Design Patterns

Grigory Vartanyan, Lukas Wilke

1 Einleitung

Bei der Entwicklung von Programmen gibt es einige Probleme, die immer wieder auftreten und gelöst werden müssen. Es wäre äußerst aufwendig und verwirrend, wenn jeder Programmierer eine neue, eigene Lösung zu diesen Problemen entwickelt. Damit dies nicht passiert, gibt es *Design Patterns*. Design Patterns sind im Grunde vorgefertigte Lösungskonzepte zu häufigen Problemen. Dabei handelt es sich nicht um fertige Codebausteine, sondern um grundlegende Ideen, die man an die aktuelle Situation anpassen muss. Zwei Implementationen desselben Patterns können sich stark voneinander unterscheiden. Dass sie keinen tatsächlichen Code beinhalten, hat zudem den Vorteil, dass Design Patterns von der gewählten Programmiersprache unabhängig sind und sich in fast allen objektorientierten Sprachen verwenden lassen.

Die Nutzung von Design Patterns bringt viele Vorteile mit sich: Zum einen muss man nicht, wie oben erwähnt, zu jedem Problem eine neue Lösung entwickeln, wenn es bereits Lösungen gibt, die sich als nützlich und effizient erwiesen haben. Stattdessen kann man diese Lösung einfach wiederverwenden und so unnötigen Arbeitsaufwand vermeiden. Zudem werden diese Lösungen durch Design Patterns standardisiert, was dazu führt, dass andere Entwickler, die mit dem Code arbeiten, diesen einfacher verstehen, wenn sie mit dem Pattern vertraut sind.

Dennoch muss man bei der Verwendung von Design Patterns einiges bedenken. Bevor man ein Design Pattern verwendet, muss man genau wissen, ob eines nötig ist, welches Design Pattern sinnvoll ist, und wie man es implementiert. Es gibt viele Design Patterns, die ähnliche Zwecke erfüllen, und das richtige Pattern auszuwählen ist an sich bereits eine gewisse Herausforderung.

Eine genaue Anzahl an Design Patterns festzulegen ist nahezu unmöglich. Ein Design Pattern entsteht nämlich nicht dadurch, dass ein Programmierer sich entscheidet, ein neues Pattern zu entwickeln, sondern dadurch, dass ein Problem in verschiedenen Programmen immer wieder auf dieselbe Weise gelöst wird. Diese Lösungen werden von verschiedenen Autoren dann in Nachschlagewerken gesammelt, klassifiziert und benannt, damit man sie einfacher nachschlagen kann und sich besser mit anderen Programmierern über sie austauschen kann. Die Klassifikationen sind folglich nützlich, da sie helfen, Design Patterns zu verstehen. Nach Gamma et al. gibt es zwei Hauptklassifikationsmethoden. Die erste fokussiert sich auf den *Scope* der Patterns und teilt Patterns in zwei Gruppen: *Class* und *Object*. Der Unterschied besteht darin, auf welcher Ebene die Patterns arbeiten. Sie

beziehen sich auf Klassen oder Objekte, was mit Methoden, die statisch oder nicht-statisch sein können, vergleichbar ist. Die zweite Weise, Patterns zu klassifizieren, definiert sich durch ihr Verhalten. Diese unterscheidet zwischen drei Kategorien:

1. *Creational*: Patterns, die Objekte erschaffen
2. *Structural*: Patterns, die die Struktur von Klassen beeinflussen
3. *Behavioural*: Patterns, die das Verhalten von Objekten ändern

Viele Design Patterns können auf diese Kategorien verteilt werden, deshalb können sie helfen, die Patterns zu finden, welche ein Programmierer braucht.

Diese Ausarbeitung wird vier Design Patterns aus zwei Sammelbänden beispielhaft näher beleuchten: Wir erläutern das Factory- und das Iterator-Pattern aus „Design Patterns: Elements of Reusable Object-Oriented Software“ von Erich Gamma, et al., einem der ersten Werke, das Design Patterns wirklich popularisierte, sowie das Service Layer- und das Money-Pattern aus „Patterns of Enterprise Application Architecture“ von Martin Fowler, der sich hauptsächlich mit Design Patterns für Unternehmenssoftware beschäftigte. Auch die Informationen aus dieser Einleitung stammen aus diesen Büchern.

