Ausarbeitung OOP

1. Grundlagen und Ziele

1.1 Konzepte objektorientierter Programmierung

Leichte Wartbarkeit und einfache Änderbeitkeit von Programmen bei Softwareentwicklungsprozessen.

1.1.1 Objekte

Ein Objekt ist wie eine **Kapsel** die aus **Variablen** und **Routinen** besteht (Zusammenfügung dieser nennt man **Kapselung**). Zugriff auf das Objekt sollte nur von außen stattfinden indem man eine nach außen sichtbare Routine aufruft.

Eigenschaften:

- *Identität* → unveränderlich gekennzeichnet, Objekte sind identisch wenn sie auf dasselbe Objekt (in einem bestimmen Speicherplatz) verweisen, aber zwei verschiedene Namen besitzen.

- *Zustand* → bestimmt durch Werte der Variablen. Zwei Objekte können gleich (equal) sein, ohne ident zu sein (gleiche Werte, verschiedene Objekte).

- *Verhalten* → Ändert sich durch Eingriff der Routinen in das Objekt (meist von außen). Diese müssen auch implementiert werden, Programmlogik bis ins kleinste Detail, da Compiler sonst nicht weiß was er tun soll.

- *Schnittstelle* → Passend dem abstraktionsgrad des menschlichen Denkens, definiert man abstrakte Methoden die erst später, durch Objekte die das Interface implementieren, ausprogrammiert werden.

Somit ist ein Objekt im Grunde eine **Blackbox**, wo man keine Details der Implementierung kennt (**data hiding**), aber die Methoden die man aufrufen kann.

1.1.2 Klassen

Jedes Objekt gehört einer Klasse, die die Implementierung derer beinhalten und mittels Konstruktoren Objekte erzeugen. Objekte sind **Instanzen** einer Klasse. Instanzen sind **nicht ident**, haben unterschiedliche Instanzvariablen.

1.1.3 Polymorphismus

Eine Variable oder Routine hat **mehrere** Typen (Vererbung und Interfaces).

- *Deklarierter* Typ → Typ der mit einer Variable deklariert wurde – existiert nur bei expliziter Typdeklaration

- *Statischer* Typ → In vielen Fällen ordnet der Compiler der Variable an verschiedenen Stellen verschiedene Datentypen zu.

- *Dynamischer* Typ → Spezifischster Typ, Werden unter anderem für die Typüberprüfung zur Laufzeit verwendet.

Arten von Polymorphismus:

Universelle Polymorphismen (spezifisch für OOP)

- *Generizität* → Ausdrücke enthalten Parameter wie List<A>, Typparameter ist nun durch Integer zb ersetzbar → generierte int liste

- *Enthaltender Polymorphismus* → subtyping bzw. Vererbung. Ein Typ U ist ein Untertyps eines Typs T (bzw. T ist Obertyp von U), wenn eine Instanz von U überall verwendbar ist, wo eine Instanz von T erwartet wird. Auszuführende Methoden werden erst bei der Programmausführung festgestellt (dynamisch).

Ad-hoc Polymorphismen

- *Überladen* → Routinen mit gleicher Funktionalität und Namen, aber verschiedenen (Anzahl von) Parametern.

- *Typumwandlung* → Umwandlung eines Wertes in ein Argument welches die Routine erwartet.

1.1.4 Vererbung

Ableitung einer neuen Klasse (**Unterklasse**) von einer bestehenden Klasse (**Oberklasse**) → kein doppelter Code, bessere Wartbarkeit. Unterklassen können **nicht beliebig verändert** werden (im Gegensatz zu Oberklassen):

- *Erweiterung* → Die Unterklasse erweitert die Oberklasse um neue Variablen, Methoden und Konstruktoren.

- *Überschreiben* → Methoden der Oberklasse werden überschrieben.

1.2 Qualität in der Programmierung

Gliederung in: Qualität der Programme und Effizienz / Wartung dieser.

1.2.1 Qualität von Programmen

Brauchbarkeit:
Wert des erstellen Programms sollte aus der Sicht des Users, weit die Entwicklungskosten übersteigen.

Beeinflussung durch:

- *Zweckerfüllung* → Alle geforderten Features des Anwenders sollten enthalten sein.
- *Bedienbarkeit* → Arbeitsschritte, Einlernaufwand, Wartezeiten etc
- *Effizienz* → Je weniger Ressourcen vom PC benötigt werden, desto hochwertiger ist das Programm

Zuverlässigkeit:
Nur Programme die gewartet werden, werden auch auf Dauer genutzt. Dementsprechend ergeben sich bis zu 70% mehr Gesamtkosten, bei sehr erfolgreicher Software sogar viel mehr. Faustregel: Gute Wartbarkeit kann die Gesamtkosten erheblich reduzieren.

Faktoren die eine Rolle spielen:

- *Einfachheit* → Komplexität so gering wie möglich halten, da dadurch Änderungen und Wartbarkeit erheblich vereinfacht werden.

- *Lesbarkeit* → vom Programmierstil abhängig – Logik sollte „leicht" nachvollziehbar sein.

- *Lokalität* → Programmänderungen sollten keine weitreichenden (unüberblickbare) Auswirkungen auf den restlichen Code haben.

- *Faktorisierung* → Mehrfach genutzte Programmlogik sollte in einer Routine ausgelagert werden, da einmalige Änderungen dann alle Aufrufe beeinflusst → bessere Wartbarkeit. Refaktorisierung ist spätere Faktorisierung, da neue Zusammenhänge klar werden.

Man sollte soweit die „reale Welt" simulieren, solange die Einfachheit nicht gefährdet ist!

1.2.2 Effizienz der Programmerstellung und Wartung

Softwareentwicklungprozess aka **Wasserfallmodell (+ Wartung)**:

- *Analyse* → Anforderungsdokumentation beschreibt die Anforderungen an die Software
- *Entwurf* → Programm wird entworfen
- *Implementierung* → Entwurf wird in ein Programm umgesetzt → Implementierung
- *Verifikation* und *Validierung* → Überprüfung der Erfüllung der Punkte im Anforderungsdokument und wie gut das Programm die Aufgaben der Anwender wirklich löst.

