1. Kapitel

Paradigma bestimmte Denkweise, Art der Weltanschauung.
Programmierparadigma Programmiersprachen nach Eigenschaften und Stil, der beim Programmieren praktiziert wird einteilen.

Die Eigenschaften:

- Dahinterliegendes Berechnungsmodell
- Struktur von Kontrollfluss (Programmablauf) und Datenfluss
- Problemlösung durch Aufteilung in überschaubareren Teilen

- First Class Entities
- Umgang mit Querverbindungen durch Seiteneffekte

Beliebte Programmierparadigmen:

Imperativ
auf Maschinenbefehlen aufbauend, man gibt Anweisungen.
- *Prozedurales Paradigma*
- *Objektorientiertes Paradigma*

Deklarativ
auf formalen Modellen beruhend, man deklariert das erwünschte Resultat.
- *Funktionales Paradigma*
- *Logikorientiertes Paradigma*

Programmier-Paradigmen

Prozedurale Programmierung
Praktisch in allen imperativen Sprachen. Nicht nur aufeinanderfolgende Befehle.
Algorithmus in überschaubare Teile (Unterprogramm, Routine, Prozedur oder Funktion) zerlegen.
Manche sehen es als Gegenstück zur OOP, weil keine Verkapselung von Prozeduren und Variablen.

Objektorientierte Programmierung
Programmierung mit abstrakten Datentypen, behandelt Daten und Funktionen als eine Einheit -> Objekte als FCEs.

Strukturierte Programmierung
Erweitert prozedurale Programmierung. Verlangt die Beschränkung auf genau drei Kontrollstrukturen:
- Sequenz hintereinander auszuführende Programmanweisungen
- Auswahl/Selektion Verzweigung - if, switch
- Wiederholung/Iteration Schleifen - loop, Rekursion

Sie dient dazu:
- GOTO Anweisung zu vermeiden
- jede Kontrollstruktur bekommt eindeutigen Einstiegs- und Ausstiegspunkt
- mehr Lesbarkeit, aber geringere Flexibilität und Ausdruckskraft (aber Lesbarkeit so stark dass alle verringerte Flexibilität in Kauf genommen haben).

Beschränkt sich nicht nur auf imperative Paradigmen - Auch funktionale und logikorientierte Programmierung nutzen eine Variante davon:
- Schleifen -> Rekursion
- Hintereinanderausführung von Prozeduren -> Komposition von Funktionen

Applikative Programmierung
Programmierung mit Ausdrücken und Funktionen die elementare Werte als Argument und Resultat nehmen.

Funktionale Programmierung
Programmierung mit Ausdrücken und Funktionen die elementare Werte und Funktionen (FCEs) als Argument und Resultat nehmen
-> Funktionen höherer Ordnung / Funktionale.
Man versucht Funktionen höherer Ordnung auch in OOP einzuführen ohne, dass sie referentiell transparent sind. Siehe Streams in Java.

Logikorientierte Programmierung
logische Aussagen die verbunden werden - Prädikatenlogik.

Erfolgsfaktoren eines Paradigmas

Paradigmenerfolg ist beeinflusst durch Programmierwerkzeuge, Syntax, Semantik-Details und *nicht das Berechnungsmodell*.
Die Einflussfaktoren sind:

Kombinierbarkeit und Skalierbarkeit

Bestehende Programmteile sollen sich möglichst einfach zu größeren Einheiten kombinieren lassen.

+ Komposition von Funktionen zur Beschreibung komplexerer Funktionen
+ Objektorientierte Programmierung
- Kombination von Automaten (wird zu komplexer Struktur)

Konsistenz

- Ein Formalismus allein reicht nicht
- Alle Formalismen sollen gut zusammenpassen die für eine Sprache gewählt werden.
  Formalismen sind von Haus aus oft miteinander inkompatibel, deshalb wählt man eine möglichst kleine Anzahl.

Abstraktion

- Höhere Programmiersprachen sollen meistens nicht von Hardware oder (Betriebs-)System-Details abhängen,
  sondern möglichst *portabel sein* und auf möglichst vielen Systemen laufen können.
- Es gibt verschiedene Abstraktionsarten und Abstraktionsgrade

Systemnähe

- Paradigma ist Systemnahe wenn es Hardware Aspekte direkt anspricht, Befehle auf Systemebene möglich sind.
- ↑Systemnähe bedeutet: ↑Effizienz, ↓Portabilität, ↓Sicherheit da direkt auf das System zugegriffen werden kann

Unterstützung

- Von Community: Bibliotheken, Programmiersprachen, IDEs, Tools usw.
- Geld von Investoren, einflussreichen Unternehmen

Beharrungsvermögen

Spannungsfeld: Neue praktische Erfahrungen vs. Widersprüchliche Forderungen an Programmiersprachen innerhalb verschiedener Paradigmen.

Für Wechsel braucht es sehr überzeugende Gründe, wie eine *Killerapplikation* (erfolgreiche Software) die den Erfolg des Paradigmas deutlich macht.

Wichtige Werkzeuge in der Softwareentwicklung sind Compiler und Interpreter.

JIT (Just-In-Time) Übersetzung

- Portabilität und einfache Bedienbarkeit von Interpretern mit Effizienz von Compilern.
- Hat erst Durchsetzung mancher Sprachen ermöglicht

Spannungsfeld: widersprüchliche Ziele in Entwicklung von Programmierparadigmen

Spannungsfeld: Neue praktische Erfahrungen vs. Widersprüchliche Forderungen an Programmiersprachen innerhalb verschiedener Paradigmen.
Es ist *nicht* möglich alle diese Forderungen gleichzeitig und im vollem Umfang zu erfüllen:

1. **Flexibilität** / **Ausdruckskraft**
   Knappe Darstellungsform / Text die alle möglichen Programmabläufe darstellt.
   Eigentlich mit Flexibilität Freiheit gemeint. Möglichst wenige Einschränkungen von Sprache.
   - Eher alte dynamische Programmierung auf Kosten von [2]

2. **Sicherheit** / **Lesbarkeit**
   Inkonsistenzen, Absichten und Sicherheitslücken lassen sich leicht erkennen
   - Eher alte statische Programmierung auf Kosten von [1]

3. **Einfachheit** / **Verständlichkeit**
   - Wird in jüngeren Sprachen für mehr [1],[2] eher vernachlässigt obwohl immer damit beworben wird

Berechnungsmodelle

Berechnungsmodelle müssen sich als Grundlage für eine Programmiersprache innerhalb eines Programmierparadigmas eigenen.
Müssen sich dazu eignen praktische Aufgaben zu lösen.

Ein Berechnungsmodell ist ein Kalkül.
Arbeitet mit Symbolen, Zahlen und Termen. Formale System mit eigenen Axiomen (Regeln).
Mathematische Berechnungsgrundlage (formale Grundlage) für jede Programmiersprache.

Kalkül muss:

- in sich konsistent und vollständig sein sein (widerspruchsfrei)

Zusätzlich erforderlich:

- Turing-vollständig sein (alles berechnen können was mit einer Turingmaschine berechenbar ist)
- Praktisch zu einer Programmiersprache umsetzbar sein.

Formalismen

Die Berechnungsmodelle von Programmiersprachen werden aus diesen Formalismen (Konzepten) entwickelt wenn sinnvoll.

Funktionen

- primitiv-rekursive Funktionen
  Gehen von einer vorgegebenen Menge an einfachen Funktionen aus.
  Komposition von anderen Funktionen oder sich selbst (Rekursion) bildet neue Funktionen.
  Nicht alleine turing-vollständig.
- Turing-Vollständigkeit durch μ-rekursive Funktionen und λ-Kalkül

Prädikatenlogik

- Turing-Vollständigkeit durch μ-rekursive Funktionen und λ-Kalkül
- Mathematisches Werkzeug
- Heutzutage häufig bei Abfragesprachen von relationalen Datenbanken.
- Horn Klauseln und halbautomatisierte Beweise

Constraint-Programmierung

- Das Problem wird deklarativ mittels Einschränkungen beschrieben
- Überall vereinbar außer mit logikorientierter Programmierung

Temporale Logik und Petri-Netze

- Temporale Logik: logische Ausdrücke mit zeitlichen Abhängigkeiten (Aussage gilt vor oder nach Ereignis)
- Petri-Netze: visualisieren zeitliche Abhängigkeiten
- Lassen sich ineinander umwandeln
- Sinnvoll für Nebenläufigkeit

Freie Algebren

- Sinnvoll für Nebenläufigkeit
- stark vereinfacht formulierte Algebren
- ermöglichen Erzeugung von allen beliebigen Strukturen, auch Datenstrukturen
- „universelle Algebra“, Axiome selbst aufstellen

Prozesskalküle

- Sinnvoll für Nebenläufigkeit
- Prozesse vergleichbar mit Threads in nebenläufigen Systemen
- CSP (Communicating sequential processes), Operationen nur zum senden und Empfangen von Daten
- π-Kalkül für reaktive Systeme und λ-Kalkül

Automaten

- Sinnvoll für Nebenläufigkeit
- Automatentheorie, beschreiben nur Syntax, sind aber turing-vollständig
- Visualisierung

WHILE, GOTO …

- Typische Sprachkonstrukte die sich über die Zeit entwickelt haben mit einfachen Operationen aus Assembler
- PRAM (Parallel Random Access Memory) – Sprachen erweitern dadurch, dass mehrere Operationen gleichzeitig auf unterschiedlichen Speicherzellen zugreifen können
- Stark verbunden mit imperativer Programmierung

First-Class-Entities (FCEs) in Paradigmen

- In OOP sind es die Objekte
- In funktionalen Sprachen sind es Funktionen

Einheiten können wie alle Werte der Sprache uneingeschränkt verwendet werden:

- Zur Laufzeit erzeugt werden
- als als Eingabeparameter übergeben und Ausgabe übernommen werden
- in Variablen gespeichert werden

Bringen viele Sonderfälle mit sich die berücksichtigt werden müssen, großer Aufwand bei Umsetzung, deshalb geringe Anzahl an FCEs pro Sprache - Aufwand muss sich rentieren:

- In funktionalen Sprachen: Funktionen nur sinnvoll als FCEs wenn Funktionen höherer Ordnung möglich.

Seiteneffekte und Querverbindungen

Seiteneffekt
Auch Wirkung genannt, bewirkt Veränderung eines Zustandes (Variablenwert).
Jeder Programmfortschritt wird in imperativen Paradigmen über Seiteneffekte erzielt (z.B. Variablenzuweisungen, Ein- und Ausgaben, Zustandsänderungen).

Versteckter Seiteneffekt
Gefährlich, versucht man zu vermeiden.

Beispiel:
f() sortiert ein Array und sammelt statistische Daten. Die sequentielle Ausführung f(x);f(y) wäre kein Problem,
aber f(x);f(x);f(y) würde Array doppelt sortieren und statistischen Daten fälschen.

Querverbindung
Eine Querverbindung ist eine Zustandsänderung (Variablenwert-Zuweisungen) ausgelöst in einem anderen Objekt.
Bei vielen Objekten schwer zu überblicken, deshalb große Gefahr für dadurch erzeugte versteckte Seiteneffekte.

Aufrufhierarchie (auch mit Funktionen möglich) ≠ Querverbindung („Quer" zwischen Objekten)

Umgang mit Querverbindungen durch Seiteneffekte:

funktional	Referentielle Transparenz
OOP	Querverbindungen sichtbar machen

Umgang mit versteckten Seiteneffekten und Querverbindungen in verschiedenen Paradigmen

> Imperative Paradigmen
Befehlsbasiertes Programmieren: Zustandsänderungen
Versteckte Seiteneffekte, Programme sind zustands- und 'zeitbehaftet'

Anweisungen
Anweisungen bewirken Zustandsänderungen (Seiteneffekte).

Wertzuweisungen
Variablen sind Namen für Speicherplätze, beliebig oft mit neuen Werten belegbar (in Schleife oder Rekursion).

Umgang in OOP
versteckte Seiteneffekte möglichst reduzieren durch:

- vollständige Dokumentation
- Grundsätzliche Annahme, dass es immer Querverbindungen gibt.
  Man beschränkt Zustandsänderungen möglichst auf einzelne Objekte (schwache Objektkopplung), damit sie möglichst überschaubar bleiben -> Man muss somit nicht den gesamten Programmzustand überblicken können (prozedual).
- Primitive Operationen (Addition, Division,…) sind immer referentiell transparent.

> Deklarative Paradigmen
Gleichungsbasiertes Programmieren: Werteberechnungen, Wir beschreiben was wir wollen, nicht wie wir es wollen.
Referentielle Transparenz aller Ausdrücke.
Keine Variablen (Zustandslos) und keine Wertezuweisungen (Seiteneffektfrei) - zustandsfrei und 'zeitlos'.

Ausdrücke
Ausdrücke können ausgewertet werden. Das Resultat ist der Wert eines bestimmten Typs. Keine Zustandsänderungen und Seiteneffekte.

Wertvereinbarungen
Variablen sind Namen für Ausdrücke, genau einmal für immer an einen Wert gebunden (keine Speicherzelle).
Späteres Ändern, Überschreiben oder Neubelegen ist nicht möglich.

Referentielle Transparenz

Wenn Ausdruck durch seinen Wert ersetzt werden kann, ohne die Semantik des Programms dadurch zu ändern.

- 3 + 4 darf mit 7 vertauscht werden

Wenn der Wert des Ausdrucks überall im Programm gleich ist. Nicht Abhängig von Zeitpunkt oder Auswertungsreihenfolge.
Nur möglich wenn f seiteneffektfrei ist und nicht von Variablen abhängt deren Werte sich im Laufe der Zeit ändern könnten.

- f(x) + f(x) darf mit 2 * f(x) vertauscht werden (obwohl Bedeutung der Funktion unbekannt ist)

Funktionen in imperativen Paradigmen können referentiell Transparent sein, müssen es aber nicht.

Umgang mit I/O bei referentieller Transparenz

Problem: Wenn eine Ausdruck keine Zustände ändern kann, dann kann man nicht herausfinden was es berechnet hat.
Ein- und Ausgaben sind Seiteneffekte – machen referentielle Transparenz unmöglich.
Sie werden benötigt, aber sind nicht erlaubt.

Funktionale Sprachen
Ein- und Ausgaben nur gut sichtbar ganz oben in der Aufrufhierarchie (für die erste aufgerufene Funktion die das End-Ergebnis zurückliefert) erlaubt, sonst in allen anderen Funktionen nicht.

Ausgabe auf Konsole wird nicht als Seiteneffekt gezählt.

OOP
Ein und Ausgabe von Funktionen sind nicht referentiell transparent.

Modularisierungseinheiten

In OOP. Wir zerlegen das Programm in einzelne Einheiten.
Geringe Abhängigkeit voneinander, leicht gegen einander austauschbar:
-> Programmorganisation, Skalierbarkeit, höchste Flexibilität und Wartbarkeit über lange Zeit, unabhängiges Arbeiten von Entwicklern.

Modul

- *Übersetzungseinheit* - Einheit, die Compiler in einem Stück abarbeitet
- Getrennt übersetzt
- Schnittstellen bestimmen Übersetzungs-Reihenfolge:
  B *importiert* A (kann A aufrufen) -> A muss vor B übersetzt werden.
  Durch getrennte Übersetzung gegenseitiges Importieren nicht möglich, deshalb müssen Abhängigkeiten zyklenfrei sein.
  Man möchte nur Dinge referenzieren die schon übersetzt sind, dafür braucht man Übersetzungsreihenfolge.

  Zum Zeitpunkt der Erzeugung können Objekte nicht zyklisch voneinander abhängen, zuerst werden Objekte unabhängig voneinander erzeugt und erst danach werden die gegenseitigen Referenzen eingesetzt. Zyklen kommen erst nachdem die Objekte erzeugt sind.

- Erst mit *Linker* zur Laufzeit zu ausführbarem Programm verbunden.
  Durch Wiederverwendung von übersetzten Einheiten bei Import schnellere Übersetzung.
  Bei Änderungen nicht alle Module neu übersetzen sondern nur die die voneinander abhängig sind.

- Klassen und Interfaces sowie darin enthaltene statische Variablen und Methoden eigentlich Module (In Java und den meisten Sprachen)
- Datenabstraktion möglich

Modulschnittstellen

Daraus werden Abhängigkeiten (Inhalte, die von außen zugreifbar sind) zwischen dahinterliegenden Modulen hergeleitet (importierte Module werden namentlich genannt).
Wichtig: `module` Dateien in Java sind keine Module sondern Modulschnittstellen

Import und Export

Exportiert:

- Von anderen Modulen zugreifbar, Änderungen wirken sich auf alle die importieren
- Dadurch neue getrennte Namensräume (eigener und aus importiertem Modul) – müssen Eindeutig sein
  Manchmal *Namensraum-Konflikte*: während Import kann man durch Umbenennung oder Qualifikation lösen (Überschreibung von bestehenden Namen)
- Keine Instanzen wie bei Objekten -> jeder eindeutige Name bedeutet ein eindeutiges Verhalten

Privat:

- Nur vom eigenen Modul aus zugreifbar, kann vom Compiler optimiert werden.

Klasse

Existiert zur Übersetzungszeit.
Hat selbst eigentlich keine statischen Variablen und Methoden weil sie ausschließlich als Vorlage zum Erzeugen von Objekten dient.
Die statischen Variablen und Methoden sind Inhalte vom darüberstehenden Modul.

- ein Modul und damit eine Übersetzungseinheit
- Zyklische Abhängigkeiten zwischen Klassen verboten
- Als Vorlage / Bauplan zur Instanzierung von Objekten
  Der Begriff „Klasse“ kommt von der Klassifizierung anhand des Verhaltens:
  alle Objekte mit selben Namen = selben Klasse = gleiches Verhalten
  Deshalb sind Java Interfaces auch in diesem Sinn Klassen. Gleiche Klasse -> Gleiches Verhalten.

  Klassenableitung: Vererbung oder Ersetzbarkeit, dadurch kann ein Objekt mehrere Typen haben.

Objekt
Existiert erst zur Laufzeit, aus einer Klasse erzeugt.
- Keine Übersetzungseinheiten
- Datenabstraktion
- können einander zyklisch referenzieren
- Anders als Module sind sie FCEs
- haben Identität, Zustand und Verhalten (genauso wie Module)
- mehrere Instanzen mit selbem Namen/Verhalten möglich, deshalb differenziert man zwischen Identität und Gleichheit:

  Identisch: Das selbe Abbild im Speicher (mit == verglichen)
  Zwei durch verschiedene Variablen referenzierte Objekte sind identisch wenn es sich um dasselbe Objekt handelt.

  Gleich: Wenn die Parameter gleich sind (mit equals() verglichen).
  Zwei Objekte sind gleich wenn sie denselben Zustand und dasselbe Verhalten haben, auch wenn sie nicht identisch sind. (Kopien)

Komponente (Metadaten, über dependencies, nicht in Java)
- Aus einem Modul wird durch *Parametrisierung* eine Komponente.
  Zeitliche Reihenfolge:
  1. Übersetzung: Im System eingebunden, Festlegung der zu importierenden Quellen - keine gegenseitigen / zylischen Referenzen
  2. Laufzeit, Deployment-Zeitpunkt: Bestimmung von deren Herkunft -> Deshalb zyklische Abhängigkeiten erlaubt.
  3. Verwendung

- Übersetzungseinheit, eigenständiges Stück Software, dass in ein Programm eingebunden wird
- Für sich alleine nicht lauffähig, nicht unabhängig, setzt Existenz anderer Komponenten voraus und nimmt deren Dienste in Anspruch.
- Verschachtelung möglich: Module in Komponenten oder Komponente in Modulen

Namespace / Namensraum (in Java: Package)
- Jede Modularisierungseinheit bildet ein eigenes Namensraum.
  Die Namen müssen darin eindeutig sein. In anderen Einheiten können die Namen aber gleich sein.

  Der Namensraum als Modularisierungseinheit dient aber zur globalen Verwaltung aller Einheiten die im Namensraum stehen.
  Innerhalb eines Namensraums sind Namen von Modulen, Klassen, Komponenten eindeutig.

- Hilfreich gegen Namenskonflikte.
- Ohne globale Namensraumverwaltung müssen alle konkreten Dateipfade benötigter Modularisierungseinheiten angeführt werden – flexibel, aber unsicher.
- Für globale Namen (außerhalb von Packages) ist Entwickler Verantwortlich -> Eindeutige Namen wählen.
- Namensräume fassen Modularisierungseinheiten zu einer Einheit zusammen, hierarchisch organisiert (wie Package in Java oder URL) ohne getrennte Übersetzbarkeit zu zerstören
  Beispiel: a.b.C bezeichnet die Klasse C im Namensraum b welcher im Namensraum a steht.

Schnittstellen in Modularisierungseinheiten

Allgemein
Schnittstellen trennen Außensicht von versteckten Implementierungsdetails (Abstraktion).
Clients (andere Einheiten die auf Schnittstelle zugreifen) können sich auf Schnittstellenstruktur verlassen.
Keine Änderungen notwendig wenn Server (genutzte Einheit) Implementierung ändert.

