OO-Design-Patterns erweitert um Lambdas – eine Auswahl

Christian Nockemann, viadee Unternehmensberatung GmbH

Die Entwurfsmuster der „Gang-of-Four" haben sich als nützliche Werkzeuge für die Lösung gängiger Herausforderungen in der objektorientierten Programmierung bewährt. Wie profitiert diese Lösung von einem Einsatz der mit Java 8 eingeführten Lambda-Ausdrücke?

Hinweis: Alle der hier vorgestellten Code-Beispiele sind frei auf GitHub unter *https://github.com/cnockemann/lambda-enhanced-patterns* verfügbar.

„Design Patterns", das Buch der Autoren Gamma, Helm, Johnson und Vlissides (Gang-of-Four) ist ein wegweisendes Werk der objektorientierten Software-Entwicklung. Entwurfsmuster wie Beobachter, Strategie oder Adapter gehören inzwischen zu den Standardwerkzeugen von nahezu allen Java-Software-Entwicklern – teilweise ohne dass ihnen deren Ursprung bewusst ist, da viele weitverbreitete Java-Frameworks auf ihnen basieren. Sie bieten standardisierte Lösungsansätze für wiederkehrende Probleme und Herausforderungen im Software-Design.

In ihrer ursprünglichen Form leiten sich die Entwurfsmuster der GoF her aus den grundlegenden Paradigmen der Objektorientierung: Polymorphie, Vererbung und Kapselung. Zusätzlich hat mit den Lambda-Ausdrücken seit Java 8 die funktionale Programmierung Einzug in die objektorientierte Welt gehalten. Daher ist es an der Zeit, die Entwurfsmuster daraufhin zu prüfen, ob sie durch den Einsatz von Lambda-Ausdrücken vereinfacht oder sinnvoll erweitert werden können.

Dieser Artikel stellt eine Auswahl von Entwurfsmustern der GoF sowie jeweils eine um Lambda-Ausdrücke erweiterte entsprechende Variante vor. Die ausgewählten Entwurfsmuster bringen bei Design und Implementierung von Software-Komponenten erfahrungsgemäß den größten Nutzen und erscheinen beim Einsatz von Lambda-Ausdrücken überhaupt sinnvoll.

Warum überhaupt Entwurfsmuster?

Zunächst ein paar Worte zur Notwendigkeit des Einsatzes der Entwurfsmuster beziehungsweise die Beantwortung der grundsätzlicheren Frage „Wozu braucht man so etwas?". Die Software-Entwicklung ist im Vergleich zu vielen anderen Disziplinen sehr jung. Im Jahr 1958 wurde der Begriff „Software" das erste Mal in seinem heute gebräuchlichen Sinn verwendet *[1]*. Gegenwärtig ist also erst die dritte Generation von Software-Entwicklern tätig. Tatsächlich ist diese Disziplin so jung, dass derzeit noch kaum übergreifende Standards bestehen. Damit sind nicht die unzähligen IEEE-Normen, ISOC-RFCs oder Java-JSRs gemeint, sondern allgemein anerkannte, verbindliche Vorgaben für das Erstellen einer aus vielen Komponenten bestehenden Anwendung.

Wie bereits gesagt, Software-Entwickler arbeiten in einer jungen Disziplin und sie haben sich – bildlich gesprochen – bereits von improvisierten Holzverschlägen zu Konstruktionen vorgearbeitet, die gemäß einer reflektiert-übergeordneten Architektur gestaltet sind. Es setzt sich das Bewusstsein durch, dass nicht jedes Software-Projekt gleichsam das Rad neu erfinden muss; es gibt Standard-Lösungen für Standard-Probleme. An dieser Stelle setzen Entwurfsmuster an. Bei ihnen handelt es sich um Werkzeuge, die Software-Entwickler nutzen können, um wiederkehrende Probleme auf eine bewährte Art und Weise zu lösen. Qualitätskriterien wie Wiederverwendbarkeit, Änderbarkeit und Verständlichkeit stehen dabei im Vordergrund *[2]*. Sie sind quasi der Versuch, eine Baustatik für objektorientierte Software zu etablieren.

Erzeugungsmuster

Die GoF unterteilt die Entwurfsmuster in drei Kategorien: Erzeugungs-, Struktur- und Verhaltensmuster. Erzeugungsmuster beschäftigen sich mit der Konstruktion von Objekten. Sie verstecken die Erzeugungsmechanismen von Objekten hinter abstrakten Fassaden und erlauben so eine Nutzung, die entkoppelt ist von konkreten Implementierungen.

Hinter einer „abstrakten Fabrik" steht die Idee, konkrete Erzeugungs-Mechanismen (als „Fabriken" bezeichnet) für Objekte, die sich in der Struktur gleichen, also das gleiche Interface implementieren oder eine gemeinsame Oberklasse besitzen, zusammenzufassen und über eine einheitliche Schnittstelle verfügbar zu machen. Dabei wissen Aufrufende der abstrakten Fabrik nicht, welche konkrete Fabrik letztlich genutzt wird, um ein Objekt zu erzeugen. Darüber hinaus hat die abstrakte Fabrik nur Informationen zur allgemeinen Struktur des erstellten Objekts (Interface oder Oberklasse). Es wird gewissermaßen ein Fahrzeug ausgeliefert, ohne dass angegeben wird, ob es sich um einen Fabia, A8 oder eine E-Klasse handelt.

