

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
<b>2</b>	<b>Background</b>	<b>3</b>
2.1	Fundamentals of Object-Oriented Programming . . . . .	3
2.1.1	Encapsulation . . . . .	3
2.1.2	Inheritance . . . . .	4
2.1.3	Polymorphism . . . . .	5
2.1.4	Single Responsibility Principle . . . . .	5
2.1.5	Open Closed Principle . . . . .	6
2.2	Design Patterns and Architectural Patterns . . . . .	6
2.2.1	Repository . . . . .	7
2.2.2	Model-View-Controller . . . . .	7
2.2.3	Model-View-Presenter . . . . .	8
2.2.4	Client-Server . . . . .	11
2.2.5	Peer-to-Peer . . . . .	11
2.2.6	Multi-Tier and Multi-Layer Architecture . . . . .	12
2.3	Technologies . . . . .	13
2.3.1	Java . . . . .	13
2.3.2	Swing . . . . .	14
2.3.3	JavaFX . . . . .	14
2.3.4	Relational Databases and PostgreSQL . . . . .	15
2.3.5	MongoDB . . . . .	15
2.3.6	Caching . . . . .	16
<b>3</b>	<b>Methods</b>	<b>21</b>
3.1	Domain Description . . . . .	21
3.2	Experiment set-up . . . . .	23
3.2.1	Best practice implementation . . . . .	24
3.2.2	Ad-hoc implementation . . . . .	25

## Contents

3.2.3	Experiment phases . . . . .	25
3.2.4	Phase 2: Support of a different graphical user interface library . . . . .	26
3.2.5	Phase 3: Support of a different data tier implementation	26
3.2.6	Phase 4: Implementation of a cached controller . . . . .	26
3.3	Phase 1: Basic Requirements . . . . .	28
3.3.1	Best-Practice Code Samples . . . . .	28
3.3.2	Ad-hoc Code Samples . . . . .	31
3.4	Phase 2: More Databases / UIs . . . . .	32
3.4.1	Monolithic Code Samples . . . . .	32
3.4.2	Best-Practice Code Samples . . . . .	32
<b>4</b>	<b>Results</b>	<b>33</b>
4.1	Lines of Code . . . . .	33
4.2	Programming Effort . . . . .	33
4.3	Efficiency and Performance . . . . .	33
4.4	Touched and Edited Files . . . . .	33
4.5	Readability . . . . .	33
<b>5</b>	<b>Discussion</b>	<b>35</b>
5.1	Programming Overhead . . . . .	35
5.2	Use of OOP in General . . . . .	35
<b>6</b>	<b>Conclusion</b>	<b>37</b>
<b>7</b>	<b>Future Work</b>	<b>39</b>
7.1	Physical Separation of Tiers . . . . .	39
7.2	MVC-Frameworks . . . . .	39
7.2.1	Spring Framework . . . . .	39

# 1 Introduction

The topic 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 the topic of software design and architectural styles, 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* <sup>1</sup>, Robert C. Martin with *Agile Software Development. Principles, Patterns, and Practices* <sup>2</sup> or *Design Patterns. Elements of Reusable Object-Oriented Software* by John Vlissides, Richard Helm, Ralph Johnson and Erich Gamma <sup>3</sup>.

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, the 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

---

<sup>1</sup>IBAN: 978-0321127426

<sup>2</sup>IBAN: 978-0135974445

<sup>3</sup>IBAN: 978-0201633610

## 1 Introduction

the 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.

Another study published covering a similar subject came to different results, concluding that the use of established design patterns in most of the cases led to a significant improvement of code quality, contradicting the previous study. <http://ieeexplore.ieee.org/abstract/document/4493325/?reload=true> and <http://www.sciencedirect.com/science/article/pii/S0950584911002151>;

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 most of the research is done via *Controlled Experiments*, as is this thesis. Therefore it has to be said that most of 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 nevertheless 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.

## 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* **Castagna97**

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 <sup>1</sup> each class should prevent direct access to its 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

---

<sup>1</sup>Refactoring describes the process of re-writing classes to improve readability, performance or other criteria while preserving its signature, inputs and outputs.

