Design Patterns for Applications Inspired by Cloud Infrastructure

Arselan Sultani

Thesis submitted for the degree of
Master in Informatics: Programming and Network
60 credits

Institute for Informatics
Faculty of mathematics and natural sciences

UNIVERSITY OF OSLO

Spring 2018

Design Patterns for Applications Inspired by Cloud Infrastructure

Arselan Sultani

Design Patterns for Applications Inspired by Cloud Infrastructure

Arselan Sultani

2nd May 2018

Abstract

Design patterns are solutions to common problems that developers face. Cloud is an already popular platform and its popularity keeps increasing. This paper focuses on how design patterns fits into and area which is inspired by this popular platform and how it can be used on applications inspired by it.

It is an empirical research using case study for experimenting different design patterns for each case. Two different cases has been developed and three different implementations for each case, where each case is a different design pattern. Each case gets measured and compared to find the best implementation among each implementation.

Measurements that are going to be used, are meant for static analysis of a system. An analysis of systems complexity and maintainability will be done with these metrics. Halstead complexity measures, Cyclomatic complexity, Cognitive complexity, maintainability index and class coupling will be used as metrics.

There are some design patterns that are implemented, but since they are visible in each implementations anyway, they will be given a subjective opinion on how they fit into the cloud platform. These design patterns are Observer and Mediator pattern.

Contents

List of Figures

List of Tables

Acknowledgements

I would like to thank all my supervisor Eric Bartley Jul for being very helpful during my thesis. For his patience, advice and structural advices.

I would also like to thank my family and friends. Who have been nothing but helpful, supporting and motivated me to keep working hard.

Part I

Introduction

Chapter 1

Introduction

Developing applications is something almost everyone can do nowadays, either as a hobby or as a professional. Only basic programming skills are required for being able to develop a small application, like turning on and off the flashlight of your mobile phone. The more one develops, the more they learn. As a developer, one spends plenty of time debugging their code. And over time, debugging would take less and less time because they would learn from past mistakes. Once they face a similar problem faced earlier a similar approach can be taken to solve problems of the same calibre. These are the solutions that are called Design patterns.

Applications using cloud infrastructure keeps increasing in popularity. It is because cloud infrastructure can be used in almost any kind of applications. For example, games use it for storing the player's score and compare it to their friends from social media. Storage services like Dropbox, Google Disk, and Microsoft OneDrive use it for storing data in their server, and there are services that offer their client run software that requires great hardware on cloud right in user's browser.

In this paper, with having cloud as an inspiration, an empiric study is conducted for investigating and experimenting with design patterns on application using client-server architecture. Also whether design patterns actually help when it comes to systems maintainability and complexity. It will be compared against other implementations with well defined metrics. Some design patterns will be in all implementations and those patterns will be given a subjective opinion, rather the objective with the implementations that can be calculated to metrics.

1.1 Motivation

Considering how popular applications using cloud infrastructure have become, I believe it is interesting to see how design patterns fit for applications using client-server architecture. To see some chosen design patterns, which would fit in, which would not fit and require major changes for the pattern to help the overall structure of the system.

1.2 Problem statement

The bigger the project gets, the more important program's structure is. In this thesis, a look at experimenting with design patterns to make a better solution for application's structure in the context of the application inspired by cloud.

1.3 Goal

The goal of this thesis is to obtain a better understanding of which design patterns could be used with applications using client-server architecture and which design patterns that should not be used. Either by it being limited or it not being the better solution.

1.4 Approach

My approach is by using empirical case study to determine the preferable implementation among other by using various maintainability and complexity metrics for comparison. By using case study, two different cases with different implementations are developed, where two of the implementations are with different design patterns and the last is an implementation without any specific design pattern.

The purpose of each case is different, one is for a chat service, while the other one is a storage service. But something that both must have in common, is that both must be implemented with client-service architecture. Which means that the service must be able to run from anywhere on the planet and the client must also be able to connect to the service from anywhere.

1.5 Evaluation what/how

Static analysis tools will be used for comparing the different implementations. SonarQube and MetricsReloaded are the tools that will be used. SonarQube is for continuous inspection of code and uses static analysis to gather information about the code, like weakness and vulnerabilities. MetricsReloaded is a tool that is integrated into the IntelliJ IDEA as a plugin, and can calculate various metrics.

The metrics that are being used, are metrics that are popular and relevant for the purpose of their goal. They have been used in the market as a way to find bugs and weakness in their code. Complexity and maintainability metrics are some of the metrics that is being used. For complexity, McCabe's cyclomatic metric is used, along with Halstead metrics, such as total number of tokens, volume, difficulty and effort. Maintainability index and Microsoft's maintainability used in Microsoft Visual Studio IDEA, is being used as maintainability indexes. There are not tools that are available for measuring the maintainability index, but since all the variables that are used in the formula are calculated, it can be plotted in manually for the calculation of the maintainability index. Microsoft Excel and WolframAlpha is being used to double check the results from the each other.

1.6 Work done

Two services that uses client-server model has been developed with chosen design patterns. Each service is considered a case. Each case has three different implementations, one of which is without any specific design pattern and the two others are implemented with using Strategy along with Singleton and Template on the last implementation.

Case one is a chat service where multiple clients can subscribe to an chat room and get notification everytime any of the clients in the same room has sent a message to the room. Case two is a storage service where there is one client for each area, and only that client can change data in that area.

Both cases have been implemented with the three different implementations. All six implementations are runnable and can be run from any server. However the client need to know the IP address and port number of the service for it to be able to connect to the service.

MetricsReloaded and SonarQube is then run the code for comparison and finding the better implementation. Both tools is used for finding different metrics and sometimes, they offer the same metric, and then it is used for validation of the data, for making sure we are receiving the same index for both metrics.

1.7 Results

There are two cases and three implementation for each case. The metrics that are chosen is used for finding out about which implementation has the least complexity and best maintainable.

The result for each case showed that the implementation with Template pattern is the preferable implementation among the three different implementations for each case when it comes to complexity, however with maintainability, the implementation without any design pattern is the preferable implementation.

1.8 Conclusion

During doing this research, I found out that Template pattern does help reduce the complexity of the system, but it does not always help with the maintainability of the system. An implementation without any design patterns necessary with interface that are implemented by different client implementation and service implementation, is better for maintainability. If developing a client-server application, then Strategy pattern may not be the better solution compared to Template pattern and basic implementation without any specific design pattern, as it received the worst score on both cases for both complexity and maintainability.

Part II

Background

The evolution of computer science has people excited. With technologies that took a whole room some decades ago, is now small enough to fit in peoples pockets. The evolution's influence on day-to-day basis, can vary from small to huge. As small as tweaking a product to make it better and maybe a bit different so the users can see the difference or creating a product that will change the world in any form or shape. New technologies gets created, and along with that, comes new platforms, platforms for the developers to develop a product for. A platform is a software-based product or service that first or third party developers can develop applications or complementary services or products for. It serves as a foundation for third-party developers for their services and products. Platforms like cloud, web, mobile, etc.

Every new platform comes with pros and cons. There are always something positive about it, that is why they get created in the first place, but most likely will not be perfect on first release. Which is why they get tested and experimented on to see which functions works and what does not, which functionalities must be improved and which should be removed. They get updates to make them safer, faster and more stable, along with new tools to make it faster and easier for developers to create new products for the platform. It also means new ideas, new ways to develop something that might not exist already. A new world for creators and developers to enter and expand their knowledge and skill. It can be either a new piece of hardware that developers can create new software for, or it can be something new on top of an already existing platform. It can be revolutionary in a sense that it changes the life of millions or even billions, like mobile platform or cloud platform, or just a small change that affects a smaller number of people, like couple of hundreds.

Platforms like mobile platforms have increased extremely in popularity in the couple of decades [3]. It is a platform that has affected the life of billions and is one of, if not the most popular platform. Mobile platform is a software or hardware environment for portable devices such as mobile phones, laptops, tablets, etc. Everyone has access to it and can use it from almost anywhere in the world. It can be used for multiple purposes, communicating a family members or friends than can live on the other side of the planet, entertainment like listening to music or watching movies, and for contacting ambulance, police or fire department for emergency situation. The times it takes to contact the emergency service is reduced thanks to mobile platform. It also helps people who voluntarily travels to third world countries for helping people in need, and using mobile devices for finding the correct data for giving proper medication and care. However not all devices on mobile platform is powerful enough to do a task that requires some power. Which is where cloud platform comes in.

Chapter 2

Cloud Platform

For a person that has minimal knowledge of computer science, the term cloud may be confusing. Because the term they understand is the thing that is levitating in the sky, which NASA website has defined as "a mass of water drops or ice crystals suspended in the atmosphere. "[18]. However, in the world of computer science, it is basically a metaphor for the internet. The term comes from when back in a days, the symbol for cloud was used for describing network [20], whether it was sending data over internet or whatever the reason was, that symbol was used and the metaphor stuck.

A cloud platform is a platform for cloud computing, which means everything the user is doing, is happening on a machine or a server somewhere else in the world. A machine or server that is more powerful than the machine that is available to the user, but the user can use their own device as a tool to connect to that server and work through the cloud. That is why it is a popular platform, there is no need to buy a high-end computer for an outrageous price for only a temporary work. Renting a server that fulfils the requirement that is needed for the task at hand and for that period that that tasks needs is a better option and save plenty of time, money and resources.

It can be that a user needs it for processing power, needing more RAM than the user has available on their own machine, or it can be for storage purposes. Being able to have unlimited storage available in the cloud is a good luxury to have, there is no need to buy multiple hard drives. Files that are just available on your hard drive, can be corrupted, or the hard drive can be stolen, which means all of the data is gone. But if it is stored on the cloud, or a backup of the work is saved on the cloud, then it is available whenever it is needed, and it can be accessed from anywhere, all that is needed is a device with a network connection. It does not have to be the same device, it can be any device.

An exception is using network attached storage, also called NAS. NAS is a device that is dedicated to file storage and is connected with Ethernet cable. Because the data is stored on a home network, which is not using the cloud.

The internet has existed for a long time, and the reason for cloud computing to not getting popular earlier is because it was expensive. It was not only more expensive but also easier for companies to store their data and also run their applications on their own machines. Some software requires a higher bandwidth speed for executing the software properly, that kind of bandwidth speed was not available at that point, but it is now in certain parts of the world.

Storing a large amount of data was also not an option, for the very same reason as above. Storage was expensive and bandwidth speed was not ideal for data of large magnitude. Messages could be sent because it is not a large amount of data that are sent back and forth. But files can be larger. A 1 GB file, would take 42 hours with best modem connection one and a half-decade ago[22]. Which would not be a good solution if data was flowing back and forth all the time. It would take time for the larger files and sometimes too much time.

2.1 Cloud platform for businesses

For business, cloud platform can come in a form of different services. Software as a Platform[7], also known as SaaS, offers subscription-based applications to business that they access through the cloud. No need to worry about installation or setting it up on their own computers. The clients just have to pay and use the applications that are offered, the platform suppliers handle the rest. Examples of SaaS is Google Apps like Google Drive, Docs, Gmail and so on. Office 365 is another, where the client can use the office version that runs on their browsers.

Platform as a Service[7], also known as PaaS, is providing a platform for the companies so they can develop their own application that they can run and maintain. Administration of network, servers, operating systems and runtime environment are some of the features that are included in PaaS. Some features are optional, some platform owners offer them and some do not, which is databases, queue systems, and other storage services. Examples of PaaS are Google App Engine, Windows Azure, AWS Elastic Beanstalk and so on.

Infrastructure as a platform[7], also known as IaaS, is when the suppliers offer the backbone of the system only to rented out to costumers so the clients can build their system on top of it. For customers, this means that they have to do everything themselves. Like maintenance, installation, setting it up and so on. For suppliers, this means that the resources, such as servers, network, and storage are all moved to cloud for the clients to use. Services of this type are Microsoft Azure, Google Compute Engine, AWS Elastic Compute Cloud.

All types of cloud services and their features are shown in figure 2.1. The grey boxes are what the customer has to do, while the white boxes are what the suppliers are taking care off. If using any of them are out of question, then the responsibilities of all 9 are on the user self.

2.2 Cloud platform for single users

With cloud computing increasing in popularity, more and more companies offer cloud computing solution to their users. For example, Microsoft offers Office 365, for running Office application from users browser and using Microsoft's own storage service, OneDrive for storage. Google has plenty, like Gmail, Drive, Docs, and Maps and so on, all services that the clients can run from their browser.

Apple has iCloud, where they synchronize the client's mail, contacts, calendar and more. The interesting with this is that Apple uses Google Cloud Platform and Amazon S3's storing services for storing their client's data. Google and Amazon cannot read those data, as they are all encrypted and Apple has the encryption key and metadata stored on their own servers. The reason it is interesting is that Apple is a competitor with both these two companies. Using your competitor's services is a way of saying "use what works".

Amazon has also a cloud storage services, called Amazon Cloud Drive. It is more of a library for their client's digital purchases, like music, books for Kindle and unlimited storage for images if they are subscribed to Amazon Prime.

There is also hardware made for cloud purposes only as well. Like Chromebook is a laptop that only uses cloud features[10]. It is basically having Google's browser as an operating system. It has a small local storage because the main focus of Chromebooks is using Google's services, such as Cloud for storage, Docs as office pack, Google Photos for images and so on. Basically using Google's own application online.

ChromeBit is another hardware made for the cloud from ASUS [1], with it being in the size of a chocolate bar and using its HDMI output, it can turn any monitor with HDMI input to a fully functioning computer. It runs also ChromeOS as operative system.

Both ChromeBit and Chromebook has one major disadvantage, in that its offline functionalities are limited, but it is expanding. Some of them are usable, but they can not be synchronized without a network connection, for example creating and editing a file is possible but other devices cannot see the updated file, because the network is needed for synchronizing the file with the version on the cloud. Storage is limited because it is meant for downloading content for offline use, like music, movies or playing games.

There have been other hardware devices meant for cloud usage, like New Internet Computer, released in 2000. There was also NetPliance iOpener which was used from 1999 and 2002. And also 3Com Audrey released in 2000. A reason why this did not take off and became popular, could be because of the dial-up network. The limitation of dial-up speed held it back. Now the speed of the internet, at least in some places, are high and almost competing with the speed of local storage.

2.3 Arguments against cloud computing

Whenever a new technology is becoming popular, there are people that likes to bring up the disadvantages of it for discussion. Some are skeptical, some will not like the new technology at all. Cory Doctorow, an editor of a popular blog focusing on technology, futurism, gadgets called Boing Boing, says "network access remains slower, more expensive, and less reliable than hard drives and CPUs."[6] and that the connection is controlled by the user's internet service provider (ISP). It is a good point if the user is using Chromebooks for example. A lot of faith would be put on their ISP.

Steve Wozniak, co-founder of Apple Computer Inc. said following about cloud, "I think it's going to be horrendous. I think there are going to be a lot of horrible problems in the next five years."[16] He followed it with saying that he wants to feel like he has control over his data, which is the feeling he has when the data is on his computer. But he loses that feeling more and more when storing on the cloud.

Wozniak's concern is that who really owns the data, the user or the company storing it[16]. For example, Facebook and Instagram are known for changing terms and services often, which has caused controversy, regarding what to do with the user photos. The photo that the user has uploaded and the photo that the user is using with help of filters from the service providers in the cloud, are two different photos. Who can claim the one created in the cloud?

Another thing that is a big disadvantage is outages. It is a disadvantage for everyone.Like for example, in 2012, some of Amazon's servers went down because of electrical storms in Northern Virginia, USA [2]. This lead to services like Netflix, Instagram and Pinterest going down down, because they all use Amazon's cloud storage services. Another one happened in 2014, when services from multiple big companies like Microsoft, Adobe, Gmail, Dropbox, and iCloud went down due to another outage, and again in 2015 when services from some of the companies went down again. The power outage often leads to services going down for couple of hours. That is a disadvantage with the cloud, especially when multiple other services use their cloud service.

2.4 Summary

Cloud computing is the platform that is not new, but its popularity has increased in the last decade. The internet speed and access to network has contributed for its rise. With internet speed that are available today on wireless network and 4G network, the time it takes to send and retrieve large data has been reduced to minutes, rather than hours and even days some decades ago. It has lead to big companies to invest in cloud computing services.

There are different kind of services that are available for both enterprise purposes and for non-enterprise purposes, in other word, for people that wants to use it for themselves, like Google Drive, private online storage service from Google.

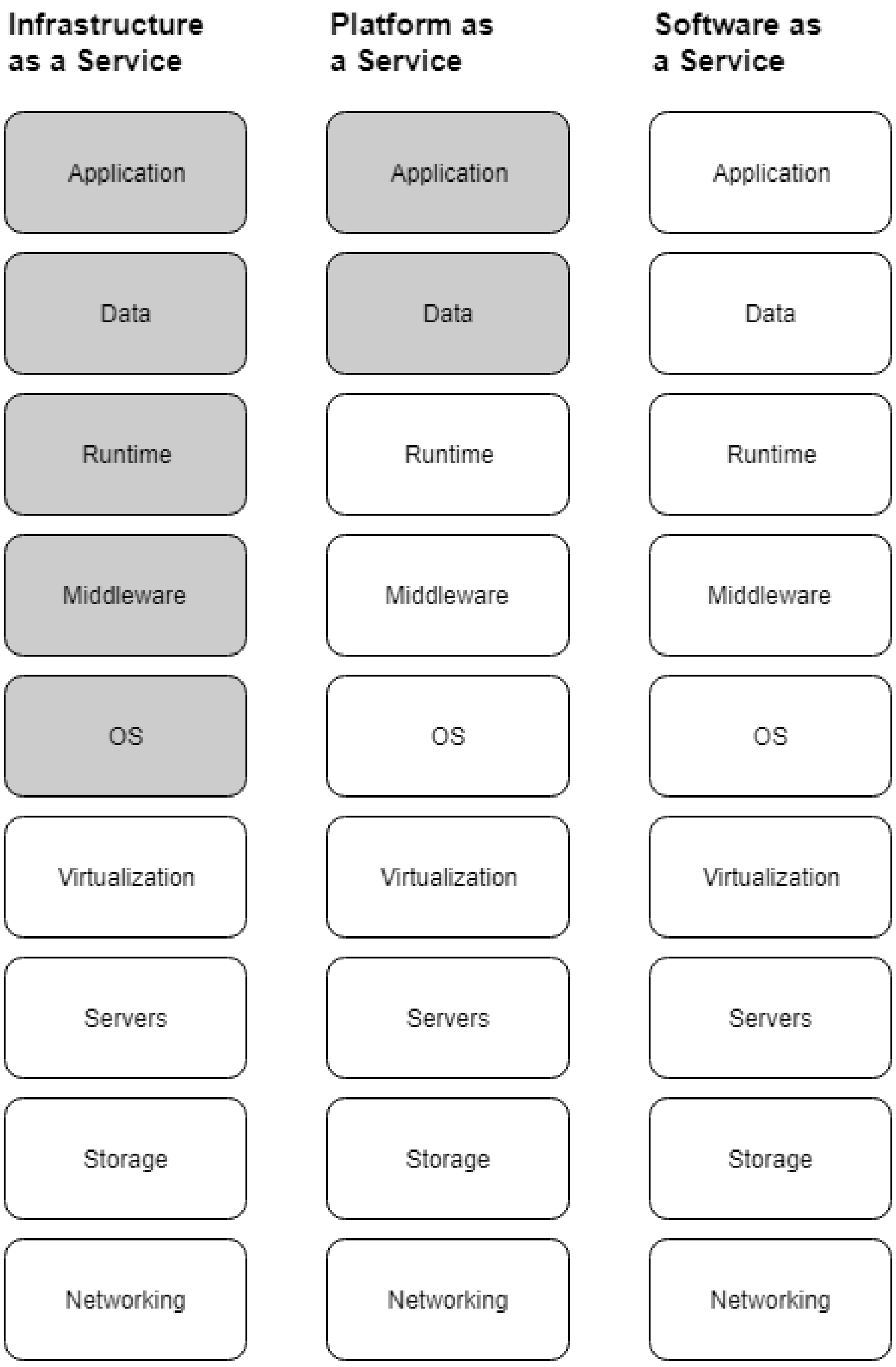


Figure 2.1: Each type of service types and what they offer, demonstrated in white boxes.

