Stephan Valentan

How the use of design patterns impacts code quality: A controlled experiment

Bachelor's Thesis

Graz University of Technology

Knowledge Technologies Institute
Head: Stefanie Univ.-Prof. Dr. Stefanie Lindstaedt

Supervisor: Dipl.-Ing. Ass. Prof. Roman Kern

Graz, September 2017

This document is set in Palatino, compiled with pdfLaTeX2e and Biber.

The LaTeX template from Karl Voit is based on KOMA script and can be found online: https://github.com/novoid/LaTeX-KOMA-template

Statutory Declaration

I declare that I have authored this thesis independently, that I have not used other than the declared sources/resources, and that I have explicitly marked all material which has been quoted either literally or by content from the used sources.

Graz, _____ _____

Date Signature

Eidesstattliche Erklärung[1]

Ich erkläre an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst, andere als die angegebenen Quellen/Hilfsmittel nicht benutzt, und die den benutzten Quellen wörtlich und inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Graz, am _____ _____

Datum Unterschrift

[1] Beschluss der Curricula-Kommission für Bachelor-, Master- und Diplomstudien vom 10.11.2008; Genehmigung des Senates am 1.12.2008

Contents

1. Abstract **1**

2. Introduction **3**

3. Background **5**

3.1. Fundamentals of Object-Oriented Programming 5
3.1.1. Encapsulation 5
3.1.2. Inheritance 6
3.1.3. Polymorphism 7
3.1.4. Single Responsibility Principle 7
3.1.5. Open Closed Principle 8
3.2. Design Patterns and Architectural Patterns 8
3.2.1. Repository 9
3.2.2. Dependency Injection 10
3.2.3. Model-View-Controller 10
3.2.4. Model-View-Presenter 11
3.2.5. Client-Server 13
3.2.6. Peer-to-Peer 14
3.2.7. Multi-Tier and Multi-Layer Architecture 14
3.3. Technologies 17
3.3.1. Java 17
3.3.2. Swing 17
3.3.3. JavaFX 18
3.3.4. Relational Databases and PostgreSQL 18
3.3.5. MongoDB 19
3.3.6. Caching 19

4. Methods **23**

4.1. Domain Description 23

4.2. Experiment set-up . . . 25
4.2.1. Best practice implementation . . . 26
4.2.2. Ad-hoc implementation . . . 27
4.3. Experiment phases . . . 27
4.3.1. Phase 1: Basic requirements . . . 27
4.3.2. Phase 2: Support of a different graphical user interface library . . . 28
4.3.3. Phase 3: Support of a different data tier implementation 28
4.4. Phase 1: Basic Requirements . . . 29
4.4.1. Best-Practice Code Samples . . . 29
4.4.2. Ad-hoc Code Samples . . . 32
4.5. Phase 2: Support of a different graphical user interface library 35
4.5.1. Best-Practice Code Samples . . . 35
4.5.2. Ad-hoc Code Samples . . . 36
4.6. Phase 3: Support of a different data tier implementation . . . 36
4.6.1. Best-Practice Code Samples . . . 37
4.6.2. Ad-hoc Code Samples . . . 37

5. Results **39**
5.1. Lines of Code . . . 39
5.2. Touched and Edited Files . . . 41
5.3. Average line count per file . . . 42

6. Discussion **43**
6.1. Meaning and significance of the file count . . . 43
6.2. Meaning and significance of the line count . . . 43
6.3. Maintainability . . . 44

7. Conclusion **45**

A. Source Code **47**

Bibliography **48**

1. Abstract

While design patterns are proposed as a standard way to achieve good software design little research is done on the actual impact of using these strategies on code quality. Many books suggest that their methods increase flexibility and maintainability however they often do not provide any evidence.

This bachelor thesis intends to empirically prove the hypothesis that the use of design patterns improves code quality.

To gather data about the code two applications are implemented which need to meet the same requirements. While one application is developed following widespread guidelines and principles proposed by the object oriented programming, the other is implemented without paying attention to the topics of software maintenance. After complying to the basic requirements new features are implemented in two additional phases. At first a new graphical user interface is being supported, then another data tier is added.

The results show that the initial effort of implementing the program version following object oriented programming guidelines are significantly higher in terms of code lines and necessary files. However during the implementation of additional features less files needed to be modified and during one phase traversal significantly less code was needed to be written while not performing worse in the other.

2. Introduction

The topics of design patterns and architectural styles are vital to the subject of software development. They empower inexperienced programmers to write maintainable applications and provide guidelines that may be adopted but are implemented in each application in one way or another.

Many books are available on this area of expertise, on how to write maintainable and extensible applications. The books are written by software engineers with many years of experience , like Martin Fowler with his book *Patterns of Enterprise Application Architecture* [1], Robert C. Martin with *Agile Software Development. Principles, Patterns, and Practices* [2] or *Design Patterns. Elements of Reusable Object-Oriented Software* by John Vlissides, Richard Helm, Ralph Johnson and Erich Gamma [3].

The same topics are to a certain degree taught at universities however they all lack one substantial flaw: While they propose some pattern, explain when it is applying and how to implement it, they do not provide any evidence that when using the pattern in a real-world application there is an advantage to gain. Many patterns claim that they improve the extensibility and increase the decoupling of code however in most of the example code only the basic implementation is showed. The impact on the code when a extension is actually implemented is not being shown by the books which leaves the reader to believe the promises made by the author.

While the patterns are in extensive use and thus have empirically proven themselves useful, scientific research on this topic is far from being exhaustive. A study of 2008 was published at IEEE that tried to cover the effect of using design patterns in software development. This study came to the

[1] IBAN: 978-0321127426
[2] IBAN: 978-0135974445
[3] IBAN: 978-0201633610

result that while the implementation of best practice methods in many cases enhanced some quality aspects while at the same time performing worse in other factors. It concluded that the use of patterns should be considered carefully as they lead to a certain trade-off and stated clearly that more research is necessary as some effects could not being explained. [**quality**].

Another study published covering a related subject came to similar results, sugesting that the use of established design patterns in most of the cases leads to an improvement of code quality in some aspects while weakening others [**quality2**].

It is hardly possible to perform scientific research on real-world software projects as there are only little ways for serious comparison. The patterns only provide a coarse guideline, the actual implementation largely depends on the use case and will vary substantially from application to application. Because of this the research is done partially via *Controlled Experiments*, as is this thesis. Therefore it has to be said that this kind of studies are considered at least partially theoretical as a number of assumptions have to be made. The development of an application is highly subjective and depends on the coding style and experience of the engineer, code review policies can influence the quality as well.

However this thesis intends to give insights on how the use of well-established architectural and design pattern that claim to improve extensibility impact the code quality when this extensions are in fact implemented. This is done by developing two applications that are both designed to meet the same use cases and requirements. One is programmed by using established patterns while the other is developed without considerations regarding code quality while implementing the same basic program structure in order to improve comparability.

After satisfying the basic requirements new features are being implemented, namely another type of persistence, a different kind of user interface and lastly means for caching data in the application to reduce the need of database queries. After realizing these changes in both programs the costs of implementing can be compared in both quantitative and qualitative ways.

3. 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.

3.1. 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* [2, p.80] [4, p.14].

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*.

3.1.1. Encapsulation

To encourage refactoring [1] 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

[1] Refactoring describes the process of re-writing classes to improve increase cohesion, decrease coupling, separate concerns, modularize system concerns, shrink our functions and classes, choose better names and so on. [9, p. 172].

class to enforce invariants [2]. This means that it hides all information not relevant to others as they are only implementational details and thus not relevant for consumers. As other classes now rely on an interface rather than concrete implementations the code is called *loosely coupled*. Modifying the internals does not break interdependencies which encourages programmers to perform refactoring resulting in improved code quality and flexibility.

Many programming languages provide different access levels varying from visible to all others (in the case of Java: *public*), accessible only within the class (*private*) or visible from within and derived classes (*protected*). The latter access level can be problematic as they can effectively break the encapsulation. When many protected variables are declared it may happen that subclasses depend strongly on them. If this happens a change of the base class could lead to necessary changes in the subclasses, which is exactly what encapsulation tries to avoid and thus effectively breaks it[4, p.19].

