Design Patterns mit Java 8

Michael Inden

Freiberuflicher Consultant, Buchautor und Trainer

Agenda

- Builder
- Template Method
- Factory Method / Factory
- Strategy
- NEW: Execute Around Pattern

Builder

Builder herkömmlich als DTO mit Explaing Methods

```
public static final class PizzaBuilder
{
    // Defaultwerte, gelten wenn keine korrespondierende Methode aufgerufen wird
    boolean mitSalami         = false;
    boolean mitExtraSardellen = false;
    String  info              = "";
    Size    size              = Size.MEDIUM;

    public PizzaBuilder mitSalami()
    {
        this.mitSalami = true;
        return this;
    }

    public PizzaBuilder small()
    {
        this.size = Size.SMALL;
        return this;
    }

    public Pizza create()
    {
        return new Pizza(this);
    }
```

...

Builder im Einsatz

- Einsatz

```
public static void main(final String[] args)
{
    final PizzaBuilder builder = new PizzaBuilder();
    builder.mitExtraSardellen();
    System.out.println("Normale Pizza mit extra Sardellen:\n" + builder.create());


    final PizzaBuilder builder2 = new PizzaBuilder();
    builder2.mitSalami().small().bitteBeachten("Ohne Mais!");
    System.out.println("Kleine Salami-Pizza ohne Mais:\n" + builder2.create());
}
```

- Sehr gute Lesbarkeit und Verständlichkeit
- Kompakte Schreibweise möglich
- Refactoring-sicher
- Optionale Attribute können sinnvoll vorbelegt und verarbeitet werden

Builder mit Consumer statt spezifischer Methoden

```
public class PizzaBuilder
{
    enum Size {
        SMALL, MEDIUM, LARGE
    }

    public boolean mitSalami          = false;
    public boolean mitExtraSardellen = false;
    public String  info               = "";
    public Size    size               = Size.MEDIUM;

    public PizzaBuilder with(final Consumer<PizzaBuilder> builderFunction)
    {
        builderFunction.accept(this);
        return this;
    }

    public Pizza build()
    {
        return new Pizza(this);
    }
}
```

Builder mit Consumer statt spezifischer Methoden

- Einsatz mit Lambdas

```
public static void main(String[] args)
{
    Pizza pizza = new PizzaBuilder().
                        with($ -> $.info = "Kein Mais").
                        with($ -> $.size = Size.MEDIUM).
                        with($ -> $.mitSalami = true).build();


    System.out.println(pizza);
}
```

- Für einfache Builder okay
- Lesbarkeit schlechter, Spezialsyntax bei Lambdas
- Problematisch: direkter Zugriff auf Attribute, aber nicht Refactoring-sicher

Template Method

Template Method herkömmlich

```
public final void templateMethod()
{
    first();
    step1();
    step2();
    step3();
    hook();
    last();
}

abstract protected void first();

abstract protected void last();

void step1()
{
    System.out.println("step1");
}
```

```
// ...
```

Template Method herkömmlich mit Ableitung und Überschreiben

```
public static void main(String[] args)
{
    TemplateMethodExample tme = new TemplateMethodExample()
    {
        @Override
        protected void first()
        {
            System.out.println("FIRST");
        }

        @Override
        protected void last()
        {
            System.out.println("LAST");
        }
    };
    tme.templateMethod();
}
```

```
FIRST
step1
step2
step3
LAST
```

Template Method mit Lambdas als Parameter

```
public final void templateMethod(Runnable first, Runnable last)
{
    first.run();
    step1();
    step2();
    step3();
    hook();
    last.run();
}


void step1()
{
    System.out.println("step1");
}

// ...
```

Template Method neu mit Lambda als Parameter

```
public static void main(String[] args)
{
    TemplateMethodExample tme = new TemplateMethodExample();

    tme.templateMethod(() -> System.out.println("FIRST"),
                       () -> System.out.println("LAST"));
}
```

```
FIRST
step1
step2
step3
LAST
```

Factory (Method)

Factory Method herkömmlich

```
// Problems: not easily extensible
public static WebDriver getDriver(DriverType type)
{
    WebDriver driver;

    switch (type)
    {
        case CHROME:
            System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");
            driver= new ChromeDriver();
            break;
        case FIREFOX:
            System.setProperty("webdriver.gecko.driver",
                               "/Users/username/Downloads/geckodriver");
            driver = new FirefoxDriver();
            break;
        default:
            throw new IllegalArgumentException("Unsupported driver type");
    }
    return driver;
}
```

