Zusammenfassung
Objektorientierte Programmiertechniken (OOP)

1. Paradigmen der Programmierung

- imperativ
    - prozedural
    - objektorientiert
- deklarativ
    - funktional
    - logikorientiert

Referentielle Transparenz: Ausdruck (z.B. Funktion) kann durch ihren (Return-)Wert ersetzt werden (z.B. in rein funktionalen Sprachen). Keine Seiteneffekte möglich!

First Class Entities
Einheiten, “um die sich alles dreht”. Diese Einheiten können wie Daten verwendet werden. Man kann sie Variablen zuweisen oder als Argumente übergeben.
-> in objektorientierten Sprachen das Objekt, in funktionalen Sprachen die Funktion

1.2. Programmorganisation

1.2.2. Modularisierung

Modularisierungseinheiten Zerlegen das Programm in einzelne Einheiten, um bessere Struktur zu ermöglichen.

Modul
Eine *Übersetzungseinheit*, also die Einheit, die ein Compiler in einem Stück bearbeitet, z.B. in Java ein Interface oder eine Klasse. Wenn einzelne Module geändert werden, muss man nur diese Module (und evtl. von Ihnen abhängige) neu übersetzen, nicht alle.

Module exposen ihre Schnittstellen nach außen (export), aber nicht interne Vorgehensweisen. Zyklische Abhängigkeiten werden in Java zB durch getrennte Übersetzung von Interfaces und Klassen verarbeitet.

Helfen auch, wenn mehrere Entwickler an einem Projekt arbeiten, um Zuständigkeiten zu definieren.

Objekt
Keine Übersetzungseinheiten, sondern werden zur Laufzeit erzeugt. Sie Kapseln Methoden und Variablen zu logischen Einheiten (Kapselung) und schützen private Inhalte vor Zugriffen von außen (Data-Hiding). Kapselung und Data-Hiding zusammen nennt man Datenabstraktion.

Klasse
Eine Klasse wird als Schablone für die Erzeugung neuer Objekte beschrieben.
In Java ist eine Klasse ein Modul, also eine Übersetzungseinheit. Statische Variablen und Methoden sowie Konstruktoren entsprechen Modulinhalten, und zyklische Abhängigkeiten zwischen Klassen sind verboten. Durch Ableitung von (abstrakten) Klassen oder Interfaces lassen sich zyklische Abhängigkeiten für nicht-statische Methoden immer auflösen.

Komponente
Aus einem Modul wird eine Komponente, wenn man zunächst offen lässt, von welchen anderen Komponenten etwas importiert wird; erst beim Zusammensetzen des Systems werden diese Komponenten bestimmt

Namensraum
Jede oben angesprochene Modularisierungseinheit bildet einen eigenen Namensraum und kann damit Namenskonflikte gut abfedern. Globale Namen (z.B. Namen von Klassen) können jedoch nach wie vor nur einmal bestehen. Deshalb kann man eigenen Namensräume (in Java packages) definieren.

1.2.2. Parametrisierung

In Modularisierungseinheiten werden Lücken gelassen, die später befüllt werden.

- **Befüllen zur Laufzeit**
  - im Konstruktor
  - Initialisierungsmethode - z.B. beim klonen, oder wenn zwei zu erzeugende Objekte voneinander abhängen
  - zentrale Ablage von Variablen/Konstanten, auf die Objekte bei Erzeugung zugreifen
- **Generizität**
  Lücken werden zumindest konzeptuell bereits zur Übersetzungszeit befüllt. Daher können alle Arten von Modularisierungseinheiten außer Objekte generisch sein. Später, aber noch vor der Programmausführung, werden die Parameter (meist Typparameter) durch das Einzufüllende ersetzt.
- **Annotationen**
  Siehe Annotationen weiter unten
- **Aspekte**
  Siehe Aspekte weiter unten

1.2.3. Ersetzbarkeit

Möglichkeit zu nachträglichen Änderungen von Modularisierungseinheiten. Eine Modularisierungseinheit A ist durch B ersetzbar, wenn wenn ein Austausch von A durch B keinerlei Änderungen an Stellen, an denen A verwendet wurde, nach sich zieht.

Schnittstellen einer Modularisierungseinheit können spezifiziert werden durch

- Signatur
- Abstraktion
- Zusicherungen

- Überprüfbare Protokolle

Abstraktionen stellen die Basis für die Schnittstellenbeschreibung dar - in jüngerer Zeit oft gepaart mit Zusicherungen. Abstraktionen werden häufig über Klassen realisiert. Vererbung und Ersetzbarkeit sind jedoch klar voneinander zu trennen.

1.3. Typisierung

Typen schränken ein, welche Werte, Ausdrücke etc. wo verwendet werden dürfen, aber Erhöhen die Planbarkeit und Lesbarkeit/Verständlichkeit des Programms und verringern Fehleranfälligkeit zur Laufzeit.

Typprüfungen
In dynamischen Sprachen werden Typen erst bei ihrer Anwendung (zur Laufzeit) geprüft. In statischen Sprachen hat bereits der Compiler viele Informationen über Typen und prüft diese zur Compile-Zeit - Typfehler zur Laufzeit werden ausgeschlossen. In Java z.B. ein Mix aus beidem, in Haskell z.B. nur statisch.
Auch bei statischen Sprachen muss nicht jeder Typ explizit hingeschrieben werden -> Typinferenz.

Statische Typisierung erhöht die Planbarkeit und erleichtert Refactoring.

Abstrakte Datentypen
Sind im wesentlichen Schnittstellen von Modularisierungseinheiten. In diesem Fall spricht man vom strukturellen Typ. Zwei strukturelle Typen können bezogen auf ihre Schnittstellen identisch sein, deshalb werden oft nominale Typen (z.B. in Java) verwendet. Hier sind zwei strukturell gleich Typen nicht wechselseitig ersetzbar, wenn sie unterschiedliche Namen haben. Programmiersprachen, die mit strukturelle Typen arbeiten verwenden Duck-Typing (z.B. TypeScript).

Untertypen
Werden durch das Ersetzbarkeitsprinzip definiert (siehe unten)

Für strukturelle Typen kann man Ersetzbarkeit einfach ableiten. Bei nominalen Typen ist dies dem Programmierer überlassen (in Java extends, implements)

Generizität
Während Subtyping den objektorientierten Sprachen vorbe- halten ist, wird Generizität in Sprachen aller Paradigmen mit statischer Typprüfung verwendet. Mehr siehe unten.

