Zusammenfassung Test 2

194.023 Programmierparadigmen 2023W

22. Jänner 2024

Inhaltsverzeichnis

4. Dynamische Typinformation und statische Parametisierung

Unterscheidung zwischen Typinformationen, die schon dem Compiler zur Verfügung stehen und dynamische Typinformationen, die erst zur Laufzeit zur Verfügung stehen.

In statisch typisierten prozeduralen und funktionalen Sprachen gibt es **keine** dynamische Typinformation.

In objektorientierten Programmiersprachen wird dynamische Typinformation benötigt für das **dynamische Binden** zur Ausführungszeit. In Java kann man z.B. direkt die dynamische Typinformation überprüfen (mit `instanceof`).

4.1 Generizität

Generische Klassen, Typen und Methoden enthalten Parameter, für die Typen eingesetzt werden (*Typparameter*). Kein dynamisches Binden erforderlich $\Rightarrow$ effizienter Einsatz, aber manchmal beim Programmieren eingeschränkt.

Wozu Generizität?

Spart Schreibaufwand und Wartungsaufwand. Compiler erzeugt Kopien von Codestücken, die sonst händisch erzeugt werden müssten.

In Java erzeugt der Compiler keine Kopien der Codestücke, sondern kann durch Typumwandlungen (Casts) den gleichen Code für mehrere Zwecke verwenden; daher hängt Generizität (als rein statischer Mechanismus) mit dynamischer Typinformation zusammen.

Java Beispiel

```
public interface Collection<A> {
    void add(A elem);       // add elem to collection
    Iterator<A> iterator(); // create new iterator
}
public interface Iterator<A> {
    A next();               // get the next element
    boolean hasNext();      // further elements?
}
```

Gebundene Generizität in Java

Schranken In Java kann für jeden Typparameter eine Klasse und beliebig viele Interfaces als *Schranken* angegeben werden. Nur Untertypen der Schranken dürfen den Typparameter ersetzen.

Beispiel:

```
public interface Scalable {
    void scale(double factor);
}
public class Scene<T extends Scalable> implements Iterable<T> {
    public void addSceneElement(T e) { ... }
    public Iterator<T> iterator() { ... }
    public void scaleAll(double factor) {
        for (T e : this)
            e.scale(factor);
    }
    ...
}
```

Rekursion Beispiel:

```
public interface Comparable<A> {
    int compareTo(A that); // this < that if result < 0
                           // this == that if result == 0
                           // this > that if result > 0
}
public class Integer implements Comparable<Integer> {
    private int value;
    public Integer(int value) { this.value = value; }
    public int intValue() { return value; }
    public int compareTo(Integer that) {
        return this.value - that.value;
```

```
    }
}
public class CollectionOps2 {
    public static <A extends Comparable<A>>
        A max(Collection<A> xs) {
            Iterator<A> xi = xs.iterator();
            A w = xi.next();
            while (xi.hasNext()) {
                A x = xi.next();
                if (w.compareTo(x) < 0)
                    w = x;
            }
            return w;
        }
}
```

Diese Form der Generizität mit rekursiven Typparametern nennt man *F-gebundene Generizität* nach dem formalen Modell, in dem solche Konzepte untersucht wurden: System $F_{\leq}$, ausgesprochen „F-bound“.

Keine impliziten Untertypen Keine Untertypbeziehung zwischen `List<X>` und `List<Y>` wenn Y Untertyp von X ist oder umgekehrt.

Natürlich gibt es explizite Untertypbeziehungen wie

```
class MyList<A> extends List<List<A>> { ... }
```

Dann ist `MyList<String>` ein Untertyp von `List<List<String>>`. Jedoch ist `MyList<X>` kein Untertyp von `List<Y>` wenn Y möglicherweise ungleich `List<X>` ist. Die Annahme impliziter Untertypbeziehungen ist ein häufiger Anfängerfehler.

Z.B. kann folgender Fehler ohne Generizität passieren:

```
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];
    }
}
```

Hier kommt es zu einem Fehler, da der Compiler annimt, dass `String[]` Untertyp von `Object[]` ist, da `String` ein Untertyp von `Object` ist. Diese Annahme ist falsch. Generizität schließt solche Fehler durch das Verbot impliziter Untertypbeziehungen aus:

```
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 Nichtunterstützung impliziter Untertypbeziehungen hat auch einen Nachteil, z.B. kann die Methode

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

nur mit Argumenten vom Typ `List<Polygon>` aufgerufen werden, nicht aber mit Argumenten vom Typ `List<Triangle>` oder `List<Square>`. Dies ist bedauerlich, da `drawAll()` nur Elemente aus der Liste liest und nie in die Liste schreibt, Sicherheitsprobleme durch implizite Untertypbeziehungen wie bei Arrays treten aber nur beim Schreiben auf. Lösung: *gebundene Wildcards* als Typen, die Typparameter ersetzen:

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

Der Compiler liefert eine Fehlermeldung, wenn die Möglichkeit besteht, dass in den Parameter p geschrieben wird. Genauer gesagt erlaubt der Compiler die Verwendung von p nur an Stellen, für deren Typen in Untertypbeziehungen **Kovarianz** gefordert ist (Lesezugriffe).

Bei Parametern, deren Inhalte nur geschrieben werden, z.B.:

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

Hier wird in `to` nur geschrieben aber nicht gelesen. Als Argument für `to` können daher `List<Square>`, aber auch `List<Polygon>` oder `List<Object>` übergeben werden (keyword: `super`). Der Compiler erlaubt die Verwendung von `to` nur an Stellen, für deren Typen in Untertypbeziehungen **Kontravarianz** gefordert ist (Schreibzugriffe).

4.2 Verwendung von Generizität

Wenn Wartbarkeit verbessert wird; bei gleich strukturierten Klassen und Methoden.

Faustregel: Containerklassen sollen generisch sein.

Faustregel: Klassen und Methoden in Bibliotheken sind generisch.

Faustregel: Generizität (oft gebundene Generizität) ist immer dort sinnvoll, wo mehrere Variablen vom gleichen (aber nicht von Anfang an fix festgelegten) Typ notwendig sind.

Faustregel: Wir sollen Typparameter als Typen formaler Parameter verwenden, wenn Änderungen der Parametertypen absehbar sind.

Faustregel: Generizität und Untertyprelationen ergänzen sich. Wir sollen stets überlegen, ob wir eine Aufgabe besser durch Ersetzbarkeit, durch Generizität, oder (häufig sinnvoll) eine Kombination aus beiden Konzepten lösen.

Faustregel: Wir sollen Überlegungen zur Laufzeiteffizienz beiseite lassen, wenn es um die Entscheidung zwischen Generizität und Untertypbeziehungen geht.

4.3 Typabfragen und Typumwandlungen

Verwendung dynamischer Typinformation

`getClass()`: liefert interne Repräsentation der Klasse des Objekts (vom Typ `Class`). Objekte vom Typ `Class` lassen sich einfach mit == vergleichen. Objekte vom Typ `Class` lassen sich auch direkt durch Anhängen von `.class` an einen Typ ansprechen: z.B.: `int.class`, `int[].class`, `Person.class`, `Comparable.class`, ...

Überprüfung auf Untertypbeziehung mit `instanceof`:

```
int calculateTicketPrice(Person p) {
    if (p.age() < 15 || p instanceof Student)
        return standardPrice / 2;
    return standardPrice;
}
```

Typumwandlungen und Generizität

Homogene Übersetzung einer generischen Klasse oder Methode in eine Klasse oder Methode ohne Generizität folgendermaßen:

- Spitze Klammern werden samt ihren Inhalten weggelassen.
- Jedes verbliebene Vorkommen eines Typparameters wird durch `Object` oder, falls vorhanden, die erste Schranke ersetzt.
- Ergebnisse und Argumente werden in die nötigen deklarierten Typen umgewandelt, wenn die entsprechenden Ergebnistypen und formalen Parametertypen Typparameter sind, die durch Typen ersetzt wurden.

Viele ältere, nicht-generische Java-Bibliotheken verwenden Klassen, die so aussehen, als ob sie aus generischen Klassen erzeugt worden wären. Vor der Verwendung von aus solchen Datenstrukturen gelesenen Objekten steht meist eine **Typumwandlung**. Die durchgehende Verwendung von Generizität würde den Bedarf an Typumwandlungen vermeiden oder zumindest erheblich reduzieren.

Faustregel: Wir sollen nur sichere Formen der Typumwandlung (die keine Ausnahmen auslöst) einsetzen.

Sichere Typumwandlungen Typumwandlungen sind sicher wenn

- in einem Obertyp des deklarierten Objekttyps umgewandelt wird,
- oder davor eine dynamische Typabfrage erfolgt, die sicherstellt, dass das Objekt einen entsprechenden dynamischen Typ hat,
- oder das Programmstück so geschrieben ist, als ob Generizität verwendet würde und alle Konsistenzprüfungen, die normalerweise der Compiler macht, vor der homogenen Übersetzung der Generizität händisch von uns durchgeführt werden.

Kovariante Probleme

TODO

Binäre Methoden

TODO

4.4 Überladene Methoden und Multimethoden

Dynamisches Binden erfolgt in Java über den dynamischen Typ eines speziellen Parameters. Z.B. wird bei `x.equal(y)` die auszuführende Methode durch den dynamischen Typ von `x` festgelegt. Der dynamische Typ von `y` ist bei der Methodenauswahl irrelevant. Aber der deklarierte Typ von `y` ist bei *überladenen* Methoden von Bedeutung. In anderen Programmiersprachen könnte auch der dynamische Typ von `y` von Bedeutung sein. Dann spricht man nicht von Überladen, sondern von *Multimethoden* (mehrfaches dynamisches Binden bei Methodenaufruf).

Überladen wird oft mit Multimethoden verwechselt $\Rightarrow$ schwere Fehler.

Deklarierte vs. dynamische Argumenttypen