Chapter 3

Design pattern

After having developed in an object oriented language (OOL) for multiple years, a developer comes across a multitude of problems. After some while, the developer have a way to solve most of their common problems and note the solutions to self for later use. Doing this for some time and gaining some experience along the way, the developer have now a catalogue of solutions to common problems that the developer has come across. This catalogue of solutions is what we call design patterns[9, p. 3]. Solutions for the most common problems that the developers face when programming in OOL.

Design patterns was first introduced in the book from *Gang of Four*, Erich Gamma, Richard Helm, Ralph Johnsen and John Vlissidis, called Design patterns; Elements of Reusable Object-Oriented Software[9].

It is not some form of package or library that you import into your project and expect all the problems in your program to disappear. Guidelines is more like what design pattern can be explained as, a guideline on how a certain number of problems can be tackled with certain solutions. And the only thing that the developer should know for using design pattern, is just a regular OOL, like Java, Smalltalk, C++, etc.

Like explained earlier, it is not a a solution for finding problems, it is a solution to problems. It is not something a developer should force upon the development. That would be falling in a trap, called pattern abuse, and that makes the code a mess and that would not help the developers at all. The developers should also know that if design patterns has been used properly, then they should be able to update the program in the future without any big issues, because the code should be flexible and reusable, and that is the point with design pattern. Also you do not have to create new solutions for problems with existing solution already. Why reinvent the wheel. Solution that has solved the same problems earlier, should be good enough to solve the same problem this time.

Every design pattern should have these essential elements: a pattern name, problem, solution and consequence. A pattern name is used to specify the problem, solution and consequence in one word or two.

- Finding a good name is important as it makes it so easier to understand the pattern, to talk about the pattern and make reference to it when trying to compare with other design patterns[9, p. 3].

- The problem should describe which problem this pattern is designed for and its context. In certain situations, there are conditions to a problem, and in order for it to make sense to apply this pattern to those situations, these conditions has to be met•.

- The solution is not a description of an implementation that solves the problem, because this is a template, it should apply to different situations with similar or same problem. The solutions should give us an abstract solution to the problem and how to solve it with how the elements are arranged[9, p. 3].

- The consequence tells us what result of applying this pattern to this problem is and the trade-off's. This is important because it tells the developers about the systems flexibility, extensibility and portability as a consequence of applying this pattern. The consequence should also show the benefit of using this pattern compared to alternatives that could solve the same or similar problems.

The design patterns can be divided into three types: creational[9, p. 81], structural[9, p. 137] and behavioral[9, p. 221] pattern. Creational patterns describes a moment in the programs runtime, like when an object gets created. Structural patterns in a way describes the static structure of the program while the behavioural patterns describe a flow of the program. A better look on these different design patterns may be necessary to comprehend it even better.

3.1 Creational patterns

Creational pattern contains the design patterns that are focused toward creating, composing and representing objects. What makes creational patterns important, is that the system are more depended on creating objects, rather than how it is structured. So the focus shifts towards creating multiple smaller object behaviours that could be put together to create a bigger, more complex behaviour. In a way that would make the system more flexible, as the system in general only knows about the interfaces of the objects. It makes it flexible in what object gets created, who it is created by, how it got created and when. They, as the design patterns, also lets the system vary the structure and the functionality of objects widely.

3.2 Structural patterns

Structural patterns include the design patterns that are focused towards how entities in a system use each other, or answering the question "how to build a software component?" is asked. It helps with forming larger structures thanks to how it composes the classes and objects. In structural class pattern, inheritance is used to compose interfaces. While in structural object pattern, new functionalities gets realized by describing ways to compose objects. Since class composition is static, the composition doesn't change in runtime, unlike object composition, which can change at runtime, making it dynamic, which add flexibility.

3.3 Behavioural patterns

Behavioural patterns include the design patterns that are focused towards assigning responsibilities between the objects. It identifies the communication patterns between the objects. So these patterns runs the behaviour in software component. Behavioural patterns lets you focus on the way the object are interconnected rather than focusing on the flow between them.

To distribute behaviour between the classes in behavioural class pattern, inheritance is used. While for behavioural object patterns, object composition is preferred over inheritance.

3.4 Design Patterns used

In this section, the design patterns used will be introduced and explained what they actually do. How they are implemented is not explained here but will be in chapter 7.

3.4.1 Observer

The book from GoF describes it as a pattern that "defines a one-to-many dependency between objects. And that when the one object's state is changed, all dependencies are notified and updated automatically."[9, p. 293]

Problem

Imagine having a couple of objects of the same type, and they are all dependent on a state of another object. But without making them tightly coupled, because then the reusability of the objects would get reduced. The objects must somehow be notified when a state of that one object is changed.

Solution

Observer patterns allow an object to have a one-to-many dependency with other objects. It does it by separating all objects into observer or subject objects. The subject is the object that the observers subscribe to. Subscribers get notified when a change is made in the subject and their state can synchronize with the subjects state.

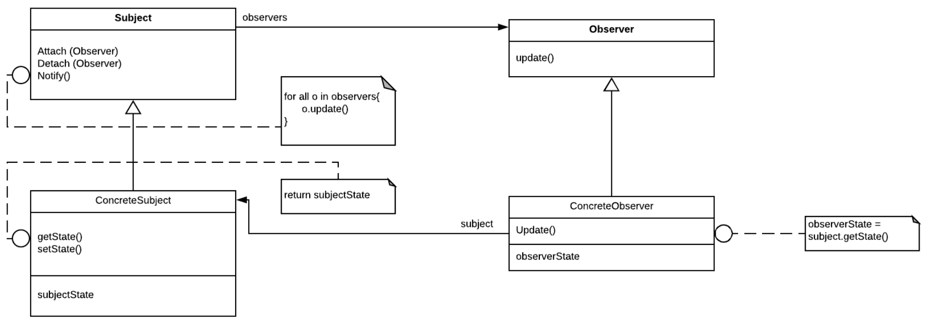


Figure 3.1: Sketch of a Observer pattern structure

Consequence

The subject just notifies the observers that the state of the object has changed, nothing about what has changed. If there are thousands of clients, then all clients must work hard for bringing the latest version of the subject. If all is running from the same machine, then it might take some resources to get it done fast, before any more changes are made in the subject. [9, p. 296]

3.4.2 Mediator

The GoF describes the mediator pattern as follows "Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently."[9, p. 273]

Problem

Imagine wanting to have multiple objects trying to communicate with each other, but not wanting to have them all closely coupled with each other, because that would increase the system complexity and maintainability. Adding more client would increase much more, as all the other object has to be updated about this new object.

Solution

With mediator pattern, a new object, called the mediator, gets created to handle the communication logic between the objects, called the colleagues. For each colleague that want to communicate with each other, instead of each one of them being coupled with all the other colleagues, it can just be coupled with the mediator. When new colleagues are created, instead of coupling it with every colleague in the system, it can just be coupled with the mediator.

Mediator pattern promotes one-to-many relationships, rather than many-to-many relationship, and also loose coupling. By reducing the coupling for the whole system, it makes the system less complex and more maintainable because each class is not very dependent on each other anymore.

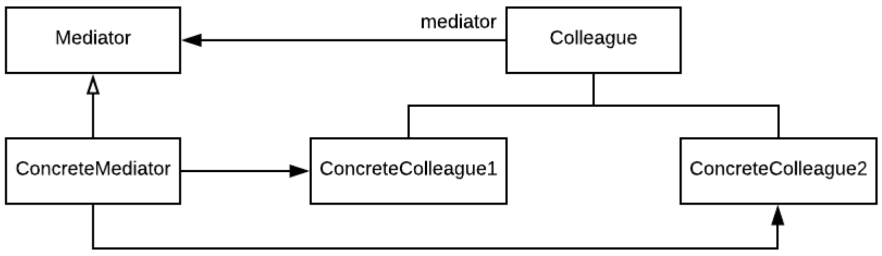


Figure 3.2: Sketch of a Mediator pattern structure

Consequence

The mediator pattern centralizes control of interaction in one class, all of the logic is in one class, by encapsulating the logic of mediation in the mediator, it makes it easier to understand the system.

By decoupling the colleagues, adding new colleagues is easy, it just needs to be coupled with the mediator, and it is ready to communicate with the other colleagues. By simplifying the object protocol from many-to-many relationships to 1-to-many relationships, it makes the system easier to understand, maintain and extend.

Sometimes extending the logic is necessary and it is easy to do by just extending the mediator. But it is important to note that the mediator pattern trades the complexity of object interaction with complexity in the mediator. The mediator can become big and hard to change if not careful about extending it properly, which makes it harder to maintain. [9, p. 278]

3.4.3 Strategy

Strategy pattern's intent is described as follows in the book from GoF, "Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets algorithm vary independently from clients that use it"[9, p. 315].

Problem

Imagine a user having two different algorithms, but does not know for sure which algorithm to use because the data is of different types. Another factor is that the situation can change and each algorithm is related to different situations.

Solution

With Strategy pattern, each algorithm gets isolated into their own classes. So whenever the user wants, they can change to another algorithm based on the situations and data. Situations can change at runtime and is possible with Strategy pattern, the algorithm related to the situation or data can be changed at runtime.

An example is the operating system of mobile phones. When the battery level is above 20%, it will run on normal mode, when its between 10-20%, its in power saving mode and for some phones, under 10%, it turns on ultra power saving mode.

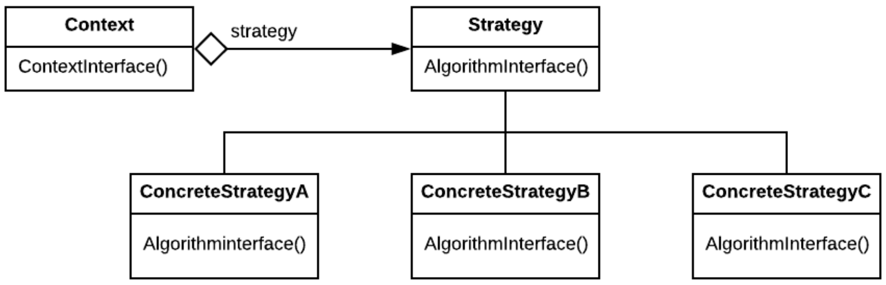


Figure 3.3: Sketch of a Strategy pattern structure

Consequence

The Strategy pattern defines a family of related algorithms and behaviours. The common functionality gets implemented in a class, and each algorithm can implement the functionalities that differ from other algorithms.

Another possibility is to implement a default algorithm, whenever it gets checked whether a strategy object exists or not, if it does not exist, the default algorithm would get executed, if a strategy object does exist, execute the strategy object algorithm.

A drawback is that the client must know the difference between each strategy implementation for it to choose the appropriate strategy for the situation. Which is why Strategy pattern should only be used when the variation of each strategy is known to the client. If not, then the client might choose the wrong strategy and the Strategy pattern fails to fulfil its intention. [9, p. 317]

3.4.4 Singleton

Singleton pattern makes sure that only one instance of the object can be created, and making that instance global for everyone to access [9, p. 127].

Problem

Sometimes there should be only one instance of something. Like for example file system on a computer. It should be globally accessed, which means it can always be opened from anywhere.

Solution

Singleton pattern makes sure that a class, called the Singleton class, have only one instance. And it makes sure by having the responsibility to instantiate self and have a global point of access for that instance.

It has a method, which returns the instance. If it is the first time and it has not been instantiated yet, then it creates an instance and returns it. But if it has an instance available, then it returns that instance.

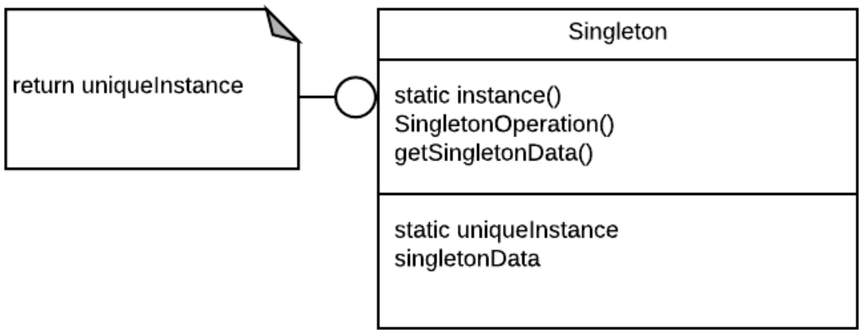


Figure 3.4: Sketch of a Singleton pattern structure

Consequence

By not handling multiple threads accessing the method that returns the instance of the class, multiple instances can be created.

Singleton pattern can help reduce the global variable for a class, by storing them all in one Singleton class and retrieve it from there whenever needed to avoid pollution of the namespace. [9, p. 128]

3.4.5 Template

"Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template Method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure. " is how the book from the Gang of Four describes Template patterns intent[9, p. 325]

Problem

Imagine having two different components that follows the same steps. Imagine it as an algorithm. The steps are the same but each algorithm has some unique parts for each one.

Solution

Template pattern has a method in the parent class that describes each step of the algorithm, which other classes can extend and implement the abstract methods that are different for each class. Like for example, there are similar steps in building all kind of houses, which is preparing the base, building the walls, adding roof and floors. But the difference is what kind of materials is used to build that house, whether it is made of wood or cement.

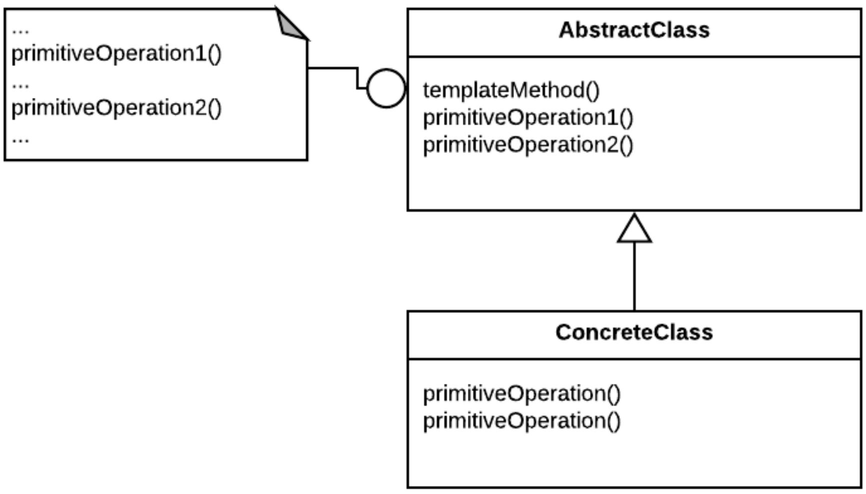


Figure 3.5: Sketch of a Template pattern structure

Consequence

Unlike common parent and children classes, in Template pattern, the parent class calls the methods on their children classes. It is done from the template method.

A disadvantage is that since the methods from the parent class can be overwritten, the template method can be overwritten in the child class. It can be on purpose or accident, but the possibility is there. It can reorder the ordering of the invocation of the methods[9, p. 327].

3.5 Summary

Design patterns was first introduced in the book from *Gang of Four*, called Design patterns; Elements of Reusable Object-Oriented Software[9]. They describe design patterns as recurring solutions to problems that the developers face regularly. This is not a package that can be integrated with

IDE and solve all the problems. It is an idea that has to be adapted to the problem in hand.

As the title of book states, these solutions are for problems that occur in object-oriented programming, for languages like Java, C++, C#, etc. As previously mentioned, all design patterns can be categorized into three categories:

- **Creational patterns:** creating, composing and representing objects.
- **Structural patterns:** how entities in a system uses one another.
- **Behavioural patterns:** distribution of behaviours among the objects in the system.

Chapter 4

Measurement

Measuring is vital part in engineering. It is through measuring the engineers find out whether the product they have created is good or not. In this chapter, it gets explained why measuring is vital, what kind of data there are, why data is important and why the right kind of data is necessary for the measurements to mean any anything.

Measuring is also important in computer science, for example when comparing algorithms, which one is faster and more secure that the other ones. It is also important for whole systems for calculating its complexity and maintainability.

4.1 Data

Data is the most important part of a research. Data is factual evidence collected during the research and a conclusion is made from these data. There are multiple types of data, the type of data that is relevant for a research, depends on the research method. Data that can be collected during a research is called research data, and it can split into objective or subjective data.

4.1.1 Objective data

Objective data is data that is based on observing facts that can be measured[23]. Data that is *not* altered by personal opinions or views. Objective data is like facts that are information delivered in a non-partial manner. It is the kind that can be found in encyclopedias and textbooks for example. Another area where objective data is essential is news reporting. News that is being reported has to be based facts and delivered in an in a non-partial manner.

For example in a war zone, news can only report the number of casualties or what was it that sparked the situation that got it out of control, not who the "bad" or "good" guys are. A computer science related example would be the number of downloads an application has. It is just a number, that does not get affected unless someone downloads the application. It is objective data that should not be altered in any circumcises except when someone is downloading the application.

4.1.2 Subjective data

Subjective data is data that is related to peoples experiences and their opinions on them in their own mind[**WebsterMerriam**]. Rather than being based on facts, it is affected by personal views, experience or background. Subjective data is useful for editorials and reviews.
If we go back to war zone example, both sides see themselves as the "good" guys and the other side the "bad guys". Because of their views and background, they believe that whatever they are doing is right, no matter what the rest of the world thinks. In their mind, they are protecting themselves against the "bad guys". In computer science example above, it would be for example the reviews of the application. If it gets a bad review, it is the opinion of the individual who tested it and did not like it, the same application can get a great review from another user.

In research, objective data is the kind we prefer. It is data that can be measured, data that are not affected by views, personal feeling or experience. It is facts. That is why we prefer it. However, we do prefer good subjective data that are related to the topic of the study, over objective data that is not related to the topic. Good subjective data can give good and valid information that can help, while the unrelated objective data would not help.

4.2 Metrics

One vital criteria that are important in all form of improvement work, is measurements. To find out if a process, for example, is improved, then it has to be measured against another process, either older version of the same process or different but relevant process. It can also help us improve our understanding of a topic. As the physicist Lord Kelvin says:

> *"I often say that when you can measure what you are speaking about, and express it in numbers, you know something about it; but when cannot measure it, when cannot express it in numbers, your knowledge is of a meagre and unsatisfactory kind."* [13]

It is an important factor for science to advance. Metrics are often used for comparison. For finding potentially the best result among other options, based on metrics that are pre-defined. Without proper metrics, science might not have been in the position it is today. Results from newer research can contradict results from older research, and potentially replace it as fact if multiple reliable sources can confirm it. An example of this is cars that use diesel engines. It was widely known that diesel cars were good for the environment, but as new data suddenly showed that is not the case, it is not better than normal petrol.

There are two main types of measurement. Qualitative and quantitative. Both types are widely used form of measurement and have their purposes for the different kind of measurements and research they are used for.

4.2.1 Quantitative measurement

Quantitative measurement is a measurement of quantitative data. In their book *"Statistics"*, Robert S. Witte and John S. Witte define it as follows:

> *"When, among a set of observations, any single observation is a number that represents an amount or a count, then the data are quantitative."* [24]

Running statistical analysis is the goal of quantitative measurement, which means that the data has to be expressed in numerical form. Examples like size, length and amount are typical examples of quantitative data.

4.2.2 Qualitative measurement

Qualitative measurement is data that consist of words. Robert S. Witte and John S. Witte have a definition of it as well, and it is as follows:

> *"When, among a set of observations, any single observation is a word, or a sentence, or a description, or a code that represents a category then the data are qualitative." [24]*

Qualitative data cannot be used for statistical data because it does not deal with numbers, it deals with pictures and words. Common methods for collecting data for qualitative measurement are in-depth interviews, direct observation of participant of the project and reviewing already existing documents. The goal is to capture the subjective and less quantifiable aspects of people.

4.2.3 Qualitative Vs. Quantitative measures

This might look like that all data that can be used in quantitative measurements has to be objective data. But not necessarily, quantitative data can be subjective as well. An example of subjective data that is also quantitative is asking: *"On a scale from 1-10, how good is the user interface of an application?"* and someone answers *"8"*. 8 is quantitative but it is from a subjective opinion, 8 is from the users perspective compared to experience from past application. Another user would maybe answer *10* or maybe even *1*.

