ITRW324 Research project

**Leonard Wassenaar**

Research Project in ITRW324 Submitted for the degree BSc in information technology at the North-West University Vaal Triangle Campus

Student number:

25948083

**Table of contents**

[1.1 Introduction 1](#_Toc520662663)

[1.2 Concepts for discussion 1](#_Toc520662664)

[1.2.1 Architectural patterns 1](#_Toc520662665)

[1.2.1.1 Examples of architectural patterns 1](#_Toc520662666)

[1.2.1.1.1 Model-view-controller 1](#_Toc520662667)

[1.2.1.1.2 Layered pattern 2](#_Toc520662668)

[1.2.1.1.3 Event-driven architectural pattern 2](#_Toc520662669)

[1.2.1.1.4 Client-server(Distributed) pattern 3](#_Toc520662670)

[1.2.1.1.5 Peer-to-peer pattern 4](#_Toc520662671)

[1.2.2 Code refactoring 5](#_Toc520662672)

[1.2.2.1 Refactoring Techniques 5](#_Toc520662673)

[1.2.2.1.1 Red-Green Refactoring 5](#_Toc520662674)

[1.2.2.1.2 Preparatory Refactoring 5](#_Toc520662675)

[1.2.2.1.3 Abstraction Refactoring 6](#_Toc520662676)

[1.2.2.1.4 Method Refactoring 6](#_Toc520662677)

[1.2.2.1.5 User interface Refactoring 6](#_Toc520662678)

[1.2.3 Advanced design principles (Including SOLID) 6](#_Toc520662679)

[1.2.4 Unit testing 9](#_Toc520662680)

[1.3 Conclusion 11](#_Toc520662681)

List of Tables

[Table 1 : Advantages and disadvantages of common patterns https://bit.ly/2HR4N0M 4](#_Toc520662683)

**LIST OF FIGURES**

[Figure 1 : Example of a layered pattern https://bit.ly/2OgEPYS 2](#_Toc520662684)

[Figure 2: Figure of the Client/Server model https://bit.ly/2AuLgoz 3](#_Toc520662685)

[Figure 3: Figure of a peer-to-peer model https://bit.ly/2LrBlEY 4](#_Toc520662686)

[Figure 4 : Testing foundation https://bit.ly/2Aqa7tg 9](#_Toc520662687)

[Figure 5 : Unit testing framework https://bit.ly/2Lu56VT 9](#_Toc520662688)

Assignment 1

## Introduction

The principle of the research is aimed at providing an argumentative overview of the various software engineering concepts. This requires the investigation of various sources to collect information about the specified topics to analyse the gathered evidence from the different argumentative perspectives. The goal is to determine the state of the subject in a brief but comprehensive manner.

Topics such as architectural patterns (Section 1.2.1) in commonly occurring problems, the re-factoring of code (Section 1.2.2) for more readable and reusable code, progressed design principles (Section 1.2.3) for more productive usability and unit testing (Section 1.2.4) of computer programs will be discussed. These important topics are the basic building blocks of software engineering. Software engineering can be described as the design, development and maintenance of computer programs. Software engineering is essential for the production of quality software that is adaptable and scalable for future constraints and requirements. It is a never-ending loop of constant preservation and maintenance.

## Concepts for discussion

### Architectural patterns

Before moving to the development phase, an appropriate architecture must be chosen (Mallawaarachchi, 2017). Architectural patterns are a necessity for useful architectural design. An architectural pattern is a resolution to generally arising issues in software architectures (Mallawaarachchi, 2017). According to Chhatpar (2007), architectural patterns can be considered as an organizational representation of various software systems. It offers several subsystems, with their duties and directions to organize the associations among them (Chhatpar, 2007).

#### Examples of architectural patterns

##### Model-view-controller

This type of architectural pattern is used to speed up the development of desktop, web and mobile applications. Large interfaces can be implemented by separating the application into different parts, namely, the model, viewer and controller (Kumar, 2016). Coodrich (2017), also describes the model-view-controller as an architecture that is divided into separate components. The model handles the data and logic of the business, the controller coordinates the user interfaces and the viewer controls the objects of the graphical user interface and appearances (Coodrich, 2017).