```
1 Cow cow = new Cow();
2 Food grass = new Grass();
```

```
3 cow.eat(grass);          // Cow.eat(Food x)
4 cow.eat((Grass) grass); // Cow.eat(Grass x)
```

Hier wird (in Java) wegen dynamischen Binden auf jeden Fall `eat` in der Klasse Cow ausgeführt. Dadurch dass der deklarierte Typ des Methodenarguments zur Methodenauswahl herangezogen wird, wird in Zeile 3 und 4 nicht die selbe Methode ausgeführt obwohl in beiden Fällen der dynamische Typ `Grass` ist.

Hätten wir statt der ersten Zeile

```
1 Animal cow = new Cow();
```

würde wegen des dynamischen Bindens weiterhin `eat` in Cow ausgeführt. Aber zur Auswahl der überladenen Methode kann der Compiler nur den deklarierten Typen von `cow` verwenden (also `Animal`).

Faustregel: Wir sollen Überladen nur so verwenden, dass es keine Rolle spielt, ob bei der Methodenauswahl deklarierte oder dynamische Typen der Argumente verwendet werden.

Unter folgenden Bedingungen ist die Unterscheidung zw. deklarierten und dynamischen Typen bei der Methodenauswahl nicht wichtig, das Überladen von Methoden also *sicher*: Für je zwei überladene Methoden gleicher Parameterzahl

- gibt es zumindest eine Parameterposition, an der sich die Parametertypen unterscheiden, wobei diese Typen nicht in Untertyprelation zueinander stehen und auch keinen gemeinsamen Untertyp haben.
- oder alle Parametertypen der einen Methode sind Obertypen der Parametertypen der anderen Methode, und bei Aufruf der einen Methode wird nichts anderes gemacht, als auf die andere Methode zu verzweigen, falls die entsprechenden dynamischen Typen der Argumente dies erlauben.

Würden wir **immer dynamische** Typen hernehmen, hätten wir diese Probleme nicht. Statt überladenen Methoden hätten wir dann **Multimethoden**. Würde Java Multimethoden unterstützen, könnten wir die Cow Klasse folgendermaßen schreiben:

```
class Cow extends Animal {
    public void eat(Grass x) { ... }
    public void eat(Food x) {
        fallIll();
    }
} // Achtung: In Java ist diese Lösung falsch !!
```

Multimethoden wären in dem Fall praktisch, die Methodenauswahl ist aber oft nicht so offensichtlich bzw. eindeutig. Eine Regel besagt, dass immer jene Methode mit den speziellsten Parametertypen, die mit den dynamischen Typen der Argumente kompatibel sind, auszuführen ist. Diese Regel ist aber nicht hinreichend wenn Multimethoden mehrere Parameter haben, z.B.:

```
public void eatTwice(Food x, Grass y) { ... }
public void eatTwice(Grass x, Food y) { ... }
```

Wird `eatTwice` mit zwei Argumenten mit dynamischen Typ `Grass` aufgerufen, sind beide Methoden kompatibel, aber keine ist spezieller als die andere (mögliche Lösung: Auswahl der ersten passenden, oder von links nach rechts die speziellere).

Simulation von Multimethoden

Multimethoden nutzen mehrfaches dynamisches Binden. Java kennt nur einfaches dynamisches Binden. Simulation von Multimethoden durch wiederholtes einfaches Binden:

```
public abstract class Animal {
    public abstract void eat(Food food);
}
public class Cow extends Animal {
    public void eat(Food food) { food.eatenByCow(this); }
}
public class Tiger extends Animal {
    public void eat(Food food) { food.eatenByTiger(this); }
}
public abstract class Food {
    abstract void eatenByCow(Cow cow);
    abstract void eatenByTiger(Tiger tiger);
}
public class Grass extends Food {
    void eatenByCow(Cow cow) { ... }
    void eatenByTiger(Tiger tiger) { tiger.showTeeth(); }
}
public class Meat extends Food {
    void eatenByCow(Cow cow) { cow.fallIll(); }
    void eatenByTiger(Tiger tiger) { ... }
}
```

Beim Aufruf von `animal.eat(food)` wird zweimal dynamisch gebunden. Das erste dynamische Binden unterscheidet zwischen Objekten von Cow und `Tiger` und spiegelt sich im Aufruf von `eaten-ByCow` und `eatenByTiger` wider. Ein zweites dynamisches Binden unterscheidet zwischen Objekten von `Grass` und `Meat`. In den Unterklassen von `Food` sind insgesamt vier Methoden implementiert, die alle Kombinationen von Tierarten mit Futterarten abdecken.

4.5 Annotationen und Reflexion

Annotationen: Programmteile werden mit Markierungen versehen ⇒ Laufzeitsystem und Entwicklungswerkzeuge prüfen das Vorhandensein bestimmter Markierungen und reagieren entsprechend darauf.

So einfach dieses System zu sein scheint, so komplex sind Details der Umsetzung. Einerseits sollte das Hinzufügen von Annotationen zu Lava die Syntax nicht allzu sehr ändern und trotzdem aus der Syntax klar hervorgehen, dass es sich um möglicherweise ignorierte Programmteile handelt. Andererseits muss es möglich sein, zur Laufzeit das Vorhandensein von Annotationen abzufragen. Dafür wird **Reflexion** eingesetzt.

Annotationen und Reflexion in Java

Beispiel einer Annotation vom System: `@Override` vor einer Methodendefinition. Der Compiler prüft, ob die Methodendefinition mit dieser Annotation versehen ist und verlangt nur in diesem Fall, dass die Methode eine andere Methode überschreibt.

Es können auch eigene Annotationen implementiert werden, diese müssen deklariert werden (abgewandelte Interface-Definition Syntax):

```
@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
}
```

Solche Annotationen können wir zu Klassen, Interfaces und Enums hinzufügen um auf Korrekturen hinzuweisen:

```
@BugFix(who="Kaspar", date="1.10.2023", level=3,
        bug="class unnecessary and maybe harmful",
        fix="content of class body removed")
public class Buggy {}
```

Weiteres: mögliche Datentypen für Felder in Annotationen sind elementare Typen (wie `int`), Enum-Typen, `String`, `Class` und andere Annotationen sowie eindimensionale Arrays dieser Typen. In der Definition von Annotationen müssen die Argumente nicht die Form `name=wert` haben, wenn es nur ein Feld mit dem Namen `value` gibt (dann kann einfach nur der Wert übergeben werden) und die

runden Klammern können komplett weggelassen werden, wenn die Annotation keine Argumente hat (wie bei `@Override`).

Annotationen auf der Definition von Annotationen:

- `@Target`: was annotiert werden kann; Array von Werten des Enums `ElementType` (METHOD, TYPE, PARAMETER, CONSTRUCTOR, …). Ohne Angabe von `@Target` ist die definierte Annotation überall anheftbar.
- `@Retention`: legt fest, wie weit die definierte Annotation sichtbar bleiben soll. Mit dem Wert SOURCE der Enum `RetentionPolicy` wird die Annotation vom Compiler genau so verworfen wie Kommentare. Solche Annotationen sind nur für Werkzeuge, die auf dem Source-Code operieren, von Interesse. Andere Werte: CLASS … Annotation bleibt in der übersetzen Klasse vorhanden, aber zur Laufzeit nicht; RUNTIME … Annotation bleibt auch zur Laufzeit verfügbar.
- `@Documented`: parameterlos; sorgt dafür, dass die Annotation auch in der generierten Dokumentation vorhanden ist.
- `@Inherited`: parameterlos; sorgt dafür, dass das annotierte Element auch in einem Untertyp als annotiert gilt.

Default-Belegung für Parameter einer Annotation: z.B. Erweiterung von `BugFix` um

```
    String comment() default "";
```

Verwendung zur Laufzeit Wurde `@Retention(RUNTIME)` verwendet, generiert der Compiler für die Annotation ein entsprechendes Interface. Z.B. für obiges Beispiel:

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

Folgendermaßen kann auf die Annotation zugegriffen werden:

```
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";
}
```

Nur sinnvoll, wenn wir genau wissen, auf welche Annotation wir zugreifen möchten. Durch `getAnnotations` können wir alle Annotationen der Klasse gleichzeitig auslesen:

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

Wieder nur sinnvoll, wenn wir die Annotation im Vorhinein kennen.

Die Technik, mit der wir zur Laufzeit auf Annotationen zugreifen, nennt sich *Reflexion* bzw. *Reflection* oder *Introspektion*.

Anwendungen von Annotationen und Reflexion

Übliche Annotationen Obwohl Annotation heute nicht mehr wegzudenken sind, werden sie meist nur in ihrer einfachsten Form verwendet: nämlich um zusätzliche syntaktische Elemente in Programmen zu erlauben, ohne dafür die Programmiersprache ändern zu müssen.

Annotationen ermöglichen syntaktische Erweiterungen auch für ganz spezielle Einsatzgebiete, ohne gleichzeitig andere Einsatzgebiete mit unnötiger Syntax zu überladen.

`@Override` treffen wir häufig. Statt dieser Annotation wäre auch ein Modifier sinnvoll gewesen, aber aufgrund der geschichtlichen Entwicklung hat sich eine Annotation angeboten.

Manchmal stolpert man auch über `@Deprecated`, welche dem Compiler ermöglicht, bei Verwendung von deprecated Methoden, eine Warnung anzuzeigen.

Eine gefährliche Rolle spielt `@SuppressWarnings`, da diese alle Warnings vom Compiler unterdrückt.

Seit Java 8 kann man Interfaces mit `@FunctionalInterface` kennzeichnen. Diese Interfaces dürfen nur genau eine abstrakte Methode enthalten und können als Typen von Lambda-Ausdrücken verwendet werden.

Reflexion Reflexion ist eine Variante der *Metaprogrammierung* (Programme die Programme erstellen).

Während durch Metaprogrammierung das gesamte Programm zur Laufzeit sicht- und änderbar ist, kann Reflexion die Programmstruktur nicht ändern. In speziellen Fällen kann durch Reflexion sehr viel erreichbar sein, dennoch ergeben sich auch sehr viele Gefahren. Z.B.:

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