1.4. Objektorientierte Programmierung

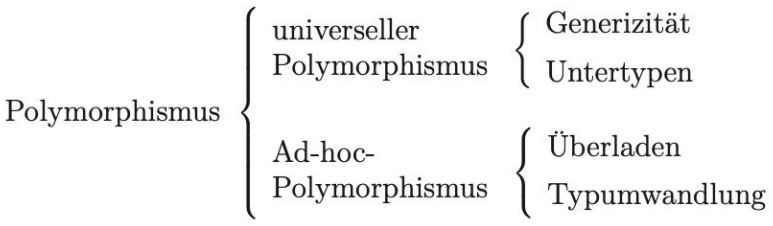
1.4.1. Basiskonzept

Ein Objekt verfügt über

- Identität (Identity) - vereinfacht: Speicheradresse)
- Zustand (State) - aktuelle Werte der Variablen im Objekt; zwei Objekte sind gleich, wenn sie den gleichen Zustand haben
- Verhalten (Behavior) - wie verhält sich das Objekt, beim Empfang einer Nachricht (Methodenaufruf)

1.4.2. Polymorphismus

Wenn eine Variable oder eine Methode gleichzeitig mehrere Typen haben kann.



Eine Variable hat gleichzeitig folgende Typen:

- Deklarierter Typ: Typ der bei der Deklaration angegeben wird
- Dynamischer Typ: der spezifischste Typ, den der in der Variablen gespeicherte Wert hat (zur Compile- oder Laufzeit)
- Statischer Typ: wird vom Compiler ermittelt und liegt zwischen Deklariertem und Dynamischem Typ

Durch dynamisches Binden wird zur Laufzeit entschieden, welche Methode eines etwaigen Untertyps ausgeführt werden soll.

Vererbung
Erlaubt Wiederverwendung von Code.

Eine Klasse kann durch Vererbung **erweitert** werden oder deren Variablen/Methoden **überschrieben** werden.

Ein Objekt der Unterklasse kann überall dort verwendet werden, wo ein Objekt der Oberklasse erwartet wird - es besteht also eine Untertypbeziehung. Java unterstützt nur Einfachvererbung (eine Klasse kann nur von einer einzigen Klasse erben).

Ein Interface kann von mehreren Interfaces erben.

1.4.3. Vorgehensweise in der Programmierung

Faktorisierung
Zusammengehörige Eigenschaften und Aspekte des Programms sollen zu Einheiten zusammengefasst werden. z.B. Mehrfachvorkommen von ähnlichen Sequenzen zu einer Methode zusammenfassen. Bei guter Faktorisierung muss dann nur eine Stelle anstatt x Stellen geändert werden bei zukünftiger Refaktorisierung.

Zusammenhalt und Kopplung

Beim Entwickeln von Software ist vorallem gute Faktorisierung ein Ziel. Diese ist leider erst eher gegen Ende der Entwicklung beurteilbar, aber es gibt ein paar Faustregeln, die einen Weg zu guter Faktorisierung weisen:

- **Verantwortlichkeiten einer Klasse**
  - Was ich weiß - Beschreibung des Zustands der Objekte
  - Was ich mache - Verhalten der Objekte
  - Wen ich kenne - sichtbare Objekte, Klassen usw.

  Das ich steht dabei jeweils für die Klasse

- **Klassen-Zusammenhalt (sollte hoch sein)**
  Der Zusammenhalt ist hoch, wenn alle Variablen und Methoden der Klasse eng zusammenarbeiten. Einer Klasse mit hohem Zusammenhalt fehlt etwas wichtiges, wenn man beliebige Variablen oder Methoden entfernt. *“Ein Programm ist dann perfekt, wenn man nichts mehr wegnehmen/refactoren kann.”* Außerdem wird der Zusammenhalt niedriger, wenn man die Klasse sinnändernd umbenennt.

- **Objekt-Kopplung (sollte gering sein)**
  ist stark, wenn
  - viele Variablen und Methoden nach außen sichtbar sind,
  - zur Laufzeit Zugriffe zwischen Objekten häufig auftreten,
  - die Anzahl der Parameter dieser Methoden groß ist.

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

Perfekte Faktorisierung von Anfang an ist nicht möglich, deshalb *ständig refactoren!*

Das A und O der Objektorientierten Programmierung ist die Datenabstraktion, Ersetzbarkeit, eindeutige Aufteilung von Verantwortlichkeiten, angemessene und starke Kapselung und das Anstreben von Wiederverwendbarkeit.
Im Zweifel jedoch am Anfang nicht zu sehr auf Wiederverwendbarkeit versteifen und später refactoren.

Für andere Programmierparadigmen (z.B. funktionale, prozedural, logikorientiert) stehen Algorithmen und nicht Datenabstraktion im Vordergrund.
Objektorientierte Programmierung eignet sich zur Entwicklung von Systemen, deren Gesamtkomplexität jene der einzelnen Algorithmen deutlich übersteigt. Sonst sind andere Paradigmen besser geeignet.

2. Untertypen und Vererbung

2.1. Das Ersetzbarkeitsprinzip

Ein Typ U ist Untertyp von Typ T, wenn ein Objekt vom Typ U überall dort verwendbar ist, wo ein Objekt vom Typ T erwartet wird.

2.1.1. Untertypen und Schnittstellen

Insbesondere benötigt bei:

- Aufrufen einer Methode mit einem Argument, dass Untertyp des Typs des entsprechenden formalen Parameters ist.
- Zuweisung einer Objekts an eine Variable, wobei der Typ des Objekts ein Untertyp des deklarierten Typs ist.

In Java: Typen sind Schnittstellen von Objekten, die in Klassen oder Interfaces spezifiziert worden sind.

Alle Untertypbeziehungen sind stets:

- reflexiv: T ist ein Untertyp von T
- transitiv: ist B ein Untertyp von A und C ein Untertyp von B, dann ist C auch ein Untertyp von A
- antisymmetrisch: ist T ein Untertyp von U und U ein Untertyp von T, dann sind U und T äquivalent

Bedingungen an Untertypen:

- Für jede **Konstante** (nur lesender Zugriff) in T gibt es eine entsprechende Konstante in U, wobei der deklarierte Typ **B** der Konstante in U ein **Untertyp** des deklarierten Typs **A** der Konstante in U sein muss (weil nur lesend).
- Für jede **Variable** (lesender und schreibender Zugriff) in T gibts es eine entsprechende Variable in U, wobei der deklarierte Typ der Variablen **äquivalent** sein muss (weil lesend und schreibend)
- Für jede **Methode** in T gibt es eine entsprechende gleichnamige Methode in U, wobei
    - der deklarierte **Ergebnistyp** der Methode in U ein **Untertyp** des deklarierten Ergebnistyps der Method in T sein muss,
    - die **Anzahl** der formalen Parameter beider Methoden **gleich** sein muss,
    - der **deklarierte Typ von jedem Parameter** der Methode in U ein **Obertyp** des entsprechenden Parameters in T sein muss.