Ein wesentlicher Vorteil dieses Entwurfsmusters ist die Möglichkeit, konkrete Fabriken zu ergänzen oder auszuwechseln, ohne andere Komponenten der Software dafür anfassen zu müssen. Dadurch wird dem unschätzbar wertvollen Open-Closed-Prinzip Rechnung getragen, das besagt, dass „eine Klasse offen für Erweiterungen sein muss, jedoch geschlossen gegenüber Modifikationen" *[3]*. Anders ausgedrückt: Wenn einer Komponente Verhalten hinzugefügt werden soll, darf es dazu nicht nötig sein, bereits vorhandenes Verhalten zu ändern.

Wer schon einmal in der Situation war, ein kompliziertes (vielleicht auch noch ungetestetes) Stück Code ändern zu müssen, das vor langer Zeit geschrieben wurde, kennt die Gefahr, dass die Änderung das Kartenhaus der Gesamt-Software zusammenbrechen lässt. Eine konsequente Anwendung des Open-Closed-Prinzips reduziert diese Gefahr deutlich. Dieses Prinzip wird uns im Verlauf des Artikels noch wiederholt begegnen. Im Folgenden werden konkrete Implementierungen mit und ohne Lambda-Ausdrücke vorgestellt.

Klassisch

Listing 1 zeigt eine Beispiel-Implementierung für eine abstrakte Autofabrik. In der aufrufenden Komponente wird lediglich angegeben, welches Modell gebaut werden soll. Die Entscheidung darüber, welche konkrete Fabrik für diese Aufgabe gewählt wird, übernimmt die abstrakte Fabrik. Das Ergebnis ist ein allgemeines Auto, das erst im Rahmen der weiteren Verwendung (Log-Ausgabe der konkreten Attribute) die inneren Eigenschaften preisgibt.

Die konkreten Fabriken sind dabei Implementierungen eines allgemeinen Auto-Fabrik-Interface, die in der abstrakten Fabrik regist-

```
public static void main(String[] args) {
     AbstractFactoryClassic classicFactory = new AbstractFactoryClassic();
     Car fabia = classicFactory.getCarFactoryByModel(Model.FABIA).assemble();
     System.out.println(fabia.toJson());
     // Assembling Fabia...
     // {"brand":"Skoda","model":"Fabia","ps":54}

     Car a8 = classicFactory.getCarFactoryByModel(Model.A8).assemble();
     System.out.println(a8.toJson());
     // Assembling A8...
     // {"brand":"Audi","model":"A8","ps":190}

     Car eKlasse = classicFactory.getCarFactoryByModel(Model.EKLASSE).assemble();
     System.out.println(eKlasse.toJson());
     // Assembling E-Klasse...
     // {"brand":"Mercedes","model":"E-Klasse","ps":110}
  }
```

Listing 1: Nutzung der abstrakten Fabrik

```
public AbstractFactoryClassic() {
     carFactoryRegistry.put(Model.FABIA, new FabiaFactory());
     carFactoryRegistry.put(Model.A8, new A8Factory());
     carFactoryRegistry.put(Model.EKLASSE, new EKlasseFactory());
  }
```

Listing 2 : Registrierung von konkreten Fabriken

riert werden (in *Listing 2* als Beispiel unter Verwendung einer Map). *Listing 3* zeigt das allgemeine Interface und ein Beispiel für eine konkrete Implementierung.

Man benötigt also für jede Fabrik eine eigene Implementierungsklasse *(siehe Listing 4)*. Nun ist zu klären, ob Lambdas die Umsetzung des Entwurfsmusters „Abstrakte Fabrik" vereinfachen und/oder verbessern.

Lambda

Konkrete Fabriken können, statt mit eigenen Klassen, als Lambda-Ausdrücke umgesetzt werden. Dazu ist zunächst ein Functional-Interface zu definieren, das die allgemeine Struktur der Erzeugung vorgibt *(siehe Listing 5)*.

Per Definition ist jedes Interface, das lediglich eine Methode definiert, ein Functional-Interface (die Annotation „@FunctionalInterface" ist optional). Das in *Listing 3* dargestellte Interface erfüllte diese Voraussetzung bereits, allerdings ermöglicht die Extendierung des Supplier-Interface die Nutzung im Rahmen einiger Standard-JDK8-APIs. Da die „get()"-Methode des Supplier-Interface in *Listing 6* in Form einer Default-Methode überschrieben wird, ist dadurch die Functional-Interface-Voraussetzung nicht verletzt. So muss Code, in dem das „CarFactory"-Interface bereits genutzt wird, nicht geändert werden. Es wird lediglich die Möglichkeit ergänzt, es auch als Supplier zu verwenden – ganz im Sinne des Open-Closed-Prinzips. Nun können Lambda-Factories in der abstrakten Fabrik registriert werden.

Die Nutzung der „AbstractFactoryLambda" ist identisch zur klassischen, in *Listing 1* dargestellten Art und Weise. Neben schlankerem Code ergeben sich dadurch noch weitere Vorteile, die *Listing 7* zeigt.Durch die abstrakte Fabrik bereitgestellte Lambda-Fabriken lassen sich im JDK8-Stream-API nutzen, um auf einfache Art und Weise mehrere Autos zu produzieren. Daneben ist beispielsweise auch eine Verwendung als Fallback-Lösung beim Einsatz von Optionals sinnvoll.