Sehr nützlich bei kleineren Projekten, mit sehr klaren Anforderungen.

Zyklische Softwareentwicklungsprozesse:
Zyklische Arbeit mit den Punkten aus dem Wasserfallmodell. Meist überlappend wird gleichzeitig analysiert, entworfen, implementiert und überprüft. Beginnend mit einem kleinen Teil des Programms, bei stetiger Ausweitung → **schrittweise Verfeinerung**

Zyklische Prozesse verkraften Anforderungsänderungen besser, aber Zeit und Kosten sind schwer planbar.
In der Praxis hat jedes Unternehmen eigene Standards.

1.3 Rezept für gute Programme

Es existiert eigentlich kein allgemein gültiges Rezept für gute Programme! Erfahrung ausschlaggebend.

1.3.1 Zusammenhalt und Kopplung

Entwickler müssen zu jedem Zeitpunkt wissen wie sie Vorgehen müssen, um hochwertige Software zu erstellen. Einige Faustregeln:

- *Verantwortlichkeiten* → einer Klasse („was ich weiß“ (Zustand der Instanzen), „was ich mache“ (Verhalten der Instanzen), „wen ich kenne“ (sichtbare Objekte, Klassen etc).

- *Klassenzusammenhalt* → Grad der Beziehungen zwischen den Verantwortlichkeiten einer Klasse, sobald man wichtige Methoden oder variable entfernt, fehlt der Klasse etwas „wichtiges“

- *Objekt – Kopplung* → Abhängigkeit der Objekte voneinander. Stark wenn hohe Anzahl von nach außen sichtbare Methoden vorhanden, im System Nachrichten und Zugriffe unterschiedlicher Objekte auftreten, Anzahl der Parameter dieser Methoden groß ist.

Faustregel: Klassenzusammenhalt soll hoch sein, Objektkopplung schwach! Ein vernünftiges Maß rechtzeitiger Refaktorisierungen, führt häufig zu gut faktorisierten Programmen.

1.3.2 Wiederverwendung

Gut faktorisierte Programme lassen sich leichter Wiederverwenden. Bewährte Software sollte man öfters einsetzen. Arten von Software die wiederverwendet werden können:

- *Programme* → Entwicklung darauf ausgelegt das Programm öfters zu nutzen
- *Daten* → Daten in Datenbanken und Dateien werden auch öfters benötigt.
- *Erfahrungen* → Ideen und Konzepte wieder einsetzen.
- *Code* → Wiederverwendung von Globalen Bibliotheken, Fachspezifische, Projektinterne Wiederverwendung, Programminterne Wiederverwendung.

Faustregel: Code-Wiederverwendung erfordert beträchtliche Investitionen in die Wiederverwendbarkeit. Man soll diese tätigen, wenn ein tatsächlicher Bedarf dafür absehbar ist

1.3.3 Entwurfsmuster

Erfahrung ausschlaggebend. Einsatz von **Design Patterns** bzw. Entwurfsmustern sehr beliebt, besteht aus: *Name, Problemstellung, Lösung, Konsequenzen*

Faustregel: Entwurfsmuster sollen zur Abschätzung der Konsequenzen von Designentscheidungen eingesetzt werden und können (in begrenztem Ausmaß) als Bausteine zur Erzielung bestimmter Eigenschaften dienen.

1.4 Paradigmen der Programmierung

Gemeint ist der Programmierstil an sich. Unterscheidung in:

Imperative Programmierung
wird dadurch charakterisiert das Programme aus Anweisungen (Befehlen) aufgebaut sind. Diese werden in einer festgelegten Reihenfolge ausgeführt. Untergruppen der imperativen Programmierung sind **Prozedurale** Programmierung, die den konventionellen Programmierstil beschreibt. Programme sind in sich gegenseitig aufrufende Prozeduren zerlegt. Programmzustände werden als global gesehen, das heißt Daten können überall im Programm verändert werden. **Objektorientierte** Programmierung ist eine Weiterentwicklung der strukturierten prozeduralen Programmierung und stellt den Begriff des Objekts in den Mittelpunkt. Der wesentliche Unterschied zur prozeduralen Programmierung ist, dass Routinen und Daten zu Objekten zusammengefasst werden. Dies macht die Wartung einfacher, hat aber zur Konsequenz das ein Algorithmus manchmal nicht nur an einer Stelle im Programm steht, sondern aufgeteilt sein kann. *Destruktive Zuweisung* → Variable bekommt neuen Wert, egal was vorher für ein Wert da war.

Deklarative Programmierung
Entstammen mathematischen Modellen, Grundlegende Sprachelemente sind *Symbole* die sich manchmal in mehreren Gruppen wie Variablensymbole, Funktionssymbole und Prädikate einteilen lassen. Kommt bei **funktionaler** und **logikorientierter** *Programmierung* zum Einsatz.

Faustregel: Objektorientierte Programmierung eignet sich zur Entwicklung von Systemen, deren Gesamtkomplexität jene der einzelnen Algorithmen deutlich übersteigt. Sonst sind andere Paradigmen besser geeignet.

1.4.3 Paradigmen für Modularisierungseinheiten

Kombinierbar mit imperativen und deklarativen Paradigmen:

- Programmierung mit abstrakten Datentypen
- Programmierung mit Modulen
- Komponentenprogrammierung
- Generische Programmierung

2. Enthaltender Polymorphismus und Vererbung

Wichtigste Grundlage ist das **Ersetzbarkeitsprinzip**:
Ein Typ U ist ein Untertyp eines Typs T, wenn überall eine Instanz von Typ U verwendbar ist, wo eine Instanz von Typ T erwartet wird.

Man erreicht höhere Flexibilität beim erweitern von Klassenkomplexen (Beispiel Grafiktreiber), Schnittstellen müssen nicht geändert werden -> genauere Implementierung in Untertypen.