Deklarierte Elemente in Unter- und Obertypen können variieren. Es gibt folgende Arten der Varianz:

- **Kovarianz:** Der deklarierte Typ eines Elements im Untertyp ist ein Untertyp des deklarierten Typs des entrsprechenden Elements im Obertyp (z.B. Konstanten, Methodenergebnisse, Ausgangsparameter (z.B. in Ada))
- **Kontravarianz:** Der deklarierte Typ eines Elements im Untertyp ist ein Obertyp des deklarierten Type des entsprechenden Elements im Obertyp. (z.B. Eingangsparameter)
- **Invarianz:** Der deklarierte Typ eines Elements im Untertyp ist äquivalent zum deklarierten Typ des entsprechenden Typs im Obertyp. (z.B. Variablen, Durchgangsparameter)

Java beruht nicht auf strukturellen Typen, sondern auf nominalen. Nur weil zwei Typen die gleiche Struktur haben, sind sie nicht äquivalent (in TypeScript z.B. schon -> Duck Typing).

Eine Methode wie equals, 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. Binäre Methoden werden häufig benötigt, sind über Untertypbeziehungen (ohne dynamische Typabfragen und Casts) aber prinzipiell nicht realisierbar.

Kovariante Eingangsparametertypen und binäre Methoden widersprechen dem Ersetzbarkeitsprinzip. Es ist sinnlos, in solchen Fällen Ersetzbarkeit anzustreben.

In **Java** sind **alle Typen invariant**, abgesehen von kovarianten Ergebnistypen, weil bei Zugriffen auf Variablen und Konstanten nicht dynamisch sondern statisch gebunden wird und weil Methoden überladen sein können (da überladene Methoden anhand der Typen ihrer formaler Parameter unterschieden werden, wäre es schwierig diese von Methoden mit kontravariant veränderten Typen außeinander zu halten.

2.1.2. Untertypen und Codewiederverwendung

Ein Untertyp kann mehrere Obertypen haben, die zueinander in keiner Untertypbeziehung stehen. Diese Mehrfachvererbung kann in Java nur über Interfaces realisiert werden.

Schnittstellen bzw. Typen sollen stabil bleiben. Interne Änderungen sollen sich, wenn möglich, nicht auf Clients, die Objekte dieses Typs verwenden auswirken.
Die Stabilität von Schnittstellen an der Wurzel der Typhierarchie ist wichtiger als an den Blättern. Man soll nur Untertypen von stabilen Obertypen bilden.
Typen von formalen Parametern sollten möglichst allgemein gehalten werden.
Schnittstellen werden trotz guter Planung oft erst nach einigen Refakorisierungsschritten stabil -> nicht zu früh Kopf zerbrechen.

2.1.3. Dynamisches Binden

Bei Verwendung von Untertypen kann der dynamische Typ einer Variablen oder eines Parameters ein Untertyp des deklarierten Typs sein. Meist ist zur Übersetzungszeit der dynamische Typ nicht bekannt -> dynamischer Typ kann sich vom statischen Typ unterscheiden. In Java wird unabhängig vom deklarierten Typ immer die Methode ausgeführt, die im spezifischsten dynamischen Untertyp definiert ist. Dieser Typ entspricht der Klasse des Objekts in der Variablen.

Dynamisches Binden ist switch und geschachtelten if-Anweisungen vorzuziehen.

2.2. Ersetzbarkeit und Objektverhalten

2.1.1. Client-Server-Beziehungen

- Objekt fungiert als **Client**, wenn es auf über Schnittstellen eines anderen Objekts zugfreift auf dessen Dienste zugreift.
- Objekt fungiert als Server, wenn es anderen Objekte über Schnittstellen seine Dienste zu Verfügung stellt.

Zusicherungen sind Verträge zwischen Server und Client zur Fehlervermeidung („Design by Contract“)

- **Vorbedingungen:** Beschreiben, welche Bedingungen ein Argument vor dem Methodenaufruf erfüllen muss. Server verlässt sich (blind) darauf.
    - z.B. Methodenparameter >= 0, Objektvariable des Servers == 1
    - Verantwortlich: Client

- **Nachbedingungen:** Beschreibt den Zustand, der nach Methodenaufruf gegeben sein muss (Methodenergebnis oder Änderungen beziehungsweise Eigenschaften des Objektzustandes). Lesen sich oft, wie Methodenbeschreibungen (*“addiert summe zu guthaben”*).
    - z.B. Methodenergebnis > 0 oder Objektvariable des Servers == 1
    - Verantwortlich: Server

- **Invariante:** Bestimmungen auf Objektvariablen des Objekts für die komplette Lebenszeit.
    - z.B. Objektvariable des Servers == 1
    - Verantwortlich: Vorrangig Server, aber bei public Variablen (direkte Schreibzugriffe) auch Client

- **History-Constraint (Server-kontrolliert):** Einschränkung von zeitlichen Veränderungen von Objektvariablen.
    - z.B. Objektvariable der Servers darf immer nur um 1 erhöht werden (counter)
    - Verantwortlich: Vorrangig Server, aber bei public Variablen (direkte Schreibzugriffe) auch Client (wie bei Invariante)

- **History-Constraint (Client):** Einschränkung der Reihenfolge von Methodenaufrufen.
    - z.B. zuerst initialize-Methode aufrufen, dann erst andere Methoden
    - Verantwortlich: Client

Zusicherungen werden als zum Typ (und zur Schnittstelle) zugehörig betrachtet.

Ein nominaler Typ besteht demnach aus:
- Name des Typs (Klasse, elementarer Typ oder Interface)
- Signatur (alle vom Compiler überprüfbaren Bestandteile des Vertrags zwischen Client und Server)
- zugehörige Zusicherungen (alle nicht vom Compiler überprüfbaren Bestandteile).

Faustregeln für Zusicherungen:
- ❖ Zusicherungen sollen stabil bleiben (vor allem nahe der Wurzel in der Typhierarchie). Änderung von Zusicherungen (also Kommentaren) kann sich sehr wohl auf andere Programmteile auswirken.
- ❖ Zusicherungen sollten keine unnötigen Details festlegen.
- ❖ Durch Zusicherungen soll Klassenzusammenhalt maximiert und Objektkopplung minimiert wird.
- ❖ Ziel von Zusicherungen ist Unabhängigkeit von Client und Server
- ❖ Keine “versteckten” Zusicherungen” -> Alle von geschulten Personen nicht in jedem Fall erwarteten, aber vorausgesetzten Eigenschaften sollen explizit als Zusicherungen im Programm stehen.

