Elias Anton 19IN-B

Best Practices für Libraries – Cheatsheet

2 Grundmuster:

- „broad and shallow“
  - viele public-Methoden, die alle nicht so umfangreich sind
  - Javadoc
  - hat möglichst keine oder wenige Abhängigkeiten
- „narrow and deep“
  - wenige aber dafür umfangreichere Methoden
  - meist auf höherem Level zu dokumentieren

→ Ich denke „broad and shallow“ ist für unser Projekt sinnvoll, da es alles übersichtlicher hält und am Ende auch einfacher erweitert werden kann. Zudem ist es mit guter Dokumentation deutlich einfacher für verschiedene Entwickler, die Arbeit von Anderen nachzuvollziehen.

Design:

- Library **muss** hohe Qualität haben und verlässlich sein
- Am Anfang Umfang abstecken:
  - Was soll die Library (nicht) leisten?
  - Was muss die nutzende Anwendung selbst implementieren?
  - Welche Eingaben und Ausgaben gibt es?
- keine/wenige Dependencies (z.B. andere Libraries) haben
  - wenn man nur einen kleinen Teil einer anderen Library braucht, den Teil entweder selbst implementieren oder mit **deutlicher Markierung** kopieren
- Wie werden die Ausgaben genutzt? → Was muss dann bedacht werden?
- Paket-Sichtbarkeit nutzen (nicht zu vieles public haben)

API:

- bietet Konstruktoren und Methoden
- API schon mit den Anforderungen eventueller Benutzer im Kopf erstellen
- Konstruktoren sollten selbsterklärend sein
- Nicht mehr als drei oder vier Parameter in Konstruktoren
- evtl. alternative Konstruktoren anbieten
- möglichst alle veränderlichen Parameter durch Konstruktoren annehmen
  - essenzielle Parameter nicht über Methoden einfügen
- wo möglich, mit unveränderlichen Objekten zwischen Anwendung und Library kommunizieren
- Konsistente Benennung von API-Methoden
  - Abkürzungen nur, wenn man sie immer benutzt
  - Der Nutzer sollte nur anhand des Namens der Funktion die gewünschte Funktionalität finden können
- keine Aufrufe und Rückgaben mit „null“ zulassen (führt zu klarerem Code)

Dokumentation:

- Dokumentation ist bei Libraries essenziell
- bei der API gute Javadocs!
- Paket-Level Javadocs
- Dokumente zur Nutzung (Gesamtüberblick und Erklärungen zum Umgang mit der Library)
  - z.B. mit welchen Paketen soll der User möglichst nicht direkt interagieren

Quellen:

- https://www.oracle.com/corporate/features/library-in-java-best-practices.html
- https://www.baeldung.com/design-a-user-friendly-java-library

Beispiel-Library:

Quelle: https://thegreatapi.com/blog/create-a-library-from-scratch/
(zur Übersichtlichkeit habe ich Kommentare entfernt)

Interface und Implementation sind im selben Paket. Das Interface ist public, damit es vom Benutzer genutzt werden kann.

```java
package stringpadder;

public interface StringPadder {

    String padLeft(String stringToPad, int totalLength);

    String padLeft(String stringToPad, int totalLength, char paddingCharacter);

    String padRight(String stringToPad, int totalLength);

    String padRight(String stringToPad, int totalLength, char paddingCharacter);
}
```

Die Implementationsklasse ist nicht public und somit nur Paketsichtbar. Der Nutzer kann also nicht direkt auf die Methoden zugreifen.
Mit @Override werden die Methoden konkret implementiert. Die Funktion „getStringToBeAdded(...)“ ist private, da sie nur intern benutzt wird.

```java
package stringpadder;

class StringPadderImpl implements StringPadder{

    StringPadderImpl() {
    }

    @Override
    public String padLeft(String stringToPad, int totalLength) {
        return padLeft(stringToPad, totalLength, ' ');
    }

    @Override
    public String padLeft(String stringToPad, int totalLength, char paddingCharacter) {
        return getStringToBeAdded(stringToPad, totalLength, paddingCharacter) + stringToPad;
    }

    @Override
    public String padRight(String stringToPad, int totalLength) {
        return padRight(stringToPad, totalLength, ' ');
    }

    @Override
    public String padRight(String stringToPad, int totalLength, char paddingCharacter) {
        return stringToPad + getStringToBeAdded(stringToPad, totalLength, paddingCharacter);
    }
    private String getStringToBeAdded(String stringToPad, int totalLength, char paddingCharacter) {
        int quantity = totalLength - stringToPad.length();
        if (quantity > 0) {
            return Character.toString(paddingCharacter).repeat(quantity);
        } else {
            return "";
        }
    }
}
```

Um Objekte der Klasse „StringPadder“ zu erstellen, bietet sich eine Factory-Klasse an. Die Klasse ist final und die erzeugten Objekte damit immer eindeutig und unveränderbar (immutable Objects).

```java
package stringpadder;

public final class StringPadderFactory {

    private StringPadderFactory() {
    }

    public static StringPadder createStringPadder() {
        return new StringPadderImpl();
    }
}
```

Wenn man die Library nun nutzen möchte muss man das erstellte Paket importieren und kann dann ein Objekt erstellen. Mit dem Objekt kann man dann die Methoden vom „StringPadder“-Interface nutzen.

```java
import stringpadder.StringPadder;
import stringpadder.StringPadderFactory;

public class Main {

    public static void main(String[] args) {

        StringPadder sp = StringPadderFactory.createStringPadder();

        System.out.print(sp.padLeft("Hallo", 10));
    }
}
```

Beachte: Man kann auf die Klasse „StringPadderImpl“ nicht von Außen zugreifen, da sie nicht public ist. Somit bleibt die konkrete Implementierung verborgen und nur die öffentlichen Methoden können von Außen genutzt werden.
Alternativ könnte man auch „import stringpadder.*;“ nutzen, um alle öffentlichen Klassen des Pakets zu importieren.