Contents

Abstract and Exposeé

There are many many different approaches to achieve good software design, however often it is not clear whether the benefits of the best practices overweight the additional effort of implementing them. Little research is done on comparisons between the designs in terms of efficiency and programming effort. Many books do show sample code of their proposed designs, however they do not cover how the code would look like if implemented in a more straight-forward and monolithic way.

This bachelor's thesis tries to cover this topic by implementing the same application two times: as a monolithic program and as a program that pays attention to the information provided about software design and architecture. In the first step, the program will use only one data source and one graphical interface (GUI). In the second step in each program another data source and GUI was added.

The sourcecode will be available on Github [1] under GPLv3 [2]. The application will be written in Java using Swing as a front-end and PostgreSQL as database. In phase two JavaFX should be supported as a front-end, XML should be supported as a database. The user will be able to choose between all options using parameters when starting the application.

[1] https://github.com/Vallant/design-patterns-study.git
[2] https://www.gnu.org/licenses/gpl-3.0.de.html

1 Introduction

TODO Ask the question that will be solved by this thesis e.g. ”What is the impact of using best practice software architecture patterns over ad-hoc monilithic software development?”

2 Background

This chapter is intended to provide information on some basic principles in terms of the Object Oriented Programming. The list is by no means exhaustive, however is meant to help understanding the focus of this thesis.

2.1 Fundamentals of Object-Oriented Programming

The paradigm of Object Oriented Programming (OOP) uses *classes* as primary mean to gather and structure data. The data within a class is mostly called *attributes*, means to interact with it are called *methods* Castagna, 1997, p. 80.
While a class is the abstract definition of such a container an *object* is a concrete instance filled with actual data. Attributes that may differ between each instance are therefore also called *instance variables*. Variables that belong to the class itself and thus are only instantiated once per class are called *class variables*.

2.1.1 Encapsulation

To encourage refactoring each class should prevent direct access to is internals from the outside. It should however provide a well-defined interface in terms of methods for manipulating the data, as this allows the class to enforce invariants. This means that it hides all information not relevant to others as they are only implementational details. As other classes now rely on an interface rather than concrete implementations the code is called

loosley coupled. As modifying the internals does not break interdependencies it encourages programmers to perform refactoring results in improved code quality. Many programming languages provide different access levels varying from visible to all others, accessible only within the class or visible from within and derived classes. The latter access level can be problematic as they effectively break the encapsulation by providing direct access to subclasses.

2.1.2 Inheritance

Inheritance describes the concept of a child class inheriting all attributes, methods and other properties from a parent class. The child class is connected with the base class trough a *is-a*-relationship. The child class is therefore a superset of the base class as it can be extended to meet the requirements. This concept is important as it encourages developers to reuse existing code and in that way lower the risk of programming errors Ralph E. Johnson, 1991.
A prominent problem often mentioned in this context is the *Diamond Problem* in the sense of multiple inheritance. It describes a situation in which at least two parents of a derived class share a single base class **Truyen04** If now a method of the topmost class is overridden by both ancestors of lowermost class, the question arises which of the two possible methods should be called. Some languages, such as Java or C#, do not support multiple inheritance for this reason, while others explicitly allow it, such as C++ or Python. In this cases if a situation as described in the Diamond Problem arises the results can cause undefined behaviour.

2.1.3 Polymorphism

Polymorphism describes the ability to tie the same interface to different belonging types. There are two main kinds of polymorphism: The *overriding* polymorphism, which is tied closely to inheritance and describes the ability to choose at runtime between equally-called methods and attributes of a base class and its child class. For example, if a base class `Animal` has a

method `speak`, each derived class `Dog` and `Cat` both inherit this method. With overriding polymorphism if the method is called the two subclasses are able to behave in different ways while providing the same programming interface. It is determined at runtime which method should be executed for an object.

The other important kind is of *overloading* polymorphism which is used to provide methods with the same name but different signatures (and thus attributes). An example could be two methods called `add`, one taking a number, one taking a text as a parameter. Here it is determined at compile time which method will be used. [1].

2.1.4 Single Responsibility Principle

This important principle states that each class should only full-fill one particular purpose and as a result does only have one reason to change. The computer scientist D.L. Parnas wrote that in software development each design decision which is likely to change should be placed in a single, independent module and hides this decision from others **srp** When followed it avoids side effects on other responsibilities when changing the class. A example of a class violating this concept could be a class that reads two numbers from the user, calculates the sum and prints the result. While this program seems quite simple three different responsibilities are placed in the same class. If either the means to provide the input, for presenting need to be modified or the algorithms should support other data types, the class need to be changed. In a conforming program each of this three actions would be placed in a own module.