Factory Method mit Supplier

```
public enum DriverType
{
    CHROME, FIREFOX, SAFARI, IE;
}

// Factory Method, Variante mit Optional später
public static final WebDriver getDriver(DriverType type)
{
    if (DriverType.CHROME == type)
        return chromeDriverSupplier.get();
    if (DriverType.FIREFOX == type)
        return firefoxDriverSupplier.get();

    throw new IllegalArgumentException("Unsupported driver type");
}
```

Factory Method mit Supplier

```
Supplier<WebDriver> chromeDriverSupplier = () ->
{
    System.setProperty("webdriver.chrome.driver", "/path/to/chromedriver");

    return new ChromeDriver();
};

Supplier<WebDriver> firefoxDriverSupplier = () ->
{
    System.setProperty("webdriver.gecko.driver",
                       "/Users/username/Downloads/geckodriver");

    return new FirefoxDriver();
};
```

Factory mit Supplier (Kür)

```
public static class DriverFactory
{
    private static final Map<DriverType, Supplier<WebDriver>> driverMap =
                         new HashMap<>();

    static
    {
        driverMap.put(DriverType.CHROME, chromeDriverSupplier);
        driverMap.put(DriverType.FIREFOX, firefoxDriverSupplier);
    }

    public static final Optional<WebDriver> getDriver(DriverType type)
    {
        return Optional.ofNullable(driverMap.get(type).get());
    }

    void registerDriver(DriverType type, Supplier<WebDriver>> driver)
    {
       driverMap.put(...);
    }
}
```

Strategy

«Strategy Pattern» mit Predicate und Streams

```
public static void main(String[] args)
{
    List<String> names1 = List.of("Tim", "Tom", "Peter", "Mike", "Michael");
    Stream<String> names2 = Stream.of("Tim", "Tom", "Peter", "Mike", "Michael");
    Stream<String> names3 = Stream.of("Tim", "Tom", "Peter", "Mike", "Michael");

    // Strategy by Definition Predicates
    Predicate<String> startsWithT = str -> str.startsWith("T");
    Predicate<String> moreThan4Chars = str -> str.length() > 4;
    Predicate<String> moreThan6Chars = longerThan(6);

    names1.stream().filter(startsWithT).forEach(System.out::println);
    names2.filter(moreThan4Chars).forEach(System.out::println);
    names3.filter(moreThan6Chars).forEach(System.out::println);
}

// Trick dynamische Parametrierung
static Predicate<String> longerThan(int lowerBound)
{
    return str -> str.length() > lowerBound;
}
```

```
Tim
Tom
Peter
Michael
Michael
```

Execute Around Pattern

Typische Ressource, die Freigabe erfordert

```
class Resource
{
    public Resource()
    {
        System.out.println("created");
    }

    public void op1() throws IOException
    {
        System.out.println("op1");
    }

    public void op2()
    {
        System.out.println("op2");
    }

    public void close()
    {
        System.out.println("close");
    }
}
```

Resource

- Einsatz oftmals fehlerhaft, entweder ohne `close()` oder Problem im Exception-Fall

```
public static void main(String[] args)
{
    Resource rs = new Resource();
    try
    {
        rs.op1();
        rs.op2();
        // rs.close();
    }
    catch (IOException ioe)
    {
        handleIoException(ioe);
    }
    finally
    {
        // rs.close();
    }
}
```

Resource

- Schön wäre es, einen einfachen Aufruf zu machen, der an alles «denkt»

```
Resource rs2 = new Resource();
withClose(rs2, res -> {
    try
    {
        res.op1();
        res.op2();
    }
    catch (IOException ioe)
    {
        handleIoException(ioe);
    }
});
```

Java Locks

- **Wie ist es nochmal richtig?**

```
final Lock lock = ...

lock.lock();
try
{
    // access the resource
    // protected by this lock
}
finally
{
    lock.unlock();
}
```

```
final Lock lock = ...

try
{
    lock.lock();

    // access the resource
    // protected by this lock
}
finally
{
    lock.unlock();
}
```

- **Was ist der Unterschied???**

Java Locks

- Wie ist es nochmal richtig?

```
final Lock lock = ...

lock.lock();
try
{
    // access the resource
    // protected by this lock
}
finally
{
    lock.unlock();
}
```

```
final Lock lock = ...

try
{
    lock.lock();

    // access the resource
    // protected by this lock
}
finally
{
    lock.unlock();
}
```

- Im linken Fall wird der Lock nur dann freigegeben, wenn wir ihn erhalten haben, dann aber egal, ob es im Block zu Problemen kam oder nicht
- Im rechten Fall wird der Lock auf jeden Fall freigegeben, auch wenn man ihn nicht erhalten hat!

Thank You