

3 Objektorientiertes Design

Dieses Kapitel gibt einen Einblick in den objektorientierten Entwurf von Software. Die zugrunde liegende Idee der Objektorientierung (OO) ist es, **Zustand** (Daten) mit **Verhalten** (Funktionen auf diesen Daten) zu verbinden. Prinzipiell lässt sich diese Idee in jeder Programmiersprache realisieren. Sprachen wie z. B. C++, Java und Smalltalk unterstützen das objektorientierte Vorgehen von Hause aus. Ziel dieses Kapitels ist es, die Grundgedanken der Objektorientierung und ihre verfolgten Ziele **Kapselung**, **Trennung von Zuständigkeiten** und **Wiederverwendbarkeit** zu vermitteln.

Abschnitt 3.1 stellt zunächst einige Grundbegriffe und -gedanken vor und verdeutlicht diese am Entwurf eines Zählers. Das Beispiel ist bewusst einfach gewählt. Nichtsdestotrotz kann daran bereits sehr gut der OO-Entwurf mit seinen Tücken kennengelernt werden. Die gewonnenen Erkenntnisse helfen, die OO-Grundkonzepte besser nachvollziehen zu können. Anschließend lernen wir in Abschnitt 3.2 die grundlegenden OO-Techniken, wie Interfaces und abstrakte Klassen, kennen. Die technischen Grundlagen sind damit vorgestellt. Um Objektorientierung als Konzept zu erfassen und das **Denken in Objekten und Zuständigkeiten** zu verinnerlichen, stellt Abschnitt 3.3 Irrwege der »funktionalen Objektivierung«¹ vor. Darunter versteht man, dass häufig fälschlicherweise von Objektorientierung gesprochen wird, nur weil bei der Programmierung Klassen und Objekte eingesetzt werden. Objektorientierung umfasst aber viel mehr als das Programmieren mit Klassen und die intensive Benutzung von Vererbung. Nachdem wir die Vorteile der objektorientierten Programmierung kennengelernt haben, beschäftigen wir uns in Abschnitt 3.4 mit einigen fortgeschritteneren OO-Techniken. Dies sind unter anderem das Read-only-Interface, die Immutable-Klasse, das Marker-Interface und das Enum-Muster. Wir diskutieren außerdem kritisch das Thema Vererbung und betrachten sowohl Situationen von »explodierenden« Klassenhierarchien als auch Wege, dieses Problem zu lösen.

Mit JDK 5 wurden generische Typen, auch **Generics** genannt, eingeführt, um typsichere Klassen und insbesondere typsichere Containerklassen realisieren zu können. Dieses Thema stellt Abschnitt 3.6 vor. Zuvor wird in Abschnitt 3.5 der Begriff Varianz eingeführt. Das Verständnis verschiedener Formen der Varianz hilft, einige Besonderheiten beim Einsatz von Generics nachvollziehen zu können.

¹Danke an meinen Freund Tim Bötzmeyer für diese schöne Wortschöpfung und die Erlaubnis, diese hier einsetzen zu dürfen.

3.1 OO-Grundlagen

Die objektorientierte Softwareentwicklung setzt eine spezielle Denkweise voraus. Kerngedanke dabei ist, den Programmablauf als ein Zusammenspiel von Objekten und ihren Interaktionen aufzufassen. Dabei erfolgt eine Anlehnung an die reale Welt, in der Objekte und ihre Interaktionen ein wesentlicher Bestandteil sind. Beispiele dafür sind Personen, Autos, CD-Spieler usw. All diese Dinge werden durch spezielle Merkmale und Verhaltensweisen charakterisiert. Auf die Software übertragen realisieren viele einzelne Objekte durch ihr Verhalten und ihre Eigenschaften zur Laufzeit die benötigte und gewünschte Programmfunktionalität.

Bevor ich diese Gedanken anhand eines Beispiels mit dem Entwurf eines Zählers wieder aufgreife, möchte ich zuvor einige Begriffe definieren, die für ein gemeinsames Verständnis dieses und aller folgenden Kapitel eine fundamentale Basis darstellen.

3.1.1 Grundbegriffe

Klassen und Objekte Sprechen wir beispielsweise über ein spezielles Auto, etwa das von Hans Mustermann, so reden wir über ein konkretes *Objekt*. Sprechen wir dagegen abstrakter von Autos, dann beschreiben wir eine Klasse. Eine *Klasse* ist demnach eine Strukturbeschreibung für Objekte und umfasst eine Sammlung von beschreibenden *Attributen* (auch *Membervariablen* genannt) und *Methoden*, die in der Regel auf diesen Daten arbeiten und damit Verhalten definieren. Ein Objekt ist eine konkrete Ausprägung (*Instanz*) einer Klasse. Die Objekte einer Klasse unterscheiden sich durch ihre Position im Speicher und evtl. in den Werten ihrer Daten.

Objektzustand Der *Objektzustand* beschreibt die momentane Wertebelegung aller Attribute eines Objekts. Er sollte bevorzugt durch Methoden des eigenen Objekts verändert werden. Dadurch können Veränderungen leichter unter Kontrolle gehalten werden.

Interfaces Mithilfe von *Interfaces* definieren wir ein »Angebot von Verhalten« in Form einer Menge von Methoden ohne Implementierung. Man spricht hier von abstrakten Methoden. Eine Klasse, die das Interface implementiert, muss diese abstrakten Methoden mit Funktionalität füllen.

Abstrakte Klassen *Abstrakte Klassen* erlauben ähnlich wie Interfaces die Vorgabe von Methoden. Um eine Basisfunktionalität anzubieten, können abstrakte Klassen im Unterschied zu Interfaces bereits Funktionalität in Methoden implementieren. Methoden, die noch keine Implementierung besitzen, müssen dann mit dem Schlüsselwort `abstract` gekennzeichnet werden.

Typen Durch Klassen und Interfaces kann man neue, eigenständige *Datentypen* oder kurz Typen mit Verhalten beschreiben, etwa eine Klasse `Auto`. Im Unterschied dazu existieren *primitive Datentypen*, die lediglich Werte ohne Verhalten darstellen. Beispiele hierfür sind `int`, `float` usw.

Realisierung Das Implementieren eines Interface durch eine Klasse wird als *Realisierung* bezeichnet und durch das Schlüsselwort `implements` ausgedrückt. Das bedeutet, dass die Klasse alle Methoden des Interface anbietet.² Semantisch bedeutet dies, dass ein Objekt einer Klasse sich so verhalten kann, wie es das Interface vorgibt. Daher spricht man von *Typkonformität* oder auch von einer »*behaves-like*«-*Beziehung* bzw. »*can-act-like*«-*Beziehung*.

Deklaration und Definition Eine *Deklaration* beschreibt bei Variablen deren Typ und ihren Namen, etwa `int age`. Bei Methoden entspricht eine Deklaration dem Methodennamen, den Typen der Übergabeparameter sowie angegebenen Exceptions. Diese Elemente beschreiben die *Signatur* einer Methode. Von einer *Definition* einer Variablen spricht man, wenn ihr bei der Deklaration ein Wert zugewiesen wird. Die Definition einer Methode entspricht der Methodenimplementierung.

Info: Signatur einer Methode

Unter der Signatur einer Methode versteht man die Schnittstelle, die aus dem Namen, der Parameterliste (Anzahl, Reihenfolge und Typen der Parameter) und optional angegebenen Exceptions besteht. Betrachten wir folgende Methode `open()`:

```
boolean open (final String filename, final int numberOfRetries) throws
    IOException
{
    // ...
}
```

Weder die Implementierung, der Rückgabewert noch die Namen der Parameter und die `final`-Schlüsselworte sind Bestandteil der Signatur. Als Signatur ergibt sich demnach:

```
open (String, int) throws IOException
```

Referenzen Definiert man in Java eine Variable vom Typ einer Klasse, so stellt diese nicht das Objekt selbst dar, sondern nur eine Referenz auf das Objekt. Eine Referenz ist ein Verweis, um das Objekt zu erreichen.³

²Werden nur einige Methoden des Interface implementiert, so ist die Klasse abstrakt.

³Diese entspricht der Position im Speicher, wo das Objekt abgelegt ist, nachdem dieses erzeugt wurde.

Call-by-Value / Call-by-Reference Beim Aufruf von Methoden können Parameter an diese übergeben werden. Alle primitiven Typen werden als Wert (*Call-by-Value*) übergeben. Änderungen an den Parameterwerten innerhalb der Methode sind für aufrufende Methoden nicht sichtbar. Parameter in Form von Objekten werden in Java als Referenz (*Call-by-Reference*) übergeben. Es wird lediglich ein Verweis auf das Objekt übergeben. Das hat weitreichende Konsequenzen bei Änderungen. Diese wirken sich auf das Originalobjekt aus und sind auch außerhalb der aufgerufenen Methode sichtbar. Man spricht hier von der *Referenzsemantik* von Java. Auf weitere Auswirkungen werde ich im Verlauf dieses Buchs detailliert eingehen.

Sichtbarkeiten Beim objektorientierten Programmieren unterscheidet man verschiedene Sichtbarkeiten, die festlegen, ob und wie andere Klassen auf Methoden und Attribute zugreifen dürfen. Java bietet folgende vier Sichtbarkeiten:

- `public` – von überall aus zugreifbar
- `protected` – Zugriff für abgeleitete Klassen und alle Klassen im selben Package
- `Package-private` oder `default` (kein Schlüsselwort⁴) – nur Klassen im selben Package haben Zugriff darauf
- `private` – nur die Klasse selbst und alle inneren Klassen haben Zugriff

Kapselung Die *Kapselung* von Daten (*Information Hiding*) stellt einen Weg dar, die Attribute eines Objekts vor direkter Manipulation durch andere Objekte zu schützen. Das geschieht durch Einschränkung der Sichtbarkeit auf `protected`, `Package-private` oder `private`. Über Methoden kann der Zugriff auf Attribute gesteuert und eingeschränkt werden. Auch für Methoden kann man über Sichtbarkeitsregeln eine Strukturierung und Kapselung erreichen.

Objektverhalten und Business-Methoden Das Verhalten der Instanzen einer Klasse ist durch die bereitgestellten Methoden definiert. Diese Methoden weisen unterschiedliche Abstraktionsgrade und verschiedene Sichtbarkeiten auf. Einige Arbeits- und Hilfsmethoden verwenden viele Implementierungsdetails und arbeiten direkt mit den Attributen der Klasse. Sie sollten bevorzugt `private` oder `Package-private` definiert werden. Meistens realisieren nur wenige der Methoden einer Klasse »High-Level-Operationen« mit einem hohen Abstraktionsgrad. Diese verhaltensdefinierenden Methoden bilden in der Regel die Schnittstelle der Klasse nach außen.⁵ Derartige Methoden nennt man auch *Business-Methoden*. Sie verstecken die komplexen internen Vorgänge. Auf diese Weise erreicht man folgende Dinge:

⁴Man kann diese Sichtbarkeit durch einen Kommentar der Form `/*private*/` bzw. `/*package*/` andeuten. Dies ist hilfreich, um versehentliche Sichtbarkeitserweiterungen zu verhindern. Man dokumentiert derart, dass bewusst diese Sichtbarkeit gewählt wurde.

⁵Die Schnittstelle kann explizit über ein Interface beschrieben werden oder aber durch die Definition von Methoden mit der Sichtbarkeit `public`.

- **Abstraktion und Datenkapselung** – Implementierungsdetails werden versteckt.
- **Klare Abhängigkeiten** – Wenn Zugriffe durch andere Klassen nur über die Business-Methoden erfolgen, existieren wenige und klare Abhängigkeiten.
- **Austauschbarkeit und Wiederverwendbarkeit** – Werden die Business-Methoden durch ein Interface beschrieben, so kann ein Austausch der Realisierung ohne Rückwirkung auf Nutzer erfolgen.