In der „orElseGet()"-Methode lässt sich mithilfe der abstrakten Fabrik eine Alternative im Falle der Abwesenheit eines Autos generieren. Dabei wird einer der großen Vorteile von Lambdas genutzt: Die Objekt-Repräsentation des Lambda-Ausdrucks wird erst zur Laufzeit (mithilfe der mit Java 7 eingeführten ByteCode-Instruktion „invokedynamic") instanziiert und zwar nur dann, wenn sie wirklich notwendig ist (der optionale Wert also nicht vorhanden ist). Dadurch entsteht kein Aufwand für die Erzeugung des Fallback-Objekts im Falle des Vorhandenseins des Optional-Inhalts.

Die Variante, die ohne Lambdas arbeitet („orElse()"), führt den darin übergebenen Code hingegen immer sofort aus – unabhängig davon, ob das Ergebnis tatsächlich benötigt wird. Abgesehen von der Einsparung von Implementierungsklassen profitiert die Nutzung von Lambdas im Kontext der abstrakten Fabrik also von Synergie-Effekten mit JDK8-APIs.

```
public interface CarFactoryClassic<T extends Car> {
     T assemble();
  }
```

Listing 3 : Interface für konkrete Fabriken

```
  public class EKlasseFactory implements
  CarFactoryClassic<EKlasse> {
     @Override
     public EKlasse assemble() {
        return new EKlasse("Mercedes", "E-Klasse",
110);
     }
  }
```

Listing 4 : Beispiel-Implementierung einer Autofabrik: klassisch

```
@FunctionalInterface
  public interface CarFactoryLambda<T extends Car>
extends Supplier<T> {

     T assemble();

     @Override
     default T get() {
        return assemble();
     }
  }
```

Listing 5 : Functional-Interface für konkrete Fabriken

```
public AbstractFactoryLambda() {
     carFactoryRegistry.put(Model.FABIA, () -> new Fabia("Skoda", "Fabia", 52));
     carFactoryRegistry.put(Model.A8, () -> new A8("Audi", "A8", 190));
     carFactoryRegistry.put(Model.EKLASSE, () -> new EKlasse("Mercedes", "E-Klasse", 110));

  }
```

Listing 6 : Registrierung von Lambda-Factories

```
System.out.println("\nAssemble 3 Fabias in a row:");
  Stream.generate(abstractLambdaFactory.getCarFactoryByModel(Model.FABIA))
        .limit(3)
        .map(Car::toJson)
        .forEach(System.out::println);
  // Assemble 3 Fabias in a row:
  // Assembling Fabia...
  // {"brand":"Skoda","model":"Fabia","ps":52}
  // Assembling Fabia...
  // {"brand":"Skoda","model":"Fabia","ps":52}
  // Assembling Fabia...
  // {"brand":"Skoda","model":"Fabia","ps":52}

  Car absent = null;
  Car present = Optional.ofNullable(absent)
        .orElseGet(abstractLambdaFactory.getCarFactoryByModel(Model.A8));
  System.out.println(present.toJson());
  // Assembling A8...
  // {"brand":"Audi","model":"A8","ps":190}
```

Listing 7 : Nutzung einer Lambda-Factory in der JDK8-API

Erbauer

Abstrakte Fabriken sind nützlich in Szenarien, in denen es eine überschaubare Anzahl möglicher Ausprägungen von konkreten Objekten gibt. Wenn jedoch kaum gemeinsame Eigenschaften von Objektgruppen gefunden werden können oder jedes Objekt in seiner Beschaffenheit sogar einzigartig ist, ist es nicht sinnvoll beziehungsweise sogar unmöglich, ihre Erzeugung in Fabriken zu vereinheitlichen. In diesem Fall kann man zum Beispiel auf die Erzeugung per Konstruktor zurückgreifen. Wenn nun aber die Anzahl der Attribute eines Objektes sehr groß ist und davon einige optional sind beziehungsweise nur in Kombination mit bestimmten anderen benötigt werden, können die Konstruktoren beziehungsweise deren Anzahl (mit jeweils unterschiedlichen Signaturen) sehr unübersichtlich werden.

In solch einem Fall kommt das Erbauer-Entwurfsmusters zur Anwendung. Dabei werden die Parameter, die bei der Nutzung von Konstruktoren gleichzeitig übergeben werden, sequenziell eingefordert. Häufig wird es auch genutzt, um ganze Objekt-Bäume zu erzeugen.

Die grundlegende Funktionalität des Erbauer-Entwurfsmusters kann überdies durch die Nutzung eines Fluent-API um Semantik angereichert werden. Ein SQL-Ausdruck hat beispielsweise immer eine bestimmte sinnvolle Reihenfolge von Befehlen („select" – „from" – „where"). Ein Standard-Erbauer definiert jedoch keine Einschränkungen bezüglich der Reihenfolge der übergebenen Parameter. Hier können Fluent-Interfaces genutzt werden, um einen spezifischen Fluss (inklusive verschiedener Flussarme) von Parametern einzufordern und so die Idee eines Fluent-API umzusetzen. Sie schaffen somit die Voraussetzung für die Erstellung von Domänen-spezifischen Sprachen (Domain-Specific-Language, DSL) *[4]*. Dieser Umstand wird im folgenden Beispiel genutzt, um eine Pizzeria-Sprache zu definieren.