The model-view-controller architectural pattern is commonly used in the development of web applications. This type of architectural pattern provides good scalability, meaning, that the room for improvement and growth is high. It is easier to maintain, because the parts of the architecture have low dependencies. The model used, can be reused several times. Front and backend developers can be employed to accelerate the development process (Kumar, 2016). This architectural pattern also has a drawback. A number of developers think that this architecture increases complexity, because of the rapid development time (Kumar, 2016). Despite the benefits and shortcomings, the model-view-architecture is very popular in today’s development framework.

##### Layered pattern

Layered patterns are commonly used in desktop and e-commerce applications. A layered pattern can be defined as the decomposition of structural programs into sub-tasks (Richards, 2015:1). Each of these tasks has a certain level of abstraction known as the UI layer, service layer, domain layer and persistence layer. These layers are ordered into horizontal layers, where each horizontal layer has a specific role within a program (Richards, 2015:1). Example of a layered pattern seen in **figure 1**

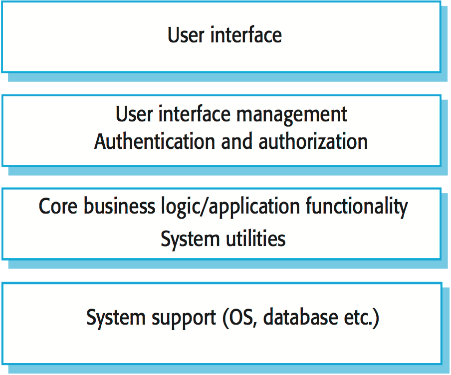


Figure 1 : Example of a layered pattern (<https://bit.ly/2OgEPYS> )

##### Event-driven architectural pattern

This architectural pattern is used in highly scalable programs. Event-driven architectures are generally very adaptable and is utilized in small and big applications (Richards, 2015:11). Richards (2015:11) states that this type of architectural pattern is compiled of decoupled, single-oriented event processing mechanisms that can collect and process events. In contrast Rouse (n.d) states that the event-driven architecture is compiled by event creators and event users. Examples of the type of events is sensors, business rules, transactions, processes and errors.

The pattern of an event-driven architecture has high agility. It has the ability to respond quickly to never ending fluctuating requirements (Richards, 2015:18). It is also hassle-free to deploy, as a result of the characteristics of the processing event mechanisms. The event driven architecture has a high level of performance. It accomplishes high performances because of its ability to carry out parallel operations with success. This type of architecture accomplishes high levels of scalability, because of the event processors independence (Richards, 2015:19).

##### Client-server(Distributed) pattern

This type of pattern consists of two components; clients and the server, seen in **figure 2**. Clients request and receive services from the server (Mallawaarachchi, 2017). A good example of client-server patterns is banking and document sharing. The messages sent between the parties involved have the exact same format (Kumar, 2015:1021). The client can be thought of as a quest and the server can be considered as a host (Kumar, 2015:1021). Security is effectively handled in this type of system. Another advantage is that data can be effectively added and discarded without any doubles. The main disadvantages are, it is costly to set up, more maintenance is required, and all computers are impacted when there is an error on the server.

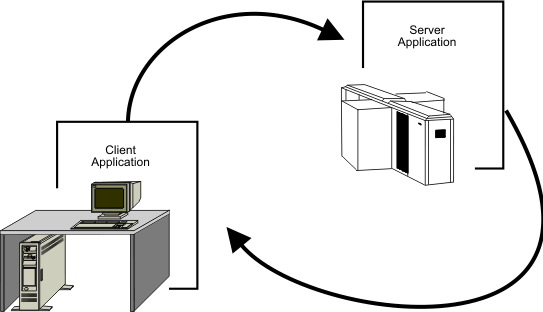


Figure 2: Figure of the Client/Server model (<https://bit.ly/2AuLgoz> )

##### Peer-to-peer pattern

In a Peer-to-peer pattern, peers are considered as clients who receive and request services from each other, **seen figure 3**. It is almost similar to a Client-server pattern. One peer may represent a server, that delivers services to other peers (Mallawaarachchi, 2017). Hyderabad (n.d) describes the peer-to-peer pattern as two or more computers who are interconnected where no computer is in a favourable position.