Tipp: Zusammensetzung und Aufrufhierarchien von Methoden

Um den Sourcecode übersichtlich zu halten, sollten sich öffentliche Methoden aus Methodenaufrufen anderer Sichtbarkeiten zusammensetzen und wenig Implementierungsdetails zeigen. Zudem sollten Methoden niemals andere Methoden höherer Sichtbarkeit aufrufen; eine private Methode »darf« demnach keine öffentliche Methode aufrufen. Diese Strukturierung hilft vor allem, wenn man Datenänderungen verarbeiten und an andere Klassen kommunizieren muss. Ohne die Einhaltung dieses Hinweises kommt es schnell zu einem Chaos.

Kohäsion Beim objektorientierten Programmieren verstehen wir unter *Kohäsion*, wie gut eine Methode tatsächlich genau eine Aufgabe erfüllt. Je höher die Kohäsion, desto besser realisiert eine Methode lediglich eine spezielle Funktionalität. Bei hoher Kohäsion ist eine gute Trennung von Zuständigkeiten erfolgt. Methoden mit höher Kohäsion können normalerweise gut kombiniert werden, um neue Funktionalitäten zu realisieren. Dies hilft bei der Wiederverwendbarkeit. Je niedriger auf der anderen Seite die Kohäsion ist, desto mehr wird unterschiedliche Funktionalität innerhalb einer Methode realisiert. Analog können wir von Kohäsion einer Klassen reden. Sie beschreibt, wie genau abgegrenzt die Funktionalität der Klasse ist.

Info: Orthogonalität und Wiederverwendung

In der Informatik spricht man von *Orthogonalität*, wenn man eine freie Kombinierbarkeit unabhängiger Konzepte – hier Methoden und Klassen – erreicht. Eine derartige Implementierung zu erstellen, erfordert allerdings einiges an Erfahrung. Häufig sind Methoden daher in der Praxis eben nicht so gestaltet, dass sie nur genau eine Aufgabe erfüllen. Um eine benötigte Teilfunktionalität einer Methode oder Klasse zu verwenden, und weil man nur diese eine Aufgabe in Form eines Methodenaufrufs nicht bekommen kann, werden dann häufig die entsprechenden Zeilen kopiert, statt eine Methode mit der gewünschten Funktionalität herauszulösen. Eine derartige Sourcecode-Duplizierung (Copy-Paste-Wiederverwendung) führt häufig zu einer schwachen Kohäsion und sollte daher möglichst vermieden werden.^a

^aWird eine ganze Methode in eine andere Klasse kopiert, könnte sie durchaus eine hohe Kohäsion aufweisen. Allerdings ist dies wegen der Duplizierung wartungsunfreundlich.

Assoziationen Stehen Objekte in Beziehung zueinander, so spricht man ganz allgemein von einer *Assoziation*. Diese Beziehung kann nach der Zusammenarbeit wieder aufgelöst werden und es können neue Assoziationen zu anderen Objekten aufgebaut werden.

Man kann die Assoziation noch feiner in Aggregation und Komposition unterteilen. *Aggregation* drückt aus, dass ein Objekt eine Menge von anderen Objekten referenziert. Von *Komposition* spricht man, wenn eine Ganzes-Teile-Beziehung realisiert wird, sprich: Das enthaltene Teilobjekt kann ohne das Ganze nicht selbstständig existieren.

Je nach Modellierungsabsicht stellt die in Abbildung 3-1 dargestellte Beziehung zwischen einer Musik-CD und einigen MP3-Liedern eine Aggregation bzw. eine Komposition dar. Für die Betrachtung als Aggregation existieren die MP3-Lieder unabhängig von der CD, z. B. als Dateien einer Play-Liste. Bei der Komposition sind die MP3-Lieder Bestandteil der CD in Form von Bits und Bytes auf dem Datenträger selbst.

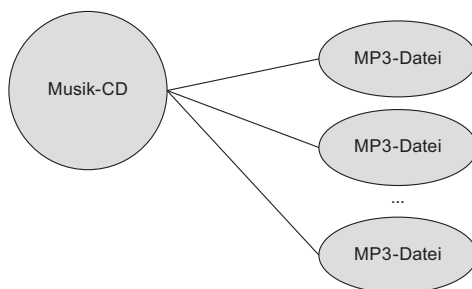


Abbildung 3-1 Aggregation / Komposition

Kopplung Unter *Kopplung* verstehen wir, wie stark Klassen miteinander in Verbindung stehen, also den Grad ihrer Abhängigkeiten untereinander. Einen großen Einfluss hat, welche Attribute und Methoden sichtbar und zugreifbar sind und wie dieser Zugriff erfolgt. Zwei Klassen sind stark miteinander gekoppelt, wenn entweder viele feingranulare Methodenaufrufe der jeweils anderen Klasse erfolgen oder aber auf Attribute der anderen Klasse direkt zugegriffen wird. Viele Methodenzugriffe zur Veränderung des Objektzustands deuten auf das Designproblem mangelnder Kapselung hin. Bei Unachtsamkeit pflanzt sich diese fort und führt dazu, dass jede Klasse viele Details anderer Klassen kennt und mit diesen eng verbunden (stark gekoppelt) ist. Man kann dann von »Objekt-Spaghetti« sprechen. Häufig ist starke Kopplung durch mangelnde Trennung von Zuständigkeiten, also eine schlechte Kohäsion, begründet.

Ziel einer Modellierung ist es, eine möglichst lose Kopplung und damit geringe Abhängigkeiten verschiedener Klassen untereinander zu erreichen. Durch eine gute Datenkapselung sowie eine Definition von Objektverhalten in Form von Business-Methoden erreicht man dies: Durch eine gute Kohäsion definiert man klare Zuständigkeiten. Klassen sind möglichst eigenständig und unabhängig von anderen. Mithilfe einer guten Kapselung wird dieser Effekt dahingehend verstärkt, dass weniger Realisierungsdetails für

andere Klassen sichtbar sind. Als Folge kann eine kleine Schnittstelle mit wenigen Methoden zur Kommunikation mit anderen Klassen definiert werden.

Tip: Vorteile loser Kopplung und guter Kohäsion

Durch den Einsatz von loser Kopplung lassen sich nachträgliche Änderungen meistens einfacher und ohne größere Auswirkungen für andere Klassen umsetzen. Eine starke Kopplung führt dagegen häufig dazu, dass Änderungen in vielen Klassen notwendig werden. **Gutes OO-Design besteht darin, jede Klasse so zu gestalten, dass deren Aufgabe eindeutig ist, jede Aufgabe nur durch eine Klasse realisiert wird und die Abhängigkeiten zwischen Klassen möglichst minimal sind.**

Vererbung Eine spezielle Art, neue Klassen basierend auf bestehenden Klassen zu definieren, wird **Vererbung** genannt. Das wird durch das Schlüsselwort `extends` ausgedrückt: Die neu entstehende Klasse erweitert oder übernimmt (erbt) durch diesen Vorgang das Verhalten und die Eigenschaften der bestehenden Klasse und wird **abgeleitete** oder **Subklasse** genannt. Die wiederverwendete Klasse bezeichnet man als **Basis-, Ober- oder Superklasse**. Im Gegensatz zum Copy-Paste-Ansatz muss die Subklasse nur noch die Unterschiede zu ihrer Basisklasse beschreiben und nicht komplett neu entwickelt werden. Durch Vererbung entsteht eine **Klassenhierarchie**. Von einer **Spezialisierung** spricht man, wenn man die Klassenhierarchie gedanklich in Richtung Subklassen durchläuft. Eine **Generalisierung** ist der Weg in Richtung Basisklassen. Dies zeigt Abbildung 3-2.

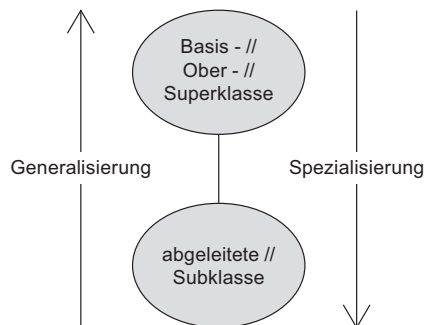


Abbildung 3-2 Generalisierung und Spezialisierung

Vererbung sollte nur dann eingesetzt werden, wenn die sogenannte »is-a«-Beziehung erfüllt ist. Diese besagt, dass **Subklassen tatsächlich eine semantische Spezialisierung ihrer Basisklasse darstellen und alle Eigenschaften der Basisklasse besitzen**. Wird diese Forderung nicht beachtet, so handelt es sich um eine sogenannte **Implementierungsvererbung**. Diese ist zu vermeiden, weil dann durch Vererbung nur benötigte Funktionalität übernommen wird, jedoch semantisch kein Subtyp definiert wird: Objekte einer

Subklasse können dann konzeptuell nicht mehr als Objekte der Basisklasse betrachtet werden. Abbildung 3-3 zeigt ein Positiv- und ein Negativbeispiel.

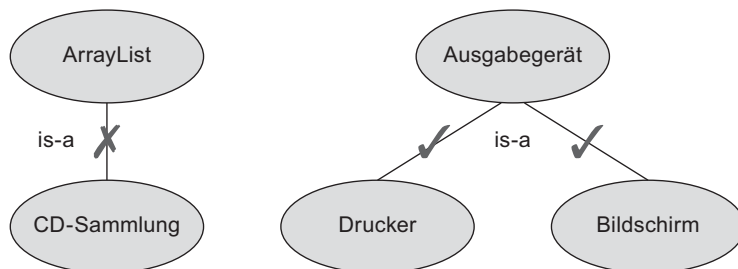


Abbildung 3-3 Vererbung und »is-a«-Beziehung

Sub-Classing und Sub-Typing Spezialisierung ist sowohl zwischen Klassen als auch zwischen Interfaces möglich. Beides wird durch das Schlüsselwort `extends` ausgedrückt. Zwischen Klassen wird durch Spezialisierung ein Vererben von Verhalten erreicht, d. h., eine Klasse ist eine spezielle Ausprägung einer anderen Klasse, übernimmt deren Verhalten und fügt eigene Merkmale und Verhaltensweisen hinzu. Hier spricht man von **Sub-Classing**. Eine Spezialisierung eines Interface erweitert die Menge der Methoden eines anderen Interface. In diesem Fall spricht man von **Sub-Typing**. Für Klassen spricht man häufiger der Einfachheit halber auch von Sub-Typing. Dies erleichtert die Diskussion, denn eine Vererbung zwischen Klassen ist streng genommen sowohl Sub-Classing als auch Sub-Typing. Das Implementieren eines Interface ist jedoch nur Sub-Typing.