3.1.2. 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 trough a *is-a*-relationship. The child is therefore a superset of the base class as it can be extended to meet the new requirements. This concept is important as it encourages developers to reuse existing code and in that way lower the risk of programming errors [12] and is often called *white-box-reuse* as the internals of the base class in general are visible to the programmer [4, p.19].

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 [3]. 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

[2] A invariant is some kind of requirement that must be fulfilled at all times before and after a method call. An example of such a invariant could be the descending sort order of an integer list.

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.

3.1.3. Polymorphism

Polymorphism is closely tied to the term of *dynamic binding*, which describes the process that a method call is not statically bound to an implementation at compile time but associated with an actual implementation at run-time. Polymorphism extends this process by stating that objects of the same interface can be used for each other and swapped at compile time. [4, p.14].

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 [7]. 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 instance.

The other important kind is *overloading* polymorphism which is used to provide methods with the same name but different signatures (and thus attributes) [8]. 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. [3].

3.1.4. 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

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

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 [11]. 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. The modifications may introduce errors to the other functionality, which could be avoided if the program fould conform the principle. This would require placing each responsibility in a independent module.

3.1.5. Open Closed Principle

The Open-Closed-Principle states that each class should be open to extension and closed for modification [10]. As already written code is assumed to be well tested and working as intended it should be avoided to modify it 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 detect.

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.

3.2. Design Patterns and Architectural 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 John 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 3.2.7) is more common.

The main difference between architectural and design patterns lies in different phases they are applied at. Architectural patterns provide high-level strategies providing a big picture often describing the interaction between so-called *components*, which can vary widely in its complexity and each of them providing a well-described service and fulfilling a certain purpose. Design patterns focuses more on the actual implementation and describes a way to solve a problem on class and interface level.

3.2.1. Repository

The repository design 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 [4]. All actions dealing with specialized database operations, object retrieval and others are hidden by a repository, because of this it is considered part of a data-abstraction-layer (see section 3.2.7). A repository is often used if only basic operations with the data source are necessary. Other patterns and frameworks supporting advanced functionality and are available however are not subject of this thesis.

To create one repository per business object (data class) is the most basic way of implementing this pattern. Asides from the basic *CRUD*[5]-Operations , each repository should provide specialized object retrieval methods that are only valid for the business model it was programmed for. The addition

[4] https://martinfowler.com/eaaCatalog/repository.html
[5] Create, read, update, delete

of more complex object retrieval methods differentiates a data mapper [6] from a repository.

For example, 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 [7].

3.2.2. Dependency Injection

Dependency Injection is a very common and important pattern that is used in object oriented programming. It serves the Single-Responsibility-Principle in saying that a object should not be responsible of creating its own dependencies, instead some third supervising party *injects* the dependency either using constructor parameters or associated setters [9, p.159]. Furthermore it encourages the use of interfaces and therefore helps to switch between implementations of dependencies. Especially in the terms of testing this is considered a huge advantage as it enables testers to introduce mock objects [8].

3.2.3. Model-View-Controller

The Model-View-Controller (MVC) is a pattern that describes a separation of a program in three layers, it is considered both a design and a architectural pattern, depending on the point of view. Its goal is to improve flexibility and extensibility of an application which can help to improve maintainability. It consists of three components, from which the *model* contains the business logic, the *controller* processes interactions and the *view* presents some kind of user interface [5].

[6] https://martinfowler.com/eaaCatalog/dataMapper.html

[7] You aren't gonna need it. It means that it should be avoided to implement functionality in advance as it is not clear what the actual requirements will be.

[8] https://javaranch.com/journal/200709/dependency-injection-unit-testing.html

This pattern is not very well defined and is subject of constant discussions. While it is commonly agreed upon that the application should be split in the three layers, the data flows and connections between the components are described in many different ways. The original concept of MVC developed for Smalltalk as described before the model holds logic and data of the application [4, p.4]. Views subscribe at the model for notifications on changes of data in order to be able to update and refresh themselves. The controller is responsible of managing user interaction, which includes switching between views and forwarding requests to the model. This situation can be seen in Figure 3.1. This direct interaction between model and view is often not desirable, especially in the context of web applications, which leaves it deprecated today.

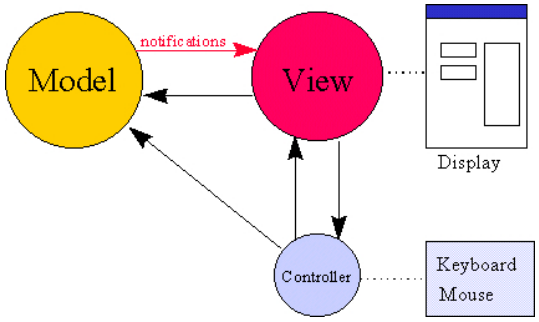

Figure 3.1.: Classical Model-View-Controller concept as used in Smalltalk. Model and View know each other.

Source: https://www.mimuw.edu.pl, accessed 5.9.2017

3.2.4. Model-View-Presenter

The Model-View-Presenter (MVP) pattern is similar to the MVC-model as it shares the concept of separating the application into three layers. The responsibilities between the two patterns differ considerably, as can be seen in Figure 3.2. The view only communicates with the presenter and does not know about the model. The presenter handles user input and requests, gathers data from the model, updates it and finally is responsible for refreshing the view with new data. This implies that the model does not

know about the existence of view or presenter and at least in the original concept of MVP all business logic is placed in the presenter classes [13]. It is important to note that it is vital for the use of this pattern to implement it using interfaces. This ensures that its main goal, easy testability, can be reached easily.

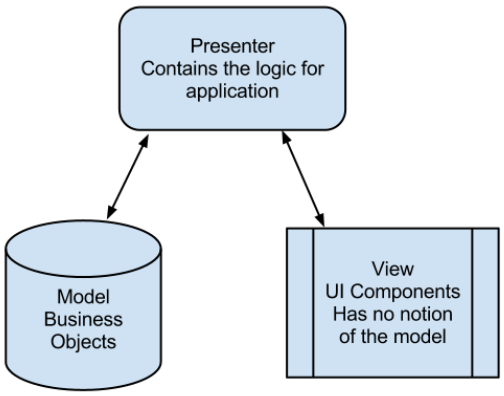

Figure 3.2.: Diagram of Model-View-Presenter. While the view again only holds the means for displaying the data and interacting, in opposite to the MVC-model it does not know about the model but only corresponds with the presenter. The presenter is responsible for gathering the necessary data from the model and forwards it to the view. The model does not know about the presenter.

Source: https://commons.wikimedia.org/w/index.php?curid=19794348, accessed on 5.9.2017

After some years Martin Fowler drafted two design patterns that are closely connected to MVP. In his blog entry [9] he explained that he felt the need to split the MVP-pattern into two resulting in the following concepts, briefly described in Figure 3.3.

Supervising Controller

In this concept it is the views responsibility to perform data synchronization using shared classes between it and the model. This simplifies the tasks to

[9] https://martinfowler.com/eaaDev/ModelViewPresenter.html

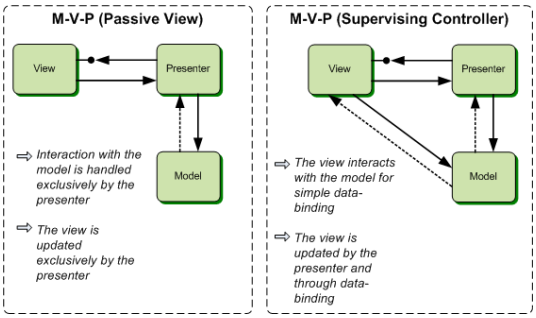

Figure 3.3.: Basic diagrams of the Passive View (left) and Supervising Controller patterns (right). In the former as little logic as possible is placed in the view making it easy replaceable by a mock object for testing. Synchronization logic needs to be placed in the Presenter. In the latter view and model are connected trough data bindings empowering these two to perform synchronization tasks on their own. This simplifies the role of the presenter.