Benötigt man bei:

- Aufruf einer Routine mit Argument dessen Typ ein Untertyp des Typs des formalen Parameters ist.
- Zuweisung eines Objektes an einer Variable, wobei Typ des Objektes Untertyp des deklarierten Typs der Variable ist.

Bedingungen für das Bestehen einer Untertypsbeziehung:
reflexiv → jeder Typ ist Untertyp von sich selbst
transitiv → Ist Typ U Untertyp eines Typs S und S Untertyp vom Typ T, so ist U auch Untertyp vom Typ T
antisymmetrisch → Ist T Untertyp von U und U Untertyp von T, so sind T und U gleich.

Untertyp U von T ist, wenn:

- Für jede Konstante in T gibt es eine entsprechende Konstante in U.
- Für jede Variable in T gibt es eine Variable in U, wobei die deklarierten Typen dieser Variable gleich sind.
- Für jede Methode in T gibt es eine entsprechende Methode in U.

Untertyp kann nicht nur um neue Elemente erweitern, er kann auch deklarierte Typen gegenüber des Obertypen ändern → deklarierten Typen der Elemente können **variieren**:

Weitere Bedingung dass U Untertyp von T ist (Zusicherungen):

- *Kovarianz* → (Nachbedingung) Der deklarierte Typ eines Elements im Untertyp ist ein Untertyp des deklarierten Typs des entsprechenden Elements im Obertyp. Typen variieren in dieselbe Richtung.

- *Kontravarianz* → (Vorbedingung) Der deklarierte Typ eines Elements im Untertyp ist ein Obertyp des deklarierten Typs des Elements im Obertyp. Typen variieren also in entgegengesetzte Richtungen.

- *Invarianz* → Die deklarierten Typen eines Elements im Untertyp ist gleich dem dem deklarierten Typs eines Elements im Obertyp.

Ist in einer Methode der formale Parametertyp stets gleich der Klasse, so spricht man von einer *binären* Methode. **Kovariante Eingangsparametertypen** und binäre Methoden widersprechen dem Ersetzbarkeitsprinzip und daher sinnlos dieses anzustreben.

Stabilität bei Schnittstellen ist wichtig, da eine Änderung auch zu Änderungen in den darauf aufbauenden Typen zu nötigen Änderungen führt. Besonders wichtig ist die Stabilität an der **Wurzel der Typhierarchie**. Man sollte Untertypen nur von stabilen Obertypen bilden. Faktorisierungen helfen dabei.

2.1.3 Dynamisches Binden

Man spricht von dynamischem Binden wenn ein Methodenaufruf zur Laufzeit anhand des tatsächlichen Typs eines Objekts aufgelöst wird. Dynamisches Binden wird anstellen von switch Anweisungen und geschachtelten if Anweisungen verwendet und erhöht die Lesbarkeit und damit auch die Wartbarkeit.

2.2 Ersetzbarkeit und Objektverhalten

Weitere Bedingungen für die Ersetzbarkeit die der Compiler nicht überprüft.

2.2.1 Client-Server-Beziehungen

Meisten Objekte spielen gleichzeitig beide Rollen. Objekt als Server austauschbar, wenn das neue Objekt dieselben Dienste anbietet → **Objektverhalten** ausschlaggebend.
Schwer zu ermitteln bzw nicht exakt definierbar was Objekt machen soll, wenn Nachricht angekommen ist → Einsatz von **Verträgen** („Design by Contracts“).

Regeln für jeden Methodenaufruf (Zusicherungen, assertions):

- *Vorbedingungen* → Client verantwortlich. Beschreiben hauptsächlich welche Bedingungen das Argument beim Methodenaufruf erfüllen muss.

- *Nachbedingungen* → Server verantwortlich. Beschreibet den Zustand nach einer Änderung.

- *Invarianten* → Server verantwortlich für Erfüllung der Bedingung vor und nach Methodenaufruf. Schreibzugriffe auf Variablen des Servers hat aber nur der Client. Get/Setter Methoden oder Refaktorisierung im Sinne einer besseren Struktur (Objektkopplung)

Zusätzlicher Bestandteil eines Typs! **Zusammenfassung**: *ein Typ besteht aus dem Namen einer Klasse, eines Interfaces oder eines einfach Typs, der entsprechenden Schnittstelle und den dazugehörenden Zusicherungen.*

Zusicherungen sollten stabil sein, den Klassenzusammenhalt maximieren und Objektkopplung minimieren.

2.2.2 Untertypen und Verhalten

Auch folgende Bedingungen müssen gelten damit U Untertyp von T ist:

- Jede Vorbedingungen auf einer Methode in T muss eine Vorbedingungen auf der entsprechenden Methode in U implizieren. Vorbedingungen in U können schwächer , in T aber müssen sie stärker sein. Umsetzung mittels ODER Verknüpfung

- Jede Nachbedingungen impliziert eine Nachbedingung in T. In U muss sie stärker als in T sein, damit T nachbedingungen automatisch erfüllt sind. Umsetzung mittels UND Verknüpfung.

- Invarianten in Untertypen können stärker als in Obertypen sein. Umsetzung analog wie bei Nachbedingungen.

Zusicherungen müssen unmissverständlich formuliert sein und permanent im Hinterkopf des Programmierers bleiben.

2.2.3 Abstrakte Klassen

Sind wie übliche Klassen, allerdings kann man davon keine Instanzen erzeugen. Klassen die von dieser ableiten müssen alle Methoden beinhalten bzw. abstrakte implementieren. Je weniger Code die abstrakte Klasse hat, desto stabiler ist sie.