The peer-to-peer pattern has several advantages and disadvantages. The main advantages of a peer-to-peer pattern is that the setup is trouble-free, no dedicated server is required, and it is less expensive. The main disadvantage is that any computer involved can be accessed (Rehman, 2017). According to Hyderabad (n.d), sperate passwords must be used for each computer and unsigned codes can access data from a remote location.

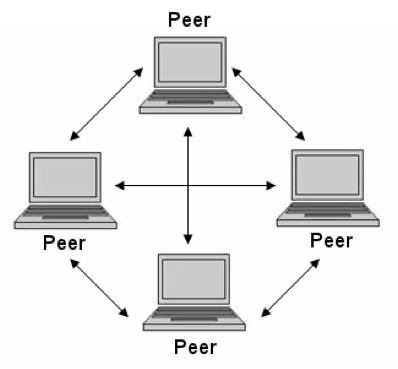
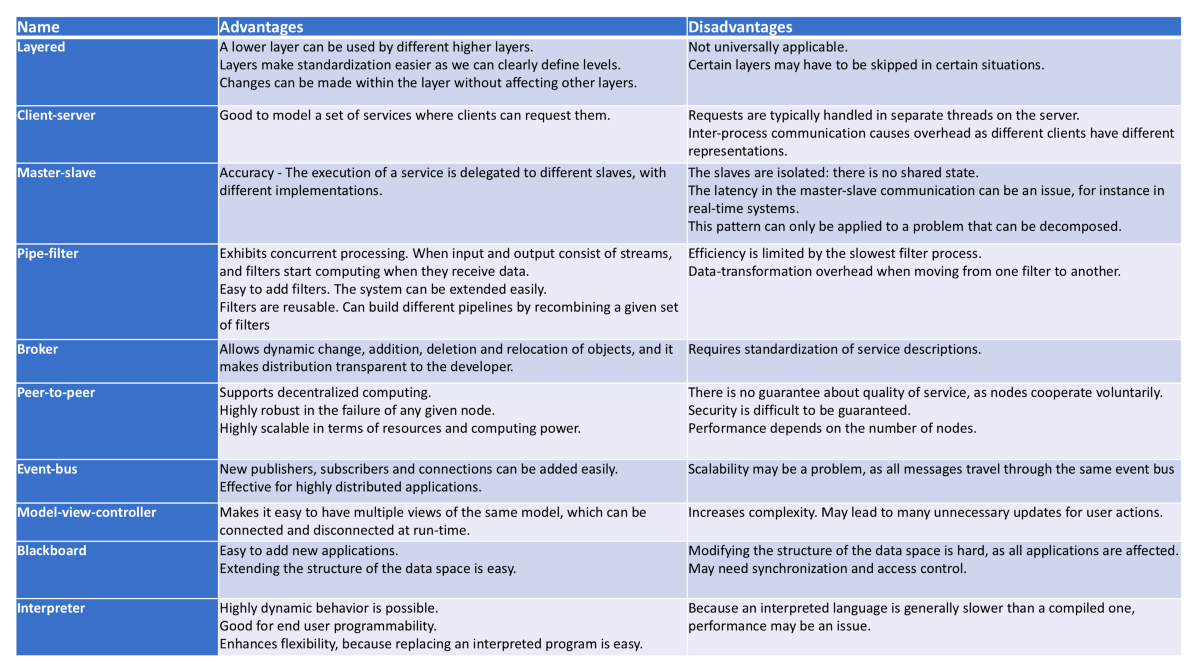


Figure 3: Figure of a peer-to-peer model (<https://bit.ly/2LrBlEY> )

Table 1 : Advantages and disadvantages of common patterns (<https://bit.ly/2HR4N0M> )



### Code refactoring

Refactoring involves taking already written computer code and changing it to make it more readable and comprehensive. Fowler (1999: xvi), describes code refactoring as the changing of the inner structure of a software system without changing the external structure. da Silva (2010:145), supports Fowler’s description of code refactoring, where code refactoring is described as a technique used in software engineering that adjusts existing code to make it more sustainable. The term refactoring is a well-known concept that dates back to the early 90’s (Barbary, 2016).