Source: https://msdn.microsoft.com/en-us/library/ff709839.aspx, accessed 5.9.2017

be performed by the presentation layer while reducing the level of testability possible.

Passive view

In the concept of passive view the model is not connected with the view, which in terms only corresponds with the presenter. It is desired to minimize the logic placed in the view, leaving it (as the name describes) passive and therefore easy to mock. This is especially useful for automated testing, as it allows the tester not only to test the basic logic placed in the presenter but also its synchronization capabilities. The drawback of this clearly lies in the additional responsibilities that need to be met by the presenter [10].

3.2.5. Client-Server

The Client-Server model describes a way to separate tasks within a network. While the client, which can be any kind of computer program, requests a

[10] http://www.wildcrest.com/Potel/Portfolio/mvp.pdf, accessed 4.9.2017

resource from the server. It is designated to be used within the internet and is still on of the cornerstones of current web infrastructure [6, p. 3ff] . A prominent example of the client-server model is the Domain-Name-System (DNS). When a user types in the domain name of a service he wants to access, the browser needs to translate the name into a IP-Address he can work with. In order to do that he sends a request to a DNS-Server, making him a client. While he does not need to know how exactly the resolving takes places he requires the server response to have a certain format described in the communication protocol.

There are several benefits of using this architecture. For starters it centralizes the administration which lowers the efforts of managing access rights, accounts and so on. Because the important data is in one place it is easier to create backups and ensure security policies. However a server is considered a *Single Point of Failure*, meaning that a whole service will be unavailable if the server for any reason is not reachable, making it a worthy target of attacks. While the impact depends on the actual service built, scalability and extensibility are important factors. It is easier to extend the features provided by a service when using the client-server-model, while it reduces scalability as the hardware needs to be upgraded from time to time to meet a increased number of requests directed to the server.

3.2.6. Peer-to-Peer

Peer-to-Peer (P2P) infrastructure is the counterpart of client-server-models. No central server exists which means that each of the clients are equally featured, each client has the means to request and provide resources. P2P-Networks are considered more robust and scalable than Client-Server set-ups, as the failure of one node does not lead to the breakdown of the whole system. In picture 3.5 a typical P2P-setup can be seen.

3.2.7. Multi-Tier and Multi-Layer Architecture

Multi-Tier architecture is mostly mentioned in the context of web applications. In general it describes the separation of responsibilities on different

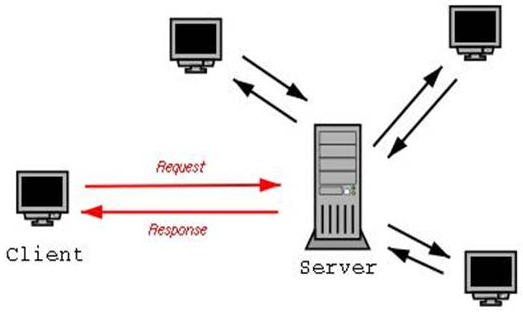

Figure 3.4.: Client-Server setup. The client sends a request to a server by messages defined in the protocol. The processing of the request is transparent to the client, it does not know about actions taken by the server and awaits a response in a certain format.

Source: https://abcnetworking.wikispaces.com/Client+Server+Networking, accessed on 6.9.2017

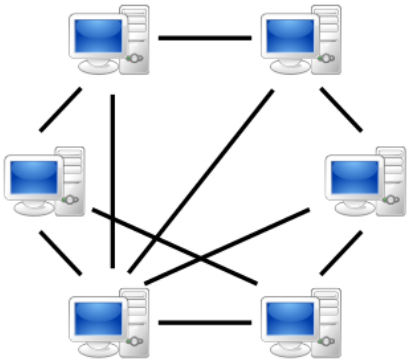

Figure 3.5.: Peer-to-peer setup. Each peer has the same rights, no central server exists.

physical places [11]. This is the main difference to a multi-layer-system where the responsibilities are separated but placed at the same process which indicates the use of interfaces.

A 3-Tier-Architecture nowadays refers mostly to the typical set-up of a web application as it can be seen in Figure 3.6. It has to be noted that it is very common to use multi-tier and multi-layer architecture mutually. For example, the logic tier in the figure could use a data-access-layer to abstract the communication with the data tier.

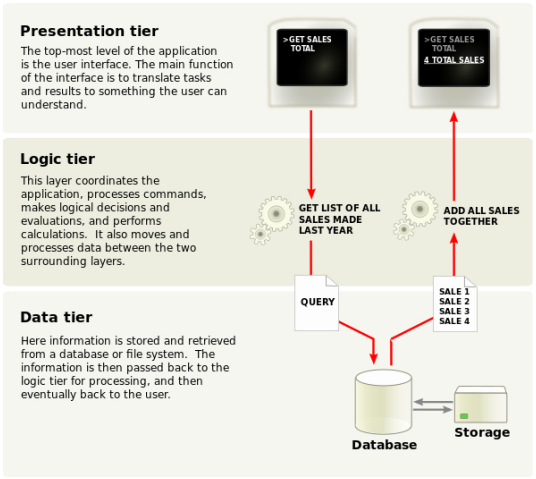

Figure 3.6.: Multi-Tier set-up. The tiers are physically separated.

Source: https://en.wikipedia.org/wiki/Multitier_architecture, accessed on 6.9.2017

[11] In this context physical separation means that there has to be going on any kind of communication between independent processes within the same machine or connected trough the internet.

3.3. Technologies

3.3.1. Java

This project is written in Java, an object oriented programming language released by Sun Microsystems in 1995 [12]. In 2010 the company Oracle bought it and is developing and distributing the language since. It is split up into two main components, its development kit (JDK) and its runtime environment (JRE). The JRE is providing a virtual environment called Java Virtual Machine (JVM) which is used to run so-called Java Bytecode which is comparable to x86 assembler. This bytecode can be compiled from plain Java programs as well as other languages such as Scala or Groovy. This approach helps reaching one major goal of the language: portability. Java bytecode can be executed on every platform, as long as a JRE is installed, which nowadays includes mobile phones, embedded platforms and with reduced functionality even smart cards [13].

The JRE uses an interpreter and a just-in-time (JIT) compiler to generate native machine code that can be executed of the platforms processor. A JIT-Compiler generates the necessary machine code at runtime for some parts of the application. This can lead to significant performance improvements compared to the exclusive use of an interpreter.

As of September 2017 according to the TIOBE-Index it is the most popular programming language with a share of more than 12% [14].

3.3.2. Swing

Swing is a Graphical-User-Interface (GUI) Library for Java and was first published in 1997. It is rendered directly by Java thus it is not necessary for a platform to provide specific controls. However this leads to a significant difference in appearance to applications natively written for the platform

[12] https://java.com/en/download/faq/whatis_java.xml, accessed 7.9.2017
[13] http://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html, accessed 7.9.2017
[14] https://www.tiobe.com/tiobe-index/, accessed 7.9.2017

as some control concepts may differ or buttons may be placed at other positions. This problem is bypassed by the introduction of different *Look-and-Feels* which intend to adopt the application to be as similar to a native program as possible.

3.3.3. JavaFX

JavaFX is a framework [15] for GUI programming in Java and was released in 2014, replacing Swing as the main method for graphical programming. In opposite to Swing it implements its own multimedia-stack able to access graphics card feature through OpenGL and Windows Direct3D.

3.3.4. Relational Databases and PostgreSQL

PostgreSQL is a open source object-relational database management system (ORDBMS) that was first published in 2002. Other than many database management systems it is focused on maintaining a good SQL standard compliance and is currently ranked as the 4th most popular engine. [16] [17]

Almost all database systems that are considered as relational use the Structured Query Language (SQL) or a dialect of it as language for querying and manipulating the database. Such a system is characterized by structuring the data into *tables*, which consists of *columns* and *rows* (also: *records*). Many systems require each row to have a unique identifier, also called *primary key*. This key can consist of one or more columns per row, if more columns are considered it is also called *composite key*.