```
    }
}
```

Hier kann viel passieren, mit dem man nicht rechnet — ganz zu schweigen von der Gefahr, dass wir nicht wissen, was die mit `invoke` aufgerufenen Methoden machen. Möglicherweise ist die Methode nicht `public`, verlangt weitere/andere Argumente oder existiert gar nicht ⇒ Exceptions.

Ein Spezialbereich — JavaBeans JavaBeans ist ein Werkzeug, mit dem grafische Benutzeroberflächen ganz einfach aus Komponenten aufgebaut werden. Der Großteil der Arbeit wird von Werkzeugen bzw. fertigen Klassen erledigt. JavaBeans-Komponenten sind gewöhnliche Klassen, die bestimmte Namenskonventionen einhalten.

„Properties“: Objektvariablen, deren Werte von außen zugreifbar sind. Existieren z.B. die Methoden

```
public void setProp(int x) { ... }
public int getProp() { ... }
```

nehmen die Werkzeuge automatisch an, dass `prop` eine Property des Typs `int` ist. Existiert nur eine der Methoden, ist die Property nur les- oder schreibbar. Lesbare Properties des Typs `boolean` können statt mit `get` auch mit `is` beginnen. Die Werkzeuge verwenden hier Reflexion, da alle nötigen Informationen in den Namen, Ergebnistypen und Parametertypen der Methoden stecken.

In seltenen Fällen benötigen JavaBeans Information, die nicht über Reflexion verfügbar ist. Dafür gibt es z.B. `@ConstructorProperties`-Annotation: da in einer übersetzten Klasse nicht gesagt werden kann, welcher Parameter eines Konstruktors welcher Property entspricht, zählen die Argumente dieser Annotation einfach die Properties entsprechend ihrer Reihenfolge auf.

4.6 Aspektorientierte Programmierung

Zwei Sichtweisen: konzeptuelle Sichtweise auf sehr hoher Ebene und Sicht der Implementierung, wobei auf niedriger Ebene in das Gefüge der Objekte eingegriffen wird.

Konzeptuelle Sichtweise

Ergebnisse von Berechnungen hängen von

- dem Programm,
- der Semantik der Sprache, und
- der Daten

ab. Um die Ergebnisse zu ändern, passen wir oft entweder das Programm oder die Daten an. Aber wir können auch die Semantik der Sprache an unsere Änderungswünsche anpassen ⇒ **Aspekte**.

Aspektorientierte Programmierung zielt darauf ab, Ergebnisse von Berechnungen auf eine gewisse Weise abzuändern, ohne den Programmtext oder die Daten zu ändern. Sie bewegt sich in einem Graubereich, was die genaue Zuordnung zu Programm, Daten oder Sprachsemantik betrifft.

Beispiel: wir wollen in ein Banksystem eine Benutzerauthentifizierung einführen, wollen aber nicht händisch an jeder kritischen Stelle eine Benutzerüberprüfung machen. Durch Aspekte kann die Benutzerüberprüfung z.B. vor jeder Methode (Performance könnte leiden), vor dem Aufruf von Methoden, die in anderen Paketen liegen oder nur vor Methoden, die auf eine Datenbank zugreifen, erfolgen.

Separation-of-Concerns: unterschiedliche Belange sollen durch unterschiedliche Klassen abgebildet sein. In der aspektorientieren Programmierung werden zwei grundsätzliche Arten von Belange (Concerns) unterschieden:

- *Kernfunktionalitäten* (*Core-Concerns*): lassen sich gut und eindeutig bestimmten Klassen zuordnen und führen zu hohem Klassenzusammenhalt bei gleichzeitig schwacher Objektkupplung.
- *Querschnittsfunktionalitäten* (*Cross-Cutting-Concerns*): betreffen hingegen immer mehrere Klassen (hoher Klassenzusammenhalt und schwache Objektkupplung unmöglich).

Im Bank Beispiel: Verwaltung von Konten, Geldtransfers usw. wären Kernfunktionalitäten. Auch die zentrale Stelle für die Überprüfung von Zugriffsrechten ist eine Kernfunktionalität. Aber es muss an sehr vielen Stellen eingegriffen werden um zu veranlassen, dass die zentrale Stelle die Überprüfung der Zugriffsrechte durchführt. Das ist eine Querschnittsfunktionalität.

Implementierung — AspectJ

AspectJ (www.eclipse.org/aspectj) ist ein Werkzeug für die aspektorientierte Programmierung in Java.

Zusätzlich zu unseren normalen Java Klassen schreiben wir in `.aj`-Files unsere gewünschten Änderungen der Semantik. Statt `javac` verwenden wir `ajc` zum Kompilieren. Alle Java Klassen werden zusammen mit den `.aj`-Files und der Bibliothek `aspectjrt.jar` gleichzeitig übersetzt.

AspectJ baut auf folgenden Begriffen auf:

- **Join-Point:** zur Laufzeit identifizierbare Stelle in einem Programm, z.B. der Aufruf einer Methode oder der Zugriff auf ein Objekt.
- **Pointcut:** syntaktisches Element in einer `.aj`-Datei, das einen Join-Point (bzw. mehrere gleichartige Join-Points) auswählt und kontextabhängige Information dazu sammelt, z.B. die Argumente eines Methodenaufrufs oder eine Referenz auf das Zielobjekt.
- **Advice:** syntaktisches Element in einer `.aj`-Datei, das den Programmtext definiert, der an einem Join-Point ausgeführt werden soll. Dabei kann man den Programmtext vor (`before()`), nach (`after()`) oder anstatt (`around()`) dem Join-Point ausführen (wobei bei `around()` der Join-Point häufig innerhalb des Programmtexts explizit ausgeführt wird).

- **Aspect:** zentrales syntaktisches Element in einer `.aj`-Datei, das alle Teile zu einer Einheit zusammenführt. Ein Aspekt enthält Deklarationen von Variablen und Definitionen von Methoden (wie eine Java-Klasse) sowie Pointcuts und Advices.

Syntax eines Pointcuts: [*Sichtbarkeit*] `pointcut` *Name* `(` [*Argumente*] `) :` *Pointcuttyp* `(` *Signatur* `) ;`

Beispiel: Pointcut für einen Methodenaufruf, der alle Methoden umfasst, die mit beliebiger Sichtbarkeit im Paket `javax` oder Unterpaketen davon vorkommen, deren Namen mit `add` beginnen, mit `Listener` enden und deren einzige Argumente Untertypen von `EventListener` sind:

```
public pointcut AddListener() :
    call(* javax..*.add*Listener(EventListener+));
```

(`*` für beliebige Anzahl von Zeichen außer `.`; `..` für beliebige Anzahl jedes Zeichens; `+` für jeden Untertyp eines Typen)

Weitere Beispiele für Pointcuttypen:

- `execution(`*MethodSignature*`)`: Ausführung einer Methode
- `call(`*MethodSignature*`)`: Aufruf einer Methode
- `execution(`*ConstructorSignature*`)`: Ausführung eines Konstruktors
- `call(`*ConstructorSignature*`)`: Aufruf eines Konstruktors
- `initialization(`*ConstructorSignature*`)`: Initialisierung eines Objekts
- `staticinitialization(`*TypeSignature*`)`: Initialisierung einer Klasse
- u.v.m.

Syntax eines before-Advice: [*Sichtbarkeit*] `before(` [*Argumente*] `) :` {*Programmtext*}

Beispiel: ein Advice mit einem anonymen Pointcut und einer mit einem Namen versehenden Pointcut:

```
before() : call(* Account.*(..)) { checkUser(); }

pointcut connectionOperation(Connection connection) :
    call(* Connection.*(..) throws SQLException)
    && target(connection);

before(Connection connection) :
    connectionOperation(connection) {
        System.out.println("Operation auf " + connection);
    }
```

5. Applikative Programmierung und Parallelausführung

Applikative Programmierung = fortgeschrittene Form der funktionalen Programmierung.

Wunsch nach applikativen Programmiertechniken in objektorientierten Programmiersprachen, da sich funktionale und applikative Programmierung als recht erfolgsversprechende Basis für nebenläufige und parallele Programmierung erwiesen hat.

5.1. Lambdas und Java-8-Streams

Anonyme innere Klassen und Lambdas

Lambdas ähneln Objekten innerer Klassen.

Anonyme innere Klassen sind innere Klassen ohne vorgegebenen Namen. Beispiel:

```
public class List<A> implements Collection<A> {
    private Node<A> head = ...;
    ...
    public Iterator<A> iterator() {
        return new Iterator<A>() {
            private Node<A> p = head;
            public boolean hasNext() { return p != null; }
            public boolean next() { ... }
        }
    }
    ...
}
```

Anonyme innere Klassen kriegen vom Compiler einen internen Namen (z.B. `List$1`). Anonyme innere Klassen erweitern die Fähigkeiten von Java gegenüber normalen inneren Klassen in keiner Weise. Sie stellen nur eine Syntaxvereinfachung für den häufig vorkommenden Fall dar, dass Objekte einer inneren Klasse nur an genau einer Stelle im Programm erzeugt werden.

Lambdas stellen wiederum eine weitere Syntaxvereinfachung von anonymen inneren Klassen dar, wobei jede solche Klasse nur genau eine Methode definiert, sonst nichts. Dabei können der Methodenname, der Ergebnistyp und die Typen der Parameter weggelassen werden. Besteht der Methodenrumpf nur aus einer Anweisung, können auch die die geschwungenen Klammern und das `return` weggelassen werden. Das Weglassen der runden Klammern um einen einzigen Methodenparameter ist nur noch eine Kleinigkeit.

Eine Einschränkung bei Lambdas (im Gegensatz zu abstrakten inneren Klassen) besteht darin, dass im Methodenrumpf nur unveränderliche Variablen aus der Umgebung zugreifbar sind, das sind solche, die als `final` deklariert sind oder so verwendet werden, als ob sie als `final` deklariert wären.

Für ein Beispiel können wir Interfaces verwenden, die in den Java-Standard-Bibliotheken für diesen Einsatzzweck vordefiniert sind. Viele davon sind im Paket `java.util.function` zusammengefasst. Z.B. `Function<T,R>` mit `R apply(T t)`, `BiFunction<T,U,R>` mit `R apply(T t, U u)`, `Consumer<T>` mit `void accept(T t)`, ...

Jedes Interface in diesem Paket ist mit der Annotation `@FunctionalInterface` versehen, die den Compiler anweist, eine Fehlermeldung auszugeben, falls es sich nicht um ein Interface mit genau einer abstrakten Methode handelt. Sie können aber Methoden mit Default-Implementierungen und statische Methoden enthalten.

Beispiele:

```
Consumer<String> p = s -> System.out.println(s);
p.accept("Hello world.");
Function<Integer,String> value = i -> "value = " + i;
p.accept(value.apply(8));
BiFunction<String,Boolean,String> opt = (s,b) -> b ? s : "";
p.accept(opt.apply("maybe", true));
```

Obwohl Lambdas Objekten innerer Klassen ähneln, sind sie eigenständige Konstrukte, für deren Einführung die JVM erstmals in der Geschichte erweitert wurde. Grund: unzureichende Effizienz im Umgang mit einer großen Zahl sehr kleiner Klassen. Ein Großteil der Komplexität im Umgang mit Klassen ist für Lambdas unnötig. Die Semantik von Lambdas orientiert sich stark an der von anonymen inneren Klassen, sodass es kein Fehler ist, Lambdas als Spezialfall anonymer innerer Klassen anzusehen, wobei die Einschränkungen die gröbsten Probleme geschachtelter Klassen beseitigen.

Aus Abschnitt 1.1.2 wissen wir: untypisierte λ-Kalkül erreicht die Mächtigkeit einer Turing-Maschine. Viel mehr als λ-Abstraktion ist nicht nötig. Frage: *Sind Lambdas in Java auch so mächtig?* Antwort vielschichtig und komplex: Alle Parameter und Ergebnisse von Lambdas haben einen deklarierten Typ. Eine einfache typisierte Variante the λ-Kalküls, die große Ähnlichkeit zu Java-Lambdas hat, erreicht nicht mehr die Mächtigkeit der Turing-Maschine, macht Programme dafür aber einfacher verständlich. Grund: wir können keine unendlich großen Typen aufbauen, die wir bräuchten, um mit den Mitteln des λ-Kalküls Rekursion darzustellen. Einer typisierten Variante des λ-Kalküls können wir wieder die Mächtigkeit der Turing-Maschine verleihen, indem wir eine weitere Regel hinzufügen, die rekursive Aufrufe ermöglicht[1] (auf Kosten der Einfachheit).

Weiter syntaktische Variante zur Spezifikation von Lambdas: `Klassenname::Methodenname` steht für eine Methode in einer Klasse (oder für die Erzeugung eines Objekts der Klasse, wenn statt dem Methodennamen new verwendet wird).

```
BiFunction<String,String,Integer> cmp = String::compareTo;
    // entspricht cmp = (s,t) -> s.compareTo(t);
```

[1] Genau genommen handelt es sich um eine mit einem Typ parametrisierte, also generische Regel, was äquivalent zu einer Familie von Regeln ist, häufig *Y-Kombinator* genannt.

```
BiFunction<Object,Object,Boolean> eq = Objects::equals;
    // entspricht eq = (a,b) -> Objects.equals(a,b);
Function<StringBuilder,String> mk = String::new;
    // entspricht mk = sb -> new String(sb);
```

Java-8-Streams

Java-8-Streams sind Objekte der Klassen `Stream<T>`, `IntStream`, `LongStream` und `DoubleStream`, die jeweils als sequentielle oder parallele Datenströme verwendbar sind. Im Mittelpunkt stehen Methoden, die auf den Datenströmen operieren. Man unterscheidet 3 Arten:

- **Stream-erzeugende Operationen:** z.B. unterstützen iterierbare Klassen die Methoden `stream()` und `parallelStream()`, die jeweils einen (sequentiellen oder parallelen) neuen Datenstrom mit den iterierbaren Elementen erzeugen. Die Stream-Klassen selbst bieten statische Methoden zum Erzeugen neuer Streams an.
- **Stream-modifizierbare Operationen:** Objekt-Methoden der Stream-Klassen, die Operationen auf den Elementen eines Streams ausführen und Ergebnisse wieder in einen Stream füttern. Ergebnisse dieser Methoden sind von einem Stream-Typ. Vieler dieser Methoden dienen als Funktionen höhere Ordnung, denen Lambdas als Parameter übergeben werden. Beispiele: `map`, `filter`, `limit`, `sorted`, `distinct`, ...
- **Stream-abschließende Operationen:** führen ebenfalls Operationen auf den Elementen eines Streams aus, Ergebnisse werden jedoch nicht mehr in einen Stream gefüttert, sondern der Stream wird abgeschlossen. Beispiele: `reduce` (Elemente eines Streams werden durch Lambdas zu einem einzigen Wert zusammengefasst), `collect` (Elemente eines Streams werden in irgendeine Art von Collection abgelegt), `forEach`. Spezielle abschließende Operationen sind z.B. `allMatch` und `anyMatch`, die ein Boolean zurückgeben oder `count`, das einfach die Elemente zählt.

Die Ausführung der Stream-Operationen erfolgt mittels Lazy-Evaluation: Hinter jeder Operation, die einen Stream erzeugt oder modifiziert, steht ein Iterator. Bei erzeugenden und modifizierenden Methoden passiert noch keine inhaltliche Berechnung, sondern es werden nur die dahinter stehenden Iteratoren erzeugt und miteinander verknüpft. Erst die Ausführung einer Stream-abschließenden Operation stößt die eigentlichen Berechnungen an.

Die Iteratoren, die hinter den Stream-Operationen stecken, sind vom Typ `Spliterator<T>`. Wir können die Fähigkeiten von Streams selbst erweitern, indem wir neue Spliteratoren schreiben (das Interface `Spliteratoren` implementieren). Über die Klasse `StreamSupport` werden Spliteratoren in Streams eingebunden. Wir erhalten einen modifizierenden Operator, wenn unser Spliterator über Elemente iteriert, die zuvor aus einem anderen Spliterator gelesen wurden; andernfalls erhalten wir einen erzeugenden Operator. Jede Methode, die Elemente aus einem Spliterator liest, kann als abschließende Operation verstanden werden. Spliteratoren existieren nur aus der Sicht der Implementierung.

Beim Programmieren mit Streams bleiben sie meist versteckt.

Stream-Beispiel: Faktorielle-Berechnung

```
import java.util.*
import java.util.stream.*
...
    public static long fact(int n) {
        return LongStream.rangeClosed(2, n).reduce(1, (i, j) -> i * j);
    }
...
```

Weiteres Beispiel: `sales` stellt eine Ansammlung von Verkäufen dar, die jeweils aus einer Menge von Produkten (als Strings) bestehen. Das Methodenergebnis bildet jedes Produkt auf eine `Map` ab, die angibt, welche anderen Produkte wie häufig zusammen mit diesem verkauft wurden.

```
...
    public static Map<String, Map<String, Long>>
                  toMap(Collection<Set<String>> sales) {
        return sales.stream()
            .flatMap(set -> set.stream()
                .flatMap(p -> set.stream()
                    .filter(q -> !p.equals(q))
                    .map(q -> new AbstractMap.SimpleEntry<>(p, q))
                )
            )
            .collect(Collectors.groupingBy(e -> e.getKey(),
                Collectors.groupingBy(e -> e.getValue(),
                    Collectors.counting())));
    }
...
```

Andere Lösung mit Lambdas statt Streams:

```
...
    public static Map<String, Map<String, Long>>
                  toMap2(Collection<Set<String>> sales) {
        Map<String, Map<String, Long>> res = new HashMap<>();
        sales.forEach(set ->
            set.forEach(p -> {
                Map<String, Long> map = res.computeIfAbsent(p, k -> new
                ↪  HashMap<>());
                set.forEach(q -> {
                    if (!p.equals(q))
                        map.compute(q, (k,v) -> v==null ? 1 : v+1);
                });
```