Code should be refactored when the room for improvement is large, where identical code is used twice and when too long or unnecessary methods are used (Ershad, 2017). Larger methods or classes with more complex structures gets broken down for a simpler understanding with code refactoring. A peer-review can aid in identifying code that needs to be refactored. The reviewer can detect bad or unnecessary code known as code smell (Ershad, 2017). Refactoring code can aid in maintenance and can make a program more adaptable for future changes (Novoseltseva, 2018).

#### Refactoring Techniques

##### Red-Green Refactoring

This refactoring technique is the foundation of all the refactoring techniques. It involves starting with a fail test, and secondly, writing basic code to get the code to pass(green) and maintaining the code to keep the state green (Novoseltseva, 2018). Michell (2017) states that the intent of Red-Green Refactoring is to optimize the existing solution without effecting other parts of the code. Red-Green Refactoring ensures that code is kept clean in the development process. There are two phases in this refactoring technique, namely, adding code to provide more functionality, and making enhancements that provides more functionality (Novoseltseva, 2018).

##### Preparatory Refactoring

This type of refactoring technique can be done by following the red-green refactoring method (Muthupalaniappan, 2015). This refactoring technique also includes the fixing of technicalities that was hoarded in the earlier development phases (Novoseltseva, 2018). Novoseltseva (2015), also states that the red-green method can be used in preparatory refactoring. Preparatory refactoring has several advantages as it enables you to follow all the necessary steps. It enables you to comprehend the code you want to modify through planning and designing changes. Code can then be refactored while being in the green zone (above mentioned technique).

##### Abstraction Refactoring

The abstraction refactoring technique has its own unique refactoring methods (Novoseltseva, 2015). It involves constructing new classes and interfaces and substituting inheritance with delegation. This technique is mostly used in refactoring with a broad scope. The idea of abstraction refactoring is to isolate the parts of the application that needs refactoring (Muthupalaniappan, 2015). The new refactored code will then ultimately replace the isolated parts.

##### Method Refactoring

Method Refactoring enables a person to better comprehend methods and method calls. It simplifies the communication among the different classes. The main idea of method refactoring is good naming conventions (Alexander, 2018). If you follow the naming conventions of programming, the need for refactoring will be greatly reduced. Method Refactoring involves adding new parameters, removing unnecessary parameters, giving methods more suitable names and removing methods that is not applicable to the scope of the system. In C#, an operation can be used to create a new method from existing code (microsoft, n.d). This operation is known as an extract method that is located in the refactor menu of C#.

##### User interface Refactoring

To refactor the user interface is also important. It includes changing buttons to appropriate sizes, using a suitable readable font, using the correct colour contrast and using appropriate labels as headings and sub-headings (Novoseltseva, 2018). The refactoring of the user interface will enhance the usability for your team members in the long term. The construction of the user interface is the first impression a user will have of the application. First impressions are a key factor that can make or break an application’s popularity.

### Advanced design principles (Including SOLID)

Developing software can be numerous things. Passion, methodology and creativity is a good starting point (Maksimovic, 2017). In software engineering, design principles and practices must be used to ensure effortless readability for users. When these principles are successfully incorporated, it should be able to keep maintenance to a minimum, reduce expenses, increase usability and ensures the application is error free. If effort is put into the design phase, less time will be spent on changing requirements (Terek, 2018).

Before a team can begin with the design phase, the problem needs to be understood (Telkar, n.d).After the scope of the problem is studied and understood, different versions of software can be designed. This increases the possibility of having a good design structure. A good design is a product of good systems thinking and facts (Star, 2018). The present design of today mostly relies on deploying modular components.

