Research Report

DRAFT

Jaco Sittig

27483533

**Table of Contents**

[**Introduction** 2](#_Toc520492186)

[**Architecture Patterns** 3](#_Toc520492187)

[Layered Architecture 3](#_Toc520492188)

[Broker Architecture Pattern 4](#_Toc520492189)

[Microkernel Architecture 4](#_Toc520492190)

[Pipe-filter pattern 5](#_Toc520492191)

[MVC (Model View Controller) pattern 6](#_Toc520492192)

[Code refactoring 7](#_Toc520492193)

[References 10](#_Toc520492194)

# **Introduction**

It is commonly known that there is a big shortage of IT/Software developers in South Africa. Various questions can be asked about this subject, but the most common question is: Why is there such a big shortage, especially if this is the digital and computing era? The answer to most of these questions, is skill, or rather the lack thereof. Computer Science and Information technology students have a lack of skills when they are graduate ([Bateman, 2014](#_ENREF_3)). Universities cannot keep up with the needs of enterprise IT companies. Most of the “soft skills” like communication skills (verbal and writing), professionalism, presentation skills and critical thinking are being taught at universities in various subjects. The technical skills that are taught in IT modules are mostly basic and teach the students some key values. These skills are not enough for the IT industry because they expect IT graduates to be educated in projects that are professionally relevant for today’s technology([Liebenberg *et al.*, 2014](#_ENREF_12)).

Universities have limited time to teach students all the fundamentals needed for the industry, that is why this report will be about understanding the concepts regarding the following advanced skills (Architectural Patterns, Code Re-factoring, Advanced Design Principles and Unit Testing). By reading this report and understanding the basic fundamentals of each skill, graduates will have an advantage when starting to develop software and encounter these skills.

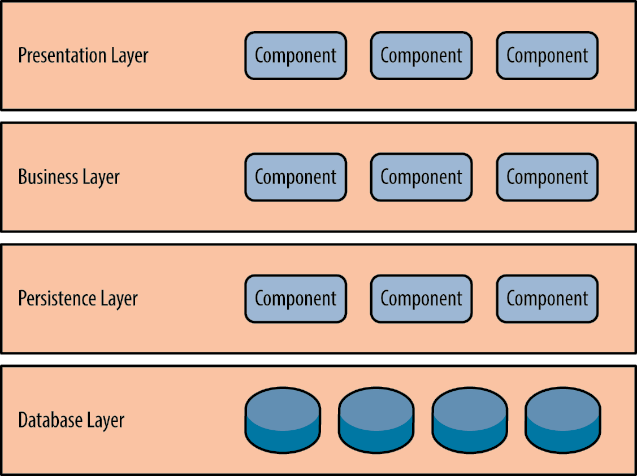
# **Architecture Patterns**

An Architectural Pattern is a proven structural organization schema for software systems ([OU, 2015:36](#_ENREF_21)). Richards ([2015:7](#_ENREF_23)) assert that “Architecture Patterns help define the basic characteristics and behavior of an application”. Before starting with a new program or system, the developer must first decide which architectural pattern to use for this specific program/system. Programs that does not include a well thought out architectural pattern can suffer from unorganized source code, unclear relationships, difficulty to change and is generally not reusable. Programs are not limited to only one pattern, different patterns can be used in sections of code for maximum optimization. A good programmer should know most of the patterns to decide which pattern is most suitable for the program/ system.

There are many architectural patterns, each with different characteristics, pros and cons. For this report I will be distinguishing between 5 different patterns.