Schnittstelle von Modulen
- Compiler braucht Schnittstelleninformation für Übersetzungsreihenfolge
- um zu wissen welche Modulinhalte von außen zugreifbar und aufrufbar sind (exportiert) und welche innerhalb des Moduls gebraucht werden (privat)

Schnittstelle von Klassen und Objekten
Durch *Klassenableitung* bei Klassen: Vererbung oder Ersetzbarkeit, dadurch kann ein Objekt mehrere Typen (Schnittstellen) haben.

Parametrisierung

Eigenschaft aller Modularisierungseinheiten: sie lassen sich parametrisieren.
Modularisierungseinheiten Lücken (formale Parameter) lassen, die man später füllt (aktuellen Parameter einsetzen).

- Kann Flexibilität erhöhen aber auch Laufzeit: bei Änderung einer Einheit müssen alle verlinkten Einheiten ebenso geändert werden.
- Kann Wartung erschweren.

Dynamisches Befüllen der Lücken (zur Laufzeit)

Zur Laufzeit gibt es nur Objekte die befüllt werden können. -> Zyklische Referenzen können erst in diesem Schritt eingefügt werden!
Daten sind FCEs und können während Laufzeit erstellt werden. Lücken sind Variablen.

Ermöglicht *Dependency-Injection*:
Die Verantwortung für das Erzeugen und Initialisieren von Objekten wird an eine zentrale Stelle (z.B. eine Klasse) delegiert.
Von der zentralen Stelle kann man die Abhängigkeiten zwischen den Objekten leichter überblicken und steuern.

Konstruktor
Bei Objekterzeugung wird ein Konstruktor mit formalen Parametern ausgeführt, der die Objektvariablen initialisiert.
Nicht nur in OOP Sprachen.

- Statischer Konstruktor Wird nur ein einziges Mal bei Erstverwendung einer Klasse aufgerufen.
- Anonymer Konstuktor Wird nur ein einziges Mal aufgerufen.
- Objekt-Konstruktor Wird bei jeder Erzeugung eines Objekts aus einer Klasse aufgerufen. Initialisiert Objektvariablen.

Reihenfolge der Konstuktor-Aufrufe: statischer-, anonymer-, Objekt-Konstruktor.

Initialisierungsmethode (Setter-Methode)
Setter die nur zum Initialisieren verwendet werden.
Wenn Konstruktoren nicht verwendbar -> Initialisierung nach Erzeugung, nur in imperativen Paradigmen möglich.
zB: Objekte durch Kopieren erzeugt (Aufruf von Konstruktor danach nicht möglich) oder bestehendes Objekt muss zur Laufzeit befüllt werden muss bevor man es verwenden darf.

Zentrale Ablage
Werte an zentraler Stellen (z.B globalen Variablen oder Konstanten, Datenbanken) ablegen, von wo sie bei der Objekterzeugung oder Verwendung benutzt werden. Objekt holt sich die Werte selbst. Zugriff auch durch Vererbung möglich. Auch statisch einsetzbar.

Statisches Befüllten der Lücken (zur Übersetzungszeit)

Nicht alles ist eine FCE, deshalb kann man nicht alles in eine Variable schreiben. Zur Laufzeit kann man Variablen schreiben aber in vielen Sprachen nicht den Typ in eine Variable schreiben.
Problem: Nachträgliches Ändern erschwert.

Generizität
Explizite Lücken vor allem für Typen geeignet.
Alle Arten von Modularisierungseinheiten (außer Objekte) können generische Parameter haben die mit Generizität befüllt werden.
Meistens sind generische Parameter für Konzepte die keine FCEs sind, wie zB Typen – dann Typparameter genannt.

Annotationen
Kenntnis der Lücken nicht Notwendig. Lassen sich an fast allen Programmkonstrukten anheften.
Annotationen beginnen mit einem @-Zeichen und sind optionale Parameter.
Lücken die gefüllt werden beim Setzen werden selbst definiert.
Können Einfluss auf System-Werkzeuge wie Compiler oder Betriebssystem haben (zB Compiler Warnungen) oder ignoriert werden.
Auch dynamisch abfragbar mit Reflexion.

Aspekte (aus AOP)
Querschnittsfunktionalität von Kernfunktionalität getrennt.
Metaprogrammierung: Spezifiziert Punkte im Programm und was dort passieren soll (bestimmten Code ausführen, oder Aufruf durch anderen Aufruf ersetzen) vor der Übersetzung.
Man fügt zu einem bestehenden Programm von außen neue Aspekte hinzu - Die Lücken zum Füllen selbst definiert.
Aspect-Weaver führt es aus.

Zeitliche Abfolge der Füllung
(Übersetzungszeit ist die Zeit, die ein Compiler braucht den Sourcecode zu übersetzen.)

- Generizität: zur Übersetzungszeit
- Annotationen: Übersetzungszeit und / oder Laufzeit
- Aspekte: vor Übersetzung

Typisierung

Typsysteme
Das Typsystem ist der Teil der Programmiersprache welcher die Typisierung (die Überprüfung) durchführt.

Typisierung
Ziel: Vermeidung von Typverletzungen durch Zuweisungsfehler / Erhaltung der Typkonsistenz in der Sprache.
Fast alle aktuellen Programmiersprachen haben Typsystem = sind typisiert.

+ Bessere Planbarkeit, Lesbarkeit, Zuverlässigkeit und Möglichkeit zu abstrahieren weil man weiß welche Typen zu erwarten sind
+ Erstellung von Operationen die nur für Werte bestimmter Typen zugelassen sind
- Einschränkung der Flexibilität weil nicht beliebige Werte und Ausdrücke verwendbar sondern nur die, die vorgegebenen Typen entsprechen

Typkonsistenz und Typverletzungen
Typen sind miteinander konsistent, wenn die Typen der Operanden mit der Operation zusammenpassen (Die Operation auf diesen Typen definiert ist), andernfalls tritt ein Typfehler auf.
Nur wenige Sprachen überprüfen Typkonsistenz nicht - zB Assembler, dann treten Bitmuster Fehler auf.

Auch bei elementaren Typen: wenn Typ von Operand und Operation übereinstimmt

Einteiltung der Typisierungsarten

Strongly vs. Weakly / loosly	stark vs. schwach
Static vs. Dynamic	Kompilierzeit vs. Laufzeit
Manifest vs. Inferred	explizit vs. implizit mit Typinferenz
Nominal vs. Structural	nominal vs. strukturell

Starke / Schwache Typisierung
Abhängig von der Typsicherheit einer Sprache.

Statische / Dynamische Typisierung
Typprüfungen können zur Übersetzungszeit oder zur Laufzeit vorgenommen werden. Beeinflusst Zeitpunkt an dem Fehler entdeckt werden

Dynamische Typisierung	JavaScript, Python und Ruby
Statische Typisierung	Java (hat aber auch dynamische Typprüfungen), C#, Eiffel, Swift, Haskell, TypeScript

Statische Typisierung
Wenn Typ einer Variable bekannt: klar welche Werte in ihr enthalten sein können.
Frühe Entscheidung, welcher Typ verwendet werden soll.

+ Weniger Typfehler zur Laufzeit, bessere Zuverlässigkeit, Fehler zeigen sich früher
+ Bessere Planbarkeit, Verständlichkeit, Lesbarkeit da Information von statischen Typen zuverlässig (wenn explizit deklariert)
+ Typinferenz auch möglich
+ Typ früher bestimmt: mehr Informationen verfügbar, weniger Fallunterscheidungen: verbessert Planbarkeit und Programmoptimierung für Compiler

- Geringere Flexibilität zur Laufzeit. Benötigt Typumwandlungen die unsicher sein können.
- Man kann nicht alles statisch herleiten zb Array-Grenzen. Aber manche Sprachen kommen ausschließlich mit statischer Typprüfung aus zB Haskell.
- Schränkt Sprache ein, mehr Aufwand für Programmierer.
- Dynamische Sprachen möglicherweise sogar sicherer, weil diese besser getestet werden (bei Suche nach Typfehlern werden nebenbei auch andere Fehler erkannt)

Dynamische Typisierung
+ optionale Typdeklarationen möglich oder Typ in den Variablennamen schreiben

Propagieren von Eigenschaften mittels statischer Typisierung
Funktion kann nur aufgerufen werden, wenn der Typ des Arguments == dem des formalen Parameters.
Dafür wird Information über das Argument (den Wert) an die aufgerufene Funktion propagiert.
Es propagiert auch Information von der aufgerufenen Funktion zum Aufrufer wenn man Ergebnistypen verwendet oder einer Variable einen Wert zuweist.

Diese Arten des Propagierens von Information funktioniert nicht nur für Typen, sondern für alle statisch bekannten Eigenschaften: zB Wert ist Integer und Primzahl uvm.

Explizite / Implizite Typisierung

Datentyp kann explizit genannt werden oder per Typableitung (*type inference*) ermittelt werden: wenn Typen im Quellcode nicht explizit hingeschrieben werden, aber der Compiler diese im Rahmen der statischen Typprüfung aus der Programmstruktur ableiten kann.
Dann kann die Deklarierung der Typen optional sein.

Implizite Typisierung

+ Typinferenz spart das Anschreiben von Typen.

- Zur Verbesserung der Lesbarkeit kann und soll man Typen explizit anschreiben -> Explizite Typisierung verbessert Lesbarkeit.
- Typinferenz und Ersetzbarkeit durch Untertypen verträgt sich nicht (im selben Teil des Programms).

Nominale / Strukturelle Typisierung

Struktureller Typ

Nur Signatur bestimmt Typen und die Untertypen-Beziehung, unabhängig davon ob Untertypbeziehung explizit deklariert wurde oder nicht.

gleiche Signatur == gleicher struktureller Typ

Problem: Es können unterschiedliche Einheiten zufällig dieselbe Struktur haben, obwohl sie für unterschiedliche Konzepte stehen – strukturelle Typen unpraktisch!

Lösung: jeder Modularisierungseinheit einen eigentlich nicht verwendeten Inhalt mitgeben, dessen Name das Konzept dahinter beschreibt / die Einheit identifiziert.

Duck-Typing

Bei struktureller und dynamischer Typisierung. Strukturelle Typen werden dynamisch bei Bedarf geprüft.
Dynamische Sprachen wie Python, Objective-C, Smalltalk und Ruby betrachten Klassen als getrennt von Typen, sodass sich strukturelle Typen ergeben.

Nominaler Typ

Man denkt beim Programmieren hauptsächlich abstrakt in Konzepten, nur selten an Signaturen.
Daher verwenden Programmiersprachen zum Großteil nominale Typen statt strukturelle Typen. (in Java, C#, C++, . . .)

Eindeutiger Name der Klasse, von der das Objekt erzeugt wurde bestimmt Typen.
Dadurch dass alle Objekte der selben Klasse, die gleiche Signatur haben: Name und Zusicherung entscheiden ob Untertypbeziehung oder nicht.

Bestandteile eines nominalen Typs
Name der Klasse / Interface
Signatur (= Schnittstelle)
Zusicherungen

Explizite Deklarationen in der Klasse bestimmen Untertypbeziehungen.

Nur wenn gleicher Name == gleicher nominaler Typ
Zwei Objekte können die gleiche Struktur haben aber der Name entscheidet.

Strukturelle und nominale Typen

	strukturell	nominal
Typäquivalenz	gleiche Struktur	gleicher Name
Subtyping	implizit	explizit (Vererbung)
Verwendung	einfach	komplizierter
Plug & Play	besser	schlechter
Namenskonflikte	möglich	leicht möglich
zufällige Beziehungen	möglich	unwahrscheinlich
Lesbarkeit	schlechter	besser
Verhaltensabstraktion	nein	ja

Entscheidungs-Zeitpunkte für Typen

Bedeutende Entscheidungen über Programmstruktur und Programmablauf welche die Typen und Typendarstellung beeinflussen.
Typen verknüpfen zu unterschiedlichen Zeitpunkten vorliegende Informationen miteinander.

Sprachdefinition
- In der Sprachimplementierung z.B. int als 32-Bit Zweierbitkomplement definiert
- Programme können daran nichts ändern.

Erstellung der Übersetzungseinheiten
- Hier werden die meisten wichtigen Entscheidungen getroffen, auf die man beim Programmieren den meisten Einfluss hat.
- Flexibilität entscheidend.

Parametrisierung
- manche Entscheidungen erst durch Parametrisierung bei Einbindung vorhandener Module, Klassen, Komponenten getroffen.
- Unterschied, ob generischer Typparameter oder referentieller Typ (wie Object)
  Bei primitiven Datentypen in dieser Stufe automatisch das Auto-Boxing / Auto-Unboxing.
  Bei abstrakten Datentypen reserviert Compiler Platz für Referenz.

Übersetzungszeit
- Entscheidungen und Optimierungen.
  Eher weniger Bedeutung, alles wichtige bereits im Programmcode festgelegt oder liegt erst zur Laufzeit vor.

Laufzeit
- Zur Laufzeit kann man Initialisierungsphase (bei dynamischer Parametrisierung) von der eigentlichen Programmausführung unterscheiden.

Es zeigt sich folgendes:
Je früher Entscheidungen getroffen werden, desto weniger ist zur Laufzeit zu tun und desto weniger Programmcode (Fallunterscheidungen) braucht man.

Unsichere Entscheidungen werden eher nach hinten verschoben, zu einem Zeitpunkt getroffen, zu dem bereits viele andere damit zusammenhängende Entscheidungen getroffen wurden.

Vor allem statisch geprüfte Typen helfen dabei: Sie verlagern die Entscheidung nach vorne, später daher weniger Fallunterscheidungen (dynamische Typprüfungen) nötig.

Vorteile von Statisch geprüften Typen
+ Ohne wäre mehrfacher Aufwand nötig: Sogar wenn man weiß, dass die Variable eine ganze Zahl enthält, muss Compiler eine beliebige Referenz annehmen und zur Laufzeit eine dynamische Typprüfung durchführen.

+ Helfen getroffene Entscheidungen über gesamten folgenden Zeitraum konsistent zu halten

+ Compiler reserviert je nach Typ einer Variable mehr oder weniger Speicherplatz - je nachdem, wie viel Speicher für diesen Typ nach Sprachdefinition vorgesehen – in Java für jeden primitiven Datentyp bestimmte Anzahl an Bits, für Typparameter so viel wie für Referenz nötig.

Basiskonzepte der objektorientierten Programmierung

Abstraktion, Abstrakte Datentypen (ADT)

Abstraktion
bedeutet, dass der Typ einer Modularisierungseinheit gedanklich für ein Konzept aus der realen Welt steht.
Abstraktion der realen Welt als Grundlage für Ersetzbarkeit: *konzeptuelle Ersetzbarkeit*.
Eine Abstraktion kann aber der informelle Text als Kommentar sein, der Konzept aus der realen Welt beschreibt.

Datenabstraktion
Kapselung + Data Hiding.
Vereinfacht die Repräsentation eines Datenkörpers.

Encapsulation	Zustand (Variablen) und Verhalten (Methoden) zu einer Einheit kapseln
Data Hiding	Trennung der Innenansicht von der Außenansicht einer Modularisierungseinheit. Private Inhalte verstecken.

Abstrakter Datentyp
Nominale Schnittstelle einer Modularisierungseinheit (eindeutiger Name) steht für einen Namen aus der realen Welt.
Gemeinsamer Name für etwas aus der realen Welt und einem Software-Objekt.

Objekteigenschaften

Objekte sind Instanzen von Klassen und Instanzen aller der durch die Klasse beschriebenen Sichten / Schnittstellen / Typen.
Die Klasse selbst ist die spezifischste Schnittstelle mit der genauesten Verhaltensbeschreibung.

Identität (Identity)
Jedes Objekt ist über eindeutige und unveränderliche Adresse identifizier- und ansprechbar.
Vereinfacht: Speicheradresse des Objekts zum anzusprechen (Referenz).

> Vereinfachung stimmt eigentlich nicht ganz: Identität bleibt erhalten wenn sich die Speicher-Adresse ändert, zB Auslagerung in eine Datenbank.

Wenn 2 Objekte nicht denselben Namen haben aber auf dieselbe Speicheradresse zeigen sind sie identisch.

Zustand (State)
Werte der Objektvariablen, verglichen mit equals Methode - Objekte können equal sein auch wenn sie verschiedene Speicheradressen haben. Dann sind sie Kopien voneinander (gleich aber nicht identisch).

Verhalten (Behaviour)
Was Methoden machen: Zur Laufzeit besteht die Software aus einer Menge von Objekten, die einander teilweise kennen und untereinander Nachrichten (Messages) austauschen.

Reaktion eines Objekts beim erhalten einer Nachricht. Ist abhängig von:

1. Der aufgerufenen Methode (Routine)
2. Den übergebenen Parametern
3. Den momentan aktuellen Zustand des Objekts (Variablenwerte)

Alle Objekte einer Klasse haben dieselbe Implementierung und dieselben Schnittstellen. Aber unterschiedliche Objekte einer Klasse müssen nicht gleich sein obwohl die Objektvariablen gleiche Namen und Typen tragen (Zustände können sich unterscheiden).

Verantwortlichkeiten einer Klasse

beschrieben werden durch:

was ich weiß	Zustand der Objekte (Objektvariablen)
was ich mache	Verhalten der Objekte (Methoden)
wen ich kenne	sichtbare Objekte, Klassen, etc. (alle erreichbaren Objekte und Klassen im *Scope*)

Schnittstellen

(Siehe Ersetzbarkeit, Kapitel 2)

Schnittstellen spezifizierbar durch:

Signatur	Methodensignatur ist die Schnittstelle einer einzelnen Funktion. Die Signatur einer Modularisierungseinheit ist aber die gesamte Schnittstelle - die nach außen sichtbare *Struktur*. Beinhaltet Namen und Typen (deklarierter Typ bei Variablen und alle Parametertypen bei Methoden).
Abstraktion	Intuitiv
Zusicherungen	Informelle Verträge, Design by Contract
Protokolle	Formale Verträge, noch nicht etabliert

Typen

Es gibt 2 Arten von Datentypen:

1. elementare / primitive Datentypen können keine Untertypbeziehungen haben
2. referentielle / abstrakte Datentypen können Untertypbeziehungen haben

Alle Datentypen haben Typen.
Bei Polymorphismus (Viel-Typigkeit) haben sie sogar mehrere Typen zugleich. (Die spezifischste und alle Oberklassen)

Typen sind Schnittstellen / Sichten von Objekten, die in Klassen beziehungsweise Interfaces spezifiziert wurden.
Jedes Objekt dass einen Typ implementiert ist eine Instanz des Typs.

Im Skriptum verstehen wir unter der *Klasse des Objekts* immer dem spezifischsten Typ und sprechen allgemein von der Schnittstelle wenn wir einen beliebigen Typen meinen. Es ist mit Schnittstelle nie eine Interface Klasse selbst gemeint.

1. Elementare Typen wie `int` können auch als ADT betrachtet werden:
   Sie abstrahieren über Implementierungsdetails: genaue Repräsentation in der Maschine.
   Berechnungsmodelle arbeiten mit Zahlen, in Programmiersprachen gibt es aber mehrere Typen für Werte (zB Fließkomma, ganze Zahlen etc) - Aber ihre Schnittstelle beinhaltet aber nur einen Wert.

2. Die referentiellen Typen enthalten alle public Variablen und Methoden.

Polymorphismus in objektorientierten Sprachen

Polymorphismus bedeutet, dass eine Variable gleichzeitig mehrere Typen haben darf.

- OOP Sprachen sind polymorph
- Konventionelle statisch typisierte Sprachen (C, Pascal) sind monomorph: Jede Variable oder Funktion hat einen eindeutigen Typ.

In einer OOP Sprache hat eine Variable oder ein formaler Parameter gleichzeitig folgende Typen:

Deklarierter Typ	(explizite) Anfangsdeklaration des Typs
Dynamischer Typ	Der spezifischste Typ, des in der Variablen gespeicherten Werts. → Dynamisches Binden (Compiler kennt dynamische Typen nicht)
Statischer Typ	Wird vom Compiler (statisch) ermittelt und liegt irgendwo zwischen deklariertem und dynamischem Typ. Abhängig von Compiler-Qualität. (Meistens für Optimierungen verwendet).

Gestaltungsmöglichkeiten innerhalb einer Sprache

Rekursive Datenstrukturen
zB für die Beschreibung unbeschränkter Datenstrukturen.
Geht nur mittels *induktiven Konstruktionen* und *Mengenvereinigung*.
Benötigt eine *Fundiertheit* bei Konstruktion die nicht leer sein darf (zB Leafs bei Bäumen, ab denen dann keine weiteren Verzweigungen folgen). Diese bestimmt quasi den Abbruchpunkt der Rekursion.

(Haskell): `data Lst = end | elem(Int,Lst)`

In Java Fundiertheit in Sprachdefinition durch Wert „null“ gegeben -> Nachteil: man muss immer mit null als Wert rechnen, auch bei nicht rekursiven Datenstrukturen.