```
            })
        );
        return res;
    }
...
```

Streams und Lambdas **erhöhen nicht die Mächtigkeit der Sprache**, sondern geht es darum, eine zusätzliche Abstraktionsebene einzuziehen.

Applikative Programmierung in der Praxis

Zusammenfassung einiger Erfahrungen bezüglich Java-8-Streams und Lambdas:

- Abarbeitung folgt fixem Schema: Ansammlung von Daten ⇒ Umformung der Einträge in beliebig vielen Schritten (allgemein als *Map*) ⇒ umgeformte Einträge sammeln und ins gewünschte Format bringen (allgemein als *Reduce*). Schema nennt sich *Map-Reduce*.
- Viele Programmieraufgaben sind nach diesem Schema lösbar.
- Für die meisten so lösbaren Aufgaben reicht es, die vorgefertigten Funktionen höherer Ordnung auf Lambdas anzuwenden — Kombinieren bestehender Funktionen statt Entwicklung eigener Funktionen ⇒ sehr effiziente Form der Programmierung.
- Generizität spielt eine große Rolle. Die meisten Typen werden durch Typinferenz ermittelt.
- Wenn Ausdrücke so weit ausgereift, dass alle Typen in sich konsistent sind ⇒ häufig auch inhaltlich fehlerfrei. Fehlende Typkonsistenz ⇒ kann Hinweise auf Verbesserungen liefern, aber auch zu Fehlinterpretationen führen.
- Functional Interfaces enthalten keine über Signaturen hinausgehende Informationen über das erwartete Verhalten, also keine Zusicherungen. Alle der Signatur entsprechenden Lambdas sind akzeptabel. Nur auf diese Weise ist Typkonsistenz ein guter Indikator für Korrektheit. Aufgrund umfangreicher Typinferenz wäre es praktisch unmöglich, die Konsistenz von Zusicherungen händisch zu prüfen (strukturelle Abstraktion $\neq$ nominale Abstraktion ⇒ funktionale Abstraktion $\neq$ objektorientierte Abstraktion).
- Map-Reduce-Schema nicht die einzig mögliche Form der applikativen Programmierung. Es kann zu jedem beliebigen Programmierschema eine Menge von Klassen mit Funktionen höherer Ordnung entwickelt werden, die dieses Schema auf abstrakte Weise unterstützen. So eine Entwicklung kann sehr aufwendig sein, kann aber auch die Programmierung in so einem Schemata stark vereinfachen. Es entsteht quasi eine eigene Sprache innerhalb der Programmiersprache.

„*Wer (nur) einen Hammer hat, sieht in jedem problem einen Nagel.*“: Wer mit Java-8-Streams gut umgehen kann, betrachtet jedes Problem als Map-Reduce-Problem und finden oft sehr kreative Lösungen. Manko: Leute, die das Programm lesen, können die kreativen Ideen dahinter nur schwer erkennen und das Programm kaum verstehen.

Faustregel: In nichttrivialen applikativen Programmteilen sollen wir Idden hinter Vorgehensweisen durch Kommentare skizzieren. Zusicherungen auf dabei verwendeten kleinen Hilfsmethoden (Lambdas) sind dagegen zu vermeiden.

Faustregel: Im Umfeld applikativer Programmteile sind Variablen so zu verwenden, als ob sie `final` wären.

Faustregel: Meist ist es vorteilhaft, entweder ganz in einer funktionalen (nicht auf Zustandsänderungen ausgelegten) oder ganz in einer prozedural-objektorientierten Denkweise zu bleiben.

Von einer *applikativen* Denkweise sprechen wir, wenn es darum geht, ganze Programme nur aus vorgefertigten Funktionen zusammenzusetzen. Das kann gut gelingen, wenn wir auf destruktive Zuweisungen verzichten und Lambdas einsetzen. Nicht jede applikative Denkweise muss funktional sein, sie kann auch prozedural oder objektorientiert sein. Von einer *funktionalen* Denkweise sprechen wir, wenn keinerlei Zustandsänderungen mitbedacht werden müssen. Das impliziert den Verzicht auf destruktive Zuweisungen und den Einsatz von Lambdas. In einer *prozeduralen* Denkweise müssen Zustandsänderungen mitbedacht werden, unabhängig davon, ob auf destruktive Zuweisungen verzichtet wird oder Lambdas eingesetzt werden.

Faustregel: Funktionen (höherer Ordnung) sollen so allgemein wie möglich sein und Zustandsänderungen lokal halten.

5.2 Funktionen höherer Ordnung

Frage: wie können Lambdas als Funktionen höherer Ordnung auch ohne Streams eingesetzt werden?

Nachbildung typischer Kontrollstrukturen

Bedingte Anweisung zählen zu den wichtigsten Kontrollstrukturen.

In Java ist man auch ohne `boolean` und `if`-Anweisungen in der Lage, mit Booleschen Ausdrücken zu arbeiten. Die Basis für Fallunterscheidungen bildet dynamisches Binden:

```
interface Bool {
    <A> A ifThenElse(A t, A f);
    default Bool negate() {
        return ifThenElse(False.VALUE, True.VALUE);
    }
    default Bool and(Bool b) {
        return ifThenElse(b, False.VALUE);
    }
    default Bool or(Bool b) {
        return ifThenElse(True.VALUE, b);
    }
```

```
    default Bool isEqual(Bool b) {
        return ifThenElse(b, b.negate());
    }
}
final class True implements Bool {
    private True() {}
    public static final True VALUE = new True();
    public <A> A ifThenElse(A t, A f) { return t; }
}
final class False implements Bool {
    private False() {}
    public static final False VALUE = new False();
    public <A> A ifThenElse(A t, A f) { return f; }
}
```

Problem: in jedem Aufruf von `ifThenElse` werden die beiden Argumente sofort ausgewertet. Das ist nicht die übliche Semantik einer bedingten Anweisung. Wir erwarten uns, dass nur eines der beiden Argumente ausgewertet wird, für `True` das erste und für `False` das zweite. Mit Funktionen höherer Ordnung ist diese Problem lösbar, z.B. mit `import java.util.function.Supplier`:

```
    ...
    default <T> T getIfThenElse(Supplier<T> t, Supplier<T> f) {
        return ifThenElse(t, f).get();
    }
    default Bool andThen(Supplier<Bool> b) {
        return getIfThenElse(b, () -> False.VALUE);
    }
    default Bool orElse(Supplier<Bool> b) {
        return getIfThenElse(() -> True.VALUE, b);
    }
    ...
```

Beispielsweise gibt der Aufruf

```
True.VALUE.orElse(() -> False.VALUE)
          .getIfThenElse(() -> "True", () -> "False");
```

als Ergebnis `"True"` zurück, ohne `() -> False.VALUE` und `() -> "False"` auszuwerten.

Hier eine Variante von `Bool` mit Lazy-Evaluation:

```
import java.util.function.*;
@FunctionalInterface
interface LazyBool extends Supplier<Bool> {
    static final LazyBool TRUE = () -> True.VALUE;
    static final LazyBool FALSE = () -> False.VALUE;
```

```
    default <T> Supplier<T> ifThenElse(Supplier<T> t,
                                       Supplier<T> f) {
        return () -> get().ifThenElse(t, f).get();
    }
    default LazyBool negate() {
        return () -> get().ifThenElse(False.VALUE, True.VALUE);
    }
    default LazyBool and(LazyBool b) {
        return () -> get().ifThenElse(b, FALSE).get();
    }
    default LazyBool or(LazyBool b) {
        return () -> get().ifThenElse(TRUE, b).get();
    }
    default LazyBool isEqual(LazyBool b) {
        return () -> get().ifThenElse(b, b.negate()).get();
    }
}
```

Faustregel: Es gibt zwei sinnvolle Ausführungszeitpunkte für Funktionen: so früh wie möglich (Eager-Evaluation) oder so spät wie möglich (Lazy-Evaluation). Andere Zeitpunkte sind eher zu vermeiden.

Nachbildung der Hintereinanderausführung durch Zusammensetzen von zwei Lambdas:

```
public static <T,V,R> Function<T,R>
              compose(Function<V,R> f, Function<T,V> g) {
    return t -> f.apply(g.apply(t));
}
```

Ergebnis ist ein Lambda, das zuerst `g` auf das Argument `t` des Lambdas anwendet, danach `f` auf das Ergebnis davon. Beispielsweise führt

```
compose(String::length, String::trim).apply(" a ");
```

`" a ".trim().length()` aus und gibt `1` zurück.

Funktionale Elemente in Java

Praktisch werden wir keine bestehende Kontrollstrukturen nachbilden, sondern neue Funktionalität hinzufügen. Wir müssen uns nicht auf funktionale Programmierung beschränken, nur die Lambdas selbst sollten sich an der funktionalen Programmierung orientieren.

Beispiel: Anwendung eines Lambdas auf ein Array

```
public static <T> void arrayMap(T[] xs, Function<T,T> f) {
    for (int i = 0; i < xs.length; i++) {
        xs[i] = f.apply(xs[i]);
```

```
    }
}
```

Es gibt schon eine vordefinierte Methode die unsere Arbeit erleichtert:

```
public static <T> void arrayMap2(T[] xs, Function<T,T> f) {
    Arrays.setAll(xs, i -> f.apply(xs[i]));
}
```

Faustregel: Vor der Implementierung einer eigenen Funktion höherer Ordnung sollten wir uns vergewissern, dass nicht eine ähnliche Methode schon standardmäßig vordefiniert ist. Die vordefinierte Methode ist zu bevorzugen.

Wer die wichtigsten vordefinierten Funktionen höherer Ordnung kennt und in der Lage ist, Ähnlichkeiten richtig zu erkennen, kann sehr effizient programmieren und dabei Programme von hoher Qualität schreiben. Sowohl das Kennen der Methoden als auch das Erkennen von Ähnlichkeiten hängt von der Erfahrung ab.

Optional Ein Object von `Optional<T>` enthält einfach nur ein Objekt vom Typ T oder ist leer. `isPresent()` liefert genau dann `true` zurück, wenn das enthaltene Objekt nicht `null` ist. Interessanter ist z.B. `T orElse(T other)`. Diese Methode liefert als Ergebnis das enthaltene Objekt, oder, falls das `Optional` Objekt leer ist, den Wert `other`. `orElseGet` nimmt statt `other` ein Lambda und gibt bei leerem `Optional` das Ergebnis einer Ausführung des Lambdas zurück. `orElseThrow` wirft bei leerem `Optional` eine Exception. `ifPresent` führt bei leerem `Optional` das übergebene Lambda aus.

Faustregel: Zusammen mit Lazy-Evaluation soll auf den expliziten Umgang mit `null` verzichtet und stattdessen `Optional` eingesetzt werden. Zusammen mit Eager-Evaluation ist `Optional` wenig sinnvoll und ein expliziter Umgang mit `null` vorteilhaft.

Currying Benannt nach Haskell Curry, ist eine Technik, mit der man, nur durch Funktionen mit einem Parameter, Funktionen mit beliebig vielen Parametern darstellen kann. Die Technik ist einfach: Statt einer Funktion mit zwei Parametern schreiben wir eine Funktion mit nur einem Parameter, die als Ergebnis eine Funktion zurückgibt, die den zweiten Parameter hat und das eigentliche Ergebnis berechnet. Wiederholt angewendet lässt sich die Zahl der Parameter damit beliebig erhöhen.

Beispiel: `f` und `g` machen das Gleiche, aber `f` hat 2 Parameter und `g` verwendet Currying

```
BiFunction<String,String,String> f = (s, t) -> s + t;
Function<String,Function<String,String>> g = s -> t -> s + t;
```

Aufrufe:

```
String s = f.apply("a", "b");
String t = g.apply("a").apply("b");
```

Merkmale von Currying:

- Auswertung von Lambdas mit Currying ist etwas aufwändiger, da für jeden Paramter ein eigener Aufruf nötig ist. Das erhöht den Ressourcenverbrauch etwas, lässt sich aber durch optimierte Compiler verbessern.
- Currying erhöht die Flexibilität bei der Auswertung. Es ist nicht notwendig, dass alle, für die Berechnung notwendigen Parameter, an der selben Stelle im Programm verfügbar sind. Man kann die Aufrufe auch schrittweise an verschiedenen Stellen im Programm machen indem man z.B. das Lambda zwischen den Schritten an verschiedene Programmteile weiterreicht.

Pattern-Matching Werte in Parametern bestimmen, welche Funktion auszuführen ist. Ähnlich zu Multimethoden, aber statt den Parametertypen werden die konkreten Werte in den Parametern betrachtet. Wenn es in Java Pattern-Matching gäbe, könnte ein Beispiel zur Berechnung der Länge eines Strings folgendermaßen aussehen:

```
int strLength("") { return 0; }
int strLength([char c, String s] c + s) { return 1 + strLength(s); }
```

5.3 Nebenläufige Programmierung in Java

Grundlegende Mechanismen für das Erzeugen von Threads und die Synchronisation in Java in Abschnitt 2.5.

Thread-Erzeugung und Synchronisation in Java

Beispiel:

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

`i` und `j` sollten stets die gleichen Werte enthalten. Wenn jetzt aber mehrere Threads im selben Counter Objekt die Methode `flip()` ausführen, kann es vorkommen, dass sich `i` und `j` voneinander unterscheiden. In einer `synchronized` Methode kann das nicht passieren.