An example of objective quantitative data is for saying *"The chip speed of my computer is 2 GHz"*. It is objective because 2GHz is objective data, the chip speed is not from a personal view or opinion, it is fact that comes from the chips manufacturers and that has been tested multiple times to find the correct speed. If the manufacturers, does not specify the correct data of the speed, then they will lose credibility and respect among their competitors, current and potentially new clients. They will lose clients and potentially partners for future projects. The data tells what the speed of the chip is, in this example, it is 2GHz. Which is a number, which means it is quantitative data.

Since subjective data can be quantitative, objective data should be able to be qualitative, right? Yes, objective data can also be qualitative. For example saying "Yes, I bought a car". It is expressed as text and cannot be measured but observed, which means it is qualitative. Since it is based on facts rather than feeling or opinions, it is also objective, because the individual bought something.

An example of subjective data that is qualitative is like saying "I think it is quite cold in here". It is subjective because it one individuals opinion, another might not share the same opinion. Therefore it is a subjective opinion. The fact that the individual just says that it is cold and does not specify temperature, qualifies it as qualitative and not quantitative.

4.2.4 Measurement in software engineering

A software engineer is always out to improve a process or product. When a software engineer put pieces together and produces a product, they always try to improve it further. No matter how good it is, it can always be improved.

The data that are being collected, can be either objective or subjective, for qualitative or quantitative measurements, or measured directly or indirectly. What measuring directly and indirectly means is whether the data is internal attribute or external. Internal attributes mean that the data is measured directly from the program, like lines of code, coupling or bugs per lines of code [8, p.88]. External attributes are data that are measured indirectly through the systems appearance, line usability, maintainability, and reliability[8, p.88].

Good data that is relevant is essential for the research to have a meaning and impact for the system. Which metrics that gets used, depends on which of the categories that are going to be measured. The different categories are:

- **Product metrics:** Measuring the product itself, like size, complexity, performance, features and quality level.[12]
- **Process metrics:** Can be used for maintenance and improvement of software development, like how effective the removal of defective is. [12]
- **Project metrics:** Describes the characteristics and execution of the project itself, and not the product, like resources like cost, developers, life cycle, schedule of the project.[12]

Inspection of the code of the application is called a software application assessment, and it is important to identify areas that may be at risk for potential security flaws and/or performance constraints. Fixing potential vulnerable areas will help the software reach its full potential. The maintainability and architecture are some the areas an inspection can do and it will make life easier in the long run.

Imagine if being assigned to a new project, and a developer, developer A, from this project have not used assessment of the code at all, and the code will most likely be a mess unless the former developer, developer B, on the project was a very good developer. But if B did not use assessment, then it would take time to comprehend the code and even more time to fix it to make it more comprehensible, so that the developer A may use less time on this part. But if the developer B has used assessment wisely and continuously throughout the project, it leads to shorter time spent on trying to comprehend the code for developer A, because it is much easier, and more time to actually expand the functionalities.

Since software application assessment is about a systems improvement, metrics are needed to compare the old versions with the new version. It is easier to see if the systems maintainability has gotten easier or harder, maybe how the programs structures complexity has changed in the last month or week.

4.2.5 Basic Code Metrics

There are many metrics that gives good information that some can be learned from when analysing software source code. Some of these metrics are very easy to measure, others might be a little more complicated, like lines of code is easy while finding the maintainability index may not be that simple. Here are some of the most commonly used metrics used for analysing source code.

Maintainability index

The maintainability index shows how easy or hard the systems maintainability is [19]. This metric can be used on multiple levels, like method, class or project level. What the index means, depends on which formula gets used in the calculation. The original formula for finding maintainability index is as follows:

$$index = 171 - 5.2 * ln(avg(H)) - 23 * avg(C) - 16.2 * ln(avg(L))$$

where H stands for *Halsteads Volume*, C for *Cyclomatic Complexity* and L for *Lines of Code*. The index result from this formula ranged between 171 and a negative number which was unbound. This can be used with finding average values for H, C and L on same level, either that is project, class or method level.

According to engineers from Microsoft, the difference between 0 and a negative index was not useful for them to learn anything from. That is why they created their own formula. And the new formula from Microsoft is as follows:

$$index = MAX(0, I) * \frac{100}{171}$$

	Total	**Unique**
Operators	N_1	n_1
Operands	N_2	n_2

Table 4.1: Four basic metrics of Halstead

Where I is the maintainability index[19]. The index in this new formula is between 0 to 100, where lower index indicates low maintainability, which means it is hard to maintain. While higher index indicates high maintainability, which means the system is easier to maintain.

Halstead compexity measures

Maurice Halstead developed Halstead complexity metrics in 1977 [11], with a goal, which was to identify measurable properties of a software, and how these properties were related. Operands and operators are used for determining the quantitative measures of complexity, and it is used directly on the source code. Since the measurement is directly in the code, it can also be used for maintainability. And maintainability is something the developers should have in the back of their head during the systems development.

Operands and operators are used for interpreting and measurement. Explaining what word or symbol operands and operators are, differs from language to language. Like for example *abstract* is reserved word in Java, but not in C. So *abstract* is operand in C but operator in Java. So considering Java is the language of choice, the examples of what is considered operands or operators, will be given in Java.

Operands are considered tokens from following categories:

- **Identifiers:** strings that are not reserved keywords, like variable names.
- **Constants:** characters, numbers or string constants
- **Type names:** Keywords that are reserved in the language like variable type specifiers, like *String, integer, float, long, double, char etc.*

Operator are considered tokens from following categories:

- **Reserved:** keywords that are reserved in Java, like *while, for, break, const, final, break, abstract, switch, interface, extends etc.*
- **Operators:** symbols that can be used in the Java language, some examples like *=,!, !=, <, <=, [,], (,), ,* .

When all operands and operators have been interpreted from the source code, then four basic metrics are made, which can be seen in table 4.1.

These four metrics are all that is needed for using Halsteads more complex metrics. The complex metrics can be used for other measurements that may not be one of Halstead metrics, like Halstead's volume metric is used in maintainability index.

The first complex metric is finding the total number of tokens in the program[11]. Which says the total amount of tokens that are in the area that is calculated. The formula for finding the total N is:

$$N = N_1 + N_2$$

The second metric that is important, is finding the total number of unique tokens in the program[11]. So unlike previous, no matter how many of same tokens there are, it is still counted as one. The formula for the number of *unique* tokens are:

$$n = n_1 + n_2$$

Halstead's volume is a calculation of information content of the program in bits[11]. It is based on the number of operations that got performed and the number of operands that got handled in the algorithm. The volume should be at least 20 and a maximum score of 1000 for method level. If it is higher than 1000, then the method is probably doing too much and should be split into multiple methods. For a file, it should be at least 100 and at most 8000. The formula is used for finding the program's volume V is:

$$V = N \log_2 n$$

The difficulty metric is meant to show the difficulty level of the program implementation or how hard or easy it is to understand the program during a code review for example[11]. Formula for finding difficulty D is:

$$D = (n_1/2) * (N_2/n_2)$$

The effort to implement is a metric that tells how much effort is required to implement or understand[11]. The metric is proportional to volume and difficulty metrics. If a class score a high score on difficulty metric and on volume metric, then it will take a longer time to understand it and take a higher amount of effort to implement it further. The formula is as follow:

$$E = V * D$$

All of these metrics are important because they gives plenty of information about our system. They are all relevant objective data, which is exactly what is needed.

Cyclomatic Complexity and Cognitive Complexity

Cyclomatic complexity calculates logical paths through source code. Each path is independent, which means that it is a path where there is at least one unvisited node[14]. It can calculate on multiple levels, like methods, classes or modules. However it is best in method level. It is at least not recommended on class level, and definitely not on module level. A class with a medium score does not tell us if it is a big class with low complexity or small class with high complexity. On method level, the amount of path that can be taken, will make sense and help improve it when the index is too high.

The cyclomatic complexity index is recommended to be below 10, that means the method is structured and well written and has high testability. Between 10 to 20 should be rare, more complex code and does not have the highest testability but it will do. Higher than 20 means that it is too complex, needs to be simplified and low on testability.

Thomas J. McCabe developed this in 1976. Which makes cyclomatic complexity an old metric that is still being used, which means it is a metric that gives good information. It has been implemented for multiple programming languages, which makes it widely spread used metric.

Representing the program as control flow diagram, which describes the flow of the process. A control flow diagram is made up of nodes, which represents processing task, and nodes, which represents the flow between each node.A simple description of how it looks like is shown in 4.1

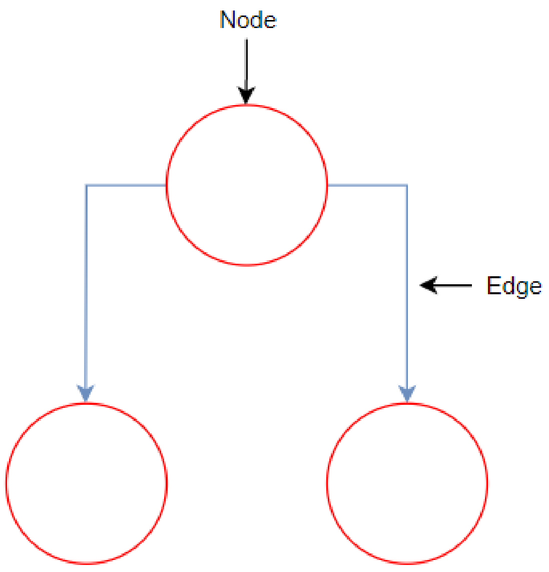


Figure 4.1: A model showing what nodes and edges are in a control flow diagram.

The mathematical formula of finding cyclomatic complexity index is:

$$V(G) = E - N + 2$$

and

$$V(G) = P + 1$$

where E is the number of edges, N is the number of nodes in the graph and P is number of nodes that contains conditions. For example, in 4.2 shows a method's control flow diagram. It has 6 nodes, 7 edges and 2 nodes that contains conditions. The cyclomatic complexity would then be:

$$V(G) = 7 - 6 + 2 = 3$$

$$V(G) = 2 + 1 = 3$$

The cyclomatic complexity of the graph on 4.2, is 3. The set of potential paths are:

- $1, 2, 3, 5$
- $1, 4, 3, 5$
- $1, 4, 6, 5$

Which is correct. There are no other paths left. Another phase where this is important is in the testing phase, where it is required to know the number of test cases required for a class or method. Testers make tests for every outcome and with a number of test cases required are equal to the cyclomatic complexity index, it is easier to find the number of test cases required for getting 100% branch coverage. Depending on the size of the program, if it is small, then the cyclomatic complexity index can be counted manually, but if it is a big and complex system, then automated tools are used to calculate the metric.

Cyclomatic complexity gives a good indication for testability of the program, information about the least amount of tests that are necessary, but it does not give information on the maintainability of the program. It is made in 1977 since than languages has changed. It is not necessarily good with modern languages features like try-catch. Which is why Cognitive complexity is developed.

While cyclomatic complexity gives useful information on method level, Cognitive complexity can give useful information on class and application level[4]. Which is useful when a class has too many methods and testing each would take too much time.

The assessment of the program is based on three basic metrics:

- **Ignore shorthand**, for example, having methods allows the developers to compress statements into a single call, therefore Cognitive complexity does not increment for methods.
- **Increment for breaks in the linear code**, these breaks leads to developers use more time and effort to understand the code. These breaks can be loop structures like for- or while-loops, or it can be conditionals like ifs, else ifs or else. Catches are increased with one, no matter how many types of exceptions are caught. try and finally are ignored. In switches, no matter how many cases there are, it only increments with a single increment in Cognitive complexity, while in cyclomatic complexity, it increases for each case like an if-else-if chain. It is because it is much easier to understand a switch-case from maintainers point of view.

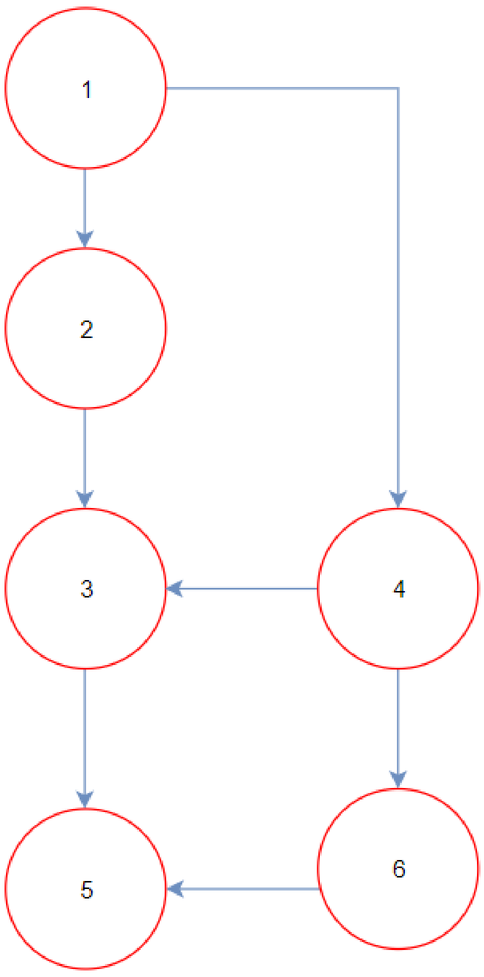


Figure 4.2: Example of a methods control flow diagram

- **Increment of nested flow-break structures**, which means that for each if-statements, for-loops or any other flow-break structures, nested into one another, it increases according to it. If the structures are after each other, it is easier to read and understand for maintainers point of view than it is if all of them are nested into each other.

Class coupling

Class coupling indicates the coupling to other classes. Coupling is when methods or variables that are declared in one class, are referenced in another class. Then these two classes are coupled. Even if there are multiple references for methods or variables in the other class, coupling metric does not increase, it is not for each reference, just as long as it happens at least once.

A low number of class coupling is desirable because it means that the system is not complex, easier to maintain because of its dependencies to other classes and it has low sensitivity to changes. A high coupling index indicates that it is highly sensitive to changes, because a change in one class, that is referenced in multiple other classes, can crash the program. The maintainability of that situation might take time to fix.

Lines of code

Lines of code indicates the approximate lines of code. It counts only lines that are part of the code, which means white spaces, tabulations and lines that are part of comment does not get counted. Otherwise it would be unfair, for some prefer to have a lot of comments on their code, while others prefer to have minimal of comments.

Lines of code metrics can be used for methods, classes, projects or application level. It is easy metric to use for comparing two classes for example. Some prefer to use it for methods, to see if the it is too long. Some say that if the whole method cannot be seen on the screen with scrolling, then it is too big.

Other metrics

There are many more metrics that are being used, like class count (describes number of total classes, classes that are part of test, etc.), Martin Packaging also called Software package Metrics and many more. But the one explained above are the most popular ones, however there might be more that are popular metrics, but that has not been explained above.

4.3 Tools

Some metrics can be easily checked without tools, like finding lines of code in the project or in a file. But sometimes that is not enough, sometimes better metrics are needed to find some deeper information about the system, metrics that may find a weakness we may not find with the naked eye, and for that, tools may be needed. At least two tools will be used to compare the solution against each other to find the better ones.

The kind tools that are going to be used for testing the system are static analysis tools. Which means that the code is not executed or run but the tools themselves are executed. The compiler itself has static analysis features, but they are minimal at best, these tools work as extensions for compilers. Some of the features that static analysis tools have are that they are enforcing coding structures, analyzing the structure and dependencies, identifying anomalies and defects in the code.

The name of the static analysis tools that are going to be used in this thesis is SonarQube[21] and MetricsReloaded[17]. SonarQube is a tool that is not internal in the IDE but a software that has to be downloaded, installed and then used, while MetricsReloaded is not. MetricsReloaded can be executed from the IDE itself.

4.3.1 MetricsReloaded

MetricsReloaded is a static analysis tool that can be integrated into in IDE for developers and test the quality of the system. It has plenty of metrics that can be used, but only some of them is relevant for us. The ones that are relevant to us is:

- **Chidamber-Kemerer:** consists of 6 metrics calculated, like method count for each class, depth of inheritance tree, number of children, coupling between each classes etc. [5]
- **Complexity:** Details of cyclomatic complexity, essential complexity and extended cyclomatic dependencies of methods.
- **Lines of Code (LoC)**

4.3.2 SonarQube

SonarQube is a free open source software used for static code analysis. According to their own website sonarqube.org, SonarQube is "the leading product for continuous code quality" and it is used by more than 85000 organizations. It is developed by SonarSource. It is developed in Java but can be used for more than 20 other programming languages. Unlike MetricsReloaded, SonarQube will not be integrated into the IntelliJ IDE. It will be an external software that is installed separately on the computer.

There are many measuring metrics that are available in SonarQube. Some metrics are available on both, like cyclomatic complexity. Having two different static analysis tools can be good for having multiple objective data on the same metric for comparison. However, the metrics that are going to be interesting to us from SonarQube will be:

- **Cognitive Complexity:** A new metric that has been created for covering the shortcomings of complexity cyclomatic. With telling how difficult the code will be to read and understand, and not only the difficulty of being tested like complexity cyclomatic.
- **Number of classes, files and methods**

For using this software, a scanner application has to be downloaded and added to Window's environment variables for being called from the command line. Another little program must also be downloaded for starting the server. When the scanner has completed its run on all the files from the project, the score gets sent to local server, which can be seen from the browser.

4.4 The problem

More and more people use cloud services. Because it is more convenient for them, rather than having to bring extra devices or a heavier device. For example, instead of having an extra hard drive, why not using a storage service. Some services offer unlimited storage for clients that pay for it in a subscription model. Instead of buying a high-end computer for a temporary work, it is easier to just rent it for a period. There is no need for finding the right hardware for it all to be compatible. It is easier to have someone find and set up all in a proper way.

Pictures and videos are examples of this because they have increased in massively in size, because of their increase in resolution and frames per second (FPS) for videos. The higher resolution and/or FPS, the more storage space it requires. Which is why storage services are extremely popular for photographers and people who make videos, instead of having to bring a couple of hard drives and laptop.

More and more people are using cloud services as a communication tool. Either it is by chatting, or talking and having video chat with Skype, Viber, Facebook Messenger or similar. Being able to run Photoshop from a cloud in the browser, is helping the users with not needing to buy a high-end machine for being able to run the offline version of the software.

Design patterns are solutions that can help the structure of the program, making it more flexible, easier to maintain and reduce complexity if used correctly. It should not be forced to choose and implement a certain pattern. It should make sense to use the pattern if the structure and the intent of the system match the patterns.

With the cloud becoming as popular as it is, this thesis will take a closer look at how design patterns fit into this popular area. Whether design patterns helps with reducing complexity and increasing maintainability of the system or not.

4.5 Summary

Metrics are important in engineering. It makes measuring simple by comparing the result from the metrics to see which one is better and which one is worse. It can also be used for finding the weaknesses and strength of a system by using well-defined metrics. It all depends of course on data that is gathered. Whether this is objective or subjective data, and whether this is used in a quantitative or qualitative measurement.

In software engineering, metrics are divided into three different metrics: product, process and project. Product metrics are the metrics that measure the product itself. Process metrics are used for maintenance and improvement of software development process and project describes the characteristics of the project itself, like cost and developers.

There are some code metrics that are available for measuring complexity and maintainability of systems, like the maintainability index, Halstead complexity metrics, class coupling, Cyclomatic and Cognitive complexity among some that are explained. If the project is big, then tools can and is used for calculating some of these metrics but not all of them.

Part III

The project

Chapter 5

The Experiment

This is an empirical case study research that focuses on the impact of design patterns on applications using client-server architecture, as they are inspired by cloud. Thus two different cases have been chosen and implemented with three different design patterns. In this chapter, a short explanation of what kind of cases they are, how they are related to cloud and how they are measured against each other. The metrics used is explained in earlier chapters for better understanding of what they do and mean.

5.1 Cases

Two cases are implemented for this research. Both cases must use client-server architecture. The cases could be different but the ones that are chosen, are chosen because it is two very popular type of services and different enough that it is worth experimenting with.