## Layered Architecture

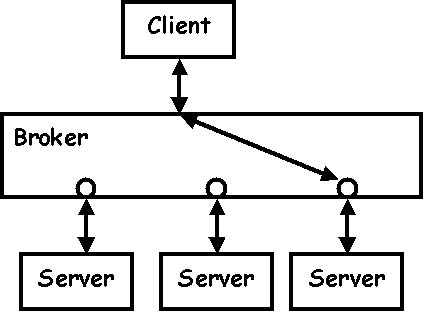
Layered Architecture is the most commonly used architecture in programs and is also known as the n-tier pattern. This pattern helps to structure programs that can be decomposed into tasks([OU, 2015:37](#_ENREF_21)). Developers that does not know what specific architecture to use in their system will use this pattern. It is also used in systems that needs a database to store information. The components in this pattern are organized into layers. Layered architecture must have, but is not limited to, at least four layers (presentation, business, persistence and database) with each completing a specific task. The layers are arranged in an order so that information enters from the top and works down towards the database layer. Layered architecture makes code maintainable, testable, assign roles and update layers separately. Wrong implementation of this architecture can result in the program being slow, code can become unorganized and layers can be skipped.



*Layered Architectural Pattern*

## Broker Architecture Pattern

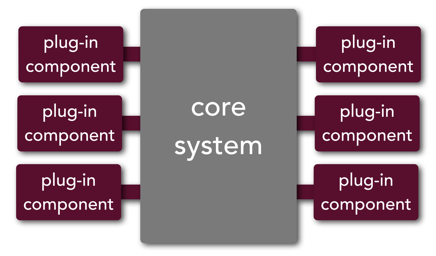
This architecture is used to structure distributed systems with decoupled components and interacts by means of remote service invocations ([OU, 2015:44](#_ENREF_21)). A Broker Pattern consists of two components, the broker component and event processor component ([Richards, 2015:14](#_ENREF_23)). The Broker component is central and contains client events. The event processor component is used for processing the events from the broker component. Advantages is that the component is easily adaptable, location transparency and components are developed independently. Disadvantages include, low fault tolerance and is difficult to test.



*Broker*

## Microkernel Architecture

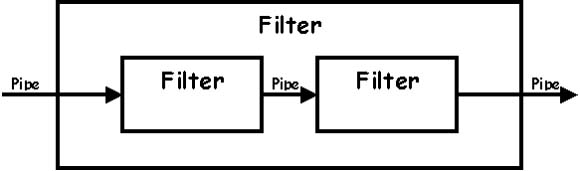
Richards ([2015:21](#_ENREF_23)) asserts that “the Microkernel is a natural pattern for implementing product-based application”. Microkernels are also implemented in many operating systems. This pattern makes use of two components, namely the core system and plugin modules. The core system keeps the system operational by means of minimal functionality. Plugin modules add extra features to the core system, adds specialized processing and business capabilities. With these two components in mind, the programmer can make different small programs that is easily separated and add it to the system, then every time that the system needs updates, the programmer can create another small program. Some disadvantages are that there can be dependency issues between plugins and having a performance overhead in multiple views of the system.



*Microkernel*

## Pipe-filter pattern

Avgeriou and Zdun ([2005](#_ENREF_2)) contends that “In a pipes and filters architecture a complex task is divided into several sequential subtasks”. This pattern is a structure that is used when data streams are produced by a system ([OU, 2015:42](#_ENREF_21)). As stated in the name of this pattern, it consists of two components, namely the Filter and Pipe. The filter is used to hold each processing step or task and the pipe is used to move the data between the filters. The data or information that has been processed by the filter is then used as input, delivered by a pipe, to the next filter as shown in the figure. With pipes and filters there is no need for intermediate files, filters can be replaced with ease and some filters can be programmed independently ([Edinburgh, 2006](#_ENREF_6)). Disadvantages for this pattern include difficulty sharing global data, with many filters and high data cost it can be expensive and problem solving errors can be difficult ([Edinburgh, 2006](#_ENREF_6)).



