Übungsgruppe: Qianli Wang und Nazar Sopiha

Aufgabe 8-1:

a) 1. **Schnittstelle und Signatur:**
   - Eine formale Deklaration, **welche Funktionen vorhanden sind** und wie sie angesprochen werden können
   - **Definition einer Methode** oder Prozedur, bestehend aus Methodenname und Parametern.

2. **Klasse und Komponente:**

   Komponente: in der Entwicklung in Bezug auf Softwarearchitektur ein Teil einer Software.
   Klasse: gemeinsame Struktur und Verhalten von realen Objekten im Softwaredesign.
   Die Klasse ist quasi die Spezifizierung von den Komponenten.

3. **Komponente und Modul:**

   Zu unterscheiden ist ein Modul von einer Komponente, die in der Funktionalität eine Hierarchieebene höher angesiedelt ist und die (Basis-)Funktionalitäten von Modulen zu (fachspezifischen) Diensten kombiniert. Jedoch werden derartige Komponenten im Sprachgebrauch (zum Beispiel bei SAP) manchmal ebenfalls „Module“ genannt.
   (Quelle: https://de.wikipedia.org/wiki/Modul_(Software))

4. **Kohäsion und Kopplung:**

   Kohäsion und Kopplung sind ein Maß um die Komplexität ihrer Software zu analysieren. Das Ziel soll es sein, die Kopplung zwischen den Klassen möglichst gering zu halten und die Kohäsion innerhalb der Klasse möglichst hoch zu halten.
   (Quelle: https://sites.google.com/site/koesterprogramming/home/softwareentwicklung/kohaesion-und-kopplung)

b) **Komponent** kann ohne Änderungen mit anderen Komponenten verknüpft und ausgeführt werden.
(Quelle: https://de.wikipedia.org/wiki/Komponente_(Software))

Entwurfsmuster: sind bewährte Lösungsschablonen für wiederkehrende Entwurfsprobleme/Teilprobleme eines Systems in der Softwarearchitektur
(Quelle: https://de.wikipedia.org/wiki/Entwurfsmuster)

Architekturstile: beschreibt die Struktur und Eigensch aft eines Systems.

c) 1. **The singleton pattern** is a software design pattern that restricts the instantiation of a class to **one "single" instance**. This is useful when exactly one object is needed to coordinate actions across the system.
(Quelle: https://en.wikipedia.org/wiki/Singleton_pattern)

2.

- How can it be ensured that a class has only one instance?
- How can the sole instance of a class be accessed easily?
- How can a class control its instantiation?
- How can the number of instances of a class be restricted?

(Quelle: https://en.wikipedia.org/wiki/Singleton_pattern)

3. **Creational Pattern**. Our goal here is to provide an abstraction for a (possibly complex) instantiation process.
(Quelle: Folien)