Case one is a chat service where multiple clients can connect to service and join the same room. When they have joined the same room, they can communicate with each other through this room. There are multiple services like this, like Facebook Messenger, MSN messenger (which does not exist anymore) and Slack, which is quite popular for companies when their employees can discuss an issue together.

Case two is a storage service where clients are able to store their data in their private area on the service. In this service, each area can only be "owned" by one client at a time. It is so customers can store their private data that they do not want to share with other. There are similar services that give their users the possibility for storing and sharing the data from their room with other users, but in this case, that will not be possible.

5.2 Design patterns that are implemented

The chosen design patterns are:

1. Strategy pattern with Singleton pattern

2. Template pattern
3. Mediator pattern
4. Observer pattern

Number 1 and 2 are implemented in both cases. Where each case has implementations with these two. While number 3 and 4, all implementations in case 1 uses these patterns.

5.3 Measure

Halstead complexity metrics are used for decades, along with Cyclomatic complexity metric for calculating the complexity of a system. Sonar decided to make their complexity metric that is supposed to be modernized and updated with the evolution of programming languages.

Maintainability index is used for calculating the maintainability of the system. But engineers from Microsoft was not fond of it, as it had a maximum score(171), but the no minimum score, it could just receive a negative number. Therefore they decided to tweak it to give it a minimum score(0) and a maximum score (100).

Class coupling tells us how much coupling there is between classes. The higher the coupling is, the more sensitive it is to changes and higher complexity is.

Static analysis tools will be used for calculating these metrics. The tools that are going to be used are SonarQube and MetricsReloaded. They are explained further in chapter 4.

5.4 Report results

The metric for both cases and all three implementations is calculated and compared to each other. Implementation with Template pattern gets the overall better score, thus making it the better implementation among these three implementation when it comes complexity and maintainability.

Observer pattern works but has its disadvantages in the protocol used. But it may be the best protocol to use for this patter. Mediator pattern also works great. But it might work better with remote method invocation (RMI), however, it has not been tested.

5.5 Relation to cloud

It is a client-server architecture system. The service is running on a cloud server, in this case, the service is tested and executed from a server on PlanetLab. Multiple nodes have been used and tested to make sure it can work on multiple nodes, from different parts of the world.

For establishing the connection between client and server, TCP-protocol has been used. It is a safe protocol, which means that a connection has to be established first before data can be sent. Also, the data that are sent must be received in same order, and the new data will not be sent before the previous data is received.

5.6 Summary

Both cases gets inspiration from cloud in a different manner. One is using it for chat service and the other for storage. Each case has three different implementations that are measured with well know metrics and compared against each other.

The three different implementations are:

1. Implementation without any specific design pattern.
2. Implementation with Strategy and Singleton pattern.
3. Implementation with Template pattern

Since Mediator and Observer pattern are included in all implementation for case 1, there will be my opinion of how they fit into the structure of this system.

Chapter 6

Cases

In this chapter, both cases will be explained, not the implementation, but what they are, what kind functionalities they have and why they have been chosen. Which functionalities are similar and which are not.

The two cases that are chosen are a chat service and a storage service. Both cases are using a service that is running on a server. But the services has different purposes.

The purpose with one is to communicate with other clients that are subscribing to the same room and service, while the other is for storing private data that cannot be shared with others.

6.1 Case 1: The chat service

Case one is a chat service. A chat service is a service where multiple clients can connect to same virtual "room" and communicate with each other. Only the clients in the room can read and write messages in the room for other clients to read.

The clients can either subscribe into an existing room, or a new room that the service creates for the users. There are 5 different kind of functionalities that are available for clients. The beginning of each interaction from client to service, tells the service what the client wants to do. In this case, there are 5 options:

- 1: Adding a new client to a new room.
- 2: Adding a new client to an existing room.
- 3: Get a list of all messages that has been sent and stored in the room
- 5: Send message to room
- 6: Unsubscribe from service

Each message contains a colon (:) for separating each section of the message for decoding in the service. A sequence of how two clients can subscribe to the chat service is illustrated in figure 6.1.

In figure 6.1, client one starts off with message "1:0", the number 1 in that message represents the intention of the client. 0 is the user ID. In this

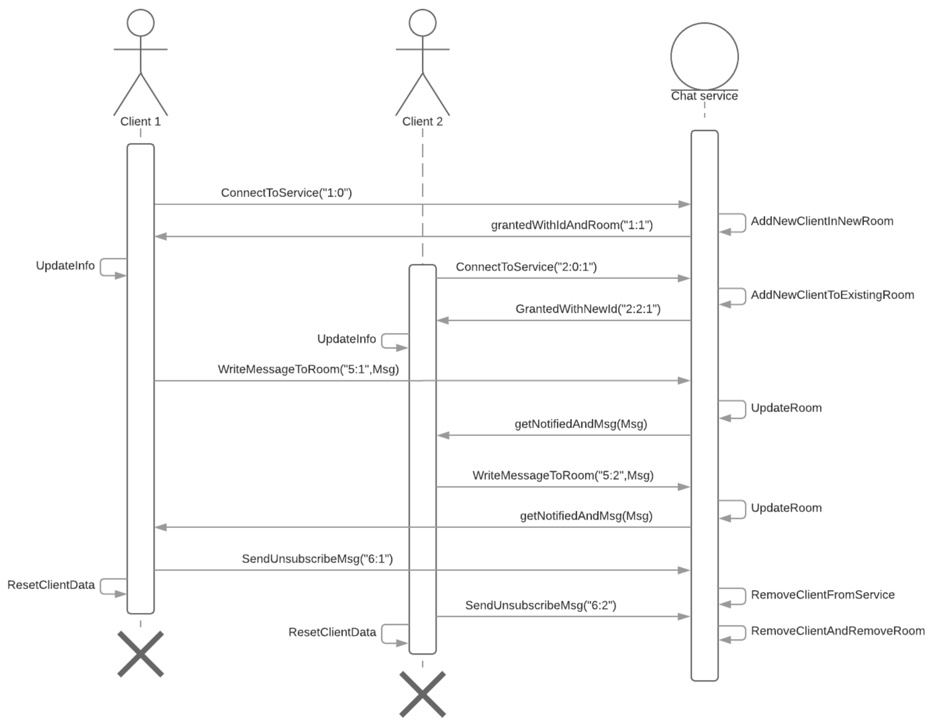

Client 1
Client 2
Chat service
ConnectToService("1:0")
grantedWithIdAndRoom("1:1")
AddNewClientInNewRoom
UpdateInfo
ConnectToService("2:0:1")
AddNewClientToExistingRoom
GrantedWithNewId("2:2:1")
UpdateInfo
WriteMessageToRoom("5:1",Msg)
UpdateRoom
getNotifiedAndMsg(Msg)
WriteMessageToRoom("5:2",Msg)
UpdateRoom
getNotifiedAndMsg(Msg)
SendUnsubscribeMsg("6:1")
ResetClientData
RemoveClientFromService
SendUnsubscribeMsg("6:2")
ResetClientData
RemoveClientAndRemoveRoom

Figure 6.1: A sequence diagram of two clients subscribing to a chat service

implementation, the clients gets their ID from the the service when they connect for the first time.

When the service receives this message, it creates a new room and adds the client to that room. The room also gets an ID. All this information gets sent back to ID. The message "1:1", the first number 1 represents the new client ID, while the second 1, represents the room ID which the client is subscribed to.

When client 2 wants to subscribe, it sends the message "2:0:1", 0 represents the client ID and the last number, 1, represents the room ID that the client wants to subscribe to.

When the client 2 receives a message back from service, for successfully subscribing to an existing room, the message says, "2:2:1", where first number 2 represents that the client has been subscribed to an existing room, the second number 2 represents the new client ID and the last number 1, represents the room number.

Then comes the more interesting part of the service, where a message gets sent. When sending a message, the structure of the message is "5:u", where *u* is the user id. When client 1 wants to send a message to service, the message looks like "5:1", and the message gets sent right after. Since it is TCP protocol, it means that the data sent will be received in the same order. When the message is received in the room and the room is updated, the message gets sent to all the other clients in the same room.

When a user who joins the group later, long after the first two clients. It can ask for all the messages that has been sent. By just writing 3 and send, the message to service will be then "3:3", where the second 3 is the client ID. Then the service sends a list of messages which is printed out for client with ID 3.

When the client does not want to be in the room longer, they can simply unsubscribe by sending the number 6. The service receives it as "6:userID". For example "6:2", the service will find the room the client is subscribed to, then removes the client. Then checks whether there are any clients still subscribed. If there are none other clients left, then the service removes the room. No further actions are taken if there are clients still left.

6.2 Case 2: The Storage service

The second case for this research is a storage service. Storage service providers lets their users store their private data on providers server. It is like having another storage device with the user, but it is on the cloud, so there is no need to carry around an extra physical device.

Cloud storage is expensive, which is why all storage service providers only lets users have limited with space of storage. For example only 2 GB with DropBox and Google Drive gives their free users 15 GB. For more space, there is a subscription model for it. Some services offer unlimited storage for higher price.

The service in this implementation is not a complete implementation as it does not store the actual data. It does pretends like it does, it prints that file has been received or sent to or from the client and the server. Since in this research, only static analysis of the system will be checked and compared, the runtime effect of it does not matter. This applies to all implementation for this case.

The different options that the service offers to client is similar in most areas. But there are some changes. Like in case 2, there are no options to add client to an existing room, because it is not allowed. It is meant as every client gets their private area. But an option that is new is getting a single file from service. This is not in case 1, because in chat service, there is no point in returning a single message. And of course, it is not add client to a new room, it is add client to a new area, get all filenames in clients private area, add file to new area and unsubscribe from area.

Since only one client is allowed in each area, once a client unsubscribes from an area, the area and all clients data is deleted as well, next time the client wants to subscribe to the service, a completely new room will be created for them, with none of their old data available.

An example of sequence between two clients subscribing and using the service is illustrated in figure 6.2.

6.3 Summary

The cases used for the empirical case study is two different cases. Each case has client-server architecture where the service is running on a server.

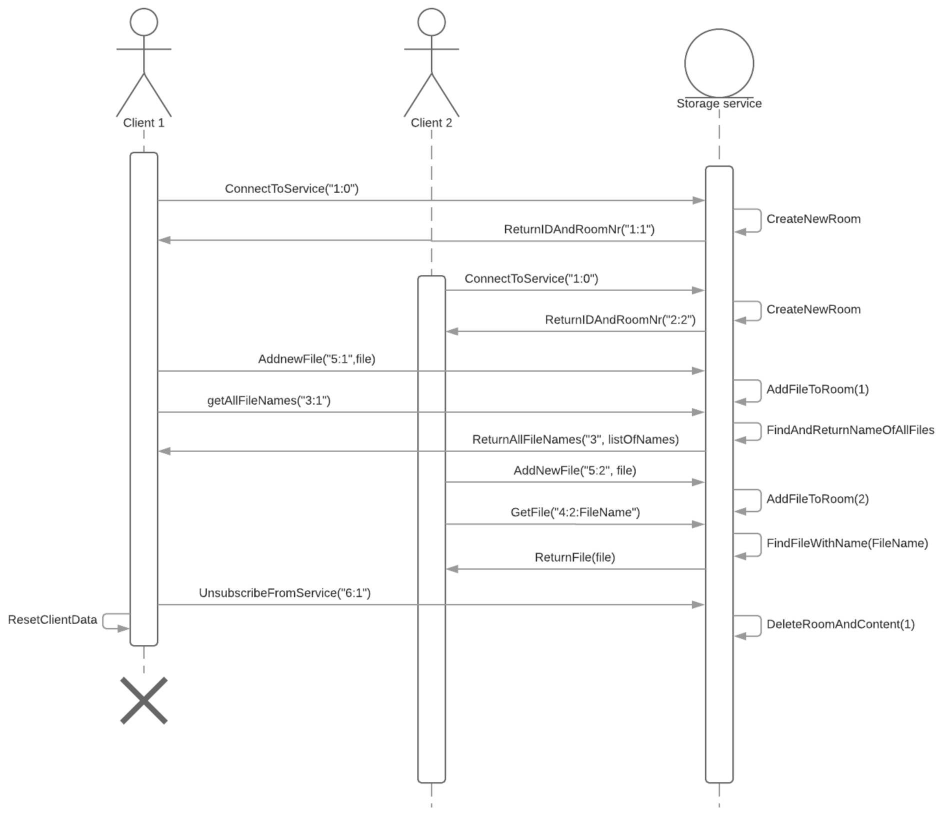


Figure 6.2: A sequence diagram of two clients subscribing to the storage service and using it

Case 1 is a chat service that has the possibility to add clients into same room and let them communicate using the service. Whenever clients are has sent a message, every one else in the area gets notification with the message. When new clients wants to connect, they can either start a new room, or join an already existing room.

Case 2 is a storage service where one client can add objects to their private area. They can add and remove data from their area. Each area is reserved for one client.

Chapter 7

Implementation

In this chapter, each design pattern gets explained how they got implemented. The structure of all three design patterns is quite similar, although there are some changes and these changes can be crucial in some situations. The major changes,however, are in the code.

The full code and implementation is accessible online at my GitHub account: https://github.com/ArselanSultani/MasterService

Some of the classes that are common to all implementations will be explained first. Proceeding that the difference between the implementations will then be explained.

From figures 7.1, 7.2 and 7.3, it is perhaps easier to see where the classes that are about to be explained is in the system.

- **FromService:** It is an inner class for clients. What this does, is when something is sent from service, it is handled in this class. This class extends thread, which means that the process in this class is running on another thread. This is just for receiving from service. While the main thread is running from ChatClient or StorageClient, for sending to service and handling user input for instructions.

- **DataObject:** This is the object that is being sent over the network. This class must be available for both service and client classes. Thus, having a copy on both sides is a must. The files that are stored, must have some information about each of them. Like unique ID, the object and time of last time it was access. If it has been never accessed, then it is set to the time it was created. The object is of type Object, which means, it can be of type String, int, boolean whatever type that is a subclass of Object.
  One thing to consider about this object is that it is not a primitive object, but each of its variables are. Which means that it is these primitive variables that are sent over the network separately because they are available on every machine. So when a DataObject is sent, the order of variable sent and received must be in the same. The order of the variable sent and received are the ID, then the variable of type Object and the last one is the time of last access time.

- **ServerClient:** This object is on service side. Each client that is connected to the service gets saved as this object. It only stores the essential information, like client ID and clients IP address. This is important for finding clients in the service. Like for example in case 2, in Storage service, when a client is disconnected, and returns for access to their area. With this class it can be checked whether the client exists or not, if it does, then retrieving the object's ID for finding the right area.

- **ChatRoom:** This is the room object for case 1. Each client gets to subscribe to only one room. But each room can be subscribed by multiple clients. The client and ChatRoom has a 1-to-many relationship, just like an Observer pattern. Each room has an ID and a lock. Which means that there are some functionalities where only one client can do at a time. Basically, if it changes some data within the room, it has to be one client at a time. It can be sending a message, adding a client or removing client.

  This object is the main difference between case 1 and case 2. This is the core functionality and the object that separates each object from each other. This object implements the ServiceObject interface. The reason for that is, whenever this has to be sent to an object that is inside the Service interface or abstract class, the type must be given. ChatRoom object is only visible from within ChatService and not outside of it.

- **StorageArea:** This object also implements ServiceObject for the same reason as ChatRoom. Their superclass or the interface that StorageService implements, must know the type. This is the area where each client that subscribes to this service, gets and can store their data in. Only one StorageArea for each client like in case 1, but unlike case 1, only one client per StorageArea, which is a 1-to-1 relationship between StorageArea and Client. This service is for storing private data and not sharing their data with other clients. The variable in DataObject about the time of last time it was accessed is used here. The object, in this case where Object is supposed to of type File, that is last accessed, will be first on the list.

These classes have common purposes and goals in all implementation. They cannot be changed, as they are not big objects or classes. They are used as a tool. I wanted this to be fair. Of course, I could change them and make one implementation much better than other, but the point of this research is to find the main differences between each implementations. All of them could have been implemented really bad or really good, and that would make it uneven, it would not be objective. If their implementations are similar, then the real difference can be calculated and give us a better result.

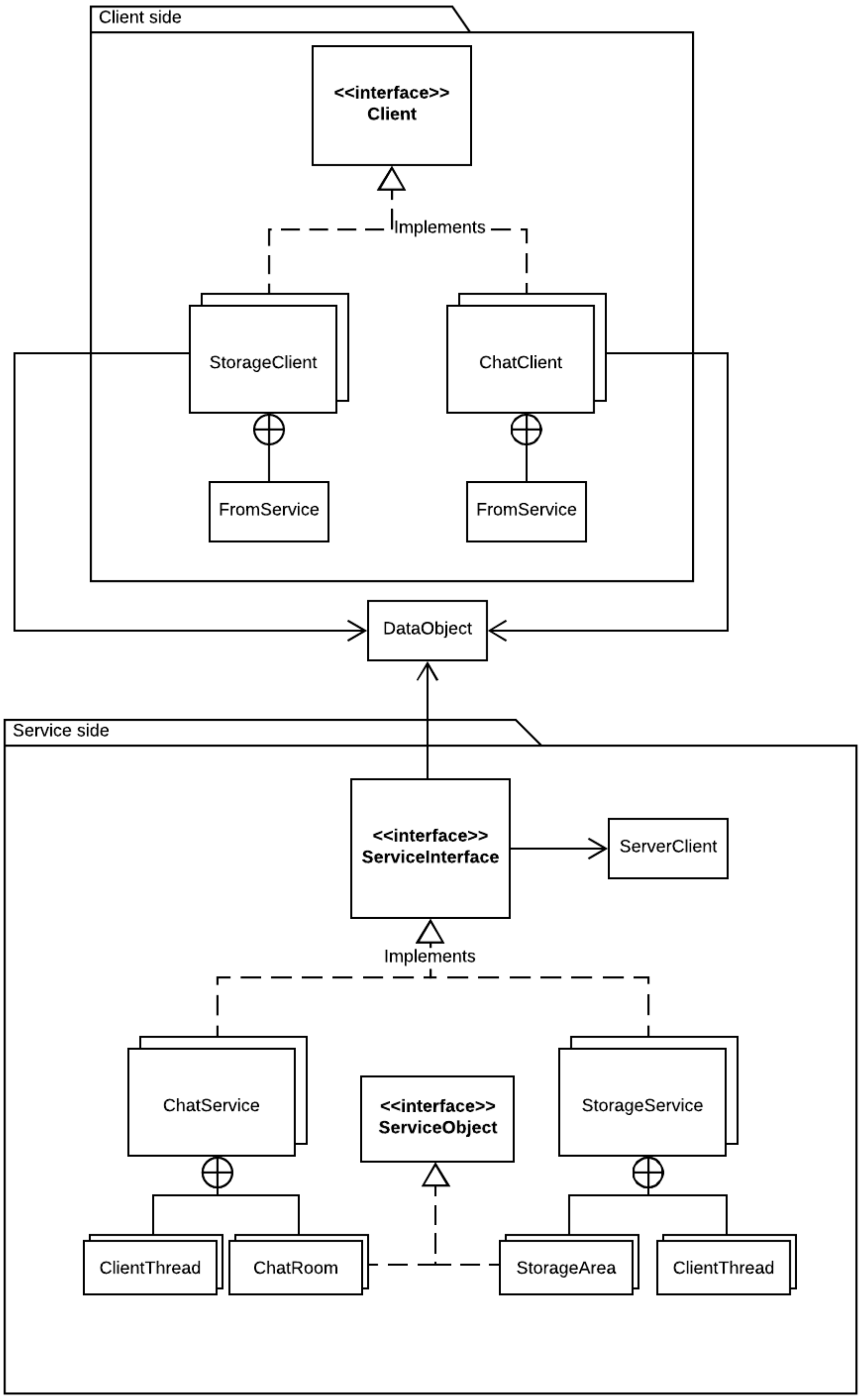


Figure 7.1: The diagram of basic interface implementation

7.1 Interface implementation

The first implementation is the one without any unique design pattern. This implementation is for measuring and finding whether design patterns actually helps the system or not when it comes to maintainability and complexity. Using the metrics explained in chapter 4, to see whether design patterns that are chosen, are better to use or not could give an interesting answer. Also, it would be interesting to see how good an implementation with an interface is compared to abstract classes.