If it is wanted to refer from one table row to the a record from another table a *foreign key* is used which corresponds to the primary key of the original table. This is vital for maintaining the so-called *referential integrity*,

[15] The main difference between a framework and a library is the so-called *Inversion of Control*. While a library is called by the programmer, when using a framework it is in control and calls methods provided by the developer.

[16] https://www.postgresql.org/about/, accessed 7.9.2017

[17] https://db-engines.com/de/ranking, accessed 7.9.2017

which indicates that all references have valid targets. Depending on the policy set in the database management system or specified in the foreign key definition of the table the deletion of a row that is referred by another record could be prevented.

3.3.5. MongoDB

As a prominent example for NoSQL[18]-databases MongoDB is a database engine written in C++ and published in 2009. In opposite to relational databases it persists the data in *documents* which are structured in a JSON[19]-like manner.

Documents can be stored in *collections* which are comparable to tables in SQL-databases. While all rows in tables possess the same columns this is not necessarily true for collections. Each document in a collection can hold arbitrary attributes which improves the flexibility while as a trade-off reducing data integrity. In opposite to relational databases in MongoDB foreign keys cannot be created, however each document is provided with a built-in unique identifier.

3.3.6. Caching

Caching is the process to hold often needed data in memory to reduce the time necessary for gathering. While most of modern database management systems already provide advanced caching techniques out of the box, the application could be physically separated from the server by a low-bandwidth network. Therefore client-side caching may be necessary for reducing latency.

[18] Not only SQL

[19] JavaScript Object Notation, a syntax for exchanging data which is easy to read for humans and efficiently processable by machines. It is build around key-value-pairs which can be nested.

In most of the cases placing the cache in the data access layer (see section 3.2.7) leads to the simplest implementation as all data accesses are passing through here, however depending on the structure and needs of the application it may be necessary to implement the cache in the business code.

Caching strategies

There are two main types of caching policies: *write-through* and *write-back*. When performing write-trough-caching all data is first persisted and then immediately updated in the cache while blocking the operation until all necessary writes are completed. This ensures that at all times a up-to-date copy of the data is available [20].

The write-back-policy on the other hand also caches write accesses without directly persisting it. The write operation onto the data tier is being performed asynchronously, this may then happen when the system load is low and therefore the overall performance may be improved. However it is considered more dangerous in terms of data loss due to system crashes or other errors as the only valid copy of the data is only resided in the cache [21].

Cache Replacement Policies

As memory is a scare resource it is necessary to limit caching space and thus not all data can be cached at the same time. This introduces the necessary of means to decide which cache entries are to abolish, this mechanism is called a *Cache Replacement Policy*.

The best policy clearly would be to replace the cache entry that will be needed again the latest. As this is information that is not available the associated **Bélády's algorithm** is considered theoretical and is only used for measuring and comparing the performance of other policies.

[20], accessed 10.9.2017

[21] http://www.computerweekly.com/feature/Write-through-write-around-write-back-Cache-explained, accessed 10.9.2017

There are several possibilities which differ significantly depending on the data access pattern (list not exhaustive):

- **First-In, First-Out (FIFO)**: When caching with the FIFO policy as soon the cache runs full the *oldest* entry will be replaced. This is typically implemented as a Queue because it reduces the efforts to keep track of the entries age.
- **Least Recently Used (LRU)**: This algorithm keeps track of the last reference to cache entries. When a replacement is necessary it frees the page which reference dates back the longest.
- **Least Frequently Used (LFU)**: Research has shown that LRU performs poorly for some access patterns, so instead of deciding which page is to replace upon the access time, LFU decides upon the number of accesses.
- **Most Recently Used (MRU)**: Access patterns exist where it is preferred to abolish the entry that was accessed most recently, an example for this scenario would be the sequentially scanning of a file. The implementation is very similar to LRU.
- **Random**: Interestingly research has shown that random replacement can be considered a good strategy as well, especially as it is easy to implement and efficient. Most of the more advanced policies may perform better with certain access patterns while performing worse in others. In many cases it is not possible to predict how the accesses will take place which introduces the risk of experiencing a unfavourable pattern. Random replacement is not sensitive to this problems while providing solid performance.

Cache invalidation

In the context of cache invalidation a quote which is said to come from Phil Karlton is especially famous:

> There are only two hard things in Computer Science: cache invalidation and naming things.

A Cache invalidation mechanism is necessary if the data tier can be updated from more than one location or clients which is especially in the terms of

web application the standard case.

Most caching strategies therefore feature a so-called time based expiry. It provides each cache record with a time-out after that it will be evicted [22]

Another method is active cache invalidation which evicts either cache entry or the whole cache after a change in the data tier. To support this some kind of messaging mechanism is necessary to let the persistent storage notify all clients about the update.

In the case of multiple clients it has to be noted that it could be desirable to store additional version information for the table rows, for instance a incrementing counter or a hash of the data set. Because of the possibility of multiple clients updating the same row concurrently from different locations race conditions can occur that lead to the overwriting of newly entered data. Even with active cache invalidation it cannot be guaranteed that the update notification reaches the second client in-time.

[22] http://tutorials.jenkov.com/software-architecture/caching-techniques.html, accessed 10.9.2017

4. Methods

In order to compare the effects of using design patterns as stated before an application is being developed. This section describes the requirements of the program in detail and presents how the two versions are being implemented.

4.1. Domain Description

The application provides a time-tracking solution for small businesses. The basic work-flow includes users (members), projects, project phases, project members and activities.

- **User**: A user is considered employee of the firm. He can participate in existing projects or create new projects on his own.
- **Project**: A project is the overall term for a distinguishable amount of work that is done for a specific customer. A project has one or more project leaders and zero or more project members.
- **Project Phase**: Each project exists of one or more phases that are used to monitor the progress of the project. Project leaders are able to create, adopt and delete phases for their projects.
- **Project Member**: A user is considered project member of a specific project as soon he participates or leads it. Two roles exist, he can either be leader or member. 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. As a simple member one is

allowed to add, delete and modify activities created by the user and is only allowed to perform this actions for projects he is assigned to. He is also only allowed to monitor his own statistics for projects he participates.

- **Activity**: A activity represents a workload that is done by a specific member in a specific phase for a specific project.

Use-Cases

To specify in depth how the program should work the following use-cases are set.

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. New users can register by clicking on a button in the login-view by providing his personal data and a password. A nickname needs to be picked that is not already used by another user. After successful registration the user is able to login without further action.
2. **Starting / Stopping activities:** A banner at the top of the applications main view provides means to add new activities. In order to do so the user picks one of his associated projects from a drop down menu. After selecting one a second drop down menu is populated with the available project phases. After selecting a belonging project phase he is able to click the start button. After starting a timer indicates the current time spent in the chosen phase for the activity. After clicking the stop button a input mask should pop up for the user to provide additional information of the activity he just finished, namely the description and (optional) some comments. Then the timer is being reset and a new activity can be started.
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, for instance it should be possible to filter the projects for activities created in the last month.

To review a specific project in detail after a double-click onto one of the project-entries all phases of the project should be listed, again with the time spent in each phase. The filtering of a specific period is possible too. By double-clicking at a phase all activities by the user 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. As project leader he is able to create, modify and delete new phases. He is furthermore able to promote simple members to project leaders and adding or excluding members from the project. In the project panel all project he is member of are listed too. The user is able to leave associated projects. When a phase is deleted, all activities related with this phase are removed too. The same applies for the deletion of a whole project.
5. **Project statistics:** As project leader it is possible to review the overall performance of a project. In opposite to the personal statistics now actions from *all* project members are visible, broken down by phase and period as explained in the point *personal statistics*. In addition to the filter possibilities there it is also possible to review the performance and activities of a specific project member.
6. **Settings:** Utilizing the associated button of the left side bar the main panel provides means to change the password, email-address and name.