2 Factory Method

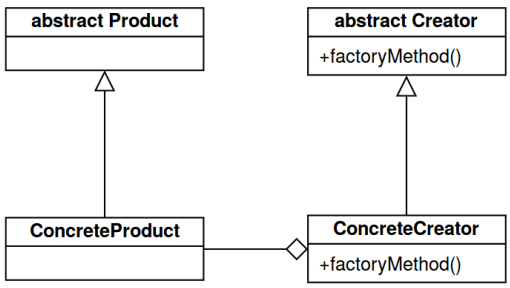

Abbildung 1: Struktur des Factory Method-Patterns

Das erste Pattern, das wir besprechen, ist die *Factory Method*. Es ist eins der einzigen Creational Patterns, die auf der Class-Ebene arbeiten. Das beschreibt auch den Zweck des Patterns: Es existiert, um Objekte zu erzeugen. Ein sehr verbreitetes Problem bei der Erzeugung von Objekten besteht darin, dass der Erzeuger (in diesem Fall der Benutzer) bestimmte Details über ein Objekt kennen muss, um es erzeugen und initialisieren zu können. Diese Informationen sind Details wie der Name der Klasse, wie man auf das Objekt zugreifen kann und welche Methoden benötigt werden, um es zu erzeugen. Das schafft eine enge Kopplung zwischen dem Benutzer und dem Programm, welche es schwierig macht, in einem Teil des Codes zu arbeiten, ohne immer Änderungen in anderen Teilen des Codes berücksichtigen zu müssen.

Die allgemeine Idee einer Factory Method ist sehr einfach: Es handelt sich im Wesentlichen um einen Zwischenhändler. Anstatt den Benutzer die Klasse des Objekts, das er konstruieren möchte, spezifizieren zu lassen, kann er eine einzige bekannte Methode aus einer speziellen Creator-Klasse benutzen, um dieses Objekt zu konstruieren. Dies ermöglicht dem Benutzer eine unbegrenzte Zahl von Objekten und Objekttypen zu erstellen, ohne alle seine jeweiligen Klassen zu kennen. Stattdessen wird dies von der Factory Method-Klasse erledigt. Die indirekte Objekterstellungstechnik ermöglicht der Factory Method auch, eine gewisse Kontrolle darüber haben, welche Objekte unter verschiedenen Umständen erstellt werden. Das heißt auch, dass die Factory Method einen Objekttyp auswählen kann, falls der Benutzer nicht weiß, welcher Objekttyp benötigt wird.

Die Struktur der Factory Method ist weniger intuitiv, als man erwarten würde: Sie benötigt nicht nur eine *Creator*-Klasse mit einer *create()*-Methode, sondern auch Unterklassen, die bei der Konstruktion der verschiedenen Objekttypen helfen. Im Allgemeinen gibt es abstrakte *Product*- und *Creator*-Klassen, wobei die Creator-Klasse eine Methode zur Konstruktion von Objekten definiert. Beide Klassen haben mehrere Unterklassen, von denen jede sich mit der Konstruktion eines Typs beschäftigt. Auf diese Weise haben alle Produkt- und Creator-Klassen eine ähnliche Struktur, aber unterschiedliche Implementierungen von create()-Methoden. Diese Methoden entscheiden, welcher Objekttyp erstellt und zurückgegeben werden soll, sowie welche Parameter und internal States die erstellten Objekte haben müssen.

Im Allgemeinen ist die Verwendung der Factory Method viel flexibler als die direkte Objekterstellung, da sie eine völlig neue Schicht zwischen dem Benutzer und dem Produkt erschafft. Diese neue Schicht ermöglicht nicht nur eine einfachere Code-Editierung, sondern bietet auch einen neuen Platz für die Implementierung neuer benötigter Funktionalitäten.

Obwohl die Factory Method sehr nützlich ist, hat sie auch ihre Nachteile. Der offensichtlichste besteht in der Anzahl der neuen Creator-Klassen, die erstellt werden müssen. Für jeden Typ von Objekten, die erzeugt werden sollen, muss es auch eine neue Unterklasse der abstrakten Creator-Klasse geben, um dieses Objekt zu konstruieren. Wenn die Anzahl der benötigten Objekte steigt, kann das schnell außer Kontrolle geraten. Ohne diese Methode müsste der Benutzer jedoch jede einzelne konkrete Produkt-Unterklasse direkt kennen, dieser Nachteil muss also gegen die potenziellen Probleme (wie verstärkte Kopplung) abgewägt werden. Ein nützlicher Weg, um den Schaden der großen Anzahl von Creator-Klassen zu mindern, ist die Implementierung mehrerer abstrakter Creator-Klassen. Das erlaubt es, eine größere Trennung zwischen verschiedenen Teilen des Programms zu erschaffen, sowie eine logische Struktur für Objekte, welche zu besserer Lesbarkeit führt, zu implementieren.

