Objektorientierte Programmierung als bewusste Entscheidung

Torben Fojuth, neuland – Büro für Informatik

Java gilt als objektorientierte Programmiersprache – dies ist hinlänglich bekannt. Das Beherrschen der Syntax allein geht jedoch nicht automatisch mit einem Verständnis der zugrunde liegenden Paradigmen einher. Dass durch die bloße Wahl von Java als Programmiersprache automatisch auch objektorientierter Code entsteht, ist ein Irrtum.

Bei der objektorientierten Programmierung (nachfolgend kurz OOP genannt) handelt es sich um ein Programmierparadigma. Fragt man Wikipedia *[1]*, so ist OOP nur ein Paradigma unter vielen aus dem Bereich der imperativen Programmierung. Etwas kompakter dagegen ist die Sicht von Robert „Uncle Bob" Martin *[2]* auf die Welt der Programmierparadigmen. Aus seiner Sicht ist OOP eines von insgesamt drei fundamentalen Paradigmen:

1. objektorientierte Programmierung
2. funktionale Programmierung
3. strukturierte Programmierung

Für Martin definiert sich ein Paradigma vor allem durch das Beschränken der Möglichkeiten, die dem Entwickler bei der Program-

mierung zur Verfügung stehen. Im Falle der OOP wird die Möglichkeit entfernt, beliebige Funktionen aufzurufen, da Funktionen – in Form von Methoden – in der Idee der OOP im Kontext von Objekten existieren und entsprechend verwendet werden müssen.

Objektorientierte Programmiersprachen bieten in der Regel viele weitere Features wie beispielsweise Polymorphie und Vererbung. Diese Techniken sowie deren sachgemäße Verwendung werden aufgrund des engen Rahmens jedoch nicht behandelt. Es soll im Folgenden vielmehr um das Konzept der Bündelung von Daten sowie Operationen auf diesen Daten gehen, also um die Idee des Objektes. Im Kontrast dazu nun ein Beispiel *(Listing 1)* für nicht objektorientierten Code in Java.

```
public static void main(String[] arguments) {
  List<String> lines = readLinesFromFile();
  lines = sortLinesByFirstCharacter(lines);
  print(lines);
}
```

Listing1: Code ohne Anwendung von OOP

Rein prozeduraler Code in Java

Dass man in Java mit Leichtigkeit Code schreiben kann, der nichts mit OOP zu tun hat, zeigt *Listing 1*. Man erkennt auf den ersten Blick, unter anderem am fehlenden Schlüsselwort new, dass OOP hier keine Rolle spielt. Es werden ausschließlich statische Hilfsmethoden ohne einen umgebenden Objektkontext aufgerufen. So handelt es sich zwar um imperativen, nicht aber um objektorientierten Code.

Listing 1 ist bewusst darauf getrimmt, rein imperativen Code zu zeigen. Bei der täglichen Arbeit an echtem Produktiv-Code fällt es etwas schwerer zu erkennen, wo Chancen für objektorientierte Modellierung verpasst wurden.

Welche Vorteile eine objektorientierte Modellierung mit sich bringt und welche potenziellen Gefahren der Verzicht von OOP birgt, soll nun näher betrachtet werden.

Geschäftslogik außerhalb des Objektes

In *Listing 2* folgt ein weiteres, weniger offensichtliches Beispiel für nicht konsequent objektorientierten Code, der aus einem regulären Softwareprojekt stammen könnte:

Es geht um einen Service, der einen Artikel zu einem Warenkorb hinzufügt. Obiger Code enthält viel Potenzial für Verbesserungen, es kommt aber zunächst auf die enthaltene Geschäftslogik an: Offenbar ist es in diesem System wichtig, dass der Gesamtwert eines Warenkorbs 500 Währungseinheiten nicht übersteigen darf. Konsequenterweise findet sich diese Regel auch im Code wieder. Worin besteht also die Kritik?

Das Problem an der gezeigten Implementierung besteht darin, dass diese Geschäftsregel außerhalb des Objektes Cart implementiert ist. Verwendet ein Mitarbeiter nicht diesen CartService zum Hinzufügen des Artikels, sondern beispielsweise direkt den Cart, umgeht er oder sie ungewollt die obige Regel und könnte so größere Warenkörbe erstellen als vorgesehen.

Geschäftslogik in das Objekt verschieben

Die Lösung des Problems besteht darin, diese Logik in das Objekt Cart zu verlagern, wie in *Listing 3* gezeigt. Durch diese Maßnahme ist gewährleistet, dass die Logik an der richtigen Stelle steht und der Warenkorb stets in sich konsistent bleibt.