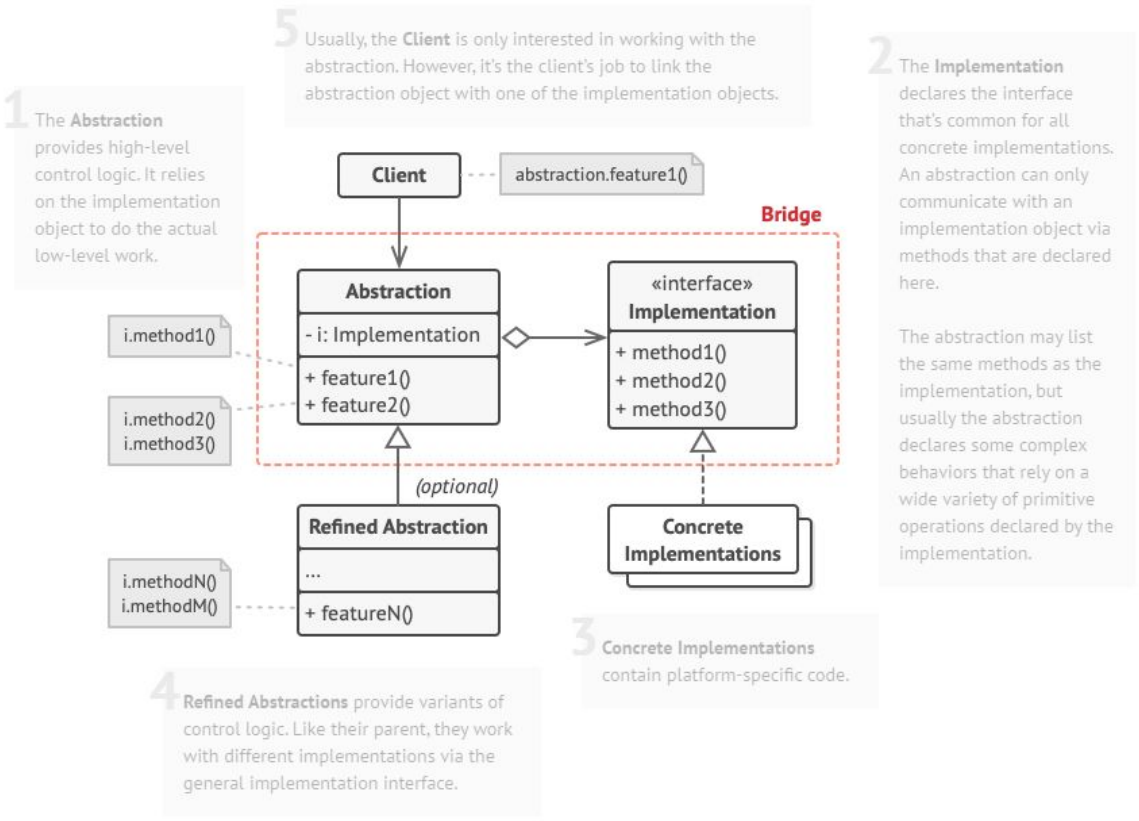
4.

- **Proxy**: Provide an replacement object that can perform the direct access for the client.
- **Adapter**: Interfacing to existing system. Provide a service that conforms to a given target interface T.
- **Facade**: Interfacing to subsystems. Provides a unified interface to a set of objects in a subsystem.
- **Bridge**: Interfacing to existing and future systems. Decouple an abstraction from its implementation so that the two can vary independently.

