Aufgabe 8

Nathan Ritter
5566519
Lorenzo Tecchia
5581906
2023.06.11

Contents

Chapter 1

Task 1

(a) Each case explained:

1. Interface and signature: Interface is the abstraction(non implementation) of a class, whereas the signature is the declaration(non implementation) of a method. [1]
2. Class and component: Using the role phase to differentiate them, a component is a design-time unit whereas a class is an implementation-time unit. Designers speak about components but programmers speak about classes.[5]
3. Component and module: Components are usually very small in scope, but modules are composed of components and are of very large scope. [5]
4. Cohesion and coupling: Cohesion is a concept intra-modules, instead coupling is a concept inter-modules. [6]

(b) Design patterns are usually associated with code level commonalities. It provides various schemes for refining and building smaller subsystems. It is usually influenced by programming language. Some patterns pale into insignificance due to language paradigms. Design patterns are medium-scale tactics that flesh out some of the structure and behaviour of entities and their relationships. While architectural designs are seen as commonality at higher level than design patterns. Architectural patterns are high-level strategies that concerns large-scale components, the global properties and mechanisms of a system.[7] In general architectural designs describe how all the system works and how users of the system interact with the system itself, but different design patterns are usually concerned with how components of the system interact with one and other and multiple design patterns can coexist in a single system, whereas for a system usually one architectural design is chosen.
Components are usually the building blocks constituting architectural designs.

(c) The singleton pattern:

1. Ensures that a class has only one instance and provides a global point of access to it. Resolves the problems of polluting the code with global variables and sole instances, resolves many security problems since the singleton class implementation facilitate control on how and when clients access to the class. It's also more flexible than Class operations(for languages like C++). [3]
2. When establishing via an OOP language a connection to a DB, and every query must be executed atomically, there would be implemented a singleton-style class(e.g a Connection class), every class that need to access the DB has to report to the same Connection Class and access the DB one at a time.
   When the sole instance should be extensible by subclassing, and clients should be able to use an extended instance without modifying their code. In a software environment where a second instance of a class shouldn't be created a better solution is to make the class itself responsible for keeping track of its sole instance. The class can ensure that no other instance can be created (by intercepting requests to create new objects), and it can provide a way to access the instance. [3]
3. The singleton design pattern belongs to the branch of Creational design patterns.

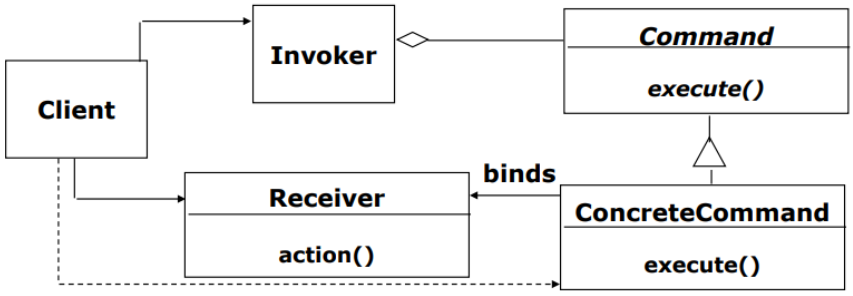
(d) **Proxy**'s purpose is to provide a surrogate or placeholder for another object to control access to it.
The **Adapter** design pattern converts the interface of a class into another interface clients expect. Adapter lets classes work together that couldn't otherwise because of incompatible interfaces.
The **Bridge** design pattern decouple an abstraction from its implementation so that the two can vary independently. **Bridge** is very similar to **Adapter**, but we call it Bridge when you define both the abstract interface and the underlying implementation
The **Facade** design pattern purpose is to provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher- level interface that makes the subsystem easier to use.[4][3]

Chapter 2

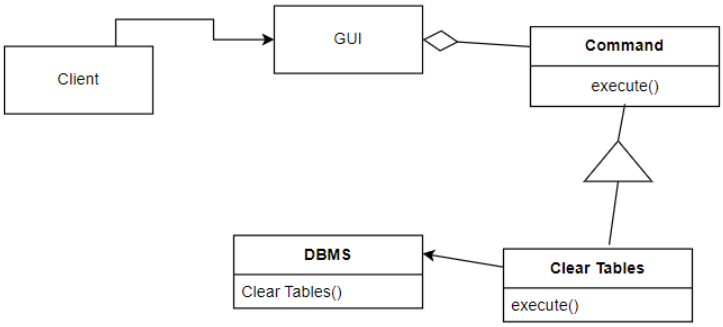
Task 2

(a) One could use the "Command" design pattern for our project, because commands are sent to the database and they can fail or take too long. It would be useful to save them as objects so that you can undo the changes of the commands to the database if the command is aborted or fails. The "Command" design pattern solves the problem of treating activities/commands like data by wrapping the activities/commands in objects with a consistent interface. And this also allows to undo these commands because the different command objects can also store the data that is modified to be restored when needed.