Mit dem Umbau gehen ganz automatisch weitere Änderungen einher, die der Idee der OOP entgegenkommen. So ist in der neuen Variante zum einen das Warenkorblimit nun Bestandteil der richtigen Klasse Cart - zuvor war die Konstante außerhalb definiert. Zum anderen muss der Warenkorb (zumindest zu diesem Zweck) nun keine Methode zum Abfragen seines aktuellen Gesamtwerts mehr bereitstellen und kann so seine Interna für sich behalten.

Beide Veränderungen sorgen dafür, dass Logik in die Objekte hineinwandert, statt außerhalb implementiert zu werden. Dieses Grundprinzip der OOP ist auch unter dem Namen „Tell, don't ask" *[3]* bekannt und besagt, dass man Objekten per Methodenaufruf sagen soll, was sie tun sollen, statt sie nach ihren Daten zu fragen, um es selbst zu tun:

„Procedural code gets information then makes decisions. Object-oriented code tells objects to do things."
— Alec Sharp

```
public void addItemToCart(Item item, CartId cartId) {
  Cart cart = loadCart(cartId);
  if(cart.total() + item.price() > 500.00f){
     throw new CartLimitExceeded();
  }
  cart.addItem(item);
}
```

Listing 2: Hinzufügen eines Artikels zum Warenkorb

```
public void addItemToCart(Item item, Cart cart) {
  Cart cart = loadCart(cartId);
  cart.addItem(item);
}

public class Cart {
  private static final Float LIMIT = 500.00f;

  List<Item> items = new ArrayList<>();

  public void addItem(Item item) {
     if(this.total() + item.price() > LIMIT){
        throw new CartLimitExceeded();
     }
     items.add(item);
  }
}
```

Listing 3: Domänenlogik liegt nun innerhalb des Cart-Objekts

```
public class Cart {
  private static final Money LIMIT = new Money(500, 0, EURO);
  ...
  public void addItem(Item item) {
     if(this.total().plus(item.price()).exceeds(LIMIT)){
        throw new CartLimitExceeded();
     }
     items.add(item);
  }
```

Listing 4: Währungsbeträge als Value Objects

```
public void processOrder(Order order){
  String orderId = order.id();
  if(orderId.endsWith("-23")){
     germanLogisitcs.process(order);
  } else {
     austrianLogistics.process(order);
  }
}
```

Listing 5: Bestellnummern als String

```
public void transferOrder(Order order){
  OrderId orderId = order.id();
  if(orderId.isGermanOrder()){
     germanLogisitcs.process(order);
  } else {
     austrianLogistics.process(order);
  }
}
```

Listing 6: Die Bestellnummer als Value Object

```
public void updateAverageCartValueForCurrentMonth(){
  Orders orders = ordersCurrentMonth();
  AverageCartValue average = new AverageCartValue(orders);
  monitoring.updateAverageCartValueKPI(average.toMon-
ey());
}

public class AvergageCartValue(){
  public AverageCartValue(Orders orders) {
     this.orders = orders;
  }
  public Money toMoney(){
     Money sum = new Money(0,0, EUR);
     orders.forEach(order -> sum = order.addTotalTo(sum));
     Money average = sum.divideBy(orders.size());
     return average;
  }
}
```

Listing 7: Durchschnittlicher Warenkorbwert als Klasse

Von allen Prinzipien zum Erstellen von gut wartbarem objektorientierten Code ist dieses Prinzip sicher eines der wichtigsten.

Werden, bei vorgeblich objektorientiertem Code, Prinzipien der OOP konsequent nicht beachtet, spricht man in der Softwareentwicklung von sogenannten Anti-Patterns. Das Anti-Pattern für die Nichtbeachtung des „Tell, don't ask"-Prinzips wird als anämisches Datenmodell *[4]* bezeichnet. In einem anämischen Datenmodell besteht ein Großteil der Klassen lediglich aus Feldern sowie deren Settern und Gettern. Sämtliche Logik ist hingegen außerhalb der Klassen implementiert. Diese Klassen stellen also reine Daten-Konstrukte dar und verdienen es nicht, als Objekte im Sinne der OOP bezeichnet zu werden. Beim Antreffen eines solchen anämischen Datenmodells in einem Projekt ist höchste Vorsicht geboten. Bevor ein Beispiel diese Gefahr erläutert, soll aber zunächst der Begriff der Value Objects eingeführt werden.