## 2 Background

enforce invariants <sup>2</sup>. This means that it hides all information not relevant to others as they are only implementational details and thus not relevant. As other classes now rely on an interface rather than concrete implementations the code is called *loosely coupled*. Modifying the internals now does not break interdependencies which encourages programmers to perform refactoring resulting 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. TODOTODO

### 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 through 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 **johnson91**

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.

---

<sup>2</sup>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.

### 2.1.3 Polymorphism

Polymorphism describes the ability to tie the same interface to different belonging types. <sup>3</sup> 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 *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. <sup>3</sup>.

### 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. The modifications may introduce errors to the other

---

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

## 2 Background

functionality, which could be avoided if the program could conform to the principle. This would require placing each responsibility in an independent 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 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 as the original class is preserved as-is.

## 2.2 Design Patterns and Architectural Patterns

In opposite to the previous chapter which describes rather abstract guidelines in 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 2.2.6) 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



## 2.2 Design Patterns and Architectural Patterns

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.

### 2.2.1 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. 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 2.2.6). 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 they are not relevant to this thesis.

To create one repository per business object (data class) is the most basic way of implementing this pattern. Besides from the basic *CRUD*-Operations <sup>4</sup>, each repository should provide specialized object retrieval methods that are only valid for the business model it was programmed for.

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 <sup>5</sup>.

### 2.2.2 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

---

<sup>4</sup>Create, read, update, delete

<sup>5</sup>You aren't gonna need it

## 2 Background

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.

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. Views subscribe at the model for notifications on changes of data in order to be able to update and refresh themselves. The controller was responsible of managing user interaction, which includes switching between views and forwarding requests to the model. This situation can be seen in Figure ?? . This direct interaction between model and view is often not desirable, especially in the context of web applications, which leaves it deprecated today.

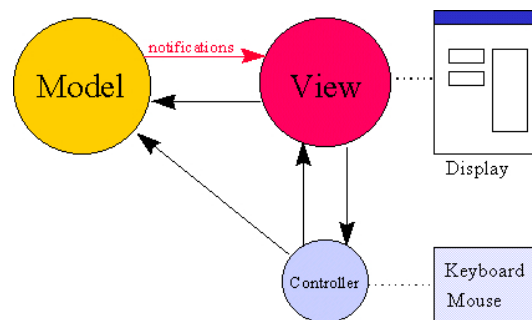


Figure 2.1: Classical Model-View-Controller concept as used in Smalltalk. Model and View know each other.

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

### 2.2.3 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. However the responsibilities differ considerably, as can be seen in Figure ?? . The view

## 2.2 Design Patterns and Architectural Patterns

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. 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.

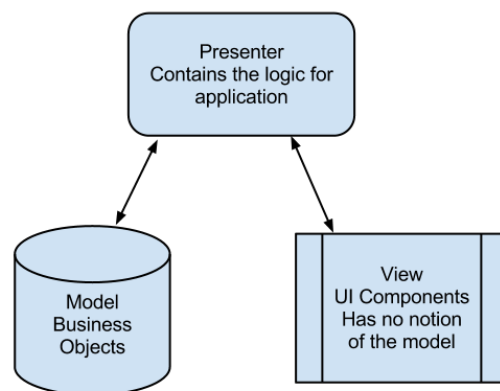


Figure 2.2: Diagram of Model-View-Presenter. While the view again only holds the means for displaying the data and interacting, in opposite to the MVC-model it does not know about the model but only corresponds with the presenter. The presenter is responsible for gathering the necessary data from the model and forwards it to the view. The model does not know about the presenter.

Source: <https://de.wikipedia.org/wiki/Model-View-Presenter>, accessed on 5.9.2017, by Daniel.Cardenas - Own Content, CC BY 3.0, <https://commons.wikimedia.org/w/index.php?curid=19794348>

After some years Martin Fowler drafted two design patterns that are closely connected to MVP. In his blog entry <sup>6</sup> he explained that he felt the need to split the MVP-pattern into two resulting in the following concepts, briefly described in Figure ??.