Client side

On the client's side, there is an interface and the user for each service inherits. It contains the methods that are vital in all implementation of a client object. The startClient method, which starts the sockets, connecting to service and handling sending messages and commands to service that both services have in common such as getting the list of all object names that are in the server (messages for ChatService and name of files for StorageService), removing the client from the service, adding objects and adding client.

Since it is an interface, each object class that inherits it must implement all of the implementation themselves, even if it is identical. This is good for some reasons but bad for others. Good for adding flexibility, so each client can be implemented differently. Bad because if each client handles some functionalities in the same identical way, there will be a lot of code duplication and an unnecessary large section of code that could have been reduced.

The difference between ChatClient and StorageClient is what their service offers. ChatClient cannot send an instruction to remove a message or get a single message. It can receive all the messages in the room, but not a particular one. StorageClient cannot let a client join an existing room unless they are the owner of the room. It is supposed to store their file on their local drive while ChatClient can only read.

Service side

All implementations are implemented on each of the services, the ChatService and StorageService. Both have some of the same variables, like a HashMap, where client's ID is the key and the value is the ServerClient object. The other HashMap is for finding the room or area for the client. Key is the client and value is room object. This is also where each client and room/area gets their ID. It is the best place because it is guaranteed that the ID is unique.

Whenever a client connects to the service, independent of the service, whether it is ChatService or StorageService, the first "handshake" happens in method startServer. It prints that a connection has been made along with the clients IP address. A thread gets created for that client so it can handle the instructions from the client.

The name of the thread class is ClientThread. Its only job is to receive and send data to and from the client. Each client gets its own thread. Which might not be the best solution if the service has a large number of clients, but for us and this purpose, it works. The difference between

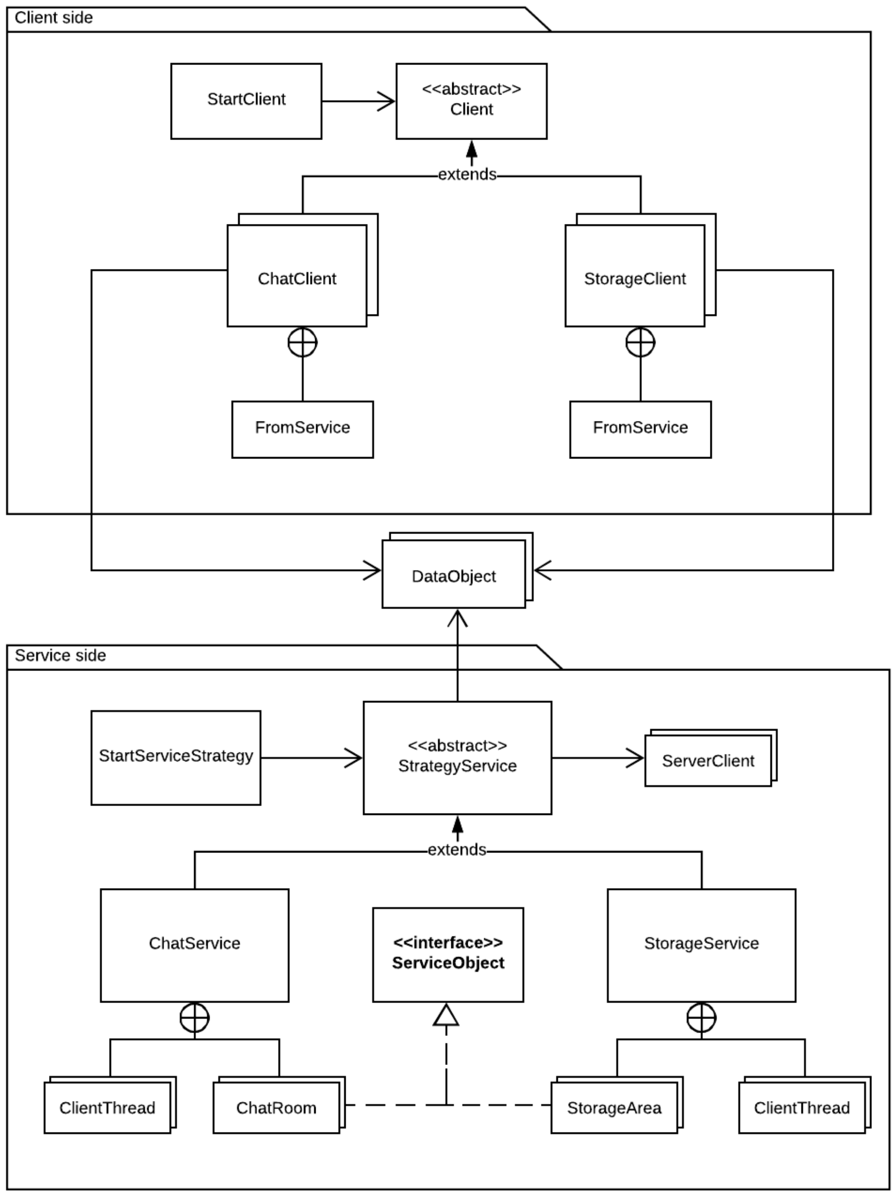


Figure 7.2: The diagram of Strategy + Singleton implementation

a ClientThread from StorageService and ChatService is the handling of different functionalities, described in chapter 6.

7.2 Strategy + Singleton implementation

This implementation is implemented using Strategy pattern along with Singleton pattern. Template pattern is used by allowing client to choose the service they want to use at runtime. Singleton is chosen because for making sure that only one instance of ChatService and one StorageService are created.

This implementation has an abstract class as parent class, instead of interface which is described in the implementation in previous section. Abstract class has some of the methods that is common for both cases implemented, so there is no need to be implemented twice. Both on client side and service side.

Client side

Unlike the other two implementation, an instance of ChatClient and StorageClient gets created on StartClient. On client side, only StartClient can be compiled and executed. When first starting, the client gets an choice, whether they want to use the ChatService or StorageService. In other implementation, this is not possible, they can either start ChatClient or StorageClient, but never change in between.

After the client has chosen which service they want to use, connection is made and text gets sent to service about the decision. If the client wants to use the ChatService, the message "chat" gets sent, if StorageService, then "storage" is the message that gets sent. Then the rest of program follows the same steps as the first implementation, receiving instruction from client, turn it into a message for the service to understand and execute the right task.

As mentioned above, this implementation makes use of abstract class, so it can implement some of the common methods that are available on both cases. Abstract is great for reduction of duplication of code. This is because some of the implementation for both cases are identical, it can be implemented in their parent class and be used from there. The common methods on client side are:

- Adding object to area
- Getting the list of object names in area. For case 1, the chat service, it would be receiving all the messages that has been sent in the room. For case two, it is the name of all the files that are stored in the area.
- Removing the client itself from the service.
- Printing the list of names or messages that are received.

Service side

In this implementation, StartServiceStrategy starts to run and then waits for connection with a client socket. When a client socket is connected, it receives the message "chat" or "storage", and depending on the message, it sends the client to that chosen service by sending the socket, input- and output stream.

In the children classes (ChatService and StorageService), is where the client gets redirected to from StartServiceStrategy, the service starts a new thread for the client to send and receive instruction from. In the first implementation, the method receives socket, creates input- and output streams and then starts a thread from each client. While in this implementation, only starting the thread is necessary, as the other steps (such as creating sockets, input- and output stream) is done in StartServiceStrategy class.

For service side, the functionality that is identical for both cases and implemented in the parent class StrategyService are:

- Adding message to room for case 1 and adding file to clients area for case 2.
- Getting the list of message in the room for case 1 and list of filenames in the client area for case 2.
- Remove client for the room or area that they are subscribed to.

Since StorageService and ChatService has different criteria for how many clients that can be in same room or area, the implementation for adding client is separate for each service. The different criteria being that many clients can subscribe to one chat room, but only one client is allowed on each storage area.

7.3 Template implementation

This implementation with Template pattern is also using an abstract class as a parent class. Abstract class is a must, because at least one of the methods in then parent class must be implemented, the template method. The method describes the steps, which calling each of the methods that has to be implemented in the children classes.

This implementation does not allow the client to be able to change the type of service on runtime. The service starts first, then depending on the service type, the client of same service must be started. ChatClient for ChatService and StorageClient for StorageService.

Client side

Independent of client type, the first event is that the template method gets invoked. In template method, each method gets invoked in a sequential matter. The purpose of this pattern is exactly that, both clients follow the same steps, but the implementation for each is different.

The path of the template method is:

1. **Open connection**. Opening sockets, input- and output streams. Also making sure that a boolean value for telling the system that the client is still using the service.
2. **Create and start client thread.** The client thread gets created with necessary information getting sent to thread class.
3. **Get instruction from client.** The system accepts the instruction from client through input from console.

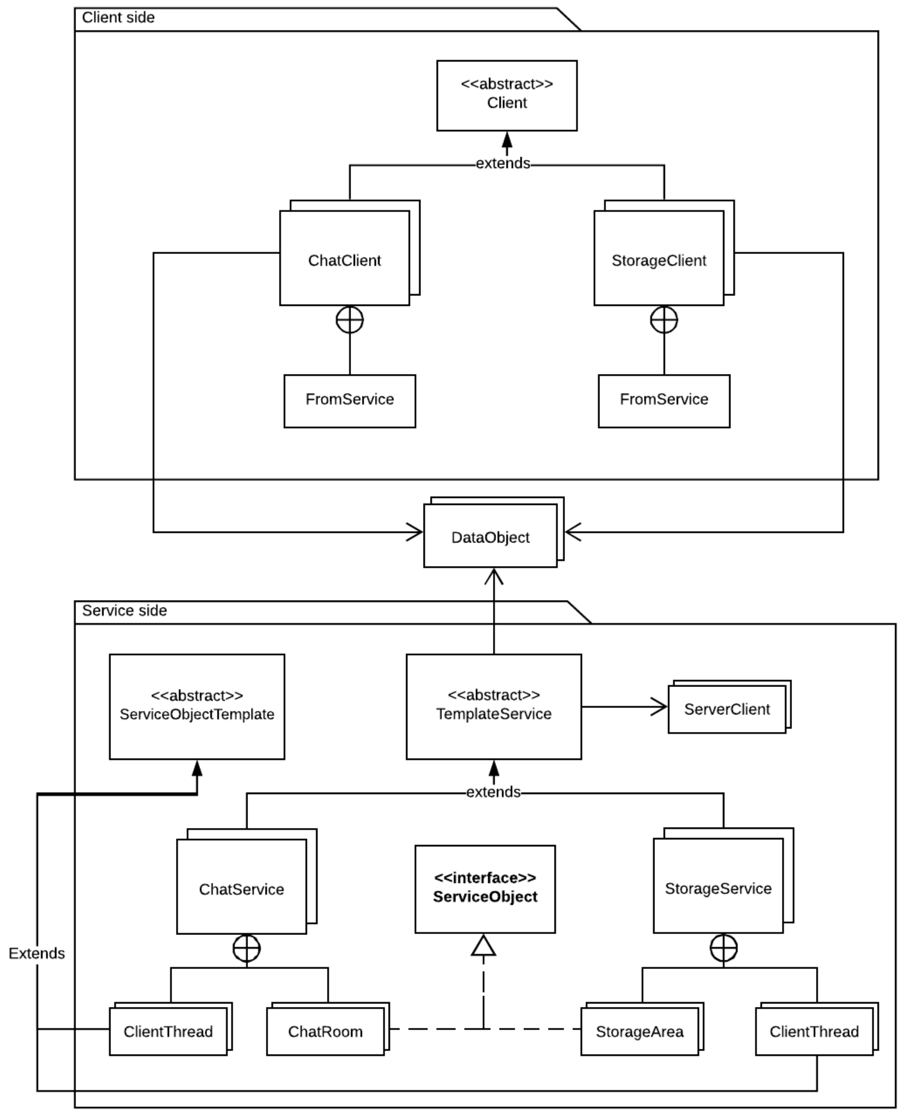


Figure 7.3: The diagram of Template implementation

4. **Create a full message.** All instructions for the service to understand starts the same way, with instruction first, followed by a colon and the client id. or example "2:3". But some instructions needs more information that. Such as removing a file from area in case 3, requires the name of the file that the client wants to remove. In the example of removing file from area, the file name then follows the already created message, like "2:7:fileName". The extra part gets created in this method.

5. **Sending the message.** After sending the instruction to service, with instruction number, client id and eventually other necessary information, sometimes something extra is needed to send. Such as when sending a message to chat room or a file to area, the file and the message gets encapsulated into DataObject. Then the object of type DataObject gets sent after the instruction.

6. **Close connection.** When the client wants to end the system, it needs to enter the number ′6′ only, because it is the instruction number. "Exit" would be better, but for this purpose, the number ′6′ is used for the client to disconnect from service. When the client gives the instruction, the boolean value from step one switches to false and this method is invoked. In this method, the sockets, console, input- and output streams gets closed.

The steps 2,3 and 4 is inside a while-loop that is loops as long as the the boolean value in step 1 is *true*, whenever the clients wants to disconnect, and that boolean value switches to false, it jumps out of the while-loop and goes to step 6.

In the other implementations, one method handles opening the connection, creating and starting thread, getting instruction from client and closing the connection is done in one and same method. Creating and sending message happens in another methods but gets invoked from this method.

Service side

The template method on the service side could not be on the service, rather it handles the steps on communication between the client and service in ClientThread class. The reason it could not be done on the service, is because it would be only two steps, which would receiving the client and starting a thread for handling sending and receiving to and from the client. The Template pattern with only two steps is not enough for it to have an impact on the system.

There are some alteration in communication logic compared to the first and second implementation. Because in this implementation, whenever a client sends an instruction that the service must execute, the service must then respond when it is done. Something that was not necessary in previous implementations. An example is when a message is sent, or an object is added to storage area, the service did not respond back to the sender in any way, but in this implementation, there has to be sent something, even if the client does not do anything with it.

The template method is in ServiceObjectTemplate class, that can be seen in figure 7.3. The ClientThread for both ChatService and StorageService is extending the class, where the abstract methods are being implemented.

The methods that are called in the template method in ServiceObject-Template are and in this order:

1. **Open stream**. While the startServer in ChatService and StorageService is the place the client's socket is created, the input- and output stream is created here. Along with the boolean variable for keeping the socket, input- and output stream alive.

2. **Get instruction**. Waiting for receiving instruction from client on what to do.

3. **Decode and execute command**. After having received an instruction, in this method it gets decoded, like what does the client want to do. Does the client want a list of with the name of objects? Be added to the service? Add an object? Whatever the client wants to do, it gets done from here. If the client wants to add a DataObject, it creates an empty DataObject object and calls its read method for receiving the variables.

4. **Create reply message.** This is where the client gets the reply from service. This can vary from case 1 to case 2. Case 2 can send a single DataObject back to client, while it cannot be done in case 1. There is no point in sending a single message in case 1 from the past, while it is more obvious that the client can receive a file that is stored in their private area.

   Because of what each case can do, each of of this methods implementation is different. If the client does not receive anything valuable information, like just replying just for having to reply, it just sends the instruction that was sent.

5. **Reply client**. The message that got created in previous method, is sent to client in this method. Just like on client side, if something that just a message is sent back, like a list of objects or an DataObject. It sends first the number '3' for sending a list of object names, and the list is sent after. The client can handle it in this manner.

6. **Close stream**. When the client has sent the message "6:C" where *C* is the client ID. Then it means the client is done with the service and wants to disconnect. The socket, input- and output streams are closed.

Like in client side, some the the steps are inside a loop which keeps looping as long as there is connection to the client or until the client wants to disconnect. The steps that is inside the while loop is 2, 3, 4 and 5. Having step 1 and 6 inside the while-loop, would mean that each client would connect, send instruction, receive feedback and disconnect, which does not seem practical.

7.4 Mediator implementation

The Mediator pattern is only used in case 1. The clients are colleagues that are communicating with each other through this Service, which works as a mediator.

Each client does not have to know about other clients and how many other clients are there to communicate with. Since this is a distributed system, the objects are not closely coupled with the mediator, it is connected to the mediator.

The connection is done with socket and using ObjectInputStream and ObjectOutputStreams for sending receiving and sending messages to the mediator.

Because of the mediator, the number of connections are much lower if the number of clients increases. For example, taking 5 clients, there have to be 10 connections in total, because each client must be connected to the next. An illustration of this example can be seen in figure 7.4.

When a new client is assigned to a new room on the service, other clients must know the room number if they want to join the room. But each client does not know how many clients have joined the room. There are no restrictions on how many clients there can be in the room, and quite frankly the client does not care about it either.

It changes from many-to-many relation, where all clients must know of each other, to 1-to-many relation, where one mediator is enough for connecting them all together.

The colleagues or Clients have the possibility to stop the connection to the mediator and then it will be gone and whenever it starts again, a new object is created.

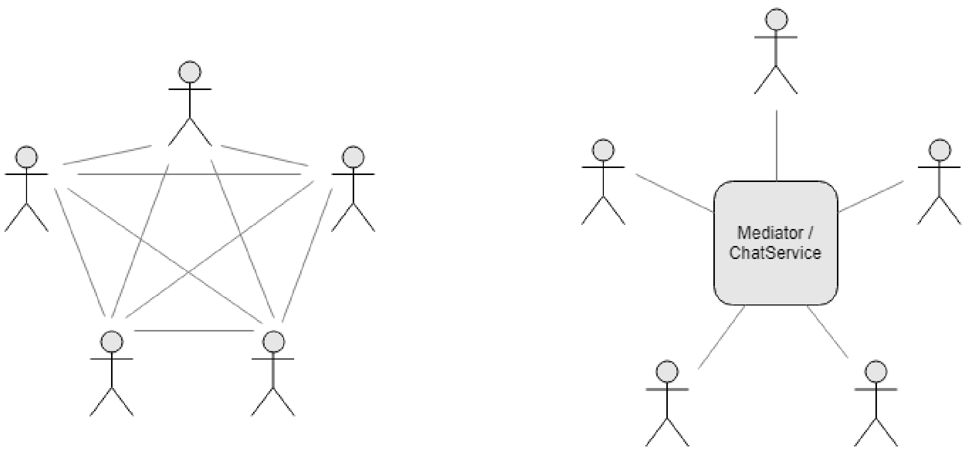


Figure 7.4: An example of Mediator pattern in this implementation

7.5 Observer pattern

This is also a design pattern that only fits the goal and purpose of case 1 and also focuses on reducing many-to-many relation to 1-to-many relation for reducing complexity and increasing maintainability, since most of the logic complexity happening in one place.

In this implementation, the Clients are the subjects and the ChatRoom object is the observer. Because whenever a subject is updated, the observer that is subscribed to an observer gets notified. Just like in this implementation where clients get a notification whenever a change is made in the chat room.

After client has found out that the ChatRoom is updated and received the update from service, it gets sent to a method that handles it with just printing it. A functionality that can be extended is which user sent the message to the service.

Sockets are used for making a connection between the client and service. Whenever a new client is connected to the service, they can join a new room or already existing one that other clients have made. Then they get added to a room and is ready to receive an update as long as the client is in that room.

When the client does not want to use the service or the connection disconnects, the clients get removed from the chat room. If it is empty and no more clients are left, the whole ChatRoom object gets removed.

7.6 Cloud setup

The cloud setup for this system is configured by using TCP protocol and sockets. A socket on client side and one on service side. The service side needs to know the port number where the service can expect to receive connection on. On service side, there is no need to specify the IP address, the system retrieves it automatically.

On the client side however, the client must specify the IP address to where the service it wants to connect is, as well as specifying the port number which the service is open for connection on.

There were multiple options for setting up the network for this system. Using other protocols than Transport Control Protocol, like Remote Method Invocation (RMI) or User Datagram Protocol (UDP). UDP is not a secure protocol, which means that it does not guarantee that the data sent is received as well. And if the is received, the protocol does not guarantee that it is received in the same order that it is sent. Neither of these two disadvantages are trivial for either of the two cases. Not receiving the messages in the same order is a big problem for the chat service, or if the data is to big and it is sent in pieces, and some pieces are missing, it will make the file unusable because parts of the file will be missing.