Werte als Objekte: Value Objects

Zurück zum Code rund um den Warenkorb: Nicht nur Dinge wie der Warenkorb, die einem Lebenszyklus in Form von Zustandsänderungen unterliegen, können und sollten als Objekte modelliert werden. Ebenso gewinnen Eigenschaften von Objekten an Bedeutung und Sicherheit, wenn sie als Objekte modelliert werden. In obigem Beispiel könnten etwa die Währungsbeträge als Objekte mit eigenen, sprechenden Methoden dargestellt werden. Diese Art von Objekten wird auch Value Object genannt *(siehe [5] und [6])*. *Listing 4* zeigt den Code nach einem weiteren Umbau.

Welcher Vorteil ergibt sich daraus? Nun, speziell über das Thema Währungsbeträge und deren Modellierung mit primitiven Typen wurde – unter anderem in *[7]* – schon viel geschrieben. Im Kern bietet die Modellierung mit Value Objects vor allem folgende Vorteile:

1. die Absicherung der Wertebereiche
2. die Kontrolle über verfügbare Operationen
3. ein zentraler Ort für Geschäftslogik, die sich auf den Wert bezieht

In obigem Kontext sind negative Zahlen für Preise nicht sehr sinnvoll. Dies kann in einer konkreten Implementierung direkt im Konstruktor sichergestellt werden, sodass es nicht mehr möglich ist, in diesem Sinne ungültige Währungsbeträge zu erstellen. Als Entwickler/in sollte man sich vor der Verwendung von primitiven Typen stets die Frage stellen: Stimmt der fachliche Wertebereich mit dem technischen Wertebereich des verwendeten Typs überein? Ist dies nicht der Fall, ist eine Modellierung des Wertes als Value Object sinnvoll.

Des Weiteren sind insbesondere die Addition und Subtraktion von Fließkommazahlen nie exakt. Das in *Listing 4* gezeigte Value Object Money verlangt deshalb separate, ganzzahlige Werte für den Vor- und den Nachkommateil des Preises und definiert eine eigene Methode für die Addition, um sicherzustellen, dass die Operation stets exakt funktioniert. Glücklicherweise steht dem JDK mit *[8]* mittlerweile ein Modul zur Verfügung, das keine Wünsche im Umgang mit Währungsbeträgen offen lässt. Probleme bei der Modellierung von Währungsbeträgen gehören damit der Vergangenheit an.

Beim dritten Vorteil der Value Objects soll es, wie versprochen, noch einmal um die Gefahren des anämischen Datenmodells gehen. Angenommen in einem System existieren zwei sogenannte Mandanten: Deutschland und Österreich. Beide haben unterschiedliche Arten von Bestellnummern, die durch ein magisches Kürzel am Ende der Bestellnummer („-23" für Deutschland und „-42" für Österreich) gekennzeichnet sind. Das System muss an verschiedenen Stellen zwischen beiden Versionen unterscheiden, so beispielsweise beim Übergeben der Bestellung an die Logistik.

In obigem Beispiel ist erneut Geschäftslogik aus dem Modell „herausgesickert" und wird stattdessen vom Benutzer der Klasse implementiert. Das Wissen zur Berechnung der Mandanten befindet sich nun an jeder Stelle im System, die diese Unterscheidung benötigt. Eventuell geschieht dies sogar in leicht abgewandelter Form, je nachdem, wer es implementiert hat, was das Auffinden und Erkennen dieser Stellen zusätzlich erschwert.

Was passiert nun, wenn sich die Kürzel ändern oder die Schweiz als weiterer Mandant mit eigenem Kürzel eingeführt wird? Es muss jede dieser Stellen gefunden und korrekt angepasst werden. Ein solches Vorgehen ist nicht nur sehr aufwendig, sondern gleichzeitig auch fehleranfällig.

Modelliert man hingegen die Bestellnummer als Value Object, so wird die Logik an genau einer Stelle implementiert: der Bestellnummer selbst *(siehe Listing 6)*.

Modellierung abstrakter fachlicher Konzepte als Objekt

Bei der Modellierung eines Geschäftsfeldes durch Klassen sollte man sich, gerade mit Blick auf das „Tell, don't ask"-Prinzip, nicht darauf beschränken, ausschließlich greifbare Subjekte der Domäne wie beispielsweise den Kunden, den Warenkorb oder den Artikel zu modellieren.