2.2.2. Untertypen und Verhalten

- **Vorbedingungen:** darf in Untertypen schwächer sein, aber nicht stärker (wegen dynamischer Bindung - Aufrufer der Methode, der nur T kennt, kann nur die Erfüllung der Vorbedingungen in T sicherstellen, aber nicht in U)
- **Nachbedingungen:** können in Untertypen stärker sein, aber nicht schwächer (wegen dynamischer Bindung - Aufrufer der Methode, der nur T kennt, muss sich auf Erfüllung verlassen können, auch wenn tatsächlich U verwendet wird)
- **Invarianten:** können in Untertypen stärker sein, aber nicht schwächer (wegen dynamischer Bindung - Aufrufer der Methode, der nur T kennt, muss sich auf Erfüllung verlassen können, auch wenn tatsächlich U verwendert wird).
  Ausnahme bei Objektvariablen, die Public sind - hier Äquivalenz benötigt (sollte aber vermieden werden).
- **History Constraint (Server-kontrolliert):** wie bei Invarianten: darf nur stärker sein z.B. T fordert, dass X immer auf ein höheres vielfaches von sich selbst gesetzt wird, U setzt immer auf ein vierfaches.
  Ausnahme bei Objektvariablen, die Public sind - hier Äquivalenz benötigt (sollte aber vermieden werden).
- **History Constraint (Client-kontrolliert):** wie bei Vorbedingungen: darf nur schwächer sein, jedoch bezogen auf die Reihenfolge der Methodenaufrufe (Trace). Es kann sein, dass U ein größeres Trace-Set erlaubt als T. Das Trace-Set in T muss also eine Teilmenge der Trace-Sets von U sein.

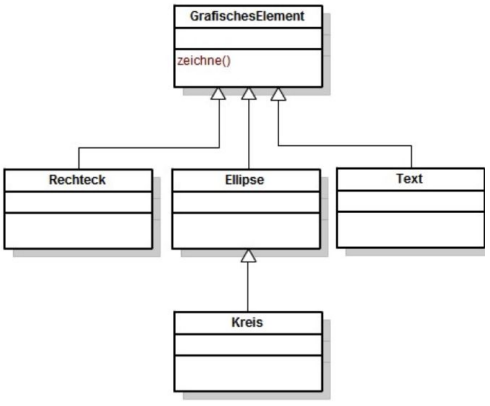
Es ist empfehlenswert, als Obertypen und Parametertypen hauptsächlich abstrakte Klassen (in Java vor allem Interfaces) zu verwenden.

Interfaces sind abstrakte Klassen, die nur abstrakte Methoden und static final Konstanten unterstützen (und sonst nichts), dafür aber Mehrfachvererbung erlauben.

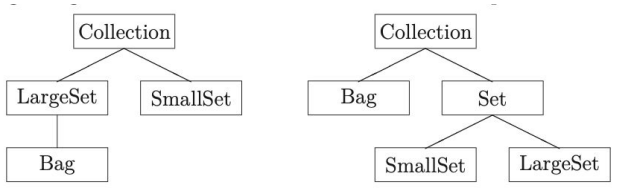
2.3. Vererbung versus Ersetzbarkeit

Klassen können in zumindest 3 Arten in Beziehung zueinander stehen:

- **Untertypen:** (siehe oben)
- **Vererbung:** Dabei entsteht eine neue Klasse durch Abänderung einer bestehenden Klasse. Es ist nicht nötig aber wünschenswert, dass Code aus der Oberklasse in der Unterklasser wiederverwendet wird.
- **Is-A-Beziehung (Reale-Welt-Beziehung):** Beziehungen, die sich intuitiv aus den Objekten der “realen Welt” ergeben. Werden meistens als Untertypbeziehungen implementiert, manchmal als Vererbungen. Manchmal stellt sich aber heraus, dass sich die Intuition nicht sinnstiftend übertragen lässt (z.B. Kreis-Ellipse-Problem)



Ziel der reinen Vererbung ist es, möglichst viel Code aus der Oberklasse in der Unterklasse wiederzuverwenden. Dies führt aber manchmal zu Vererbungshierarchien, die anders aussehen, als wenn man sich durch das Konzept von Typen zu einer Hierarchie leiten lässt.



Wiederverwendung durch das Ersetzbarkeitsprinzip ist wesentlich wichtiger als direkte Wiederverwendung durch Vererbung. Ein wichtiges Ziel ist die Entwicklung geeigneter Untertypbeziehungen. Vererbung ist ein Mittel zum Zweck. Sie soll sich Untertypbeziehungen unterordnen. Im Allgemeinen soll es keine Vererbungsbeziehung geben, die nicht auch eine Untertypbeziehung ist, bei der also alle Zusicherungen kompatibel sind.

Aber auch auch reine Vererbung kann sich positiv auf die Wartbarkeit auswirken, wenn sie richtig eingesetzt wird und sinnvolle Untertypbeziehungen nicht zu ihren eigenen Gunsten verdrängt.

2.4. Klassen und Vererbung in Java

	`public`	`protected`	—	`private`
sichtbar im selben Paket	ja	ja	ja	nein
sichtbar in anderem Paket	ja	nein	nein	nein
ererbbar im selben Paket	ja	ja	ja	nein
ererbbar in anderem Paket	ja	ja	nein	nein

Fast alle Variablen sollten private sein.
Protected nur einsetzen, wenn unvermeidbar (schlechter Stil, “My biggest mistake”)
So wenig Getter und Setter wie möglich verwenden (viele Getter/Setter deuten auf hohe Objektkopplung und niedrigen Klassenzusammenhalt hin).

Alle Variablen eines Objekts sind stets auch von einem anderen Objekt der selben Klasse zugreifbar. Direkte Zugriffe (vorallem Schreibzugriffe) dieser Art sollten jedoch vermieden werden.

3. Generizität und Ad-hoc-Polymorphismus

3.1. Generizität

Generische Klassen, Typen und Methoden enthalten Parameter, für die Typen eingesetzt werden. Andere Arten generischer Parameter unterstützt Java nicht. Daher nennt man generische Parameter einfach *Typparameter*.