---

<sup>6</sup><https://martinfowler.com/eaDev/ModelViewPresenter.html>

## 2 Background

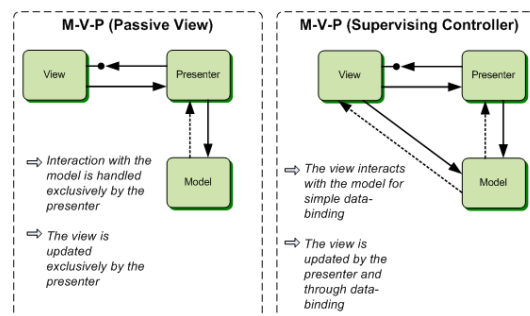


Figure 2.3: Basic diagrams of the Passive View (left) and Supervising Controller patterns (right). In the former as little logic as possible is placed in the view making it easy replaceable by a mock object for testing. Synchronization logic needs to be placed in the Presenter. In the latter view and model are connected through 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 on 5.9.2017

### 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 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.

7

---

<sup>7</sup><http://www.wildcrest.com/Potel/Portfolio/mvp.pdf>

### 2.2.4 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 resource from the server. It is designated to be used within the internet and is still one of the cornerstones of current web infrastructure. 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 place 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 effort to 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 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.

### 2.2.5 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.

## 2 Background

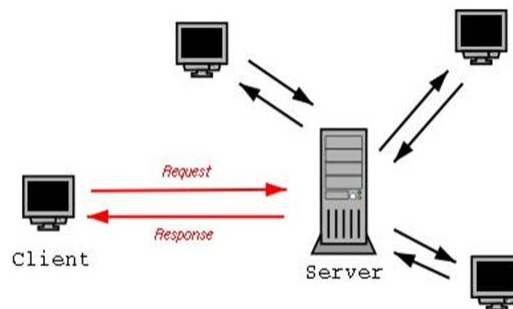


Figure 2.4: Client-Server setup. The client sends a request to a server by messages defined in the protocol. The processing of the request is transparent to the client, it does not know about the actions taken by the server and awaits a response in a certain format.

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

### 2.2.6 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 *physical* places <sup>8</sup>. 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 ?? . 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.

---

<sup>8</sup>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.

## 2.3 Technologies

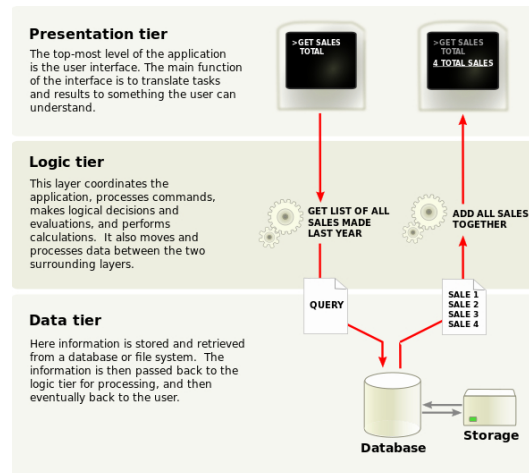


Figure 2.5: Multi-Tier set-up. The tiers are physically separated.

Source: [https://en.wikipedia.org/wiki/Multitier\\_architecture](https://en.wikipedia.org/wiki/Multitier_architecture), accessed on 6.9.2017

## 2.3 Technologies

### 2.3.1 Java

This project is written in Java, an object oriented programming language released by Sun Microsystems in 1995<sup>9</sup>. 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<sup>10</sup>.

<sup>9</sup>[https://java.com/en/download/faq/whatis\\_java.xml](https://java.com/en/download/faq/whatis_java.xml), accessed 7.9.2017

<sup>10</sup><http://www.oracle.com/technetwork/java/embedded/javacard/overview/index.html>, accessed 7.9.2017

## 2 Background

The JRE uses an interpreter and a just-in-time (JIT) compiler to generate native machine code that can be executed on the platform's 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% <sup>11</sup>.

### 2.3.2 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 to applications natively written for the platform 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 adapt the application to be as similar to a native program as possible.

### 2.3.3 JavaFX

JavaFX is a framework <sup>12</sup> 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.

---

<sup>11</sup><https://www.tiobe.com/tiobe-index/>, accessed 7.9.2017

<sup>12</sup>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.