According to McGee (2018) modular components work together to achieve one functional unit. IBM (n.d) argues that modular programming is a process that divides a problem into single units that are independent from each other. This increases the understanding and ability to modify each part. There are plenty of benefits linked to modular programming. It greatly increases the speed of development. Different programmers can work on different units at once. The debugging process becomes easier and less maintenance is required. It also increases the understandability of how these individual parts work with each other. Lastly, most of the units can be reused, bypassing code that needs to be re-written (Jackson, n.d).

The design of a program should be based on the analysis model (Thakur, n.d). This means that the design should satisfy previously defined requirements (Thakur, n.d). Segue Technologies (2013) describes these type of requirements as obligations that provides us the necessary ability to describe the values and scope of the type of software that is to be developed. Segue Technologies (2013) states that the following questions should be asked to determine the requirements; What is the reason for developing the particular project? What is the value of the project for the development team? What is the developed produced supposed the do? How sure is the team that the developed product will be adaptable for future requirements?

The right programming paradigm must be chosen. The structure of the system to be developed must be defined (Thakur, n.d). Procedure-oriented or object-oriented paradigms can be used. Constraints like, time and resources should be kept in mind when choosing a certain paradigm. An object-oriented paradigm uses classes and objects. The main benefit of this type of paradigm, is hassle-free maintainability of the application program (Eliason, 2013). Half (2017), states that the main benefits of object-oriented programming, is simpler troubleshooting, reusability of code by making use of inheritance, more flexibility because of polymorphism and effective problem solving. Procedural-oriented paradigms follows a top to bottom approach in programming (Eliason, 2013). It is a list of instructions, where each line tells the computer to perform a certain action. Procedural programming focusses on a process and not data. Problems are handled as a sequence of certain things to be completed, such a reading and output. It is important to choose the appropriate programming paradigm for the type of application program to be developed.

The SOLID laws are principles of object-oriented programming (Martin, n.d). These principles are known as SOLID, meaning, single responsibility principle, open/close principle, liskov substitution principle, interface segregation principle and dependency inversion. These principles are commonly used by developers. The principles can enable a person to write code that is good and efficient. The segregation Principe states that a class should only have one reason to be modified (Kelmendi, 2017). The open-close principle states, that a behaviour of a class should be extended without modification. The liskov substitution principle proclaims that classes that are derived must have the capacity to be substituted by the base class(Martin, 2000: 8).. Interface segregations involves making interfaces that are specific to the clients using it. Lastly, dependency inversion, states that details should not be the dependent of abstractions, but details must dependent on abstractions (Martin, n.d).

Designing an application for testability is important when designing software. Testing should be done separately from the implementation phase. The testers of the application ensure that the application is suitable for operation to be distributed to the users. Testing for usability contributes hugely for the flexibility of programs (Zilberfeld, 2012). In contrast, Thakur (n.d) states that there is a serious flaw in the process of splitting testing from the design phase, because there might be errors in the design and implementation phase. Errors in the design and implementation phase can cause a part of the application needed to be re-written, therefore the testers should be involved from the beginning (Thakur, n.d).

Prototyping is a popular design technique used for the identification of unknown system requirements. The designer can directly interact with the user to receive and verify requirements. With prototyping, a representation of the program can be provided (Thakur, n.d). This allows the user to provide quick feedback to the designer on the prototype and get a feeling on how the system operates and look like. Cornell (2018), strengthens on Thakur (n.d) statements, in stating that an advantage of prototyping is the speed in which it operates and can be enhanced by rapid prototyping systems. The scope of his statements is based on the prototypes of physical products, but it is also applicable in designing software, because it has the same beneficial results.

Other design practices that can be used, includes, the insurance of less semantic errors, adapting to unfamiliar deviations, present software as a real-word occurrence, design for reusability and lastly a design must be integrated. Most of the time people think, that creating an application means that it must only be fully functional (Nazrul, 2017). Nazrul (2017) explains, that when we try to adapt the fully functional software application, we ran into all sorts of errors. Therefore, advanced design principles should be used in the development of a software system.