2.3.1 Reale Welt vs. Vererbung vs. Ersetzbarkeit

Drei verschiedenen Arten von Beziehungen zwischen Klassen:

- *Untertypbeziehungen* → siehe oben
- *Vererbungsbeziehungen* → Codewiederverwendung, ignoriert Ersetzbarkeitsprinzip
- *Reale-Welt-Beziehung* → Analog zur realen Welt (zb Student ist auch eine Person)

2.3.2 Vererbung und Codewiederverwendung

Bei **Vererbung** wird scheinbar eine Kopie der Oberklasse angelegt und die Klasse durch Überschreiben und Erweitern abgeändert. Vererbung ist zur direkten Wiederverwendung von Code einsetzbar und damit unabhängig vom **Ersetzbarkeitsprinzip**. Eine Klasse die von einer anderen erbt muss nicht dort Einsetzbar sein, wo die Oberklasse einsetzbar ist.

Faustregel: Wiederverwendung durch das Ersetzbarkeitsprinzip ist wesentlich wichtiger als direkte Wiederverwendung durch Vererbung.
(Verzichtet man auf Ersetzbarkeit wird die Wartung erschwert da sich kleine Änderungen auf das gesamte Programm auswirken können. Viele Vorteile der objektorientierten Programmierung gehen damit verloren).

2.4 Exkurs: Klassen und Vererbung in Java

Begriffe (fett markiert):

1. **Instanzvariablen** sind Variablen die zu den Instanzen einer Klasse gehören. Werden durch Hinschreiben eines Typs gefolgt vom Namen und Strichpunkt deklariert.

2. **Klassenvariablen** gehören nicht zu einer Instanz, sondern zur Klasse selbst und werden mit Voranstellen von static deklariert.

3. **Statische Methoden** werden über den Namen einer Klasse aufgerufen, nicht über den Namen einer Instanz. Sie gehören ebenfalls zur Klasse, nicht zur Instanz.

4. **Static Initializer** besteht nur aus dem Schlüsselwort static gefolgt von geschwungenen Klammern. Diese Sequenz wird vor der ersten Verwendung der Klasse ausgeführt.

5. **Geschachtelte Klassen** sind Klassen innerhalb einer andere Klasse. Sie können überall dort definiert sein wo auch Variablen definiert werden dürfen.

6. **Final Klassen und Methoden** können in Unterklassen nicht überschrieben werden.

7. Jede compilierte Klasse in Java wird in einer eigenen Datei gespeichert. Das Verzeichnis, das diese Datei enthält, entspricht dem **Paket**, zu dem die Klasse gehört.

8. Nur auf Interfaces wird **Mehrfachvererbung** unterstützt. Im restlichen Java wird nur **Einfachvererbung** unterstützt.

9. Es dürfen beliebig viele **imports** am Anfang einer Datei mit Quellcode stehen, sonst aber nirgends. Man kann entweder einzelne Pakete lokal bekanntmachen durch zb „import myclass examples.test“. Dann sind alle public Methoden von Klassen im Paket test bekannt und mittels test.Klasse.methode() aufrufbar. Eine Klasse lässt sich mittels „import myclass.examples.test.Klasse“ importieren. Ein Aufruf wäre dann mit Klasse.methode() möglich. Man kann aber auch alle Klassen in einem Paket importieren mittels „import myclass.examples.test.*“ Auch danach lässt sich mit Klasse.methode() jede public Methode der Klasse aufrufen.

10. Es gibt die Möglichkeit eine Methode public zu definieren, dann ist sie überall **sichtbar** und werden vererbt. private Methoden sind nur in dem Bereich sichtbar und verwendbar in dem sie erstellt wurden. protected Methoden sind im selben Paket sichtbar, aber nicht in anderen Paketen.

11. **Interfaces** sind eingeschränkte **abstrakte Klassen** in denen alle Methoden abstrakt sind. Sie beginnen mit dem Schlüsselwort interface. Normale Variablen dürfen nicht deklariert werden, wohl aber als static final. Alle Methoden sind public, das Schlüsselwort kann man weglassen da alles andere Verboten ist. Nach dem Schlüsselwort extends können mehrere, durch Komma getrennte Namen von Interfaces stehen. Dadurch gibt es hier Mehrfachvererbung. Interfaces sollten immer dann eingesetzt werden wenn die oben beschriebenen Einschränkungen zu keinen Nachteilen führen.

3. Generizität und Ad-hoc-Polymorphismus

Generische Klassen, Typen und Routinen enthalten Typparameter, für die später Typen eingesetzt werden. **Generizität** erspart also Schreibarbeit. Generizität eine weitere Form des *universellen* **Polymorphismus**, die Wiederverwendbarkeit unterstützt. Statischer Mechanismus !

3.1.2 Einfache Generizität in Java

- Klassen und Interfaces können einen oder mehrere Typparameter (in spitzen klammern), getrennt von Beistrichen, besitzen.
- Einfache Typen wie int, char brauchen Angabe von Referenzobjekten, also Integer, Character etc… Seit Java 1.5 gibt es aber autoboxing und unboxing die den Typ in beiden Richtungen automatisch ändert.

Generizität beseitigt dynamische Typfehler und gleichzeitig wird die Lesbarkeit des Programms erhöht.

Typinferenz = Vorgang zur Berechnung eines Typs

3.1.3 Gebundene Generizität in Java

Bei gebundener Generizität wird einem Typparameter noch eine **Schranke** hinzugefügt. Nur Untertypen dieser Schranke dürfen dann den Typparameter ersetzen. Das führt zu **gebundenen Typparametern** die zusätzliche Informationen liefern und so Zugriff auf mehr und spezifischere Methoden ermöglichen.. Schranke wird nach dem Schlüsselwort extends geschrieben. Wenn nichts angegeben ist default Schranke Object, da jede Klasse von Object erbt.

F-Gebundene Generizität = Form der Generizität mit rekursiven Typparametern. Keine **implizite** Untertypsbeziehungen möglich, sogar wenn ein Typ vom anderen erbt. Zum Beispiel besteht zwischen List<Y> und List<Y> keine Untertypbeziehungen wenn X und Y verschieden sind, auch dann nicht, wenn Y von X abgeleitet ist. Anders hingegen **explizite** Untertypsbeziehung, wie üblich ala: class MyList<A> extends List<List<A>> { ... }

Gebundene Wildcards = Wildcards werden als **Platzhalter** für Typen in gebundener Generizität verwendet, in Zusammenhang mit einer oberen bzw unteren Schranke:
void drawAll (List<? extends Polygon> p) { ... }

3.2 Verwendung von Generizität im Allgemeinen

3.2.1 Richtlinien für die Verwendung von Generizität

Generell sinnvoll wenn es um die Verbesserung der Wartbarkeit geht.