## MVC (Model View Controller) pattern

The MVC pattern is a widely used pattern in object-oriented systems with the aim of the pattern to keep the application core away from all the interfaces in the system ([Deacon, 2009:1](#_ENREF_4)). With the Model-View Controller an application/ system is split up into three components: the program core and data are stored in the model component; information that is displayed to the end user is stored in the view component and then all input gathered from the user is stored in the controller component (Open University IM0203: 47). The MVC can be implemented by means of three abstract superclasses (Model, View and Controller) with each having many concrete subclasses holding all the information and data ([Krasner & Pope, 1998:5](#_ENREF_11)). This specific pattern is a good way for programmers to decouple components and allow code to be reused in an efficient capacity. Making changes to the interface without effecting any other components and view synchronization can be seen as advantages of the MVC pattern. Disadvantages for this pattern is that it can be too complex for simple problems and that the views and controllers are not modular ([Edinburgh, 2006](#_ENREF_6)).



*MVC*

# Code refactoring

([Fowler *et al.*, 1999:9](#_ENREF_8)) asserts: “Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure.” For a simpler approach a definition can also be that code refactoring means that the programmer changes code to be more organized, readable and easy to maintain. Refactoring can also improve software to be more reliable, better performance capabilities and finally make code more viable ([Park *et al.*, 2014:1](#_ENREF_22)). Code refactoring is an important skill to learn for programmers and software developers because without the knowledge of refactoring, the code that the programmer does not refactor, can rot (slow deterioration).

When code rots it can be in the form of bad dependencies between packages and classes, classes that has to perform too many responsibilities and duplicate code. Without refactoring, these rotten code just becomes worse and worse and in the end it will make the system unusable. Other reason to make use of refactoring is that it can improve the design of software because when code gets changed to accommodate new features, some of the code becomes structure less and difficult to read and understand ([Fowler *et al.*, 1999:47](#_ENREF_8)). Refactoring can be used to find bugs. When refactoring code, programmers can better understand what the code does and should do for the system and therefore better spot potential bugs ([Fowler *et al.*, 1999:48](#_ENREF_8)).

Code refactoring does not have a specific time to be done in the development life cycle. Knowing when to refactor is just as important as knowing how to refactor. There are several ways to know when it is time to refactor code. A good rule to use for knowing when to refactor is the rule of three, the rule states that by the third time that you code something similar, you should refactor ([Roberts, 1999](#_ENREF_24)). Another good way of knowing when to refactor is when you want to speed up with the task that you are currently busy with. When adding a new feature to a program is when you can refactor the previous code, to better understand the code and change the code if it causes problems when adding the new feature ([Fowler *et al.*, 1999:50](#_ENREF_8)). Another reason to refactor your code is when you need to dig deeper in your code to find and fix bugs.

Code smells are a good indication that there can be something wrong with the code and it needs to be refactored. There is a lot of different types of smells that exists, see table 1-1.

*Table 1-1*

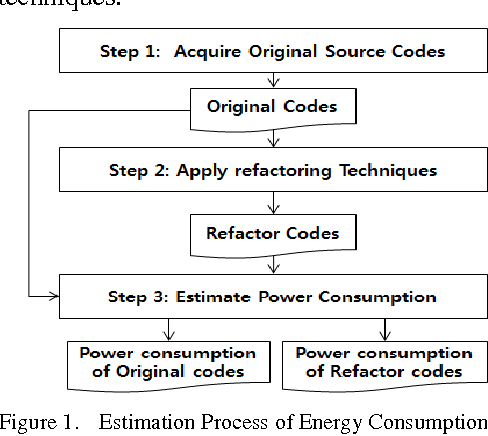
|  |  |
| --- | --- |
| ([Fowler *et al.*, 1999](#_ENREF_8)) | ([Van Deursen *et al.*, 2001](#_ENREF_28)) |
| Duplicate code | **Mystery guests** |
| Comments | **Eager tests** |
| Large classes | **Resource optimism** |
| Too many parameters |  |
| Data clumps |  |

Duplicate code is code structures that is repeated in multiple places. Duplicate code is the first thing that is mentioned when refactoring ([Head, 2015](#_ENREF_9)). Large classes that has too much to do can create duplicate code. Too many parameters can cause inconsistencies in data. Data clumps are the same types of data items that group together. Comments can be used to hide bad written code. A mystery guest is when external sources are used to test the system, the external sources cannot have all the needed information resulting in bad testing. When too many tests are performed on methods and the results does not make sense, it is called eager tests. Resource optimism is the result of optimistic tests about the state of external sources. These are just a few code smells that can indicate to a programmer that refactoring is needed.