4.2. Experiment set-up

Two applications were developed both serving and meeting the requirements stated in the previous section. The programs must not differ in its features, behaviour or appearance, the difference has to be in the code.

4.2.1. Best practice implementation

In the first application well-known and widely spread patterns are being applied. The data access to any persistent storage is abstracted by implementing the repository-pattern (see section 3.2.1). The program is structured according to the Model-View-Controller pattern (section 3.2.3), layers are connected by interfaces rather than concrete implementations. Each of the use-cases listed before corresponds to a own set of model, view and controller to keep the program structure clear. In addition to the classes described a main model, controller and view exists. These classes hold instances of all other use-case-specific classes. This allowed to pair the models with the controllers and the views with the controllers. Furthermore the main model holds the reference to the data access layer. Each specific model has therefore a reference to the main model. This is also needed for switching between the five big areas of the application.

The **model** holds all of the business logic and controls the program flow. It utilizes the data access layer for retrieving and saving data from and to the persistent storage. Furthermore it is responsible for instantiating controller and view, generating responses to requests after user actions. The requests are forwarded from the view by the controller. This means that each model has a reference to the corresponding controller instance of the use-case.

The **view** is only responsible for presenting the data to the user and providing means to interact with the program trough buttons and other controls. However the view does *not* know about underlying data structures and is therefore only supplied with plain-old-java-objects (POJOs). This setup provides high flexibility on how the view is implemented and any front-end can theoretically be supported. The view does not hold any business logic, does not react to user actions on its own and forwards requests and interactions to the corresponding controller.

The **controller** is the layer between model and view. As stated in the description of the view, all user interactions are routed to the controller and processed here. Any actions, for instance clicks, are here translated into requests to the model. All responses from the model are translated into POJOs and with whom the view is being populated. The controller is also responsible for switching views according to the user actions.

As stated before the **data access layer** is realized by implementing the repository pattern, access to the repositories is provided through the main model. Again only a interface to the repositories is returned to maximize flexibility.

4.2.2. Ad-hoc implementation

To ensure comparability the structure described in the previous section applies to the ad-hoc implementation as well. However two serious differences are realized: Firstly the different layers are *not* connected by interfaces but rather by concrete implementations. Secondly no data access layer is being used, the functionality is provided by the data classes. Therefore asides from the basic getter and setter methods static methods for data retrieval and updating the persistent storage are supplied. This is shown by the sample code in section 4.8.

4.3. Experiment phases

After the setting of basic requirements more and more modifications and extensions being added. This should simulate an ongoing discussion process with the (virtual) stakeholders of the application, the extensions are called *phases* and are described in detail in this section. The new requirements can be seen as a refinement of the use cases specified in the previous sections.

4.3.1. Phase 1: Basic requirements

For starters the application should provide a graphical user interface by using the Swing-library (section 3.3.2) and as data tier a PostgreSQL database (section 3.3.4) should be used.

4.3.2. Phase 2: Support of a different graphical user interface library

After meeting the basic requirements the virtual stakeholders are requesting the support of a different GUI library, namely JavaFX (section 3.3.3). By providing application start-up arguments the user should be able to decide which GUI should be presented to him. All other use-cases and requirements regarding the work flow apply nevertheless.

4.3.3. Phase 3: Support of a different data tier implementation

During the process the stakeholders request that the application should add support for MongoDB (section 3.3.5). They want to allow the user to choose between the two databases as both implementations have advantages and disadvantages.

4.4. Phase 1: Basic Requirements

In this section it is explained in detail how the two applications meet the requirements. This is only a short overview, the whole project is available on Github. For more information on the project structure see the appendix section.

4.4.1. Best-Practice Code Samples

Database Access

As a entry point for all data tier accesses, a class `DBManager` was created. It defines all available persistent storage options and is implemented as a singleton. The class itself is abstract, because each data access layer implementation needs to provide implementations for providing the repositories.

```
public abstract class DBManager implements RepositoryFactory {
  private static final String DRIVER_POSTGRES = "org.postgresql.
   Driver";
  private static DBManager instance;

  public static DBManager get(String driver, String url, String
   username, String password) {
    switch(driver) {
    case DRIVER_POSTGRES:
      if(instance == null) instance = new DBManagerPostgres(driver
   , url, username, password);
      break;
    default:
      throw new UnsupportedOperationException("Not yet implemented
   ");
    }
    return instance;
  }
}
```

Listing 4.1: DBManager singleton class. Used for instantiating the implementation of the chosen interface.

The constructor of the PostgreSQL-database manager now initializes the connection based on the arguments provided. The interface `RepositoryFactory` is used to specify which methods each data access layer implementation has to override and which repositories there are. The overridden methods of the implementation invoke the constructor of the corresponding repository implementation and provides any data necessary for their work. In this case it is the Postgres-DBManager itself as it is often necessary for the repositories to load data by using other repositories.

```
public class DBManagerPostgres extends DBManager {
  ...
  @Override
  public UserRepository getUserRepository() {
    return new UserRepositoryPostgres(this);
  }

  @Override
  public ProjectRepository getProjectRepository() {
    return new ProjectRepositoryPostgres(this);
  }
  ...
}
```

Listing 4.2: Code snippet on how a DBManager implementation instantiates its repositories.

In the business code it is now very easy to invoke the methods. The main model holds the only instance of the DBManager, each submodel has a reference to the main model. This process can be seen in listing 4.3.

```
public void login(String username, char[] password) throws
    Exception {
  UserRepository repo = mainModel.db().getUserRepository();
  User u = repo.getByPrimaryKey(username);
  ...
}
```

Listing 4.3: Code snippet showing how repositories are used in business code. The main model holds the single instance of the DBManager. Each sub-model accesses it by using a getter-method.

UI Interaction

As stated before there is a main controller that instantiates the sub-controllers. Because a pairing is necessary between model and controller the main controller provides methods to do so. After the main model has instantiated all sub-models he pairs it with the controller. This process is the process of *dependency injection* as described in section 3.2.2.

```
1  private MainModelImpl(String driver, String url, String username,
       String password, String controllerImpl, String
   frontend) throws Exception {
3    db = DBManager.get(driver, url, username, password);
     ControllerManager.initInstance(controllerImpl, frontend);
5    controller = ControllerManager.getInstance();
     controller.setModel(this);
7
     login = new LoginModelImpl();
9    pairLogin();
     ...
11 }

13 private void pairLogin() {
     login.setMainModel(this);
15   controller.pairLogin(login);
   }
```

Listing 4.4: Code snippet showing initialization code for the controller. The main model retrieves the main controller by calling a getter which is similarily implemented as the DBManager class shown in listing 4.1. After that he calls a method that pairs the sub-model with the sub-controller, a code snippet showing this process can be seen in listing 4.5.

The method call seen in line 5 of listing **??** states a call to the static ControllerManager class. Similar to the DBManager class this is a singleton class that holds a switch case that instantiates the main controller based on the implementation provided by the arguments. After the method calls now the `LoginModel` class knows its main model and its controller.

The same process is repeated for the layer between controller and view.

```
public class MainViewSwing {
  ...
  public void pairLogin(LoginController controller) {
    controller.setView(login);
    login.setController(controller);
  }
  ...
}
```

Listing 4.5: Code snipped showing pairing code between a sub-controller and a sub-view. The pairing between the model and controller layer works in a very similar way.

4.4.2. Ad-hoc Code Samples

In the first phase of the application not many differences can be observed. The flexibility introduced by the patterns shows its effect in the following phases.

Database Access

As stated in section 4.2.2 no repositories are used for the database access. All code associated with the persistent storage is directly placed into the data classes. They provide static methods for retrieve operations and non-static methods for update operations.