Die Factory Method könnte man beispielsweise benutzen, um ein Übersetzer-Objekt zu erzeugen. Von vielen bereits implementierten Objekten, die zum Beispiel von Englisch oder Russisch nach Deutsch übersetzen, braucht der Benutzer nur einen. Ohne das Factory-Pattern würde er über jedes dieser Objekte wissen müssen und Zugriff darauf haben, um eine Sprache zu benutzen. Mit dem Factory-Pattern könnte er einfach ein *CreatorGPS*-Objekt erzeugen, welches mithilfe des genauen Standorts des Benutzers genau den Übersetzer zurückgeben kann, welchen der Benutzer auch braucht. Der Benutzer

könnte auch einen *CreatorSearch*-Creator benutzen, welcher als Argument einen Suchterm annimmt und den gewünschten Übersetzer zurückliefert. In diesem Beispiel handelt es sich bei den Übersetzern um die konkreten Produktobjekte mit verschieden implementierten Übersetzungsmethoden, während *CreatorGPS* und *CreatorSearch* die konkreten Creator mit verschieden implementierten *create()*-Methoden sind.

3 Iterator

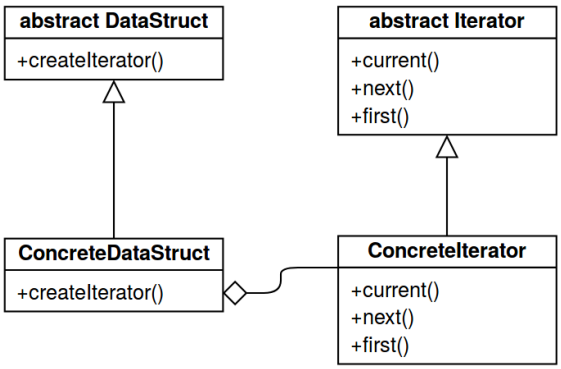

Abbildung 2: Struktur des Iterator-Patterns

Ein weiteres, häufig bei der Programmierung bestehendes Problem ist die Durchquerung von Datenstrukturen. Da diese Strukturen sehr kompliziert werden können, kann die Bewegung durch diese Strukturen schwierig werden. Die Design Pattern-Lösung dafür ist das *Iterator*-Pattern. Dieses Pattern basiert auf einem ziemlich einfachen, aber äußerst nützlichen Konzept. Es wird so häufig verwendet, dass die meisten objektorientierten Sprachen keine eigene Implementierung dafür brauchen, da Iterators standardmäßig Teil der Sprache sind. Es handelt sich um ein Behavioural Pattern, das auf Objekt-Ebene arbeitet, es steuert also, wie sich bestimmte Objekte verhalten.

Es gibt zwei Möglichkeiten, eine Iterator-Klasse zu implementieren. Bei der ersten handelt es sich um die einfache, triviale Version: Neben einer Datenstruktur-Klasse erzeugt man auch eine Iterator-Klasse mit den Methoden *current()* und *next()*. Es kann natürlich mehr implementiert werden, aber diese beiden Methoden sind alles, was für einen vollständigen Iterator gebraucht wird. Jedes Objekt der Datenstruktur hat eine Methode *getIterator()*, welche ein Iterator-Objekt zurückgibt, welches mit current() auf das aktuelle Datenstrukturobjekt zeigt. Mit diesem Iterator-Objekt kann man sich nun mithilfe von next() durch die Datenstruktur bewegen.

Besonders nützlich wird das Konzept des Iterators durch die zweite Methode seiner Implementierung. Durch die Kombination von Iterators und der zuvor beschriebenen Factory Method kann man eine ganze Klasse von Iterators für eine ganze Klasse von Datenstrukturen erstellen. Genau wie im Factory-Pattern muss man abstrakte Oberklassen *DataStructure* und *Iterator* benutzen, die verschieden implementierte konkrete Unterklassen haben.