Gleich strukturierte Klassen und Routinen = überall wo man sich andere Typen in derselben Klassenstruktur – Logik vorstellen könnte, gilt fast immer bei Collections, also sollte diese generisch sein. Standardbibliotheken sind in Java durchwegs generisch (Arrays, Stacks etc)

Abfangen erwarteter Änderungen = Man soll Typparameter als Typen formaler Parameter verwenden, wenn Änderungen der Parametertypen absehbar sind. Generizität und Untertyprelationen ergänzen sich. Man soll stets überlegen, ob man eine Aufgabe besser durch Ersetzbarkeit, durch Generizität, oder (häufig sinnvoll) eine Kombination aus beiden Konzepten löst.

Mittels Generizität kann man statisch sicherstellen dass der Typ gleich bleibt, **homogen**. Mittels Generizität und OHNE Untertypsbeziehungen kann man keine **heterogene** Collections zb erstellen.

Laufzeiteffizienz = Einsatz von Generizität hat kaum negative Einflüsse auf die Laufzeit.

statische Typsicherheit = Es wird zur Kompilierzeit überprüft ob irgendwelche Type Conversion Errors auftreten (zb Zuweisung eines dynamischen Obertypen zu dem zugehörigen deklarierten Untertypen).

3.2.2 Arten der Generizität

Bei der Übersetzung generischer Klassen und Routinen in ausführbaren Code, gibt es die homogene und heterogene Möglichkeit. Java nutzt homogene Übersetzung:

Homogene Übersetzung
Dabei wird jede generische Klasse mit JVM Code übersetzt. Jeder **gebundene** Typparameter wir im übersetzten Code einfach durch die (erste) Schranke des Typparameters ersetzt, jeder **ungebundene** Typparameter durch Object. Wenn eine Methode eine Instanz eines Typparameter zurück gibt, wird der Typ der Instanz nach dem Methodenaufruf dynamisch in den Typ, der den Typparameter ersetzt, umgewandelt. Dies entspricht der Simulation einiger Aspekte von Generizität. Im Unterschied zur simulierten Generizität wird die Typkompatibilität aber vom Compiler garantiert.

Heterogene Übersetzung
Für jede generische Klasse oder Routine wird eigener übersetzter Code erstellt. Entspricht im Wesentlichen dem **copy / paste** um Typumwandlungen durchzuführen. Dem Nachteil einer großen Anzahl übersetzter Klassen und Routinen steht der Vorteil gegenüber, da für alle Typen eigener Code erzeugt wird, sind einfache Typen (wie int, char, …) problemlos, ohne Einbußen an Laufzeiteffizienz, als Ersatz für Typparameter geeignet. Zur Laufzeit brauchen keine Typumwandlungen und damit zusammenhängende Überprüfungen durchgeführt werden. Außerdem sind auf jede übersetzte Klasse eigene Optimierungen anwendbar.

3.3 Typabfragen und Typumwandlungen

In OOP wird dynamische Typinformation für das dynamische Binden zur Ausführungszeit benötigt.

3.3.1 Verwendung dynamischer Typinformation

Überprüfung ob ein Objekt Instanz einer Oberklasse ist mittels Schlüsselwort „**instanceof**“ oder „**getClass**“. Typabfragen und Typumwandlung sollten nach Möglichkeit vermieden werden.

3.3.2 Typumwandlung und Generizität

Die Typumwandlung wandelt den deklarierten Typ in einen anderen Typ (meist den gewünschten dynamischen Typ) um, da dem deklarierten Typ keine Details über die spezifischere Implementierung im dynamischen Typ bekannt ist, wodurch er nicht auf diese Funktionalität zugreifen kann.

Sichere Formen der Typumwandlung sollte man nur einsetzen, diese sind sicher wenn:

- Wenn in einen Obertyp des deklarierten Objekttyps umgewandelt wird. (**up-cast**, selten gebraucht)
- Davor erfolgt eine **dynamische Typabfrage**, die sicherstellt, dass das Objekt einen entsprechenden dynamischen Typ hat. (alle Annahmen stehen im alternativen Zweig als Zusicherungen)
- Im Vornherein wird **der Code so geschrieben, als ob man Generizität verwendet** bzw homogene Übersetzung durchführt. Es muss sichergestellt werden, dass jedes Vorkommen des Typparameters durch denselben Typ ersetzt wird und es dürfen keine impliziten Untertypbeziehungen vorkommen.

Durch dynamisches Binden sind solche Typecasts vermeidbar, da der Compiler hierbei den dynamischen Typ automatisch aufruft. **Probleme**: der deklarierte Typ ist zu weit oben in der Typhierarchie -> sehr viele Möglichkeiten des dynamischen Bindens die nicht alle abgedeckt werden können, darüber hinaus muss die Methode für das dynamische Binden in sämtlichen Untermethoden mitimplementiert werde. Die Typhierarchie kann nicht erweitert werden --> keine weitere Möglichkeiten alternative Lösungszweige durch dynamische Methodenaufrufe zu lösen Zugriff der Methoden auf private content

3.3.3 Kovariante Probleme

Verletzen das Ersetzbarkeitsprinzip, es bieten sich dynamische Typabfragen und Typumwandlungen. **Kovariante Probleme** beschreiben die Problemstellung, dass eine Modellierung einer gewissen Situation die Kovarianz der Eingangsparameter abverlangt (sprich man will in den Untertypen speziellere Parameter haben, um die Vorteile des dynamischen Bindens genießen zu können). Die Löschung der Schnittstelle löst hier das Problem, sodass man keine Typecast Checks durchführen muss. Man kann in so einem Fall nurmehr die Methoden mit den konkreten Untertypen aufrufen.