Propagieren von Eigenschaften
zB int, dass eine Primzahl ist, Objekt, dass nicht null ist usw. damit man nicht bei jedem Funktionsaufruf alle Eigenschaften prüfen muss.

Prozesstypen: Typen die für nebenläufige Programme geeignet sind.

Best Practice – OOP Konzepte

Faktorisierung (Factoring)

Die Zerlegung eines Programms in Einheiten mit zusammengehörigen Eigenschaften - wie man ein Polynom in seine Bruchteile faktorisiert.

Ziel: Lesbarkeit, langfristige Wartbarkeit (Wartungskosten machen ca. 70 % der gesamten Entwicklungskosten aus), vereinfachte Kontrollstruktur, Modifizierbarkeit ohne Funktionalität zu ändern.

- Einsatz des Abstraktionsprinzips
- Gute Faktorisierung -> starker Klassenzusammenhalt -> gut abgeschlossene und somit leicht wiederverwendbare Klassen.
- Refaktorisierung: Änderung einer bestehenden Zerlegung eines Programms in Klassen/Objekte; Funktionalität unverändert, nur Struktur angepasst

Klassen-Zusammenhalt (Class Cohesion)

Grad der Beziehungen zwischen den Verantwortlichkeiten innerhalb der Klasse sollte hoch sein. Nicht messbar sondern nur intuitiv erfassbar. Dadurch gut abgeschlossene und somit leicht wiederverwendbare Klassen.

Hoch wenn alle Variablen und Methoden:

- eng zusammenarbeiten
- durch den Namen der Klasse gut beschrieben sind
- durch das Entfernen einer einzigen die ganze Klasse beeinflussen

Test:

Entfernen von Variablen und Methoden oder Umbenennung: Wenn Klasse nicht mehr funktioniert → guter Zusammenhalt

Ein hoher Klassen-Zusammenhalt deutet auf eine gute Zerlegung (Faktorisierung) des Programms in einzelne Klassen beziehungsweise Objekte.

Objekt-Kopplung (Object Coupling)

Abhängigkeit der Objekte voneinander sollte niedrig sein.

Hoch wenn:

- Viele Methoden und Variablen nach außen sichtbar
- einzelne Objekte häufig miteinander kommunizieren, und dabei viele Parameter übergeben

Schwache Objekt-Kopplung deutet auf gute Kapselung hin, bei der Objekte voneinander so unabhängig wie möglich sind.

> Wenn Klassenzusammenhalt hoch, dann meistens Objektkopplung niedrig und vice versa.

Deshalb:
Immer nur `private` als Modifier benutzen und möglichst getter und setter Methoden vermeiden und alles klassenintern übergeben.

Entwicklungsprozesse

Traditionell: Entwicklung nach dem Wasserfallmodell

1. Analyse
2. Design
3. Implementierung
4. Verifikation und Validierung (Feedback am Ende -> Gefährlich)

Besser: zyklische Prozesse wo man früh Feedback holt und iterativ verbessert.

Paradigmenwahl

Wiederverwendung

Ziel: Wiederverwendung von …

- bewährtem Code wichtig um Aufwand zu ersparen
  Konzepte von Untertypen, Vererbung und Generizität wurden für Code Wiederverwendung eingesetzt
  Bibliotheken, Projektinterne- oder Programminterne Wiederverwendung.
- Erfahrung (Entwurfsmuster)

Programme	werden oft genau für häufige (wieder)verwendung entwickelt
Daten	oft überdauern Daten die Lebensdauer der Programme.
Erfahrungen	können zwischen sehr unterschiedlichen Projekten ausgetauscht werden.

Code-Wiederverwendung erfordert braucht große Investitionen in die Wiederverwendbarkeit aber es kann sich manchmal langfristig rentieren.

Wartbarkeit

Da Wartungskosten ca. 70 % der Gesamtkosten ausmachen. Gute Wartbarkeit erspart viel Geld insbesondere, wenn Programm langen Lebenszyklus hat, lange verwendet wird. Langfristig Vorteile von robustem Programm am besten bemerkbar.

Richtige Paradigmenwahl

Paradigmenwahl beeinflusst Entwicklung:

Objektorientierte Programmierung

- Entwicklung von Systemen, deren Gesamtkomplexität jene der einzelnen Algorithmen deutlich übersteigt
- Wiederverwendung von Code mit Untertypen und Ersetzbarkeit
- Datenabstraktion
- Wartbarkeit in langem Einsatz-Zeitraum

Prozedurale, funktionale und logikorientierte Programmierung

- Algorithmen (in OOP ungeeignet da aufwändig über mehrere Objekte verteilt -> Erhöht Entwicklungsaufwand und senkt Verständlichkeit)
- Einmaliger Einsatz von Software
- Schnelles Programmieren

2. Kapitel

Parameterarten

Für gute Wartbarkeit: Parametertypen vorrauschauend und möglichst allgemein wählen.

Formale Parameter:
Als Teil der Funktionsdefinition im Programmcode, denen man Werte zuweist:

```
Ergebnistyp funktionsname ( <Par1>, <Par2>, ...)
```

Tatsächliche Parameter:
Auch „aktuelle Parameter“ oder „Argument“ genannt: der jeweilige Wert für `Par1` oder `Par2` bei einzelnen Funktionsaufrufen.

Methodenparameter können formale oder tatsächliche Parameter sein. Man unterscheidet zwischen:

Eingangsparameter (in Java hat man nur Eingangsparameter)
Von Aufrufer zur aufgerufenen Methode

Ausgangsparameter (nicht in Java, da hat man nur genau ein Methodenergebnis)
Am Methodenende: von aufgerufener Methode zu Aufrufer
Gleich wie Ergebnistyp aber in Form eines formalen Parameters.
Aufrufer übergibt eine Variable in der am Ende der Wert stehen soll.

Durchgangsparameter
gleichzeitig Ein- und Ausgangsparameter
Aufrufer übergibt einer Variable mit Wert die überarbeitet wird und zurückgegeben wird.

Untertypenbeziehungen (Subtyping)

Nur in OOP. Man unterscheidet zwischen nominalen und strukturellen Untertypbeziehungen.
Es muss die Erwartung erfüllt werden, dass eine Einheit die andere ersetzen kann.

Untertyp muss alles bereitstellen was vom Oberstypen erwartet wird, darf mehr Details festlegen als Obertyp (also Obertyp erweitern).
Untertyp ist spezifischer, Obertyp ist allgemeiner.

Untertypen werden durch das Ersetzbarkeitsprinzip definiert. Ohne Ersetzbarkeit gibt es keine Untertypen:

> **Ersetzbarkeitsprinzip**
> *Ein Typ U ist Untertyp eines Typs T, wenn jedes Objekt von U überall verwendbar ist, wo ein Objekt von T erwartet wird.*

Anwendungsgebiete

- Dynamisches Binden. (Wenn ein Objekt an einer Variable zugewiesen wird und der Objekttyp (spezifischster Typ) ein Untertyp des deklarierten Typs der Variable ist).
- Aufruf einer Methode mit einem Argument, dessen Typ ein Untertyp des Typs des entsprechenden formalen Parameters ist.
- Indirekte Code-Wiederverwendung, Typhierarchie mit verschiedenen Schnittstellen aufstellen
- Möglichkeit zur einfachen nachträglichen Änderung von Modularisierungseinheiten verbessert Wartbarkeit, Planbarkeit

Allgemeine Eigenschaften von Untertypbeziehungen

Reflexivität	Jede Klasse ist Untertyp von sich selbst
Transivität	[a -> b -> c] bedeutet [a -> c]
Antisymmetrie	wenn [a -> b] ist [b -/-> a]

Stabilität an der Wurzel: Bewahrung der Abwärts-Kompatibilität bei Untertypenbeziehungen
Schnittstellen und Typen sollen deshalb stabil bleiben, weil die Änderung eines Typs weiter oben in der Typenhierarchie, der viele Untertypen besitzt, auch dazu führen kann, dass alle Untertypen geändert werden müssen bzw das Ersetzbarkeitsprinzip plötzlich nicht mehr gilt. Auch bei Zusicherungen.

Überprüfung der Untertypenbeziehung durch Schnittstellen

Es kann zB bei Klassenableitung zu ungeplanten Untertypenbeziehungen kommen. Deshalb ist es wichtig zu überprüfen ob die bestehenden Untertypbeziehungen richtig sind.
Compiler ist nicht in der Lage alle Bedingungen für Ersetzbarkeit zu überprüfen, deshalb:

1. Signatur
Statisch prüfbar. Einfachste Form der Schnittstelle, bestimmt welche Inhalte der Modularisierungseinheit von außen zugreifbar sind: Namen von Inhalten, Typen von Parametern.

> Problem: Bedeutung der Inhalte bleibt offen.
> Namen in der Signatur in einer Schnittstelle sagen nichts über die Struktur (Implementierungsdetails) aus, deshalb sind sie alleine nicht ausreichend.
> Gefahr: Sie können gleich heißen aber Unterschiedliches ausführen. -> draw(), Zeichnen oder Revolver ziehen
>
> Problem: Zufällig gleiche Signatur (= struktureller Typ) möglich

2. Abstraktion realer Welt
Eine Abstraktion ist ein informeller Text, der zusätzlich zur Signatur die Modularisierungseinheit beschreibt.
Bei der *konzeptuellen Ersetzbarkeit* (reale Welt als Grundlage der Abstraktion), sind Zusicherungen besonders wichtig.
Dieser Ansatz ist gut aber beruht nur auf Intuition – Leute haben unterschiedliche Vorstellungen – deshalb gefährlich

3. Zusicherungen / Erwartungen
Genaue Beschreibung der erlaubten Erwartungen an eine Modularisierungseinheit (*Design by Contract*).

4. Überprüfbare Protokolle
Formale Beschreibungen der Erwartungen, die bereits Compiler auf Konsistenz überprüfen kann wenn sie nicht zu kompliziert sind.
Können Beziehung zwischen Client und Server (Client Server Protokolle) oder mehreren Einheiten gleichzeitig regeln.

Nominale Typen und Ersetzbarkeit (Nominal Subtyping)

Basiert mehr auf Intuition und Abstraktion der echten Welt, man muss nicht immer die Signaturen im Kopf behalten -> Intuitiver.
Sie können nicht automatisch verglichen werden um Untertypbeziehungen aufzuzeigen, weil nominale Typen auf abstrakten Konzepten aufbauen und mit Zusicherungen und Kommentaren beschrieben werden.

Deklarieren Obertyp im Untertypen explizit, Untertyp muss vom Obertyp abgeleitet sein.
Dadurch dass alle Objekte der selben Klasse, die gleiche Signatur haben entscheiden Name und Zusicherung ob Ersetzbarkeit möglich ist.

Bestandteile eines nominalen Typs

Name der Klasse / Interface
Signatur (gesamte Schnittstelle)
Zusicherungen

Strukturelle Typen und Ersetzbarkeit (Structural Subtyping)

Signatur der Einheit bestimmt den Typen und die Untertypbeziehung.
Unabhängig davon ob Untertypbeziehung explizit deklariert wurde oder nicht.

Ersetzbarkeitsregeln

Dadurch dass alle Objekte der selben Klasse, die gleiche Signatur haben und bei nominalen Typen es bereits im Voraus gesichert ist, dass die Signaturen übereinstimmen betrachtet man in theoretischen Modellen eher die strukturellen Typen um die Voraussetzungen zu definieren die die Signaturen erfüllen müssen.

Nehmen wir an:

[T <- U]	U Untertyp von T
(T (A))	Etwas in Obertyp T vom deklarierten Typ A
(U (B))	Etwas in Untertyp U vom deklarierten Typ B

Für jede Variable in T muss es eine in U geben wobei [A <-> B] (Invarianz)
Für jede Konstante in T muss es eine U geben wobei [A <- B] (Kovarianz)
Für jede Methode in T muss es eine in U geben mit gleichen Namen, gleicher Parameteranzahl, Exceptions und Methoden Ein- und Ausgaben.
bei Parametern muss man zusätzlich unterscheiden ob Lese oder Schreibzugriff.

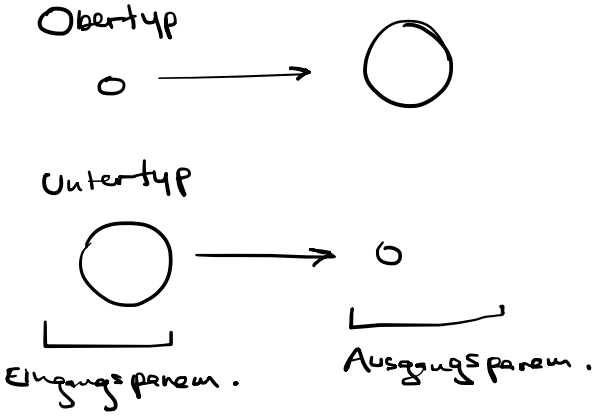
Verallgemeinert:

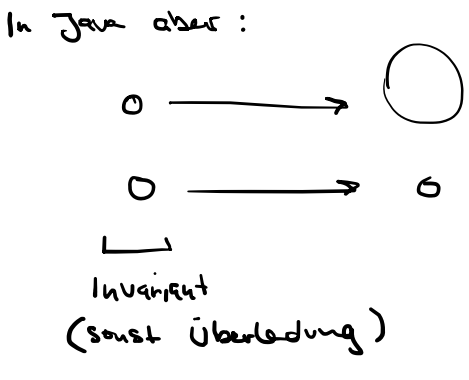
[A <- B] Kovarianz – bei Lesezugriff
Der deklarierte Typ eines Elements im Untertyp ist ein Untertyp des deklarierten Typs des entsprechenden Elements im Obertyp.
> Konstanten, Ergebnistypen, Ausgangsparameter, Ausnahmen

[A -> B] Kontravarianz – bei Schreibzugriff
Alles was im Typ A geschrieben werden kann muss auch im Typ B geschrieben werden können.
Der deklarierte Typ eines Elements im Untertyp ist ein Obertyp des deklarierten Typs des Elements im Obertyp.
> Eingangsparameter

[A <-> B] Invarianz – bei Lese und Schreibzugriff
Der deklarierte Typ eines Elements im Untertyp ist äquivalent zum deklarierten Typ des entsprechenden Elements im Obertyp.
(Wichtig: Alles was Invariant ist, ist auch Kovariant)
> Variable, Durchgangsparameter

Damit Ersetzbarkeit gewährleistet – In Methoden:





Einschränkungen in Java und ähnlichen Sprachen

Java (benutzt nominale Typisierung) ist strenger als es eigentlich sein müsste:
Ergebnistyp kovariant, alle anderen Typen invariant, damit unterscheidbar vom Überladen.

Auch wenn wir theoretisch kontravariante Eingangs-Parametertypen haben könnten gibt es sie nicht.
Damit wir Überladen von Überschreiben unterscheiden können.

Beispiel: Angenommen variante Parametertypen erlaubt und strukturelle Typen - nicht in Java

```
class T {
        public T method(U p) { ... }
}
class U {
        public U method(T p) { ... }
        public void foo() { ... }
}
```

Hier wurde korrekt abgeleitet:
- Der Ergebnistyp ist kovariant verändert
- Der Eingangsparametertyp ist kontravariant verändert

Wäre die Methode `foo` in U nicht vorhanden, dann wären U und T sogar äquivalent, da sie die gleiche Struktur hätten und die Unterklasse die Oberklasse nicht erweitern würde.

In Java würde (nach expliziter Deklaration der Untertypbeziehung) die Methode `method` nur überladen werden (da Eingangsparameter nicht Invariant).

Einschränkungen durch Typisierungsart

Einschränkungen durch statischer Typisierung

Typen von Eingangsparametern dürfen in Untertypen nicht spezifischer werden. Ansonsten Kovarianz und nicht Kontravarianz.

> Siehe kovariantes Problem. Wird in der Praxis häufig benötigt, gibt aber keine Möglichkeit das statisch zu prüfen. Dynamische Prüfungen aber möglich. (Man kann mit Typabfragen kovariantes Problem lösen)

Beispiel:

```
boolean compare(T x)
```

darf nicht überschrieben werden zu:

```
@Override
boolean compare(U x)
```

Aus demselben Grund sind auch Wertebereichseinschränkungen im Allgemeinen nicht statisch prüfbar -> das bedeutet z.B., dass int nicht als Untertyp von long betrachtet werden kann, obwohl jede Zahl in int auch in long vorkommt.

Einschränkungen durch impliziter Typisierung

Typinferenz und Ersetzbarkeit / Untertypenbeziehungen im selben Teil des Programms inkompatibel - Aber verträgt sich gut mit Generizität.

Ersetzbarkeit für Code-Wiederverwendung

Untertypbeziehungen sind wichtig um Code Wiederzuverwenden.

Stabile Schnittstellen Wurzel und dynamische Bindung

Beispiel:

Man hat eine Schnittstelle vor Jahren definiert für Drucker-Treiber und hat jetzt neuere sicherere Treiber implementiert die aber sowohl mit den alten als auch neuen Druckern kompatibel sein sollten. Dafür kann mit dem selben Treiber mehrere Schnittstellen (Interfaces) implementieren oder das Interface „neuer Treiber“ erbt von „Alter Treiber“

- *Mehrfachvererbung* ist mit Interfaces in Java möglich.
- Dynamisches Binden wegen Ersetzbarkeitsprinzip möglich.
- Struktur ist am stabilsten wenn Wurzel möglich selten verändert wird und idealerweise ein Interface ist.

Der spezifischste Typ

Beispiel:

```
public class A {
        public String foo1() { return "foo1A"; }
        public String foo2() { return  fooX(); }
        public String fooX() { return "foo2A"; }
}
public class B extends A {
        public String foo1() { return "foo1B"; }
        public String fooX() { return "foo2B"; }
}
public class DynamicBindingTest {
        public static void test(A x) {
                System.out.println(x.foo1());
                System.out.println(x.foo2());
        }
public static void main(String[] args) {
                test(new A());
                test(new B());
        }
}
```

Die Ausführung von DynamicBindingTest liefert folgende Ausgabe:

```
A.foo1() -> foo1A
A.foo2() -> foo2A
B.foo1() -> foo1B
B.foo2() -> foo2B
```

Beim letzten Aufruf „foo2B“ wird B.foo2() aufgerufen --> A.foo2() --> fooX()
An diesem Punkt wird *fooX() von B bevorzugt*. Es wird der *spezifischste Typ der Umgebung* gewählt.

Ersetzbarkeit muss Verhalten berücksichtigen:
Verhalten ist etwas abstraktes, dass zwischen Implementierung und Signatur liegt.
Es gibt verschiedene Verhaltensbeschreibungen mit ganz unterschiedlicher Genauigkeit.

Zusicherungen: Design by Contract

Beschreiben die erlaubten Erwartungen an (die Schnittstelle einer) Modularisierungseinheit.
Schnittstelle als Vertrag zwischen Modularisierungseinheit (Server) und Verwendern (Clients).
Vertrag beschreibt Verantwortlichkeiten.

Zusicherungen werden als Teil des nominalen Typs betrachtet, in Java nur durch Kommentare auszudrücken.
Weil sie als Teil des Objekttyps gelten müssen sie die Bedingungen der Ersetzbarkeit erfüllen. -> stabil sein, vor allem an der Wurzel

Vorbedingungen – Pre Conditions
Vor Methodenaufruf.
Was darf sich Server von Client erwarten?
@param, „Welche Eigenschaften sollen die Eingangsparameter haben?“

Im Untertyp sind sie schwächer oder gleich
Bei Vererbung logische OR-Verknüpfung

Nachbedingungen – Post Conditions
Vor Rückkehr aus Methode.
Was darf sich Client von Server erwarten?
@result, „Was sagt das Ergebnis aus?“ Beschreibt Methodenergebnis / Änderungen des Objektzustands

Im Untertyp sind sie stärker oder gleich
Bei Vererbung logische AND-Verknüpfung

Invarianten – Invariants
Welche Eigenschaften sind in Zuständen erfüllt?
Gelten für alle Objektvariablen, Zuständigkeit des Servers (aber auch Client wenn nicht private).

Im Untertyp sind sie stärker oder gleich -> Wenn Variable von außen sichtbar dann gleich
Bei Vererbung logische AND-Verknüpfung (wenn nicht gleich)

History-Constraints
In welchen Aufrufreihenfolgen dürfen Clients mit Server interagieren?
Schränken die Entwicklung von Objekten oder Objektvariablen im Laufe der Zeit ein.

Server-kontrolliert: Reihenfolge der Zustandsänderungen.
Ähneln Invarianten (z.B. Wert einer Variable darf nur größer werden.
Im Untertyp sind sie stärker oder gleich -> Wenn Variable von außen sichtbar dann gleich.
(Reihenfolge der Zustandsänderung der Variablen im Untertyp)

Client-kontrolliert: Reihenfolge der Aufrufe (z.B. Methode "Initialize" kann in jedem Objekt nur einmal aufgerufen werden)
Wichtig für Synchronisation bei Nebenläufigkeit.
Im Untertyp sind sie schwächer oder gleich -> Einschränkung bezogen auf Reihenfolge von Methodenaufrufe (Traces bzw. Trace-Set): Trace-Set(T) ⊆ Trace-Set(U)
U kann mehr Aufrufreihenfolgen erlauben. -> Mit allen Clients, nicht nur einzelnen.