### Unit testing

Unit testing is the foundation of other tests, seen in **figure 4**. Unit testing involves testing smaller units of an application to ensure that the specific unit is functioning correctly (Bindu et al, 2018:11). Cheon and Leavens (2003:1), proclaims that unit testing can improve the overall quality of software. The testing is done by isolating the particular unit from the rest of your code. Unit testing judges the acquired results against expected outcomes (Cheon et al, 2013:3). It is important to understand that not every test performed can be judged as a unit test (Dietrich, 2017). If code is written for testing purposes that could potentially fail, you have not done unit testing. A test that randomly generated data cannot be considered as unit testing.

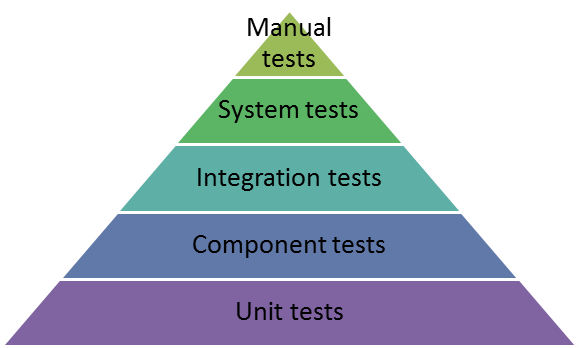


Figure 4 : Testing foundation (<https://bit.ly/2Aqa7tg>)

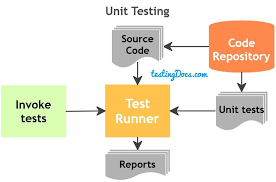


Figure 5 : Unit testing framework (<https://bit.ly/2Lu56VT> )

A unit test is different from other tests. Unit testing must comply with the FIRST principle (Newton, 2017). FIRST means, fast, independent, repeatable, self-validating and timely. Unit testing must be fast. If the test is fast, people won’t be put-off from utilizing the test. Different tests should not depend on each other. Gupta (2016) states, that tests should be isolated from each other and never depend on each other. Newton (2017) supports Gupta, by stating that tests should not be dependent on previously handled tests. By isolating tests from each other, a test can keep its focus on a small volume of behaviour. Tests must be repeatable, because if they are not, the tests that was utilized are not accurate. To make a test repeatable, the external environment except databases should be isolated from the test. Good tests can fail or pass unambiguously (Ottinger, n.d). Test should be self-validating and must establish if a test failed or passed (Gupta, 2016). No interpretation of outcomes should be done manually, because when outcomes are manually verified the test can take up a large amount of time and can increase dangers. The test must be timely. The test must be written in time before the production code is written. These five laws (F.I.R.S.T) will increase the robustness of unit testing.

There are a number of benefits an individual or company will gain when performing unit testing. A test-driven development(TDD) forces a programmer to program in modules. This will make code more understandable for novices. Test driven developments can identify problems in the chosen architecture. It also allows for the documentation of code (Ghahrai, 2017). Hiil(2015) states, that this type of documentation can be thought of, as a kind of live documentation. This ensures that code is kept up to date. Code can also be more easily refactored, because of well-defined individual modules (Ghahrai, 2017). Unit testing also increases communication and collaboration between participating members and unit testing helps to clear up missing requirements through inputs and outputs.

There are also drawbacks when performing unit testing. Firstly, unit tests are not always easy to write, because of the complexity of some unit tests (Hill, 2015). Ghahrai (2017), also states that unit tests are difficult to write. Unit testing is essential but will ultimately slow down the development process (Hill, 2015). Unit testing will not guarantee an error free application, because it only tests the functionality of isolated modules. Performance errors cannot be recognized by unit testing. Despite all the drawbacks, unit testing is here to stay and must not be underestimated, because it can greatly reduce the development cost of a company.