Klassisch

Listing 8 und *Listing 9* zeigen die Nutzung und Implementierung eines Pizza-Erbauers. Bis zu dem Punkt, an dem das zusammenzusetzende Objekt fertiggestellt ist und zurückgegeben wird, fügen die einzelnen Methoden diesem jeweils einen neuen Parameter hinzu und geben den Erbauer selbst zurück, damit daraufhin die nächste Methode zum Hinzufügen eines Parameters aufgerufen werden kann. So wird Stück für Stück das Objekt fertiggestellt.

Um einen festen Fluss von Parametern zu gewährleisten, muss der Pizza-Erbauer verschiedene (Fluent-)Interfaces implementieren. Diese sind im *Listing 10* dargestellt. Um dabei eine bestimmte Reihenfolge an Methoden-Aufrufen zu gewährleisten, müssen weitere Implementierungen hintereinandergeschaltet werden. Dies ist beispielhaft an dem „OnGoingPizzaBuilder" in *Listing 9* zu sehen.

Das Bauen einer Pizza kann nur mit der „withDough()"-Methode des „PizzaBuilder" beginnen, da dieser keine anderen Methoden besitzt. Der zurückgegebene „OnGoingPizzaBuilder" gibt daraufhin die weiteren Möglichkeiten zum Zusammensetzen der Pizza vor. Möchte man diese Möglichkeiten noch weiter einschränken (etwa wenn nach dem Teig die Wahl einer Käse-Art erzwungen werden soll), so sind weitere „OnGoingPizzaBuilder"-Implementierungen notwendig.

Lambda

Die Anzahl und Reihenfolge dieser Implementierungen kann sehr unübersichtlich sein. Hier können Lambdas die Komplexität deutlich reduzieren, denn sie bieten die Nutzung eines gängigen Prinzips der funktionalen Programmierung: das „Currying" (benannt nach dem Mathematiker Haskell Curry). Es bezeichnet die

```
public static void main(String[] args) {
     Pizza pizza = ClassicPizzeria
            .makePizza()
            .withDough(wheat())
            .andCheese(mozzarella())
            .andFirstTopping(broccoli())
            .andSecondTopping(mushrooms());

     System.out.println(pizza.toJson());
     // {"dough":wheat,"toppings":[broccoli,
mushrooms],"cheese":mozzarella}
  }

  public static PizzaBuilder makePizza() {
     return new PizzaBuilder();
  }
```

Listing 8 : Nutzung Pizza-Erbauer: klassisch

```
public class PizzaBuilder {

     Pizza pizza;

     public PizzaBuilder() {
        this.pizza = new Pizza();
     }

     public OnGoingPizzaBuilder withDough(Dough dough) {
        this.pizza.setDough(dough);
        return new OnGoingPizzaBuilder(pizza);
     }

     public class OnGoingPizzaBuilder implements CheeseBuilder, FirstToppingBuilder, SecondToppingBuilder {

        private Pizza pizza;

        public OnGoingPizzaBuilder(Pizza pizza) {
           this.pizza = pizza;
        }

        @Override
        public FirstToppingBuilder andCheese(Cheese cheese) {
           this.pizza.setCheese(cheese);
           return this;
        }

        @Override
        public Pizza andSecondTopping(Topping topping) {
           this.pizza.getToppings().add(topping);
           return pizza;
        }

        @Override
        public SecondToppingBuilder andFirstTopping(Topping topping) {
           this.pizza.getToppings().add(topping);
           return this;
        }
     }

  }
```

Listing 9 : Implementierung des Pizza-Erbauers: klassisch

```
public interface CheeseBuilder {
     FirstToppingBuilder andCheese(Cheese cheese);
  }

  public interface FirstToppingBuilder {
     SecondToppingBuilder andFirstTopping(Topping topping);
  }

  public interface SecondToppingBuilder {
     Pizza andSecondTopping(Topping topping);
  }
```

Listing 10 : Fluent-Interfaces

Umwandlung einer Funktion mit mehreren Parametern in eine Sequenz von Funktionen mit jeweils einem Parameter.

Voraussetzung für die Nutzung von Lambdas ist wiederum der Einsatz von Functional-Interfaces. Glücklicherweise erfüllen die bereits genutzten Interfaces *(siehe Listing 10)* dafür die Voraussetzungen, da sie jeweils lediglich eine Methode definieren. So ist weiter nur ein zusätzliches Interface zur Wahl des Teiges erforderlich *(siehe Listing 11)*.

Um den Pizza-Erbauer zu verwirklichen, sind keine zusätzlichen Implementierungsklassen mehr notwendig. Die Umsetzung des Erbauer-Entwurfsmusters vereinfacht sich so im Vergleich zu den vielen nötigen Implementierungsklassen für die Fluent-Interfaces in der klassischen Variante drastisch. Darüber hinaus wird eine klare Reihenfolge von Methodenaufrufen erzwungen: Jegliche Änderung (etwa der Aufruf von „andFirstTopping()" vor „andCheese()", was in der klassischen Variante möglich war) würde zu einem Compiler-Fehler führen.