### 2.3.4 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.<sup>13 14</sup>

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*, 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.

### 2.3.5 MongoDB

As a prominent example for NoSQL<sup>15</sup>-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<sup>16</sup>-like manner.

---

<sup>13</sup> <https://www.postgresql.org/about/>

<sup>14</sup> <https://db-engines.com/de/ranking>

<sup>15</sup>Not only SQL

<sup>16</sup>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.

## 2 Background

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.

### 2.3.6 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.

In most of the cases placing the cache in the data access layer (see section 2.2.6) 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-through-caching all data is first persisted and then immediately updated in the cache, this operation blocks until all necessary writes are completed. This ensures that at all times a up-to-date copy of the data is available <sup>17</sup>.

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

---

<sup>17</sup>, accessed 10.9.2017

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<sup>18</sup>.

### 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. 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.

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.

---

<sup>18</sup><http://www.computerweekly.com/feature/Write-through-write-around-write-back-Cache-explained>, accessed 10.9.2017

## 2 Background

- **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 an unfavourable pattern. Random replacement is not sensitive to these problems while providing solid performance.

### Cache invalidation

In the context of cache invalidation a quote from Phil Karlton is especially famous<sup>19</sup>:

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<sup>20</sup>

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 an 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.

---

<sup>19</sup>It is not clear if the quote is genuinely made by Karlton, however it is generally attributed to him.

<sup>20</sup><http://tutorials.jenkov.com/software-architecture/caching-techniques.html>, accessed 10.9.2017

## 2.3 Technologies

Even with active cache invalidation it cannot be guaranteed that the update notification reaches the second client in-time.



## 3 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.

### 3.1 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

### 3 Methods

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

- **Activity:** An 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 an 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 projects 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



## 3.2 Experiment set-up

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.

## 3.2 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.

## 3 Methods

### 3.2.1 Best practice implementation

In the first application well-known and widely spread patterns should be applied. The data access to any persistent storage is abstracted by implementing the repository-pattern (see section 2.2.1). The program is structured according to the Model-View-Controller pattern (section 2.2.2), the layers are connected only by interfaces. 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 of 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 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.

## 3.2 Experiment set-up

Figure 3.1: Model of the program setup of the best practice version of the application. All layers are connected only by interfaces. The data access is realized by repositories.

Figure 3.2: Model of the program setup of the ad-hoc version of the application. All layers are connected by concrete implementations. The data access is realized by static methods provided by the data classes.

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.

In figure ?? a model of the program setup can be seen.

### 3.2.2 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 besides 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 ?. Figure ?? shows the setup for the ad-hoc version of the application.

### 3.2.3 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.

## 3 Methods

### Phase 1: Basic requirements

For starters the application should provide a graphical user interface by using the Swing-library (section 2.3.2) and as data tier a PostgreSQL database (section 2.3.4) should be used.

#### 3.2.4 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 2.3.3). 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.

#### 3.2.5 Phase 3: Support of a different data tier implementation

During the process the stakeholders request that the application should add support for MongoDB (section 2.3.5). They want to allow the user to choose between the two databases as both implementations have advantages and disadvantages.

#### 3.2.6 Phase 4: Implementation of a cached controller

Several months after the deployment a customer reports performance issues and heavy load on their database server. Therefore it is decided to implement a caching algorithm for the controller to reduce expensive database calls. Again by starting the application with different arguments either the standard controller or the caching implementation should be used.

### 3.2 Experiment set-up

The persistent storage has a quite simple structure, as can be seen in figure 3.3. It is notable that the roles are described as text and therefore are human-readable.