As no interface is used the main model instantiates the DBManager implementation directly.

```
public void login(String username, char[] password) {
  User u = User.getByPrimaryKey(username, mainModel.db());
  ...
}
```

Listing 4.6: Code snipped showing database access in the ad-hoc program version. All database code is placed into the corresponding data class, retrieval methods are static and take the corresponding DBManager as parameter. The reference to the DBManager is provided by a getter method of the main model, as in listing 4.3.

```
public class MainModelImpl {
  ...
  private final DBManagerPostgres db;

  private MainModelImpl(String driver, String url, String username
    , String password, String controllerImpl, String
  frontend) throws Exception {
    db = new DBManagerPostgres(driver, url, username, password);
    ...
  }
  ...
}
```

Listing 4.7: Code snippet showing how the DBManager is instantiated. Again no interfaces but rather concrete implementations are used. The singleton class seen in listing 4.1 does not exist in the ad-hoc version.

```
public class User implements DBEntity {
  ...
  public static User getByPrimaryKey(String loginName,
    DBManagerPostgres db) throws Exception {
    ...
  }

  public void insertIntoDb(DBManagerPostgres db) throws Exception
    {
    ...
  }
  ...
}
```

Listing 4.8: Code snippet showing a database access methods in a data class. The retrieval method takes arguments necessary for the query and the database-manager object as parameters and is static, while the update method is not static and therefore performs the operation on the current object instance.

UI Interaction

The only difference between the pairing process of the best practice implementation and the ad-hoc version is that no interfaces are used and thus

the methods take the implementation classes instead of the interface as parameters. noticeable

```
public class MainViewSwing {
  ...
  public void pairLogin(LoginControllerStandard controller) {
    controller.setView(login);
    login.setController(controller);
  }
  ...
}
```

Listing 4.9: Code snippet showing pairing code of the ad-hoc version of the program. It is noticeable that the pairing method takes a concrete implementation rather than an interface as a parameter. This differs from the best practice version seen in 4.5 which takes only the interface as parameter.

4.5. Phase 2: Support of a different graphical user interface library

After adding the second GUI differences between the two implementations starting to be observable.

4.5.1. Best-Practice Code Samples

Due to the use of the interfaces aside from implementing the view itself only very little modifications have to be made. Only the ViewManager-singleton class has to be adopted to support the new view [1].

```
public class ViewManager {
  private static MainView view;

  public static void initInstance(String frontend) {
    assert (view == null);
    switch(frontend) {
      case "standard":
        view = new MainViewSwing();
        break;
      case "javafx":
        MainViewFX fx = MainViewFX.getInstance();
        launchFx(fx);
        break;
      default:
        throw new UnsupportedOperationException();
    }
  }
  ...
}
```

Listing 4.10: View Manager after implementation of a second GUI. Only a case branch has to be added.

[1] The internal structure of the JavaFX-Library needed the creation of a own thread. This is done by the calling `launchFx()`

4.5.2. Ad-hoc Code Samples

Because the controllers do not rely on interfaces but upon actual implementations now each call to a view class has to be differentiated which bloats the code. The following if-else statement is necessary whenever a call to a view class is made. Obviously with more views the code is bloated even more, maybe then a switch-case would be used.

```
public class LoginControllerStandard {
  private LoginViewSwing viewSwing;
  private LoginViewFX viewFX;
  ...
  public void loginClicked() {
    String username;
    char[] password;
    if(viewSwing != null) {
      username = viewSwing.getEnteredUsername();
      password = viewSwing.getEnteredPassword();
    } else {
      username = viewFX.getEnteredUsername();
      password = viewFX.getEnteredPassword();
    }
    ...
  }
  ...
}
```

Listing 4.11: Code snipped showing login-code after implementing a second GUI. Because no interface but concrete implementations are used each time a view access is needed the program is required to check for which of the two GUIs is currently active. Controller methods of the best pratice version do not change in any way.

4.6. Phase 3: Support of a different data tier implementation

The support of a new data tier shows even more weaknesses of the ad-hoc implementation.

4.6.1. Best-Practice Code Samples

With the best-practice program version again only very small changes to existing code were necessary. Similar to phase 2 only the DBManager static method had to be adopted to support the other database.

```
public abstract class DBManager implements RepositoryFactory {
  private static final String DRIVER_POSTGRES = "org.postgresql.
   Driver";
  private static final String DRIVER_MONGO = "mongo";
  private static DBManager instance;

  public static DBManager get(String driver, String url, String
   username, String password) {
    switch(driver) {
      case DRIVER_POSTGRES:
        if(instance == null) instance = new DBManagerPostgres(
   driver, url, username, password);
        break;
      case DRIVER_MONGO:
        if(instance == null) instance = new DBManagerMongo(url,
   username, password);
        break;
      default:
      throw new UnsupportedOperationException("Not yet implemented
   ");
    }
    return instance;
  }
}
```

Listing 4.12: DBManager implementation after support for a second database. Again only a branch of the switch-statement has to be added.

4.6.2. Ad-hoc Code Samples

After changing all data classes to support the other data tier implementation all model classes needed to be adopted as well. This is because it had to be differentiated which database implementation is currently active.

```
public class LoginModelImpl {
  ...
  public void login(String username, char[] password) throws
   Exception {
    User u = null;
    if(mainModel.dbPostgres() != null)
      u = User.getByPrimaryKey(username, mainModel.dbPostgres());
    else
      u = User.getByPrimaryKey(username, mainModel.dbMongo());
    ...
  }
  ...
}
```

Listing 4.13: Code of the login-process after implementing support of the second database in the ad-hoc version of the program. Again because the application does not rely on interfaces as in listing 4.11 it has to be differentiated each time database access is needed. The methods for database code are overloaded.

5. Results

In order to make serious statements about the quality of the code and the effects of using the design patterns some indicators were chosen to compare the two application types. Because the two programs are similar in its workflow and basic structure these comparisons are actually meaningful. Due to the use git version control bug-fixes do not influence the results.

This section only lists the findings while section 6 puts the results into context and explains them.

5.1. Lines of Code

In Table 5.1 the total lines programmed for both programs and the phases are listed. This is counted by using simple command-line on the different branches of the program. In figure 5.1 the line count development can be seen. It is notable that the best practice implementation consists of more total lines of code in all phases.

Program version	Phase 1	Phase 2	Phase 3
Best practice	9264	12275	13306
Best practice (adjusted) [1]	8804	11764	12750
Ad-hoc	7572	11172	12168

Table 5.1.: Lines of code for both program versions and all phases

[1] Due to the use of interfaces in java a lot of annotations were necessary which in fact do not have any significance to the topic. Thus the adjusted line count excludes lines that only hold the `@Override`-Annotation.

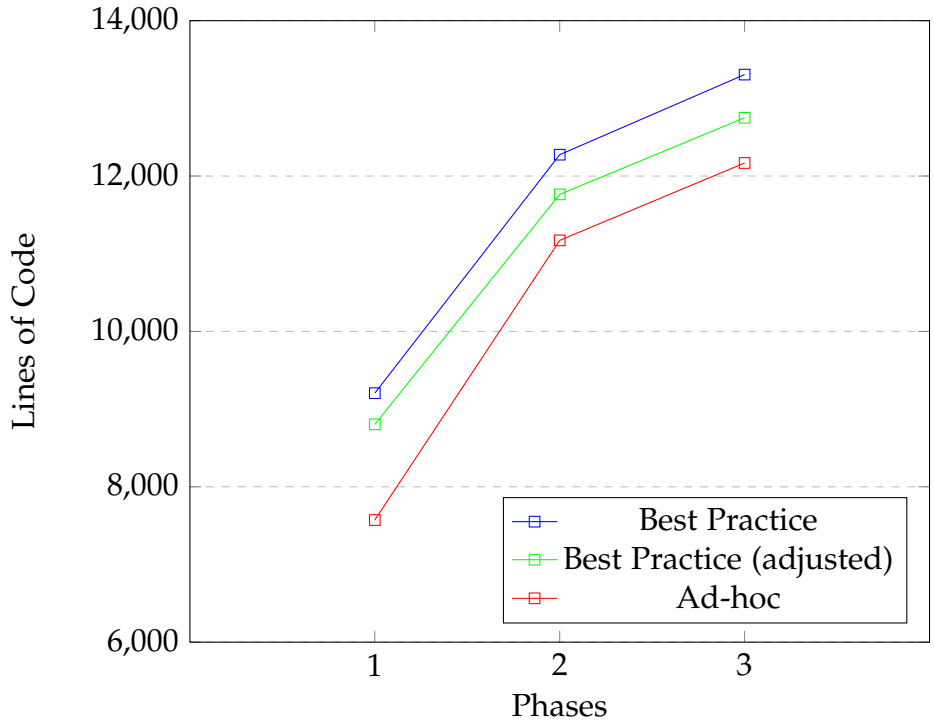

Figure 5.1.: Lines of Code for both program versions and all phases. The best practice implementation of the program both in the adjusted and original version consists of more code lines in all phases.