## Conclusion

It was clear that all these topics are interconnected. The central theme revolved around the development or engineering of software that must be adaptable for future requirements. The topics build on each other and each one of them must be followed for efficient and effective development of software. It describes the laws of good software development and how to transform these laws into good code. Debugging is a topic that also could’ve been included, because it is part of software development, but all the major concepts was covered in the research and will help a novice software developer to transform into a well-rounded developer.

# Heading 1 won’t print. Don’t delete – doing so will lead to incorrect numbering.

**References**

BARBARY, K. 2016. *The Joy of Code Refactoring* [Online]. Available: https://bids.berkeley.edu/news/joy-code-refactoring [Accessed 27 Jul 2018].

BINDU, S, BASEER, K & RAVIKANTH. 2018. OPTIMIZING THE CODE COVERAGE BY CONTROLLING ENVIRONMENTAL DEPENDENCIES IN UNIT TESTING.

CHEON, Y & LEAVENS, D. 2003. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way, 1-3.

CHHATPAR, A. 2007. *Using architecture patterns.* https://www.ibm.com/developerworks/rational/tutorials/ar-designpat2/index.html [Accessed 26 Jul 2018].

DIETRICH, E. 2017. *You Still Don’t Know How to Do Unit Testing (and Your Secret is Safe with Me).*

ELIASON, K. 2013. *DIFFERENCE BETWEEN OBJECT-ORIENTED PROGRAMMING AND PROCEDURAL PROGRAMMING*.

ERSHAD, G. M. 2017. Pros and Cons of Code Refactoring.

FOWLER, M. 1999. Refactoring: Improving the Design of Existing Code, xvi.

GHAHRAI, A. 2017. *Pros and Cons of Test Driven Development.*

GOODRICH, G. 2017. *Understanding the Model-View-Controller (MVC) Architecture in Rails.*

GUPTA, L. 2016. *FIRST Principles for Writing Good Unit Tests.*

HYDERABAD, C n.d., *PEER TO PEER NETWORK* [Online]. Available: http://infosecawareness.in/peer-to-peer-network [Accessed 27 Jul 2018].

KELMENDI, D. 2017. *SOLID Principles made easy.*

KUMAR, A. 2018. *WHAT IS MVC ARCHITECTURE?*

MALLAWAARACHCHI, V. 2017. *10 Common Software Architectural Patterns in a nutshell*.

MANTU KUMAR, N. B. 2012. CENTRALIZED HOST CONTROL, 1021.

SYSTEM

MARTIN, R. C. 2000. Design Principles and Design Patterns, 8.

MCGEE, M. 2018. *What Are Modular Components?*

MICROSOFT n.d., *Extract Method Refactoring (C#)* [Online]. Available: https://msdn.microsoft.com/en-us/library/0s21cwxk.aspx [Accessed 27 Jul 2018].

MUTHUPALANIAPPAN, K. 2015. Refactoring – Techniques and Tips.

NAZRUL, S. S. 2017. *Software Development Design.*

NEWTON, D. 2017. Writing Your F.I.R.S.T Unit Tests.

NOVOSELTSEVA, E. 2018. Code Refactoring Techniques.

OTTINGER, T n.d., Unit Test Are FIRST. *The Pragmatic Bookshelf.*

REHMAN, J. 2017. *Advantages and disadvantages of network operating.*

RICHARDS, M. 2015. Software Architecture Patterns, 1-19.

ROUSE, M n.d., *event-driven architecture (EDA).*

SEGUE. 2013. *Software Requirements Take Different Forms for Different Aspects of Development* [Online]. Available: https://www.seguetech.com/software-requirements-development/ [Accessed 28 Jul 2018].

SILVA, D. 2010. Evaluation of the impact of code refactoring on embedded software efficiency, 145.

STAR, T. 2018. *Principles of Good Software Design.*

THAKUR, D n.d., *Principles of Software Design & Concepts in Software Engineering.*

TREK, H. 2018. Software Design Principles.

ZILBERFELD, G. 2012. Design for Testability – The True Story.