Explizite Typumwandlung – Type Cast (Cast) Unter einer expliziten Typumwandlung, einem sogenannten **Type Cast** oder kurz **Cast**, versteht man eine »Aufforderung« an den Compiler, eine Variable eines speziellen Typs in einen anderen angegebenen Typ umzuwandeln. Solange man dabei in der Typhierarchie nach oben geht, ist diese Umwandlung durch die »is-a-Beziehung« abgesichert. Möchte man jedoch einen Basistyp in einen spezielleren Typ konvertieren, so geht man in der Ableitungshierarchie nach unten. Eine derartige Umwandlung wird als **Down Cast** bezeichnet und ist durch die »is-a-Beziehung« nicht abgesichert. Betrachten wir dies beispielhaft:

```
final Sub sub = (Sub) obj; // Unsicherer Down Cast: Object -> Base -> Sub
```

Nur für den Fall, dass die Referenz `obj` ein Objekt vom Typ `Sub` (oder eine Spezialisierung davon) enthält, ist der Cast erfolgreich. Ansonsten kommt es zu einer Inkompatibilität von Typen, wodurch eine `java.lang.ClassCastException` ausgelöst wird. Einen solchen Cast sollte man daher möglichst vermeiden oder zumindest durch eine explizite Typprüfung mit `instanceof` absichern:

```

if (obj instanceof Sub)    // Absicherung des Down Cast: Object -> Base -> Sub
{
    final Sub sub = (Sub) obj;

    // ...
}
else
{
    // Problem: Wie soll man auf diese Situation reagieren?
}

```

Eine erfolgreiche Umwandlung kann so zwar garantiert werden, es stellt sich dann aber die Frage, wie auf alle nicht erwarteten Typen reagiert werden sollte. In der Regel kann sinnvollerweise nur eine Fehlermeldung ausgegeben werden.

Polymorphie Variablen eines Basistyps können beliebige davon abgeleitete Spezialisierungen referenzieren. Das wird als Vielgestaltigkeit oder *Polymorphie* bezeichnet. Polymorphie basiert auf dem Unterschied zwischen dem *Kompiliertyp* und dem *Laufzeittyp*. Der Kompiliertyp ist der zur Kompilierzeit bekannte Basistyp. Der Laufzeittyp entspricht der konkret verwendeten Spezialisierung. Betrachten wir eine Klassenhierarchie der Klassen `Base`, `Sub` und `SubSub` sowie deren Methoden `doIt()` und `doThat()`. Die Klassenhierarchie ist in Abbildung 3-4 dargestellt.

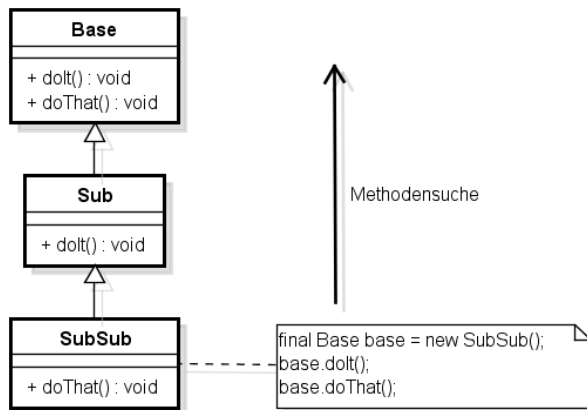


Abbildung 3-4 Polymorphie und dynamisches Binden

Ein Beispiel für Polymorphie ist, dass die Variable `base` den Kompiliertyp `Base` besitzt und während der Programmausführung den Laufzeittyp `SubSub`. Damit Polymorphie funktioniert, muss immer die spezialisierteste Methode eines Objekts verwendet werden, d. h. die Implementierung der spezialisiertesten Subklasse, die diese Methode anbietet: Zur Bestimmung der auszuführenden Methode wird dazu startend bei dem Laufzeittyp nach einer passenden Methode gesucht. Für den Aufruf von `doIt()` startet die Suche daher in der Klasse `SubSub`. Dort wird die JVM allerdings nicht fündig, so dass die Suche sukzessive weiter nach oben in der Vererbungshierarchie fortgesetzt

wird, bis eine Methodendefinition existiert. In diesem Beispiel ist dies für `doIt()` in der Klasse `Sub` der Fall. Das beschriebene Verfahren zum Auffinden der auszuführenden Methode wird *dynamisches Binden* (*Dynamic Binding*) genannt.

Achtung: Polymorphie – Einfluss von Kompilier- und Laufzeittyp

Eine Quelle für Flüchtigkeitsfehler ist die Annahme, dass die polymorphen Eigenschaften von Objektmethoden auch für Attribute und statische Methoden gelten würden. Attribute und statische Methoden sind nicht polymorph: Es wird immer auf die durch den Kompiliertyp sichtbaren Methoden bzw. Attribute zugegriffen. Dieses wird leicht übersehen und kann zu falschem Systemverhalten führen.

3.1.2 Beispielentwurf: Ein Zähler

Nachdem nun die Grundbegriffe beim objektorientierten Entwurf bekannt sind, werden diese im Folgenden anhand eines Beispiels vertieft. Eine grafische Anwendung soll um eine Auswertung der Anzahl gezeichneter Linien und Rechtecke erweitert werden. Dazu ist ein Zähler als Klasse zu entwerfen, der folgende Anforderungen erfüllt:

1. Er lässt sich auf den Wert 0 zurücksetzen.
2. Er lässt sich um eins erhöhen.
3. Der aktuelle Wert lässt sich abfragen.

Mit diesen scheinbar einfach umzusetzenden Anforderungen eines Kollegen »User« lassen wir die zwei exemplarischen Entwickler »Schnell-Finger« und »Überleg-Erst-Einmal« diese Aufgabe realisieren. Schauen wir uns an, wie die beiden arbeiten.

Bevor wir beiden Entwicklern bei der Arbeit zusehen, betrachten wir den Nutzungskontext. Der Anwendungscode ist bereits rudimentär implementiert:

```
// TODO: 2 Zähler initialisieren

for (final GraphicObject graphicObject : graphicObjects)
{
    graphicObject.draw();

    if (graphicObject instanceof Line)
    {
        // TODO: lineCounter erhöhen
    }

    if (graphicObject instanceof Rect)
    {
        // TODO: rectCounter erhöhen
    }
}

// TODO: Zähler auslesen und ausgeben
```

Beim Zeichnen wird ein `instanceof`-Vergleich⁶ verwendet, um die beiden Typen grafischer Objekte zu unterscheiden und den jeweiligen Zähler zu inkrementieren. Die eigentliche Funktionalität bleibt zunächst unimplementiert, da noch auf die konkrete Realisierung der Klasse `Counter` gewartet wird. Daher markieren `TODO`-Kommentare die Stellen, an denen später der Zähler eingesetzt werden soll.

Entwurf à la »Schnell-Finger«

Herr »Schnell-Finger« überlegt nicht lange und startet sofort seine IDE. Während diese noch hochfährt, hat er bereits ein paar Ideen. Kaum ist das Editor-Fenster geöffnet, legt er los. Klingt alles recht einfach, also wird schnell die folgende Klasse erstellt:

```
public class Counter
{
    public int count = 0;

    public Counter()
    {
    }
}
```

So, nun ist er schon fertig. Die Variable `count` ist öffentlich, kann also von überall abgefragt und verändert werden. Rücksetzen erfolgt über eine Zuweisung von 0. Zufrieden speichert er die Klasse. Am nächsten Tag fragt der Kollege »User«, wie weit die Klasse `Counter` wäre, und bekommt die Antwort: »Längst fertig!« Der Kollege »User« schaut sich die Klasse an und sagt: »Naja, richtig objektorientiert ist das nicht. Es fehlt an Kapselung: Du veröffentlichst deine Attribute und bietest keine Zugriffsmethoden an.« Etwas verärgert über den pingeligen Kollegen macht sich Herr »Schnell-Finger« nochmal an die Arbeit und kommt nach kurzer Zeit zu folgender Umsetzung:

```
public class Counter
{
    public int count = 0;

    public Counter()
    {
    }

    public int getCounter()
    {
        return count;
    }

    public void setCounter(int count)
    {
        count = count;
    }
}
```

⁶In der Regel sind `instanceof`-Vergleiche ein Anzeichen schlechten Programmierstils. Häufig lassen sich diese durch Polymorphie und gemeinsame Interfaces vermeiden, indem eine überschriebene Methode aufgerufen wird. Dies ist hier nicht gewünscht, da wirklich nur eine Fallunterscheidung getroffen werden soll. Daher ist `instanceof` in diesem Fall tolerabel.

Betrachten wir diese Klasse im Einsatz: Die `TODO`-Kommentare des Anwendungsfragments werden mithilfe der Klasse `Counter` folgendermaßen ersetzt:

```
// 2 Zähler initialisieren
final Counter lineCounter = new Counter();
final Counter rectCounter = new Counter();

for (final GraphicObject graphicObject : graphicObjects)
{
    graphicObject.draw();

    if (graphicObject instanceof Line)
    {
        // lineCounter erhöhen
        lineCounter.setCounter(lineCounter.getCounter() + 1);
    }

    if (graphicObject instanceof Rect)
    {
        // rectCounter erhöhen
        rectCounter.setCounter(rectCounter.getCounter() + 1);
    }
}

// Zähler auslesen und ausgeben
System.out.println("Number of Lines: " + lineCounter.getCounter());
System.out.println("Number of Rects: " + rectCounter.getCounter());
```

So recht zufrieden ist der nutzende Kollege »User« mit der Handhabung der Klasse `Counter` nicht. Der Sourcecode sieht irgendwie nicht rund aus. Es wird ein erster Test durchgeführt. Trotz vieler gemalter Linien und Rechtecke kommt es zu folgenden Ausgaben:

```
Number of Lines: 0
Number of Rects: 0
```

Der Anwendungscode sieht so weit okay aus. Ein Blick auf die Implementierung der Methode `setCounter(int)` offenbart jedoch den Flüchtigkeitsfehler: Der Übergabeparameter heißt genauso wie das Attribut. Dies lässt sich leicht korrigieren:

```
public void setCounter(final int newCount)
{
    this.count = newCount;
}
```

Entwurf à la »Überleg-Erst-Einmal«

Kollege »Überleg-Erst-Einmal« liest die Anforderungen und skizziert ein kleines UML-Klassendiagramm von Hand auf einem Blatt Papier, wie dies Abbildung 3-5 zeigt.

Sein Entwurf sieht zum Verarbeiten des Zählers die Methoden `getCounter()` und `setCounter()` vor. Zum Rücksetzen des Wertes dient die Methode `reset()`. Da er bereits einige Erfahrung im Softwareentwurf hat, weiß er, dass die erste Lösung meistens nicht die beste ist. Um seinen Entwurf zu prüfen, überlegt er sich ein paar Anwen-

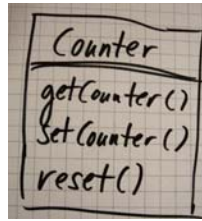


Abbildung 3-5 Die Klasse Counter

dungsfälle und spielt diese im Kopf durch. Später kann er daraus Testfälle definieren. Zudem schaut er nochmals auf die gewünschten Anforderungen. Dort steht: Der Zähler soll um eins erhöht werden. Folglich entfernt er die `setCounter()`-Methode und führt eine `increment()`-Methode ein. Da er bereits jetzt an Wiederverwendung und Dokumentation denkt, nutzt er ein UML-Tool, um die Klasse dort zu konstruieren. Abbildung 3-6 zeigt das Ergebnis.

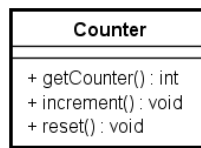


Abbildung 3-6 Die Klasse Counter, 2. Version

Als unbefriedigend empfindet er aber noch eine Inkonsistenz in der Namensgebung bzgl. des Methodennamens `getCounter()`. Der Name ist missverständlich: Hier wird kein Counter-Objekt zurückgeliefert, sondern dessen Wert. Namen wie z. B. `getValue()` bzw. `getCurrentValue()` wären damit aussagekräftiger und verständlicher. Eine weitere Alternative ist, auf das Präfix `get` im Namen zu verzichten. Damit ergeben sich `value()` und `currentValue()` als mögliche Methodennamen. In diesem Fall entscheidet er sich für letztere Variante, da sich die Namen dann besser lesen lassen. Der Einsatz des Präfixes `get` würde allerdings lesende Zugriffsmethoden auf einen Blick sichtbar machen.⁷ Ein entsprechendes Klassendiagramm ist in Abbildung 3-7 gezeigt.