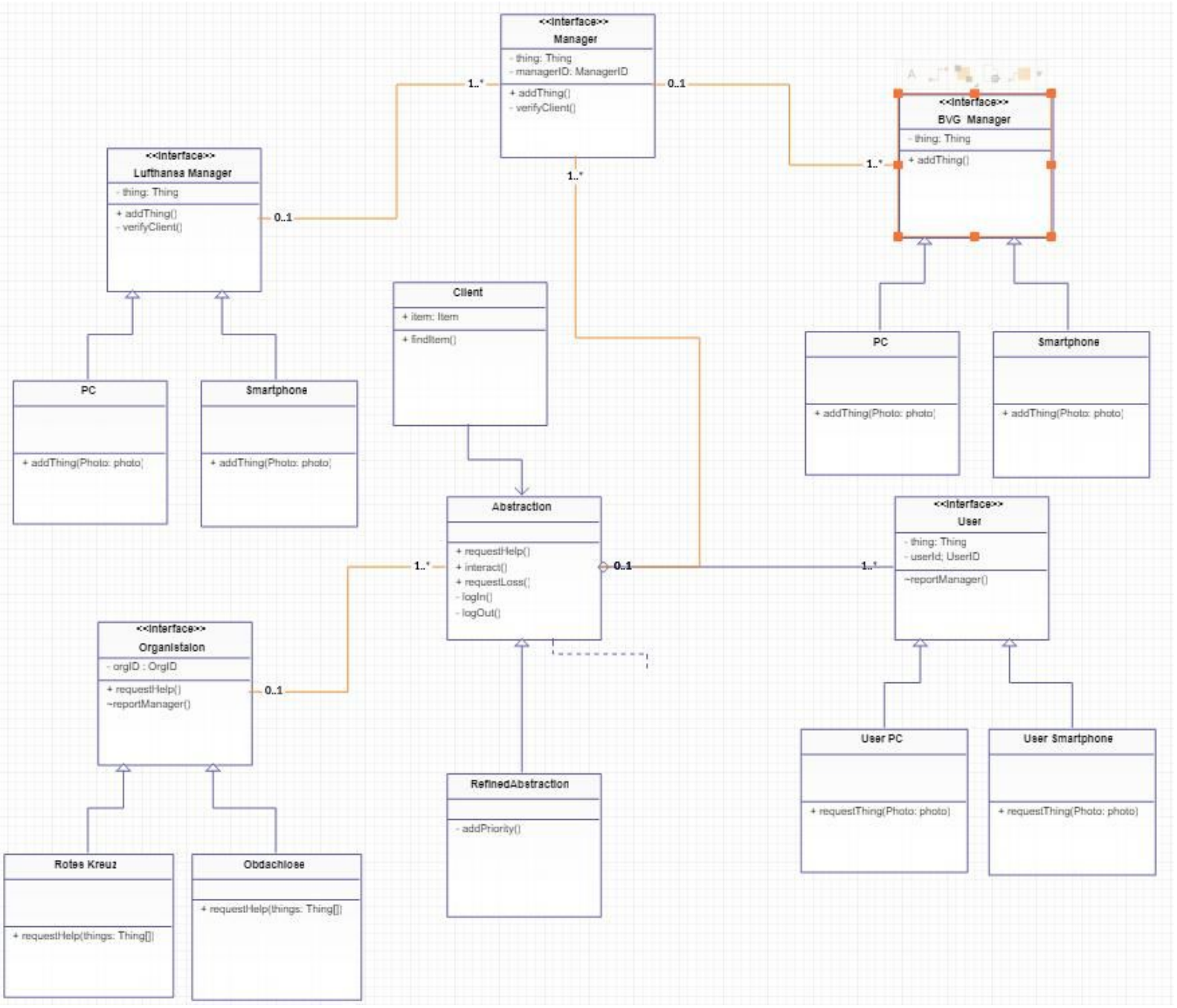
Aufgabe 8-2:

a) Wir haben ziemlich lang nachgedacht welches aus drei Design Patterns wir wählen sollen: **Client-Server** (weil potentiell wir ein Server mit Big Data erhalten und ganz ganz viele Zugriffe aus vielen Devices darauf), **Proxy** (Da es nur ein passendes Ding als Ergebnis der Suche sein kann, macht Proxy Interaction viel schneller, indem nur gesuchte Objekte dargestellt werden) und **Bridge.**
Wir haben uns für **Bridge** entschieden, weil im Endeffekt wollen wir auf ganz vielen Plattformen funktionieren. Außerdem gibt es in verschiedenen großen Firmen (Flughafen Berlin, BVG, Deutsche Bahn, u.s.w.) verschiedene interne Systeme, die mit unserer App kompatibel sein sollen. Genau das erlaubt **Bridge** als Structural Pattern.

b) **Bridge** gibt uns die Möglichkeit die Abstraktion und die genaue Implementation voneinander trennen. Und wir können auch von einer Abstraktion zu vielen verschiedenen Implementations mehrere Bridges haben, was hilft uns mit jedem Betrieb flexibel zu sein.