```
public synchronized void flip() { i++; j++; }
```

Faustregel: In nebenläufigen Programm(teil)en sollen alle Methoden, die auf Objekt- oder Klassenvariablen zugreifen, `synchronized` sein.

Faustregel: `synchronized` Methoden sollen nur kurz laufen.

Man kann auch `synchronized` Blöcke verwenden:

```
public void flip() {
    synchronized(this) { i++; }
    synchronized(this) { j++; }
}
```

Hier können `i` und `j` zwar kurzfristig unterschiedlich sein, doch am Ende vom Programmablauf sind sie gleich.

Locking: Java setzt auf Objekte für einen bestimmten Thread einen „Lock“ um zu verhindern, dass andere Threads auf das Objekt zugreifen. Bei `synchronized` Blöcken, bestimmt das Argument das Objekt, dessen Lock gesetzt werden soll. Bei `synchronized` Methoden wird immer das Objekt, in dem die Methode aufgerufen wird, also `this`, gelockt.

Einzelne Schreib- und Lesezugriffe auf `volatile` Variablen sind atomar. Einige Klassen wir `AtomicInteger` bieten Methoden an, die Werte einzelner Variablen ohne `synchronized` atomar ändern.

wait und notify Beispiel:

```
public class PrinterDriver {
    private boolean online = false;
    public synchronized void print(String s) {
        while (!online) {
            try { wait(); }
            catch(InterruptedException ex) { return; }
        }
        ... // send s to printer
    }
    public synchronized void onOff() {
        online = !online;
        if (online) notifyAll();
    }
}
```

Nebenläufige Threads laufen meist in einer Methoden namens `run` in einer Endlosschleife. Beispiel:

```
public class Producer implements Runnable {
    private PrinterDriver t;
    public Producer(PrinterDriver t) { this.t = t; }
    public void run() {
        String s = ...
        for (;;) {
            ...          // produce new value in s
            t.print(s); // send s to the printer server
        }
    }
}
```

`Runnable` spezifiziert `run`. Erzeugung neuer Threads:

```
PrinterDriver t = new PrinterDriver(...);
for (int i = 0; i < 10; i++) {
    Producer p = new Producer(t);
    new Thread(p).start();
}
```

Aufruf von `start()` bewirkt Ausführung von `p.run()`.

Nebenläufigkeit in der Praxis

Die grundlegenden Sprachkonzepte für nebenläufige Programmierung werden nur selten verwendet, da es für die meisten Probleme gute vorgefertigte Lösungen gibt. Vorallem finden wir diese in `java.util.concurrent` und `java.util.concurrent.atomic`.

Aufgaben und Threads Konzept namens *Future*: Variable in der das Ergebnis einer Berechnung abgelegt wird, die Berechnung muss bei der Definition der Variable aber noch nicht fertig sein, sondern läuft im Hintergrund. Wollen wir vor Beendigung der Berechnung auf die Variable zugreifen, blockiert der Thread. Das funktioniert aber nur, wenn die Hintergrundberechnung unbeeinflusst von anderen Berechnungen abläuft. In Java gibt es dafür die Klasse `FutureTask` und das Interface `Future` im Paket `java.util.concurrent`.

Außerdem gibt es das Interface `Executor` in `java.util.concurrent`, mit dem man Aufgaben an Threads aufteilen kann. Es gibt mehrere standardmäßige Implementierunge von `Executor`, z.B. `ThreadPoolExecutor`.

Java-8-Streams Streams bieten `.parallelStream()` für Nebenläufigkeit an. Beispiel:

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

Im Hintergrund werden die Aufgaben über einen `ThreadPoolExecutor` abgearbeitet. Voraussetzung ist, dass die einzelnen Elemente (wie ganz allgemein bei Verwendung von Streams) unabhängig voneinander sind, also keine gemeinsamen Variablen haben.

Methoden wie `sorted()` oder `distinct()` erfordern spezielle Algorithmen für den Umgang mit Nebenläufigkeit, vor allem `distinct()` kann mit Nebenläufigkeit ineffizient werden. Auch abschließende Operationen müssen für Nebenläufigkeit ausgelegt sein. Lambdas in `reduce()` müssen assoziativ sein.

Thread-sichere Datenstrukturen Klassen in `java.util.concurrent`, z.B. `ConcurrentHashMap` ähnelt `HashMap`, erlaubt jedoch gleichzeitige Zugriffe mehrerer Threads und ist tatsächlich sehr effizient wenn viele Threads gleichzeitig darauf zugreifen, da diese Implementierung ohne Locks auskommt.

Weiters gibt es auch

```
Collections.synchronizedMap(new HashMap(...));
```

eine über einen einfachen Lock synchronisierte Variante von `HashMap`. Solange Threads nur selten gleichzeitig zugreifen wollen, ist diese Variante effizienter als `ConcurrentHashMap`.

Für die meisten Datenstrukturen gilt Ähnliches.

Vorgehensweise Wenn Teilaufgaben nicht voneinander abhängen ⇒ parallele Ströme oder `Executor`.

Wenn Teilaufgaben voneinander abhängen ⇒ für möglichst wenige gleichzeitige Zugriffe, vor allem Schreibzugriffe, auf gemeinsame Daten sorgen. Klassen wir `ConcurrentHashMap` können in dem Fall helfen.

Synchronisation und die objektorientierte Sicht

Umgang mit Synchronisationsproblemen In Java kümmern sich beispielsweise die Klassen `Vector` und `Hashtable` selbst um Synchronisation, die ähnlichen Klassen `LinkedList` und `HashMap` aber nicht.

Gefürchtet sind **Liveness-Probleme**, wie Deadlock, Livelock und Starvation.

Deadlock-Vermeidung: Verhinderung von Zyklen, beruhend auf einer linearen Anordnung aller Objekte im System. Locks dürfen nur in dieser Reihenfolge angefordert werden, d.h. wenn wir in einer `synchronized` Methode vom Objekt y sind, dürfen wir keine `synchronized` Methode in einem Objekt x aufrufen, wenn entsprechend der linearen Anordnung x vor y steht. In der Praxis sind lineare Anordnung sehr einschränkend, da sie alle Arten von zyklischen Strukturen verhindern.

Vorgefertigte Lösungen für die nebenläufige Programmierung beruhen großteils auf bekannten Techniken, die nicht oder kaum anfällig für Verletzungen der Liveness-Properties sind.

Objektorientierte Sicht Basiskonzept in Java, *Monitor-Konzept*, ist schon recht alt. Objektorientierte Programmiertechniken werden kaum unterstützt: Synchronisation wird weder als zu Objektschnittstellen gehörend betrachtet, noch in Untertypbeziehungen berücksichtigt.

5.4 Prozesse und Interprozesskommunikation

Prozesse werden vom Betriebssystem verwaltet.

Erzeugen von Prozessen in einer Shell

Ausführung von Kommandos in der Shell (z.B. `bash`). Z.B. führt `java Test arg1 arg2` den Java-Interpreter (`java`) mit den Argumenten `Test`, `arg1` und `arg2` aus. Dabei erzeugt die Shell für die Programmausführung einen neuen Prozess.

Jeder Prozess bekommt drei Ein- und Ausgabekanäle zugeordnet: `stdin`, `stdout` und `stderr`.

Umgang mit Dateien und I/O-Strömen in Java

Prinzipiell:

- `main` bekommt die Programmargumente als String Array.
- `stdin` über das Objekt vom Typ `InputStream` in der Variable `System.in` lesbar (Häufig über `new Scanner(System.in)`).
- `stdout` und `stderr` je über ein Objekt vom Typ `PrintStream` in den Variablen `System.out` und `System.err` schreibbar. Häufig einfach über `System.out.println(...)` oder auch `new ObjectOutputStream(System.out)`.
- Alle Arten von Dateien werden nach dem Öffnen über diverse Arten von Strömen (nicht verwechseln mit Java-8-Streams) gelesen und geschrieben und danach wieder geschlossen. Ströme unterscheiden sich durch Kodierung (Bytes vs. `char`), Pufferung (gepuffert vs. ungepuffert) und unterstützte Zugriffsmethoden.
- Ein- und Ausgabekanäle können nur Bytes übertragen. Innerhalb von Java sind Zeichen immer im UTF-16-Format kodiert. Wenn Zeichen übertragen werden sollen, empfiehlt es sich, Ströme der Typen `Readable` und `Writer` und deren Untertypen zu verwenden.
- Außer bei `Scanner` und `PrintStream` müssen bei allen Strömen `IOExceptions` abgefangen werden. Damit Ströme auch bei Exceptions brav geschlossen werden, bietet sich die Verwendung von `try`-With-Resources an:

```
try (FileReader fr = new FileReader(path);
     BufferedReader br = new BufferedReader(fr)) {
    ... // use br
    // fr and br automatically closed at end of try block
} catch (IOException ex) { ... }
```

Beim Lesen/Schreiben von Textdateien muss die Kodierung als String dem Strömen (z.B. `InputStreamReader` oder `FileReader`) übergeben werden, etwa `"UTF-8"` oder `"ISO-8859-1"`. Darauf kann verzichtet werden, wenn es sich um die Default-Kodierung des Betriebssystems handelt.

Bei anderen Daten muss für die Umwandlung zwischen der internen Darstellung und dem externen Format gesorgt werden. Umwandlung der internen Darstellung eines Objekts in das externe Format durch `toString` oft verlockend, aber meist nicht passend.

In der parallelen Programmierung sind Datenformate häufig einfach strukturiert, etwa Listen von Zahlen, die in jeweils 4 Bytes dargestellt werden. Wir können also Zahlen in vier Bytes in einem Byte-Strom auffassen. Je nach Maschine gibt es aber Unterschiede in der Reihenfolge der Bytes (**Big-Endian vs Little-Endian**).

Die Umwandlung von internen zum externen Format heißt **Serialisierung** (andersherum **Deserialisierung**). Interface `Serializable` für Typen die automatische Serialisierung und Deserialisierung unterstützen. Generell werden als `static` oder `transient` deklarierte Variablen bei der Serialisierung nicht berücksichtigt.

Standardisierte Datenformate für *semistrukturierte* Daten: XML, JSON, ...

Shell-Variablen in Java zugreifbar über `System.getEnv()`, liefert eine `Map<String, String>` mit allen Shell Variablen. Oder `System.getEnv(String name)` zum Abfragen einer bestimmten Shell-Variable.

`Runtime.getRuntime()` gibt das einzige Objekt von `Runtime` im aktuell ausgeführten Java-Interpreter zurück. Praktische Methoden: `availableProcessors()`, oder `exec(...)` zum Erzeugen neuer Prozesse:

```
Process p = Runtime.getRuntime().exec("java -cp ~/java Test");
```

6. Entwurfsmuster und Entscheidungshilfen

Design-Patterns dienen der Wiederverwendung kollektiver Erfahrung in der Softwareentwicklung.

Idee der Software-Entwurfsmuster gründet sich im Wesentlichen auf das Gang-of-Four-Buch: E. Gamma, R. Helm, R. Johnson and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software.* Addison-Wesley, Reading, Massachusetts, 1994.

6.1 Grundsätzliches und Muster für Verhalten

Aufbau von Entwurfsmustern

Hauptsächlich aus diesen vier Elementen:

- **Name:** einziger Begriff mit dem wir ein Problem, dessen Lösung und Konsequenzen ausdrücken können.
- **Problemstellung:** Beschreibung des Problems zusammen mit dessen Umfeld.
- **Lösung:** Beschreibung einer bestimmten Lösung der Problemstellung.
- **Konsequenzen:** Liste von Eigenschaften bzw. Vor- und Nachteilen der Lösung.

Faustregel: Entwurfsmuster sollen zur Abschätzung der Konsequenzen von Designentscheidungen eingesetzt werden, können aber nur in begrenztem Ausmaß und mit Vorsicht als Bausteine zur Erzielung bestimmter Eigenschaften dienen.

Visitor

Anwendbar, wenn

- viele unterschiedliche, nicht verwandte Operationen auf einer Objektstrutkur realisiert werden soll,
- sich die Klassen der Objektstruktur nicht ändern,
- häufig neue Operationen auf der Objektstruktur integriert werden müssen oder
- ein Algorithmus über die Klassen einer Objektstrutkur verteilt arbeitet, aber zentral verwaltet werden soll.

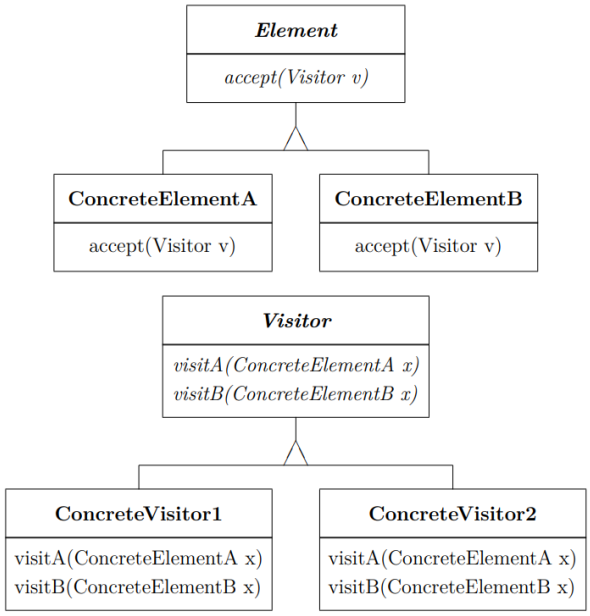

Abbildung 1: Visitor Struktur

Folgende Eigenschaften:

- Neue Operationen lassen sich leicht durch die Definition neuer Untertypen von *Visitor* hinzufügen.
- Verwandte Operationen werden im Visitor zentral verwaltet und von Visitor-fremden Operationen getrennt.
- Ein Visitor kann mit Objekten aus voneinander unabhängigen Klassenhierarchien arbeiten.
- Die gute Erweiterungsmöglichkeit der Klassen unterhalb von Visitor muss mit einer schlechten Erweiterbarkeit der Klassen der konkreten Elemente erkauft werden. Müssen neue konkrete Elemente hinzugefügt werden, so führt dies dazu, dass viele Methoden implementiert werden müssen.
- Häufig wird angeführt, dass die visit-Methoden nicht einfach auf konkrete Elemente zugreifen können; oft können dies Parameter der visit-Methoden ausgleichen, wodurch es auf Implementierungsdetails ankommt, inwieweit das relevant ist.

Iterator

Iterator, auch *Cursor*, ermöglicht sequentiellen Zugriff auf die Elemente eines *Aggregats* (Sammlung von Elementen), ohne die innere Darstellung des Aggregats offenzulegen.

Anwendbar, um

- auf die Inhalte eines Aggregats zugreifen zu können, ohne die innere Darstellung offen legen zu müssen,
- mehrere (gleichzeitig bzw. überlappende) Abarbeitungen der Elemente in einem Aggregat zu ermöglichen,
- eine einheitliche Schnittstelle für die Abarbeitung verschiedener Aggregatstrukturen zu haben, also um polymorphe Iterationen zu unterstützen.

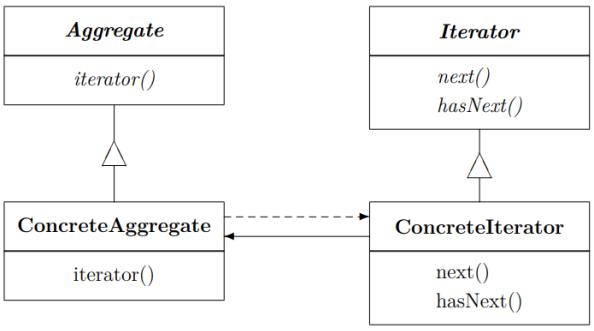

Abbildung 2: Iterator Struktur

Eigenschaften:

- Unterschiedliche Varianten in der Abarbeitung, z.B. für Baumstrukturen gibt es zahlreiche Möglichkeiten. Mehrere Iteratoren für unterschiedliche Reihenfolgen implementierbar.

- Vereinfachen die Schnittstelle von Aggregate, da Zugriffsmöglichkeiten durch Iteratoren bereitgestellt werden. Weitere Methoden möglich, z.B. in Java in `Iterator` die Methode `remove`.
- Mehrere Iterator können gleichzeitig auf einem Aggregat arbeiten, da jeder Iterator den aktuellen Abarbeitungszustand verwaltet.

Implementierungsvarianten:

- *Interne* vs. *externe* Iteratoren. Interne Iteratoren entscheiden selbst, wann die nächste Iteration erfolgt, bei externen bestimmt das die Anwendung. Bei internen liegt die Schleife (oder Rekursion) innerhalb der Iterator-Implementierung. Die Methoden `next()` und `hasNext()` sind nur bei externen Iteratoren öffentlich sichtbar.
- Es kann gefährlich sein, ein Aggregat zu verändern, während es von einem Iterator durchwandert wird. Eine scheinbar einfach Lösung besteht darin, die Elemente des Aggregats bei der Iterator-Erzeugung zu kopieren. Aus praktischer Sicht ist diese Lösung meist viel zu aufwändig. Oft will man, dass ein Iterator Änderungen auf einem Aggregat „sieht“. Es ist strittig, ob ein Iterator auf einer Kopie überhaupt als Iterator auf dem Original angesehen werden kann. Ein *robuster Iterator* erreicht das Ziel ohne Kopieren der Daten.

Template-Method

Definiert das Grundgerüst eines Algorithmus in einer Operation, überlässt die Implementierung einiger Schritte aber einer Unterklasse.

Anwendbar

- um den unveränderlichen Teil eines Algorithmus nur einmal zu implementieren und es Unterklassen zu überlassen, den veränderbaren Teil des Verhaltens festzulegen,
- wenn gemeinsames Verhalten mehrerer Unterklassen in einer einzigen Klasse lokal zusammengefasst werden soll,
- um mögliche Erweiterungen in Unterklassen zu kontrollieren, beispielsweise durch Template-Methods, die *Hooks* als primitive Operationen (siehe Strukturzeichnung) aufrufen und nur das Überschreiben dieser Hooks in Unterklassen ermöglichen.

Ein Hook ist eine Methode mit einer Default-Implementierung, die dafür vorgesehen ist, in Untertypen überschrieben zu werden.

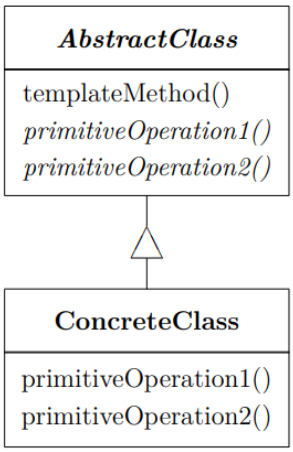

Abbildung 3: Template-Method Struktur

AbstractClass implementiert als „templateMethod“ das Grundgerüst des Algorithmus, das die primitiven Operationen aufruft. Jede von „templateMethod“ aufgerufene Methode wird als primitive Operation bezeichnet und stellt einen Schritt in der Ausführung der „templateMethod“ dar.

Eigenschaften:

- Fundamentale Technik zur direkten Wiederverwendung von Programmcode.
- Umkehrung der üblichen Kontrollstruktur, die manschmal als *Hollywood-Prinzip* bezeichnet wird („Don’t call us, we’ll call you.“). Das bedeutet, die Oberklasse ruft Methoden der Unterklasse auf — nicht wie in den meisten Fällen umgekehrt.
- „templateMethod“ ruft folgende Arten von primitiven Operationen auf
  - konkrete Operationen in *AbstractClass*, also Operationen, die ganz allgemein auch für Unterklassen sinnvoll sind;
  - abstrakte primitive Operationen, die einzelne Schritte im Algorithmus ausführen und in „ConcreteClass“ implementiert werden;
  - Hooks
  - Factory-Methods, also Methoden, die in Unterklassen neue Objekte erzeugen und zurückgeben und damit die Template-Method auch zu einem anderen Entwurfsmuster werden lassen (siehe *Factory-Method*).

Ziel bei der Entwicklung einer Template-Method sollte sein, die Anzahl der primitiven Operationen möglichst klein zu halten.

Erzeugende Entwurfsmuster

Factory-Method

Auch *Virtual-Constructor*.

Anwendbar wenn

- eine Klasse Objekte erzeugen soll, deren Klasse sie aber nicht kennt,
- eine Klasse möchte, dass ihre Unterklassen die Art der Objekte bestimmen, welche die Klasse erzeugt,
- Klassen Verantwortlichkeiten an eine von mehreren Unterklassen delegieren und das Wissen, an welche Unterklasse delegiert wird, lokal gehalten werden soll,
- die Allokation und Freigabe von Objekten zentral in einer Klasse verwaltet werden soll.

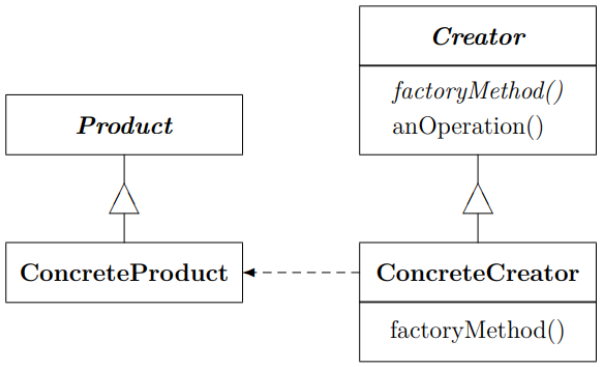

Abbildung 4: Factory-Method Struktur

Eigenschaften:

- Bieten Anknüpfungspunkte für Unterklassen, indem sie genau vorgeben, welche Methoden für das Überschreiben vorgesehen sind. Das führt zu einer Umkehrung der Abhängigkeiten: Oberklassen hängen von Unterklassen ab. Die Erzeugung eines neuen Objekts mittels Factory-Method ist fast immer flexibler als die direkte Objekterzeugung. Vor allem wird die Entwicklung von Unterklassen vereinfacht.
- Verknüpfen *parallele Typhierarchien*, die *Creator*-Hierarchie mit der *Product*-Hierarchie.

Prototype

Dient dazu, die Art eines neu zu erzeugenden Objekts durch ein Prototyp-Objekt zu spezifizieren. Neue Objekte werden durch Kopieren dieses Prototyps erzeugt.

Generell anwendbar, wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und dargestellt werden, und wenn

- die Klassen, von denen Objekte erzeugt werden sollen, erst zur Laufzeit bekannt sind, oder
- wenn Factory-Method Hierarchien vermieden werden sollen, oder
- jedes Objekt einer Klasse nur wenige unterschiedliche Zustände haben kann; es ist oft einfacher, für jeden möglichen Zustand einen Prototyp zu erzeugen.

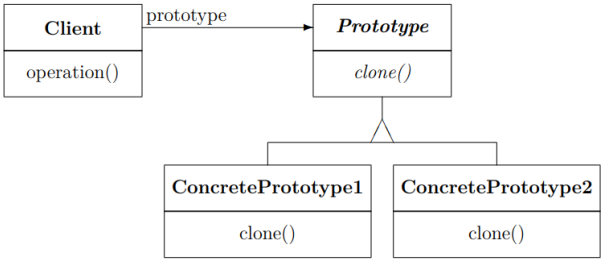

Abbildung 5: Prototype Struktur

Eigenschaften:

- Konkrete Produktklassen werden vor den Anwendern versteckt und die Anzahl der Klassen, die Anwender kennen müssen wird dadurch reduziert. Die Anwender müssen nicht geändert werden, wenn neue Produktklassen dazukommen oder geändert werden.
- Prototypen können auch zur Laufzeit jederzeit dazugegeben und weggenommen werden (Klassenstruktur zur Laufzeit nicht änderbar).
- Sie erlauben die Spezifikation neuer Objekte durch änderbare Werte.
- Sie vermeiden eine übertrieben große Anzahl an Unterklassen. Im Gegensatz zur Factory-Method ist es nicht nötig, parallele Klassenhierarchien zu erzeugen.
- Sie erlauben die dynamische Konfiguration von Programmen.

Um die Verwendung dieses Entwurfsmusters zu fördern, haben die Entwickler von Java die Methode `clone` bereits in `Object` vordefiniert.

Singleton

Sichert zu, dass eine Klasse nur eine Instanz hat und erlaubt globalen Zugriff auf dieses Objekt.

Anwendbar wenn

- es genau ein Objekt einer Klasse geben soll und dieses global zugreifbar sein soll;
- die Klasse durch Vererbung erweiterbar sein soll und Anwender die erweiterte Klasse ohne Änderungen verwenden können sollen.