Refactoring must be done in such a way that it does not break and introduce new non-existing problems or bugs into the system. There are many different methods to refactor specific code problems, in this research it includes composing methods, organizing data, generalization, moving features between objects and simplifying method calls.

Composing methods are used when methods are too long. These methods can be a hazard because the method can contain a lot of important information that are being hidden by the complex execution logic ([Fowler *et al.*, 1999:89](#_ENREF_8)). To use the composing methods refactoring technique on methods that are too large, is to take code that can be grouped together and make it into a new method. Organizing data is an important refactoring technique to improve data handling, straightening out class associations (makes a class reusable) and resolving data structure issues([Anon, 2007](#_ENREF_1)). Generalization can make its own refactoring techniques that are used when moving methods around hierarchies and inheritance ([Fowler *et al.*, 1999:259](#_ENREF_8)). Some of the techniques that generalization makes, include pull up fields and methods, replacing the inheritance with delegation and extracting superclasses.

An essential decision that has to be made when doing object designing is to decide where to place the responsibilities ([Fowler *et al.*, 1999:115](#_ENREF_8)). Moving features between classes can be used when classes become overcrowded with methods and responsibilities. By moving methods and fields, extracting classes and removing the middle man, the moving features between objects technique can be achieved. Changing the name of methods and variables can make a huge difference and is easy ([Head, 2015](#_ENREF_9)). Simplifying method calls are used to make method calls easier to understand and also simplify the interfaces when interaction is needed between classes.



# Advanced Design Principles

According to Nierstrasz ([1990](#_ENREF_17)) “object-oriented programming is a paradigm for organizing software into reusable components that combine data structure and the procedures that can be used to manipulate them”. Object-oriented design is used to define and plan how objects will interact with each other to solve a problem ([Model *et al.*, 1991:1](#_ENREF_15)). Advanced design principles has an effect on both the OOP and OOD. When applying design principles to software development it will enhance the programmer’s skill to maintain code easier, avoid code smells, refactor and apply agile development ([Oloruntoba, 2016](#_ENREF_19)). Two different principles, S.O.L.I.D and GRASP, will be discussed in this research report. These principles are not rules, they are only advice for common problems that will improve software development ([Martin, 2002](#_ENREF_14)).

SOLID is a widely used design principle in object-oriented software development that was created by Robert C. Martin (Janssen, 2018). See table 1-2 for each type of principle used in the SOLID acronym.

## Single-responsibility principle

According to Martin ([2002:155](#_ENREF_14)) “A class should have one and only one reason to change, meaning that a class should only have one job”. Using the single-responsibility principle (SRP) will make software implementation easier and avoid future failures for changes and updates.

To use the SRP a programmer should define responsibility for each class, separate coupled responsibilities into different classes and separate persistence control. If a class has more than one responsibility, it should be clarified if separation is needed. SRP is a simple principle but it can be difficult to separate responsibilities ([Martin, 2002:162](#_ENREF_14)).

## Open-closed principle

Martin ([2002:165](#_ENREF_14)) contends that “objects or entities should be open for extension, but closed for modification”. The open-closed principle is used so that a class can be extended without modifying the class itself ([Oloruntoba, 2016](#_ENREF_19)).

Two methods that are used in the open-closed principle is inheritance or composition.([Umair, 2018](#_ENREF_27)). This principle should not be used on all the code in a program, it should be used on code that can change in future updates. If the open-closed principle is used correctly then the program will be flexible, maintained and reusable.