Dies ist besonders nützlich für Situationen, in denen verschiedene Datenstrukturen den gleichen Iterationsalgorithmus unterstützen, oder wenn die gleiche Datenstruktur mehrere verschiedene Iterationsalgorithmen braucht. Das Factory-Pattern kann verwendet werden, um zu entscheiden, welcher Iterator erstellt wird, während jeder Iterator seinen eigenen internen Zustand und Implementierung der Durchquerungsmethode next() hat.

Eine Trennung zwischen einer Datenstruktur und ihrem Durchquermechanismus ist sowohl für die Entkopplung von Daten und Funktionalität, als auch für die Reduzierung der Zugriffsrechte des Benutzers auf die interne Mechanik der Datenstruktur hilfreich. Da sich die Daten selbst sehr stark von den Aktionen, die auf die Daten angewendet werden können, unterscheiden, ist es sinnvoll, diese Elemente getrennt voneinander zu halten. Dieser Unterschied der Elemente bedeutet auch, dass in einer großen Gruppe, in der wahrscheinlich verschiedene Leute an verschiedenen Teilen des Codes arbeiten, Klassen von verschiedenen Teams entwickelt werden. Der Iterator wird zum Beispiel von Clients verwendet, die Datenstruktur hingegen von den Datenwissenschaftlern. Die Trennung dient also nicht nur der Entkopplung und der effizienten Entwicklung, sondern auch der grundsätzlichen Lesbarkeit des Programms. Der Datenwissenschaftler muss nicht wissen, wie auf seine Daten zugegriffen wird, während der Client nichts über die interne Struktur der Datenstruktur wissen muss. Ein weiterer Vorteil der Verwendung von Iterator ist die Vereinfachung der Datenstruktur. Sehr oft werden Datenstrukturen übermäßig kompliziert entwickelt: Sie haben Methoden, um Daten hinzuzufügen, zu verschieben, zu löschen, zu durchqueren, zu rechnen, und so weiter. Iterators ermöglichen es dem Programmierer, einen Teil der überkomplizierten Funktionalität in der Datenstruktur auf eine neue Schicht herauszunehmen, wodurch die Struktur effizienter und einfacher zu bedienen ist.

Als Beispiel könnte man die Durchquerung von Listen betrachten. Die Datenstrukturen sind hierbei verschiedene Arten von Listen, die einfach, doppelt und zyklisch verkettet sind. Ein Iterator *IteratorNorm*, dessen *next()* Implementation einfach vorwärts zum nächsten Element geht, könnte für alle drei der Listenarten benutzt werden. Man könnte auch einen *IteratorBack*-Iterator implementieren, der nicht vorwärts, sondern nur rückwärts durch die Liste gehen kann. Dieser würde allerdings nur für die doppelt verkettete Listen funktionieren, da es in den anderen Listen keinen Zeiger auf das vorherige Element gibt. Deshalb würden nur die Elemente einer doppelt verketteten Liste eine *createIterator()*-Methode für *IteratorBack* haben. Die *createIterator()*-Methode für *IteratorNorm* kann es allerdings für jede Listenart geben. Mithilfe dieser Methoden kann der Benutzer die drei Datenstrukturen durchqueren, ohne die Strukturen selbst zu benutzen.

4 Service Layer

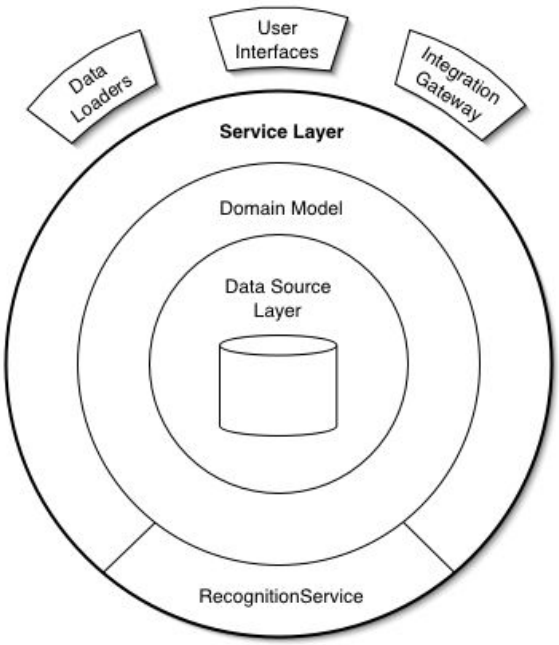

Abbildung 3: Schematische Darstellung der Layers eines Unternehmensprogramms
Bildquelle: https://www.martinfowler.com/eaaCatalog/serviceLayer.html