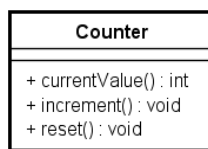


Abbildung 3-7 Die Klasse Counter, 3. Version

⁷Beide Varianten der Namensgebung sind sinnvoll, und die konkrete Wahl ist eine Frage des Geschmacks (oder aber durch sogenannte Coding Conventions (vgl. Kapitel 13) festgelegt).

Während er noch an der Implementierung arbeitet, kommt der Kollege »User« vorbei und berichtet, dass er die Lösung von Herrn »Schnell-Finger« als zu sperrig empfindet. Daher würde er gern eine weitere Lösung begutachten und in seine Anwendung einbauen. Die Implementierung der Klasse `Counter` ist zwar noch nicht komplett abgeschlossen. Deren Schnittstelle, das sogenannte *API (Application Programming Interface)*, d. h. die angebotenen Methoden, sind aber durch das UML-Diagramm bereits vollständig definiert. Daher kann der Applikationscode wie folgt auf das API der Klasse `Counter` von Herrn »Überleg-Erst-Einmal« angepasst werden:

```
final Counter lineCounter = new Counter();
final Counter rectCounter = new Counter();

for (final GraphicObject graphicObject : graphicObjects)
{
    graphicObject.draw();

    if (graphicObject instanceof Line)
    {
        lineCounter.increment();
    }

    if (graphicObject instanceof Rect)
    {
        rectCounter.increment();
    }
}

System.out.println("Number of Lines: " + lineCounter.currentValue());
System.out.println("Number of Rects: " + rectCounter.currentValue());
```

Der resultierende Sourcecode ist besser lesbar. Die Zeilen mit dem Aufruf der Methode `increment()` sind nun selbsterklärend, und die Kommentare konnten daher ersatzlos entfallen. Kollege »Überleg-Erst-Einmal« liefert später folgenden Sourcecode ab:

```
public final class Counter
{
    private int value = 0;

    public Counter()
    {
    }

    public int currentValue()
    {
        return value;
    }

    public void increment()
    {
        value++;
    }

    public void reset()
    {
        value = 0;
    }
}
```

Vergleich der Lösungen

Die Lösung von Herrn »Schnell-Finger« hat durch einige Iterationen und Kommentare von Kollegen einen brauchbaren Zustand erreicht. Allerdings macht diese Lösung ausschließlich von `get ()` -/`set ()` -Methoden Gebrauch, ohne aber Verhalten zu definieren (im Folgenden als `get ()` -/`set ()` -Ansatz bezeichnet). Hier herrscht meiner Ansicht nach das größte Missverständnis bzgl. der Objektorientierung vor: ***Ein intensiver Gebrauch von `get ()` - und `set ()` -Methoden ist kein Zeichen von Objektorientierung, sondern deutet häufig vielmehr auf einen fragwürdigen OO-Entwurf hin.*** Objekte kapseln dann kein Verhalten mehr – stattdessen wird dieses stark von anderen Klassen gesteuert. Das eigentliche Objektverhalten wird dann in den Applikationscode verlagert und dort realisiert, anstatt von den Objekten selbst definiert und ausgeführt zu werden. Dabei können Objektzustände häufig sogar derart von außen manipuliert werden, dass sich das Objektverhalten in einer unerwarteten (möglicherweise ungewünschten) Art und Weise verändert. Dies können wir im Zählerbeispiel von Herrn »Schnell-Finger« sehr schön daran sehen, wie die Inkrementierung des Zählers realisiert ist. Tatsächlich könnte der Zähler durch die angebotene `setCounter (int)` -Methode auf beliebige Werte gesetzt werden, also erhöht oder erniedrigt werden. Das Inkrement wird also im Applikationscode programmiert. Jede weitere Applikation müsste wiederum diese Funktionalität des Zählers selbst realisieren. Diese Art der Realisierung ist demnach vollständig an der Anforderung vorbei entwickelt, da lediglich ein besserer Datencontainer realisiert wird. Kurz gesagt: Die Klasse hat keine Business-Methoden. Damit verletzt diese Art des Entwurfs zum einen den Gedanken der Definition von Verhalten durch Objekte und zum anderen das Ziel der Wiederverwendbarkeit. Beim `get ()` -/`set ()` -Ansatz kann somit meistens keine Funktionalität wiederverwendet werden.

Die Klasse `Counter` von Herrn »Überleg-Erst-Einmal« realisiert dagegen ein logisches Modell. Hier stehen die technischen Details nicht im Vordergrund, sondern man konzentriert sich auf das Erfüllen einer Aufgabe. Design und Implementierung stellen die Funktionalität eines Zählens bereit und verhindern feingranulare Zugriffe auf interne Variablen. Dies verstärkt die Kapselung im Vergleich zu dem reinen `get ()` -/`set ()` -Ansatz, da die Methoden `increment ()` und `currentValue ()` Implementierungsdetails gut verbergen. Es ist somit von außen unmöglich, den Zähler um beliebige Werte zu erhöhen oder zu erniedrigen. Diese Umsetzung besitzt zudem eine höhere Kohäsion und damit eine bessere Abschirmung bei anstehenden Änderungen als der `get ()` -/`set ()` -Ansatz.

Tipp: Logisches Modell und Business-Methoden

Durch das Realisieren eines logischen Modells wird der entstehende Sourcecode in der Regel deutlich besser verständlich und menschenlesbar, da man Konzepten und nicht Programmanweisungen folgt. Veränderungen am Objektzustand werden dabei durch eine Reihe von verhaltensdefinierenden Business-Methoden durchgeführt.

Erweiterbarkeit

Wollten wir den Zähler beispielsweise um einen Überlauf bei einer bestimmten Schwelle erweitern und die Anzahl der Überläufe protokollieren, so müsste dies bei der Realisierung von Herrn »Schnell-Finger« jede Applikation selbst vornehmen. Das kann schnell zu einem Wartungsabtraum werden. Im Entwurf von Herrn »Überleg-Erst-Einmal« wäre genau eine Stelle zu ändern, und es fänden keine Fortpflanzungen der Änderungen in die nutzenden Applikationen statt. Folgendes Beispiel der Klasse `CounterWithOverflow` zeigt eine mögliche Realisierung. Die ursprüngliche Klasse wird um eine Konstante `COUNTER_MAX`, ein zusätzliches Attribut `overflowCount` und eine korrespondierende Zugriffsmethode erweitert:

```
public final class CounterWithOverflow
{
    private static final int COUNTER_MAX = 100;

    private int value = 0;
    private int overflowCount = 0;

    public CounterWithOverflow()
    {
    }

    public int currentValue()
    {
        return value;
    }

    public int overflowCount()
    {
        return overflowCount;
    }

    public void reset()
    {
        value = 0;
        overflowCount = 0;
    }

    public void increment()
    {
        if (value == COUNTER_MAX-1)
        {
            value = 0;
            overflowCount++;
        }
        else
        {
            value++;
        }
    }
}
```

Die Anpassungen in der `increment()`-Methode prüfen auf einen Überlauf und führen gegebenenfalls ein Rücksetzen des Zählers und eine Erhöhung des Überlaufzählers durch.

In einer Spieleapplikation könnte man eine solche Funktionalität beispielsweise dazu nutzen, nach 100 aufgesammelten Bonuselementen ein weiteres Leben zu erhalten.

Fazit

Anhand dieses Beispiels lässt sich sehr schön erkennen, dass neben dem reinen Einsatz von Klassen und Objekten vor allem auch die Definition von Verhalten und Zuständigkeiten (*Kohäsion*) sowie das gleichzeitige Verbergen von Implementierungsdetails (*Kapselung*) die objektorientierte Programmierung ausmachen. In komplexeren Entwürfen ergeben sich weitere Vorteile durch den sinnvollen Einsatz von Vererbung und die Konstruktion passender Klassenhierarchien. Klares Ziel ist es, Objekte mit Verhalten zu definieren. Der massive Einsatz von `get ()` / `set ()` -Methoden widerspricht diesem Ansatz und ist daher meist wenig objektorientiert. Er eignet sich vor allem für Datenbehälter, also Objekte, die eigentlich kein eigenes Verhalten besitzen und damit eher als Hilfsobjekte betrachtet werden sollten. In Abschnitt 3.4.6 wird dies als Entwurfsmuster VALUE OBJECT vorgestellt.

3.1.3 Diskussion der OO-Grundgedanken

Nachdem wir durch den Entwurf einer Zähler-Klasse einige Erkenntnisse bezüglich OO gewonnen haben, möchte ich die zu Beginn dieses Kapitels kurz vorgestellten Grundgedanken des OO-Entwurfs aufgreifen und diese mit dem neu gewonnenen Verständnis betrachten.

Datenkapselung und Trennung von Zuständigkeiten

Die Datenkapselung (das Verbergen von Informationen) stellen ein Kernkonzept der objektorientierten Programmierung dar. Beides ermöglicht, die Daten eines Objekts vor Änderungen durch andere Objekte zu schützen. Anstatt direkt auf Attribute zuzugreifen, bietet ein Objekt Zugriffsmethoden an. Diese werden im OO-Sprachjargon häufig als *Accessors* oder *Mutators* bezeichnet. Bei ersten OO-Gehversuchen verkümmern nahezu alle Methoden zu reinen Zugriffsmethoden ohne Objektverhalten. Warum dieser Einsatz von `get ()` - und `set ()` -Methoden nicht besonders objektorientiert ist, haben wir bereits im Verlauf des Zähler-Beispiels diskutiert. Des Weiteren kann nur über Zugriffsmethoden eine Konsistenzprüfung eingehender Werte und eine zuverlässige Synchronisierung für Multithreading erreicht werden.

In vielen Fällen ist der Einsatz von Zugriffsmethoden dem direkten Zugriff auf Attribute vorzuziehen. Die Datenkapselung erlaubt es, die Speicherung der Daten eines Objekts unabhängig von dem nach außen bereitgestellten Interface zu ändern. Wie bei allen Dingen gibt es aber auch Ausnahmen! Die Forderung nach Zugriffsmethoden sollte man nicht dogmatisch umsetzen: Für Package-interne Klassen oder lokal innerhalb einer anderen Klasse definierte Hilfsklassen ist es häufig akzeptabel, auf Zugriffsmethoden zu verzichten und direkt auf Attribute zuzugreifen. Dies gilt allerdings nur dann, wenn absehbar ist, dass

1. diese Klasse nicht außerhalb des eigenen Packages verwendet wird,
2. keine strukturellen Änderungen (neue Attribute, geänderte Assoziationen usw.) zu erwarten sind und
3. keine anderen Rahmenbedingungen, wie etwa Thread-Sicherheit, gegen eine direkte Nutzung und die dadurch reduzierte Kapselung sprechen.

Wir wollen uns die Vorteile der Datenkapselung anhand eines Beispiels ansehen: Es ist eine Klasse zur Modellierung von Personen und zugehörigen Adressdaten zu erstellen. Im Folgenden betrachten wir drei verschiedene Umsetzungen, die als UML-Klassendiagramm in Abbildung 3-8 dargestellt sind.