## Liskov substitution principle

Martin ([2002:180](#_ENREF_14)) asserts that for the Liskov substitution that you should “let q(x) be a property about objects of x of type T. Then q(y) should be provable for objects y of type S where S is a subtype of T”. This is a very difficult and confusing definition so to make it more understandable, all that one needs to know is that subclasses should always be the substitute for their original parent class ([Oloruntoba, 2016](#_ENREF_19)).

Without the Liskov substitution principle, class hierarchies would be difficult to understand. When the Liskov substitution principle is applied correctly then subclasses will be extensible without modification ([Martin, 2002:199](#_ENREF_14)).

## Interface segregation principle

A client should not be forced to implement an interface or methods that it does not use ([Oloruntoba, 2016](#_ENREF_19)). If a class has many clients, then the clients should be created in separate specific interfaces ([Martin, 2000:14](#_ENREF_13)). The interface segregation principle knows that some of the classes need non-cohesive interfaces ([Martin, 2002:214](#_ENREF_14)). The interface segregation principle makes that clients can only depend on the methods that are called. All clients have specific types and can be organized according to those types. After the clients have been organized, interfaces can be made for each type of client.

## Dependency inversion principle

Martins ([2000:12](#_ENREF_13)) contends that “dependency inversion is the strategy of depending upon interfaces or abstract functions and classes, rather than upon concrete functions and classes”. High level modules should not relay on low level modules, because high level modules consists of the most import information and business models. High level modules are also the modules that are more reused than low level modules ([Martin, 2002:202](#_ENREF_14)). The dependency inversion principle is mostly used when one class interacts with another class. This is a principle is a necessity for programmers that wants to create a reusable framework.

S.O.L.I.D is a very good and common way to understand what type of principles to use in certain situations, but there are other types of principles that exists for example GRASP. GRASP is an acronym that means General Responsibility Assignment Software Patterns and was developed by Craig Larman([Model *et al.*, 1991:39](#_ENREF_15)). See table 1-2 for all nine grasp patterns. These principles, either S.O.L.I.D or GRASP, is difficult for beginner programmers but to be able to master these principles, one needs to practice and learn from mistakes.

|  |  |
| --- | --- |
| S.O.L.I.D | GRASP |
| Single-responsibility principle | **Creator** |
| Open-closed principle | **Information Expert** |
| Liskov substitution principle | **Controller** |
| Interface segregation principle | **Low coupling** |
| Dependency Inversion principle | **High cohesion** |
|  | **Pure fabrication** |
|  | **Indirection** |
|  | **Polymorphism** |
|  | **Protected variations** |

# Unit Testing

Osherove ([2009:4](#_ENREF_20)) asserts that “a unit test is a piece of code that invokes a unit of work and checks one specific end result of that unit of work, if the assumptions at the end is wrong then the unit test has failed”. Another way to understand what a unit test is, is to see it as a test that is done to check that the program meets its requirements ([Koomen & Pol, 1999](#_ENREF_10)). All the different definitions come back to one point and that is that unit testing is a technical process that uses input and output parameters ([Runeson, 2006:24](#_ENREF_25)). Unit testing is not one big test, rather small tests that use one or two inputs and receives one output. Unit testing can be applied to methods, procedures and classes.

Most programmers don’t really understand why they should use unit testing, they believe it is just more work to be completed in such a short deadline. Unit testing have several benefits but the two most import uses is that it will improve the design of your code and make automated regression tests ([Sonmez, 2017](#_ENREF_26)). When saying that unit testing will improve the design of the code, it means that when the programmer is looking at all the small units of code, he can easily find problems. By making automated regression tests, it will increase the effectiveness of testing with high level code structures.