Das *Service Layer*-Pattern ist ein Pattern, das sich auf die Architektur von Unternehmenssoftware bezieht. Um es besser zu verstehen, soll hier kurz auf Letztere eingegangen werden. Der Begriff „Unternehmenssoftware“ bezieht sich auf Programme, die über einen langen Zeitraum mit großen Datenmengen arbeiten, wie zum Beispiel Buchhaltungsprogramme. Diese Programme müssen so entwickelt werden, dass einzelne Aspekte des Programms leicht an sich ändernde Anforderungen angepasst werden können, da es ein enormer Aufwand wäre, sie bei jeder Änderung komplett neu zu programmieren.

Zu diesem Zweck schlägt Fowler vor, diese Programme in drei grundlegende Ebenen, sogenannte Layer, zu unterteilen, die relativ unabhängig voneinander sind. Jeder Layer soll dabei nur direkten Zugriff auf den Layer direkt unter ihr haben. Die drei Layer, die jede Unternehmensanwendung haben sollte, sind dabei die Folgenden:

1. Der *Data Source Layer*, der für das Beschaffen von Informationen aus z.B. Datenbanken verantwortlich ist.

2. Der *Domain Layer*, der die tatsächliche Logik des Programms beinhaltet.

3. Der *Presentation Layer*, der für die Ausgabe von Informationen verantwortlich ist, z.B. in User Interfaces.

In manchen Programmen kann es sein, dass der Presentation Layer viele verschiedene Clients beinhaltet, die unterschiedliche Funktionen von dem Domain Layer benötigen. In einem Buchhaltungssystem einer Firma könnte es zum Beispiel ein Interface geben, dass den Angestellten ihre Gehälter anzeigt, aber auch ein Programm, das Zahlungen an andere Firmen für abgeschlossene Verträge tätigt. Bei komplexen Anwendungen kann es viele dieser Systeme geben, was leicht zu Verwirrung führen kann. An dieser Stelle greift das Service Layer-Pattern ein.

Der Service Layer befindet sich zwischen dem Domain Layer und dem Presentation Layer. Anders als die anderen in dieser Ausarbeitung vorgestellten Patterns, bezieht sich das Service Layer-Pattern nicht auf einzelne kleine Probleme in einem Programm, sondern befasst sich mit dem grundlegenden Aufbau einer Anwendung. Die Service Layer dient dazu, genau zu definieren, welche Operationen den Clients in der Presentation Layer zur Verfügung stehen. Dazu werden ähnliche Services gruppiert und in einer Klasse platziert, wobei diese Klassen meistens das Wort *Service* im Namen tragen.

Um das Verständnis des Service Layer-Patterns zu erleichtern, wollen wir hier noch etwas näher auf das oben genannte Beispiel von einer Buchhaltungssoftware eingehen. Dieses Buchhaltungssystem soll Verträge und Angestellte für ein Unternehmen verwalten. Wir gehen davon aus, dass es zwei Clients in dem Presentation Layer gibt: Ein Programm einer Bank, das sich um die Bezahlung von Vertragsschulden kümmert, und ein User Interface für die Angestellten, in dem sie verschiedene Informationen, z.B. ihr aktuelles Gehalt oder das Datum, an dem sie die nächste Lohnerhöhung erwarten können. In dem Data Source Layer würde es dann Methoden geben, um Verträge oder Angestellte aus einer Datenbank des Unternehmens abzurufen. In dem Domain Layer hingegen wären die Verträge und Angestellten als Objekte repräsentiert. Diese *Contract-* und *Employee*-Klassen enthalten verschiedene Attribute und Methoden. Es könnte beispielsweise eine *calculateNextRaise*-Methode in Employee, die eventuell äußerst komplexe Logik enthält, geben. In dem Service Layer hingegen gäbe es die Klassen *ContractsService* und *EmployeeService*, die Methoden enthalten, um Informationen von den Contract- und Employee-Klassen zu erhalten oder die Attribute der Klassen zu verändern, aber nicht die eigentliche Logik implementieren, sodass der Benutzer leichter die Übersicht behalten kann.