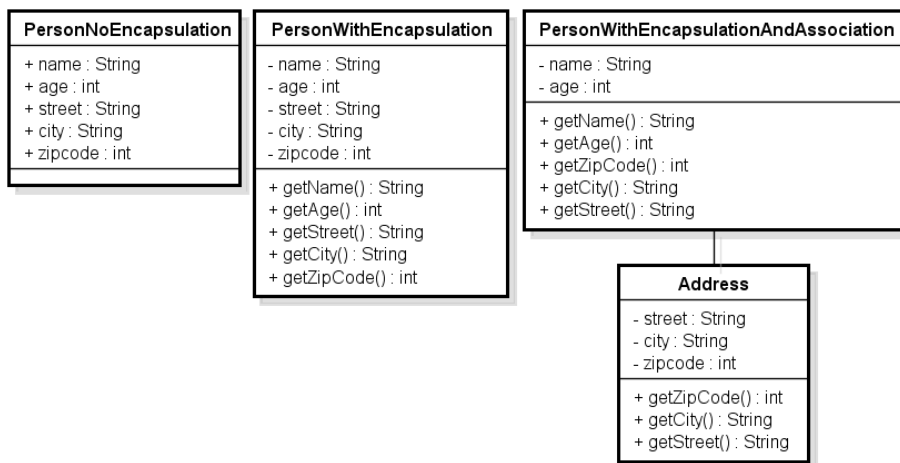


Abbildung 3-8 Kapselung am Beispiel unterschiedlicher Personenklassen

Die bisher gezeigten UML-Diagramme enthielten intuitiv verständliche Elemente. In diesem Diagramm sind bereits einige Feinheiten enthalten: Es ist zum Verständnis hilfreich, die Zeichen '+' und '-' zur Beschreibung der Sichtbarkeiten `public` und `private` zu kennen und zu wissen, dass diese für Attribute (mittlerer Kasten) und Methoden (unterer Kasten) angegeben werden können. Eine Verbindungslinie zwischen zwei Klassen beschreibt eine Assoziation. Details zur Modellierung mit der UML finden Sie in Anhang A.

Eine Realisierung ohne Kapselung ist mit der Klasse `PersonNoEncapsulation` gezeigt. Öffentlich zugängliche Attribute speichern die Informationen zu Name, Alter und Adressdaten. Zudem wird auf Zugriffsmethoden verzichtet. Dadurch müssen andere Klassen direkt auf die öffentlichen Attribute zugreifen. Schnell kommt der Wunsch auf, die Adressinformationen in eine eigenständige Klasse `Address` auszulagern, um eine bessere Trennung von Zuständigkeiten zu erreichen und die Kohäsion der einzelnen Klassen zu vergrößern. Weil andere Klassen bisher direkt auf die jetzt auszulagernden Attribute zugegriffen haben, gestalten sich selbst Änderungen an den Implementierungsdetails, hier das Herauslösen einer Klasse `Address`, als schwierig. Es sind

nämlich als Folge (massive) Änderungen in allen einsetzenden Klassen notwendig. *Voraussetzung für weniger Aufwand bei Änderungen ist demnach, dass der Zugriff auf Attribute gekapselt erfolgt und die Attribute möglichst `private` deklariert sind.*

Wäre die ursprüngliche Klasse analog zur Klasse `PersonWithEncapsulation` mit Kapselung realisiert worden, so wäre ein solcher Auslagerungsschritt wesentlich einfacher möglich. Es wäre lediglich eine Klasse `Address` zu definieren und alle zugehörigen Methodenaufrufe aus der Klasse `Person` an diese weiter zu delegieren. Die Klasse `PersonWithEncapsulationAndAssociation` setzt dies um und behält das nach außen veröffentlichte API bei. Daher müssen keine Anpassungen in anderen Klassen erfolgen.

Anhand dieses Beispiels wird klar, dass die Datenkapselung sehr wichtig ist. Anpassungen bleiben dann meistens lokal auf eine Klasse begrenzt, ohne Änderungen in anderen Klassen zu verursachen.

Achtung: API-Design – Einfluss von Kapselung und Kopplung

Kapselung und Kopplung haben einen großen Einfluss auf den Entwurf von APIs:

- Findet keine Datenkapselung statt, so sind alle öffentlichen Attribute nach außen für andere Klassen sichtbar und damit Bestandteil des APIs.
- Trotz Datenkapselung wird in Zugriffsmethoden der Rückgabedatentyp und damit häufig auch der Datentyp des korrespondierenden Attributs veröffentlicht. Im Idealfall handelt es sich jedoch nur um einen primitiven Typ oder ein Interface. Dann gibt es keine Abhängigkeiten von den Implementierungsdetails der realisierenden Klasse. Zur Trennung der internen Repräsentation von dem im API sichtbaren Datentyp kann ein spezieller Datentyp als Rückgabewert definiert werden, z. B. ein sogenanntes VALUE OBJECT (vgl. Abschnitt 3.4.6).

Es kann zu Problemen führen, wenn sich Änderungen an internen Daten im API bemerkbar machen und Änderungen in nutzenden Klassen erforderlich sind. Um in solchen Fällen überhaupt Modifikationen durchführen zu können, muss man alle Anwender der eigenen Klasse kennen und dafür sorgen, dass die notwendigen Folgeanpassungen in den benutzenden Programmen durchgeführt werden. Wird eine Klasse jedoch von diversen externen Klassen verwendet, die man nicht im Zugriff hat, so kann man nachträglich Änderungen zur Kapselung und Verbesserung des Designs nicht mehr ohne Folgen durchführen: Einige Programme lassen sich dann nicht mehr kompilieren.

Diese kurze Diskussion sensibilisiert für die Probleme, ein gutes API zu entwerfen. Wie schwer dies trotz hoher Kompetenz sein kann, stellt man am JDK mit den vielen als veraltet markierten Methoden fest. Dies kann über den Javadoc-Kommentar `@deprecated`, die Annotation `@Deprecated` oder besser sogar durch beides geschehen. Derartige Methoden sollten in neu erstelltem Sourcecode nicht mehr verwendet werden.

Vererbung und Wiederverwendbarkeit

Mithilfe von Vererbung kann durch den Einsatz von Basisklassen und das Nutzen gemeinsamer Funktionalität ein übersichtlicheres Design erreicht und so mehrfach vorhandener Sourcecode vermieden werden. Allerdings sollte man Vererbung auch immer mit einer gewissen Vorsicht einsetzen. Sie ist zwar ein Mittel, um bereits modelliertes Verhalten vorhandener Klassen ohne Copy-Paste-Ansatz wiederzuverwenden und zu erweitern, aber nicht mit dem alleinigen Ziel, keinen Sourcecode zu duplizieren. Eine Vererbung, die lediglich dem Zweck dient, bereits existierende Funktionalität aus bestehenden Klassen zu verwenden, führt fast immer zu unsinnigen oder zumindest zweifelhaften Designs und wird **Implementierungsvererbung** genannt. Die entstehenden Klassen widersprechen in der Regel dem Konzept des Subtyps. Gemeinsam verwendbare Funktionalität sollte besser in eine separate Hilfsklasse ausgelagert und dann per **Delegation**⁸ statt Vererbung angesprochen werden.

Wird Vererbung nur eingesetzt, wenn tatsächlich die »is-a«-**Beziehung** erfüllt ist, so profitiert man davon, dass beim Aufbau einer **Klassenhierarchie** durch Ableitung lediglich die Unterschiede zum vererbten Verhalten und Zustand definiert werden müssen. Erweiterungen werden durch neue Methoden und Attribute realisiert. Für Änderungen am bestehenden Verhalten müssen die abgeleiteten Klassen in der Lage sein, Methoden der Basisklasse zu verändern. Dies wird durch die Technik **Overriding** erreicht. Ganz wichtig ist die Abgrenzung zum sogenannten **Overloading**. Damit wird die Möglichkeit beschrieben, mehrere Methoden mit gleichem Namen, aber unterschiedlicher Signatur innerhalb einer Klasse zu definieren. Beides lernen wir im folgenden Abschnitt genauer kennen.

Overriding und Overloading

Die beiden Techniken Overriding und Overloading werden gelegentlich selbst von erfahrenen Entwicklern nicht immer richtig eingesetzt. Deshalb möchte ich beide Techniken etwas detaillierter beschreiben.

Overriding Mit Überschreiben bzw. **Overriding** ist das **Redefinieren von geerbten Methoden** gemeint. Dazu wird der Methodename übernommen, aber eine neue Implementierung der Methode bereitgestellt und die der Basisklasse ersetzt. Dadurch können Subklassen Methoden modifizieren oder sogar komplett anders definieren, um Änderungen im Verhalten auszudrücken. Dabei darf weder die Signatur der Methode geändert noch die Sichtbarkeit eingeschränkt werden. Eine Erweiterung der Sichtbarkeit ist dagegen erlaubt.

Damit tatsächlich eine überschriebene Methode ausgeführt wird, erfolgt die Methodenauswahl beim Overriding durch dynamisches Binden.

⁸Delegation beschreibt, dass eine Aufgabe einem anderen Programmteil übergeben wird. In diesem Fall ist damit gemeint, dass eine Klasse eine andere über eine Assoziation kennt und deren Methoden aufruft.

Overloading Unter Überladen oder *Overloading* versteht man die Definition von Methoden gleichen Namens, aber mit unterschiedlicher Parameterliste. Im Gegensatz zum Overriding besteht kein Zusammenhang mit Vererbung, sondern es wird eine *Ver-einfachung der Schreibweise* adressiert. Durch Overloading kann man den Namen von Methoden ähnlicher Intention vereinheitlichen. Betrachten wir als Beispiel grafische Figuren. Eine Klasse `Rectangle` könnte etwa folgende Methoden anbieten:

- `drawByStartAndEndPos(int x1, int y1, int x2, int y2)`
- `drawByStartPosAndSize(Point pos, Dimension size)`

Die Handhabung der Klasse wird vereinfacht, wenn man als Anwender nur einen Methodennamen zum Zeichnen kennen muss. In diesem Fall könnte man folglich zwei `draw()`-Methoden mit unterschiedlicher Parameterliste anbieten. Die Wahl der entsprechenden Methode erfolgt durch die JVM automatisch aufgrund der Typen der übergebenen Parameter.

Der Einsatz der Technik Overloading ermöglicht, API-Inkonsistenzen zu beseitigen. Arbeiten mehrere Entwickler an einer Klasse, so kann es leicht zu Unterschieden in der Benennung von Methoden kommen. Beispielsweise nennt ein Entwickler seine Methode `drawRect()` und ein Kollege verwendet stattdessen das Präfix `paint` für seine Methoden. Schöner und einfacher wäre es für Klienten dieser Klasse, zum Zeichnen aller Figuren einen einheitlichen Namen beim Methodenaufruf verwenden zu können. Indem man für diese Methoden entweder `draw` oder `paint` als Präfix nutzt, erreicht man eine Namenskonsistenz.

Dem Overloading sind allerdings Grenzen gesetzt. Die Forderung nach einer unterschiedlichen Parameterliste macht es unmöglich, folgende Methoden durch Überladen gleich zu benennen:

```
drawByStartAndEndPos(int x1, int y1, int x2, int y2)
drawByStartPosAndSize(int x1, int y1, int width, int height)
```

Der Grund ist einfach: Dem Compiler wäre es nicht möglich, die gewünschte Implementierung bei gleichem Methodennamen nur anhand der Parameterliste zu wählen.

3.1.4 Wissenswertes zum Objektzustand

Nachdem wir in den vorangegangenen Abschnitten bereits einiges über den objektorientierten Entwurf erfahren haben, möchte ich hier das Thema Objektzustand vertiefen.