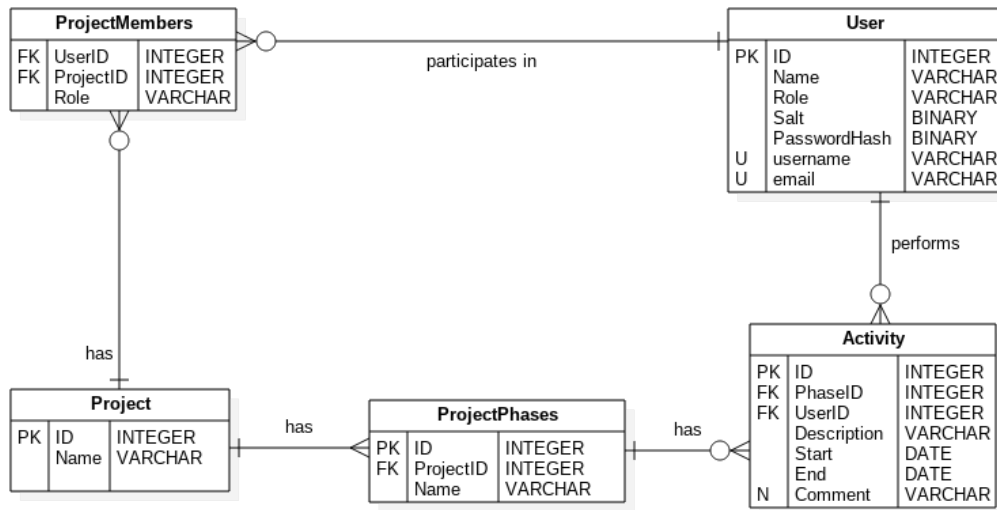


Figure 3.3: Database diagram

### 3.3 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.

#### 3.3.1 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.

```
1 public abstract class DBManager implements RepositoryFactory
2 {
3     private static final String DRIVER_POSTGRES = "org.postgresql.
4         Driver";
5     private static DBManager instance;
6
7     public static DBManager get(String driver, String url, String
8         username, String password)
9     {
10         switch(driver)
11         {
12             case DRIVER_POSTGRES:
13                 if(instance == null) instance = new DBManagerPostgres(
14                     driver, url, username, password);
15                 break;
16             default:
17                 throw new UnsupportedOperationException("Not yet
18                     implemented");
19         }
20         return instance;
21     }
22 }
```

[content/code/dbmanager.java](#)

### 3.3 Phase 1: Basic Requirements

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);
    }
    ...
}
```

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.

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

### 3 Methods

#### 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 called *dependency injection*.

```
private MainModelImpl(String driver, String url, String username,
    String password, String controllerImpl, String
2 frontend) throws Exception
{
4     db = DBManager.get(driver, url, username, password);
6     ControllerManager.initInstance(controllerImpl, frontend);

8     controller = ControllerManager.getInstance();
    controller.setModel(this);

10     login = new LoginModelImpl();
12     pairLogin();
    ...
14 }

16 private void pairLogin()
{
18     login.setMainModel(this);
    controller.pairLogin(login);
20 }
```

The method call seen in line 6 of the previous 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.



### 3.3.2 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 3.2.2 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.

```
1 public void login(String username, char[] password) throws
   Exception
2 {
3     User u = User.getByPrimaryKey(username, mainModel.db());
4     ...
5 }
```

```
1 public class MainModelImpl
2 {
3     ...
4     private final DBManagerPostgres db;
5
6     private MainModelImpl(String driver, String url, String username
7         , String password, String controllerImpl, String
8         frontend) throws Exception
9     {
10         db = new DBManagerPostgres(driver, url, username, password);
11         ...
12     }
13     ...
14 }
```

### 3 Methods

## **3.4 Phase 2: More Databases / UIs**

### **3.4.1 Monolithic Code Samples**

### **3.4.2 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 Separation of Tiers

### 7.2 MVC-Frameworks

#### 7.2.1 Spring Framework

Composing06



# List of Figures

2.1	Classical Model-View-Controller concept as used in Smalltalk. Model and View know each other. . . . .	8
2.2	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. . . . .	9
2.3	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. . . . .	10
2.4	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 the actions taken by the server and awaits a response in a certain format. . . . .	12
2.5	Multi-Tier set-up. The tiers are physically separated. . . . .	13
3.1	Model of the program setup of the best pratice version of the application. All layers are connected only by interfaces. The data access is realized by repositories. . . . .	25

## List of Figures

3.2	Model of the program setup of the ad-hoc version of the application. All layers are connected by concrete implementations. The data access is realized by static methods provided by the data classes. . . . .	25
3.3	Database diagram . . . . .	27

## List of Tables