Zusammenfassend lässt sich sagen, dass der Einsatz von Lambdas im Kontext des Erbauer-Entwurfsmusters zum einen Implementierungsklassen spart und zum anderen mithilfe des Currying das Erzwingen einer bestimmten Reihenfolge von Erbauer-Methoden vereinfacht. Das kommt der Verständlichkeit zugute und behebt so einen der Schwachpunkte der Implementierung des Erbauer-Entwurfsmusters.

```
public static void main(String[] args) {

     Pizza pizza = LambdaPizzeria
            .makePizza()
            .withDough(wheat())
            .andCheese(mozzarella())
            .andFirstTopping(broccoli())
            .andSecondTopping(mushrooms());

     System.out.println(pizza.toJson());
     // {"dough":wheat,"toppings":[broccoli, mushrooms],"cheese":mozzarella}

  }

  public static PizzaBuilderLambda makePizza() {
     // Currying:
     return dough -> cheese -> topping1 -> topping2 -> new Pizza(dough, cheese, topping1, topping2);
  }

  public interface PizzaBuilderLambda {
     CheeseBuilder withDough(Dough dough);
  }
```

Listing 11 : Implementierung und Nutzung des Pizza-Erbauers: Lambda

Verhaltensmuster

Verhaltensmuster befassen sich (wie der Name vermuten lässt) mit dem Verhalten von Objekten, beispielsweise der Interaktion zwischen Objekten sowie der Änder- und Austauschbarkeit von Verhalten zur Design- und Laufzeit. Da Verhalten über Funktionen abgebildet wird, bietet sich hier in besonderem Maße die Nutzung von Lambdas an. Einige Verhaltensmuster können dadurch sogar überflüssig werden.

Dies ließe sich etwa beim Strategie-Entwurfsmuster argumentieren. Die Idee hinter einer Strategie ist die Kapselung von Algorithmen in eine einheitliche Form (etwa über ein Interface), um sie dadurch (auch zur Laufzeit) austauschbar zu machen. Damit ist auch eine der Kern-Eigenschaften von Lambda-Ausdrücken beschrieben: dynamisch austauschbares Verhalten, gekapselt durch Functional-Interfaces.

Listing 12 zeigt die Anwendung des Strategie-Entwurfsmusters ohne Lambdas am Beispiel des Versuchs, einen Safe mithilfe verschiedener Hilfsmittel (Strategien) zu knacken. Die beiden Implementierungen unterscheiden sich also darin, dass die erste (Lockpick) erfolgreich ist, wenn der Safe ein Schloss hat, und die zweite (Dynamite), wenn die Stabilität des Safes einen gewissen Wert unterschreitet. Die Strategien können nun im Zuge des Raubs (englisch „heist") je nach Bedarf genutzt und ausgetauscht werden.

Wie *Listing 13* zeigt, lässt sich nach Bedarf jeweils von außen ein anderer Algorithmus übergeben, ohne dass dafür die innere Funktionsweise des Nutzers der Strategie verändert werden muss. Hier wird wieder dem Open-Closed-Prinzip Rechnung getragen, da so Verhalten ausgetauscht werden kann, ohne den umschließenden Code manipulieren zu müssen. Leider führt keine der beiden Strategien zum gewünschten Erfolg: Der Safe bleibt verschlossen.

Lambda

Wie bereits gesagt, handelt es sich bei diesem Szenario um ein Paradebeispiel für die Anwendung von Lambdas. Da die „SafeCrackingStrategy" wiederum ein Functional-Interface ist, sind keine

```
public interface SafeCrackingStrategy {
     void crackSafe(Safe safe);
  }

  public class Lockpick implements SafeCrackingStrategy {
     @Override
     public void crackSafe(Safe safe) {
        System.out.println("Trying to pick the lock...");
        if (safe.hasLock()) {
           safe.setOpen(true);
        }
     }
  }

  public class Dynamite implements SafeCrackingStrategy {
     @Override
     public void crackSafe(Safe safe) {
        System.out.println("\nBOOM!");
        if (safe.getSturdiness() <= 9000) {
           safe.setOpen(true);
        }
     }
  }
```

Listing 12 : Interface und Implementierungen für die Safe-Knacker-Strategie

weiteren Anpassungen nötig, es können direkt Lambda-Ausdrücke statt Instanzen der Implementierungsklassen übergeben werden *(siehe Listing 14)*.

Die Definition der „SafeCrackingStrategy" als Functional-Interface gewährleistet also auch eine Austauschbarkeit von klassischen Implementierungen und Lambda-Ausdrücken. Wenn stattdessen ein vom JDK mitgeliefertes Standard-Functional-Interface (wie „Function" oder „Consumer") verwendet wird, ist neben den einzelnen Implementierungen auch das „SafeCrackingStrategy"-Interface nicht erforderlich. Bei der Nutzung von Lambdas ist die explizite Anwendung des Strategie-Entwurfsmusters also weitgehend überflüssig, da es sich per Definition bei jedem Lambda-Ausdruck um eine austauschbare Strategie handelt.

