University of Vienna
Faculty of Computer Science
Research Group Software Architecture

Task 3

Design Patterns in Practice 2

2017S – 051050 VU – Software Engineering 2

May 24, 2017

General Remarks

- This document covers the assignment of **Task 3**.
- You can achieve a total of **18 points** for this task.
- The **deadline** for this **Task 3** is **18.06.2017 at 23:59**. No deadline extensions are given.
- This is a **group work**. You and your work group members are allowed to work together in solving this assignment. Be aware that **all** group members shall contribute to the task and document their contribution.
- If you copy code or other elements from sources other than the lecture slides, please provide a reference to it in a comment above the corresponding entry.
- If you encounter problems, please post your question in the Moodle[1] discussion page, as it probably is also of interest to other colleagues. Alternatively, you can contact the tutors via se2.tutor@swa.univie.ac.at. As a last resort you can contact the course supervisor directly via se2@swa.univie.ac.at.

Submission Guidelines

All files required by this assignment have to be submitted to our GitLab[2] server into the proper project (repository) in the Submission and Feedback System[3]. **Be aware** to push your solutions into the correct submission **branch**, which is for this task **2017s_se2_task3**.

As discussed in the Git[Lab Submission] Tutorial[4] the submission branch is created for you. For any questions regarding the **GitLab**-based submission please refer to the Git[Lab Submission] Tutorial.

[1]https://moodle.univie.ac.at/course/view.php?id=61845
[2]https://gitlab.swa.univie.ac.at
[3]https://gitlab.swa.univie.ac.at/submission
[4]https://gitlab.swa.univie.ac.at/submission/tutorial

Task 3 : Design Patterns in Practice 2

This task aims to provide a practical exercise on the topics covered in the lecture part of the Software Engineering 2 course.

Task Description

The goal of the practical exercise is the development of an accounting application for Software as a Service (SaaS) in Java. SaaS is a business model for (Web-based) applications. Software is licensed on a subscription basis and paid by charging the time or amount of use. For instance, a user pays 10 EUR per month to access and use the software.

In this task, you are developing a back-office application that creates invoices for a SaaS company.

The main focus of this task lies in writing and structuring the code properly (also see Non Functional Requirements) and using design patterns in a reasonable way. Use Java Swing[5] or JavaFX[6] to create a simple user interface. Persistence is not required. Instead, just provide a set of test data, which is always loaded when the application is started and/or used in the unit tests (see below).

Before you start

To create, send and manage customers and invoices, you have to use FastBill[7]. FastBill provides an API to manage customer data and create invoices[8].

- Create a free test account which is valid for 30 days
- To access the API, you need the FastBill API key, which can be found in the settings menu ("Einstellungen – Übersicht").

Hint

To communicate with the REST API you need a HTTP(S) client and a JSON parser. Both can be found in Apache's "HTTP Components" HTTPClient[9]. Any other set of libraries can be used as well. Keep in mind, that your submitted project must be runnable and all dependencies must be included and properly configured in the project, or a script to set up the IDE must be provided.

[5] https://docs.oracle.com/javase/7/docs/technotes/guides/swing/
[6] http://docs.oracle.com/javase/8/javafx/get-started-tutorial/jfx-overview.htm
[7] https://www.fastbill.com
[8] https://www.fastbill.com/api/
[9] http://hc.apache.org/httpcomponents-client-ga/index.html

Functional Requirements (FR)

FR1 Properties: The behavior of the application can be configured via configuration file (XML or properties). The following properties must exist:

- **api-key**: `a21...812` the fastbill api key
- **api-email**: `any-email@address.com` the fastbill user email
- **environment**: `productive` or `test` if set to productive, the real API is used. if set to test, the mockup implementation is used

FR2 API Client: For any required FastBill API request, create a client implementation. Also, provide a set of mockups for the request, so you can use them for Unit Tests. The client can be switched from productive to test environment by changing the configuration file. Your API Client must provide (at least) the following functionality:

- create a new customer
- create a new invoice for a customer
- get all unpaid invoices for a customer

Hint: Use a Strategy Pattern and an Abstract Factory Pattern to switch between the FastBill Client and the Mockup Client. Hide the client implementation using the Proxy Pattern. To avoid struggling over issues while accessing the FastBill API, start with the Mockup implementation first. Then continue with the other bullet points. And when your progress is great, implement the FastBill Client implementation at the very end.

FR3 Products: There exists a set of products, at least 3. A product offers plans, which can be subscribed by customers. One customer can only subscribe to one plan per product. Each product offers at least 2 plans. A plan has a rate, i.e. the price a user gets charged each month for subscribing to that plan.

FR4 Customers: You can create new customers. Customers must also be stored in FastBill. A customer has a local identifier as well as a remote identifier in FastBill. Each customer can subscribe to several plans. When a customer subscribes to a plan, she gets 1 month for free (also see invoicing). A customer knows the set of pending (not paid) invoices, managed by FastBill.

Hint: Use your API Client for Fastbill. Use the Proxy Pattern with an Abstract Factory to switch between access to FastBill or your Mockup implementation.

FR5 Invoicing: For each subscribed plan of a customer, the system stores a 'valid until' timestamp. The application can check for all customers, if there is any subscribed plan with an expired 'valid until' timestamp. If a subscription has expired (i.e. the 'valid until' timestamp lies in the past), a new invoice for the subscribed plan is created in FastBill for the corresponding customer. If an invoice got paid, the 'valid until' will be extended by 1 month (also see Payment).

Hint: Use your API Client for Fastbill. Use the Proxy Pattern with an Abstract Factory to switch between access to FastBill or your Mockup implementation.