Wichtig: schwächer bedeutet weiter oben in Typhierarchie, stärker bedeutet weiter unten in Typhierarchie
- Schwächer oder gleich stark eingeschränkt: Kontravarianz -> Client verantwortlich – wie bei Eingangsparameter
- Stärker oder gleich stark eingeschränkt : Kovarianz -> Server verantwortlich – wie bei Ergebnistypen
- Gleich stark eingeschränkt: Invariant -> Server und Client verantwortlich

Aufrufer beim Aufruf eines Untertypen muss weniger oder gleich viel einzuhalten (lockerer), aber alle anderen Erwartungen müssen erfüllt werden wie beim Obertyp oder noch mehr umfassen (strenger sein).

Guter Stil:
- Schwache Client-Server-Abhängigkeit -> Wenige Zusicherungen, weniger Details in Zusicherungen, dadurch „ungenauer“
- Minimale Kommentaranzahl, aber vollständig

Vererbung

Untertypbeziehungen durch Ersetzbarkeit und Vererbungsbeziehung sind Konzepte der *Klassenableitung*.
Nicht in allen Sprachen beide möglich.

BASISKLASSE	→	ABGELEITETE KLASSE
Base Class		Derived Class
Ober Klasse		Unter Klasse
Super Klasse		Sub Klasse

Bei der Vererbungsbeziehung werden Inhalte der Oberklasse kopiert.
Dadurch wird die gesamte Schnittstelle des Obertypen wie bei der Untertypbeziehung implementiert und kann erweitert werden.
Aber dadurch, dass Inhalte kopiert werden, ist eine Änderung der Implementierung aus der Oberklasse nur durch Überschreibung möglich.

Erweiterung zusätzliche Variablen, Methoden und Konstruktoren
Überschreibung @Overwrite

Java Einfachvererbung / *Single Inheritance* - höchstens 1 Superklasse pro Klasse
C++ Mehrfachvererbung / *Multiple Inheritance* - gilt auch für Interfaces in Java

Überschreibung und Zugriff auf Superklasse (in Java)

Bei Überschreibung kann man mit `super.` innerhalb der Unterklasse auf die Oberklassen-Methode und Variable zugreifen aber nur wenn `public`.
Auch Zugriff auf Konstruktor von Superklasse: `super(a,b,c);` (wird sonst leer gefüllt).
Zugriff nur über einer einzigen Vererbungsebene.

Casting bei Variable:
`((Oberklasse) this).v.`

Casting bei Methoden:
Würde nicht funktionieren.
`((Oberklasse) this).method(…)` würde nur deklarierten Typ ändern und andere Funktionen bei Überladung aufrufen (statisch gebunden), aber es würde Methoden vom dynamischen Typen aufrufen (dynamisch gebunden).

Code-Wiederverwendungsarten

Arten von Klassenbeziehung

1. Untertypbeziehung
2. Vererbungsbeziehung
3. Reale-Welt-Beziehung / Is-a-Beziehung (≠ Untertypbeziehung/Vererbung, aber oft zu Untertypbeziehung weiterentwickelbar)

Unterschiede zwischen Vererbungs- und Untertypenbeziehung

Die Untertypbeziehung basiert auf der Vererbung.
Sowohl bei der Vererbungs- als auch bei der Untertypbeziehung wird die Schnittstelle des Obertypen implementiert und kann erweitert werden.

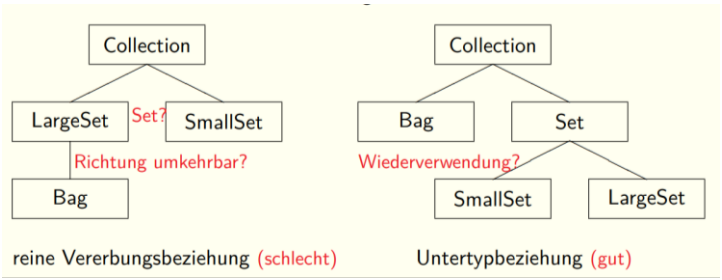
Bei reiner Vererbung ist aber die Ersetzbarkeit nicht gegeben und Änderungen des Obertyps sind nur mit Überschreibung möglich.
Der Compiler kennt keinen Unterschied zwischen Vererbung und dem Ersetzbarkeitsprinzip (weil es keine Zusicherungen prüfen kann).

Vererbung	direkte Codewiederverwendung (leicht sichtbar)
Untertypbeziehung	indirekte Codewiederverwendung durch Ersetzbarkeit (nur schwer sichtbar) weniger direkte Codewiederverwendung als Vererbung (da Zusicherungen zu berücksichtigen) -> man kann nicht einfach alles übernehmen

Indirekte Codewiederverwendung langfristig wichtiger als direkte.

Unterschiede zwischen Vererbungsbeziehungen

- Reine Vererbungsbeziehung — Ziel: reine Wiederverwendung der Implementierung
- Vererbungs- und Untertypbeziehung — Ziel: Typenhierarchie (meistens von is-a Beziehungen) und Vererbung



Nachteile überwiegen den Vorteil einen Code-Anteil nicht erneut schreiben zu müssen.

Auch wenn es auf den ersten Blick so wirkt als würde reine Vererbung eine höhere Codewiederverwendung erzielen, so ist durch das Nichtvorhandensein des Ersetzbarkeitsprinzips:

- die Wartung wesentlich aufwändiger
- keine Möglichkeit dynamisch zu binden und andere Vorteile der Ersetzbarkeit auszunutzen

Besondere Klassen

Final Classes

Mit `final` kann man verhindern, dass Methoden in überschrieben werden (final Methoden) oder Klassen erben können (final Klassen).
In manchen Programmierstilen in denen man bewusst den Einsatz von Vererbung minimieren will sehr beliebt.

(`templateMethod()` ist final)

Abstrakte Klassen

Dienen nur zur Definition einer stabilen Schnittstelle. Keine Instanzen.
Hauptvorteil zum Interface ist die Code-Wiederverwendung - Code enthalten, dass an konkrete Klassen vererbt wird (aber Interface stabiler).

Stabiler als konkrete Klassen, enthalten vollständig implementierte und *abstrakte Methoden* die nicht implementiert werden.
Ermöglichen auch Patterns wie „Template-Method“.

Interfaces

Interfaces sind stabiler als Klassen und immer diesen vorzuziehen.
Mehrfachvererbung möglich.
Zusicherungen am wichtigsten unter allen Java Klassen.

Erlauben nur `static final` Variablen. Mit `default` und `static` können auch Methoden in Interfaces implementiert werden.

```
public interface Z{
        static final double x = 3.0;
        default double fooZ() {
                return fooX() + fooY();
        }
}
```

Die Methoden by default immer public - Deshalb muss man es nicht immer extra dazuschreiben.

Nested Classes – Innere Klasse

- Gehört zu einem Objekt der umschließenden Klasse
- Alles nicht-statische von umschließender Klasse direkt von innen zugreifbar auch wenn private
- Darf keine static nested Klassen enthalten, da die Klasse selbst von Objekt abhängig – Es können innere innere Klassen nicht statisch sein.

- Neue Instanz: `a.new InnerClass()`
  wobei a eine Instanz von `EnclosingClass` ist. Ganz wie gewohnt aufrufbar.
  Bisschen ineffizient: Alle Inhalte der umschließenden Klasse bekommen eine zusätzliche (versteckte) Referenz

```
class EnclosingClass {
        class InnerClass { ... }
}
```

Static Nested Classes – Geschachtelte statische Klasse

- Gehört der umschließenden Klasse
- Alles statische von umschließender Klasse direkt von innen zugreifbar auch wenn private

- Neue Instanz: `new EnclosingClass.StaticNestedClass()`

- `static nested class` muss `public` sein, damit man von außen auf Klasse zugreifen kann, da als `EnclosingClass.StaticNestedClass` interpretiert. Verzeichnis ähnliche Struktur.

```
class EnclosingClass {
        static class StaticNestedClass { ... }
}
```

Probleme in Java mit allen Nested Classes
weil in späteren Java Versionen eingeführt:

- `private` modifier in `static nested Class` und `nested Class` keine Wirkung, Zugriff von umschließender Klasse überall möglich
- Obwohl private modifier in nested Class nicht funktioniert, nicht darauf zugreifen.

Am nützlichsten ist die Innere Klasse zB für Iteratoren, wenn man von der Nested Class auf die Enclosing Class zugreift und nicht umgekehrt.

Anonyme innere Klasse und Lambda-Ausdruck

Innere Klasse von der wir nur ein einziges Objekt erzeugen wollen: anonyme innere Klasse als Lambda-Ausdruck.

```
new T(...) { // Parameter für Konstruktor falls T Klasse
...          // Implementierung eines Untertyps von T
}            // Ergebnis ist Objekt des Untertyps von T
```

Functional Interface
(Erst seit Java 8)
Interface mit genau <u>einer</u> abstrakten Methode -> Lambda Ausdruck ist quasi vereinfachte Syntax für Objekt von functional Interface.

```
T x = (a, b) -> a + b; // wobei T Functional Interface
```

Die Klasse soll nur genau eine Methode haben:
entspricht für Methode `int f(int x, int y)` in T

```
new T() {
        int f(int a, int b) { return a + b; }
}
```

Pakete (Packages) in Java

- Paket-Name = Name des Verzeichnisses
- Jedes Paket ist ein eigener *Namespace*
- Verzeichnis mit Quellcode (.java) und compiliertem code (.class). Beide müssen gleich heißen.
- Es darf aber nur eine public class in jeder datei sein.
- Bei geschachtelten Ordnern braucht man explizite packages. (komplizierter)
- Default-Paket ist die Basis der *Verzeichnissturktur* / *Class Path*

Beispiel:
Angenommen wir haben die Methode `foo();` in `myClasses/examples/AClass.java`

Import-Anweisung: importiert Paket/Klasse aus einem Paket. Nun muss Classpath zum Paket/zur Klasse nicht immer explizit angeben werden. Es lassen sich nur public Klassen importieren.

Aufruf ohne Import:
```
myClasses.examples.test.AClass.foo();
```
Aufruf mit Import:
```
import.myClasses.examples.test;
test.AClass.foo();
```
beziehungsweise
```
import.myClasses.examples.test.AClass;
import myclasses.test.*;
AClass.foo();
```

Vor dem import darf nur stehen:
```
package PFAD_ZU_PACKAGE
```

Sichtbarkeit in Java (modifier, Scope)

Scope nur bei Bedarf ausweiten aber idealerweise alles private halten.

`public` überall sichtbar, vererbbar

`protected` in anderem Paket nicht sichtbar, aber zu einem Untertyp in einem anderen Paket vererbbar
Wichtig: `Protected` -> nur sichtbar in der Package und in den Unterklassen -> schlechter Stil!

`default` Wie protected, aber nur im eigenen Paket vererbbar.

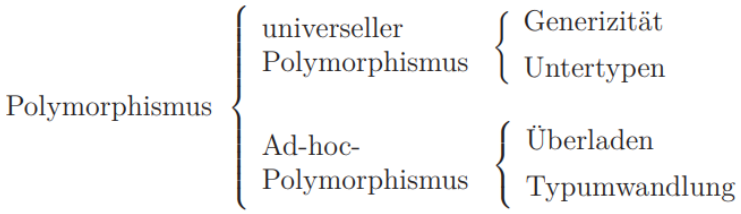
`private` Nur innerhalb der Klasse sichtbar. Nirgends vererbbar.

	public	protected	(default)	private
lokal sichtbar	ja	ja	ja →	nein
global sichtbar	ja	nein	nein	nein
lokal vererbbar	ja	ja	ja →	nein
global vererbbar	ja	ja →	nein	nein

lokal = im selben Paket, auch außerhalb der Klasse
global = auch außerhalb des Pakets

Mit vererbbar ist *Sichtbarkeit / Zugreifbarkeit* nach Vererbung gemeint. -> Bei Vererbung werden auch private Inhalte vererbt, die nicht zugreifbar sind.
lokal: im selben Paket
global: im anderen Paket

3. Kapitel
Polymorphismus
Es gibt verschiedene Arten von Polymorphismus:



1. Universelle Polymorphismen (spezifisch für OOP)
Typen die zueinander in Beziehung stehen haben die gleiche Struktur -> bei Ad-hoc nicht

Generizität: **„parametrischer Polymorphismus“**
Gemeinsamer Code, der über Typparameter mehrere Typen haben kann.
Nicht ausschließlich in OOP, sondern allen mit statischer Typprüfung - Einschränkungen auf Typparameter lassen sich so prüfen.

- *F-gebundene Generizität*
  Beschreibung von Einschränkungen mit Untertypbeziehungen (Java, C#)

- *Higher Order Subtyping / Matching*
  Beschreibung von Einschränkungen mit Untertyp-ähnlichen Beziehungen (unterscheiden sich aber) -> Direkterer Weg. Wäre andere Art der Typbeziehung die in Java ausgelassen wurde, weil es sonst zu Verwirrung führen würde. (C++, Haskell)

Untertypbeziehungen: „Enthaltender Polymorphismus“
Gemeinsame Schnittstellen für unterschiedliche Objekte - In OOP am wichtigsten.
Deshalb alles was mit Untertypen zu tun hat: meist *Objektorientierter Polymorphismus*, kurz: Polymorphismus.
Auszuführende Methoden werden erst bei der Programmausführung festgestellt (dynamisch).
Stichworte: deklarierter, dynamischer, statischer Typ, Ableitung von Klassen.

2. Ad-hoc Polymorphismen
keine strukturellen Ähnlichkeiten vorausgesetzt.

Überladen
Methoden mit gleicher Funktionalität und Namen, aber verschiedenen (Anzahl von) Parametern.

Typumwandlung
Umwandlung eines Wertes in ein Argument welches die Routine erwartet.
Casting / Typumwandlung auf elementaren Typen (fast keine Ähnlichkeiten in interner Darstellung).

Generizität

Verbessert: Wartbarkeit, Sicherheit, Wiederverwendung, Lesbarkeit

Untertypbeziehungen vs. Generizität

Eigenschaften von Generizität

- generische Parameter für Klasse oder Methoden (in Java weil es keine anderen gibt: Typparameter)
- statischer Mechanismus, Compiler macht aber keine Kopien für alle Typen
- Kaum Einschränkungen, keine kovarianten Probleme
- vereinfacht Wartung und Lesbarkeit
- vernachlässigbare Auswirkungen auf die Laufzeit
- Fehlermeldungen unverständlich

Eigenschaften von Untertypbeziehungen

- Ersetzbarkeit - geeignet für unvorhergesehene Wiederverwendung
- Es gibt kovariante Probleme -> keine binären Methoden ausdrückbar ohne dynamischen Typumwandlungen
- Dynamisches Binden

Sie ergänzen sich gegenseitig:

Mit Generizität, ohne Subtyping wären *heterogene Listen* nicht möglich.
Mit Subtyping, ohne Generizität wären *homogene Listen* nicht möglich in denen statisch zu compile-Zeit Fehler erkannt werden.

Anwendung

Bisher bekannt aus `Collection<A>` oder `List<A>`, wobei A eine Klasse sein muss.
Das hier beschriebene Modell heißt *F-gebundene Generizität*.

Richtiger Einsatz:

- gleich strukturierte Klassen und Funktionen (vor allem Containerklassen, Libraries)
- Fälle in denen Änderungen der Parametertypen absehbar

Falscher Einsatz:

- als Ersatz für Untertypbeziehungen (nicht möglich)
- wenn keine Notwendigkeit

Statische Typsicherheit

Das besondere an Generizität ist die statische Typsicherheit – Fehlermeldung zu Übersetzungszeit wenn die Typparameter nicht kompatibel sind: zB man versucht eine String-Liste mit Integern zu füllen.
Deshalb vor allem bei Containern verwenden (Liste, Stack, Hashtabelle, Menge usw.).

Autoboxing

Statt elementare Typen: Klassen wie Integer, Boolean, Character.
Dabei unterstützt Java *Autoboxing* und *Autounboxing*. Dabei werden primitive Datentypen automatisch in abstrakte umgewandelt und umgekehrt.

zB: `new Integer(0)` äquivalent zu `0`

Gebundene und ungebundene Typparameter

Durch **Schranken** (Angabe von Untertypen oder Obertypen) einen Typparameter binden:

Gebundener Typparameter

```
public class Scene <T extends Scalable & Projectable>
```

Man darf beliebig viele definieren und sie mit „&“ trennen.
Weiters ist es egal ob T Scalable implementiert oder extended.

Ungebundener Typparameter

```
public class Scene <T>
```

Ist äquivalent zu: `public class Scene <T extends Object>`

Man kann Typparameter auch **rekursiv** einsetzen:

```
public static <A extends Comparable<A>> A max (Collection<A> xs) {…}
```

Untertypbeziehungen von generischen Klassen

Der Compiler vergleicht generische Klassen durch ihre Typparameter.
X<B> ist kein Untertyp von X<A> nur weil B ein Untertyp von A ist -> **Keine impliziten Untertypen**

[B → A] ⇏ [X<B> → X<A>]

Aber

[A → X<A>]

Arrays andererseits erlauben implizite Untertypbeziehungen (Obwohl diese Annahme falsch ist)

[B → A] ⇒ [B[] → A[]]
[String → Object] ⇒ [String[] → Object[]]

Dadurch zeigen sich Typfehler erst zur Laufzeit:

```
class Loophole {
        public static String loophole(Integer y) {
                String[] xs = new String[10];
                Object[] ys = xs;       // no compile-time error
                ys[0] = y;              // throws ArrayStoreException at Run-time
                return xs[0];
        }
}

class NoLoophole {
        public static String loophole(Integer y) {
                List<String> xs = new List<String>();
                List<Object> ys = xs;   // compile-time error
                ys.add(y);
                return xs.iterator().next();
        }
}
```

Wildcards

Wenn implizite Untertypen nicht angenommen werden entsteht ein Problem:

```
void drawAll(List<Polygon> p) { // draws all polygons in list p }
```

Lösung: Man kann hier zB nicht alle Untertypen von Polygon als Liste lesen -> eigentlich sollten nur Schreibzugriffe ein Problem sein.

```
void drawAll(List<? extends Polygon> p) { ... }
```

Wildcard steht für einen beliebigen Typ, der ein Untertyp von Polygon ist. Jetzt auch aufrufe vom Typ List<Triangle> und List<Square> möglich.
Der Compiler erlaubt nur Stellen, für deren Typen in Untertypbeziehungen Kovarianz gefordert ist; das sind Lesezugriffe.

Für Parameter, deren Inhalte in einer Methode nur geschrieben und nicht gelesen werden sollten:

```
void addSquares (List<? extends Square> from, List<? super Square> to ) { //add squares from 'from' to 'to' }
```

In `to` wird nur geschrieben und von `from` nur gelesen.
Als Argument für `to` können wir daher `List<Square>`, aber auch `List<Polygon>` und `List<Object>` angeben.

Der Compiler erlaubt die Verwendung von `to` nur an Stellen, für deren Typen in Untertypbeziehungen Kontravarianz gefordert ist; das sind Schreibzugriffe.

Auch wichtig: <?> entspricht <? extends Object>.

Kompliziertere Anwendung von Wildcards

```
public class MaxList<A extends Comparable<? super A>> extends List<A> {
        // Find the Maximum of a comparable List<A>
}
```

Eine Schranke <A extends Comparable<A>>scheint auf den ersten Blick klarer auszudrücken, dass auf den Listenelementen compareTo benötigt wird, weil wir die Liste nur lesen möchten, aber durch die doppelten eckigen Klammern wird super zu extends (bzw umgekehrt).

Es ist schwierig sich als Mensch vorzustellen wie Kovarianz und Kontravarianz zusammenspielen.

Arten der Übersetzung von Generizität

Für die Übersetzung generischer Klassen und Methoden in ausführbaren Code:

- Heterogene Übersetzung — Automatisierung des Erstellens von Kopien für einzelne Typen durch Compiler
- Homogene Übersetzung — Automatisierung des Erstellens der nötigen Type-Casts durch Compiler

Raw Types / Type Erasures

sind generische Typen dessen eckige Klammern bei der Anwendung ausgelassen wurden -> Quasi wie homogen übersetzt.
Dadurch weiterhin generisch aber mit `Objects`.
Generizität wurde in Java so hinzugefügt, dass die Änderungen der ursprünglichen Bibliothek und der Sprache minimal sind.
Damit man die alten nicht-generischen Klassen der Collection Bibliothek weiterhin verwenden kann hat man Raw Types eingeführt.
-> eher vermeiden weil dadurch keine Prüfungen zur Übersetzungszeit

```
public class Box<T> {
    public void set(T t) { /* ... */ }
}
```

`Box` ist die Raw Type Form des generischen Typen `Box<T>`:

```
Box rawBox = new Box();                 // rawBox is a raw type of Box<T>

Box<String> stringBox = new Box<>();
Box rawBox = stringBox;                 // OK
Box<Integer> intBox = rawBox;           // warning: unchecked conversion
rawBox.set(8);                          // warning: unchecked invocation to set(T)
```

Homogene Übersetzung (in Java)

Übersetzt eine generische in genau eine nicht-generische JVM (JavaVirtualMachine)-Code Klasse.