3.1.1. Wozu Generizität?

z.B. generische Collection -> Einmal schreiben, öfter verwenden (anstatt Programmcode für verschiedene Typen zu kopieren)

3.1.2. Einfache Generizität in Java

Generische Klassen und Interfaces haben einen oder mehreren durch Beistrich getrennte Typparameter in spitzen Klammern direkt nach ihrem Namen stehen. Das Ersetzen des Typparameters durch einen Konkreten Typ führt zur Ersetzung aller Verwendungen dieses Typs innerhalb der Klasse bzw. des Interfaces. Der Typparameter kann durch Referenztypen ersetzt werden, nicht jedoch durch elementare Typen (int, char, boolean…).

3.1.3. Gebundene Generizität in Java

Bei einfacher Generizität ist im Rumpf einer Klasse oder Methode nichts über den Typ, der den Typparameter ersetzt, bekannt. Insbesondere ist nicht bekannt, ob Objekte dieser Typen bestimmte Variablen oder Methoden haben.

- **Schranken**
  - z.B.: <A extends B & T> (max. eine Klasse (B), mehrere Interfaces (T))
  - Innerhalb der Klasse/Methode ist nun bekannt, dass A ein Untertyp von Typen sein muss, die B erweitern und T implementieren und somit auf deren public Variablen/Methoden Zugreifen kann.
    In TypeScript z.B. auch <A extends B | T> möglich (Schnittmenge statt Vereinigung)
- **Rekursion:**
  - z.B. public class Integer implements Comparable<Integer> { … }
  - Klasse, referenziert sich selbst als Typparameter
  - sogenannte *F-gebundene Generizität*
- **Keine impliziten Untertypen:**
  - z.B. zwischen List<X> und List<Y> keine Untertypbeziehung, wenn X und Y verschieden sind, auch nicht, wenn X Untertyp von Y ist.

- Ausnahme sind Arrays, diese Kompilieren ohne Probleme und werfen zur Laufzeit eine ArrayStoreException

```
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
        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:**
  - Sicherheit durch impliziter Nichtunterstützung Untertypen hat einen Nachteil: void drawAll(List<Polygon> p) { ... } kann z.B. nicht mit Argumenten vom Typ List<Triangle> oder List<Square> aufgerufen werden.

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

    Es kann jedoch nur von p gelesen werden. Der Compiler liefert eine Fehlermeldung, wenn die Möglichkeit besteht, dass in p geschrieben wird bzw. erlaubt die Verwendung von p nur an Stellen für deren Typen in Untertypbeziehungen Kovarianz gefordert ist.

  - Gelegentlich gibt es auch Parameter, deren Inhalte in einer Methode nur geschrieben und nicht gelesen werden

    Lösung: void addSquares (List<? extends Square> from, List<? super Square> to) {...}

    Der Compiler erlaubt die Verwendung von to nun nur an Stellen, an denen Kontravarianz gefordert ist (Schreibzugriffe).

  - Verwendung auch ohne Schranken möglich, aber nur Lesen wird unterstützt <?> entspricht <? extends Object>

3.2. Verwendung von Generizität im Allgemeinen

3.2.1. Richtlinien für die Verwendung von Generizität

Generizität immer sinnvoll, wenn Wartbarkeit erhöht wird, z.B. bei gleich strukturierten Klassen und Methoden wie Containerklassen. Sobald der Verdacht aufkommt, dass etwas generisch sein sollte, dann am besten gleich refaktorisieren. Klassen und Methoden in Bibliotheken sollten generisch sein.

Man soll Typparameter als Typen formaler Parameter verwenden, wenn Änderungen der

Parametertypen absehbar sind (Beispiel Konto mit Euro-Beträgen -> sinnvoll gleich Typparameter für Währung zu verwenden anstatt Euro)

Untertypbeziehungen und Generizität sind in Java eng miteinander verknüpft. Die sinnvolle Verwendung gebunder Generizität setzt das Bestehen geeigneter Untertypbeziehungen voraus. Generizität und Untertyprelationen ergänzen sich. Man sollte stets überlegen, ob man eine Aufgabe besser durch eines der beiden oder beide Konzepte löst.

Generizität und Untertypbeziehungen ergänzen sich. Ohne Generizität sind keine (statisch überprüften) homogenen Listen möglich. Ohne Untertypbeziehungen sind keine heterogenen Listen möglich.
Untertypbeziehungen ermöglichen Code-Änderungen mit klaren Grenzen. Generizität kann betroffene Bereiche nicht eingrenzen, da Typparametersetzungen im Client-Code spezifiziert werden müssen. Deshalb im Zweifelsfall Untertypbeziehungen vorziehen.
Überlegungen zur Laufzeiteffizienz sollte man beiseite lassen. Mit Erfahrung kommt auch “Natürlichkeit” in Spiel.

3.2.2. Arten der Generizität

Für Übersetzung generischer Klassen 2 Möglichkeiten:

- Homogen: Eine generische Klasse wird in eine einzige nicht-generische Klasse übersetzt.
    - 1. spitze Klammern samt Inhalten weglassen
    - 2. Typparameter durch (erste) Schranke oder durch Object ersetzen
    - 3. Typumwandlungen bei Aufrufen einfügen wenn Ergebnis- oder Parametertyp Typparameter ist

```
class ListTest {
    public static void main(String[] args) {
        List xs = new List();
        xs.add((Integer)new Integer(0));
        Integer x = (Integer)xs.iterator().next();

        List ys = new List();
        ys.add((String)"zerro");
        String y = (String)ys.iterator().next();