FR6 Payment: Check for any customer, if there are unpaid invoices by checking invoice due date in FastBill. If payment is outstanding, the access to the product gets suspended (also see Product Lifecycle). If an invoice got paid, the 'valid until' timestamp for a customers subscribed plan will be extended by 1 month (also see Invoicing).

Note: Invoices will get marked as 'paid' via FastBill

Hint: Use your API Client for Fastbill. Use the Proxy Pattern with an Abstract Factory to switch between access to FastBill or your Mockup implementation.

FR7 Subscription Lifecycle: For each product/plan that customer subscribed to the following Subscription Lifecycle exists:

state	event	state
inactive	subscription started	active
active	subscription canceled	inactive
suspended	subscription canceled	inactive
active	payment pending	suspended
suspended	payment received	active
active	payment received	active

For any customer, the subscription state (active, suspended, inactive) of any subscribed product can be queried at any time.

The following Figure 1 will give you an overview of the model. Note, that your specific model may differ widely.

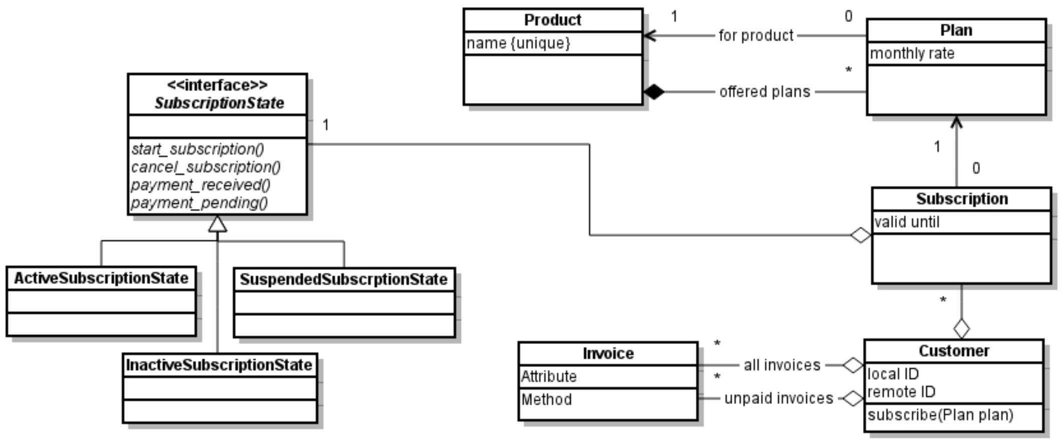

Figure 1: Overview of the model

Non-Functional Requirements (NR)

NR1 **Comment** your code and provide JavaDocs.

NR2 Your implementation must be in compliance with the **Google Java Style Guide** [10] and other **common coding practices** (see lecture slides).

NR3 Use **defensive programming** (see lecture slides).

NR4 Use **key design principles** (see lecture slides).

NR5 Make sure that your implementation works properly by **testing** your implementation thoroughly (e.g., JUnit). There exist Unit Tests for at least:

- API Client, using the mockups (flag set in configuration file)
- Customer subscription to a plans of a product
- Invoicing using API mockups
- Payment using API mockups
- Product Lifecycle

NR6 The solution **must use** the following design **patterns** whenever it is appropriate. Although there might already exist default Java implementations for some of the patterns, use your own implementation of the pattern at least once.

- **Iterator Pattern**
- **Composite Pattern**
- **Proxy Pattern**
- **Observer Pattern**
- **Strategy Pattern**
- **Abstract Factory Pattern**
- **Decorator**
- **Factory Method**

Additional Requirements (AR)

AR1 **Keep records** of your activities whenever you work on the project. That includes every activity that is directly related to this assignment. Important points that must be covered are:

- **Who**? Did you work alone or together with team members?
- **When** and for how long? Did you work in several iterations on a problem?
- **Why**? What is the purpose of your activity?
- **What** was achieved?

[10] https://google.github.io/styleguide/javaguide.html

AR2 How and to what extent have you considered **coding practices**? Discuss and show examples from your code.

AR3 How and to what extent have you considered **defensive programming**? Discuss and show examples from your code.

AR4 How and to what extent have you considered **key design principles**? Discuss and show examples from your code.

AR5 Where do required design patterns occur in the solution? Discuss at least one occurrence for each required **design pattern** in the code in detail. Support the use of a specific pattern by arguments. Create UML diagrams to illustrate how the pattern has been applied.

Submission

Deadline for this **Task 3** is on **18.06.2017 at 23:59**. Only material that has been submitted in time (pushed to your GitLab Work Group Project in the suitable branch) and following the naming conventions will be taken into consideration:

- Records of your work in a single file named `team_records.pdf` in the root directory.
- A discussion on how you applied coding practices in a file named `coding_practices.pdf` in the root directory.
- A discussion on how you applied defensive programming in a file named `defensive_programming.pdf` in the root directory.
- A discussion on how you applied key design principles in a file named `key_design_principles.pdf` in the root directory.
- A discussion on how you applied design patterns in a file named `design_patterns.pdf` in the root directory.
- Your implementation (including test cases etc.) in the folder `implementation` in the root directory. Please **DO NOT** bundle your code in archives like `.zip`, `.7z`, `.rar`, etc.
- Your documentation on how the application is to be installed, initialized, and tested in a file named `howto.pdf` in the root directory

Examination

Each team will be assigned a time slot to present the solution. Each group member should have knowledge of the **entirety** of the submitted solution, not simply the part he or she has worked on, and be able to explain the design process, decisions, and the rationale behind them. No elaborate presentation material (PowerPoint slides etc.) is required apart from the material you submitted as your solution.