Dadurch, dass der Service Layer die verfügbaren Operationen klar definiert, werden die Programme weniger verwirrend, und es wird einfacher, neue Clients anzubinden. Der Service Layer kann allerdings noch eine weitere Funktion haben. Diese besteht in der Trennung von *domain logic* und *application logic*. Dabei ist mit domain logic der Teil des Systems gemeint, der die tatsächliche Geschäftslogik beinhaltet, wie die Berechnung von Gehältern, mit application logic jene Funktionen, die speziell für diese Anwendung wichtig sind, z.B. die Benachrichtigung von Anwendern. Bei der sogenannten *Operation Script*-Implementierung wird die application logic in den Service Layer, die domain logic in den Domain Layer eingebunden. Dadurch lässt sich die domain logic in anderen Programmen wiederverwenden. Im Gegenzug dazu steht die *Domain Facade*-Implementierung, bei der der Service Layer lediglich die verfügbaren Operationen definiert und die eigentliche Programmlogik in dem Domain Layer implementiert ist.

Welche Operationen der Service Layer anbietet, hängt von den Anforderungen der Clients ab, allerdings handelt es sich meistens um sogenannte *CRUD*-Funktionen. CRUD ist ein Akronym, das für *Create, Read, Update, Delete* steht. Folglich bietet der Service Layer Operationen zum Erstellen, Ausgeben, Verändern und Löschen von Daten an.

Einen Service Layer in einem Programm einzubauen, ist insbesondere bei komplexen Programmen, die mehrere Clients mit verschiedenen Anforderungen haben oder mit mehreren verschiedenen Arten von Daten arbeiten, sinnvoll. Bei weniger komplexen Programmen hingegen ist ein Service Layer meistens nicht notwendig.

5 Money

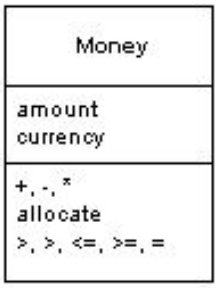

Abbildung 4: Struktur der Money-Klasse
Bildquelle: https://www.martinfowler.com/eaaCatalog/money.html

In Anwendungen für Unternehmen werden mit hoher Frequenz Berechnungen mit großen Geldmengen durchgeführt. Folglich ist es notwendig, Geldmengen in Anwendungen exakt zu repräsentieren. Auf den ersten Blick könnte man denken, dass man dafür einfach einen primitiven Datentyp für Zahlen verwenden könnte, allerdings fallen dort bei näherer Betrachtung zwei Probleme auf: Das erste Problem liegt bei der Rundung von Geldmengen vor. Wenn man z.B. 5 Cent auf zwei Personen aufteilt und dazu einen int-Wert durch 2 teilt, geht ein Cent verloren. Gleitkommazahlen haben von sich aus eine gewisse Ungenauigkeit. Das zweite Problem trifft man bei der Nutzung verschiedener Währungen an. Das Programm kann nicht zwischen verschiedenen Währungen unterscheiden, wenn einfach primitive Datentypen genutzt werden.

Als Lösung für diese Probleme wird bei Fowler das *Money*-Pattern vorgeschlagen. Die Idee hinter dem Money-Pattern ist recht simpel: Geldwerte werden in einer Klasse, die den Betrag und die Währung des Geldes als Attribute besitzt, repräsentiert. Für den Betrag sollte man hierbei einen Datentyp verwenden, der große Zahlen hat, aber nicht zu Ungenauigkeiten beim Speichern führt. Deshalb sind float oder double keine sinnvollen Datentypen hierfür. Stattdessen bietet sich in Java z.B. der primitive Datentyp *long* an. Alternativ bieten sich auch die Klassen *BigInteger* oder *BigDecimal* an, die Ganzzahlen bzw. Dezimalzahlen beliebiger Länge ohne Genauigkeitsverlust speichern können.[jav] Die Währung hingegen lässt sich zum Beispiel als String oder enum speichern.

Neben den Attributen schlägt Fowler einige Methoden vor, die in die Klasse implementiert werden können. Welche dieser Methoden implementiert werden und wie genau sie umgesetzt werden, muss natürlich an die Anforderungen des Programmes angepasst werden. Im Folgenden werden einige dieser Methoden beschrieben.