3.4 Überladen versus Multimethoden

Man soll **Überladen** nur so verwenden, dass es keine Rolle spielt, ob bei der Methodenauswahl deklarierte oder dynamische Typen der Argumente verwendet werden. Während man beim **Überschreiben** die entsprechende Methode im Obertypen neu definiert (diese ist trotzdem mit super aufrufbar), wird beim Überladen ein und die selbe Methode für mehrere Parameter implementiert. Es wird anhand dem deklarierten (!) Übergabetypen ermittelt welche Methode aufgerufen werden soll. Dies ist auch der Unterschied zu **Multimethoden**, bei welchem neben der zu wählenden Klasse auch der Eingangsparameter dynamisch ermittelt wird.

Simulation von Multimethoden: Multimethoden verwenden mehrfaches dynamisches Binden: Die auszuführende Methode wird dynamisch durch die Typen mehrerer Argumente bestimmt. In Java gibt es nur einfaches dynamisches Binden. Trotzdem ist es nicht schwer, mehrfaches dynamisches Binden durch wiederholtes einfaches Binden zu simulieren.

Das **Visitor Pattern** ist ein klassisches Entwurfsmuster. Es beschreibt die Simulierung von Multimethoden. Hierbei wird die Klasse welche durch direktes dynamisches Binden festgestellt wir als Visitorklasse, sowie Klassen die durch indirektes dynamisches Binden festgestellt werden als Elementklasse. Visitor- und Elementklassen sind oft gegeneinander austauschbar. Der große Nachteil am Visitor Entwurfsmuster ist, dass die Anzahl der benötigten Methoden schnell sehr groß wird.

3.5 Ausnahmenbehandlung

Ausnahmen sind in Java gewöhnliche Objekte, die über spezielle Mechanismen als Ausnahmen verwendet werden. Alle Instanzen von **Throwable** sind dafür verwendbar. Praktisch verwendet man nur Instanzen der Unterklassen **Error** und **Exception**. Unterklassen von Error werden hauptsächlich für vordefinierte schwerwiegende Ausnahmen des Java-Laufzeitsystems verwendet und deuten auf echte Fehler hin. Es ist praktisch kaum möglich, solche Ausnahmen abzufangen.

Unterklassen von Exception sind in zwei Bereiche gegliedert.

- Ausnahmen die von Programmierern selbst definiert wurden
- Ausnahmen die Instanzen von RuntimeException sind.

Oft ist es sinnvoll Instanzen von Exception abzufangen und den Programmablauf an geeigneter Stelle fortzusetzen.

Ausnahmen sind kontravariant, sprich im Untertyp dürfen nur weniger Exceptions geworfen werden als im Obertyp.

3.5.2 Einsatz von Ausnahmebehandlungen

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

Aus Gründen der Wartbarkeit soll man Ausnahmen und Ausnahmebehandlungen nur in echten Ausnahmesituationen und sparsam einsetzen. Man soll Ausnahmen nur einsetzen, wenn dadurch die Programmlogik vereinfacht wird.

3.6 Nebenläufige Programmierung

Mehrere Threads (statt einem) laufen scheinbar parallel (auf 1 Prozessor Kern) und wirklich parallel (multicore). Aufwändiger zu programmieren, da man auch auf die Synchronität achten muss, sonst können Werte inkonsistent werden. Schlüsselwort „synchronized“. Java verwendet „locks“ um die Synchronisation zu realisieren.

Probleme entstehen , wenn **Deadlocks** auftreten … Eigenschaften, die die Abwesenheit unerwünschter gegenseitiger Behinderungen von Threads in einem Programm betreffen, kennt man zusammengefasst **Liveness Properties**.

4. Softwareentwurfsmuster

Entwurfsmuster gegliedert in:

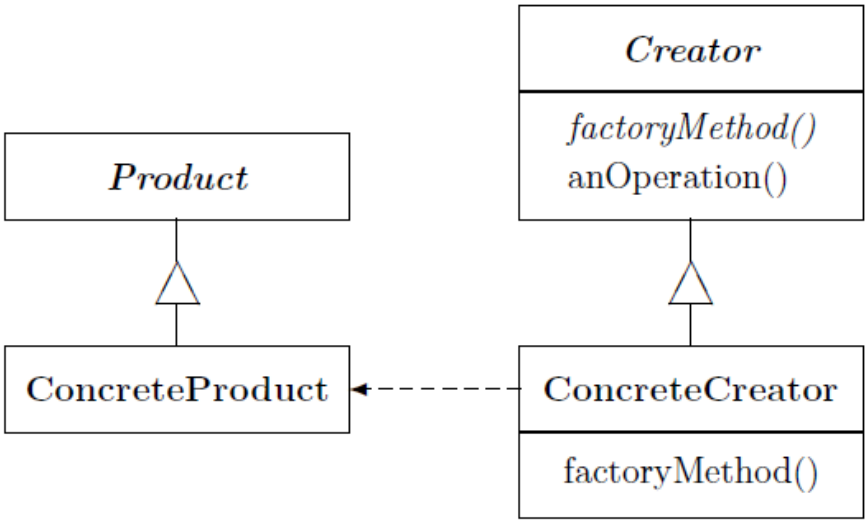
- Muster zur Erzeugung neuer Objekte (**creational patterns**)
- Muster die die Struktur der Software beeinflussen (**structural patterns**)
- Muster die mit dem Verhalten von Objekte zu tun haben (**behavioral patterns**)

4.1 Erzeugende Entwurfsmuster

sind Factory Method, Prototyp und Singleton

4.1.1 Factory Methode

Auch Virtual Constructor genannt – Unterklassen entscheiden von welcher Klasse die erzeugten Objekte sein sollen.



Eigenschaften:

- Bieten Anknüpfungspunkte für Unterklassen an, erzeugung eines Objektes mittels Factory Method ist fast immer flexibler, als die direkte Objekterzeugung.
- Verknüpfung paralleler Klassenhierarchien, wobei dieser Effekt aber unerwünscht ist.

