Definitiv kein OOP Schummelzettel

Ersetzbarkeit

- **Ersetzbarkeitsprinzip**
  Angenommen ein Typ U ist ein Untertyp eines Typen T, so kann U überall dort verwendet werden, wo T erwartet wird. (Beziehungsweise ist U ein Untertyp von T, wenn dies möglich ist)

- **Untertypbeziehungen**
  1) Jeder Typ ist ein Untertyp von sich selbst.
  2) Für jede Konstante A in T gibt es eine entsprechende Konstante B in Untertyp U, wobei B ein Untertyp von A sein muss. (**Kovarianz**)
  3) Für jede Variable in T gibt es eine entsprechende Variable in Untertyp U, wobei die deklarierten Typen gleich sein müssen. (**Invarianz**)
  4) Für jede Methode in T gibt es eine entsprechende Methode in Untertyp U, wobei der deklarierte Ergebnistyp der Methode in U ein Untertyp vom Ergebnistyp der Methode in T ist;
     Die Anzahl der formalen Parameter der beiden Methoden gleich ist;
     Der deklarierte Typ einer Methode in U ein Obertyp des deklarierten Typs in T ist (**Kontravarianz**)
     Die Methode in U nicht mehr Exceptions wirft als die in T

     Wichtig: Ergebnistypen kovariant; formale Parameter invariant in Java, in C++ nicht. In Java ist alles Invarianz bis auf Ergebnistypen.

- **Kovarianz**
  Der deklarierte Typ eines Elements im Untertyp ist ein *Untertyp* des deklarierten Elements im Obertyp.

- **Kontravarianz**
  Der deklarierte Typ eines Elements im Untertyp ist ein *Obertyp* des deklarierten Elements im Obertyp.

- **Invarianz**
  Der deklarierte Typ eines Elements im Untertyp ist gleich dem deklarierten Typ des deklarierten Elements im Obertyp.

  https://www.complang.tuwien.ac.at/franz/objektorientiert/oop13-02.pdf

Zusicherungen

Können meist vom Compiler nicht überprüft werden – muss vom Programmierer umgesetzt werden. Zusicherungen werden meist durch Kommentare beschrieben.

- ### Vorbedingungen
  Zuständig: Client
  Beispiel: Eine sortierte Liste muss reingegeben werden
  Muss gelten bevor die Methode ausgeführt wird.

  Untertyp: Vorbedingung können schwächer, aber nicht stärker werden (OR)

- ### Nachbedingungen
  Zuständig: Server
  Beispiel: Eingefügtes Element muss in Return-Array vorhanden sein.
  Gibt den Zustand an, der nach dem Aufruf der Methode gegeben sein muss.

  Untertyp: Vorbedingungen können stärker, aber nicht schwächer werden (AND)

- ### Invarianten
  Zuständig: Server
  Beispiel: Guthaben am Sparbuch muss positiv sein
  Eigenschaften, die vor und nach der Ausführung einer Methode gelten müssen.

  Der Server kann die Invarianten nicht garantieren, wenn Client direkt auf öffentlich zugänglich Variablen zugreift.

  Untertyp: Vorbedingungen können stärker, aber nicht schwächer werden (AND)

- ### History-Constraints
  Zuständig: Server und Client
  Bedingungen, die die Entwicklung von Objekten im Laufe der Zeit einschränken

  Server-kontrolliert:
  Ähneln Invarianten, schränken aber zeitliche Veränderungen der Variableninhalte eines Objekts ein.
  Beispiel: Zählervariable kann nur größer werden

  Untertyp: Vorbedingungen können stärker, aber nicht schwächer werden (AND)

  Client-kontrolliert:

Reihenfolge von Methodenaufrufen einschränken, nur Clients können Methoden aufrufen und deren Reihenfolge bestimmen
Beispiel: a() wird vor b() aufgerufen

Untertyp: Vorbedingungen können schwächer, aber nicht stärker werden (OR)

Generizität

Anstelle von konkreten Typparametern werden generische Typen verwendet, um maximale Wiederverwendbarkeit zu ermöglichen.

- **Einfache Generizität**
  Nur mit einfachen generischen Typen möglich (<T>), aber nur allgemeine Dinge möglich. Generizität bietet statische Typsicherheit – Instanz von List<String> nimmt nur Strings.