        List zs = new List();
        zs.add((List)xs);
        List z = (List)zs.iterator().next();
    }
}
```

- Heterogen: Durch Copy-and-Paste eigene Klasse (Methode) pro Typparametersetzung. Erzeugt so viele nicht-generische Klassen Methoden) wie notwendig.
    - Vorteile:
        - effizienter, da keine Typumwandlung und Code optimierbar
        - int, char usw. direkt verwendbar
    - Nachteile:
        - oft viele Klassen und große Programme

3.3. Typabfragen und Typumwandlungen

3.1.1. Verwendung dynamischer Typinformation

In Objektorientierten Programmiersprachen (im Gegensatz zu prozeduralen und funktionalen Sprachen) wird für dynamische Binden die dynamische Typinformation eines Objekts zur Laufzeit benötigt.

In Java:

- getClass() -> liefert exakten Typ der Klasse der Objekts
- a instanceof T -> liefert Information ob Typ des Objekts a ein Untertyp von Typ T ist

Um auf public Methoden und Variablen von überprüften Objekten zugreifen zu können, gibt es in Java explizite Typumwandlungen als Casts auf Referenzobjekten (z.B. (T)a). Ein ungültiger Cast führt zur Laufzeit zu einer Exception. Statische Typüberprüfungen durch den Compiler werden an diesen stellen ausgeschaltet.

Das Abfragen auf dynamische Typinformation und Casten deutet jedoch meistens auf eine unsaubere Programmstruktur hin.

Auch schlechte Wartbarkeit ist ein Grund Typinformation nur sehr sparsam einzusetzen. Statt vielen *if instance of else if instance of…*-Abfragen sollte dynamisches Binden verwendet werden.

Typumwandlungen werden auch auf elementaren Typen wie int, char, float unterstützt. Hier findet jedoch eine tatsächlich eine Umwandlung des Werts, nicht nur des deklarierten Typs statt. Typumwandlungen zwischen Referenztypen und elementaren Typen werden in Java nicht unterstützt.

3.3.2. Typumwandlungen und Generizität

Siehe homogene Übersetzung.

Sichere Typumwandlungen:

- Up-Cast: Umwandlung in Obertyp des deklarierten Typs
- Down-Cast nach dynamischer Typabfrage (durch instanceof oder getClass())
  -> Hier Frage: was soll bei Scheitern der Typabfrage passieren?
  Außerdem oft sehr viele if-Abfragen notwendig
- Down-Cast wie bei Generizität bzw. homogener Übersetzung aber händisch überprüft
  -> Wenn die Programmiersprache Generizität unterstützt, dann nicht verwenden; sehr aufwändig und Fehleranfällig

Stets spitze Klammern verwenden, um nicht versehentlich Raw-Types (Typinformationen durch Übersetzung weggelassen) zu verwenden.

3.3.3. Kovariante Probleme

Typen von Eingangsparametern können nur Kontravariant sein. Kovariante Eingangsparameter verletzen das Ersetzbarkeitsprinzip. In der Praxis wünscht man sich aber manchmal genau diese. Diese Aufgabenstellungen nennt man kovariante Probleme.

Dyn. Typabfrage für kovariantes Problem

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

Überladene Methode

```
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 erhoeheWahrscheinlichkeitFuerBSE();
    }
}
```

Friss sind in Rind und Tiger überladen (mehrere Methoden mit dem selben Namen). Es wird die Methode ausgeführt, deren formaler Parametertyp der spezifischste Obertyp des *deklarierten Argumenttyps* ist.

Kovariante Probleme sollten vermieden werden.

Binäre Methoden sind ein häufig vorkommender Spezialfall von kovarianten Problemen.

```
abstract class Point {
    public final boolean equal(Point that) {
        if (that != null &&
              this.getClass() == that.getClass())
            return uncheckedEqual(that);
        return false;
    }
    protected abstract boolean uncheckedEqual(Point p);
}
class Point2D extends Point {
    private int x, y;
    protected boolean uncheckedEqual(Point p) {
        Point2D that = (Point2D)p;
        return x == that.x && y == that.y;
    }
}
class Point3D extends Point {
    private int x, y, z;
    protected boolean uncheckedEqual(Point p) {
        Point3D that = (Point3D)p;
        return x==that.x && y==that.y && z==that.z;
    }
}
```

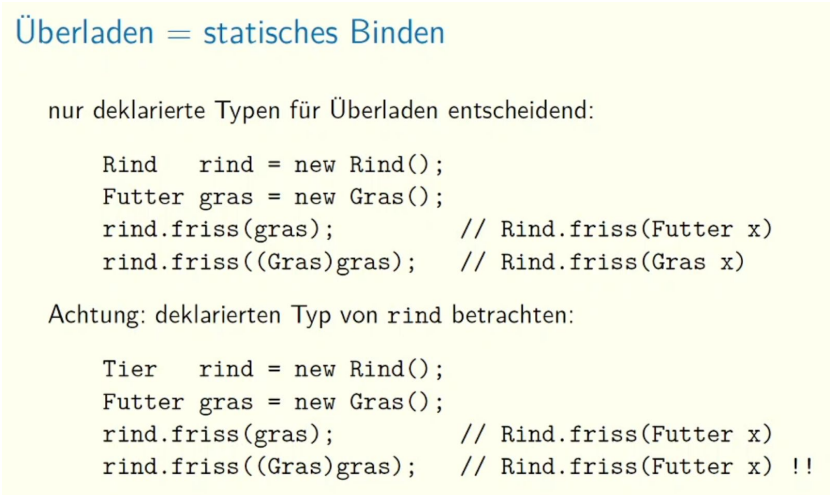
Point2D und Point3D haben nun gemeinsamen Obertyp Point.

3.4. Überladen vs Multimethoden

Dynamisches Binden erfolgt in Java über den dynamischen Typ eine Objekts.
z.B. x.equal(y) -> Auswahl der konkrete Methode hängt vom *dynamischen Typ von x* ab.
Aber auch vom *deklarierten Typ von y*, wenn equal überladen ist.

Generell, aber nicht in Java ist es auch möglich die Methodenauswahl abhängig vom Typ von y zu machen - dann kann der Compiler allerdings nicht mehr bestimmen, welche Methode zur Laufzeit ausgeführt werden wird -> man spricht dann von **Multimethoden**.

3.4.1 Deklarierte vs. dynamische Argumenttypen



Überladen = statisches Binden

nur deklarierte Typen für Überladen entscheidend:

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

Achtung: deklarierten Typ von `rind` betrachten:

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

Man soll Überladen nur so verwenden, dass es keine Rolle spielt, ob bei der Methodenauswahl deklarierte oder dynamische Typen der Argumente verwendet werden.

Das bedeutet für je zwei Methoden gleicher Parameterzahl

- gibt es zumindest eine Parameterposition in der sich die Parametertypen unterscheiden, wobei diese nicht in Untertypbeziehung zueinander stehen,
- oder alle Parametertypen der einen Methode sind Obertypen der anderen und bei Aufruf der einen wird nichts anderes gemacht als auf die andere zu verzweigen.

3.4.2 Simulation von Multimethoden