Schablonen-Methode

In einer Situation, in der mehrere Varianten eines Algorithmus sich

```
public class SafeHeist {

     private SafeCrackingStrategy strategy;

     public void chooseStrategy(SafeCrackingStrategy
strategy) {
        this.strategy = strategy;
     }

     public void performHeist(Safe safe) {
        strategy.crackSafe(safe);
        System.out.println("Safe is open = " + safe.
isOpen());
     }

     public static void main(String[] args) {
        SafeHeist heist = new SafeHeist();
        Safe safeToBeCracked = new Safe();

        heist.chooseStrategy(new Lockpick());
        heist.performHeist(safeToBeCracked);

        heist.chooseStrategy(new Dynamite());
        heist.performHeist(safeToBeCracked);

        // Trying to pick the lock...
        // Safe is open = false
        //
        // BOOM!
        // Safe is open = false
     }
  }
```

Listing 13 : Nutzung der Strategien: klassisch

```
System.out.println("\nLet's make this easy!");
  heist.chooseStrategy(safe -> safe.setOpen(true));
  heist.performHeist(safeToBeCracked);

  // Let's make this easy!
  // Safe is open = true
```

Listing 14 : Strategy mit Lambda-Ausdruck

```
public abstract class Musician {

     public void performRehearsal() {
        arrive();
        prepare();
        rehearse();
        complain();
        rehearse();
        depart();
     }

     private void complain() {
        System.out.println("complaining about bass
player...");
     }

     private void depart() {
        System.out.println("driving home...");
        System.out.println("\n");
     }

     protected abstract void rehearse();

     protected abstract void prepare();

     private void arrive() {
        System.out.println("driving to rehearsal...");
     }
  }
```

Listing 15 : Abstrakte Oberklasse "Musiker"

```
public class GuitarPlayer extends Musician {
     @Override
     protected void rehearse() {
        System.out.println("playing guitar...");
     }

     @Override
     protected void prepare() {
        System.out.println("connecting guitar to
amp...");
     }
  }

  public class Singer extends Musician {
     @Override
     protected void rehearse() {
        System.out.println("singing...");
     }

     @Override
     protected void prepare() {
        System.out.println("turning on mic...");
     }
  }
```

Listing 16 : Gitarrist und Sänger

```
public static void main(String[] args) {
     List<Musician> band = Arrays.asList(new Guitar-
Player(), new Singer());

     for (Musician musician : band) {
        musician.performRehearsal();
     }

     // driving to rehearsal...
     // connecting guitar to amp...
     // playing guitar...
     // complaining about bass player...
     // playing guitar...
     // driving home...
     //
     //
     // driving to rehearsal...
     // turning on mic...
     // singing...
     // complaining about bass player...
     // singing...
     // driving home...
  }
```

Listing 17 : Bandprobe: klassisch

```
public class Musician {

     Runnable preparation;

     Runnable rehearsal;

     public Musician(Runnable preparation, Runnable
rehearsal) {
        this.preparation = preparation;
        this.rehearsal = rehearsal;
     }

     public void performRehearsal() {
        arrive();
        preparation.run();
        rehearsal.run();
        complain();
        rehearsal.run();
        depart();
     }
     // ...
  }
```

Listing 18 : Musiker mit Functional-Interface (Runnable)

```
public static void main(String[] args) {
     Musician guitarPlayer =
           new Musician(
                 () -> System.out.println("connecting guitar to amp..."),
                 () -> System.out.println("playing guitar..."));
     Musician singer =
           new Musician(
                 () -> System.out.println("turning on mic..."),
                 () -> System.out.println("singing..."));

     Stream.of(guitarPlayer, singer).forEach(Musician::performRehearsal);
     // driving to rehearsal...
     // connecting guitar to amp...
     // playing guitar...
     // complaining about bass player...
     // playing guitar...
     // driving home...
     //
     //
     // driving to rehearsal...
     // turning on mic...
     // singing...
     // complaining about bass player...
     // singing...
     // driving home...
  }
```

Listing 19 : Bandprobe: Lambda

im Ablauf sehr stark ähneln, kann das Entwurfsmuster „Schablonen-Methode" nützlich sein. Es ergänzt einen Algorithmus um variable Teilschritte, die nur in erbenden Klassen ausprogrammiert werden müssen, und ermöglicht so die konsequente Anwendung der DRY-Regel: Don't Repeat Yourself *[5]*.

Der gemeinschaftlich genutzte Teil des Algorithmus wird in einer abstrakten Oberklasse definiert. Die variablen Teilschritte werden über abstrakte Methoden abgebildet, die dann in den erbenden Klassen implementiert sind. Als Beispiel für „klassisch" dient hier eine Bandprobe. Sie ist für alle beteiligten Musiker ähnlich, aber nicht identisch:

1. Anfahrt
2. Vorbereitung
3. Probe
4. Beschwerde über den Bassisten
5. Fortführung der Probe
6. Rückfahrt

Schritt 1, 4 und 6 sind für alle Musiker (mit Ausnahme natürlich des Bassisten) gleich. Für diese Punkte lässt sich also eine abstrakte Oberklasse definieren *(siehe Listing 15)*.

Bei der Vorbereitung und der Durchführung der Probe unterscheiden sich die Tätigkeiten der Musiker jedoch. Anstatt dafür die gemeinschaftlichen Teile zu kopieren, müssen die konkreten Musiker lediglich von der Oberklasse erben und die abstrakten Methoden implementieren *(siehe Listing 16)*. So entsteht eine größtmögliche Wiederverwendung von Code *(siehe Listing 17)*.