Einige Methoden, die sehr häufig benötigt werden, sind Methoden zur Addition und Subtraktion von Geldbeträgen. Dabei muss darauf geachtet werden, dass Geldmengen verschiedener Währungen nicht direkt addiert werden können. Dafür kann zum Beispiel nur das Rechnen mit Money-Objekten mit gleichen Währungen erlaubt und eine Exception geworfen werden, wenn die Währungen abweichen. Neben der Addition und Subtraktion von Geldbeträgen ist auch die Multiplikation mit beliebigen Zahlen oft sinnvoll, um z.B. Zinsen zu berechnen.

Wenn man mit verschiedenen Währungen arbeitet, ist es oft notwendig, zwischen diesen Währungen zu konvertieren. Auch dafür können Methoden programmiert werden, die beispielsweise die Wechselkurse aus einer Datenbank lesen, um Beträge umzurechnen.

Ebenfalls sinnvoll sind Vergleichsmethoden, um z.B. festzustellen, ob eine Rechnung vollständig bezahlt wurde. Auch bei der Implementierung dieser Methoden müssen verschiedene Währungen beachtet werden. Das lässt sich etwa umsetzen, indem man Geldbeträge vor dem Vergleich in eine Währung umrechnen lässt, aber auch hier hängt die genaue Umsetzung von den Anforderungen an die Anwendung ab.

Die letzte Methode, die hier vorgestellt werden soll, ist die *allocate*-Methode. Diese soll den Betrag eines Money-Objekts auf mehrere Objekte aufteilen. Damit bei der Aufteilung keine kleinen Beträge verloren gehen, soll die Methode speichern, wieviel Geld sie bereits verteilt hat und den Rest dann einzeln zu beliebigen Money-Objekten hinzufügen. Wenn man z.B. 8 Cent auf 3 Accounts aufteilen möchte, würde die Methode erst jedem Account 2 Cent geben und dann zwei der drei Accounts noch einen der übriggebliebenen Cents geben.

Durch die Implementierung dieser Methoden lassen sich Geldmengen sehr sinnvoll in Programmen repräsentieren. Die Verwendung dieses Patterns ist besonders vorteilhaft, wenn man mit Geldwerten in verschiedenen Währungen arbeitet. Der größte Nachteil des Patterns besteht darin, dass das Money-Objekt mehr Rechenleistung benötigt als ein primitiver Datentyp, allerdings ist dieser Unterschied nur selten wirklich signifikant.

6 Zusammenfassung

Diese Ausarbeitung sollte einen Einblick in das Konzept des Design Patterns geben. Dazu haben wir zuerst die Grundidee hinter Design Patterns vorgestellt und die Vorteile dieser erläutert. In den darauffolgenden Kapiteln wurden vier Design Patterns exemplarisch näher beleuchtet: Das Factory-Pattern, das zur Erstellung von Objekten dient, das Iterator-Pattern, mit dem man Datenstrukturen durchquert, das Service Layer-Pattern, das sich auf den Aufbau von Unternehmenssoftware bezieht, und das Money-Pattern, mit dem man Geldbeträge darstellen kann.

Zusammenfassend lässt sich feststellen, dass Design Patterns ein äußerst nützliches Werkzeug im Repertoire eines Programmierers sind, das die Entwicklung von Software in vielerlei Hinsicht erleichtert, weil sie überflüssige Arbeit sparen und bewährte Lösungsansätze bieten, wenn der Programmierer weiß, wie man sie richtig auswählt und implementiert. Diese Ausarbeitung kratzt allerdings lediglich an der Oberfläche des Themas. Die schiere Menge an Design Patterns macht es unmöglich, in einer Ausarbeitung dieser Länge auf alle einzugehen. Es gibt eine Vielzahl von Quellen, die sich weitergehend mit Design Patterns beschäftigen und verschiedene Design Patterns näher beleuchten. Dazu zählen zum Beispiel auch die Werke, auf denen diese Ausarbeitung basiert. Bei Interesse können also sehr leicht weitere Design Patterns nachgeschlagen werden, die andere Problemstellungen als die hier vorgestellten lösen können.

Literatur

[Fow03] M. Fowler. *Patterns of Enterprise Application Architecture.* A Martin Fowler signature book. Addison-Wesley, 2003.

[GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[jav] Java Platform Standard Edition API Reference, Java SE 12. Zugriff am 11.04.2019.