Program version	Phase 1	Phase 2	Phase 3
Best practice	96	125	131
Ad-hoc	59	88	89

Table 5.2.: Total number of files for both program versions and all phases

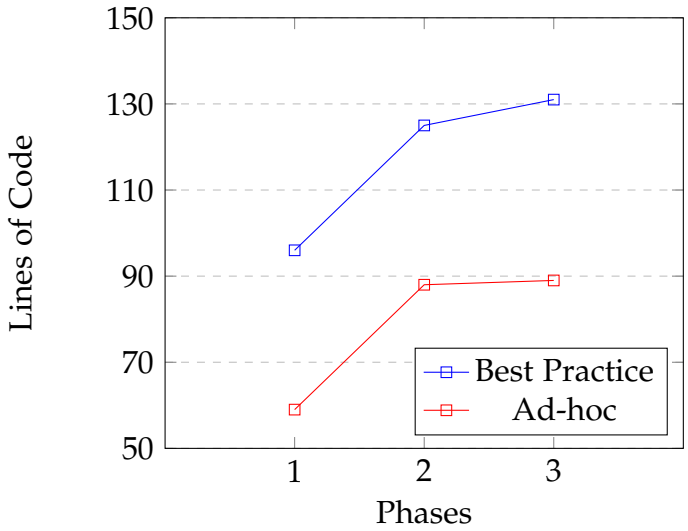

Figure 5.2.: Total number of files for both program versions and all phases

5.2. Touched and Edited Files

The development of the file count is listed in this section. Table 5.2 and figure 5.2 shows how many files are needed for the programs while table 5.3 list the number of files that were modified or created during the implementation of the requirements.

Program version	Phase 1 to Phase 2	Phase 2 to Phase 3
Best practice	30	7
Ad-hoc	39	13

Table 5.3.: Number of files modified or added for both program versions and all phases

Program version	Phase 1	Phase 2	Phase 3
Best practice	96	98	102
Best practice (adjusted)	92	94	97
Ad-hoc	128	127	137

Table 5.4.: Average lines per file for both program versions and all phases

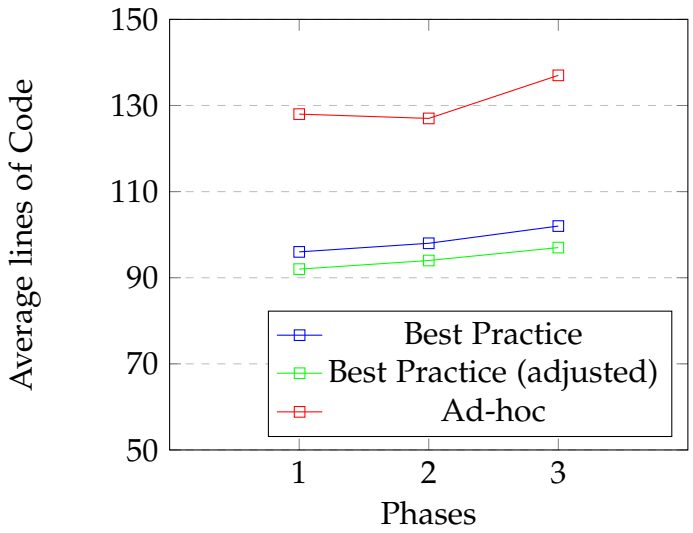

Figure 5.3.: Total number of files for both program versions and all phases

5.3. Average line count per file

Another interesting aspect of the data is the comparison of the average line count per file, as it can be seen in table 5.4 and chart 5.3. The results are rounded mathematically.

6. Discussion

In this chapter all data gathered in the last section is revisited and put into context.

6.1. Meaning and significance of the file count

As clearly visible in section 5.2 the ad-hoc version of the program does not need as many files as the best practice implementation. This is mainly due to the missing interface files, as well as the files for repositories. Especially in phase 3, where a new database should be supported, this factor can be easily observed. In the ad-hoc version all code concerning the database interaction was put into existing classes, resulting in only one additional file while the best practice implementation needs to create and derive one file per repository.

6.2. Meaning and significance of the line count

The same statement as above applies to the count of total lines as seen in section 5.1. Each interface class consists of various method declarations that are counted as well. As stated in the footnote of table 5.1 for better comparison in addition to the total line count for the best practice implementation another count was made excluding the `@Override`-Annotation lines as they bias the result.

Here a interesting observation can be made for the line count: While the total count of the best practice version is higher for both phases 1 and two,

the increase of lines is between the phases is significantly lower. While the best practice version added 2960 lines of code, the ad-hoc version added 3600 lines of code. This difference of about 700 lines does not seem much, however it has to be noted that this lines were mainly necessary because of case distinctions as can be seen in section 4.5.2. This reduces readability on the one hand and results in a greater risk of programming errors.

The fact that the increase in lines between phase 2 and 3 is nearly the same is easily explained as the added files differ significantly. Due to the overhead of declaring a class including its constructor etcetera leads to a high number of additions which are not necessary in the ad-hoc version.

Consequentially to the lower number of files and the comparable number of lines in the two programs, the ad-hoc implementation on average contains noticeably more code per file. As the two implementations offer exact the same functionality this means that responsibilities have shifted and each class is in charge of more actions. However this fact is a clear violation of the Single-Responsibility-Principle (see section 3.1.4) which may introduce more programming errors in the future.

6.3. Maintainability

The biggest effect and difference when using design patterns and interfaces can be observed exactly in conforming of the Single-Responsibility-Principle and hence in the readability of the code. While each data class of the ad-hoc version does not only have the responsibility to encapsulate the data, it is responsible for saving the data to two different kinds of persistent storage. This results in a quite long file that mixes technologies and needs to be revisited any time a new persistent storage is added, existing storage technologies are changed or the data classes change.

In the case of view frameworks due to missing use of interfaces asides from implementing the view using a new framework each and every class that depends on any kind of interface needs to be adopted. This is visible in table 5.3, where is stated that the implementation of new technologies influences lower numbers of dependent files which is a very desired behaviour.

7. Conclusion

In the last chapters some interesting findings were gathered. Two program versions were compared, one following best practice guidelines in the sense of object oriented programming, the other one featured ad-hoc programming without paying attention to topics of software maintenance.

Throughout the development process five major issues could be discovered:

- **Number of files**: The best practice version of the application needed a higher number of files than the ad-hoc implementation throughout all phases.
- **Lines of code**: The best practice version needed more lines of code in all phases.
- **Lines per file**: The best practice versions files contained significantly less lines of code per file.
- **Increase in code lines**: Especially in traversal from phase 1 to phase 2 the best practice implementation needed significantly less code additions.
- **Touched files**: The number of touched or edited files during phase traversal was lower in the best practice program version.

It has to be said that the results gathered in section 5 may vary depending on the coding style, programming language, concrete use case and size of the project, furthermore the term *code quality* does not have a unique definition. There are many different standpoints on what attributes a well written code should have, often depending on company coding policies. One factor that is not mentioned by intention in this thesis is the readability of the code as this is a very subjective matter and cannot be proven.

In terms of the definition used in this thesis, which mainly focuses on properties defined by guidelines of the object oriented programming the results support the hypothesis saying that the use of design patterns improves code quality in later stages of a project cycle. This is backed up by the lower number of files needing modification in the process of extending functionality and the lower number of lines added between phases 1 and 2.