1. spitze Klammern samt Inhalten weglassen
2. Typparameter durch (erste) Schranke oder Object ersetzen
3. Typumwandlungen bei Aufrufen einfügen wenn Ergebnis- oder Parametertyp Typparameter ist

Manchmal auch Casting notwendig (zB bei Rückgabetypen).
Nachteil in der nicht vom Compiler ausgeführten Form: keine Fehlermeldungen bei Compile-Zeit

```
public interface Collection<A> {
        void add(A elem);
        Iterator<A> iterator();
}
public interface Iterator<A> {
        A next();
        boolean hasNext();
}
```

```
public interface Collection {
        void add(Object elem);
        Iterator iterator();
}
public interface Iterator {
        Object next();
        boolean hasNext();
}
```

Heterogene Übersetzung

Bei der heterogenen Übersetzung wird für jede Verwendung einer generischen Klasse oder Methode mit anderen Typparametern ein eigener übersetzter Code erzeugt:

Templates in C ++ -> Quasi Copy-Paste vom Compiler für jeden Datentyp (effizient für primitive Datentypen weil Compiler optimiert)

Im Vergleich zur homogenen Übersetzung:

+ effizienter da keine Typumwandlung und Code optimierbar -> int, char, . . . direkt verwendbar
+ uneingeschränkt verwendbar
+ keine Typumwandlungen zur Laufzeit und entsprechende Überprüfungen

- oft viele Klassen und große Programme, Speicherintensiv
- Klassenvariablen kopiert (= unklare Semantik)
- keine Raw-Types verwendbar
- generischer und nichtgenerischer Code inkompatibel
- schlechtere Fehlermeldungen

Wo Java strikter ist als es sein müsste: Typsicherheit und Generizität

In anderen Sprachen außer C# und Java:

- implizite Untertypbeziehungen -> Keine Unterschiede bei Typsicherheit zwischen Arrays und generischen Collections
- Erzeugung neuer Objekte von Typparameter möglich: `new A()` -> Wichtig: Aber in keiner Sprache sind Arrays Typsicher mit A[]
- Explizite Umwandlung (Casting) möglich ohne Einschränkungen

Viele Sprachen wie Java verlangen, dass bei der Generizität eine Schranke angegeben wird, dabei würde in der Theorie auch eine „Matching-Beziehung“ auch reichen (siehe Haskell) --> Mehr Flexibilität, aber man hat entschieden das nicht einzuführen weil sie parallel zu Untertypen nur zu Verwirrung führen würde und die Qualität der Fehlermeldungen sinken würde.

Dynamische Typabfragen (getClass, instanceof)

Prozeduale und funktionale Programmiersprachen haben keine dynamischen Programminformationen (zur Laufzeit).
In OOP werden sie aber sogar benötigt!

A.getClass() liefert dynamischen Typen
A instanceof B boolsche Aussage über dynamischen Typen: Liefert true, wenn A ein Untertyp von B ist, sonst false.

Allgemein gilt:

- Dynamisches Binden ist besser als Dynamische Typabfragen
- Wenn es sich vermeiden lässt sollte man dynamische Typabfragen vermeiden da oft zur Umgehung statischer Typsicherheit eingesetzt

Beispiel für Vermeidung:

```
if (x instanceof T1)
        doSomethingOfTypeT1((T1)x);
else if (x instanceof T2)
        doSomethingOfTypeT2((T2)x);
...
else
        doSomethingOfAnyType(x);
```

Ersetzen durch:

```
x.doSomething();
```

Ausgenommen man muss vielfach dieselbe Methode in allen Untertypen implementieren was die Wartbarkeit negativ beeinflussen würde.

Sichere dynamische Typumwandlung (Casting)

Casting / dynamische Typumwandlung ändert den deklarierten Typen zur Laufzeit.

> Wenn möglich Casting vermeiden und stattdessen Generizität verwenden oder dynamisches Binden

Schwierigkeiten beim Ersatz der Typumwandlung durch dynamisches Binden:

- bei sehr allgemeinen Typen (sollte vermieden werden)
- manche Methoden privat
- sehr viele Untertypen werden benötigt

Gefahren bei Casting

- Class Cast Exceptions
- die statische Typ-Überprüfung durch den Compiler wird umgangen, Fehler werden versteckt
- schlechte Wartbarkeit beim Einsatz vieler Typumwandlungen

Up-Cast Casting Richtung Obertyp **(Sicher)**
nicht so gebräuchlich und nicht nicht fehleranfällig.

```
<A> Collection<A> up(List<A> xs) {
        return (Collection<A>)xs;
}
```

Down-Cast Casting Richtung Untertyp

a) nach dynamischer Typabfrage (Unsicher)
dynamische Typabfrage stellt sicher, dass das Objekt einen entsprechenden dynamischen Typ hat -> vermeiden

- alternativer Programmzweig (else) nötig
- fehleranfällig wenn Abfragen zu allgemein
- wartungsintensiv

```
<A> List<A> down(Collection<A> xs) {
        if (xs instanceof List<A>)
                return (List<A>)xs;
        else { ... }
}
```

b) Überprüfung wie bei Generizität, aber händisch (Sicher)
Ersetzung der Typparameter

- an Methoden nur Typen übergeben die den erwarteten Typ haben
- von einer Methode nur Ergebnis zurückbekommen, dass den erwarteten Typ hat
- alle Typparameter müssen gleichförmig ersetzt werden: Eine einheitliche Ersetzung zB.: keine heterogenen Listen
- Keine Impliziten Untertypbeziehungen

Vorsicht - Manchmal unintuitiv weil wir erdachte Inhalte dazudenken müssen:
[List<String> -/-> List<Integer>] ⇒ [List -/-> List]

```
List<String> bad(Object o) {
        if (o instanceof List<String>)   // error
                return (List<String>)o;  // error
        else { ... }
}
```

Problem: `Object` hat keine Typinformation
Konsistenz von Typparameter wird vom Compiler immer bei Generizität geprüft.

```
        List<String> instanceof o       // ok
        o instanceof List               // ok
        o instanceof List<String>       // error
        return (List<String>)o;         // error
```

Kovariante / Binäre Probleme

Ist ein Ersatzbarkeitsproblem: Eingabeparameter dürfen nur kontravariant zum Typen der Klasse sein in der sie sind (weil Schreibzugriff), sonst verletzen sie das Ersetzbarkeitsprinzip.
In der Praxis wünscht man sich manchmal kovariante Eingangsparametertypen. -> Kovariantes Problem

> Kovariante Probleme lieber durch das Refaktorisieren vermeiden

Beispiel: [Point2D <- Point3D] weil Point3D eine Variable mehr enthält als Point 2D.

```
public class Point2D {
        protected int x, y; // von außen sichtbar
        public boolean equal(Point2D p) {
                return x == p.x && y == p.y;
        }
}

public class Point3D {
        protected int x, y, z;
        public boolean equal(Point3D p) {
                return x == p.x && y == p.y && z == p.z;
        }
}
```

Problem:

1. `equal`-Methode verletzt Ersetzbarkeitsprinzip: Eingabe-Parameter ist kovariant und nicht kontravariant.
   Man soll Obertypen von Point 3D (also Point2D) eingeben können, aber das wäre nicht möglich.
2. Problem bei *konzeptueller Ersatzbarkeit*. Fällt ohne Zusicherungen nicht auf. (siehe Beispiel am Ende)

Binäre Methoden

Eine Methode wie `equal` oben, bei der ein formaler Parametertyp stets gleich der Klasse ist, in der die Methode definiert ist, heißt binäre Methode.

Die Eigenschaft binär bezieht sich darauf, dass der Name der Klasse in der Methode mindestens zweimal vorkommt – *einmal als Typ von this und mindestens einmal als Typ eines formalen Parameters*.

- werden häufig benötigt, sind über Untertypbeziehungen ohne dynamische Typabfragen und Casts aber prinzipiell nicht realisierbar
- Widersprechen dem Ersetzbarkeitsprinzip, lässt sich nicht ändern
- Problem bei Vererbung: Methoden empfangen einen Parameter der Klasse, der gleich der Klasse ist, was bei Vererbung problematisch ist.

Weiteres Beispiel für kovariante Probleme:

Wir möchten Tiger und Rind korrekt füttern. Dafür erfolgt eine dynamische Typabfrage des Futters.
friss() wird überladen mit Futter und Fleisch - wäre aber sinnvoller generell friss von Tier zu entfernen und das kovariante Problem an erster Stelle zu vermeiden.
Dadurch muss man sich entweder merken welches Futter ein Tier genau frisst oder man überlädt friss() mit Fleisch und Gras.

Dynamische Typabfrage und Überladung als Lösung

```
abstract class Futter { ... }
class Fleisch extends Futter { ... }
class Gras extends Futter { ... }

abstract class Tier {
        public abstract void friss(Futter x);
}
class Rind extends Tier {
        public void friss(Gras x) { ... }
        public void friss(Futter x) {
                if (x instanceof Gras)
                        friss((Gras)x);
                else
                        werdeKrank();
        }
}
class Tiger extends Tier {
        public void friss(Fleisch x) { ... }
        public void friss(Futter x) {
                if (x instanceof Fleisch)
                        friss((Fleisch)x);
                else
                        fletscheZaehne();
        }
}
```

Überladung (statisches Binden)

Für je zwei überladene Methoden gleicher Parameteranzahl soll es zumindest eine Parameterposition geben
- an der sich die Typen der Parameter unterscheiden
- nicht in Untertyprelation zueinander stehen
- keinen gemeinsamen Untertyp haben
- oder alle Parametertypen der einen Methode sollen Obertypen der Parametertypen der anderen Methode sein, und bei Aufruf der einen Methode soll nur auf die andere Methode verzweigt werden, falls die entsprechenden dynamischen Typen der Argumente dies erlauben.

Problem im vorherigem Beispiel wo Überladung als Lösung angewendet wird:
Bei der Überladung die Methode ausgeführt, deren formaler Parametertyp der spezifischste Obertyp des deklarierten Argumenttyps ist.
Überladung wird durch statisches Binden (deklarierten Typ) ausgelöst:

```
Rind rind = new Rind();
Futter gras = new Gras();
rind.friss(gras);               // Rind.friss(Futter x)
rind.friss((Gras)gras);         // Rind.friss(Gras x)

Tier rind = new Rind();
Futter gras = new Gras();
rind.friss(gras);               // Rind.friss(Futter x)
rind.friss((Gras)gras);         // Rind.friss(Futter x) → Tier hat nur friss(Futter x)
                                Oft nicht intuitiv für Programmierer: es wird die Methode der deklarierten Klasse bei
                                Überladung zuerst aufgerufen.
```

Überladung ist generell fehleranfällig -> vermeiden - vor allem aus einer Oberklasse.

Diese Problematik kann vermieden werden wenn
- sich die Parametertypen min. in einer Stelle unterscheiden (an einer Stelle keine Untertypbeziehungen bestehen)
- aller Parametertypen der einen Methode nur Oberklassen der anderen Methode beinhaltet

Überladen im Vergleich:

Überschreiben
- Methode in Unterklasse überschreibt Methode in Oberklasse, sie wird also verwendet, egal welcher Typ deklariert ist

Überladen
- Methode in Unterklasse koexisitiert zu Methode in Oberklasse (tritt bei Java immer auf wenn: Eingangsparameter nicht invariant, Ausgangsparameter nicht kovariant)
- Bei überladenen Methoden entscheidet stets der deklarierte Typ über das Verhalten, was unerwünschtes Verhalten hervorruft.
- Mit super. aufrufbar

Multimethoden
- Wie Überladen, nur dynamischer Typ des Objektes welche die Methode aufruft entscheidet welche Methode aufgerufen wird.

Multimethoden (dynamisches Binden) - Nicht in Java

Nicht syntaktisch von Überladung zu unterscheiden, aber die Sprache entscheidet nicht durch den deklarierten Argumenttyp sondern durch den dynamischen.

```
class Rind extends Tier {
        public void friss(Gras x) { ... }
        public void friss(Futter x) {
                werdeKrank();
        }
}
```

Damit Switch-Anweisungen, geschachtelte if-Anweisungen, dynamische Typabfragen ersetzen

Es ist sonst nicht möglich bei großem Programmen die Übersicht zu behalten.

```
public void gibAnredeAus(int art, String name) {
        switch(art) {
                case 1: System.out.print("S.g. Frau " + name);
                break;
                ...
```

Gründe:

- Sonst müssen beim Einfügen von neuen Bedingungen alle switch Anweisungen in allen Methoden des gesamten Programmes die eine Fallunterscheidung durchführen aktualisiert werden -> Bei Multimethoden braucht man sie aber nur mittels einer neuen Klasse einführen (Untertyp von Adressat) ohne Struktur von anderen Klassen zu ändern.
- Es ist Unintuitiv int oder Strings als Parameter für Fallunterscheidung zu verwenden.

Visitor Pattern - Simulation von Multimethoden (dynamischem Binden) in Java

In erster Linie muss man versuchen kovariante Probleme in Java zu vermeiden, aber wenn es nicht anders möglich ist, ist das Visitor Pattern ein guter Lösungsansatz.

Visitor Klassen (Tier) – dispatch Methoden leiten Besucher weiter → Element Methoden (Methoden in Futter)
(Könnte man auch umgekehrt machen)

```
abstract class Tier {
        public abstract void friss(Futter futter);
        ...
}
class Rind extends Tier {
        public void friss(Futter futter) { futter.vonRindGefressen(this); }      //  <- Dispatch Methoden
}
class Tiger extends Tier {
        public void friss(Futter futter) { futter.vonTigerGefressen(this); }    //  <- Dispatch Methoden
}

abstract class Futter {
        public abstract void vonRindGefressen(Rind rind);
        public abstract void vonTigerGefressen(Tiger tiger);
                hier würde Überladen mit .wirdGefressen(Rind r) und .wirdGefressen(Tiger t) auch funktionieren weil wir
                immer innerhalb der anderen Methoden mit this) aufrufen und deklarierter Typ mit dynamischem Typ
                übereinstimmt
}
class Gras extends Futter {
        public void vonRindGefressen(Rind rind) { ... }
        public void vonTigerGefressen(Tiger tiger){ tiger.fletscheZaehne(); }
}
class Fleisch extends Futter {
        public void vonRindGefressen (Rind rind){ rind.werdeKrank(); }
        public void vonTigerGefressen(Tiger tiger) { ... }
}
```

Nachteile:

- Große Anzahl der zu implementierten Methoden.
  M Tierarten, N Futterarten ⇒ M · N inhaltliche Methoden (in Futter), M Dispatcher Methoden
  Generell für n Bindungen: $N_1, N_2, \ldots, N_n$ Möglichkeiten ⇒ $N_1 \cdot N_2 \cdot \cdots N_n$ inhaltliche Methoden
  Echte Multimethoden verwenden daher Komprimierungstechniken und Vererbung.

- bei mehreren Parametern: Sprachen können selbst willkürlich auswählen welcher Parameter bevorzugt wird.

```
void frissDoppelt(Futter x, Gras y) {...}
void frissDoppelt(Gras x, Futter y) {...}
void frissDoppelt(Gras x, Gras y) {...} // notwendig!
```

Beispiel: Point2D, Point3D

1. Falsche Implementierung von equal

```
public class Point3D extends Point2D {
        private int z;
        public boolean equal(Point2D p) {
                if (p instanceof Point3D)
                        return super.equal(p) && ((Point3D)p).z == z;
                return false;
        }
}
```

Fehler:

- Typabfrage
- Point3D ist Untertyp von Point2D obwohl das gar nicht notwendig wäre (falsche Semantik)
- Und vor allem ein inhaltlicher Fehler:

  Angenommen man definiert:

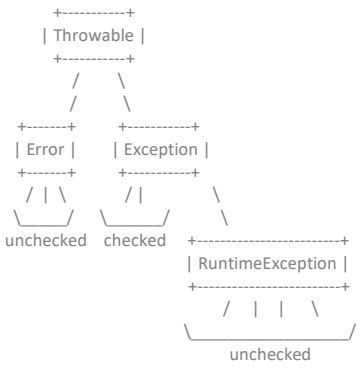
  ```
  Point2D x = …
  Point3D y = …
  ```

  Dann ergibt:

  ```
  x.equal(y) = true
  y.equal(x) = false
  ```

2. Korrekte Implementierung von equal

```
public abstract class Point {
        public final boolean equal(Point that) {
                return this.getClass()==that.getClass() && uncheckedEq(that);
        }
        public abstract boolean uncheckedEq(Point p);
}

public class Point2D extends Point {
        private int x, y;
        public boolean uncheckedEq(Point p) {
                return x==((Point2D)p).x && y==((Point2D)p).y;
        }
}

public class Point3D extends Point {
        private int x, y, z;
        public boolean uncheckedEq(Point p) {
                Point3D that = (Point3D)p;
                return x==that.x && y==that.y && z==that.z;
        }
}
```



4. Kapitel

Ausnahmebehandlung

Ausnahmen sind Objekte die das Throwable-Interface implementieren.

Checked (at compile time) / überprüfte Ausnahmen → checked Exceptions
Unchecked (at compile time) / unüberprüfte Ausnahmen → Error, Runtime Exceptions

Error (Unchecked)
Unterklassen meist für vordefinierte, schwere Fehler des Java-Laufzeitsystems.
Jeder Versuch Programm fortzusetzen führt zum selben Error.
Praktisch unmöglich abzufangen – Bei Auftritt wird Programm beendet.

Beispiele: OutOfMemoryError
StackOverflowError

Exception
Oft sinnvoll Objekte von Exception, abzufangen und den Programmablauf an anderen Stelle fortzusetzen.

a) **Checked**
vorhersehbare Ausnahmefehler, selbstdefiniert

b) **RuntimeException (Unchecked)**
unvorhersehbare Ausnahmefehler, vordefiniert

Beispiele: IndexOutOfBoundsException (Arrayzugriff außerhalb des Indexbereichs)
NullPointerException (Senden einer Nachricht an null)
ClassCastException (Typumwandlung, wobei der dynamische Typ nicht dem gewünschten Typ entspricht)

Vom Java-Laufzeitsystem werden nur Objekte der vordefinierten Unterklassen von Error und RuntimeException automatisch bei Auftritt als Ausnahmen ausgelöst. Programme können aber explizit auch andere Ausnahmen mit dem throw Befehl werfen.

Try-Catch-Blöcke
definieren Punkt ab dem ein Programm nach Exception fortgesetzt wird.

```
try { ... }
catch(Exception e) {}
finally { ... }
```

Nach try-Block auf jedem Fall auch der finally-Block ausgeführt, unabhängig davon, ob Ausnahmen beim „try-en“ aufgetreten sind:
- Ausnahme tritt im try-Block oder catch-Klausel auf: finally-Block vor Weitergabe der Ausnahme ausgeführt.
- Ausnahme tritt im finally-Block auf: finally-Block nicht weiter ausgeführt und die Ausnahme weitergegeben.

Ersetzbarkeit / Untertypenbeziehungen und Ausnahmebehandlung
1. Methode in der Unterklasse darf nur weniger Exceptions werfen als in der Oberklasse (und nur die aus der Oberklassenmethode)
2. Methoden in der Unterklasse sollen nur dann Exceptions werfen wenn Aufrufe der Oberklasse dies auch unter vergleichbaren Bedingungen erwarten (abhängig von Zusicherungen):

```
class A {
void foo() throws SelfMadeException, SyntaxError { ... }
}
class B extends A {
void foo() throws SelfMadeException { ... }
}
```

Beispiel: Nachbedingungen beeinflussen die Exceptions
(Reminder - Nachbedingungen dürfen im Untertypen nur stärker werden, nicht schwächer)

Oberklasse Wenn Eingabewert < 5 dann wirf eine Exception
Unterklasse Ausschließlich in diesem Fall / unter dieser Bedingung darf eine Exception geworfen werden (nur < 5)
Wenn der Server sich dazu verantwortet eine Exception zu werfen wenn Client einen Wert <5 übergibt, weil das die Vorbedingung ist, dann ist das eine zusätzliche Verantwortung für den Server.

Best Practice

Guter Stil: Sinnvoller Einsatz von Ausnahmen

> Ausnahmen sparsam und wirklich nur in Ausnahmesituationen einzusetzen:
Bevorzugter Weise noch in derselben Klasse gefangen werden, wo sie auch geworfen wurden.

> Typabfragen (auch bei Exceptions – durch vielen catch Blöcken bei try-catch-finally Block):
vermeiden für bessere Wartbarkeit.