Dieser Vorteil wird allerdings dadurch erkauft, dass die Struktur der Anwendung unübersichtlicher ist. So nützlich abstrakte Klassen und Methoden sind, so schwer verständlich sind sie oft auch (besonders für Personen, die den Code nicht selbst erstellt haben). Aus gutem Grund hat sich in den letzten Jahren die Grobregel „Delegation statt Vererbung" durchgesetzt *[6]*. Sie besagt, dass Aufgaben, die erbende Klassen übernehmen, stattdessen bevorzugt an referenzierte Komponenten delegiert werden sollten.

Lambda

Jetzt kommen wieder Lambda-Ausdrücke ins Spiel. Diese ermöglichen nämlich für das betrachtete Szenario den Verzicht auf Vererbung, ohne dabei die Flexibilität und Schlankheit der Schablonenmethode zu verlieren. Dafür delegiert der Musiker die variablen Teile der Probe an Functional-Interfaces beziehungsweise deren Implementierung in Form von Lambda-Ausdrücken *(siehe Listing 18)*.

Im Beispiel kann das „Runnable"-Interface genutzt werden, das keine Parameter entgegennimmt und nichts zurückgibt *(siehe Listing 19)*. Das gemeinsame Verhalten wird also weiterhin im Musiker gekapselt und die variablen Anteile als Lambda-Ausdrücke von außen übergeben. Die Vererbung und damit auch die Unterklassen von Musiker sind damit überflüssig und die Struktur des Codes vereinfacht.

Beobachter

Die Idee hinter dem Beobachter-Entwurfsmuster ist, dass Zustandsveränderungen in Form von Events bekanntgemacht werden. Um über das Eintreten eines Events benachrichtigt zu werden, lassen sich Beobachter in einem zentralen Verteiler (auch „Subjekt" genannt) registrieren. Sobald das Event ausgelöst wird, können die Beobachter darauf mit individuellem Verhalten reagieren. Dies gewährleistet eine lose Kopplung von Event zu Beobachter und eine leichte Austauschbarkeit von Verhalten.

Eingangs wurde bereits erwähnt, dass einige Entwurfsmuster die Grundlage ganzer Java-Frameworks sind und somit unbewusst von vielen Entwicklern genutzt werden. Dies trifft insbesondere auf das Beobachter-Entwurfsmuster zu, das eines der Stützpfeiler des Reactive-Programming-Paradigmas ist und damit maßgeblichen Einfluss auf RxJava und ähnliche Frameworks hat.

```
public class Operator {

     private List<CallCenterAgentListener> agents = new ArrayList<>();

     public void distributeCall(Call call) {

        for (CallCenterAgentListener agent : agents) {
           agent.acceptCall(call);
        }

     }

     public void registerAgent(CallCenterAgentListener agent) {
        this.agents.add(agent);
     }
  }
```

Listing 20 : Callcenter-Operator (Subjekt)

Betrachten wir zunächst wieder die klassische Variante. Als Beispiel dient hier ein fiktives Callcenter *(siehe Listing 20)*. Das Subjekt ist ein Operator, der Telefonanrufe verteilt. Beim ihm können sich Callcenter-Agenten (als Beobachter) registrieren, die dann benachrichtigt werden, wenn das Event „Anruf geht ein" stattfindet. Wie die einzelnen Agenten auf einen Anruf reagieren, ist diesen selbst überlassen. Um als Beobachter vom Operator anerkannt zu werden und sich registrieren zu können, müssen Callcenter-Agenten ein bestimmtes Interface implementieren *(siehe Listing 21)*.

Damit sie über einen Anruf informiert werden, müssen sich Kündigungsmanagement, Kunden-Support und Upselling beim Operator registrieren. Sobald ein Anruf eintrifft, wird dieser vom Operator verteilt (via „distributeCall()") und so die Reaktion der Agenten darauf ausgelöst *(siehe Listing 22)*.

In diesem Beispiel suggerieren die Callcenter-Agenten, bei Beobachtern handele es sich in erster Linie um Zustände (in Form von Objekten), die durch eine Aufruf-Kette gereicht werden. In Wirklichkeit steht das Weiterreichen von Verhalten im Vordergrund. Die Implementierungsklassen um das Verhalten herum sind eine Notwendigkeit aus der Zeit, in der Java noch keine Übergabe von Funktionen als Parameter (also Lambdas) erlaubte.

Lambda

Beim Einsatz von Lambda-Ausdrücken kann nun auf die Implementierungsklassen verzichtet werden. Hier ist wieder der Umstand zu nutzen, dass es sich bei „CallCenterAgentListener" um ein Functional-Interface handelt, sodass dem Operator direkt Lambda-Ausdrücke übergeben werden können *(siehe Listing 23)*.

