Lecture "Software Engineering SoSe 2023

Free University of Berlin, Institute of Computer Science, Software Engineering Group Lutz Prechelt, Linus Ververs, Oskar Besler, Tom-Hendrik Lübke, Nina Matthias

Exercise sheet 8 **Design pattern** for 2023-06-12

Task 8-1: Terms

Learning Objectives: Be able to explain central concepts from the field of software design, know common design patterns.

a) In each case, briefly explain the most important difference or connection between ...
   1. Interface and signature
   2. Class and component
   3. Component and module
   4. Cohesion and coupling

b) Explain the most obvious differences between *components*, *design patterns*, and *architectural styles*.

c) Research the *singleton* design pattern. As always, remember to cite your sources.
   1. Explain the design pattern, i.e. what problem does it solve and how?
   2. Illustrate the design pattern by formulating three hypothetical examples of its use.
   3. To which class of the design *pattern taxonomy* presented in the lecture does this pattern belong and why?

d) Characterize and compare the following design patterns: *proxy*, *adapter*, *facade*, *bridge*.

Task 8-2: Design pattern for own software idea

Learning objectives: To be able to select and apply design patterns appropriately; to be able to distinguish design patterns from their concrete application.

a) Think about which of the design patterns presented in the lecture you can usefully apply in your software to be developed (except for the singleton and the adapter).

Highlight the characteristics of the chosen design pattern (i.e., what problem does it solve and how) and justify why and for what purpose the pattern is appropriate in your context.

b) Present the design pattern in its *general form*, i.e. using the role names, in UML notation. Research as needed and cite your sources in each case.

c) Now transfer the selected design pattern *to your context* and represent the result as a UML diagram as well.

d) Not all aspects of a design pattern can be conveniently expressed in a UML diagram. Implement (in Java or pseudo code) *the use of the design pattern* in your context. Create the interfaces and classes necessary for the pattern, with the relevant code passages, i.e. at least hinted data and control structures (empty method hulls are usually *not* sufficient). Explicitly do not write any code that goes beyond the mere design pattern!

As always, remember to add your work products from subtasks **a)** through **d)** to your wiki page in addition to submitting them electronically via KVV.

Task $^{8\text{-}3Æ}$: Apply adapter design pattern

Learning Objectives: Be able to concretely craft one of the most important design patterns in a realistic context.

Preliminary note: Don't be put off by all the text in this assignment. It is neither difficult nor complex in terms of content.

Background: Imagine you are in the process of developing a simple text editor with the na- men "Notepad- -" in Java. The functionality is very reduced and you have your hands full to extend it. Therefore, you want to reuse existing solutions as much as possible.

a) For the file management of your editor, you want to reuse a `filesystem` class that you once developed. It works perfectly, and therefore you don't want to change it anymore. You have therefore decided to use the adapter design pattern.

This is an excerpt from the interface documentation of the `Filesystem` class:

```
public class Filesystem {
    // Returns the ID of the logical sector where the given file resides.
    // Returns -1 if no such file exists.
    public int position(String filename) { /* ... */ }

    // Write the content on the disk into the sector with the given ID
    public void write(int sector, byte[] content) { /* ... */ }

    // Prepares the creation of a new file. Provide the content
    // through the write method! Fails if the name is already taken.
    public void touch(String filename) { /* ... */ }

    // Erases a file, and compacts the content of the disk, such that
    // there are no gaps in between.
    public void remove(String filename) { /* ... */ }
}
```

The use of the Filesystem `class` is somewhat unguided, and in your main class `NotepadMinusMinus` you only want to make a single call to save the ac- tual editor content. You have already implemented the following framework:

```
public class NotepadMinusMinus {
    /* ... */

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

        // TODO Save to disk (replace existing file, if any)
    }
}
```

Task: Implement a new class (such as "`Storage`") with a single method (like "`store`"), which gets file name and content and reuses your old `filesystem` class - without changing it!

Also add the `save()` implementation from above fragment to get the Storage `class` to use. Do not introduce wide classes or interfaces.

b) The classic adapter pattern recognizes the following four roles: Client, Target, Adapter, and Adaptee.[1] For each of these roles, specify which of the classes in your implementation fills it?

[1] see e.g. http://www.blackwasp.co.uk/Adapter.aspx

c) Now you also want to give your editor Dropbox integration. For such purposes, Dropbox itself offers an SDK (library). For this task, assume the following simplified API (programming interface):

```java
public class DbxClient {
    public enum WriteMode {
        ADD, REPLACE
    }

    // Checks whether a file already exists on the server
    public boolean fileExists(String name) { /* ... */ }

    // Uploads a new version of a file; either as a new file or as a
    // Replacement for an existing one. New files must be transmitted
    // Using the ADD mode, replacements must be done in REPLACE mode.
    // Incorrect modes indicate inconsistencies and the upload will
    // fail.
    public void uploadFile(String name, WriteMode mode, byte[] data) {
        /* ... */ }
}
```

Task: Use the Dropbox SDK to add a storage option to your editor. As with `filesystem reuse,` you should not (or *cannot*) modify the Dropbox source code. So write another adapter for this.

To avoid depending directly on one of the two adapters (one for the local file system, one for Dropbox) in the `NotepadMinusMinus class,` please introduce an interface (something like "`IStorage`") that implements the two adapters respectively.

To make use of this in the `NotepadMinusMinus` class, please implement a setter method (see below) and use the passed value in the already existing `save()` method:

```java
public class NotepadMinusMinus {
    /* ... */

    public void setStorage(IStorage s) {
        /* ... */
    }
}
```

d) Analogous to task **b)**: What is the distribution of roles (i.e., client, target, adapter, ad- aptee) among the classes and interfaces of the second implementation? Consider that you now have *two copies of* the adapter pattern.

e) In the second implementation, the central editor class does not "know" at compile time which implementation of the storage `interface` it will actually use. This is only "passed in" at runtime via the `setStorage()` method.

Research the idea of "Dependency Injection" (cite sources). What are the advantages of this design principle? What disadvantages?