```
public class Singleton {
    private static Singleton singleton = null;
    private Singleton() {} // no object creation from outside
```

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

Obwohl die Erklärung so einfach ist, sind einige Probleme bei der Implementation kaum zu lösen, weswegen heute oft von der Verwendung dieses Entwurfsmusters abgeraten wird. Konkret wird in abgewandelten Varianten häufig auf die Unterstützung von Vererbung verzichtet.

Eigenschaften:

- Kontrollierter Zugriff auf das einzige Objekt.
- Verzicht auf globale Variablen.
- Vererbung wird unterstützt (jedoch nicht in abgewandelten Varianten).
- Unkontrollierte Erzeugung von Instanzen wird verhindert (Konstruktor sind in der Regel nicht `public`).
- Prinzipiell auch mehrere Instanzen erzeugbar. Klasse hat vollständige Kontrolle darüber, wie viele Objekte erzeugt werden.
- Auf das `instance` Objekt kann über Objektmethoden durch dynamisches Binden flexibler zugegriffen werden.

6.3 Entwurfsmuster für Struktur

Decorator

Auch *Wrapper*.

Anwendbar

- um dynamisch Verantwortlichkeiten zu einzelnen Objekten hinzuzufügen, ohne andere Objekte zu beeinflussen;
- für Verantwortlichkeiten, die wieder entzogen werden können;
- wenn Erweiterungen einer Klasse durch Vererbung unpraktisch sind, z.B. um eine sehr große Zahl an Unterklassen zu vermeiden.

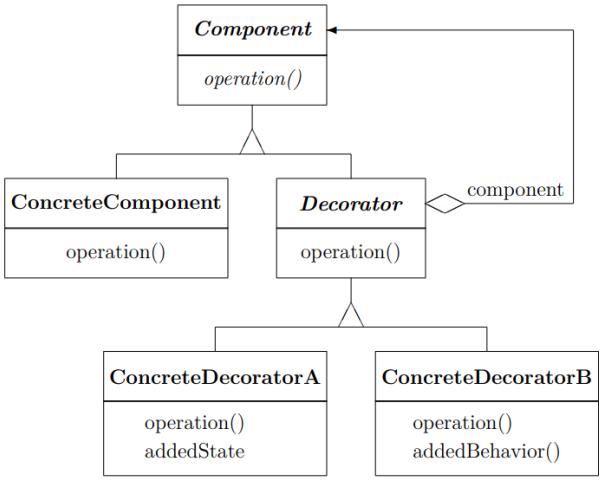

Abbildung 6: Decorator Struktur

Eigenschaften:

- Mehr Flexibilität als statische Vererbung.
- Vermeiden Klassen, die bereits weit oben in der Typhierarchie mit Methoden und Variablen überladen sind.
- Objekte von *Decorator* und die dazugehörenden Objekte von „ConcreteComponent“ sind nicht identisch.
- Führen zu vielen kleinen Objekten.

Proxy

Auch *Surrogate*, stellt einen Platzhalter für ein anderes Objekt dar und kontrolliert Zugriffe darauf.

Anwendbar, wenn eine intelligenter Referenz auf ein Objekt als ein simpler Zeiger nötig ist. Einige übliche Situationen:

- **Remote-Proxies:** Platzhalter für Objekte, die in anderen Namensräumen existieren. Nachrichten an die Objekte werden von den Proxies über komplexere Kommunikationskanäle weitergeleitet.
- **Virtual-Proxies:** erzeugen Objekte bei Bedarf (Performance Gründe).
- **Protection-Proxies:** kontrollieren Zugriffe auf Objekte.
- **Smart-References:** können bei Zugriffen zusätzliche Aktionen ausführen, z.B. Mitzählen der Referenzen, das Laden von Objekten in den Speicher bei erstmaligem Aufrufen, …

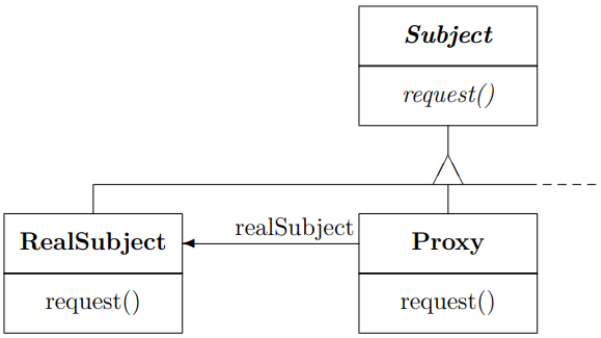

Abbildung 7: Proxy Struktur