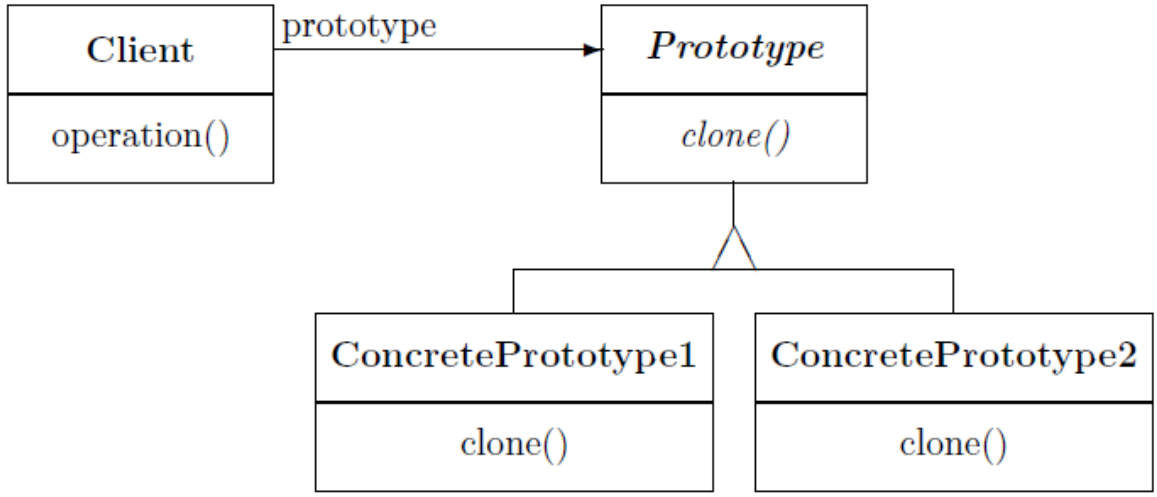
Entwurfsmuster anwendbar wenn:

- Eine Klasse neue Objekte erzeugen soll, deren Klasse aber nicht kennt
- Eine Klasse möchte dass ihre Unterklassen die Objekte bestimmen, die die Klasse erzeugt
- Klassen schicken die Verantwortlichkeiten an ihre Unterklassen, bewahren aber lokal das Wissen darüber.

Unterklassen implementieren Factory Method (abstrakt) oder diese ist selbe konkrete Klasse mit der Gefahr hin, die Factory Method zu überschreiben.

4.1.2 Prototype

Dient dazu Art eines neu erzeugten Objektes durch ein Prototyp Objekt zu spezifizieren. Neue Objekte werden durchs kopieren dieses Objekts erzeugt.



Eigenschaften:

- Reduzieren die konkreten Produktklassen vor den Anwendern und reduzieren damit die Anzahl der Klassen.
- Prototypen können zur Laufzeit jederzeit hinzugefügt oder entfernt werden.
- Spezifikation neuer Objekte durch änderbare Werte.
- Vermeidung vieler Unterklassen – parallele Klassenhierarchien sind nicht notwendig
- Erlauben dynamische Konfiguration von Programmen.

Entwurfsmuster anwendbar wenn:

- Die Klassen, von denen Instanzen erzeugt werden sollen, erst zur Laufzeit bekannt sind.
- vermieden werden soll, eine Hierarchie von „Creator“-Klassen zu erzeugen, die einer parallelen Hierarchie von „Product“-Klassen entspricht (Factory Method)
- Wenn jede Instanz einer Klasse nur wenig unterschiedliche Zustände haben kann. Prototype werden kopiert.

Für dieses Entwurfsmuster ist es notwendig, dass jede konkrete Unterklasse von „Prototype“ die Methode „**clone**“ implementiert. Diese ist bereits in Object standardmäßig definiert. Sind aber nur flache statt echten Kopien, die man (aufwendig) implementieren muss. Um die Übersicht über Prototypen zu behalten, haben sich **Prototyp-Manager** bewährt, das sind assoziative Datenstrukturen (kleine Datenbanken), in denen nach geeigneten Prototypen gesucht wird.

4.1.3 Singleton

Sichert zu dass eine Klasse nur eine Instanz hat (muss implementiert werden) und erlaubt globalen Zugriff auf diese.

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

Eigenschaften:

- Kontrollierter Zugriff auf einzige Instanz
- Verzicht auf globale Variablen, da eben nur eine Instanz
- Unterstützt Vererbung
- Je nach Implementierung sind auch mehrere Instanzen möglich, Kontrolle darüber bleibt aber in der einen Klasse.
- Flexibler als statische Methoden (erlauben kaum Änderungen, dynamisches Binden wird nicht unterstützt)