Bereits bekannt ist, dass man unter dem momentanen Objektzustand die aktuelle Belegung der Attribute eines Objekts versteht. Die Menge der Objektzustände eines konkreten Objekts umfasst sämtliche möglichen Belegungen von dessen Attributen. Die gültigen Zustände beschränken sich meistens auf eine (kleine) Teilmenge daraus. Ausgehend von einem oder mehreren Startzuständen sollten nur solche Zustandsübergänge möglich sein, die wiederum zu einem gültigen Objektzustand führen. Dieser Wunsch wird in Abbildung 3-9 verdeutlicht. Dort sind einige gültige Zustände und Zu-

standsübergänge visualisiert. Der deutlich größere Zustandsraum ist als graues Rechteck dargestellt. Die damit beschriebenen Zustände sind alle möglich, aber nicht jeder verkörpert einen sinnvollen Objektzustand; einige davon sind explizit als *Invalid 1*, *Invalid 2* bzw. *Invalid n* dargestellt. Bei einem *Person*-Objekt könnten dies etwa negative Werte für *Alter* oder *Größe* sein.

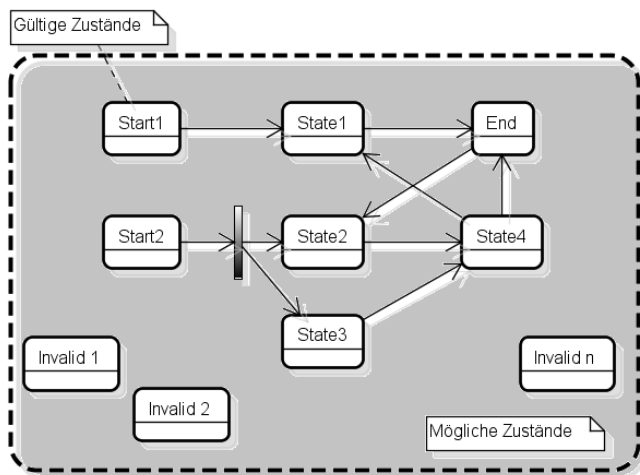


Abbildung 3-9 Gültige Objektzustände und Zustandsraum

Die Forderung nach gültigen Objektzuständen ist gar nicht so leicht zu erfüllen. Tatsächlich wird dies in vielen Programmen vernachlässigt, und der erlaubte Zustandsraum wird nicht geprüft oder ist nicht einmal festgelegt. Dadurch besitzen Objekte zum Teil beliebige, wahrscheinlich auch ungültige Zustände. Dies löst häufig unerwartete Programmreaktionen aus und führt normalerweise zu Berechnungsfehlern oder Abstürzen. Um dies zu verhindern, müssen wir ein besonderes Augenmerk auf all diejenigen Methoden werfen, die Änderungen am Objektzustand auslösen. Eine Voraussetzung ist es, einen gültigen Ausgangszustand unter anderem durch eine Prüfung von Eingabeparametern sicherzustellen. Ungültige Parameterwerte müssen zurückgewiesen werden, um eine sichere Ausführung der Methode zu gewährleisten. Zudem müssen durchgeführte Berechnungen wiederum gültige Objektzustände erzeugen.

Der Objektzustand am Beispiel

Stellen wir uns einen grafischen Editor vor, der eine Ausrichtung von grafischen Figuren an einem vordefinierten Raster erlaubt. Nehmen wir an, es soll ein Raster mit einem Abstand von 10 Punkten modelliert werden. Wir entwerfen dazu eine Klasse `GridPosition`, die ihre Koordinaten in den zwei Attributen `x` und `y` vom Typ `int` speichert. Gültige Punkte und damit Belegungen der Attribute sollen immer auf den genannten Rasterpunkten liegen. Die Wertebelegungen der Attribute vom Typ `int` beschreiben den Zustandsraum, d. h. alle potenziell möglichen Objektzustände. Die Men-

ge der erlaubten Zustände ist jedoch wesentlich geringer. Das gilt im Speziellen für dieses Beispiel und lässt sich auf viele Fälle in der Praxis übertragen.

Die Klasse `GridPosition` bietet zwei Business-Methoden, die Objektverhalten beschreiben – dies sind `addOffset(int, int)` und `setSamePos(int)`. Zur Demonstration von Inkonsistenzen im Objektzustand sind öffentliche `set()`-Methoden für die Attribute `x` und `y` definiert.

```
public final class GridPosition
{
    private static final int GRID_SIZE = 10;

    // Invariante: x,y liegen immer auf einem Raster der Größe 10
    private int x = 0;
    private int y = 0;

    GridPosition()
    {
    }

    public void addOffset(final int dx, final int dy)
    {
        // Vorbedingung: x, y auf einem beliebigen Rasterpunkt
        checkOnGrid(x,y);

        x += snapToGrid(dx);
        y += snapToGrid(dy);

        // Nachbedingung: x + (dy % 10), y + (dy % 10) auf einem Rasterpunkt
        checkOnGrid(x,y);
    }

    public void setSamePos(final int pos)
    {
        // Vorbedingung: x, y auf einem beliebigen Rasterpunkt
        checkOnGrid(x,y);

        x = snapToGrid(pos);
        y = snapToGrid(pos);

        // Nachbedingung: x = y = (pos % 10) auf einem Rasterpunkt
        checkOnGrid(x,y);
    }

    private static void checkOnGrid(final int x, final int y)
    {
        if (x % GRID_SIZE != 0 || y % GRID_SIZE != 0)
            throw new IllegalStateException("invalid position, not on grid");
    }

    private int snapToGrid(final int value)
    {
        return value - value % GRID_SIZE;
    }

    public int getX()           { return x; }
    public int getY()          { return y; }

    // Zur Demonstration von Problemen
    public void setX(final int x) { this.x = x; }
    public void setY(final int y) { this.y = y; }
}
```

Invarianten, Vor- und Nachbedingungen

Im obigen Beispiel haben wir sogenannte Vor- und Nachbedingungen sowie Invarianten eingesetzt. Diese spielen bei der Programmverifikation und dem Beweis der Korrektheit von Programmen in der theoretischen Informatik eine wichtige Rolle. Wie gesehen, kann man in der Praxis auch ohne Kenntnis der Details von deren Formulierung profitieren. Man nutzt eine Absicherung mit Exceptions (vgl. Abschnitt 4.7), um ungültige Objektzustände zu vermeiden.

Mithilfe einer *Invariante* kann man Aussagen über Teile des Objektzustands treffen. Damit beschreibt man *unveränderliche* Bedingungen, die vor, während und nach der Ausführung verschiedener Programmanweisungen gültig sein sollen. Ein Beispiel für eine solche Forderung ist, dass das Alter einer Person nicht negativ sein darf.

Über sogenannte *Vorbedingungen* lässt sich eine zur Ausführung einer Methode notwendige Ausgangssituation beschreiben: Die Vorbedingung einer Suchmethode könnte etwa sein, dass Elemente vorhanden und sortiert sind. Wenn diese Vorbedingung erfüllt ist, so wird nach Ausführung der Methode als Ergebnis die Position des gesuchten Elements zurückgeliefert, oder aber -1, wenn kein solches existiert. Bedingungen nach Ausführung eines Programmstücks werden durch sogenannte *Nachbedingungen* formuliert. Im Beispiel haben wir dies genutzt, um sicherzustellen, dass die Koordinaten weiterhin einem Rasterpunkt entsprechen.

Info: Design by Contract

Der Einsatz von Invarianten, Vor- und Nachbedingungen erinnert an die Forderungen des von Bertrand Meyer beim Entwurf der Programmiersprache Eiffel erfundenen »*Design by Contract*«. Idee dabei ist es, Vor- und Nachbedingungen bei jedem Methodenaufwurf abzusichern. Dies soll verhindern, dass eine Methode mit ungültigen Werten aufgerufen wird. Auf diese Weise versucht man, die Konsistenz des Programmzustands sicherzustellen sowie ein mögliches Fehlverhalten des Programms auszuschließen. Beim Verlassen einer Methode wird durch Nachbedingungen sichergestellt, dass nur erwartete, gültige Werte als Rückgabe möglich sind. Weitere Informationen finden Sie unter <http://archive.eiffel.com/doc/manuals/technology/contract/page.html>.

Veränderungen am Objektzustand

Nach der Konstruktion eines `GridPosition`-Objekts entspricht der Startzustand der Wertebelegung $x = y = 0$. Wie bereits erwähnt, sollten öffentliche Methoden ein Objekt aus einem gültigen Objektzustand in einen anderen gültigen versetzen. Betrachten wir dies für die beiden Business-Methoden:

- Jeder Aufruf von `addOffset(dx, dy)` führt zu einem Zustandswechsel:
 $x = x + dx, y = y + dy$.
- Jeder Aufruf von `setSamePos(pos)` wechselt in den Zustand $x = y = pos$.

Um sicherzustellen, dass wirklich nur Rasterpunkte erreicht werden können, wird durch Aufruf der Methode `snapToGrid(int)` eine Modulo-Korrektur vorgenommen.

In der Klasse `GridPosition` werden jedoch zwei öffentliche `set()`-Methoden angeboten, in denen keine Korrektur der Eingabewerte stattfindet. Rufen andere Klassen diese `set()`-Methoden statt der Business-Methoden auf, so können beliebige Werte für die Attribute `x` und `y` gesetzt werden. Diese liegen höchstwahrscheinlich nicht auf dem gewünschten Raster und stellen somit auch keinen gültigen Objektzustand dar.

Tipp: Probleme durch öffentliche `set()`-Methoden

Häufig findet man in der Praxis eine Vielzahl öffentlicher `set()`-Methoden, über die der Objektzustand auf unkontrollierte Weise geändert wird. Dadurch kann ein gültiger Objektzustand nicht mehr gewährleistet werden.

Daher sollten `set()`-Methoden in ihrer Sichtbarkeit eingeschränkt oder – wie in diesem Beispiel leicht möglich – komplett entfernt werden. Zuvor kann die Funktionalität durch Business-Methoden bereitgestellt werden. Eine Anleitung dazu liefert das Refactoring ERSETZE MUTATOR- DURCH BUSINESS-METHODE in Abschnitt 11.2.4. In dieser Klasse wurden bereits Business-Methoden bereitgestellt, die `set()`-Methoden jedoch nicht entfernt.

Weitere Komplexitäten durch Multithreading

Methodenaufrufe werden im Bytecode nicht als Ganzes abgearbeitet, sondern in Form vieler kleiner Zwischenschritte. Dadurch kommt es immer zu Zwischenzuständen im Objektzustand. Abbildung 3-10 zeigt dies für den Aufruf `addOffset(50, 30)`.

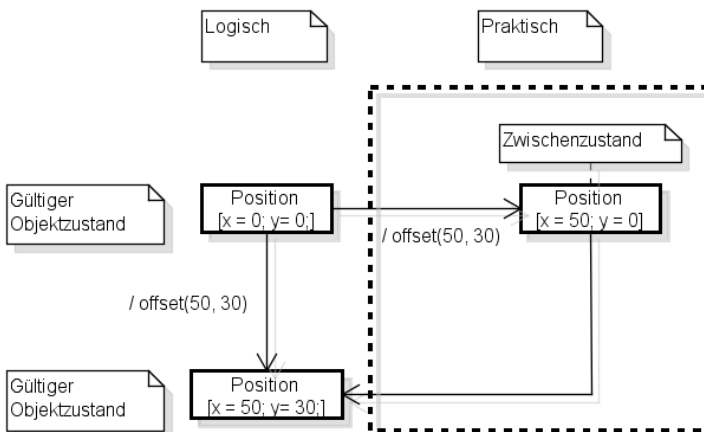


Abbildung 3-10 Objektzustandsübergänge

Diese Zwischenzustände lassen sich systembedingt nicht verhindern. Es muss allerdings dafür gesorgt werden, dass nur das Objekt selbst diese (möglicherweise ungültigen) Zwischenzustände sieht und sich nach der Ausführung einer Methode wieder in einem gültigen Objektzustand befindet. Einen wichtigen Beitrag dazu leisten sowohl Datenkapselung als auch die Definition von kritischen Bereichen.

