  definiert Grundgerüst von Algorithmus, überlässt Implementierung einzelner Schritte

  Template Method ruft zu implementierende (konkrete / abstrakte Operationen auf, bereits implementiert

  Oberklasse ruft Unterklasse auf
  ↳ primitive Operationen abstract, protected
  ↳ Template Methode final

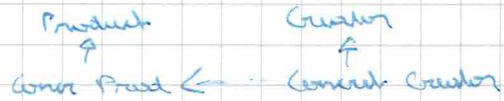
Erzeugende Entwurfsmuster

- Factory Method

  virtual Konstruktor, Schnittstelle für die Objekterzeugung,
  ↳ Unterklasse bestimmt, von welchem Typ erzeugtes Objekt ist
  ↳ Argumente zur Erstellung können verschieden verarbeitet werden

  Klasse erzeugt neue Objekte mit dynamischem Typ
  ↳ flexibler als new

  Parallele Klassen-Hierarchie

  -> Pizzaria / Burgerrestaurant <- erzeugt Essen

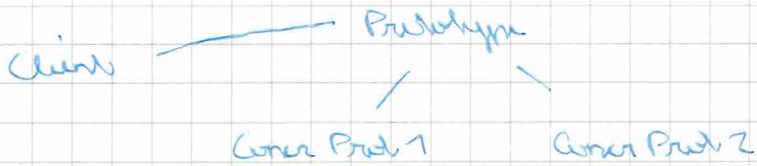
- Prototype

  erzeugt neue Objekte durch kopieren von bestehenden
  ↳ Klasse muss nicht bekannt sein -> flexibel
  ↳ Zustand kann geändert werden

  Prototypen können zur Laufzeit verändert werden (Klassen nicht)
  ↳ Verschiedene Varianten sind leicht zustand

  Vermeidet parallele Klassenhierarchie (Factory)

  -> Interface Cloneable
  -> Vorsicht bei zyklischen Referenzen

- Singleton

von einer Klasse soll es nur eine Instanz geben

kontrollierter Zugriff auf Objekt
↳ privater Konstruktor
↳ unterschiedliche Vererbung (begrenzt)
↳ Anzahl Objekte kann zentral erhöht werden

Vererbung: welches Objekt soll erzeugt werden?

Strukturentwurfsmuster

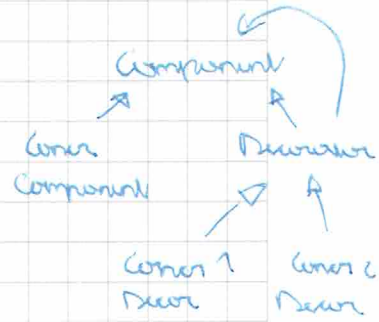
- Decorator

Wrapper, gibt Objekten dynamisch zusätzliche Verantwortung

Verantwortung beeinflusst andere Objekte nicht, kann wieder entzogen werden

vermeidet große Zahl an Unterklassen

Component, Decorator: gemeinsame Oberklasse
↳ Component und dekorierter Component nicht identisch



flexibel, aber schwierig zu warten

Eignet sich gut, erscheinungsbild zu verändern
↳ inhaltliche Veränderung nicht geeignet

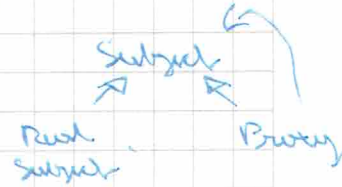
Methoden werden weitergeleitet: Decorator -> Concr Component

- Proxy

Surrogat: Platzhalter für konkretes Objekt, kontrollierter Zugriff
↳ intelligente Referenz auf Objekt

- Remote Proxy: Nachrichten werden über komplexe Kanäle weitergeleitet zB Netzwerkmodul
- Virtual Proxy: Objekt wird bei Bedarf erzeugt
- Protection Proxy: Zugriffskontrolle je nach Instanz, Situation, Nutzungsprofil
- Smart-Reference: Zusätzliche Aktionen bei Zugriff: Kopierung

-> mehrere Proxies möglich



Ähnliche Strukturen wie Dekorator möglich
-> andere Eigenschaften / Zuständigkeiten