There are other options that exists that are much better options that RMI, like SocketChannel or AsynchronousSocketChannel. However, the purpose of the this experiment is to use to use a existing library. Not use extra tools that may cause the complexity of the system to increase unnecessary.

Each service uses a distribution libraries that fits their needs. But Since I wanted to make a general system and therefore decided to go for TCP, because it is the most general protocol that has been used for a long time and it has worked great for all this time.

7.7 Summary

Both cases and all implementation have some classes that are common for them all. They all serve the same purposes for all implementations and objects. FromService is a class which starts a separate thread that is just for receiving from service for example. ServerClient is a class on the service side for storing information about the clients ID and IP address. ChatRoom is for all implementations in case 1, is the object where each client can subscribe to and receive updates whenever other clients sends a message to the room. StorageArea is a private area for each client who is connected to the service from the case 2 and the last object is DataObject, which is the general data that can store a message or a file. But that message or file is encapsulated into this DataObject.

The first implementation is using an interface as parent class for declaring some of the common functionalities, but where each service type and client type of the said service, must inherit them and implement them separately. There are no specific design pattern that are used for implementing this implementation.

The second implementation is using an abstract class as the parent class for implementing the common functionalities for all of its children classes, like ChatClient and StorageClient on client side and StorageService and ChatService for service side. The Strategy pattern has been used so the client can choose which service it wants to use, based on that choice the client can then connect to the service and the service sends the client to the right type of service. The user can change the type of service on runtime, which is not possible in other implementations. Singleton pattern is also used in second implementation, where there can be only one instance of the StorageService and ClientService objects.

The third and final implementation is implemented with help of Template pattern. In the parent classes, there is a template method, where a task is being divided and spread over multiple other methods. A task that is one method in other implementation, is spread over multiple methods. There is a task on both client side that has been divided into six other methods. Which is opening connection with sockets and streams, start the client thread for handling receiving data from service, getting instruction from client, make that instruction into a message that the service can decode and understand, send that message along with necessary data and eventually close the connection when the client wishes to.

On service side, it is the thread class that handles communication with the service. Like when the service opens the input- and output streams, receives instruction from client, decodes and executes necessary parts of the program, creates a reply message, sends that message and eventually closes connection when the client wants to.

Chapter 8

Results

In this result chapter, the data that are gathered is being presented, explained and what information was learned from these data. Results from the implementations from both cases get compared with each of their respective implementations. The different cases will not be compared with each other, as they have different purposes.

The order of each measure will not be random, rather be based on how important they are, and whether there is enough information. Like for example, the maintenance metric is dependent on three different metrics (LOC, Halstead's Volume metric, and cyclic complexity) are calculated before the maintenance index itself can be calculated.

The calculation of the metrics on method level is done by using MetricReloaded and SonarQube. The implementation with the lowest number of methods is case 1 with the first implementation with 77 methods and the highest is for case 2 with Template pattern implementation with 109 methods. To compare each would take a long time and unnecessary as it will not give us interesting information, but the methods that are compared in each of the metrics, are methods that have received the highest score in at least one of implementations. For example, if a method is among top five highest scoring methods in one implementation, then it will be compared to the same method on the other implementations, whether it is on top five on those implementations or not.

However, MetricsReloaded only calculates for methods that have an implementation, which means methods in interfaces and abstract methods in abstract classes is not part of calculating the average value per method. Constructors, on the other hand, does count as a method.

As explained above, there are three implementations of each case. The first implementation is a basic interface on both client side and service side, without a specific design pattern implementation. Within this chapter, this implementation will be referred to as A. The second implementation is with Strategy and Singleton pattern, which will be referred to B. The last implementation with Template pattern, will be referred to as C.

	General	Strategy	Template
Total methods	77	79	103
Implemented methods	60	65	77

Table 8.1: The total number of methods and the number of methods that has implementation and is the number that is used when calculating the average score for case 1.

	General (A)	**Strategy(B)**	**Template(C)**
Total N	1466	1620	1641
Total n	540	653	610
Uniqueness in %	36,83	40,31	37,17

Table 8.2: The total number of tokens(N) and vocabulary (n) in the different implementations and how much of it is unique.

8.1 Case 1: The chat service

Before I begin to discuss the results from the case, it is better if I explain the number of methods that are used for calculation of average per method for each implementation, which can be seen on figure 8.1.

8.1.1 Halstead's Complexity measures

There are some metrics from Halstead that might not be as useful as others. Some might not give us a bigger picture of the whole system but rather just simple information, like the total number of tokens and the total number of vocabulary metrics. It might not give relevant information on its own, but it is vital for measuring the more complicated metrics. Therefore, it is shown in table 8.2.

As table 8.2 shows, the number of vocabulary (in other words, unique tokens), is not necessarily proportional to the total number of tokens. The proof is B has a lower number of total tokens than C, but a higher number of unique tokens than C. What this means is that B is less complex than the other implementations, because it has the least reuse of tokens.

The uniqueness in percentage shows how much of tokens are reused. The higher the percentage is, the less reuse it has, which makes it more complex because it introduces new tokens instead of reusing the existing ones.

Halstead's volume

The first bigger metric that is going to be explained and calculated, is Halstead's volume. It calculates the content of the program in bits. It works best in method level. However, it can give some information about the whole system in total. MetricsReloaded has been used for calculation of each method. From earlier, it is known that the formula for calculating the volume is as follows:

$$V = N \log_2 n$$

Which for each implementation on system level would give:

- **A:** 13306,61 bits of content
- **B:** 15148,52 bits of content
- **C:** 15183,62 bits of content

For this research, the five methods with highest Halstead volume metric for each implementation, according to the tool MetricsReloaded are shown, along with their score on figure 8.1. As explained above, a method that ended up on top five methods with highest volume score did not receive as a high score, but instead, the score that it did receive is used for comparing alongside them. Thus, making it more than just five methods to compare in the graph, in this case, 9 different methods.

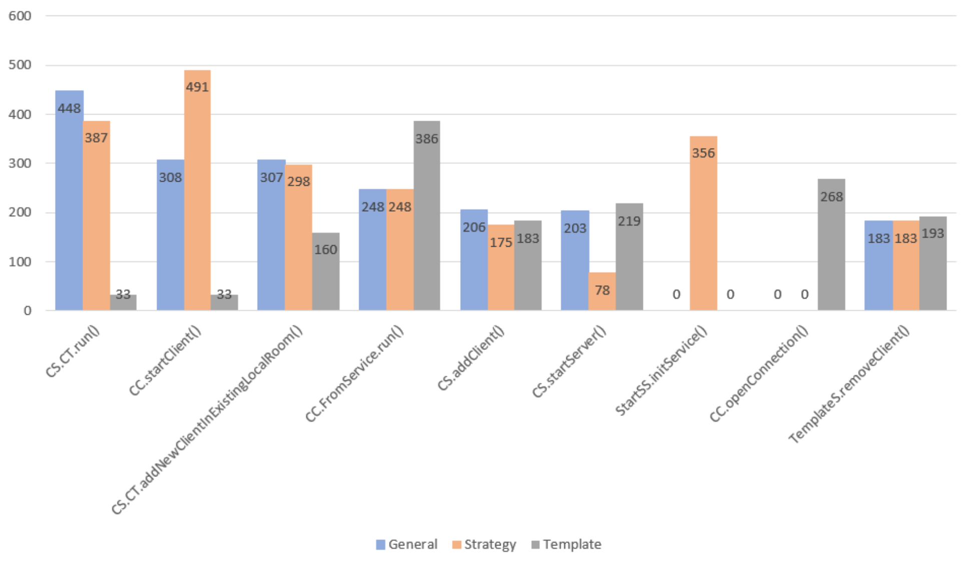


Figure 8.1: A multi chart diagram showing methods indexes on each implementation.

The graph 8.1 shows some interesting points. Since ClientService's(CS in the figure) and ClientThread's(CT in the figure) has a run method for when the client is connected to the service. Which receives a very high score for both A and B, but for C, it received a much lower score, and that is because it is in the abstract class, where the template method is. The same situation with the second method from left, which is ChatClient's(CC) startClient method. It is also part of the Template pattern, where it is in the abstract class. That is why they received such low score, because they only calls a couple of methods, while the same two methods for A and B handles connection and communication between Client and Service.

ClientThread.FromService.run method has increased volume in C because the Template patterns force the service to return a message after every instruction. Thus, making it send a message back to the sender when the sender sends a message in the chat room. In A and B, the service does not need to do that, which makes it easier for client side.

B received the lowest score by far on ChatService.startServer method, and that is because most of the implementation here is shared with StartStratService.iniService method, which starts the service by letting both ChatService clients and StorageService clients connect. If chat service clients makes a connection, the service first receives a message that says "chat", and then the service sends the client to Chatservice.startServer method, where the threads for each client gets created. Which explains StartStratService.iniService is only getting a score on B, because it is only necessary in B, thus not implemented on the other implementations. However, an important note is that the StartStratService.initService also is involved with starting service for case 2. Which is why it does receive a very high score.

ChatClient.openConnection is a method that opens all the sockets, input- and output streams, as well as a boolean value that is true as long as the client wants to stay connected to the service. It is part of Template pattern and therefore only B has this method implemented and also since it is the method with the third highest score for B, I thought it is important to see how important this method is.

In chapter 4, section 2.5, the recommended minimum and maximum score was explained. Each methods volume score should be between 20 and 1000. And the graph shows that the methods with the highest volume are way below that score.

Figure 8.1 shows only the big methods. There is no way of finding the preferable implementation from it, but it gives us a way to look at the biggest methods and how they are compared to the other implementations. The average volume per method gives us a better answer.

- A: 67,64
- B: 69,16
- C: 53,99

A has the second highest/lowest average volume per method score. But it has the fewest methods, 5 methods fewer than B. Which receives barely highest average score on the second highest number of methods. C has the lowest average score by far, but also the highest amount of methods. That is because of the Template pattern, an example is when the client thread starts in StorageService, in A and B, only one method handles receiving instruction, decoding it and call the right methods for doing what the client wants. But in C that same task is executed with 3 methods, one for each of those three steps.

The same on the client side, when sending an instruction, A and B does that is one method, which takes C 4-5 methods, like opening sockets and streams, adding the client to service, taking instructions, making a proper message for service to understand, sending the message and closing the sockets. A and B does all that in one method, which may be a big method, but still half the score the maximum score is allowed.

Implementation C receives a much lower score although it has more methods in the calculation that does reduce the average score for C.

Halstead's effort

As explained previously, Halstead's effort calculates the effort that required to understand and implement the system. The index is proportional to the volume index and difficulty index. Difficulty index is a metric that shows the difficulty level of the program implementation or how hard it is to understand the program. The methods with the highest score from this project are shown in figure 8.2.

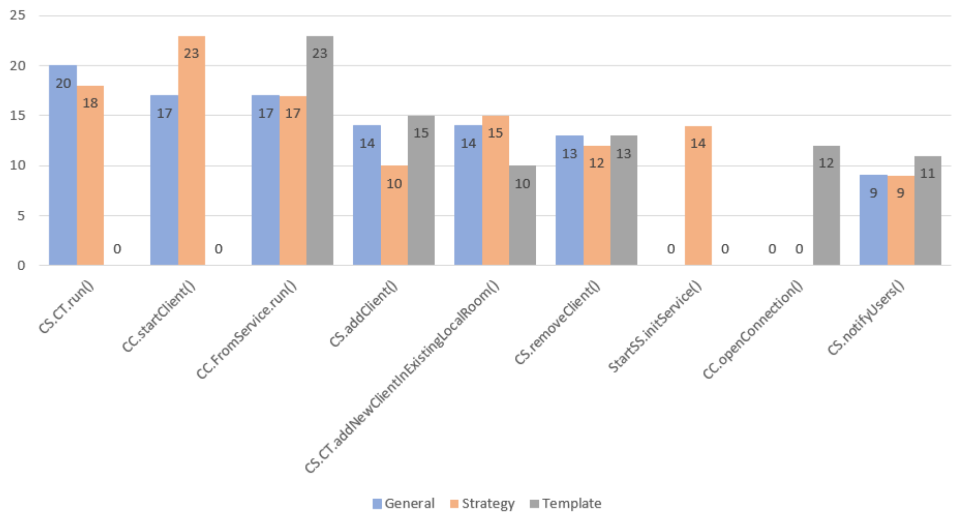


Figure 8.2: Result from Halstead difficulty measurements with five methods which has the highest scores.

For the first two methods, C get 0 points, that is because they are supposed to be the Template methods. They just call methods, which makes it easier to understand. ChatClient.FromService.run method is higher in C because of the same reason for volume, it must handle another command while A and B does not.

The reason why ChatClient.startClient (2. method from right) received a lower score for A than for B, is because when a client sends the first instruction after connecting, whether to join a new room or already existing one, is handled in a separate method. While in B, it is handled in the same method.

ChatService.startClient is lower for B, because most the job that is handled in the same method for A and C, is handled in StartStratService.initService method, like opening the socket for connection.

The threshold for this metric is 30, and as figure 8.2 shows. All metrics are below 30. It also indicates that implementation A with blue color is the preferred implementation. Overall from the figure, it can be argued that.

However, when we take a look at average difficulty per method below, it may say otherwise:

- **A:** 5,11
- **B:** 4,85
- **C:** 4,03

With the highest average score, implementation A may not be the easiest overall. C is the easiest to understand on average, it helps that the two hardest methods for A and B, received 0 points in C because they are divided into smaller methods., like for example ChatClient.openConnection is one of them, that can be seen on figure 8.2.

Figure 8.3 shows the results from the calculation of effort for top methods with the highest effort score.

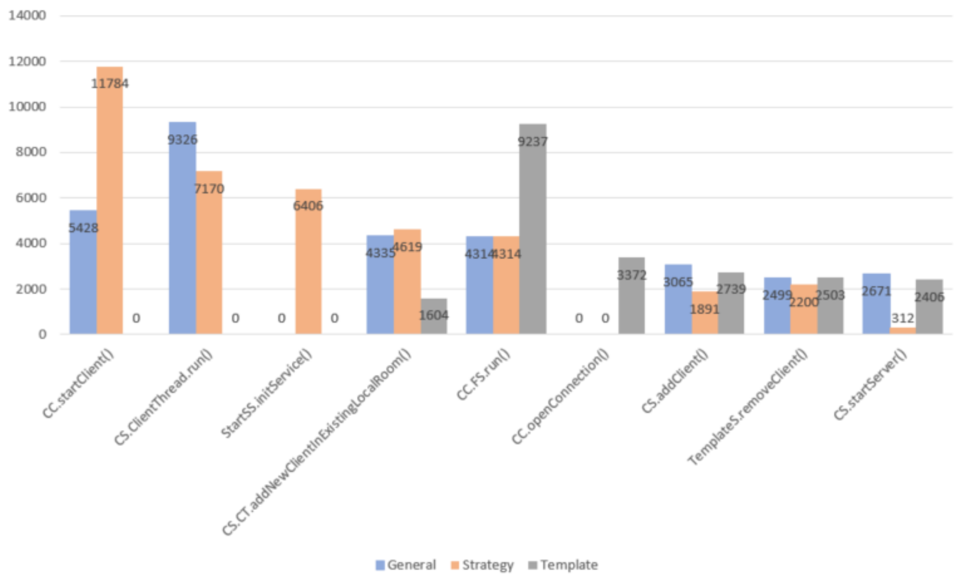


Figure 8.3: Result from Halstead effort measurements with six methods which has the highest scores.

To see whether any methods have received a higher score than the metric's threshold, the threshold must first be defined. Mccabe's website[15] has their declaration of the threshold, which is 300 for method level. However, they also have set the threshold for Halstead's volume to 1500 for method level, instead of 1000 that are set everywhere else. But for calculating effort, the formula is:

$$E = V * D$$

It would not make sense that the metric should be 300 because even if both difficulty and the volume metric is far below their threshold, it would still be far above the effort threshold score.

Therefore for this research, the threshold for effort metric should be set to the threshold of Halstead's volume metric, multiplied by the threshold for Halstead's difficulty metric. In this case, it is 30 000, which might be a high score, but this is a threshold score, which means higher than this, the methods are unreadable and should definitely be refactored. it is also the most logical way of setting the threshold. If it is above the threshold of both the difficulty and volume, then it is definitely above the threshold. However, if it is only one of the required metrics that are over the threshold score, then it is not certain that it is over the effort threshold as well, because the other metric might be lower.

With the threshold set to 30000, none of the methods are close to it, not even half of it. Only one method is higher, the ChatClient.startClient method. The reason for it being comparatively high than the same method for A is that first, it has to send a message to service for choosing the service to use and adding the client to a new room or existing room is what drives the score that high.

Handling an extra message for C is the reason ChatClient.FromService.run method (5. from right in figure 8.3) receives a much higher score, the A and B.

It is hard to choose the preferable implementation, but taking a look at average effort per method below:

- **A:** 766,10
- **B:** 830,71
- **C:** 482,94

It is clear implementation C requires the least of effort to understand, while B requires most effort out of these three implementations. Dividing big methods into smaller parts does reduce effort to understand the overall system.

8.1.2 Cyclomatic complexity and Cognitive complexity

Complexity is essential for any developer, it is for making it easier for the next person to understand the code, maintain it and add further functionalities.

The graph 8.4 shows the cyclomatic complexity score is quite similar to A and B, there are some differences here and there, but overall the score is quite similar. ChatClient.startClient and ChatService.ClientThread.run for C are the template methods. They have a while loop in them, thus having two cyclomatic complexity score. The reason ChatClient.startClient has a different score for A and B, is because of the if-else statement that adds a user to service in either new room or existing room.

Except for the first method, ChatService.ClientThread.run, for A and B, all methods are under 10, which is where it should be, between 10 and 20 should be rare, and it is. Only one method among at least 77 methods is over the limit. While for implementation C, the task is spread over multiple methods, which does reduce the complexity.

From table 8.3, C gets the highest cyclomatic complexity score in total, but the lowest average per method score. While A does receive the

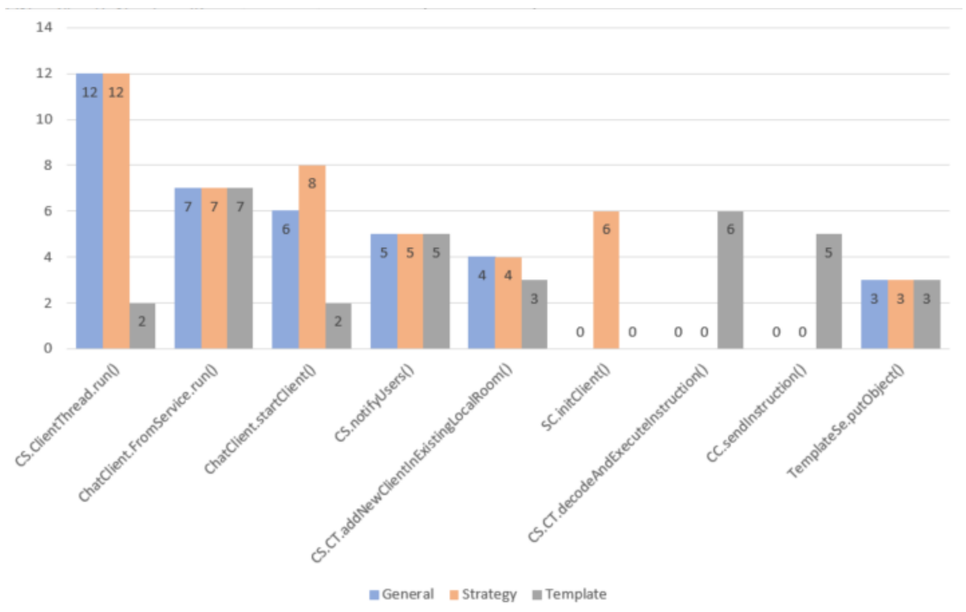


Figure 8.4: Comparison of each methods from each implementation that has the highest cyclomatic complexity score in at least one of the implementations.

	A	B	C
Average per method	1,82	1,95	1,74
Total	109	129	134

Table 8.3: The total cyclic complexity and average per method index throughout the system.

lowest cyclomatic score while coming in second place with the average per method. B receives the highest average score and mid total score.