> Unvorhergesehene Programmabbrüche:
Wenn Ausnahme nicht abgefangen wird, Programm abbrechen und Stack-Trace in der Konsole zeigen mit ausführlichem Fehlerhinweis

> Kontrolliertes Wiederaufsetzen:
im praktischen Einsatz soll das Programm auch dann noch funktionieren, wenn ein Fehler aufgetreten ist, zb: „Program crashed, …"

> Ausstieg aus Sprachkonstrukte:
allgemein vorzeitiges Abbrechen der Ausführung von Blöcken, Kontrollstrukturen, Methoden, etc. nur in außergewöhnlichen Situationen

> Rückgabe alternativer Ergebniswerte:
Es ist möglich, dass die Methode an den Aufrufer in Ausnahmesituationen Objekte zurückgibt, die nicht den deklarierten Ergebnistyp der Methode haben

Schlechter Stil: Falscher Einsatz von Ausnahmen

- Vorzeitiger Ausstieg aus Sprachkonstrukten: zu früh aus der Hierarchie im Stack heraussteigen -> es soll nachvollziehbar sein wo man aus dem Programm ausgestiegen ist.

- Rückgabe alternativer Werte statt Ausnahme zu werfen: Returnen von anderen Ergebniswerten wegen schlechter Programmstruktur
-> Beispielsweise „Ein Fehler ist aufgetreten" bei Exception in einer Methode die Strings returned.

- Kontrollierte / Überprüfte / Checked Exceptions eher gar nicht verwenden: sie sind fehleranfällig weil sie meist mit schlechtem Stil eingesetzt werden -> Java ist die einzige Sprache die nicht von überprüften exceptions zu applicated exceptions umgestiegen ist.

- Wenn man von außen eine Methode ansprechen will die eine throws Klausel beinhaltet wird der Umgang damit sehr schwer!

Beispiele für schlechten Stil

Kann zwar effizienter sein, aber es ist schlechter Stil und spart minimal Zeit weil wir sehr gute compiler haben - abgesehen davon können auch in getNext() Fehler sein die wir hier fälschlicherweise abfangen:

```
// Ohne Ausnahmebehandlung: pro Iteration zwei Vergleiche mit null
while (x != null)
            x = x.getNext();

// Mit Ausnahmebehandlung: pro Iteration nur ein Vergleich mit null
try { while (true)
            x = x.getNext();
} catch (NullPointerException e) {}
```

Rechts besser als links, aber schlechte Wartung weil wir immer neue Verzweigungen hinzufügen müssen wenn wir eine neue Exception einbauen wollen. Generell sind dynamische Typabfragen bei Exceptions zu vermeiden:

```
// trickreiche Verwendung von Ausnahmen:
                                        try { throw x }
if (x instanceof T1) {...}              catch (T1 x) {...}
else if (x instanceof T2 {...}          catch (T2 x) {...}
...                                     ...
else if (x instanceof Tn {...}          catch (Tn x) {...}
else {...}                              catch (Exception x) {...}
```

Rückgabe Alternativer Werte statt Exception vermeiden:

Ohne Ausnahmebehandlung:

```
public static String addA (String x, String y) {
        if (onlyDigits(x) && onlyDigits(y)) { ... }
        else return "Error Occurred";
}
```

Mit Ausnahmebehandlung:

```
public static String addB (String x, String y) throws
NoNumberString {
        if (onlyDigits(x) && onlyDigits(y)) { ... }
        else throw new NoNumberString();
}
```

Nebenläufige Programmierung (Multithreading)

Sequentielle Programmierung → Nebenläufige Programmierung

Früher nur ein CPU-Kern / Processor Cores und jetzt mehrere die echt parallel arbeiten können. Prozesse werden in Java einem oder mehreren Threads (in der Java Virtual Machine) zugeteilt. Wenn man mehrere Cores hat dann wird echt parallelisiert, ansonsten wird nur die Illusion der Parallelisierung bereitgestellt.

Threads

Ein Thread ist ein (von anderen Threads) unabhängiger Kontrollfluss.

Gefahren

- Mit Multithreading andere Programmausführungs / Exekutions-Reihenfolge → nur sequentially consistent innerhalb eines einzelnen Threads → andere arbeiten parallel und warten nicht aufeinander, können einander unterbrechen: Race Conditions
- Gleichzeitige und überlappte Zugriffe auf Variablen → Fehler wenn inkonsistente Zustände existieren (häufig!)
- Der Programmierer muss sich um die Synchronisation kümmern.
- Synchronisation ist die häufigste Fehlerquelle.
- Testen ist sehr schwierig.

Beispiel:

Nicht atomare Befehle innerhalb eines Threads in einer nebenläufigen Umgebung.
Wenn nicht Befehle synchronisiert werden können Fehler entstehen!
Hier: i und j werden nach wenigen Tausend Aufrufen von schnipp() nicht mehr gleich sein.

```
public class Zaehler {
    private int i = 0, j = 0;
    public void schnipp() { i++; j++; }
}
```

Synchronized

Synchronized bedeutet dass wir am Objekt nicht gleichzeitig mehrere Dinge ausführen können:
Soll gleichzeitige und überlappte Zugriffe vermeiden *„Mutual-Exclusion“*:

Nur Synchronized Blöcke / Methoden dürfen gleichzeitig miteinander interagieren → sollten nur ganz kurz laufen.
Feinst-mögliche Granularität (Körnung) wählen, da Synchronisierung teuer ist – andere Transaktionen müssen auf die aktuelle warten.

In einem synchronized Block: LOCK wird auf das Argument von synchronized gesetzt: also hier „this“ (bei synchronized Methoden passiert das automatisch).

```
public class Zaehler {
    private int i = 0, j = 0;
    public synchronized void schnipp() { i++; j++; }
}
```

Hier: i und j sind nur unterschiedlich wenn man es zwischen den beiden synchronized Blöcken abfragt oder nur kurzfristig, aber es wird kein Aufruf vergessen.

```
public class Zaehler {
    private int i = 0, j = 0;
    public void schnipp() {
            synchronized(this) { i++; }
            synchronized(this) { j++; }
    }
}
```

Rekursion

LOCK wird auf ein Objekt gesetzt. Bei rekursiven Aufrufen dürfen wir zB wieder zurück zu uns selbst weil wir im lock-objekt sind aber andere Aufrufer müssen warten. Wenn LOCK auf einem Thread gesetzt wurde, dann müssen wir warten bis dieser fertig ist.

Beispiel: Producer-Consumer Pattern - Threads warten auf Objektzustände
wait() ist in Object vordefiniert - nur innerhalb von synchronisierten Methoden zu verwenden.
InterruptedException wenn Thread von außen an einem unerlaubten Zeitpunkt unterbrochen.

notifyAll() ist auch in `Object` vordefiniert - nur innerhalb von synchronisierten Methoden und Blöcken zu verwenden.
Jeder wartende Thread wird aufgeweckt – drucken dann möglich:

```
public class Druckertreiber {
        private boolean online = false;

        public synchronized void drucke (String s) {
           while (!online) {
                try { wait(); }
                catch(InterruptedException ex){ return; }
           }
           ... // schicke s zum Drucker
        }

        public synchronized void onOff() {
           online = !online;
           if (online)
                notifyAll();
        }

}
```

Thread must implement Runnable! - run ist eine Endlosschleife

```
public class Produzent implements Runnable { //interface hat nur run()
           private Druckertreiber t;
           public Produzent(Druckertreiber t) { this.t = t; } //Konstruktor

           @Overide
           public void run() {
                String s = ....
                While(true) {
                        ... // produziere neuen Wert in s
                        t.drucke(s); // schicke s an Druckertreiber
                }
           }
}

Public static void main(String args[]){
           Druckertreiber t = new Druckertreiber(...);
           for (int i = 0; i < 10; i++) {
                Produzent p = new Produzent(t);
                new Thread(p).start(); //eines der vielen Methoden für Threads
           }
}
```

Nebenläufigkeit in der Praxis: Java Bibliothek

Man benutzt eher vorgefertigten Code aus der Bibliothek der optimiert ist, anstatt selbst Nebenläufigkeit zu benutzen.
Die wichtigsten vorgefertigten Lösungen (welche heute übliche Hardwareunterstützung für Parallelität nutzbar machen) in den Java-Paketen `java.util.concurrent` sowie `java.util.concurrent.atomic` und `java.util.concurrent.locks`.

Future

Aus der funktionalen Programmierung: Variable, in der das Ergebnis einer Berechnung gespeichert wird.
Gleichzeitig Berechnungen im Hinter- und Vordergrund: wenn man Variable lesen will bevor Berechnung fertig ist wird der lesende Thread so lange blockiert bis das Ergebnis da ist.
Das geht nur wenn die Hintergrund-Berechnung unabhängig von allen im Vordergrund ist. -> Unabhängigkeit der *Tasks* voneinander.
In Java gibt es dafür die Klasse `FutureTask` und das Interface `Future` im Paket `java.util.concurrent`.

Executor

Manchmal wenn die Aufgabe ganz klein ist, ist es nicht sinnvoll dafür einen eigenen Thread zu öffnen, deshalb wird die Aufgabe in den bestehenden Threads hineingefügt. In solchen Fällen kann ein `Executor` (Interface aus `java.util.concurrent`) sinnvoll sein.

Stream

Seit Java 8, zusammen mit Lambda-Ausdrücken funktionaler Programmierstil. Quasi ein interner Iterator (vergleichbar mit Foldr/Foldl aus Haskell), über den gegebene Funktionen auf alle Elemente einer Datenstruktur angewandt werden.

Stream: Sequentielle Ausführung

Wir erzeugen einen `num.stream()` und parsen alle Nummern zu Integern. Danach wendet `.reduce` den Lambda-Ausdruck `(i, j) -> i + j` auf jede Zahl im Stream und die bisher gebildeten Summen (beginnend mit 0) an und gibt am Ende die Summe aller Zahlen zurück.

Effizente für große Datenmengen -> Wenn die einzelnen Operationen keinen so großen Aufwand haben kann der Verwaltungsaufwand aber übersteigen.

```
HashSet<String> nums = ...; // "1", "2", ...
Int sum = nums.stream()
     .mapToInt(Integer::parseInt)
     .reduce(0, (i, j) -> i + j);
```

Stream: Parallele Ausführung

Zur genaueren Steuerung der Parallelausführung: `Spliterator`
Streams und Funktionaler Stil -> Weniger Abhängigkeiten / Mehr Datenunabhängigkeit -> Erhöhter Parallelisierungsgrad

```
int sum = nums.parallelStream()
     .mapToInt(Integer::parseInt)
     .reduce(0, (i, j) -> i + j);
```

Datenstrukturen

Generell sollte man in einer Nebenläufigen Umgebung nicht mehr die alten Datenstrukturen aus der Java-Bibliothek aus java.util verwenden.
Klassen im Java-Paket `java.util.concurrent` synchronisierte Varianten üblicher Datenstrukturen:

- ConcurrentHashMap (effizient für mehrere Threads) - stark an die ältere Version von Hashmap angelehnt (Hashtable) die anders ist

Für Zugriff ausschließlich in einem Thread kann man die gewöhnliche Hashmap synchronisieren und sie wäre effizienter.

- Collections.synchronizedMap(new HashMap(...))

Vorgehensweise bei nebenläufiger Programmierung

Zerlegung einer großen Aufgabe in möglichst voneinander unabhängige nebenläufig ausführbare Teilaufgaben.
Möglichst darauf achten dass sie nicht auf gemeinsame Daten zugreifen (schwierig). Sonst dafür sorgen, dass es möglichst wenige gleichzeitige (vor allem Schreib-)Zugriffe auf gemeinsame Daten. Dafür kann man die Datenstrukturen der Bibliothek benutzen.

ZIEL: hoher Parallelisierungsgrad -> Daten sollten möglichst unabhängig voneinander arbeiten in der gesamten Programmstruktur.
Es zahlt sich aus, in einigen Bereichen auf Nebenläufigkeit zu verzichten.

> Bei völliger Unabhängigkeit der Teilaufgaben voneinander:
> parallele Streams, aber auch Executor sinnvoll.

> Bei Abhängigkeit der Teilaufgaben voneinander:
> Phaser einsetzbar um Teilaufgaben in mehreren Phasen auszuführen.
> Sobald man aber verschiedene Formen der Synchronisation (andere Speichermodelle) braucht, muss man sich selbst um Reihenfolge kümmern und kann sich nicht mehr auf Executor verlassen.
> Es ist sehr schwer die Ausführungsreihenfolge zu kontrollieren aber zur Sicherheit kann man immer noch jederzeit alles einfach sequentiell ausführen.

Gefahren beim Einsatz von Synchronisation

Irreführende Namen: Vorgefertigte Datenstrukturen aus Java Bibliothek

Man hat versucht in Java über die Namensgebung Struktur in die Thread-Sicherheit von Klassen zu bringen, aber es ist nicht gelungen:
Beispielsweise sind Vector und Hashtable sicher, die ähnlichen Klassen LinkedList und HashMap aber nicht.

Entwurfsmuster ohne Bedacht einsetzen

Orientierung an Entwurfsmuster lösen Fehler nicht: Man muss Synchronisation wirklich nur so einfach halten wie es nur geht und gut testen.

Deadlocks

Unendliche Verzögerung durch zyklische Abhängigkeiten (die zum warten zwingen) zwischen zwei oder mehr Threads.

- Programmteil wartet darauf dass irgendwo eine Ressource frei wird - unendliches Warten.

Auftritt eines Deadlocks / Blockieren kann man feststellen.
Es gibt Beweis-Werkzeuge für Java die zeigen können, dass keine Deadlocks auftreten.

Wenn man Reihenfolge stark beeinflusst um Deadlocks und zu vermeiden, kann man auch die Nebenläufigkeit zu sehr einschränken
-> zu viel synchronisieren – dadurch Vorteile der Parallelisierung verlieren.
Deshalb riskiert man oft einfach Deadlocks.

Liveness Properties: Livelocks & Starvation

- Starvation: Jemand braucht Ressource so stark dass andere Verhungern -> sehr schwierig in den Griff zu bekommen
- Livelock: Ähnlicher Effekt wie Deadlock, aber dadurch das sich Flags ununterbrochen ändern und kein Fortschritt erreicht wird - ähnlich wie in einer Dauerschleife

Beide unterschiedliche Ursachen und Auswirkungen -> keine klaren formalen Definitionen.
Keine formalen Beweise - nur Testen möglich

Monitor-Konzept: Altes von Java unterstütztes Basiskonzept für Multithreading heute, nur leicht verändert um es an Java anzupassen. OOP Techniken werden kaum unterstützt:

- Synchronisation nicht in Objektschnittstellen
- Synchronisation nicht in Untertypbeziehungen

Daher ist es auch heute noch schwierig, gute nebenläufige objektorientierte Programme zu schreiben.

Untertypbeziehungen mit Synchronisation

Sehr schwer Zusicherungen zu schreiben:

- Client-kontrollierte History-Constraints berücksichtigen - Synchronisation bewirkt Einschränkungen auf Reihenfolge, in der Methoden abgearbeitet werden.

Untertypen müssen Nachrichten zumindest in allen Reihenfolgen verarbeiten können müssen, in denen Objekte von Obertypen sie verarbeiten können, oder noch mehr Reihenfolgen. -> Schwächerer oder gleicher Synchronisationsgrad im Untertypen.

- wait, notify und notifyAll dürfen nicht stärker synchronisiert sein als entsprechende Methoden in Obertypen.

Dort wo komplexe Synchronisation notwendig ist, sollte man deshalb auf Ersetzbarkeit verzichten.
Bei der Planung der Synchronisation immer so lokal und einfach wie möglich.

Vererbungsbeziehung mit Synchronisation

Abhängigkeiten durch Synchronisation erschwert reine Vererbung von Code.

Das Problem an Nebenläufigkeit in OOP

Bei der Faktorisierung der Software nach OOP Gesichtspunkten muss man anders vorgehen als bei der Zerlegung von Aufgaben in nebenläufige Teilaufgaben. - Nur selten ist eine Zerlegung nach beiden Gesichtspunkten gut.

Zerlegung nach nebenläufigen Gesichtspunkten:

+ gleicht zu sehr niedrigem Objektzusammenhalt
- führt langfristig zu hohem Wartungsaufwand wegen nötigen Refaktorisierungen.

Datanabstraktion und Datenunabhängigkeit widersprechen einander

Datenabstraktion	Verschiedene Zugriffe durch verschiednen Sichtweisen, zb Wiederholung
Datenunabhängigkeit	Möglichst unabhängig voneinander aufbauen von Grund auf, möglichst wenige Querverbindungen

Historische Entwicklung

Man betrachtete früher Objekte als Prozesse oder Threads, die mit anderen Objekten durch Senden und Empfangen von Nachrichten Informationen austauschten: zB. *Actor-Modell*.
Auf damaliger Hardware hat war so ein Modell noch nicht effizient: vereinfacht bis zum heutigen Objektmodell ohne Nebenläufigkeit.

Später, als man doch Nebenläufigkeit benötigte, hat man die Konzepte aus der prozeduralen Programmierung übernommen. -> Probleme zeigten sich später.
Deshalb kürzlich Actor-Modell plötzlich wieder beliebt und in vielen Programmiersprachen (auch Java) angeboten.

Die meisten solchen Lösungsansätze sind noch nicht ausgereift und effizient genug für ^^Einsatz im größeren Maßstab. (Viel in Entwicklung)

Annotation in Java

Optionaler Parameter der in Java Code selbst eingefügt wird und keine externe Beschreibung wie in AspectJ.

- an (fast) allen Sprachkonzepten anheftbar
- im Code statisch gesetzt
- haben ohne explizite Überprüfungen (zu runtime oder compile time) keine Auswirkung auf Programm
- können selbst definiert werden
- können Argumente enthalten

Beispiel:

@Override

Häufig verwendet. Stattdessen wäre auch ein Modifier (wie zb public, private, …) sinnvoll gewesen -> historische Gründe.
Man kann jede Annotation als Modifier sehen, den ein (Pre-)Compiler oder das Laufzeitsystem versteht.

Man kann auch die runden Klammer bei Annotationen wenn nicht nötig weg lassen: @Override()

Der Compiler prüft, ob die Methodendefinition mit dieser Annotation versehen ist und verlangt nur in diesem Fall, dass die Methode eine andere Methode überschreibt.

Selbst erstellte Annotationen

Anwendung

```
@BugFix(who="Kaspar", date="2017-12-20", level=3,
        bug="class unnecessary and maybe harmful",
        fix="contents of class body removed")
public class Buggy { }
```

Erstellung

Annotationen sind eine Adaption der Syntax von Interfaces:

```
@Retention(RetentionPolicy.RUNTIME)     // predefined annotation in Java
@Target({ElementType.TYPE})             // predefined annotation in Java

public @interface BugFix {
        String who() default "me";      // author of bug fix
        String date();                  // when was bug fixed
        int level() default 1;          // importance level 1-5
        String bug();                   // description of bug
        String fix();                   // description of fix
}
```

Einschränkungen in @interface:

- nur Eingabeparameter-lose Methoden
- wenn Methodenname `value` ist, dann muss man nicht zusätzlich `value =` … schreiben
- Als Ergebnistyp nur:
  - alle elementaren Typen (int, double, . . .)
  - Aufzählungstypen (Enum)
  - String
  - Class
  - andere Annotationen
  - eindimensionale Arrays dieser Typen (als „Mengen“)

Vordefinierte Annotationen

Im vorherigen Beispiel : Bugfix selbst ist mit vordefinierten Annotationen versehen.

```
@Retention(RUNTIME)                     // gleich mit @Retention(value=RUNTIME)
@Target({ANNOTATION_TYPE, TYPE})        // initialisiert eindimensionales Array
@Target(ANNOTATION_TYPE)                // auch möglich

public @interface Retention {
        RetentionPolicy value();        // value ist eine RetentionPolicy
}

public enum RetentionPolicy {
        CLASS, RUNTIME, SOURCE
}

public @interface Target {
        ElementType[] value();          // value ist eine Menge an Elementtypes
}

public enum ElementType {
        ANNOTATION_TYPE, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE
}
```

@Target

legt fest wo Annotation angeheftet wird (zB an Methode oder ganzen Klasse -> siehe API von `ElementType`)

@Retention

legt fest, wie lange die gerade definierte Annotation sichtbar bleiben soll / leben soll. (-> siehe `RetentionPolicy`)

RUNTIME

Falls `@Retention(RUNTIME)` wird echtes Interface erzeugt und man nennt es dann eine Reflexion.
Die Annotation ist dann auch zur Laufzeit zugreifbar.

SOURCE

Es wird die Annotation vom Compiler genauso verworfen wie Kommentare.
Solche Annotationen sind nur für Werkzeuge, die auf dem Source-Code operieren nützlich.

CLASS

sorgt dafür, dass die Annotation in der übersetzten Klasse vorhanden bleibt, aber während der Programmausführung nicht mehr sichtbar ist. Das ist nützlich für Werkzeuge, die auf dem Byte-Code operieren.

@Documented

sorgt dafür, dass die Annotation in der generierten Dokumentation vorkommt

@Inherited

sorgt dafür, dass das annotierte Element auch in einem Untertyp als annotiert gilt.