Das Modell gewinnt durch die Modellierung abstrakter fachlicher Konzepte, wie etwa spezieller Kennzahlen, an Bedeutung. So könnte der durchschnittliche Warenwert der Warenkörbe pro Monat eine wichtige Kennzahl für ein Unternehmen sein. *Listing 7* zeigt, wie dies aussehen könnte.

Einmal für diesen Gedanken sensibilisiert, lassen sich in fast jeder Domäne weitere Konzepte identifizieren, die einer separaten Modellierung würdig sind. Weitere Inspiration sowie einige streitbare Thesen zur Implementierung von Klassen finden sich in dem empfehlenswerten Erstlingswerk von Yegor Bugayenko *[10]*.

Fazit

Die Java-Syntax und die Klassen des JDK zu kennen, sind eine solide Basis für eine Tätigkeit als Softwareentwickler/in. Will man das Handwerk der Softwareentwicklung professionell betreiben – und dabei OOP einsetzen –, gibt es Konzepte hinter dem Code, die man sich aneignen sollte.

Neben den Referenzen, die schon im Verlauf des Artikels erwähnt wurden, ist ein Verständnis der SOLID-Prinzipien *[11]* ein guter Schritt. Ebenso gibt es – subjektiv betrachtet – Standardwerke, die man gelesen haben sollte. Zu diesen zählen meiner Meinung nach mindestens „Clean Code" *[12]*, das POODR-Buch *[13]* sowie das GOOS-Buch *[14]*.

Diese Bücher eignen sich auch hervorragend für eine kooperative Lektüre, beispielsweise in Form einer Leserunde unter Kollegen und Kolleginnen. Um das neu erworbene Wissen zu verstetigen, haben sich die Ideen des gemeinsamen Code Review als Termin mit dem gesamten Team sowie Peer Reviews (falls das Team mit Merge Requests arbeitet) als hilfreich erwiesen.

Abschließend noch ein Hinweis zur Einordnung der in diesem Artikel behandelten Themen: Der Rahmen eines Artikels ermöglicht natürlich keine erschöpfende Behandlung aller Themen aus dem Themenkomplex der OOP. Des Weiteren bleiben Themen unerwähnt, die auf den höheren Abstraktionsebenen liegen oder einen gänzlich eigenen Themenkomplex darstellen:

- Organisation von Klassen innerhalb einer Anwendung
- Organisation von Anwendungen innerhalb einer System-Infrastruktur
- Modellierung des Geschäftsfeldes mithilfe von Domain-driven Design

Der Artikel vermittelt einen ersten Eindruck davon, wie viel es über objektorientierte Programmierung, abseits von Syntax und Sprachfeatures, zu lernen gibt. Bei Fragen oder Anmerkungen kontaktieren Sie mich gern.

Literaturverzeichnis

[1] *https://de.wikipedia.org/wiki/Programmierparadigma*
[2] Robert C. Martin, Three Paradigms, 2012, *https://blog.cleancoder.com/uncle-bob/2012/12/19/Three-Paradigms.html*
[3] *https://pragprog.com/articles/tell-dont-ask*
[4] Martin Fowler, , Stand 2018, *https://www.martinfowler.com/bliki/AnemicDomainModel.html*
[5] Eric J. Evans, Domain-Driven Design: Tackling Complexity in the Heart of Software, 2003
[6] Vaughn Vernon, Implementing Domain-Driven Design, 2013
[7] Joschua Bloch, Effective Java, 28.5.2008
[8] *https://jcp.org/en/jsr/detail?id=354*
[9] *https://dzone.com/articles/solid-principles-by-examples-liskov-substitution-p*
[10] Yegor Bugayenko, Elegant Objects, 2016
[11] 2018, *https://en.wikipedia.org/wiki/SOLID*
[12] Robert C. Martin, Clean Code: A Handbook of Agile Software Craftsmanship, 2008
[13] Nat Pryce, Steve Freeman, Growing Object-Oriented Software, Guided by Tests , 2009
[14] Sandi Metz, Practical Object-Oriented Design in Ruby: An Agile Primer, 2012

Torben Fojuth
torben.fojuth@posteo.de

Seit seinem Abschlss an der Universität Bremen in 2007 arbeitet Torben Fojuth als Softwarentwickler im Web-Umfeld. Aktuell ist er bei der "neuland - Büro für Informatik GmbH" als Softwareentwickler und Ausbilder tätig. Seine Schwerpunkte liegen in den Bereichen Softwarearchitektur, Clean Code und Domain-driven Design. Seine Rolle in der Lehre und Fortbildung ist eine gute Ergänzung zur Projektarbeit.