2.1.5 Open Closed Principle

The Open-Closed-Principle states that each class should be open to extension and closed for modification **ocp** As already written code is assumed to be well tested and working as intended it should be avoided to modify it

[1] https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/polymorphism, accessed 31.08.2017

afterwards to add new functionality. Every change could lead to unwanted side effects that may only occur in very specific situations and are therefore difficult to prevent. Inheritance addresses this problem as it empowers the programmer to add new features while preserving all of the old code. Even when overriding methods in terms of polymorphism the principle is not violated at the original class is preserved as-is.

2.2 Design Patterns

In opposite to the previous chapter which describes rather abstract guidelines in the object oriented programming this section focuses on widespread methods which try to help solving common problems that might occur in bigger projects. They are called *design patterns* and are subject to a large amount of books, of which the most famous is probably *Design Patterns - Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson and Jphn Vlissides and *Patterns of Enterprise Application Architecture* by Martin Fowler. While the former book is more focused on general programming patterns, the latter one is more relevant for bigger businesses where a multi-tier architecture (see section 2.3.2) is more common.

2.2.1 Repository

The repository pattern is a programming strategy that is useful when operating with some kind of data persistence service. It provides an abstraction layer between the business code and the place where data of the application is actually saved, providing a simple and clean interface for the business code and leads to a looser coupling. All actions dealing with specialized database operations, object retrieval and others are hidden by a repository. To create one repository per business object (data class) is the most basic way of implementing this pattern. Asides from the basic *CRUD*-Operations [2], each repository should provide specialized object retrieval methods that are only valid for the business model it was programmed for. For example,

[2] Create, read, update, delete

in a booking system there may be a data class `Order`. A repository for this class may then provide a method called `getOrdersByUsername`. In general only methods that are actually needed should be implemented, which is described in the *YAGNI*-principle [3].

2.2.2 Model-View-Controller

2.2.3 Model-View-Presenter

2.3 Architecture Patterns

2.3.1 Multi-Layer Architecture

2.3.2 Multi-Tier Architecture

2.3.3 Client-Server

2.3.4 Publish-Subscribe

2.4 Software Project Management

2.4.1 Waterfall

2.4.2 Scrum

2.4.3 Extreme Programming

[3] You aren't gonna need it

3 Methods

3.1 Domain Description

As stated before the application should provide a time-tracking solution for commercial projects. Therefore it is divided into three general areas:

- **Project**: A project is the overall term for a distinguishable amount of work that is done for a specific customer.
- **Phase**: Each project exists of one or more phases that are used to monitor the progress of the project.
- **Activity**: A activity represents a workload that is done by a specific member in a specific phase for a specific project.

In order to be able to safely administer the system, three roles of members exist:

- **Administrator**: This member is responsible for creating and deleting users, resetting passwords and is allowed to perform all actions to project, phases, ect.
- **Project Leader**: A project leader has the responsibility of maintaining project specific data, such as description, phases and assigning project members. He is *not* allowed to create new users. After creating a new project, one is automatically project leader. A project leader is allowed to monitor the statistics of his projects including the workloads of the project members
- **Member**: As a simple member one is allowed to add, delete and modify activities created by the user. A member is only allowed to perform this actions for projects he is assigned to. He is also only allowed to monitor his own statistics.

3.1.1 Use-Cases

General Use-Cases

1. **Login:** After starting the program, a login prompt shall show up. The user provides his credentials which will be compared to the saved credentials in the database. If they are correct, he is forwarded to the main program.
   The user has the possibility to reset his password by activating a button and providing his email-address. A new password will be sent to the address, if a user with this address is known by the system.
2. **Starting / Stopping activities:** Through a banner at the top of the program each user can select a project he is assigned to and furthermore a phase of this project. After providing a name of the activity he can start the activity by clicking on a start button. The same button can be used to stop the activity. After stopping (or closing the application), the activity is complete and is added to the database and project statistics.
3. **Personal statistics:** Utilizing the associated button of the left side bar the main panel lists all project the user is assigned to and the time he spent working for these projects. At the top of the main panel he can choose the period he wants to see. After clicking on one of the entries a statistic of the working time for each phase of the project is shown. By clicking at a phase all activities in this phase are listed and can be modified and deleted, furthermore a new activity can be created here too. By clicking on a button at the top the user can switch the view to the previous view.
4. **Create and manage projects:** Each user can create new projects by using the associated button of the left side bar. The main panel then lists all projects which project leader he is. After clicking on a entry, the main window shows the details of the project, such as the description, the created phases and the project members. He is able to perform all necessary operations associated with the project in this view, as well as declaring a new project leader.
5. **Settings:** Utilizing the associated button of the left side bar the main panel provides means to change the password, email-address and name.