Some other benefits of unit testing can include finding bugs early, reduce costs, provide documentation and simplify integration ([Novoseltseva, 2017](#_ENREF_18)). By testing code early and in small units, bugs are found and can be corrected before it causes major problems. Now that the bugs are found and solved it will reduce the costs that would have been higher in later stages. By understanding what the function is of each unit of code is, it will improve the documentation for the developers unit interface. To help with system integrations, unit testing ensures that when code or libraries are updated, they will still work in the correct way. In a short survey done by ([Runeson, 2006:25](#_ENREF_25)), he asked developers why unit tests are conducted. He then gave the reason as: to ensure that unit functions are as expected, accept units from other sources and to improve the quality. The results show that to improve the quality was the most agreed with reason ([Runeson, 2006:26](#_ENREF_25)).

Before using unit tests, it must first be written using some core techniques. The core techniques consists of how stubs can be used to break dependencies, interaction testing using mock objects and isolation frameworks to stop repetitive code ([Osherove, 2009:47](#_ENREF_20)).

Osherove ([2009:49](#_ENREF_20)) contends that “a stub is a controllable replacement for an existing dependency in the system”. Stubs are used to test a unit without having to worry about the dependency. More than one stub can be used in a test. Mocks are used to capture information during object interaction, then test this information to check if the interaction has failed or not ([Mosalla, 2017](#_ENREF_16)). A key difference between choosing whether to use a mock or stub, is that a stub is used because it cannot fail a test but a mock can. Mocks use expectations that are pre-programmed into them([Fowler, 2007](#_ENREF_7)). If an implementation should be tested and there is no other way then mock should be used. Osherove ([2009:91](#_ENREF_20)) contends that “an isolation framework is a set of programmable API’s that make creating fake objects much simple, faster and shorter than hand coding them”. Isolation frameworks create a bunch of fake objects to set behaviors on them and wait for this fake object to be called and verified. These are just a few examples of what an isolation framework does, see *figure* for more.

Two types of unit testing are used, manual or automated testing. Manual testing are done by the developer without using any tools or scripts and automated testing are the opposite because it only uses tools and scripts ([Donaldson, 2017](#_ENREF_5)). Each type of testing have advantages and disadvantages, see *table* for some of the advantages and disadvantages. Manual testing should be done when the developer wants to self-explore because of the lack of specifications, check how user-friendly the software is and when ad-hoc testing is needed. Automated testing is preferred in certain scenarios where regression, load and performance testing’s are needed.

Unit testing is an important skill to have because without unit testing, integration can have many bugs and problems. Unit testing will be difficult for new programmers to understand but once they write a few tests and see the benefits that it holds, it will become easier and enjoyable.

|  |  |
| --- | --- |
| Manual testing | Automated testing |
| Not accurate | **Reliable** |
| Time consuming | **Done by tools and scripts** |
| Needs human resources | **When used more than twice** |
| Used once or twice | **Not user friendly** |
| Human observation is useful |  |

# Conclusion

With programming technology advancing every day and thousands of IT students graduating each year without all the skills needed for the job market, a logical conclusion is that universities should reevaluate their curriculum. Until reevaluation are done graduates can, as a solution, understand and practice the architectural patterns, code refactoring, unit testing and advance design patterns that are researched in this paper.

# References

Anon. 2007. Refactoring techniques. https://sourcemaking.com/refactoring/refactorings Date of access: 25 Jul 2018.

Avgeriou, P. & Zdun, U. 2005. Architectural patterns revisited-a pattern language.

Bateman, K. 2014. The irony of an unemployment problem and an IT skills shortage within the IT industry, ComputerWeekly. com, Oct.

Deacon, J. 2009. Model-view-controller (mvc) architecture. *Online][Citado em: 10 de março de 2006.] http://www. jdl. co. uk/briefings/MVC. pdf*.

Donaldson, W. 2017. What's the Difference Between Automated Testing and Manual Testing? https://dzone.com/articles/automated-testing-vs-manual-testing Date of access: 29 Jul 2018.