(b) The "Command" pattern Source: Lecture



(c) The "Command" Pattern Applied



(d) Pseudo-code for the design pattern:

```
class Gui{
    Command  clear_command;
    void Init(){
        clear_command = new ClearTables()
    }
    void OnClickClearTable(){
        clear_command.execute()
    }
}
Interface Command{
    void exectue()
}
class table{
    int id;
    char data[1000];
}
class DBMS{
    void cleartables(List<table>){...}
    List< table> getalltables(){...}
}
class cleartables implements Command{
    List<tables> old_tables;
    @override
    execute(){
        old_tables = DBMS.getalltabels()
        DBMS.cleartables(DBMS.getalltabels())
    }
}
```

Chapter 3

Task 3

(a) This is the *Storage* class with the updated method to save the content.

```
package Storage;
pulic class NotePadminusminus{

    /* ... */

    public void save() {
        String content = DocBuffer.getContent();
        String name = DocManager.getCurrentFileName();

        Storage.store(name, content);
    }
}
```

This is the *Storage* class that implements the old class *Filesystem*.

```
package Filesystem;

public class Storage implements Filesystem{

    Filesystem fs = new Filesystem();

    public int store(String filename, String content){

        if (fs.position(filename) > 0) {
            fs.remove(filename);
            fs.touch(filename);
            fs.write(fs.position(filename),content.getBytes
                ());
        }else{
            fs.touch(filename);
            fs.write(fs.position(filename),content.getBytes
                ());
```

```
        }

    }

}
```

(b) Here the client is the class *Notepadminusminus*, the Target is the methods for the class *Filesystem*, the Adaptee is the class *Filesystem* and finally the Adapter is the class *Storage*.

(c) Dropbox solution:

```
public interface IStorage{
    public void store(String name, byte[] data);
    }


public class NotepadMinusMinus {
    IStorage Storage;
    publicic void setStorage(IStorage s) {
     Storage = s;
}

public void save() {
    String content = DocBuffer.getContent();
    String name = DocManager.getCurrentFileName();
    Storage.store(name, content)
    }
}

public class DropBoxStorage implements IStorage {
    public void store(String filename, byte[] content){
        if(DbxClient.fileExists(filename)){
            uploadFile(filename, DbxClient.WriteMode.REPLACE
                , content);
        }
        else{
            uploadFile(filename, DbxClient.WriteMode.ADD,
                content);
        }
    }
}

public class Storage implements IStorage {
    public void store(String filename, byte[] content){
    int fd = Filesystem.position(filename);
    if(fd == -1){
        Filesystem.touch(filename);
        fd = Filesystem.position(filename);
```

```
36      }else{
37          Filesystem.remove(filename);
38          Filesystem.touch(filename);
39          }
40          Filesystem.write(fd, content);
41      }
42  }
```

(d) Client: Notepad minus minus.
Target: IStorage.
Adapter: Storage, DropBoxStorage.
Adaptee: Filesystem, DbxClient.

(e) Advantages of Dependency Injection are :

- It decouples classes from their dependencies so the code is more easily reused because it no longer depends on the implementation of the dependencies.
- It allows several developers to build classes that depend on each other at the same time, without having to finish one before the other, because it only needs to know the interface through which the classes communicate.

Disadvantages are:

- It makes the code harder to track because it separates behaviour from construction.
- It is harder to implement.
- Creates clients that need configuration details that can be hard to figure out.[2]

Bibliography

Deitel, Paul J and Harvey M Deitel. *Java for programmers*. Pearson education, 2009.

Dependency injection - Wikipedia — en.wikipedia.org. https://en.wikipedia.org/wiki/Dependency_injection. [Accessed 11-Jun-2023].

Gamma, Erich et al. "Elements of Reusable Object-Oriented Software". In: *Design Patterns* (1995).

How do the Proxy, Decorator, Adapter, and Bridge Patterns differ? — stackoverflow.com. https://stackoverflow.com/questions/350404/how-do-the-proxy-decorator-adapter-and-bridge-patterns-differ. [Accessed 09-Jun-2023].

Is there a difference between a component and a module — softwareengineering.stackexchange.com. https://softwareengineering.stackexchange.com/questions/178927/is-there-a-difference-between-a-component-and-a-module. [Accessed 09-Jun-2023].

Software Engineering | Differences between Coupling and Cohesion - GeeksforGeeks — geeksforgeeks.org. https://www.geeksforgeeks.org/software-engineering-differences-between-coupling-and-cohesion/. [Accessed 09-Jun-2023].

What's the difference between design patterns and architectural patterns? — stackoverflow.com. https://stackoverflow.com/questions/4243187/whats-the-difference-between-design-patterns-and-architectural-patterns. [Accessed 09-Jun-2023].