Project leader Use-Cases

1. **Project statistics:** Other than normal users, a project leader can monitor the statistics of all of his projects by using the according button on the side bar. The main panel lists all projects he is leader of. After choosing one project he can choose the period he is interested in at the top of the panel. At the left side of the main window now all project members are listed. After choosing one or more project members, the time they spent working for the different phases of the project is listed at the center.

Administrator Use-Cases

1. **Manage Projects:** As administrator it is possible to manage *all* projects as he would be project leader.
2. **Manage Users:** By clicking on the associated button, users can be managed, e.g. changing their passwords or create new users.

The persistent storage has a quite simple structure, as can be seen in figure 3.1. It is notable that the roles are described as text and therefore are human-readable.

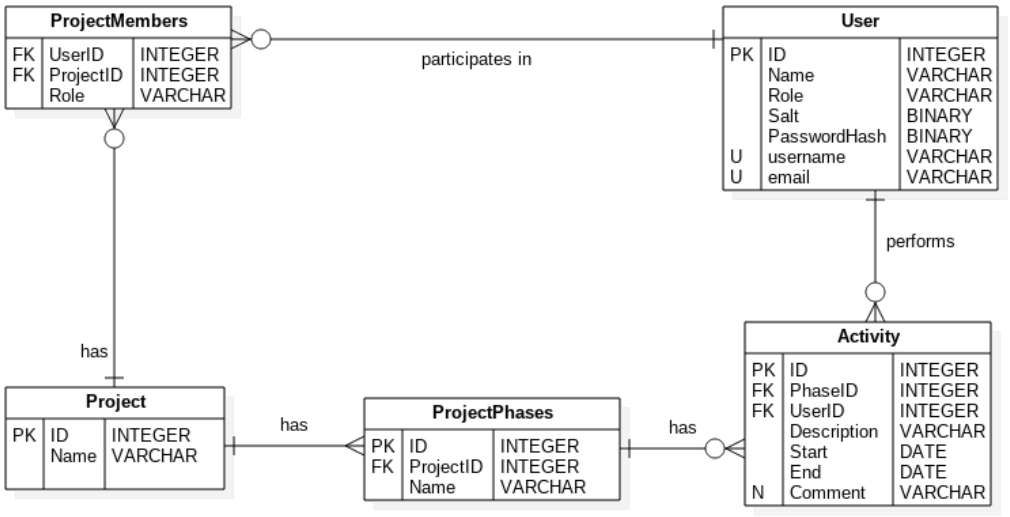

Figure 3.1: Database diagram

3.2 Phase 1: One Databse and UI

3.2.1 Monolithic Code Samples

3.2.2 Best-Practice Code Samples

3.3 Phase 2: More Databases / UIs

3.3.1 Monolithic Code Samples

3.3.2 Best-Practice Code Samples

4 Results

4.1 Lines of Code

4.2 Programming Effort

4.3 Efficiency and Performance

4.4 Touched and Edited Files

4.5 Readability

5 Discussion

5.1 Programming Overhead

5.2 Use of OOP in General

6 Conclusion

7 Future Work

7.1 Physical Seperation of Tiers

7.2 MVC-Frameworks

7.2.1 Spring Framework

Alessandro Garcia et al., 2006

List of Figures

List of Tables

Bibliography

Alessandro Garcia, Cláudio Sant'Anna et al. (2006). "Composing design patterns: a scalability study of aspect-oriented programming." In: *AOSD '06 Proceedings of the 5th international conference on Aspect-oriented software development*, pp. 109–121 (cit. on p. 21).

Castagna, Giuseppe (1997). *Object-oriented programming. In: Object-Oriented Programming A Unified Foundation. Progress in Theoretical Computer Science*. Birkhäuser Boston. DOI: 10.1007/978-1-4612-4138-6_3" (cit. on p. 3).

Ralph E. Johnson, Brian Foote (1991). "Designing Reusable Classes." In: (cit. on p. 4).