@Deprecated

Annotation, mit der Programmelemente, die man nicht mehr verwenden sollte, gekennzeichnet werden.
Eigentlich ein Kommentar aber Annotation ermöglicht dass Compiler bei Benutzung der Programmelemente eine Warnung ausgibt.

@SuppressWarnings

Gefährliche Annotation: weist den Compiler an, alle Warnungen zu unterdrücken.

@FunctionalInterface

seit Java 8: Interfaces die genau eine abstrakte Methode enthalten.

Reflexion in Java

Technik, mit der man zur Laufzeit auf Annotationen zugreifen kann, nennt man Reflexion / Reflection / Introspektion.
Erlaubt zur Laufzeit auf Programmdetails zuzugreifen.

Wurde über `@Retention(RUNTIME)` festgelegt, dass eine Art von Annotationen auch zur Laufzeit zugreifbar ist, dann generiert der Compiler ein entsprechendes (echtes) Interface. Dieses sieht für obiges Beispiel so aus:

```
public interface BugFix extends java.lang.annotation.Annotation {
        String who();
        String date();
        int level();
        String bug();
        String fix();
}
```

Werte lassen sich zur Laufzeit aufrufen.

Ähnlich wie `getClass` die Klasse eines Objekts als ein Objekt von der Klasse Class ermittelt, enthält statische Variable `class` jeder Klasse Objekte mit entsprechenden Informationen und Details -> viele vordefinierte Methoden:

So wie man über getAnnotation und getAnnotations Informationen über Annotationen bekommt, kann man sich über getMethod und getMethods Informationen über Methoden und über getField und getFields Informationen über Objekt- und Klassenvariablen holen usw. auch für Konstruktoren, Oberklassen, das Paket und so weiter.

```
String s = "";
BugFix a = Buggy.class.getAnnotation(BugFix.class);

if (a != null) {        // null if no such Annotation
        s += a.who() + " fixed a level " + a.level() + " bug";
}

Method[] ms = Buggy.class.getMethods();
                        // verschiedene analoge Methoden auch auf Method, Field

Annotation[] as = Buggy.class.getAnnotations();
for (Annotation a : as) {
        if (a instanceof BugFix)
                String s = ((BugFix)a).who; ...
}
```

Class hat viele nützliche Methoden, genau wie die Klassen Method und Field.

Anwendung von Reflexion

Reflexion ist eine Variante der Metaprogrammierung (alte, ausgereifte Programmiertechnik)

Metaprogrammierung	gesamtes Programm zur Laufzeit sichtbar und änderbar
Reflexion	nicht änderbar

Mit diesen Techniken in speziellen Fällen sehr viel erreichbar aber auch sehr gefährlich. -> möglichst im Alltag vermeiden.

Beispiel:

```
static void execAll(String n, Object... objs) {
        for (Object o : objs) {
                try { o.getClass().getMethod(n).invoke(o); }
                catch(Exception ex) {...}
        }
}
```

Das ist gefährlich weil wir nicht wissen was die mit invoke aufgerufenen Methoden machen.
Möglicherweise ist aufgerufene Methode nicht public, verlangt andere Argumente, oder existiert gar nicht.
Gefahr: wir haben keine Verhaltensbeschreibungen der mit invoke aufgerufenen Methoden

Einsatz von Annotation und Reflexion: JavaBean-Komponenten

Werkzeug, mit dem grafische Oberflächen ganz einfach aus Komponenten (fast automatisch) aufgebaut werden.
JavaBeans-Konzept: „Properties“, deren Werte von außen zugreifbar sind.
Solche Properties führt man ein, indem man entsprechende Setter- bzw. Getter-Methoden schreibt. Dabei werden Properties mit Reflexionen erstellt. In der Java-EE (Enterprise-Edition) verwendet man EJB (EnterpriseJavaBeans) als Komponentenmodell mit vielen Möglichkeiten zur Darstellung von Geschäftslogiken, hauptsächlich in Web-Anwendungen.

AOP - Aspektorientierte Programmierung

Die AOP ist ein Programmierparadigma / Werkzeug für OOP, um die gleichen Funktionalitäten über mehrere Klassen verstreut, zentral zu formulieren (Metaprogrammierung) -> Ziel: Code Wiederverwendbarkeit

Separation-of-Concerns: Wir versuchen Software immer nach Funktionalität in einzelne Modularisierungseinheiten aufzuteilen.

Software hat grundsätzlich bestimmte Aufgaben zu erfüllen:

1. ***Core-Level-Concerns – Kernfunktionalitäten***: Anforderungen an die Software, die man gut in einzelnen Funktionen kapseln kann.
   zB: Berechnung eines Wertes. Für diese Art der Faktorisierung eignet sich OOP gut mit Kapselung in Klassen.

2. ***System-Level-Concerns*** (betreffen das gesamte System) oder technische Randbedingungen: Anforderungen können nicht einfach gekapselt werden, da sie an vielen Stellen implementiert werden müssen.
   zB: *Logging*, die Protokollierung des Programmablaufs in Logdateien mit einem Logger,
   Transaktionierung von Zugriffen auf eine Ressource wie zB eine Datenbank.

Das Problem der miteinander verwobenen Anforderungen wird auch als Cross-Cutting Concerns bezeichnet: sie „schneiden“ quer durch alle logischen Schichten des Systems.

> **Cross-Cutting Concerns – Querschnittfunktionalitäten :** meist nichtfunktionale Anforderungen an Software wie Sicherheitsaspekte, die bei konventioneller Programmierung quer verstreut über den gesamten Code implementiert werden:
  zB: immer wiederkehrende Prüfungen der Form „darf dieser Code gerade ausgeführt werden?“,
  Bestimmte Aktionen im Log-Datei protokollieren, ohne den Source-Code des Programms dafür ändern zu müssen,
  das erleichtert zB debugging

Aspekte

Die AOP kapselt Verhalten, das mehrere Klassen betrifft, in Aspekten. Diese beschreiben:

- eine Funktionalität
- alle Stellen im Programm, an denen diese Funktionalität angewendet werden soll

AspectJ

Erweitert Java mit AOP - (Für fast alle anderen Programmiersprachen auch z.B. AspectL - Lisp und LOOM.NET - C#).
Open-Source, von Eclipse entwickelt, Teil der Eclipse IDE. (Programmierframework Spring gerne in Kombination verwendet)

Meta-Programmierung, nur mit großen Einschränkungen.
Es werden alle Stellen im Code mit einer Aspect Datei angesprochen zu denen Code-Snippets hinzugefügt werden sollen.

join point Ausführungspunkt in einem Programm
z.B. der Aufruf einer Methode oder der Zugriff auf ein Objekt

pointcut Wählt joint point und sammelt kontextabhängige Information dazu
z.B. die Argumente eines Methodenaufrufs oder das Zielobjekt

advice Programmcode der vor (before()), um (around()) oder nach (after()) dem Join-Point ausgeführt wird

aspect Ein Aspekt ist wie eine Klasse das zentrale Element in AspectJ
Enthält alle Deklarationen, Methoden, Pointcuts und Advices

```
01                              public class Test{
02 Methodenausf.                        public static void main(String[]a) {
03 Konstruktoraufruf                    Point pt1 = new Point(0,0);
04 Methodenaufruf                               pt1.incrXY(3,6);
05                                      }
06                              }
07
08 Klasseninit                  public class Point {
09 Objektinit                           private int x;
10                                      private int y;
11                                      public Point(int x, int y) {
12 Feldzugr.(write)                             this.x = x;
13                                              this.y = y;
14                                      }
15                                      public void incrXY(int dx, int dy){
16 Feldzugr.(read)                              x = this.x + dx;
17                                              y =+ dy;
18                                      }
19                              }
```

Zeile 02 ist ein möglicher Join-Point die Ausführung des Beginns der Methode main
Zeile 03 der Aufruf des Konstruktors Point
Zeile 12 der schreibende Zugriff auf des Feld this.x

Anonyme Pointcuts

Ein Pointcut definiert Join-Points. Es können anonyme Pointcuts definiert werden (selten – meistens benannte Join-Points verwendet).
Signatur einfach nur die syntaktische Darstellung eines Join-Points im Java-Code:

```
[Sichtbarkeit] pointcut Name ([Argumente]) : Pointcuttyp(Signatur)
```

Joint-Point-Punkte im Programm

- Konstruktor (Ausführung, Aufruf)
- Feldzugriff (schreibend/lesend)
- Initialisierung einer Klasse Objektinitialisierung (Konstruktor, (pre-)initialization) Exceptions(handler)
- …

5. Kapitel

Objekterzeugung
- Factory Method / Virtual Constructor
- Prototype
- Singleton

Programmstruktur
- Decorator / Wrapper
- Proxy / Surrogate

Verhalten
- Visitor Pattern
- Iterator / Cursor
- Template Method (Hook)

Software Entwurfsmuster / Design Patterns

Nicht zu oft verwenden sonst keine Übersicht, erschwerte Wartung.
Viele Implementierungsmöglichkeiten pro Entwurfsmuster zum Wählen.

1. Name
Wiederverwendung kollektiver Erfahrung - immer wieder auftauchende Problemstellungen und zugehörige Lösungen.

2. Problemstellung **(Anwendungsgebiete)**
Sehr allgemein.

3. Lösung **(Implementierung, beliebig abstrakt)**
Beliebig abstrakte Beschreibung einer *bestimmten* Lösung der Problemstellung die zu den Konsequenzen führt.

Erklärungen für Klassenstrukturen, Abhängigkeiten zwischen den Klassen, Verhalten der Methoden.
- Verschiedene beliebig genaue Implementierungsdetails
- Meistens nicht nur eine einzige empfohlene Struktur sondern mehrere, einander ähnliche

4. Konsequenzen **(Vor- und Nachteile)**
Eigenschaften der *bestimmten* Lösung - abhängig von den Implementierungsdetails.

Factory-Method / Virtual-Constructor

Problemstellung (Anwendungsgebiete)

- Objekterzeugung zur Laufzeit wenn zu erzeugender Typ davor zu Compile-Zeit nicht bekannt - weil zB dynamisch gebunden.
- Wenn Unterklassen Objekte bestimmen, die Oberklasse erzeugen soll.
- Bei kovarianten Problemen hilfreich.

Lösung (Implementierung, beliebig abstrakt)

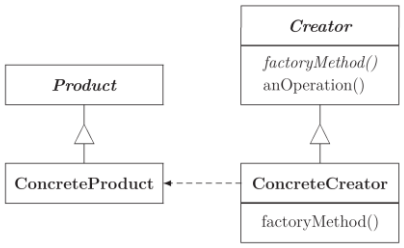
Anwendungsbeispiel mit Implementierung : System zur Verwaltung von Dokumenten

Wie in NewDocManager ist der genaue Typ des zu erzeugenden Objekts zu Compile Zeit oft nicht bekannt. Ein einfaches new reicht nicht aus.

```
public abstract class Document { ... }          //Zu erstellende Objekte
public class Text extends Document { ... }
... // classes Picture, Video, ...


public abstract class DocCreator {
        protected abstract Document create();
}
public class TextCreator extends DocCreator {   //Zuständig für Text Dokumenten-Erstellung
        protected Document create() {
                return new Text();
        }
}
... // classes PictureCreator, VideoCreator, ...


public class NewDocManager {                    //Wählt einen DocCreator
        private DocCreator c = ...;
        public void set(DocCreator c) {         //Wählt Untertyp von DocCreator mit set()
                this.c = c;
        }
        public Document newDoc() {
                return c.create();              //Ruft create() im Untertyp von DocCreator
        }
}
```

Entwurfsstruktur-Diagramm

Die „factoryMethod" (create) retourniert ein ConcreteProduct. Alle Creator müssen das tun.
Die Methode „anOperation" kann von der abstrakten Klasse selbst als factoryMethod verwendet werden oder für sonstiges.
Es wird bei der Notation angenommen, dass jede solche Vererbungsbeziehung gleichzeitig auch eine Untertypbeziehung ist.



Alternative Implementierung

Die factoryMethod() kann abstrakt sein oder eine default Implementierung haben.

Wenn die zu erzeugenden Objekte noch *Konstruktorparameter* haben:

- Argumente an factoryMethod weiterleiten
- default Argumente an zentraler Stelle ablegen
- Parameter mitzugeben, die bestimmen, welche Art von ConcreteProdukt mit vordefinierten Konstruktorparametern erzeugt werden soll.

lazy initialization: Neues Objekt wird nur einmal in einer Unterklasse mit createProduct erzeugt und gespeichert. getProduct gibt bei jedem Aufruf *dasselbe* Objekt zurück. Sobald einmal dieses Objekt erstellt wurde wird kein neues mehr erstellt.

```
public abstract class Creator {
        private Product product = null;
        protected abstract Product createProduct();

        public Product getProduct() {
                if (product == null)
                        product = createProduct();
                return product;
        }
}
```

Konsequenzen (Vor- und Nachteile)

+ Erhöhte Flexibilität, Objekterzeugung zur Laufzeit
+ *Anknüpfungspunkte (Hooks nach Template-Method)* für Unterklassen die überschrieben werden - beidseitige Abhängigkeit.
+ Bei kovarianten Problemen hilfreich: Beispielsweise erzeugt eine Methode generiereFutter in der Klasse Tier, basierend auf das aufrufende Tier das richtige Futter in Form eines Objektes.
  Klassenhierarchien Tier und Futter. Untertyp von Futter wird in Untertyp von Tier erzeugt.

- *parallele Klassenhierarchien*: die Creator-Hierarchie mit der Product-Hierarchie.
  Viele Unterklassen von „Creator“ zu erzeugen, die nur new mit einem bestimmten „ConcreteProduct“ aufrufen.
  In Java gibt es (abgesehen von aufwändiger Reflexion) keine Möglichkeit die vielen Klassen zu vermeiden.
  Es geht aber mit Templates in C++.

Prototype

Neue Objekte durch das Kopieren von bestehenden „Prototypen“ erzeugen.

Problemstellung (Anwendungsgebiete)

- Klasse des zu erzeugenden Objekts erst zur Laufzeit bekannt
- wenn Objekterzeugung teuer/aufwändig - Wegen Anzahl der Zustände, kopieren schneller als einzelnes initialisieren mit new
- Wenn zu erzeugenden Klassen sich in Zuständen sehr ähnlich sind

Lösung (Implementierung, beliebig abstrakt)

Anwendungsbeispiel mit Implementierung : Klonung von Polygonen

Ziel: Neues Polygon durch Kopieren eines bestehenden Polygons erzeugen:

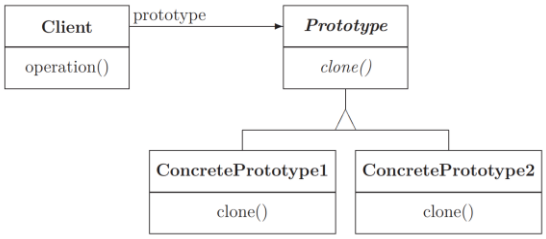
- Die Klasse muss der duplizierenden Methode nicht bekannt sein
- Kann vom kopierten Polygon verschiedenen Zustand erhalten

```
public Polygon duplicate(Polygon orig) {
        Polygon copy = orig.clone();            //clone() im Untertyp aufgerufen durch dynamisches Binden
        copy.move(X_OFFSET, Y_OFFSET);          //neuer Zustand des neuen Objekts
        return copy;
}
```

Entwurfsstruktur-Diagramm

„Client“ (Zeichenprogramm) hat Zugriff auf „Prototype“ in operation (duplicate).

Durchgezogene Linie mit Pfeil: Referenz



Alternative Implementierung

Verwaltung von Prototypen und mit denen Objekte erstellt werden schwer zur Laufzeit -> *Prototyp-Manager*: assoziative Datenstrukturen.
Setter-Methoden sinnvoll um nachträglich Dinge zu ändern.

Clone() in Java

Um die Verwendung dieses Entwurfsmusters zu fördern, haben die Entwickler von Java die Methode clone bereits in Object definiert
Muss überschrieben werden - das zu klonende Objekt muss cloneable implementieren:

flache Kopien	default Implementierung – Variablenwerte von Original und Kopie *identisch* (gleiche Referenz)
tiefe Kopien	Variablenwerte von Original und Kopie *gleich*

Probleme bei flacher Kopie:

legt nur Referenzen auf die Variablen eines Objekts an. Dadurch ist neues Objekt abhängig vom alten.

Probleme bei tiefer Kopie:

Wenn man zB das überschriebene clone() aber einfach nur rekursiv auf zyklische Strukturen anwendet, entsteht ein Stackoverflow.

Konsequenzen (Vor- und Nachteile)

+ Client muss nicht alle Klassen kennen die geklont werden
+ parallele Hierarchie aus Factory-Method vermieden
+ leicht clone() zu implementieren – mehr Prototyp Klassen zu machen
  In hochdynamischen Sprachen neues Verhalten durch *Objektkomposition*:
  Zusammensetzen neuer Objekte aus mehreren bestehenden Objekten statt durch einzelne Definition.

+ In statischen Sprachen wie C++ und Java besonders sinnvoll -> dynamische Wertzuordnungen in C++ sonst nicht möglich.
  In dynamischen Sprachen wie Smalltalk und Python wird ähnliche Funktionalität bereits direkt von der Sprache unterstützt.

Im Vergleich: Factory Method vs. Prototype

Factory Method: Delegiert Erzeugung an Unterklassen

Eher geeignet, wenn man Typ des Objekts kennt und Initialisierungs-Implementierung vor Client verstecken will.

- Klassenstruktur kann nicht zur Laufzeit geändert werden
- Parallele Hierarchie: Jede Klasse braucht eine Erstellerklasse - Anzahl der Klassen nimmt schnell zu

Prototyp: Erzeugt Kopien von deklarierten Typen

Eher geeignet in sehr dynamischen Systemen oder wenn Objekterzeugung teuer/aufwändig.

- Benötigt keine neuen Klassen, da es in jeder Klasse clone() gibt
- mühsam die neu erzeugten Typen zu verwalten
- Implementierung tiefer Kopie sehr aufwändig

Singleton

Problemstellung (Anwendungsgebiete)
Es ist besser, die Klasse selbst für die Verwaltung ihrer einzigen Objekte verantwortlich zu machen. (Aufgabe des Singleton-Patterns)

Es soll …
- genau ein Objekt einer Klasse geben
- global zugreifbar sein
- die Klasse zusätzlich durch Vererbung erweiterbar sein:
  Man benötigt häufig mehrere unterschiedliche Unterklassen von Singleton von denen aber immer nur eines instanziert werden und zurückgegeben werden darf

Lösung (Implementierung, beliebig abstrakt)
Entwurfsstruktur
Singleton Klasse mit statischer Methode instance, welche das einzige Objekt der Klasse zurückgibt.

Implementierung

```
public class Singleton {
        private static Singleton singleton;

        protected Singleton() {                 //kein Aufruf von außen
                singleton = null;
        }

        public static Singleton instance() {
                if (singleton == null)          //erzeugt nur eine Instanz von dieser Klasse
                        singleton = new Singleton();
                return singleton;               //retourniert diese immer bei jedem Aufruf
        }
}
```

Anwendungsbeispiel
Es soll in einem System nur ein Drucker-Spooler existieren (Queue und Buffering).
Angenommen es gibt mehrere Implementierungen und Unterklassen von denen aber nur eine instanziiert werden darf.
Man könnte eine globale Variable verwenden aber diese verhindern nicht, dass mehrere Objekte der Klasse erzeugt werden.

Implementierung: Mehrere Unterklassen, switch Anweisung
Implementierung mit korrektem Stil sehr schwierig:
Nur der erste Aufruf von instance erzeugt das Objekt. Nach Erzeugung des Objekts hat „kind“ keinerlei Bedeutung.

```
public class Singleton {
        private static Singleton singleton = null;

        protected Singleton() { ... }

        public static Singleton instance(int kind) {
                if (singleton == null)
                        switch (kind) {
                                case 1: singleton = new SingletonA(); break
                                case 2: singleton = new SingletonB(); break
                                default: singleton = new Singleton();
                        }
                return singleton;
        }
}

public class SingletonA extends Singleton {
        protected SingletonA() { ... }
}

public class SingletonB extends Singleton {
        protected SingletonB() { ... }
}
```

Flexiblere Variante: statt switch Block, ein Zugriff in einer Datenbank (hat Vor- und Nachteile)

Implementierung: Mehrere Unterklassen, ohne switch Anweisung
Um „kind“ zu entfernen: Klasse in der zuerst instance() aufgerufen wird entscheidet Singleton Typen endgültig (in Obertyp gespeichert).

```
public class Singleton {
        protected static Singleton singleton = null;

        protected Singleton() { ... }

        public static Singleton instance() {
                if(singleton==null)
                        singleton = new Singleton();
                return singleton;
        }
}

public class SingletonA extends Singleton {
        protected SingletonA() { ... }

        public static Singleton instance() {
                if(singleton==null)
                        singleton = new SingletonA();
                return singleton;
        }
}
```

Konsequenzen (Vor- und Nachteile)
+ Kontrollieren Zugriff auf das einzige Objekt, nur eine Instanz.
+ Man kann sich leicht um-entscheiden noch mehr Instanzen zuzulassen.
+ Flexibler als statische Methoden und globalen Variablen.
+ Unterstützen Vererbung

- Es sind einige Probleme bei der Implementation schwierig zu lösen -> heute oft von der Verwendung abgeraten.