Datenkapselung Mithilfe von Datenkapselung kann man vermeiden, dass andere Objekte jederzeit einen Blick in die »Innereien« eines anderen Objekts werfen können, und somit auch verhindern, dass möglicherweise mit ungültigen Zwischenzuständen weitergearbeitet wird. Im ungünstigsten Fall sind diese durch externe, verändernde Zugriffe verursacht. Bei Datenkapselung verhindern die `private` definierten Attribute Derartiges.

Kritische Bereiche Ein wichtiger Faktor zum Schutz vor unerwünschten Zwischenzuständen ist, Unterbrechungen von Berechnungen bei Multithreading zu vermeiden. Ein aktiver Thread kann nahezu jederzeit in seiner Abarbeitung unterbrochen werden. Anschließend wird ein anderer Thread aktiviert. Geschieht ein solcher Thread-Wechsel während der Abarbeitung einer der Methoden `addOffset(int, int)` bzw. `setSamePos(int)`, kann dies problematisch sein: Sind zum Zeitpunkt des Thread-Wechsels noch nicht alle Zuweisungen an die Attribute erfolgt, so sieht ein anderer Thread beim Aufruf einer `get()`-Methode einen Zwischenzustand. Eine derartige Unterbrechung kann durch die Definition eines kritischen Bereichs verhindert werden. Kapitel 7 behandelt dies ausführlich.

Objektzustände während der Objektkonstruktion

Manchmal kommt es während der Objektkonstruktion zu ungewünschten, vermeidbaren oder sogar ungültigen Zwischenzuständen. Dies gilt insbesondere, wenn lediglich ein Defaultkonstruktor und diverse `set()`-Methoden angeboten werden.

Die Grundlage für die Diskussion bildet folgende Klasse `SimpleImage`. Sie verwendet einen Namen, eine Breite und eine Höhe sowie die eigentlichen Bilddaten und speichert diese Werte in vier privaten Attributen. Der Zugriff erfolgt über öffentliche `get()`- und `set()`-Methoden – hier nur für das Attribut `name` gezeigt:

```
public class SimpleImage
{
    private String name;
    private int width;
    private int height;
    private byte[] imageData;

    public SimpleImage()
    {}

    public void setName(final String name)
    {
        this.name = name;
    }
}
```

```
public String getName()
{
    return this.name;
}

// weitere Getter und Setter ...
}
```

Das von der Klasse angebotene API erfordert zur Herstellung eines gültigen Objektzustands nach der Objektkonstruktion, dass alle Attribute durch Aufruf von `set()`-Methoden sukzessive initialisiert werden. Betrachten wir dies anhand des Einlesens aus einer Datei in der folgenden Methode `createSimpleImageFromFile()`:

```
public static SimpleImage createSimpleImageFromFile() throws IOException
{
    final SimpleImage simpleImage = new SimpleImage();

    // ACHTUNG: Nur zur Demonstration Beobachter zu früh registrieren
    registerObserver(simpleImage);

    final String imageName = readNameFromFile();
    simpleImage.setName(imageName); // Name setzen

    try
    {
        final int imageWidth = readWidthFromFile();
        simpleImage.setWidth(imageWidth); // Breite setzen

        final int imageHeight = readHeightFromFile();
        simpleImage.setHeight(imageHeight); // Höhe setzen

    }
    catch (final NumberFormatException e)
    {
        // ACHTUNG: Keine Fehlerbehandlung zur Demonstration
    }

    final String imageData = readImageDataFromFile();
    simpleImage.setImageData(imageDate); // Daten setzen

    return simpleImage;
}
```

Ähnliche Programmausschnitte trifft man in der Praxis immer wieder an. Was daran problematisch ist, wollen wir nun analysieren:

1. Es werden ungültige Objektzustände sichtbar. Im Beispiel wird dies dadurch ausgelöst, dass ein Beobachter zu früh – vor Abschluss der Objektkonstruktion und der Initialisierung – angemeldet wird. Im Allgemeinen werden ungültige Objektzustände sichtbar, wenn man eine noch nicht vollständig initialisierte Referenz an andere Komponenten übergibt.
2. Treten Fehler beim Einlesen aus der Datei oder beim Verarbeiten der eingelesenen Daten auf, so verbleibt das Objekt in einem teilinitialisierten Objektzustand, der höchstwahrscheinlich keinem gültigen Objektzustand entspricht.

3. Es müssen viele (unnötige) Zugriffsmethoden für Attribute bereitgestellt werden. Dies verhindert eine Realisierung, die lediglich in den wirklich notwendigen Teilen veränderlich ist.
4. Öffentlich definierte Zugriffsmethoden erlauben beliebige Änderungen durch andere Objekte. Daraus können inkonsistente Objektzustände resultieren.

Analysieren wir die Punkte Schritt für Schritt:

Ungültige Objektzustände werden sichtbar Im gezeigten Listing wird zunächst ein `SimpleImage`-Objekt über einen Defaultkonstruktor erzeugt. Unter der Annahme gültiger Eingabedaten werden sukzessive vier `set()`-Methoden aufgerufen und führen demnach auch zu vier Änderungen im Objektzustand. Wenn sich nach dem Konstruktoraufruf bereits ein Beobachter (vgl. Abschnitt 12.3.8 zum Entwurfsmuster `BEOBSACHTER`) beim Objekt registriert hat, so erhält dieser vier Änderungsbenachrichtigungen in kurzer Abfolge nacheinander. Allerdings wird erst durch die letzte Mitteilung ein gültiger Objektzustand sichtbar.

Teilinitialisierter Objektzustand möglich Nehmen wir an, einer der eingelesenen Werte für Breite und Höhe wäre ungültig. Der leere `catch`-Block verhindert, dass dieser Fehler zum Aufrufer propagiert wird. Stattdessen kommt es zu einem unbemerkten Fehler. Das neu konstruierte `SimpleImage`-Objekt besitzt nach dem Einlesen der korrekten Werte von Name und Bilddaten die falschen Werte 0 für Breite und Höhe. Dies stellt einen inkonsistenten Objektzustand dar.

Unnötige Zugriffsmethoden Alle Attribute der Klasse `SimpleImage` sind über `set()`-Methoden veränderbar. Diese Methoden wurden nur eingeführt, um ein sukzessives Setzen der Werte zu ermöglichen. Es ist wahrscheinlich, dass Höhe, Breite und Bilddaten voneinander abhängig sind und nur miteinander als Einheit zu verändern sein sollten. Statt einzelner `set()`-Methoden sollte man daher eine Business-Methode `changeImageData(int, int, byte[])` anbieten. Zudem soll der Name des Bildes unveränderlich sein. Dies wird durch die Deklaration des Attributs `name` als `final` erreicht:

```
public class SimpleImage
{
    private final String name;
    private int width;
    private int height;
    private byte[] imageData;

    public SimpleImage(final String name, final int width, final int height,
                      final byte[] imageData)
    {
        this.name = name;
        changeImageData(width, height, imageData);
    }
}
```

```
public void changeImageData(final int width, final int height,
                           final byte[] imageData)
{
    this.width = width;
    this.height = height;
    this.imageData = imageData;
}

public String getName()
{
    return this.name;
}

// weitere get()-Methoden ...
}
```

Die genannten Änderungen machen die Klasse besser lesbar. Wichtiger ist aber, dass die Abhängigkeiten der Attribute untereinander nun deutlich zu erkennen sind und nur durch eine Methode in der öffentlichen Schnittstelle beschrieben werden. Diese Business-Methode ist auch die richtige Stelle, um Änderungen an mögliche Beobachter zu kommunizieren.

Betrachten wir, wie sich das neue API der Klasse `SimpleImage` im Einsatz auswirkt. Statt der direkten `set()`-Aufrufe erfolgt zunächst eine Zwischenspeicherung in lokalen Variablen. Erst nachdem alle Daten erfolgreich eingelesen werden konnten, wird eine Objektkonstruktion ausgeführt. Die Anmeldung des Beobachters ist nun problemlos – es werden nur gültige Zwischenzustände nach außen sichtbar. Selbst die fehlende Behandlung eines Parsing-Fehlers hat bei dieser Art der Realisierung deutlich weniger negative Auswirkungen. Es wird dann einfach kein Objekt erzeugt. Folgendes Listing zeigt das neue API im Einsatz:

```
public SimpleImage createSimpleImageFromFile() throws IOException
{
    final String imageName = readNameFromFile();

    try
    {
        final int imageWidth = readWidthFromFile();
        final int imageHeight = readHeightFromFile();
        final String imageData = readImageDataFromFile();

        final SimpleImage simpleImage = new SimpleImage(imageName, imageWidth,
                                                         imageHeight, imageData.getBytes());
        registerObserver(simpleImage);
    }
    catch(final NumberFormatException ex)
    {
        log.error("failed to create SimpleImage object from file", ex);
    }

    return simpleImage;
}
```

Öffentliche Zugriffsmethoden Alle Attribute der eingangs gezeigten Klasse `SimpleImage` waren über öffentliche `set()`-Methoden veränderlich. Damit konnten Klienten⁹ beliebige Änderungen am Objektzustand durchführen. Im vorherigen Schritt haben wir bereits die Anzahl öffentlicher, den Objektzustand beeinflussender Methoden verringert. Als Folge konnten sämtliche `set()`-Methoden entfallen. Eine Zustandsänderung ist nur noch durch Aufruf der Methode `changeImageData(int, int, byte[])` möglich. Erfolgt innerhalb dieser Methode eine Parameter- und Konsistenzprüfung, so kann das Objekt von anderen Klassen nicht mehr in einen ungültigen Objektzustand versetzt werden. Dies gilt jedoch nur für Singlethreading. Für Multithreading müssten dazu in diesem Fall noch alle Methoden synchronisiert werden, um Inkonsistenzen sicher auszuschließen.

3.2 Grundlegende OO-Techniken

Bis hierher haben wir bereits einige wichtige Einblicke in den objektorientierten Entwurf bekommen. Dieser Abschnitt geht nochmals etwas genauer auf drei grundlegende OO-Techniken ein, die wir für den Entwurf größerer Softwaresysteme als Basisbausteine benötigen. Sie helfen dabei, Funktionalität zu kapseln und Subsysteme voneinander unabhängig zu halten.

Zunächst betrachten wir in Abschnitt 3.2.1 abstrakte Basisklassen, die beim Entwurf entstehen, wenn man gemeinsame Funktionalität aus Subklassen in einer eigenständigen Basisklasse zusammenführt. Zur Beschreibung der Funktionalität einer Klasse kann man sich im einfachsten Fall die öffentlichen Methoden anschauen. Sinnvoller ist es jedoch, diese Methoden zu gruppieren und diesen Funktionsblöcken einen eigenen Namen zu geben. Dies ist durch die Technik der Interfaces möglich. Details dazu beschreibt Abschnitt 3.2.2. Häufig ist eine Kombination der beiden genannten Techniken sinnvoll und vereinfacht den Entwurf weiter. Abschnitt 3.2.3 erläutert das genauer.