This means that implementation A is the preferred implementation when it comes to cyclomatic complexity because it is the least complex of all. The relative margin between average per method score is much smaller than the total between A and C. What this means is that, The difference between average per method score between A and C has less significance as the total between A and C because the total score is bigger.

While cyclic complexity is great and gives useful information on method level, there might be needed a metric for class or application level for giving a complexity score for the bigger picture. It is for class or project level and it is an "updated version" of cyclic complexity with it being able to handle for example try-switch statements better and are adapted to the way development has evolved since the original maintainability index was created. The cognitive complexity score is shown in figure 8.5.

The reason the ChatService and ChatClient are included is that they are the classes that received the highest score among all the classes in their implementation. They are also the classes that are most affected by each design patterns considering that they are the classes that are changed the

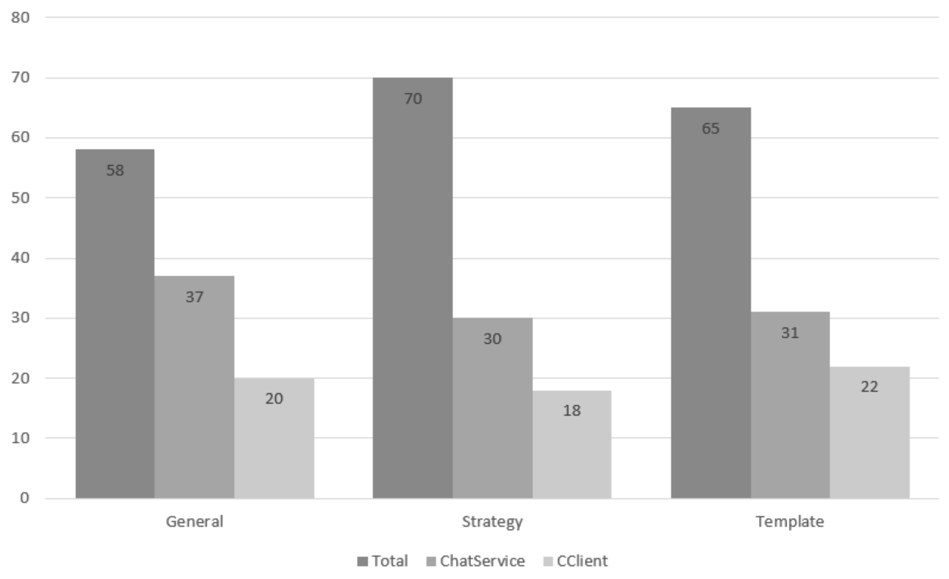


Figure 8.5: Cognitive complexity score of the implementations

most and it is only logical that they are compared in this metric. It is also useful to see how much of each implementation is covered by these two classes alone, so it can be seen how much the pattern has changed the rest of the system as well.

As seen in figure 8.5, implementation B receives the highest cognitive score in total, while A receives the lowest total score and C in between. The lower the score the better, which makes A the least complex according to the cognitive complexity score and B the most complex implementation.

The interesting part here is seeing how much the score for classes ChatService and ChatClient covers of the whole metric. For A it covers almost all of it, 57 out of 58. The reason these class received so much of the score is that, all of the services and clients implementations are done in those two classes. ChatService class does receive the highest score compared to the ChatService for other classes. The ChatService and ChatClient for B receive the lowest for both, making it the least complex compared to A and B.

While for B, the ChatService and ChatClient do not cover that much as A, it only covers 48 out of 70 and 53 out of 65 for C. These classes only implement the differences between each class. And since their superclass has implemented part of it, they also gather some of the points and not only these two classes. Since for B, it receives the highest score, while the two classes receive the lowest in total compared to other two implementations, it makes it that the rest of the program for B is more complex than the rest of the program for the other implementations.

When it comes to cyclomatic and cognitive complexity scores, the data that is gathered does point to A to be the preferred implementation. The reason why it is the preferable object according to cyclomatic complexity is explained above, and according to cognitive complexity, it does look to be

	A	**B**	**C**
Total lines of code	554	664	677

Table 8.4: Total lines of code in each implementation.

a similar situation. A does receive the lowest cognitive complexity score in total, but the ChatService class is the highest and ChatClient in the middle compared to other implementations.

The difference between the scores for ChatService from each implementation is not big. It is 7 points difference for a class that has 38 methods, which ChatService for implementation A does.

8.1.3 Lines of code

Lines of code gives information about the total number of lines that code. Comments and white-space do not count. As table 8.4 shows, implementation A has the least amount of lines of code while implementation C has the most lines of code. This metric may not be seen that important on itself, as it only gives information of total lines of code, however combining this metric with another can be used for calculating more complicated metrics like maintenance index.

8.1.4 Maintenance index

Having one metric for calculation of a systems maintenance is not simple, but combining multiple metrics can help make a single metric that can give relevant information about systems maintainability. The formula explained earlier, can do exactly that. As previously mentioned, the formula for calculating the maintenance metric is:

$$index = 171 - 5.2 * ln(avg(H)) - 23 * avg(C) - 16.2 * ln(avg(L))$$

where H is Halstead volume metric, C is cyclic complexity metric and L is lines of code. All these metrics had to be calculated before the maintainability index could be used.

avg(H) is average for every method Halstead's volume metric. MetricsReloaded has support for this and calculates for the user. $avg(C)$ is average for every method cyclomatic complexity metric. The same procedure goes here, but MetricsReloaded does offer to calculate this for their user and it has been double checked and proofed to be correct. avg(L) is calculated manually. Both SonarQube and MetricReloaded calculated the Lines of code metric, and they give the same answer, but only SonarQube gives their users information about the size of their system, like the number of methods in their system. By dividing lines of code metric with the number of methods, gives us information about lines of code per method. By placing them in the formula, the maintainability index gets calculated and it is shown in table 8.5.

The maintenance of these systems, as shown in table 8.5 gets a good score for all implementations. A high score would imply that the system

	A	**B**	textbf C
Avg(H)	67,64	69,16	53,99
Avg(C)	1,82	1,95	1,74
Avg(L)	7,19	8,41	6,57
index	**75,26**	**69,63**	**79,73**

Table 8.5: Maintainability index for each implementation along with the other metrics.

	A	**B**	**C**
MAX(0, I)	75,26	69,63	79,73
Microsoft's index	**44,01**	**40,72**	**46,63**

Table 8.6: Microsoft's maintainability index for each implementation.

is easily maintainable. But for this system, the score for this metric is at a decent level, definitely room for improvement.But it is hard to tell whether it is closer minimal score or maximum when the minimum score is unknown.

Microsoft's engineers are no happy with the maintainability index that already exists, they decided that a metric that gives us more information is needed, one that at least tells us the maximum score and as well as the minimum score. While the maintainability index above starts from 171 and goes to an unbound negative number, it does not give useful information if the index is -1000 or -13142, although there are big differences in numbers. They wanted a metric that goes for 0 to 100 and the higher the index is, the easier it is to maintain it. The formula is:

$$index = MAX(0, I) * \frac{100}{171}$$

where I is the maintainability index calculated above.

In this system, with these three implementations, the result of Microsoft's maintainability index is shown in table 8.6. Although the results from table 8.5 and 8.6 are showing the similar score, the only difference is, the Microsoft's metric actually has a minimum and maximum score, which tells us far more.

From these the two tables, 8.5 and 8.6, it is evident that C is the best maintainable implementation for this case. It has the highest score. A does get a decent score right behind C, but B is the worst maintainable system.

8.1.5 Class coupling

Class coupling is an important metric, as it tells us how sensitive the program is to changes because a part of the program is dependent on another part of the program and the way it should work. When the second part of the program is changed, the first part will not work as it should.

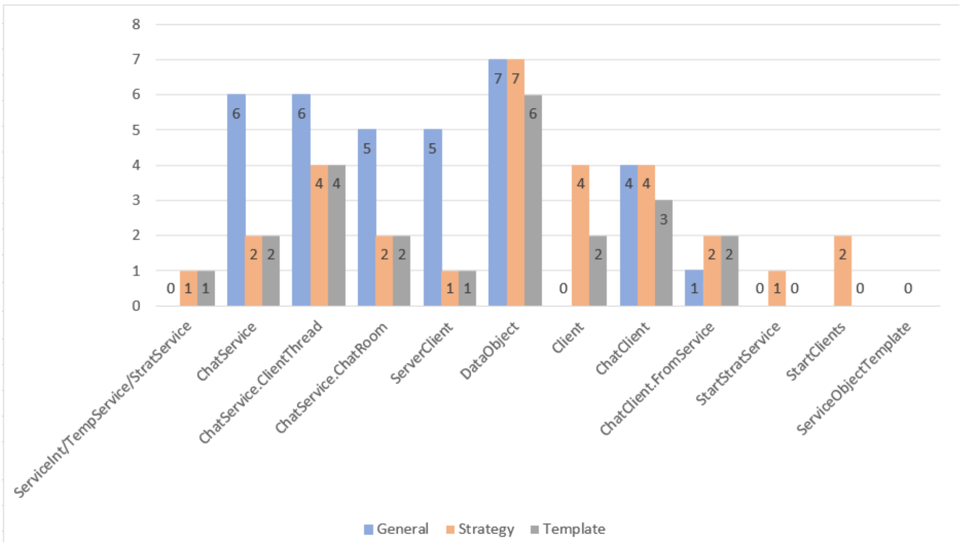


Figure 8.6: Class coupling for the different implementations

In this metric, the lower score it is the better because it just tells us that the classes are not that much dependent on each other is more robust and less sensible toward changes.

From figure 8.6, it can be seen that the classes from A have the highest coupling compared to classes from B and C. It is much closer between implementations B and C, where on multiple occasions, they have equal score, but since C does receive lower on more occasions than B, it makes C the better implementation among these three. It is the one that has the lowest coupling between the class, thus making it less prone to faults after changes.

On average, A has 3,40 coupling in average per class, B has 2,50 and C has the lowest score of 2,09 coupling per class. It is spread over 10 classes for A, 12 classes for B and 11 classes for C. ServiceObject is an interface on all implementations, it received 0 coupling in all implementations, therefore, I decided to remove from graph, because it looked like a bug caused the software that is used for making the graphs, to remove them, which is not the case.

Since the ChatService is implementing an interface, all implementation in A has coupled with a class with implementation, which is the ChatService. The proof is Client and ServiceInterface from figure 8.6. It has 0 coupling, while for B and C, they have because they have some of the implementations.

A fair way to look at it would perhaps be if combining the Client and ChatClient for B and C and compare it to ChatClient in A. Then we would see that for B the coupling would be 8 and 6 coupling for C. Which would make B very complex on the client side at least. Doing the same for server-side would add adding ChatService with TemplateService and StrategyStratService. For B and C, it would be 3. Still making A the most complex on Service side.

	General	Strategy	Template
Total methods	86	86	109
Implemented methods	69	73	83

Table 8.7: The total number of methods and the number of methods that has implementation and is used when calculating the average score for case 2.

	Composite (A)	**Flyweight (B)**	**Template(C)**
Total N	1686	1673	1759
Total n	627	707	653
Uniqueness in %	37,19	42,26	37,12

Table 8.8: The total number of tokens and vocabulary in the different implementations and how much of it is unique.

According to class coupling metric, implementation C is the best implementation, with least class coupling between them, making it the least complex out of all the implementations.

8.2 Case 2: The storage service

Although it is a different case, and it will not be directly compared to implementations from case 1, the three implementations, in this case, will also be called implementation A, B, and C. The total number of methods and the number of methods used in calculation for average score per method is shown in figure 8.2.

8.2.1 Halstead's Complexity metrics

The total number of token and number of vocabulary can help with reusability within the system. How much of operands or operators are being used and them also being reused. A total number of tokens N, and the number of vocabulary n, in the system, is shown in table 8.8.

Once again, C shows that just because it has the highest total tokens, it does not have the highest vocabulary. Which in uniqueness in percentage shows that, B has the implementation with least reuse of code. A shows that it does not have the highest number of tokens, neither the lowest but it does have the lowest vocabulary and very close to C when it comes to most reuse of code.

Halstead's volume

Halstead's total number of tokens and the size of its vocabulary alone may not give a lot of useful information, however using them to find the volume of each implementation might give more useful and relevant information. It is worth remembering that the lower scores are better since they all have the same features, but different implementations. The one that is the least amount of volume is better than others that have the same features, but with higher volume. With the volume of each implementation being:

- **A:** 15666,85
- **B:** 15835,89
- **C:** 16448,30

As explained in the previous chapter, the score for each file should be within 100 and at most 8000. For A, it has 7 files, which means it has in average 2234,12 volume per class, which is a good way below the max limit. For B, it is 1750,54 but it has 9 files in the implementation. The two extra is for the classes that start the service and the other is for starting the client.

Most of the Halstead metrics are meant for method level. A graph of top method from each implementation is shown in figure 8.7. Implementation B does not look like the preferable option from the figure as it receives a high score throughout most of the methods. Especially with methods, StartStratService.initService and StartClients.initClient. It does make it look worse. For A and C, it does look a little closer between them.

Looking at the average volume per method, that tells us the average amount of information in a bit for each method, A and C is not that close. The average volume for methods in :

- **A:** 66,33
- **B:** 64,40
- **C:** 50,59

Showing that C has the lowest average although it has the highest volume in total. Also the highest number of methods that are part of the calculation, which does reduce the average score.

B has neither the highest, nor the lowest volume in total, and the same position in average and number of methods part of the calculation. While A has the lowest in total volume for the whole system, the lowest number of methods part of the calculation and highest average volume per method.

Halstead's effort

The higher the score is, the more effort is required to understand and implement the method. Based only from the score for top methods, shown in figure 8.8, implementation A looks to be the one that requires most effort to understand and implement, while C is the implementation that requires the least amount of effort.

The average effort required per method is following:

- **A:** 751,83
- **B:** 693,75
- **C:** 380,38

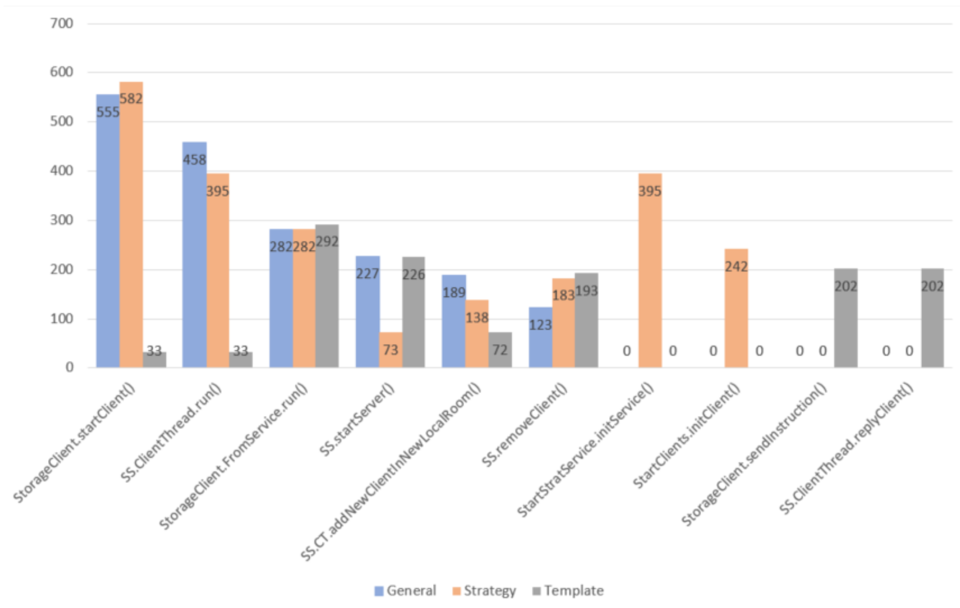


Figure 8.7: Result from Halstead volume measurements with seven methods which has the highest scores.

It does back up the figures from figure 8.8, also average effort required per method, A requires the highest effort while C is the lowest by far. The number of methods does reduce the average score for C, but overall C is easier as it requires less effort.

A reason why C has received such low score is that of the pattern, Template pattern. It defines the skeleton of how an algorithm should be performed. The two highest scoring methods for A and B has not that pattern, which means that it is doing all that in one single method. Especially the StorageClient.startClient method. The algorithm is following:

1. Make a connection to service
2. Add client to a new area
3. Receive instruction from the client
4. Decode and convert the instruction into a message for service to understand.
5. Send it to service
6. When client eventually wants, end connection to service.

In A and B, that is done in one method. It is the structure of the algorithm. It does not change a step, or else things will go wrong. The method is still below the threshold of the maximum recommended. While the same procedure is spread over 6 methods. Which reduces the average score. Two of the biggest methods from A and B received 33 each for C while receiving 555 and 458 for A and 582 and 395 for B.

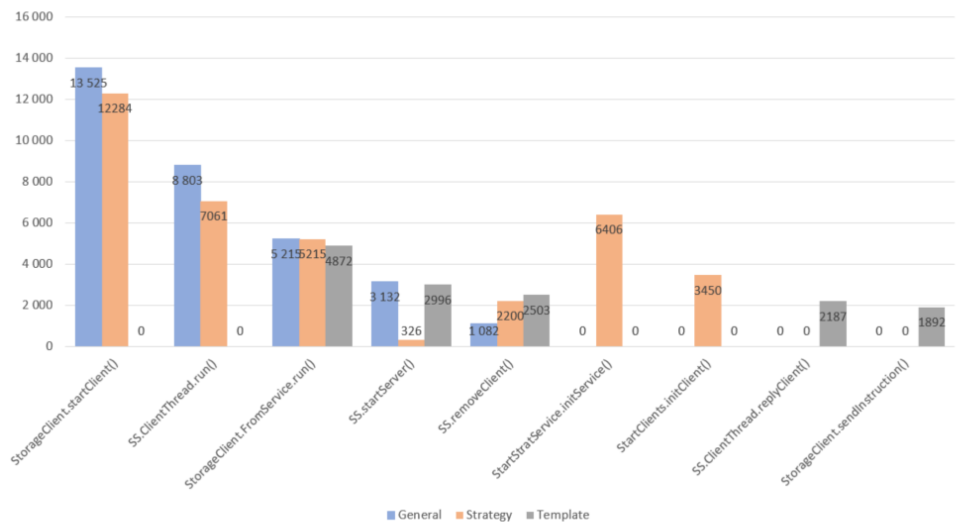


Figure 8.8: Result from Halstead effort measurements.

	A	B	C
Average per method	1,78	1,82	1,71
Total	123	133	142

Table 8.9: The total cyclic complexity and average per method index throughout the system.

When it comes to least effort required for understanding and implementing the system, C is by far the better implementation, as the average per method score states. It is much easier to understand it than the other.

8.2.2 Cyclomatic complexity and Cognitive complexity

The cyclomatic complexity of the system is even. It is not clear to see which implementation is preferable from figure 8.9.

The total number of cyclomatic complexity for the whole system is lower for implementation A than it is for B and C. It has the most amount of methods, each of these methods does increase with one. And since C has the highest number of methods that can be calculated, it is easy to see why has the highest score. C has 10 more methods than B and 11 more in points. So that definitely comes to play when calculating the total. When just looking at the total, C might be the most complex out of these three.

But the average per method score, however, say otherwise. The lowest average cyclomatic complexity score is for C, while B receiving the highest. Showing that in total, C is less complex than the other implementations, if we follow the cyclomatic complexity metric.

Cyclomatic complexity works great for method level, but cognitive complexity should give us a information on a system level, which is also nice because a calculation of a metric on a much wider level gives us information of the system as a whole. While on method level, the score from each method is used for calculating the system. It is not worse, but just on method level, it is prone to potential human error prone. It is also an updated metric for more modern languages, like Java 8, which is used here.

Seeing figure 8.9, A has the least cognitive complexity score in total with 60 and C has the highest with 68. In total, it makes C the most complex of these implementations. B has the second highest with 64, making it more complex than A but less than C. If we take a look at the score for StorageService and StorageClient class, we see that the StorageService class for A, is far more complex than the same class on B and C and making B the least complex according to the cognitive complexity. For StorageClient class, C has the most complex while B again is the least complex.