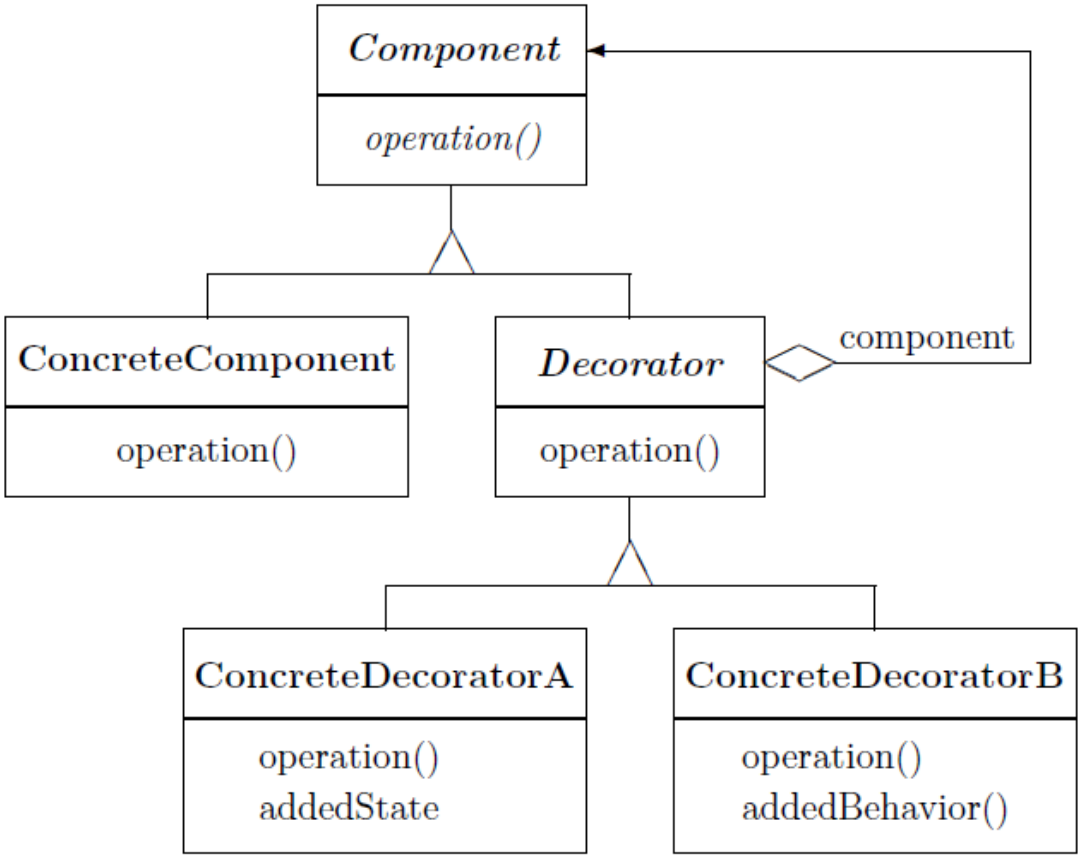
Anwendbar wenn:
Eine Instanz einer Klasse existieren soll und diese global zugreifbar sein soll.
Klasse durch Vererbung erweiterbar und ohne Änderungen verwendbar

Ein Singleton besteht nur aus einer gleichnamigen Klasse mit einer statischen Methode „**instance**“, welche die einzige Instanz der Klasse zurückgibt.

4.2 Strukturelle Entwurfsmuster

4.2.1 Decorator

Ist Alternative zu Vererbung, auch Wrapper genannt, gibt Objekten dynamisch zusätzliche Verantwortlichkeiten.



Eigenschaften:

- Verantwortlichkeiten werden während der Laufzeit zu einzelnen Objekten hinzugefügt, anstatt ganzen Klassen.
- Vermeiden Klassen die ganz oben in der Klassenhierarchie sind.
- Objektidentität nicht gegeben (Objekt selbst hat andere ID als Zugriff über Dekorator auf das gleiche Objekt)
- Führen zu vielen kleinen Systemen, welches sehr leicht zu konfigurieren ist aber sehr schwer zu verstehen und zu warten.

Anwendbar wenn:

- Dynamische Verantwortlichkeiten zu einzelnen Objekten hinzuzufügen, ohne andere Objekte zu beeinflussen.
- Für Verantwortlichkeiten die wieder entzogen werden.
- Wenn Vererbung unpraktisch wird, weil zb zu viele Unterklassen vorhanden sind.

Praktisch um ein Objekt besser zu beschreiben, aber unpraktisch wenn es um die inhaltliche Erweiterung dessen geht.

4.2.2 Proxy

Auch **surrogate** genannt, stellt einen Platzhalter für ein anderes Objekt dar und kontrolliert Zugriffe darauf. Objekt wird erst erzeugt, wenn es wirklich gebraucht wird (spart den Aufwand der Objekt Erzeugung.

Wo Proxys zum Einsatz kommen:

- *Remote Proxies* = sind Platzhalter für Objekte, die in anderen Namensräumen (zum Beispiel auf Festplatten oder auf anderen Rechnern) existieren
- Virtual Proxies = erzeugen Objekte bei Bedarf
- Protection Proxies = kontrollieren Zugriffe auf Objekte.
- Smart References = ersetzen einfache Zeiger

Klasse Proxy:

- Verwaltet eine Referenz auf „realSubject“, über die der Proxy auf dessen Instanzen zugreifen kann.
- Stellt Schnittstelle bereit, damit der Proxy als Ersatz für ein Objekt verwendet werden kann.
- Kontrolliert Zugriff und Erzeugung auf der Instanz
- Hat weitere Verantwortlichkeiten die von der Art abhängen.

Zusammengefasst: Proxy kontrolliert Zugriff auf das Objekt

4.3 Entwurfsmuster für Verhalten

4.3.1 Iterator

Auch Cursor genannt, ermöglicht Zugriff auf die Elemente eines Aggregats (Collection).

Anwendung bei:

- Zugriff auf Elemente eines Aggregats ohne innere Darstellung offen legen zu müssen.
- Sequentielle Abarbeitung in einem Aggregat zu ermöglichen
- Einheitliche Schnittstelle für Abarbeitung verschiedener Aggregatstrukturen , also polymorphe Iteration.

Java: Durchlaufen einer Collection mittels Iterator, mit einer „for each“ Schleife

4.3.2 Template Method

Grundgerüst eines Algorithmus hat Unterklassen, die bestimmte Schritte des Algos überschreiben.

Eigenschaften:

- Fundamentale Technik zur direkten Wiederverwendung vom Programmcode

- Führen zu einer umgekehrten Kontrollstruktur, Hollywood Prinzip („dont call us, we will call you“). Oberklasse ruft Methode der Unterklasse auf, nicht umgekehrt.
- Rufen eine von mehreren Arten von Operationen auf wie konkrete Operationen, Factory Methods, hooks.

Anwendung bei:

- Um den unverändlichen Teil des Algorithmus nur einmal zu implementieren.
- Refaktorisierungen in einer Klasse , um Duplikate im Code zu vermeiden
- Kontrolle bei Erweiterung des Algos in Unterklassen, mittels hooks.

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“.