```
public abstract class Tier {
    public abstract void friss(Futter futter);
}
public class Rind extends Tier {
    public void friss(Futter futter) {
        futter.vonRindGefressen(this);
    }
}
public class Tiger extends Tier {
    public void friss(Futter futter) {
        futter.vonTigerGefressen(this);
    }
}
public abstract class Futter {
    abstract void vonRindGefressen(Rind rind);
    abstract void vonTigerGefressen(Tiger tiger);
}
public class Gras extends Futter {
    void vonRindGefressen(Rind rind) { ... }
    void vonTigerGefressen(Tiger tiger) {
        tiger.fletscheZaehne();
    }
}
public class Fleisch extends Futter {
    void vonRindGefressen(Rind rind) {
        rind.erhoeheWahrscheinlichkeitFuerBSE();
    }
    void vonTigerGefressen(Tiger tiger) { ... }
}
```

-> Doppeltes dynamisches Binden (Zuerst in Tier, dann in Futter), keine Typabfragen und keine Typumwandlungen notwendig. Entspricht dem Visitor-Pattern, wobei Methoden in Futter Visitormethoden sind und Tier Elementklasse (könnte auch umgekehrt implementiert sein). Nachteil: benötigte Anzahl von Methoden wird bei steigender Anzahl von Parametern enorm groß.

4. Kreuz und Quer

4.3. Annotation und Reflexion

Annotationen sind Markierungen im Programmcode, die von Laufzeitsystem und anderen Entwicklungswerkzeugen (zur Compile-Zeit) überprüft werden und dementsprechend darauf reagieren.

z.B. @Override verlangt von einer Methode, dass sie eine Methode der Oberklasse überschreibt, @Deprecated markiert eine Methode als nicht mehr zu verwenden (eigentlich nur ein Kommentar, aber durch Annotation kann der Compiler ein Warning ausgeben), @SuppressWarnings (Achtung, besser nicht verwenden),

Eigene Annotationen werden wie Interfaces nur mit einem “@” definiert.

- Mit @Target kann spezifiziert werden, auf welchen Arten von Operationen die Annotation verwendet werden darf (z.B. @Target({ElementType.METHOD, ElementType.CONSTRUCTOR}));
- Mit @Retention kann spezifiziert werden, ob die Annotation nur zur Compilezeit (RetentionPolicy.SOURCE) oder auch zur Laufzeit (RetentionPolocy.RUNTIME) zur Verfügung stehen soll.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface BugFix {
    String who();     // author of bug fix
    String date();    // when was bug fixed
    int    level();   // importance level 1-5
    String bug();     // description of bug
    String fix();     // description of fix
}
```

```
@BugFix(who="Kaspar", date="1.10.2020", level=3,
        bug="class unnecessary and may be harmful",
        fix="content of class body removed")
public class Buggy { }
```

Wenn die Annotation nur ein Argument namens value hat, dann kann die Angabe des Namens weg gelassen werden.

Man can default Werte für Parameter von Annotationen angeben.

```java
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Target(ElementType.ANNOTATION_TYPE)
public @interface Retention {
    /**
     * Returns the retention policy.
     * @return the retention policy
     */
    RetentionPolicy value();
}
```

Reflexion

Wurde mit @Retention(RetentionPolicy.RUNTIME) festgelegt, dass die Annotation zur Laufzeit zur Verfügung steht, so generiert der Compiler ein Interface, dass das vordefinierte Interface Annotation erweitert. Auf dieses kann mittels Reflexion zur Laufzeit zugegriffen werden (class.getAnnotation(s)(), class.getMethod(s)(), class.getField(s)()...)

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

Man kann wie in folgendem Code-Stück auf die Annotation zugreifen:

```java
    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";
    }
```

Reflexion erlaubt z.B. auch Methodenaufrufe (über getMethod().invoke), die die Methode “ganz normal” zur Laufzeit ausführen und dynamisch binden. Reflexion sehr mächtig, sollte jedoch sparsam eingesetzt werden, da undurchschaubar und kaum wartbar.

Reflexion ist eine Variante der **Metaprogrammierung**. “Richtige” Metaprogrammierung erlaubt es, auch den Programmcode selbst zur Laufzeit zu verändern (also nicht nur Informationen zu lesen).

4.4. Aspektorientierte Programmierung

Wird verwendet für Cross-Cutting-Concerns, die alle Bereiche eines Programms betreffen (z.B. Logging, Authentisierung, Debuggin) - Aspekte, die nicht Kernfunktionalität sind, aber für das Programm notwendig.

- **Join Point** - möglicher Ausführungspunkt in einem Programm (z.B. Methodenaufruf, Konstruktoraufruf, Variablenzugriff…)
- **Pointcut** - ausgewählter Join Point
- **Advice** - Code der vor oder nach dem gewählte Joint Point ausgeführt werden soll (mittels before(), after() oder around())
- **Aspekt** - Kombination aus Join Point, Pointcut und Advice

join point

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

pointcut

```
public pointcut accountOperation() : call(* Account.*(..))
    Schlüselwort    PCName          PCTyp     Signatur


execution(Signature)
call(Signature)
get(Signature)
set(Signature)
handler(Signature)
initialization(Signature)
cflow(Pointcut)
within(Signature)
```

advice

```
before() : Pointcut {Programmcode}
around()
after()


before() : call(* Account.*(..)) {Benutzer überprüfen}

pointcut connectionOperation(Connection connection) :
   call(* Connection.*(..) throws SQLException);
before(Connection connection) :
   connectionOperation(connection) {
   System.out.println("Operation auf" + connection);
   }
```

aspect

```
01 public aspect JoinPointTraceAspect {
02    private int _callDepth = -1;
03    pointcut tracePoints() : !within(JoinPointTraceAspect);
04    before() : tracePoints() {
05       _callDepth++;
06       print("Before", thisJoinPoint);
07    }
08    after() : tracePoints() {
09       _callDepth--;
00       print("After", thisJoinPoint);
11    }
12    private void print(String prefix, Object message) {
13       for (int i=0, spaces = _callDepth * 2; i < spaces; i++) {
14          System.out.print(" ");
15       }
16       System.out.println(prefix + ": " + message);
17    }
18 }
```

5. Software-Entwurfsmuster

5.2 Erzeugende Entwurfsmuster

5.2.1 Factory Method

https://www.youtube.com/watch?v=ub0DXaeV6hA

Die **Factory Method** abstrahiert die Objekterzeugung. Eine Factory Klasse wird als Schnittstelle für die Objekterzeugung verwendet, wobei Unterklassen entscheiden, von Welcher Klasse die erzeugten Objekte sein sollen. Ein Objekt wird also durch den Aufruf eine Methode anstatt durch den direkten Aufruf des Konstruktors erzeugt.

Die Methode, die zur Erzeugung eines Objekts aufgerufen wird, kann parametrisiert werden, um konkrete gewünschte Unterklassen zu erzeugen
(z.B. pizzaFactroy.createPizza(“Salami”).

Durch Lazy Initialization kann verhindert werden, dass ein Objekt, dass bereits erzeugt wurde noch einmal neu erzeugt wird (null-Abfrage vor Erzeugung).