  Für Methoden: <T> vor Typparameter (public <T> T test(…) {)

- **Gebundene Generizität**
  Erlaubt Schrankensetzung, welche Untertypen der Schranke zulässt.
  Beispiel: public <T extends Futter> String test(…) { lässt sowohl Futter als auch alle Untertypen von Futter (Gras) zu. Wenn kein extends angegeben ist, wird in Java automatisch von Object extended. Typparameter dürfen auch rekursiv verwendet werden. (<A extends Comparable<A>): F-Gebundene Generizität.
  Typparameter unterstützen keine implizite Untertypbeziehung. Kann durch gebundene Wildcards wieder aufgehoben werden.

  Beispiel:
  void drawAll(List<? *extends* Polygon> p) nimmt alle Typparameter, die Untertypen von Polygon sind. (Nur lesen, Kovariant)
  void addPolygons(List<? *super* Polygon> p) nimmt alle Typparameter, die Obertypen von Polygon sind. (Nur schreiben, Kontravariant)

  In Java kann man mit Typparametern keine neuen Objekte erzeugen.

Generizität ist überall dort sinnvoll, wo es Wartbarkeit erhöht. Man sollte sie dort verwenden, wo gleich strukturierte Klassen oder Methoden vorhanden sind, weil Wartungsaufwand reduziert wird. Listen, Stacks, etc.

Generizität sollte dann gewählt werden, wenn man weiß, dass sich Parametertypen ändern werden. (Bank will anfangs nur Euro-Noten verwalten. Später muss Dollar dazu genommen werden – Umschreiben geht sehr schnell, da generischer Parameter gewählt wurde)

Übersetzungen der Generizität

- **Homogene Übersetzung**
  Beispielsweise in Java.
  Übersetzt eine generische Klasse in genau eine Klasse mit JVM-Code und übersetzt die möglichen Werte für den generischen Wert zur Laufzeit.

- **Heterogene Übersetzung**
  Beispielsweise in C++.
  Übersetzt eine generische Klasse in alle möglichen Kombinationen, dadurch ist die Compilierzeit länger, aber das Programm zur Laufzeit schneller.

Kovariante Probleme

Typen von Eingangsparametern können nur kontravariant sein, weil Kovarianz bei Eingangsparametern das Ersetzbarkeitsprinzip verletzt. Benötigt man nun kovariante Eingangsparameter, so spricht man von einem *kovarianten Problem*.

Kovariante Probleme lassen sich unter anderem durch dynamische Typabfragen und dynamische Typumwandlungen lösen.

Binäre Methoden

Eine Methode ist eine binäre Methode, wenn die Methode einerseits auf *this* zugreifen kann und andererseits die Klasse, in der sich die Methode befindet, als Eingangsparameter vorhanden ist.

Überladen

Methoden in Java werden nicht überschrieben, sie werden überladen. Überladene Methoden müssen bis auf den Namen keine Ähnlichkeit besitzen – welche Methode dann ausgewählt wird, hängt von den Eingangsparametern ab.

Eine Routine heißt Überladen, wenn sie Argumente mit unabhängigen Typen akzeptiert und sich für jeden Typen anders verhalten kann. Überladene Methoden betrachtet nur den deklarierten Typen, auch wenn der Dynamische bekannt ist. Man sollte überladene Methoden jedoch so verwenden, sodass es keine Rolle spielt, ob deklarierter oder dynamischer Typ benutzt werden.

Stichwort: Statisches Binden.

Multimethoden

Multimethoden verwenden dynamisches Binden. Werden in Java nicht unterstützt, können aber simuliert werden, indem man mehrfaches dynamisches Binden durch wiederholtes einfaches Binden anwendet. Während die Methodenauswahl bei der Überladung einzig und allein von dem deklarierten Typen des Parameters abhängt, kann durch Multimethoden der dynamische Typ herangezogen werden.

Siehe: Skriptum S. 155 & S. 160

Entwurfsmuster

Entwurfsmuster dienen zur Wiederverwendung kollektiver Erfahrung – bewährte Muster, die für Programmstrukturen sinnvoll sind.

Bestandteile eines Entwurfsmusters:

- **Name**: geeigneter Name, dass das Muster beschreibt
- **Problemstellung**: Beschreibung des Problems & Einsatzmöglichkeiten
- **Lösung**: Lösungsbeschreibung
- **Konsequenzen**: Eigenschaften der Lösung & Vor- und Nachteile

Erzeugungsmuster

Factory Method (Virtual Constructor)

Objekt wird anstelle von Constructoraufruf durch Methodenaufruf erzeugt.