Herausforderungen bei Implementierung
Mit mehreren Untertypen von Singletons trotzdem nur eine Instanz der Oberklasse Singleton erlaubt zu existieren.
Lässt sich meist nur mittel switch- Anweisungen usw. erreichen -> Schlechte Wartbarkeit und Flexibilität.

Jede Variante hat eigene Nachteile
- Alternativen durch switch, instance() in Untertypen – alle Unterklassen instance implementieren
- Verwendung von Datenbank – Eintrag neuer Alternativen nicht in Verantwortung von Singleton
- …

Entweder der Obertyp kennt alle Untertypen, oder die Untertypen müssen alle instance() implementieren, dass in Obertyp gespeichert wird.

Decorator / Wrapper

Proxy und Decorator können ähnlich implementiert sein, unterscheiden sich aber in Verwendung und Eigenschaften.

Problemstellung **(Anwendungsgebiete)**

- dynamisch Verantwortlichkeiten zu einzelnen Objekten (nicht der ganzen Klasse) hinzufügen oder entziehen
- große Anzahl an Unterklassen vermeiden
- für Spezialfälle wo Vererbung nicht möglich ist (beispielsweise bei final Klassen) – flexible Alternative zu reiner Vererbung
- Erfüllt Ersetzbarkeit: Wrapper ist Typ derselben Schnittstelle wie die gewrappte Klasse.

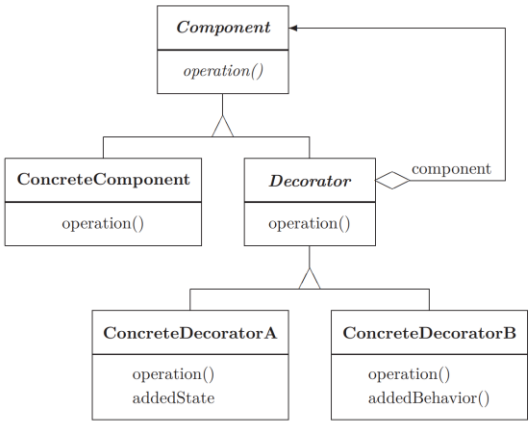
Lösung **(Implementierung, beliebig abstrakt)**

Anwendungsbeispiel mit Implementierung : Fenster mit Scrollbar

Einem Fenster am Bildschirm Bestandteile wie einen Scroll-Bar geben, anderen Fenstern aber nicht.
(Üblich, dass Scroll-Bar dynamisch während der Verwendung eines Fensters nach Bedarf dazukommt und auch wieder weggenommen wird.)

```
public interface Window {
        void show(String text);
}

public class WindowImpl implements Window {
        public void show(String text) { ... }
}

public abstract class WinDecorator implements Window {
        protected Window win;

        public void show(String text) {
                win.show(text);
        }
}

public class ScrollBar extends WinDecorator {
        public ScrollBar(Window w) { // Constructor
                win = w;
        }

        public void scroll(int lines) { ... }

        public Window noScrollBar() {
                Window w = win;
                win = null; // no longer usable
                return w;
        }
}

        Window w = new WindowImpl(); // no scroll bar in w
        ScrollBar s = new ScrollBar(w); // add window to scroll bar
        w = s; // s aware of scroll bar, w isnt
        w.show("Text"); // actually calls win.show() in scrollbar (therefore wraps win)
---->   s.scroll(3); // the added functionality
        w = s.noScrollBar(); // remove scroll bar, return w
```

Entwurfsstruktur



Pfeil mit einem Kästchen steht für Aggregation (eine Referenz als Bestandteil weil Decorator eine Component als Objektvariable besitzt).
- „Component“ (Window) definiert eine Schnittstelle für Objekte, an die Verantwortlichkeiten dynamisch hinzugefügt werden können.
- „ConcreteComponent“ (WindowImpl) ist eine konkrete Unterklasse davon.
- „Decorator“ (WinDecorator) - Jedes Objekt des Untertyps enthält eine Referenz namens „component“ (win), zu dem die Verantwortlichkeit hinzugefügt ist und wieder entfernt werden kann.

Implementierungs-Tipps
- Man kann die ConcreteDecorator-Klassen entfernen wenn es nur eine davon gibt
- Component soll so klein wie möglich gehalten werden - wirklich nur die notwendigen Operationen, keine Daten
- Bei Verwendung soll man sich nicht auf Objektidentität verlassen.

- *Verkettung von Decorators:* zB Objekt einen Rahmen hinzufügen, und diesen mit einem anderen grün färben.
  Auf diese Art und Weise z.B. umfangreiche GUIs gebildet werden.

Konsequenzen (Vor- und Nachteile)
+ mehr Flexibilität als statische Vererbung - Alles zur Laufzeit, Verantwortlichkeiten können wieder weggenommen werden
+ Eignen sich gut für GUI (graphical user interfaces)

- Nicht gut für inhaltliche Erweiterungen / für Objekte, die bereits umfangreich sind.
- Führt oft zu System, mit vielen kleinen ähnlichen Objekten – Unübersichtlich, schlecht wartbar.

Proxy / Surrogate

Problemstellung (Anwendungsgebiete)

- Platzhalter für ein anderes Objekt, Zugriffe darauf kontrollieren.
- Für Objekte, dessen Erzeugung sehr teuer sind – Kein Zugriff, Keine Objekterzeugung.
- wenn eine intelligente Referenz auf ein Objekt nötig ist

Lösung (Implementierung, beliebig abstrakt)

Allgemeine Implementierung

```
public interface Something {
        void doSomething();
}

public class ExpensiveSomething implements Something {
        public void doSomething() { ... }
}

public class VirtualSomething implements Something {
        private ExpensiveSomething real = null;

        public void doSomething() {
                if (real == null)
                        real = new ExpensiveSomething();
                real.doSomething();
        }
}
```

Alternative Implementierungen

Einige Nachrichten werden manchmal auch direkt vom Proxy behandelt.
Es kann unterschiedliche Untertypen für Proxies geben die Zugriffe auf „RealSubject" kontrollieren, aber pro Proxy nur ein Objekt.

Virtual-Proxies: erzeugen Objekte, nur bei Bedarf

Remote-Proxies: übernehmen die Kommunikation mit Objekt aus anderem Namensraum (zB Festplatten, anderen Rechnern)
Für nicht existierende Objekte könnte man zum Beispiel null verwenden und für Objekte in einer Datei den Dateinamen.

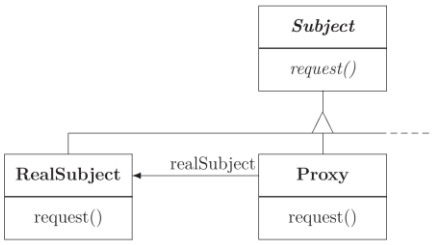
Protection-Proxies: kontrollieren Zugriffe auf Objekte. Sinnvoll wenn Objekte je nach Zugreifer oder Situation unterschiedliche Zugriffsrechte haben sollen.

Smart-References: können bei Zugriffen zusätzlich

- Referenzen auf Objekt mitzählen, damit es entfernt wird, wenn es keine mehr darauf gibt (Reference-Counting)
- Laden von persistenten Objekten in den Speicher, wenn das erste Mal darauf zugegriffen wird (fast gleich mit virtual Proxy)
- Zusichern dass kein gleichzeitiger Zugriff durch einen anderen Thread erfolgt (beispielsweise durch „Locks").

Entwurfsstruktur

Proxy implementiert auch Subject und kann als Ersatz benutzt werden, kontrolliert Zugriffe auf das eigentliche Objekt und kann für dessen Erzeugung oder Entfernung verantwortlich sein.



Unterschiede: Wrapper vs. Proxy

Ein Proxy kann die gleiche Struktur wie ein Decorator haben - dient aber einem ganz anderen Zweck:

Ein Decorator / Wrapper:

- erweitert ein Objekt um zusätzliche Verantwortlichkeiten
- Er kann einem Objekt Funktionalitäten hinzufügen, aber auch wieder entziehen.
- Decorator erzeugt Objekt nicht.

Proxy / Surrogate:

- Kontrolliert Zugriff auf das Objekt - Bindeglied zwischen dem „realen Objekt" und seinem Surrogat.
- Proxy kann Objekt erzeugen falls nötig

Verkettung von mehreren Proxies

Bei Proxies würde es zB Sinn machen, ein Virtual Proxy zu verwenden, um die Erzeugung eines Objekts zu verzögern und dieses mit einem Protection Proxy zu verknüpfen, um etwaige Zugriffsbeschränkungen zu implementieren.
Beliebige Möglichkeiten, wie z.B. das Kapseln mehrerer Funktionalitäten.

Visitor Pattern

Problemstellung (Anwendungsgebiete)

- Algorithmus der über die Klassen einer Objektstruktur verteilt arbeitet, soll zentral verwaltet werden
- Simulation von Multimethoden (dynamischem Binden) in Java

Lösung (Implementierung, beliebig abstrakt)

Implementierung

Visitor Klassen (Tier) – dispatch Methoden leiten Besucher weiter → Element Methoden (Methoden in Futter)

```
abstract class Tier {
        public abstract void friss(Futter futter);
        ...
}
class Rind extends Tier {
        public void friss(Futter futter) { futter.vonRindGefressen(this); }      //  <- Dispatch Methoden
}
class Tiger extends Tier {
        public void friss(Futter futter) { futter.vonTigerGefressen(this); }    //  <- Dispatch Methoden
}

abstract class Futter {
        public abstract void vonRindGefressen(Rind rind);
        public abstract void vonTigerGefressen(Tiger tiger);
                hier würde Überladen mit .wirdGefressen(Rind r) und .wirdGefressen(Tiger t) auch funktionieren weil wir
                immer innerhalb der anderen Methoden mit this) aufrufen und deklarierter Typ mit dynamischem Typ
                übereinstimmt
}
class Gras extends Futter {
        public void vonRindGefressen(Rind rind) { ... }
        public void vonTigerGefressen(Tiger tiger){ tiger.fletscheZaehne(); }
}
class Fleisch extends Futter {
        public void vonRindGefressen (Rind rind){ rind.werdeKrank(); }
        public void vonTigerGefressen(Tiger tiger) { ... }
}
```

Konsequenzen (Vor- und Nachteile)

+ Wichtigste Eigenschaft der betrachteten Version des Patterns: unerwünschte dynamische Typabfragen und Typumwandlungen durch dynamisches Binden zu ersetzen.
+ Verbessert dadurch Wartbarkeit und Flexibilität (Abänderbarkeit)
+ Für zweifaches dynamisches Binden und wenige Klassen ist es gut geeignet

- Für vielfaches dynamisches Binden und viele Klassen ist dieses Entwurfsmuster nicht geeignet:
  Große Anzahl der zu implementierten Methoden.
  M Tierarten, N Futterarten ⇒ $M \cdot N$ inhaltliche Methoden (in Futter), M Dispatcher Methoden
  Generell für n Bindungen: $N_1, N_2, \ldots, N_n$ Möglichkeiten ⇒ $N_1 \cdot N_2 \cdots N_n$ inhaltliche Methoden
  Echte Multimethoden verwenden daher Komprimierungstechniken und Vererbung.

- bei mehreren Parametern: Sprachen können selbst willkürlich auswählen welcher Parameter bevorzugt wird.

```
void frissDoppelt(Futter x, Gras y) {...}
void frissDoppelt(Gras x, Futter y) {...}
void frissDoppelt(Gras x, Gras y) {...} // notwendig!
```

Iterator / Cursor

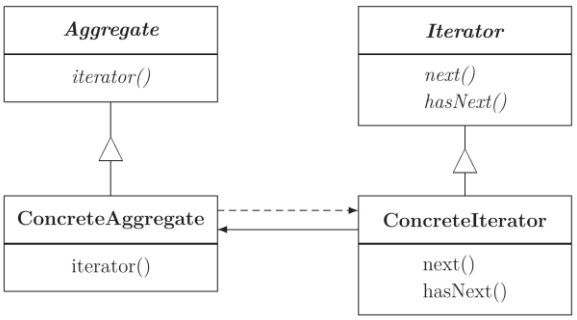
Aggregat = Sammlung von Elementen, beispielsweise eine Collection

Problemstellung (Anwendungsgebiete)

- Gute Schnittstelle zu Aggregaten: ermöglicht sequentiellen Zugriff auf die Elemente ohne innere Darstellung offenzulegen.
- Mehrere (gleichzeitige bzw. überlappende) Abarbeitungen der Elemente im selben Aggregat möglich – nur wenn Abarbeitungszustand nicht im Aggregat verwaltet wird.
- gleiche Schnittstelle für unterschiedlichen Aggregatsstrukturen / Datenstrukturen
- unterschiedliche Abarbeitungsarten von Aggregaten möglich - Es gibt zB viele Arten über einen Baum zu iteriere
- können auch andere Methoden enthalten wie remove() um Elemente zu entfernen.

Lösung (Implementierung, beliebig abstrakt)

- Ein Aufruf von iterator() erzeugt ein neues Objekt von „ConcreteIterator“ (strichlierter Pfeil).
  Um die aktuelle Position im Aggregat verwalten zu können, braucht jedes Objekt von „ConcreteIterator“ eine Referenz auf das entsprechende Objekt von „ConcreteAggregate“ (durchgezogener Pfeil)



Implementierungs-Tipps

- Es kann gefährlich sein ein Aggregat zu verändern, während es von einem Iterator durchwandert wird -> passiert leicht, dass Elemente doppelt oder gar nicht abgearbeitet werden.

  Lösung 1: Aggregat kopieren und über Kopie iterieren

  Lösung 2: ***robusten Iterator*** implementieren (schwierig). Ermöglicht das Verändern eines Aggregats, während es von einem anderen Iterator durchlaufen wird ohne es vorher zu kopieren.

- oft praktisch, Iteratoren auch auf leeren Aggregaten bereitzustellen (auch bei internen Iteratoren, damit keine Exceptions auftreten. In Java: flatMap, wie Map nur keine Sonderbehandlung für leere Streams notwendig)

Konsequenzen (Vor- und Nachteile)

Vorteile siehe Problemstellung (Anwendungsgebiete)

Ort für Algorithmus zum Durchwandern des Aggregats

a) Im Aggregat selbst:
   besser, da Implementierungsdetails des Aggregats private bleiben können und sich Algorithmen-Teile mehrfach verwenden lassen.
   -> Java: Innere Klassen, Iterator in Containerklasse integrieren -> direkte Zugriffe.
   Sie haben ermöglichen Zugriff auf private Inhalte von Klassen (mehr Implementierungsdetails).
   Dies erhöht aber die Klassenabhängigkeit zwischen Aggregat und Iterator noch weiter.

b) Im Iterator:
   Das iterieren von mehreren Iteratoren über demselben Aggregat einfacher und das Bereitstellen von mehreren Durchwanderungsalgorithmen.

Interne und externe Iteratoren

Interne Iteratoren (Streams oder *map* aus Haskell)

Man übergibt dem Iterator eine Operation (z.B. in Form einer Methode oder Funktion), die auf alle Elemente des Aggregats gemapped wird. zB Stream-Operationen oder ein Aufruf der Methode forEach in einer Map.

- Kontrolliert selbst wann nächste Iteration erfolgt bzw enthält die Schleife selbst.
- Von außen next und hasNext nicht zugänglich. **->** Einfacher weil keine Logik für für Schleifenposition
- Gut in funktionalen Sprachen umsetzbar
- Auch mit Lambda Ausdrücken und Streams in Java. -> Gute parallele Verarbeitung von großen Datenmengen.
- Sinnvoller wenn Beziehungen zwischen den Elementen (zB Position im Baum) erhalten werden soll, da sie bei externen Iteratoren verloren gehen. Das kann aber Parallelisierung erschweren.

Externe Iteratoren (bekannt aus EP2)

In objektorientierten Sprachen gut umsetzbar.

- Anwender bestimmt wann nächste Abarbeitung erfolgt.
- Einsatzmöglichkeiten flexibler weil man mit ihnen zwei Aggregate miteinander vergleichen kann.
- Schlechter für komplexe Beziehungen, schwieriger zu steuern.

Template-Method

Hook
Manche Programme bestimmen Stellen, die explizit dazu vorgesehen sind, von anderen Programmen erweitert zu werden.
Das wird durch den Hook ermöglicht: eine Funktion, die im ursprünglichen Programm leer implementiert ist, also nichts bewirkt.
Diese kann dann einfach auf eine fremd-implementierte Funktion umgelenkt werden.

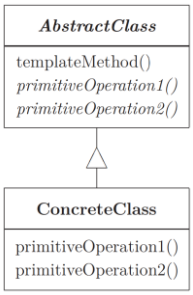
Es gibt mehrere Möglichkeiten, Hooks zu implementieren. In OOP durch Vererbung oder durch Delegation:

- Template-Method-Entwurfsmuster: (Vererbung) Hooks als leere Methoden, die in Unterklassen ausimplementiert werden können. Hier werden mehrere Methoden hintereinander als Hooks aufgerufen, um Algorithmen mit festem Ablaufrahmen spezialisieren oder erweitern zu können.

- Strategie-Entwurfsmuster: (Delegation) Hook in Form eines Objektes, dass eine bestimmte Schnittstelle implementiert an Aufrufer returned.

Problemstellung (Anwendungsgebiete)
- keine Code Duplikate: unveränderlicher Teil eines Algorithmus nur einmal implementiert
- Teilt Algorithmus zwischen Oberklasse und Unterklasse auf. Erlaubt einer Unterklasse bestimmte Schritte zu überschreiben, ohne die Struktur des Algorithmus zu ändern

Lösung (Implementierung, beliebig abstrakt)
Entwurfsstruktur-Diagramm
Template-Methoden werden auch vererbt: führen einzelne Schritte im Algorithmus aus, aber enthalten auch Teile im Algorithmus, welche die anfangs leeren Hook-Methoden / primitiveOperations aufrufen.



Es muss vor Vererbung jede Methode genau spezifiziert sein:
- template Methoden — nur in „AbstractClass“ implementiert – Überschreiben verboten
- Hooks — kann Standard-Implementierung enthalten oder abstract sein – Überschreiben optional
- abstract Methoden — „primitiveOperations“ (meistens protected) – Überschreiben verpflichtend

Standardimplementierungen in Hooks
Hooks können leer sein, Hooks können aber auch eine Standard-Implementierung besitzen, die von den konkreten Klassen genutzt werden kann, aber nicht muss -> bspw: Factory-Method Hooks.

Implementierungs-Tipps
- Ein Ziel bei der Entwicklung einer Template-Methode sollte sein, die Anzahl der primitiven Operationen möglichst klein zu halten. Je mehr Operationen überschrieben werden müssen, desto komplizierter wird die direkte Wiederverwendung von „AbstractClass“.
- Stellen die nicht überschrieben werden sollten, sollten mit dem `final` modifier gekennzeichnet sein.

Konsequenzen (Vor- und Nachteile)
+ Technik zur Wiederverwendung von Programmcode - vor allem in Klassenbibliotheken und Frameworks - gut zum Faktorisieren.
+ Umgekehrte Kontrollstruktur (*Hollywood-Prinzip*: „Don’t call us, we’ll call you“): Die Oberklasse ruft die Methoden der Unterklasse auf – nicht wie in den meisten Fällen umgekehrt. -> Beidseitige Abhängigkeiten

Anzahl der benötigten Klassen und Objekte im Vergleich zu System ohne Patterns

Factory-Method / Virtual-Constructor

Anzahl der benötigten Klassen: steigt, wegen paralleler Creator-Klassenhierarchie
Anzahl der benötigten Objekte: bleibt gleich - nur „Ort“ der Objekterzeugung verschoben

Prototype

Anzahl der benötigten Klassen: bleibt gleich aber im Vergleich zu Factory deutlich vermindert
Anzahl der benötigten Objekte: eher erhöht da Prototypes, die nicht zwingend verwendet werden, auch erstellt werden.

Decorator / Wrapper

Anzahl der benötigten Klassen:

- sinkt, da man durch das einfache Erweitern von mit Wrappern Unterklassen einsparen kann
- steigt, da man Wrapper Klassen erstellen muss für jedes Basis-Objekt

Anzahl der benötigten Objekte: steigt, da pro Basis-Objekt ein Wrapper-Objekt zum „verknüpfen“ erstellt wird
-> Verkettung mehrerer Wrapper für z.B. umfangreiche GUIs

Proxy / Surrogate

Anzahl der benötigten Klassen: steigt leicht, für jedes Objekt andere Klasse notwendig zum Erzeugen oder Verwalten des Zugriffes.
Anzahl der benötigten Objekte: steigt leicht, da für jedes zusätzliche Zugriffs-Feature weiteres Proxy-Objekt nötig (Verkettung)
-> Verkettung mehrerer Proxies für z.B. das Kapseln mehrerer Funktionalitäten