5.2.2 Prototype

https://www.youtube.com/watch?v=AFbZhRL0Uz8

Das Entwurfsmuster **Prototype** wird verwendet, um ein neues Objekt durch kopieren (clone) eines bestehenden Prototype-Objekts zu erzeugen. Jede Konkrete Klasse von Prototype (welche Cloneable extended) muss clone implementieren.

Achtung: die default Implementierung von clone in Object erzeugt flache Kopien (Referenzen bleiben Referenzen), keine tiefen Kopien. Das Erzeugen tiefer Kopien (jede Variable rekursiv geklont) kann sehr komplex sein (zyklische Referenzen).

Wird eingesetzt wenn:

- die Klassen, von denen Objekte erzuegt werden sollen erst zur Laufzeit bekannt sind,
- vermieden werden soll, dass eine Hierarchie von Creator-Klassen zu erzeugen, die einer parallelen Hierarchie von Product-Klassen entspricht (Factory-Method soll vermieden werden,
- jedes Objekt einer Klasse nur wenige unterschiedliche Zustände haben kann (kopieren dann weniger umständlich als initialisierung mit new)

Nach Erzeugung einer Kopie ist es oft notwendig den Objektzuständ zu ändern. Dies ist mit clone nicht möglich (im Gegensatz zur Verwendung des Konstruktors). Methoden zur Initialisierung müssen dann implementiert werden.

Vermeidet eine große Zahl an Unterklassen. Es empfiehlt sich einen Prototyp-Manager zu verwenden.

5.2.3. Singleton

https://www.youtube.com/watch?v=NZaXM67fxbs

Das Entwurfsmuster **Singleton** sichert zu, dass eine Klasse nur eine einzige Instanz hat und erlaubt globalen Zugriff auf dieses Objekt. Besteht aus einer einzigen Klasse mit ein einer statischen Methode “instance”, welche das einzige Objekt der Klasse zurück gibt.

```
public class Singleton {
    private static Singleton singleton;
    protected Singleton() {  // kein Aufruf von außen
        singleton = null;
    }
    public static Singleton instance() {
        if (singleton == null)
            singleton = new Singleton();
        return singleton;
    }
}
```

Inzwischen eher als Anti-Pattern angesehen.

- erlaubt kontrollierten Zugriff auf einzige Instanz
- vermeidet unnötige Namen im System (keine globale Variable)
- unterstützt Vererbung (aber Achtung, führt schnell zu Problemen, z.B. Drucker-Spooler Beispiel: nach Instanzierung von SingletonA keine weitere Instanzierung von z.B. SingletonB möglich)
- leicht änderbar, wenn doch mehrere Instanzen notwendig
- flexibler als statische Methoden

5.3. Entwurfsmuster für Struktur und Verhalten

5.3.1. Decorator

https://www.youtube.com/watch?v=j40kRwSm4VE

Decorator (Wrapper) stellen eine flexible Alternative zur Vererbung bereit (*Composition over Inheritance*). Beispiel Pizza und Toppings.

Anwendbar

- um dynamisch Verantwortlichkeiten bzw. Funktionen zu einzelnen Objekten hinzuzufügen (bzw. entziehen), ohne andere Objekte dadurch zu beeinflussen
- wenn Erweiterungen einer Klasse durch Vererbung unpraktisch sind (z.B. Vermeidung sehr großer Zahl an Unterklassen)

```
interface Window { void show (String text); }
class WindowImpl implements Window {
    public void show(String text) { ... }
}
abstract class WinDecorator implements Window {
    protected Window win;
    public void show(String text) { win.show(text); }
}
class ScrollBar extends WinDecorator {
    public ScrollBar(Window w) { win = w; ... }
    ...
}
...
    Window myWindow = new WindowImpl(); // no scroll bar
    myWindow = new ScrollBar(myWindow); // add scroll bar
```

Bietet mehr Flexibilität als statische Vererbung.

Gut geeignet für Oberflächliche Veränderungen, schlecht geeignet für inhaltliche Veränderungen (Beispiel Torte aus Video - Glasur sehr einfach veränderbar, Mehl jedoch schwierig).

5.3.2. Proxy

https://www.youtube.com/watch?v=cHg5bWW4nUI

Ein Proxy stellt einen Platzhalter auf ein anderes Objekt dar und kontrolliert den Zugriff auf dieses. z.B.

- Remote-Proxy: Platzhalter für Objekte, die z.B. auf einem anderen Rechner existieren
- Virtual Proxy: Erzeugung eines Objekts ist sehr teuer und soll verzögert werden
- Protection-Procy: aus Sicherheitsgründen sollen nur gewisse Methoden verfügbar sein sollen und andere nicht

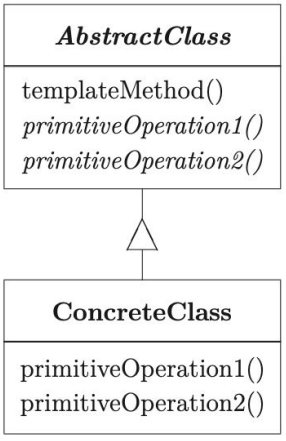
- Smart-References: ersetzt einfache Zeiger durch Zeiger mit Zusatzfunktion (als Proxy) (z.B. für Mitzählen der Referenzen, Laden von Objekten in einen Speicher, Locks)

Ein Proxy kann die gleiche Struktur wie ein Decorator haben. Ein Decorator erweitert ein Objekt jedoch um zusätzliche Funktionalität/Verantwortlichkeit, währen ein Proxy den Zugriff auf ein Objekt kontrolliert.

5.3.3. Template-Method

https://www.youtube.com/watch?v=aR1B8MlwbRI

Die **Template-Method** definiert das Grundgerüst eines Algorithmus, überlässt die Implementierung einiger Schritte aber einer Unterklassse (Beispiel Computerspiel Schießen -> Op1 = Animation, Op2 = Damage machen)



- Der unveränderliche Teil des Algorithmus muss nur einmal implementiert werden.
- Template Methods können auch Hooks (leere Methoden, die nur zur Überschreibung von Unterklassen da sin) verwenden. Dh. eine Unterklasse kann an dieser Stelle Code einfügen, muss es aber nicht.

Template-Methods führen zu einer umgekehrte Kontrollstruktur - die Oberklasse ruft die Methoden der Unterklasse auf (*Hollywood Prinzip - Don't call us, we call you*).

Es muss genau definiert sein, welche Operationen Hooks sind (dürfen überschrieben werden), welche abstrakt sind (müssen überschrieben werden) und welche nur in AbstractClass implementiert sein sollen. Die Template-Methode selbst kann final sein. Die Anzahl der primitiven Operationen sollte möglichst klein gehalten werden, um die Wiederverwendbarkeit zu erhalten.