Die Methode selbst befindet sich in einer Factory Klasse. Jede Klasse (bzw. Klassenebene) besitzt ihre eigene Creator/Factory-Klasse; somit entsteht eine Parallelhierarchie mit der normalen Hierarchie auf der einen und der Creator-Klassenhierarchie auf der anderen Seite.

Benötigt, wenn:

- Eine Klasse Objekte erzeugen soll, deren Klasse aber nicht kennt
- Wenn Allokation und Freigabe (Organisation) von Objekten zentral in einer Klasse verwaltet werden soll
- Klassen Verantwortlichkeiten an eine von mehreren Unterklassen delegieren

Factory Methods sind flexibel, allerdings sind meist eine große Anzahl an Unterklassen nötig, die diese Objekte generieren

Prototype

Objekte werden mittels einer Kopie eines vordefinierten Prototyp-Objekts erzeugt. Das dient dazu, die neu generierten Objekte genau zu spezifizieren. Dabei können bestimmte Rechenschritte übersprungen werden, die man eigentlich für ein Objekt benötigen würde, indem man ein bereits vorhandenes Objekt in den Prototype hineinkopiert und aus diesem Prototype danach wieder Objekte erzeugt.
Es sollten Deep Copies, keine Shallow Copies gemacht werden, da sonst mit Referenzen gearbeitet wird.

- Klassen müssen Cloneable Interface implementieren.
- Können zur Laufzeit verändert werden
- Prototype-Zustand kann sich ändern
- Vermeidet große Anzahl an Unterklassen
- Reduziert Anzahl der Klassen, die der Anwender kennen muss

Benötigt, wenn:

- Klasse des neuen Objekts erst zur Laufzeit bekannt
- Vermeidung von Doppelhierarchie wie bei Factory Method
- Erzeugende Klassen sich nur in wenigen Parametern unterscheiden/wenige unterschiedliche Zustände annehmen

Es ist einfacher, für jeden möglichen Zustand einen Prototyp zu erzeugen und diesen zu kopieren, als durch einen Constructor-Aufruf ein Objekt zu erzeugen und dabei die richtigen Parameter anzugeben

Singleton

Singleton Klasse hat nur eine einzige Instanz und erlaubt einen globalen Zugriff auf dieses Objekt. Ein Singleton besteht aus einer gleichnamigen Klasse mit einer statischen Methode instance(), init(), etc., welche das einzige Objekt der Klasse zurückgibt.

```
class Singleton {
        private static Singleton instance; // Singleton Objektvariable
        private Singleton() { } // privater Konstruktor
        public static Singleton get() { // Initialisierungs/Anforderungsmethode
                if (Singleton.instance == null) Singleton.instance = new Singleton();
                return Singleton.instance;
        }
}
```

Singleton besitzt einen privaten Konstruktor, der verhindert, dass das Singleton über andere Methoden erzeugt wird. Erlaubt kontrollierten Zugriff auf das einzige Objekt. Nützlich um Daten an einem Fleck zu haben.

Strukturelle Entwurfsmuster

Decorator

Gibt Objekten dynamisch zusätzliche Verantwortlichkeiten, indem der Decorator eine Instanz des zu verändernden Objekts speichert. Durch diesen Decorator können nun mittels extends weitere Features angebunden werden, da ein Aufruf von dem Überobjekt durch die Instanz im Unterobjekt referenziert wird. Das in der Hierarchie weiter unten befindliche überschreibt somit die Implementierung des Überobjekts.

Decorator werden benutzt, um dynamisch Verantwortlichkeiten zu einzelnen Objekten hinzuzufügen, ohne andere Objekte dadurch zu beeinflussen. Sie sind auch nützlich, da diese Verantwortlichkeiten schnell wieder entfernt werden können, ohne das Programm zu beeinflussen.

Weiters bieten Decorator mehr Flexibilität als statische Vererbung – bei Decorators findet dies zur Laufzeit statt und beeinflusst nur einzelne Objekte, nicht ganze Klassen.

Proxy

Proxys stellen Platzhalter für andere Objekte dar und kontrollieren die Zugriffe darauf. Ist ein Objekt sehr kostspielig zu erstellen, weil viele Daten geladen werden müssen, kann man sich mit einem Proxy Abhilfe schaffen. Man benutzt das richtige Objekt erst dann, wenn und falls es wirklich gebraucht wird.