(Quelle https://refactoring.guru/design-patterns/bridge)



Als ein Beispiel für genaue Implementation nennen wir verifyClient(). Für jede genaue Situation soll es unterschiedlich vorgehen, da für verifyClient() bei BVG reicht es ein Photo des Tickets hochzuladen und es auch optional sein kann. Für Flughafen ist es schon ganz wichtig, den Passagier im System zu finden.
Code schreiben wir für ein anderes Feature und nämlich für Photo upload beim requestLoss(). Bei Android/IOS kann es auch mit dem Kamera eine Aufnahme gemacht werden. Mit PC kann es nur hochgeladen werden.

```
public interface Device {
    public void requestHelp();
    public void interact();
    public void requestLoss();
    private int logIn();
    private int logOut();
}

public interface User implements Abstraction {
    public Thing thing = new Thing ();
    private UserID userID = LogIn.getLoginInfo();
    protected int reportManager();
}
```

```
public class PCUser implements User {
    @Override
    public void requestLoss (Photo photo) {
        if (!photo) {
            photo = uploadPhoto();
        }
    }
}
```

```
public class SPUser implements User {

    enum PhotoOption {
        UPLOAD,
        MAKE
    }

    int PhotoOption option = askHowToPhoto();

    @Override
    public void requestLoss (Photo photo, Option option) {
        if (!(photo)){
            switch(option) {
                case UPLOAD:
                    photo = uploadPhoto();
                    break;
                case MAKE:
                    photo = makePhoto();
                    break;
                default:
                    System.out.println("Wrong input");
                    System.exit(-1);
                    break;
            }
        }

}
```

Aufgabe 8-3:

a)

```
1  public class NotepadMinusMinus{
2      private Storage storage;
3      public void save(){
4          String content = DocBuffer.getContent();
5          String name = DocManager.getCurrentFileName();
6          storage = new Storage();
7          storage.store(name, content);
8      }
9  }
```

```
public class Storage {
    private Filesystem fs;

    public void store(String name, String content) {
        fs.remove(name);
        fs.touch(name);
        fs.write(fs.position(name), content.getBytes());
    }
}
```

b) - **Client**: not defined
- **Target**: NotepadMinusMinus
- **Adapter**: Storage
- **Adaptee**: Filesystem

c)

```
1  public class NotepadMinusMinus{
2      private IStorage storage;
3
4      public void save(){
5          String content = DocBuffer.getContent();
6          String name = DocManager.getCurrentFileName();
7          storage.store(name, content);
8      }
9
10     public void setStorage(IStorage s){
11         this.storage = s;
12     }
13 }
14
15 public class DbxClient{
16     public enum WriteMode{
17         ADD, REPLACE
18     }
19
20     public boolean fileExists(String name){}
21
22     public void uploadFile(String name, WriteMode mode, byte[] data){}
23 }
24
```

```
25 public class Dropbox implements IStorage{
26     private DbxClient client = new DbxClient();
27
28     public void store(String name, String content){
29         DbxClient.WriteMode mode = WriteMode.ADD;
30         if(client.fileExists(name)){
31             mode = WriteMode.REPLACE;
32         }
33         client.uploadFile(name, mode, content.getBytes());
34     }
35 }
36
37 public interface IStorage{
38     public void store(String name, String content);
39 }
40
41 public class Storage implements IStorage{
42     private Filesystem fs;
43
44     public void store(String name, String content){
45         fs.touch(name);
46         fs.remove(name);
47         fs.write(fs.position(name), content.getBytes());
48     }
49 }
```

d) - **Client**: NotepadMinusMinus
   - **Target**: IStorage
   - **Adapter**: Storage, Dropbox
   - **Adaptee**: Filesystem, DbxClient

e) **Dependency injection** is a technique in which an object receives other objects that it depends on. These other objects are called dependencies.

Adventages:

- allows a client to remove all knowledge of a concrete implementation that it needs to use.
- decreases coupling between a class and its dependency.
- allows concurrent or independent development.

Disadventages:

- can make code difficult to trace (read) because it separates behavior from construction.
- creates clients that demand configuration details be supplied by construction code.
- forces complexity to move out of classes and into the linkages between classes which might not always be desirable or easily managed.

(Quelle: https://en.wikipedia.org/wiki/Dependency_injection)