Edinburgh, U.o. 2006. Architectural Patterns: SAPM: University of Edinburgh http://www.inf.ed.ac.uk/teaching/courses/sapm/2005-2006/slides/architecture\_4up.pdf Date of access: 22 Jul 2018.

Fowler, M. 2007. Mocks aren’t stubs. *Martin Fowler Articles* 6.

Fowler, M., Beck, K., Brant, J., Opdyke, W. & Roberts, D. 1999. Refactoring: improving the design of existing code: Addison-Wesley Professional.

Head, J. 2015. Intro to Refactoring: Making Code Easy to Understand in 4 Steps https://spin.atomicobject.com/2015/03/02/code-refactoring/ Date of access: 27 Jul 2018.

Koomen, T. & Pol, M. 1999. Test Process Improvement. *A practical step-by-step guide to structured testing: Addison-Wesley*.

Krasner, G. & Pope, S. 1998. A cookbook for using the model - view controller user interface paradigm in Smalltalk - 80: Journal of Object-oriented Programming. https://www.researchgate.net/publication/244954334\_A\_cookbook\_for\_using\_the\_model-view\_controller\_user\_interface\_paradigm\_in\_Smalltalk-80 Date of access: 26 Jul 2018.

Liebenberg, J., Huisman, M. & Mentz, E. 2014. Knowledge and Skills Requirements for Software Developer Students. *International Journal of Social, Behavioral, Educational, Economic, Business and Industrial Engineering*, 8(8):2604-2609.

Martin, R.C. 2000. Design principles and design patterns. *Object Mentor*, 1(34):597.

Martin, R.C. 2002. Agile software development: principles, patterns, and practices: Prentice Hall.

Model, W., Model, S. & Model, I. 1991. Object-oriented design.

Mosalla, H. 2017. Interaction Testing With Mocks Leads To A Less Maintainable Unit Test. http://hamidmosalla.com/2017/11/28/interaction-testing-with-mocks-leads-to-a-less-maintainable-unit-test/ Date of access: 29 Jul 2018.

Nierstrasz, O. 1990. Object-oriented design principles.

Novoseltseva, E. 2017. 8 Benefits of Unit Testing. https://dzone.com/articles/top-8-benefits-of-unit-testing Date of access: 29 Jul 2018.

Oloruntoba, S. 2016. SOLID: The First 5 Principles of Object Oriented Design. *Scotch. io*.

Osherove, R. 2009. The art of unit testing manning Publications Co. http://accorsi.net/docs/TheArtofUnitTesting.pdf Date of access: 29 Jul 2018.

OU. 2015. IM0203: Architectural patterns: Open University https://www.ou.nl/documents/40554/349790/IM0203\_03.pdf Date of access: 22 Jul 2018.

Park, J.J., Hong, J.-E. & Lee, S.-H. 2014. Investigation for Software Power Consumption of Code Refactoring Techniques*.* (*In*. SEKE organised by. p. 717-722).

Richards, M. 2015. Software architecture patterns: O'Reilly Media, Incorporated.

Roberts, D. 1999. Refactoring: Improving the Design of Existing Code.

Runeson, P. 2006. A survey of unit testing practices. *IEEE software*, 23(4):22-29.

Sonmez, J. 2017. What is TDD? What is Unit Testing? https://simpleprogrammer.com/tdd-unit-testing/ Date of access: 29 Jul 2018.

Umair, M. 2018. SOLID, GRASP, and Other Basic Principles of Object-Oriented Design. https://dzone.com/articles/solid-grasp-and-other-basic-principles-of-object-o Date of access: 28 Jul 2018.

Van Deursen, A., Moonen, L., Van Den Bergh, A. & Kok, G. 2001. Refactoring test code*.* (*In*. Proceedings of the 2nd international conference on extreme programming and flexible processes in software engineering (XP2001) organised by. p. 92-95).