3.2.1 Abstrakte Basisklassen

Mithilfe von Basisklassen kann von Subklassen gemeinsam genutzte Funktionalität zentral definiert werden. Ohne eine Basisklasse wäre die Funktionalität ansonsten jeweils mehrfach in den Subklassen zu implementieren. Manchmal entstehen durch die Sammlung von Funktionalität im Rahmen einer Generalisierung aber auch Basisklassen, die für sich betrachtet ein so allgemeines Konzept realisieren, dass nicht alle Bestandteile des Objektverhaltens sinnvoll definiert werden können. Es werden dort bewusst Modellierungsdetails offengelassen, die von abgeleiteten Klassen realisiert werden müssen. In diesem Fall spricht man von **abstrakten Basisklassen**. Ist eine Klasse

⁹Unter Klienten versteht man beliebige andere Klassen, die Methoden der eigenen Klasse aufrufen.

vollständig implementiert, so spricht man von einer *konkreten Klasse* bzw. einer *konkreten Realisierung*.¹⁰

Am Beispiel von grafischen Figuren lässt sich die zugrunde liegende Idee gut verdeutlichen. Stellen wir uns eine Applikation vor, die Figuren zeichnen soll. Diese Figuren werden jeweils als eigenständige Klassen modelliert. Dort werden Methoden zum Zeichnen definiert. Nehmen wir an, dass im Laufe der Zeit durch Unachtsamkeit verschiedene Signaturen der jeweiligen Methode zum Zeichnen entstanden wären, etwa `drawLine()`, `drawRect()` usw. Abbildung 3-11 zeigt dies für einige Figurenklassen.

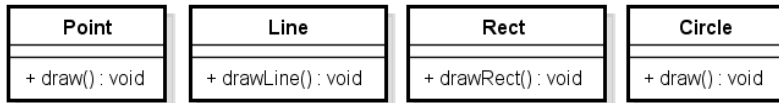


Abbildung 3-11 Klassen für grafische Figuren (ohne Basisklasse)

Die Handhabung ist für Klienten durch die unterschiedlichen Signaturen und durch das Fehlen einer gemeinsamen Basisklasse umständlich und nicht intuitiv. Um diesen Missstand zu beheben, führen wir eine Basisklasse `Figure` ein und sorgen mit einer `draw()`-Methode für konsistente Methodennamen. In diesem Fall kann die Basisklasse `Figure` nichts Sinnvolles zeichnen. Daher implementieren wir die `draw()`-Methode dort zunächst leer. Eine solche Leerimplementierung kann hilfreich sein, wenn diese Methode nicht zwangsläufig von Subklassen mit spezifischer Funktionalität versehen werden muss. In diesem Beispiel stellt die `draw()`-Methode jedoch Kernfunktionalität dar, die auf dieser Abstraktionsebene noch ohne konkrete Realisierung ist. Die Methode wird daher `abstract` deklariert und besitzt damit keine Implementierung. Die Klasse wird dann zu einer abstrakten Klasse. Die Implementierung der `draw()`-Methode wird dadurch den konkreten Subklassen übertragen. Dies ist gewünscht, da nur diese wissen, wie die konkreten Figuren gezeichnet werden sollen. Weiterhin werden die zuvor uneinheitlichen Methodennamen nun alle konsistent zu `draw()`. Abbildung 3-12 zeigt das resultierende Klassendiagramm.

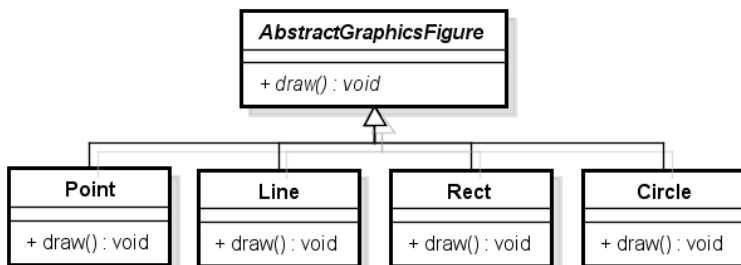


Abbildung 3-12 Grafische Figuren mit abstrakter Basisklasse

¹⁰Trotz vollständiger Implementierung kann eine Klasse explizit über das Schlüsselwort `abstract` als abstrakt gekennzeichnet werden.

Bewertung

Abstrakte Basisklassen ...

- + helfen bei einer konsistenten Implementierung von Funktionalitäten. Subklassen bringen nur an speziellen, gewünschten Stellen ihre Realisierungen ein.
- + vermeiden Sourcecode-Duplikation und dadurch bedingten Wartungsaufwand.
- + erleichtern das Erzeugen neuer Subklassen, da in der Regel nur wenige Anpassungen erfolgen müssen.
- haben den Nachteil, dass sie dazu verleiten, Vererbung unüberlegt einzusetzen und dadurch eine tiefe Klassenhierarchie zu erzeugen. Häufig ist Ableitung jedoch nicht der richtige Designweg und sollte nur eingesetzt werden, um eine »is-a«-Beziehung auszudrücken (vgl. Abschnitt 3.4.3).

3.2.2 Schnittstellen (Interfaces)

Schnittstellen, häufig auch Interfaces genannt, kann man sich als Vertrag zwischen aufrufender und bereitstellender Klasse vorstellen. Mithilfe von Interfaces wird festgelegt, welche Methoden eine Klasse implementieren muss, die dieses Interface realisiert. Die Technik der Interfaces ermöglicht eine Entkopplung von Realisierung und Spezifikation. Das ist die Basis für viele Entwurfsmuster, wie ITERATOR, DEKORIERER, NULL-OBJEKT und viele weitere (vgl. Kapitel 12).

Durch ein Interface wird keine Realisierung vorgegeben, daher müssen die Methodennamen besonders sprechend und klar gewählt werden. Außerdem ist eine Dokumentation zwingend notwendig, damit alle Implementierer die gleiche Vorstellung von der auszuführenden Aktion haben. Das Implementieren eines Interface ist wie die Vererbung eine Technik, um gewisse Funktionalität bereitzustellen. Hier gilt die »*can-act-like*«-Beziehung. Der Schwerpunkt liegt hier mehr auf dem Ausüben einer speziellen Funktionalität, als dies bei der »is-a«-Beziehung der Vererbung der Fall ist. Zudem können Klassen beliebig viele Interfaces implementieren. Dies erleichtert es, logisch zusammengehörende Methoden in eigenen Interfaces zu gruppieren. Klassen können so mehrere voneinander unabhängige Funktionalitäten oder Verhaltensweisen gemäß »can-act-like« ausüben. Es wird demnach keine Spezialisierung definiert, wie dies durch die »is-a«-Beziehung geschieht.

Namensgebung von Interfaces

Interfaces sollten als solche auch durch ihren Namen erkennbar sein. Im JDK wird dazu die Endung `-able` verwendet, um die »can-act-like«-Beziehung zu betonen. In diesem Buch werden stattdessen das Präfix `I` bzw. alternativ das Postfix `IF` verwendet, um die Interface-Eigenschaft zu betonen.

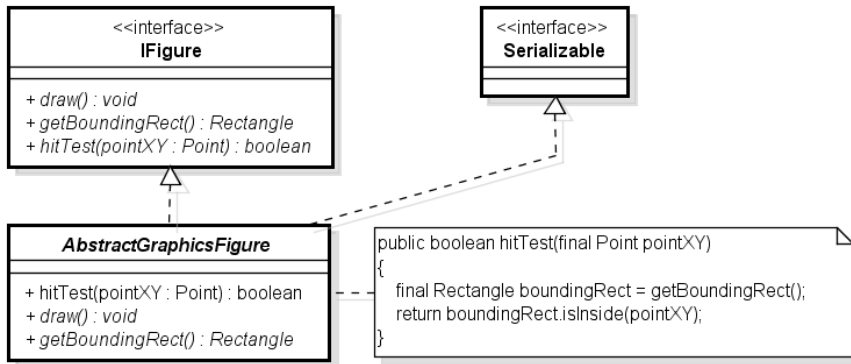


Abbildung 3-13 Zwei Interfaces und eine abstrakte Klasse

Bewertung

Der Einsatz von Interfaces ...

- + entkoppelt Spezifikation und Realisierung: Dadurch kennen sich nutzende und Funktionalität bereitstellende Klassen nicht. Nutzende Klassen dürfen sich daher nicht auf spezielle Implementierungen oder Implementierungsdetails verlassen.
- + erleichtert den Austausch einer konkreten Implementierung: Ein Beispiel dafür ist das Interface `List` aus dem Collections-Framework mit seinen Realisierungen `ArrayList`, `LinkedList` und `Vector`.
- + ermöglicht die Definition der tatsächlichen Schnittstelle. In Interfaces werden die angebotenen Methoden zentral definiert, statt verstreut im Sourcecode in Form von öffentlichen Methoden aufgeführt zu sein.
 - o bietet keinen Zugriff auf den Konstruktor¹¹, da sie nur Methoden deklarieren.
 - o erfordert die Realisierung öffentlicher Methoden.

Tipp: Implizite Zugriffsmodifizier

In Java sind alle in Interfaces definierten ...

- **Methoden** immer mit den Zugriffsmodifizier `public` und `abstract` definiert.
- **Variablen** implizit als `public`, `static` und `final` definiert. Diese sind dadurch als Konstanten nach außen sichtbar.

Die obigen Zugriffsmodifizier können explizit im Sourcecode angegeben werden. Sie dürfen einzeln oder gemeinsam angegeben werden und sind unabhängig davon immer vorhanden. Alle anderen Zugriffsmodifizier sind in Interfaces nicht erlaubt.

¹¹Wobei dies in vielen Fällen eher ein Vorteil als ein Nachteil ist. Mithilfe der Entwurfsmuster `ERZUEGUNGSMETHODE` und `FABRIKMETHODE` (vgl. Abschnitte 12.1.1 und 12.1.2) kann man implementierende Klassen verstecken.

3.2.3 Interfaces und abstrakte Basisklassen

Die beiden zuvor betrachteten Techniken Interface und abstrakte Basisklasse lassen sich gewinnbringend kombinieren. Man erreicht eine lose Kopplung: Klienten werden unabhängig von Details realisierender Klassen. Außerdem können diese Realisierungen nach außen hin einheitlich behandelt werden.

Für Klassen, die aus anderen Packages verwendet werden, sollten Interfaces an der Schnittstelle des eigenen Packages bereitgestellt werden, um eine losere Kopplung zu erzielen. Zusätzlich kann eine Kombination mit einer abstrakten Basisklasse erfolgen, wenn man mehrere Varianten einer Realisierung anbieten muss oder möchte. Abbildung 3-14 zeigt diesen Fall für ein Interface `ServiceIF`, eine abstrakte Basisklasse `AbstractBaseService` sowie zwei konkrete Realisierungen.

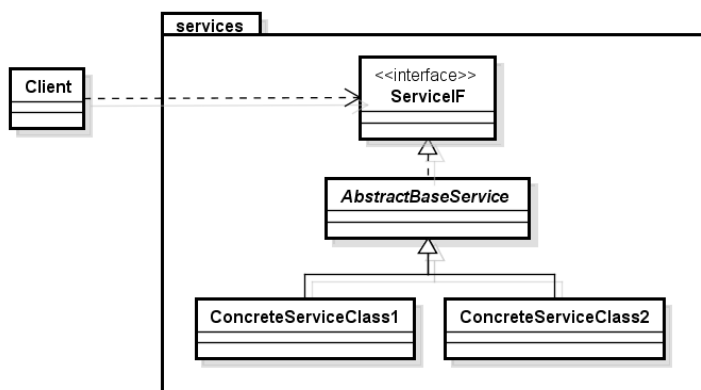


Abbildung 3-14 Kombination von Interface und abstrakter Basisklasse

Bewertung

Eine Kombination beider Techniken ...

- + ermöglicht sowohl eine Kapselung der Funktionalität über Interfaces sowie eine sinnvolle Vorgabe von Basisfunktionalität durch eine abstrakte Basisklasse.
- + ermöglicht, dass Subklassen nur noch an speziellen, gewünschten Stellen ihre Erweiterungen einbringen müssen.
- + erlaubt, die konkreten Implementierungen einfach auszutauschen.
- + bietet alle Vorteile der jeweiligen Techniken.
- o erfordert einen geringen Mehraufwand in der Implementierung.