Arten von Proxys/Situationen, in denen Proxys benutzt werden:

- **Remote Proxys**
  Platzhalter für Objekte, die in anderen Namensräumen (z.B. auf Festplatten oder anderen Rechnern) existieren. Nachrichten werden an diese Objekte von den Proxys gehandled.
- **Virtual Proxies**
  Erzeugen Objekte bei Bedarf (siehe oben).
- **Protection Proxies**
  Kontrollieren Zugriffe auf Objekte. Sinnvoll, wenn nur bestimmte Zugreifer (User/Methoden/etc.) unterschiedliche Zugriffsrechte auf ein Objekt haben sollen.
- **Smart References**
  Mehrere Einsatzgebiete:

a) Mitzählen von Referenzen auf das eigentliche Objekt, damit Objekt entfernt werden kann, wenn es keine Referenzen mehr gibt (Reference-Counting & Garbage Collecting)
b) Das Laden von großen Objekten in den Speicher (Unterschied zu Virtual Proxy schwammig)
c) Thread-Locking

Wichtig: Proxy kontrolliert Zugriff auf Objekt, Decorator erweitert Objekt

Entwurfsmuster für Verhalten

Iterator

Ermöglicht sequentiellen Zugriff auf ein Aggregat (Sammlung von Elementen) ohne die innere Darstellung offenzulegen. Mit Iteratoren ist es möglich, mehrere (gleichzeitige) Abarbeitungen der Elemente zu ermöglichen. Ebenso ermöglichen Iteratoren eine einheitliche Schnittstelle für verschiedene Strukturen. (Polymorphie)

Wichtige Eigenschaften:

1. Mehrere Iteratorklassen pro Aggregatklasse möglich
2. Schnittstelle wird vereinfacht
3. Mehrere Abarbeitungen gleichzeitig pro Aggregat möglich, da jeder Iterator den Zustand für sich selbst speichert

<u>Interner vs. Externer Iterator:</u>

- **Intern**
  Interne Iteratoren enthalten die Schleife selbst, next und hasNext sind nicht zugänglich. Man übergibt dem Iterator eine Operation (eine Funktion z.B.), die für alle Elemente einzeln angewandt wird. Bsp.: Map.forEach()
- **Extern**
  Anwender bestimmen, wann das nächste Element abgearbeitet wird. Sind flexibler als interne Iteratoren – zwei Aggregate miteinander zu vergleichen ist bspw. Einfacher mit Externen. Externe Iteratoren erschweren Parallelverarbeitung.

Aggregatänderungen während Iteratorzugriff kann gefährlich sein, da durch Änderung Elemente ausgelassen oder doppelt gelesen werden können. Dafür gibt es *robuste Iterator*, die entweder vor dem Lesen eine Kopie anlegen, oder das Aggregat so lange sperren, bis die Leseaktion abgeschlossen ist.

Template-Method

Überklasse (sehr häufig abstrakt) enthält unveränderlichen Teil einer Methode und ruft dabei abstrakte Methoden auf, die dann in den jeweiligen Unterklassen ausprogrammiert werden.

- Dabei sollten alle Methoden, die überschrieben sollen werden, *abstract* sein.
- Ausdefinierte Methoden sollten *protected* sein, um nicht in anderen Kontexten zu erscheinen.
- Die Hauptmethode kann *final* sein.
- Anzahl der „primitiven Methoden“ (also Methoden, die überschrieben werden müssen) sollte so klein wie möglich sein.
- Methoden, die überschrieben werden können, aber nicht überschrieben werden müssen, nennt man *Hooks* – sie werden schon in der Klasse ausprogrammiert.

Dienen zur Duplikatvermeidung, da gemeinsames Verhalten an einer Stelle lokal zusammengefasst wird.

Fundamentale Technik zur direkten Wiederverwendung: In Klassenbibliotheken und Frameworks sehr sinnvoll für Faktorisierung.

Hollywood-Prinzip: Oberklasse ruft Unterklasse auf („Don’t call us, we’ll call you“)

Visitor

Bei der Erweiterung eines Programms um neue Operationen müssen viele Klassen erweitert werden. Das Visitor-Pattern lagert die Operationen auf externe Besucherklassen aus, die über eine zentrale Schnittstelle auf das Hauptprogramm zugreifen:

Es wird eine abstrakte Besuchsklasse definiert, die für jede Klasse eine Besuchsmethode besitzt. Diese Methoden werden dann in spezifischen Besuchern ausprogrammiert.

Vorteile:

- Neue Operationen lassen sich leicht hinzufügen
- Verwandte Operationen werden im Besucher lokal verwaltet (Andere Besucher haben einen anderen Zweck)
- Besucher können mit Objekten aus voneinander unabhängigen Klassenhierarchien arbeiten