Durch den Verzicht auf Implementierungsklassen muss erst zum Zeitpunkt der Registrierung entschieden werden, welches Verhalten das beobachtete Event auslösen soll. Ein weiterer Vorteil ergibt sich aus einem Szenario, in dem das Beobachter-Entwurfsmuster oft eingesetzt wird: mehrere nebenläufige Threads. Darin ist jeder Zustand aufgrund der konkurrierenden Zugriffe ein Risiko und sollte daher vermieden werden. Hier haben Lambdas gegenüber anonymen inneren und implementierenden Klassen die Nase vorn, da die Zustände (sprich „Felder") von Letzteren über ihren gesamten Lebenszyklus hinweg geändert, während die Felder in Lambda-Ausdrücken nur während der Deklaration manipuliert werden können.

```
public interface CallCenterAgentListener {
     void acceptCall(Call call);
  }

  public class CustomerRetention implements CallCenterAgentListener {
     @Override
     public void acceptCall(Call call) {
        System.out.println("\"Please stay with us!\"");
     }
  }

  public class CustomerSupport implements CallCenterAgentListener {
     @Override
     public void acceptCall(Call call) {
        System.out.println("\"Have you tried turning it off and on again?\"");
     }
  }

  public class Upselling implements CallCenterAgentListener {
     @Override
     public void acceptCall(Call call) {
        System.out.println("\"Try this new feature for only 10$ a month!\"");
     }
  }
```

Listing 21 : Callcenter-Agenten

```
public static void main(String[] args) {

     Operator operator = new Operator();

     operator.registerAgent(call -> System.out.println("\"Try this new feature for only 10$ a month!\""));
     operator.registerAgent(call -> System.out.println("\"Have you tried turning it off and on again?\""));
     operator.registerAgent(call -> System.out.println("\"Please stay with us!\""));

     operator.distributeCall(new Call());
     // "Try this new feature for only 10$ a month!"
     // "Have you tried turning it off and on again?"
     // "Please stay with us!"

  }
```

Listing 22 : Beobachter: klassisch

```
public static void main(String[] args) {
     Operator operator = new Operator();

     operator.registerAgent(new Upselling());
     operator.registerAgent(new CustomerSupport());
     operator.registerAgent(new CustomerRetention());

     operator.distributeCall(new Call());
     // "Try this new feature for only 10$ a month!"
     // "Have you tried turning it off and on again?"
     // "Please stay with us!"
  }
```

Listing 23 : Beobachter: Lambda

Beiden Varianten ist gemein, dass sie Zustände des umgebenden Kontexts nur manipulieren können, wenn es sich um effektiv finale Variablen handelt, was allerdings nur erzwungen werden kann, wenn sie immutable sind.

Fazit

Es gibt noch viele andere Entwurfsmuster der Gang-of-Four, die von Lambda-Ausdrücken profitieren würden. Dies gilt in besonderem Maße für Verhaltensmuster, die Probleme lösen sollen, die seit der Einführung von Lambda-Ausdrücken in Java viel leichter zu beseitigen oder gar nicht mehr vorhanden sind. Für diesen Artikel musste verständlicherweise eine Auswahl getroffen werden.

Es ist vermutlich aufgefallen, dass eine ganze Kategorie von Entwurfsmustern aus dem GoF-Basismodell fehlt: die Strukturmuster. Der Grund dafür liegt in ihrer Natur: Sie basieren mehr noch als die anderen Kategorien auf den eingangs erwähnten Grundparadigmen der Objekt-Orientierung (Polymorphie, Vererbung und Kapselung). Daher findet sich für Lambda-Ausdrücke und ihre funktionale Natur nur schwer eine gewinnbringende Anwendungsmöglichkeit.

Wie dieser Artikel verdeutlicht, haben die Entwurfsmuster der Gang-of-Four auch in Zeiten der Einführung von funktionaler in die objektorientierte Programmierung nicht an Relevanz und Nützlichkeit verloren. Der Einsatz von Lambda-Ausdrücken macht einen wesentlichen Anteil der Entwurfsmuster schlanker, effizienter und eröffnet weitere neue Anwendungsmöglichkeiten. Darüber hinaus sind die klassischen Implementierungen der Entwurfsmuster mit den jeweiligen Lambda-Varianten kompatibel und austauschbar, wenn Functional-Interfaces sinnvoll definiert und genutzt werden. Dadurch sind bereits vorhandene Umsetzungen der einzelnen Entwurfsmuster nicht komplett zu überarbeiten.

Vielen Dank an Tobias Voss („@tobiaslvoss"), der beim Erstellen dieses Artikels als Sparringspartner fungiert hat.

Quellen

[1] John W. Tukey, The Teaching of Concrete Mathematics, The American Mathematical Monthly, Vol. 65, no. 1 (Januar 1958), Seite 2
[2] Erich Gamma, Richard Helm, Ralph E. Johnson, John Vlissides, Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software, Addison Wesley, 2004, Vorwort
[3] Bertrand Meyer, Object Oriented Software Construction, Prentice Hall, 1988, Seiten 57–61
[4] Martin Fowler: *https://martinfowler.com/bliki/FluentInterface.html*
[5] Andrew Hunt, David Thomas, The Pragmatic Programmer, The Pragmatic Bookshelf, 1999, Seite 27
[6] Robert C. Martin: *http://collaboration.cmc.ec.gc.ca/science/rpn/biblio/ddj/Website/articles/CUJ/2001/cexp1908/martin/martin.htm*

Christian Nockemann
christian.nockemann@viadee.de

Christian Nockemann ist IT-Berater und Software-Entwickler bei der viadee IT-Unternehmensberatung. Sein Fokus liegt auf Design und Umsetzung von individuellen Enterprise-Anwendungen auf Basis von Java/Spring mit einem besonderen Schwerpunkt auf der Verbesserung von Code-Qualität (Testbarkeit, Clean-Code, Nutzung von Entwurfsmustern etc.).