At the same time in the first stage the initial programming effort for a clean implementation using well-known patterns is significantly higher, and even later on in the best practice implementation utilizes both a larger number of files and more lines of code, which on the other hand may increase the efforts for software maintenance.

In conclusion it can be said that the use of design patterns can help to increase code quality if the requirements of a software project are likely to change, which is the common case especially for bigger applications.

As this work only focuses on a small number of design patterns and other literature does not come to clear results as well, further research is certainly necessary. If future work is focusing on existing software projects, challenges will likely include the difficulties in comparing different approaches applied in the compared applications. However if new research rather studies effects of design patterns by observing controlled experiments, as this thesis does, the main challenge will likely include the question if small scale results also apply for enterprise scale applications.

Appendix A.

Source Code

All source code including this thesis is online available on Github [1] under the GPLv3 licence. [2].

The repository consists of a main branch, a thesis branch and 6 branches corresponding to the phases for each program implementation. The main branch contains the latest version of both applications as well as the latest version of the thesis branch. In the thesis branch naturally only tex sources for creating the thesis can be found. The branches corresponding to the phases are called `bpV{1|2|3}` (best-practice version 1,2 or 3) and `monoV{1|2|3}` (monolithic version 1,2 or 3 [3])

In order to run the application, some pre-requirements have to be met. Firstly a PostgreSQL and a MongoDB database must be installed. Secondly the database initialization code that can be found in the directory `<repository-root>/src/db` needs to be imported into the corresponding DBMS.

The application itself needs JDK8 and JRE8 to run, all necessary libraries can be found in the directory `<repository-root>/src/libraries`. These libraries can be included if it is wanted to compile the source code.

After compiling the source code to a .jar file the program can be started by providing the following command:

[1] https://github.com/Vallant/design-patterns-study.git

[2] https://www.gnu.org/licenses/gpl-3.0.de.html

[3] The term monolithic was used in a early stage of development and was later dropped for *ad-hoc*

```
java -jar <application-name> <driver> <url:port> <username> <password>
<controller> <frontend>
```

from which `driver` can be either "`org.postgresql.Driver` or "`mongo`", `url:port` points to the location where the database is running (presumably this is localhost), `user` and `password` correspond with the database user of the chosen DBMS, `controller` must be "`standard`" at the moment and `frontend` can either be the string "`swing`" or "`javafx`".

Bibliography

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

[2] Giuseppe Castagna. *Object-oriented programming. In: Object-Oriented Programming A Unified Foundation. Progress in Theoretical Computer Science*. Birkhäuser Boston, 1997. DOI: 10.1007/978-1-4612-4138-6 (cit. on p. 5).

[3] Bo Nørregaard Jørgensen Eddy Truyen Wouter Joosen et al. "A Generalization and Solution to the Common Ancestor Dilemma Problem in Delegation-Based Object Systems." In: (2004) (cit. on p. 6).

[4] Richard Helm Erich Gamma et al. *Desing Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994 (cit. on pp. 5–7, 11).

[5] Stephen T. Pope Glenn E. Krasner. "A Description of the Model-View-Controller User Interface Paradigm in the Smalltalk-80 System." In: (1988) (cit. on p. 10).

[6] Gilbert Held. *Server Management (Best Practices)*. Auerbach Publications, 2000. ISBN: 978-0849398230 (cit. on p. 14).

[7] Michael Schrefl Johann Eder Gerti Kappel. “Coupling and Cohesion in Object-Oriented Systems.” In: (1994) (cit. on p. 7).

[8] Peter Wegner Luca Cardelli. “On Understanding Types, Data Abstraction, and Polymorphism.” In: (Dec. 1985) (cit. on p. 7).

[9] Robert C. Martin. *Clean Code*. Pearson Education Inc., Dec. 2009. ISBN: 978-0-13-235088-4 (cit. on pp. 5, 10).

[10] Bertrand Meyer. “Object Oriented Software Construction.” In: (1988) (cit. on p. 8).

[11] D.L. Parnas. “On the Criteria To Be Used in Decomposing Systems into Module.” In: (1972) (cit. on p. 8).

[12] Brian Foote Ralph E. Johnson. “Designing Reusable Classes.” In: (1991) (cit. on p. 6).

[13] Octavian Stanescu Valentin Corneliu Pau Marius Iulian Mihailescu. “Model View Presenter Design Pattern.” In: (2010) (cit. on p. 12).

List of Figures

3.1. Classical Model-View-Controller concept as used in Smalltalk. Model and View know each other. 11

3.2. Diagram of Model-View-Presenter. While the view again only holds the means for displaying the data and interacting, in opposite to the MVC-model it does not know about the model but only corresponds with the presenter. The presenter is responsible for gathering the necessary data from the model and forwards it to the view. The model does not know about the presenter. 12

3.3. Basic diagrams of the Passive View (left) and Supervising Controller patterns (right). In the former as little logic as possible is placed in the view making it easy replaceable by a mock object for testing. Synchronization logic needs to be placed in the Presenter. In the latter view and model are connected trough data bindings empowering these two to perform synchronization tasks on their own. This simplifies the role of the presenter. 13
3.4. Client-Server setup. The client sends a request to a server by messages defined in the protocol. The processing of the request is transparent to the client, it does not know about actions taken by the server and awaits a response in a certain format. 15
3.5. Peer-to-peer setup. Each peer has the same rights, no central server exists. 15
3.6. Multi-Tier set-up. The tiers are physically separated. 16

5.1. Lines of Code for both program versions and all phases. The best practice implementation of the program both in the adjusted and original version consists of more code lines in all phases. 40
5.2. Total number of files for both program versions and all phases 41
5.3. Total number of files for both program versions and all phases 42

List of Tables

5.1. Lines of code for both program versions and all phases 39
5.2. Total number of files for both program versions and all phases 41
5.3. Number of files modified or added for both program versions and all phases 41
5.4. Average lines per file for both program versions and all phases 42

Listings

4.1. DBManager singleton class. Used for instantiating the implementation of the chosen interface. 29
4.2. Code snippet on how a DBManager implementation instantiates its repositories. 30
4.3. Code snippet showing how repositories are used in business code. The main model holds the single instance of the DBManager. Each sub-model accesses it by using a getter-method. 30
4.4. Code snippet showing initialization code for the controller. The main model retrieves the main controller by calling a getter which is similarily implemented as the DBManager class shown in listing 4.1. After that he calls a method that pairs the sub-model with the sub-controller, a code snippet showing this process can be seen in listing 4.5. 31
4.5. Code snipped showing pairing code between a sub-controller and a sub-view. The pairing between the model and controller layer works in a very similar way. 32
4.6. Code snipped showing database access in the ad-hoc program version. All database code is placed into the corresponding data class, retrieval methods are static and take the corresponding DBManager as parameter. The reference to the DBManager is provided by a getter method of the main model, as in listing 4.3. 32
4.7. Code snippet showing how the DBManager is instantiated. Again no interfaces but rather concrete implementations are used. The singleton class seen in listing 4.1 does not exist in the ad-hoc version. 33

4.8. Code snippet showing a database access methods in a data class. The retrieval method takes arguments necessary for the query and the database-manager object as parameters and is static, while the update method is not static and therefore performs the operation on the current object instance. 33
4.9. Code snippet showing pairing code of the ad-hoc version of the program. It is noticeable that the pairing method takes a concrete implementation rather than an interface as a parameter. This differs from the best practice version seen in 4.5 which takes only the interface as parameter. 34
4.10. View Manager after implementation of a second GUI. Only a case branch has to be added. 35
4.11. Code snipped showing login-code after implementing a second GUI. Because no interface but concrete implementations are used each time a view access is needed the program is required to check for which of the two GUIs is currently active. Controller methods of the best pratice version do not change in any way. 36
4.12. DBManager implementation after support for a second database. Again only a branch of the switch-statement has to be added. 37
4.13. Code of the login-process after implementing support of the second database in the ad-hoc version of the program. Again because the application does not rely on interfaces as in listing 4.11 it has to be differentiated each time database access is needed. The methods for database code are overloaded. . . . 38