How much of the total scores the two classes take, tell us how complex the rest of the system is. For A, StorageService and StorageClient take 59 out 60 points, making the rest of the system almost complex free, having only one point over two different classes. While for B, these two classes cover 42 out of 64 points, making the rest of system have 22 points spread over the classes, which is 5 classes. While for C, the 4 classes on the system have 12 points spread between them, because the two classes cover 56 out of total 68. Making the rest of A the easiest system, and the rest of B the hardest.

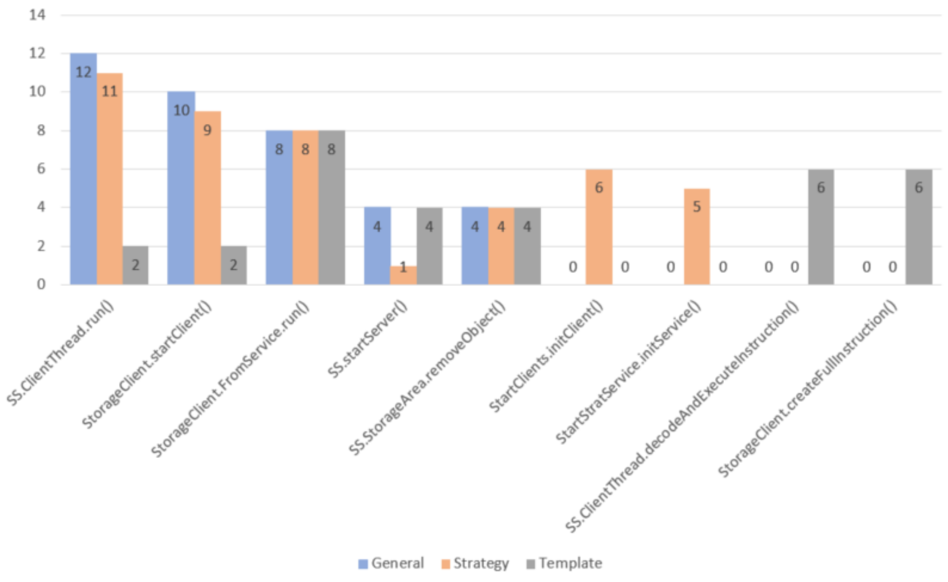


Figure 8.9: Result from Halstead cyclomatic complexity measurements

The interesting here is I conclude with two different results for Cognitive complexity and cyclomatic complexity score. With C being the better implementation with cyclomatic complexity and A for cognitive complexity.

A does have the lowest total cyclomatic complexity score, yes, just like total cognitive complexity score, however, when calculating the average cyclomatic complexity score per method, it is proven that C is the better implementation. Because the score is spread over multiple smaller methods, than fewer but larger methods.

While cyclomatic complexity works best on method level, it is interesting to see whether the metric for the whole system confirms it. The cognitive complexity scores for each implementation, shown in figure 8.10, it is again proven how close implementation A and B is. Only 2 points are the difference between them, with A receiving the lowest score. Implementation C is again proven to be the most complex of the three implementations.

The StorageService class is the class which is the class that is mostly affected by the different patterns, and also the biggest class in each implementation. As shown in figure 8.10, for implementation A, the class receives 59,01% of the whole score. For implementation B, 51,47% and for implementation C, 52,63%. It makes sense that the class in implementation A has the highest score, since it has all of the implementations, while the same class for the other two has some of it implemented in their superclass.

Based on cyclomatic and cognitive complexity scores, implementation A and B comes neck to neck when comparing. But based on the outcome of these two metrics, implementation A is the preferable option. when it comes to cyclomatic complexity, it is B, because it has 0,06 point lower, although in total it has a higher score, it is spread over more methods, which makes it easier. While for cognitive, it is A, because of its score lower in total.

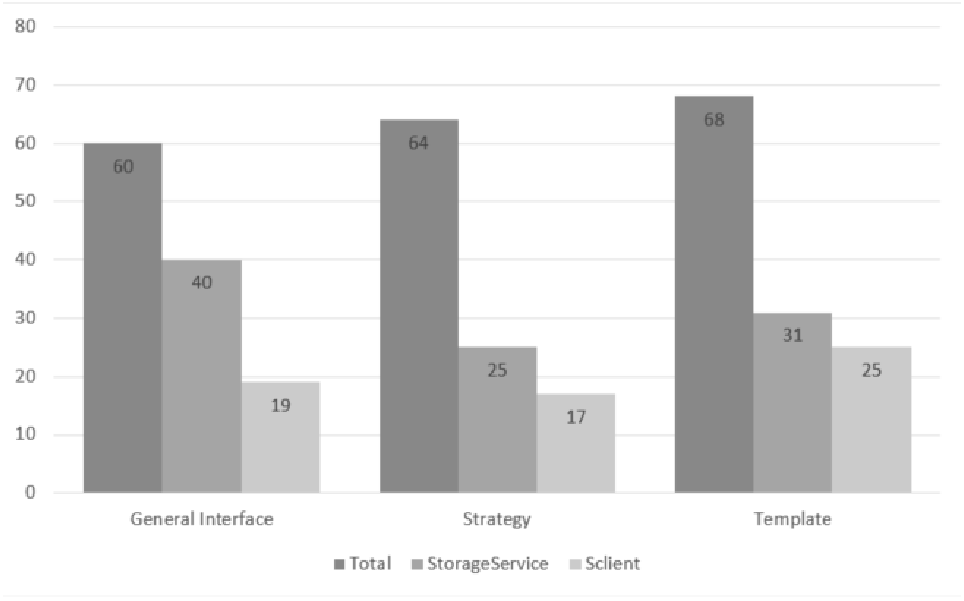


Figure 8.10: Result from Halstead cognitive complexity measurements

8.2.3 Lines of code

Table 8.10 gives information on the number of lines on each implementation, with implementation a having the lowest number of lines of code, with 58 lines separating it from implementation B which has the second lowest. While C is the one with the highest number of lines of code, with 708 lines.

	A	B	C
Total lines of code	636	693	708

Table 8.10: Total lines of code in each implementation for Storage implementation.

8.2.4 Maintenance index

Table 8.11 shows that the overall maintainability of each case is not bad. The higher score, the better maintainable the system is. The systems have a potential to improve, but they are not bad compared to systems where the score is negative. The score tells us that implementation C is the best maintainable system among these three.

Microsoft's index gives a much better indication of the system maintainability with a maximum and minimum score, while the original maintainability score only gives the highest score that is 171, as shown in table 8.12. Microsoft's maintainability index again confirms that it is not the most maintainable system. All 3 implementations receive below half of the score. However, Implementation B is the most maintainable system, even if it by a very small margin.

Table 8.12 shows that the score is just below half the top score possible. It is also showing that when it comes to maintainability index, C is the better maintainable implementation than A and B. It is the implementation with the highest score. B is the implementation with the lowest maintainability score, although it is not much lower. While A is a mid-table score, that is closer to B than it is to C.

8.2.5 Class coupling

Figure 8.11 shows how close the coupling in each class is. However, in this case, implementation A, is not the implementation with the highest coupling. It is implementation B that is shared highest or highest in all but two classes, with does make it the class with the highest coupling. A and C are close, but because of the two interfaces for A, Client and ServiceInterface, it does receive a lower score. If only looking at head-to-head metrics, where both A and C have scored higher than 0, then it is even,

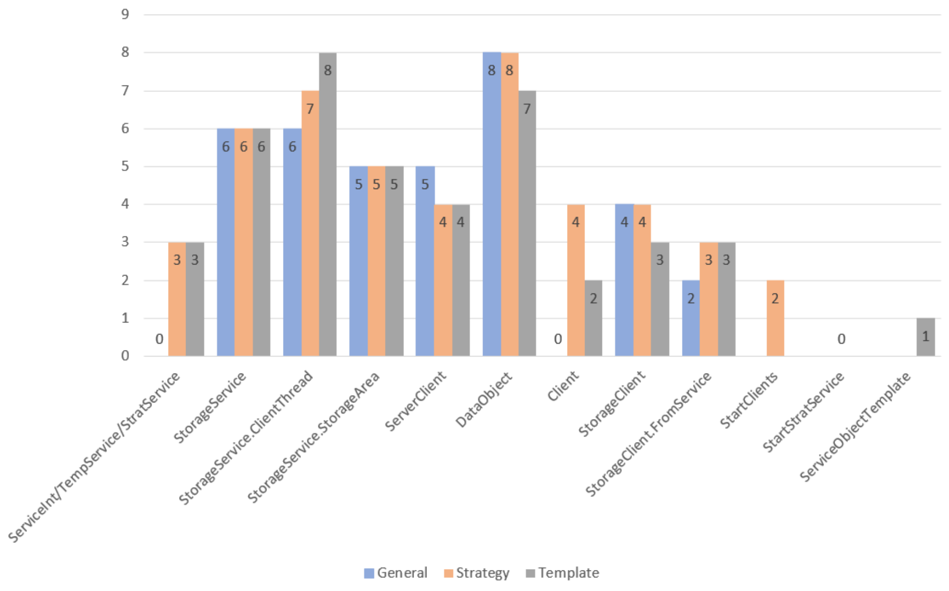
	Implementation A	Implementation B	Implementation C
Avg(H)	66,33	63,40	50,59
Avg(C)	1,78	1,82	1,84
Avg(L)	7,39	8,06	6,49
index	**75,83**	**73,76**	**80,95**

Table 8.11: Maintainability index for each implementation along with the other metrics.

	A	B	C
MAX(0, index)	75,83	73,76	80,95
Microsoft's index	44,35	43,13	47,34

Table 8.12: Calculation of Microsofts maintainability index for case 2

they would get the equal total. But because of the interfaces, and the classes that have scored higher than 0 are all the classes that are in the count for A, received the same score, while C has other classes that received higher score, is the reason A is the implementation with least coupling, making it least complex and error-prone to changes in the future, which makes it easier to maintain.



Class	General	Strategy	Template
ServiceInt/TempService/StratService	0	3	3
StorageService	6	6	6
StorageService.ClientThread	6	7	8
StorageService.StorageArea	5	5	5
ServerClient	5	4	4
DataObject	8	8	7
Client	0	4	2
StorageClient	4	4	3
StorageClient.FromService	2	3	3
StartClients		2	
StartStratService	0		
ServiceObjectTemplate			1

Figure 8.11: Class coupling for class in case 2

8.3 Total patterns

Observer pattern

Observer pattern for this system is used with case 1, the chat service. The Chat service, the observer, has one room for each client, the subjects. Multiple clients could subscribe to the same room, and when changes are made to the room, the clients that are subscribed to that room receives an update with the message.

However, it is not simple to use on with a distributed system, because of internet connection. For this experiment, the clients are connected through Transmission Control Protocol (TCP), which is a safe protocol. Which means there has to be a connection between client and server for the server to be sending a notification. If a clients connection is disconnected, then that client will not receive notifications. When they connect again, they can ask for all changes made in the system. Also, the message that gets sent to and from the client will be received in the same order it is sent.

In this system, for the client to get a notification, the server must have an active socket of the client. If there are 20 clients in one room, and one sends the message, all the other client must get the update, thus the server must have the other 19 active sockets ready for sending the update. That is a weakness in my opinion. An unstable network can lead to clients not getting notified because of a bad connection, thus making it only a decent observer pattern implementation. Because there are plenty of areas that it does work, like having subjects and observers that work well separately. When all clients are connected through a stable network, the whole system works great, all clients get the notification when their room is updated.

Mediator pattern

Mediator pattern is also used in case 1 where the chat service works as a mediator that handles the communication between clients. Any communication between the clients that share the same room, goes through the chat service.

Any implementation of the client from the different patterns would work with any other implementations service. Each client implementation does not need to know about the others clients implementation, because they now have a common object that shares the common features that also reduces the class coupling between them.

The Mediator pattern works better for this system than the observer pattern. The only minor limitation that I can think of is that something the clients must have in common is how the message that is sent. The DataObject object itself cannot be read directly, rather its content is read separately. It has a name which is of type String, a content of type Object and time that is of type LocalDateTime. All types are supported by Java and are accessible in any client object. As long as the classes are able to handle these, it should be able to connect and use the service. Since sockets are used for connecting client and service, the service can not invoke any method from the service side, instead, it sends over a socket and hopes that the service can handle it.

I believe that with RMI, Mediator pattern could have been used even better. As they know each others implementation, it can handle sending and receiving even better. If sending a message, the service can invoke a method on the client's side that handles it.

8.4 Summary

most of the result from case 2 does confirm it.

The metrics and tools that are described in chapter 4, are used here gather data on how each implementation compare against each other. Which implementation has the better score? The results are for each

	Case 1			**Case 2**		
	A	**B**	**C**	**A**	**B**	**C**
Halstead Volume			X			X
Halstead Effort			X			X
Cyclomatic Complexity	X					X
Cognitive Complexity	X			X		
Maintainability			X			X
Class Coupling			X	X		
	2	0	4	2	0	4

Table 8.13: Total score table

case and each metric is shown in table 8.13, where X represents the implementation that received the better score.

Implementation C is the implementation the does receive the best score overall for both cases. For 6 metrics, 4 of them agree on both cases for which implementation is better. Better on some complexity metrics and the maintainability index.

Observer pattern works in most areas, but when it comes to its main purpose, it fails a bit. TCP protocol is the safest protocol here, although the notification bit is where it fails.

Mediator pattern works great, using the service class as a mediator for handling the communication between the other clients. But can be even better with RMI rather than using socket and TCP protocol.

Chapter 9

Conclusion

In this thesis, which of chosen design pattern would fit in a client-server application, and would be the better implementation. Using the metrics described in chapter 4.2, the implementation with Template pattern is the implementation that received the best score overall. Receiving the score on Halstead's complexity metrics, cyclomatic complexity for case 2, maintainability metric for both cases and class coupling for case 1. While the implementation without design patterns received the best score according to the cognitive complexity, cyclomatic complexity for case 1 and lowest class coupling for case 2.

In one of the cases, observer pattern was used for seeing how it works in a distributed client-server service. Some part works great, but if sockets are used for connecting clients to the service, then it has a limitation as well. With the client's socket has to be connected and active for it to be notified when changes are made in service.

Mediator pattern was also implemented in case 1, where the chat service is used as a mediator and connecting clients to each other. Different implementations of the client can work with chat service. As long as they can send the message in a certain way, which might be a minor limitation, as sockets are connected to each other.

In conclusion, for reducing complexity and increasing maintainability, Template pattern is design pattern to use instead of Strategy patterns and just a basic interface implementation without any design patterns. Since the connection to service, the communication and ending the communication is following structural steps, Template pattern can help reduce the complexity instead of doing it all in one method.

Bibliography

[1] Asus. *Chromebit (CS10) | Chrome-enheter | ASUS Norge*. URL: https://www.asus.com/no/Chrome-Devices/Chromebit-CS10/ (visited on 01/05/2018).

[2] AWS. *Summary of the December 24, 2012 Amazon ELB Service Event in the US-East Region*. URL: https://aws.amazon.com/message/680587/ (visited on 01/05/2018).

[3] Rahul C. Basole and Jürgen Karla. 'On the Evolution of Mobile Platform Ecosystem Structure and Strategy'. In: *Business & Information Systems Engineering* 3.5 (2011), pp. 313–322. ISSN: 1867-0202. DOI: 10.1007/s12599-011-0174-4. URL: http://link.springer.com/10.1007/s12599-011-0174-4.

[4] G. Ann Campbell. *Cognitive Complexity™ | A new way of measuring understandability*. Tech. rep. 2018, p. 21. URL: https://www.sonarsource.com/docs/CognitiveComplexity.pdf.

[5] Shyam R. Chidamber and Chris F. Kemerer. 'A metrics Suite for Object Oriented Design'. In: (1992), p. 39. URL: https://dspace.mit.edu/bitstream/handle/1721.1/48424/metricssuiteforo00chid.pdf?s...

[6] Cory Doctorow. *Not every cloud has a silver lining: Cory Doctorow | Technology | The Guardian*. URL: https://www.theguardian.com/technology/2009/sep/02/cory-doctorow-cloud-computing (visited on 01/05/2018).

[7] Taymour El Erian. *The XaaS family: Understanding Iaas, PaaS and SaaS - Cloud computing news*. URL: https://www.ibm.com/blogs/cloud-computing/2014/10/31/xaas-family-iaas-paas-saas-explained/ (visited on 01/05/2018).

[8] Norman E. Fenton and James Bieman. *Software metrics : a rigorous and practical approach*, p. 595. ISBN: 1439838232. URL: https://books.google.no/books?hl=no{\&}lr={\&}id=lx{_}OBQAAQBAJ{\&}oi=fnd{\&}pg=PP1{\&}dq=Software+metrics+:+a+rigorous+and+practical+approach+/+N.E.+Fenton,+S.L.+PïňĆeeger{\&}ots={_}UiWOkYPXv{\&}sig=y-eQos-UOqiBsrUxr5pbcVOtTl0{\&}redir{_}esc=y{\#}v=onepage{\&}q=internal{\&}f=false.

[9] Erich. Gamma. *Design patterns : elements of reusable object-oriented software*. Addison-Wesley, 1995, p. 395. ISBN: 0201633612.

[10] Google. *Google Chromebooks*. URL: https://www.google.com/intl/no{_}no/chromebook/ (visited on 01/05/2018).

[11] M H Halstead. 'POTENTIAL CONTRIBUTIONS OF SOFTWARE SCIENCE TO SOFTWARE RELIABILITY'. In: (1977). URL: https://pdfs.semanticscholar.org/edde/f5d06997bf4358838f0c4fa3516b1cc2dd10.pdf.

[12] Stephen H. Kan. *Metrics and models in software quality engineering*. Addison-Wesley, 2003, p. 528. ISBN: 9780201729153.

[13] 1824-1907 Kelvin, William Thomson, Baron. *Popular lectures and addresses*. 1st ed. London Macmillan, 1889, p. 490. DOI: https://doi.org/10.5962/bhl.title.31742. URL: https://archive.org/details/popularlecturesa01kelvuoft.

[14] T.J. MacCabe et al. *Structured testing*. IEEE Computer Society Press, 1983. URL: https://books.google.no/books?id=vtNWAAAAMAAJ.

[15] McCabe. 'Metric Name / Description Additional Description Notes Threshold Min Value'. In: (). URL: http://www.mccabe.com/pdf/McCabeIQMetrics.pdf.

[16] Joe McKendrick. *Apple Co-Founder Steve Wozniak Distrusts the Cloud: Is He Right?* URL: https://www.forbes.com/sites/joemckendrick/2012/08/06/apple-co-founder-steve-wozniak-distrusts-the-cloud-is-he-right/{\#}535583376042 (visited on 01/05/2018).

[17] MetricsReloaded. *MetricsReloaded :: JetBrains Plugin Repository*. URL: https://plugins.jetbrains.com/plugin/93-metricsreloaded (visited on 01/05/2018).

[18] Jennifer Wall : MSFC. 'What Are Clouds?' In: (2015). URL: https://www.nasa.gov/audience/forstudents/5-8/features/nasa-knows/what-are-clouds-58.html.

[19] Zain Naboulsi. *Code Metrics – Maintainability Index – The Ultimate Visual Studio Tips and Tricks Blog*. URL: https://blogs.msdn.microsoft.com/zainnab/2011/05/26/code-metrics-maintainability-index/ (visited on 01/05/2018).

[20] Antonio Regalado. *Who Coined 'Cloud Computing'? - MIT Technology Review*. 2011. URL: https://www.technologyreview.com/s/425970/who-coined-cloud-computing/ (visited on 30/04/2018).

[21] SonarQube. *Continuous Code Quality | SonarQube*. URL: https://www.sonarqube.org/ (visited on 01/05/2018).

[22] USwitch. *The History of Broadband from the '80s to today*. URL: https://www.uswitch.com/broadband/guides/broadband{_}history/ (visited on 01/05/2018).

[23] Webster-Merriem. *Objective | Definition of Objective by Merriam-Webster*. URL: https://www.merriam-webster.com/dictionary/objective (visited on 01/05/2018).

[24] Robert S. Witte and John S. Witte. *Statistics*. J. Wiley & Sons, 2010, p. 556. ISBN: 0470392223.