AUT Journal of Mathematics and Computing

AUT J. Math. Com., 2(1) (2021) 81-102
DOI: 10.22060/ajmc.2021.19232.1044

Original Article

Ensuring software maintainability at software architecture level using architectural patterns

Zahed Rahmati[a], Mohammad Tanhaei*[b]

[a]*Department of Mathematics and Computer Science, Amirkabir University of Technology (Tehran Polytechnic), Tehran, Iran*
[b]*Department of Engineering, Ilam University*

ABSTRACT: Software architecture is known to be an effective tool with regards to improving software quality attributes. Many quality attributes such as maintainability are architecture dependent, and as such, using an appropriate architecture is essential in providing a sound foundation for the development of highly maintainable software systems. An effective way to produce a well-built architecture is to utilize standard architectural patterns. Although the use of a particular architectural pattern cannot have a preserving effect on software maintainability, the mere conformance of a system to any architecture cannot guarantee the system's high maintainability. The use of an inappropriate architecture can seriously undermine software maintainability at lower levels. In this article, the effect of standard architectural patterns on software maintainability quality attributes is investigated. We develop a quality model for maintainability quality attributes, which is later used to compare various standard architectural patterns. We finish by investigating two real-world experiences regarding the application of a particular pattern to two different existing architectures, exploring the effect of the change in architecture on maintainability quality attributes.

Review History:
Received:07 November 2020
Accepted:28 January 2021
Available Online:01 February 2021

Keywords:
Patterns
Software architecture
Maintainability

AMS Subject Classification (2010):
68N01; 68N99

1. Introduction

Research conducted on software-development life-cycle has revealed that the maintenance phase is very costly [37, 18]. By software maintenance, we mean the modification of each of the software artifacts after delivery [18]. Software artifacts include program code, development and testing documentations, and architecture among others. Variations in the delivered software are acceptable as so many requirements of customers become known only after using primary versions. Moreover, external obligations such as variations in technology, environment, rules, etc., necessitate the modification of software. Approaches that are used to deal with modification requests for software are divided into two general groups:

- **Adventure Approach:** In this method, change requests are dealt with instantly and improvisationally. There is no prior arrangement for changing the software. It is advantageous in that it does not need any prior investment for managing changes. However, it suffers from unpredictability of cost and time.
- **Planned Approach:** In this approach, preparations are made for encountering changes. Existing risks are managed and needed cost and time are estimated. In this approach, the first criterion is carefulness, and adventuring is avoided. Obviously, not all risks can be foreseen; it is simply not economically (an otherwise)

*Corresponding author.
E-mail addresses: zrahmati@aut.ac.ir, m.tanhaei@ilam.ac.ir

feasible. Methods in planned approach differ in their level of risk management and the amount of details they take into account.

Maintainability is more important in huge projects and systems than small ones. Larger systems have a greater number of users and more variety in system types, communications, and infrastructures. There are naturally many changes in these kinds of projects; thus, the adventure approach will lead to failure. So one has to plan ahead as regards to managing changes. Planning to increase maintainability can be performed in different categories [7]. These categories differ in the amount of details they contain.

- **Code Level:** The first and simplest solution for increasing software maintainability is considering this quality attribute at the code level. Patterns for increasing maintainability are defined in Martin Fowler's book; "Code Refactoring" [13]. Various techniques such as code abstraction, breaking the code apart and improving names are discussed within this book. Improving maintainability at the code level is usually relatively inexpensive and needs only a short time.
- **Design level:** The second level is changing software at the design level. At this level, the relations among classes and objects are investigated, and some changes are applied according to the well-known patterns [34]. For an instance, circular dependency among a group of classes decreases maintainability of software, as it makes editing classes and adding functions more difficult. This kind of dependency can be avoided by using interface classes.
- **Architecture level:** The highest planning level for software maintainability is the architecture level. Patterns which are either non-applicable to lower levels or expensive to be implemented at those levels are considered at this level. At the architecture level, there is no force to implement the system to find the quality of the software. This characteristic distinguishes this level from two others. At the architecture level, maintainability includes two distinct aspects:
  - **No necessity for system implementation:** To perform maintainability improvement patterns, in the two previous levels, system implementation is necessary; whereas performing maintainability patterns at the architecture level is possible without system implementation. This in turn will reduce the risk of the projects.
  - **Coarse-grained patterns:** Architecture level patterns are coarse-grained [16]. Their effect is not limited and various components are influenced by them. Its executive cost in implemented projects can be mentioned as one of its drawbacks. Consider a system which is formed based on a wrong architecture. Even a small change in architecture may cause the propagation of changes all over the implementation and increase the risk of the project. Nevertheless, it has some advantages:
    - *In systems which are not yet implemented:* Applying these patterns at early stages will reduce costs regarding maintenance.
    - *Implemented systems:* Detecting coarse-grained patterns in the code is difficult and referring to architecture is inevitable. Of course, some solution has to be found to deal with cost and time.

Our main aim in this paper is to investigate popular architecture pattern and specify which of them can facilitate software maintainability. For this purpose we first, compare existing models for quality attributes and propose a suitable model according to the existing quality models in Section 3. In selecting the model, we aim to omit a large part of sub-attributes mentioned in different references. After selecting a model, we search for factors corresponding to each sub-attribute and explain them in Section 3.2. Subsequently, we choose a range of frequently used architecture patterns and evaluate them considering the created model in Section 4. Finally, we present a usage of our work in Section 5. One of the applications of our work is to determine patterns suitable for performing maintainability enhancing refactoring on the software architecture. Two real-world architecture refactoring using maintainability-supporting architectural patterns presented in Section 5. In the last sections (Sections 6 and 7), related work, the aspects which are currently being investigated and future work are elaborated.

2. Background

2.1. Quality Models

Different researchers and organizations have proposed different quality models, which have differences and similarities. Comparing these models will help put together a generic model which includes properties of a good portion of existing models. In Table 1 well-known quality models regarding maintainability, are compared to each other. From among various suggestions, six sub-attributes which contain all well-known models are selected (attributes column of Table 1).

2.1.1. McCall's Quality Model

The main idea behind the McCall quality model is investigating the relation between external quality factors and product quality attributes [31]. It has three different perspectives for defining the quality of software products: product revision (ability to tolerate changes), product transition (adaptability to the new environments), and product operation (attributes related to its operation).

The McCall quality model has been used in large-scale military and aerospace projects. It was developed between 1976 and 1977 by US Air Force. It has a hierarchical structure. At the highest level, there are quality factors (quality attributes) and at the lowest level, there are metrics. These two levels are connected via various criteria.

2.1.2. Boehm's Quality Model

This model is an enhanced version of the McCall model [5]. In essence, it is focused on adding the maintainability quality attributes to the previous model. Moreover, some considerations regarding product's usability are added into the model.

Similar to the previous model, it has a hierarchical structure. The highest level includes quality attributes. Primary attributes and measures constitute its lower layers. This model was developed in 1987.

2.1.3. ISO/IEC 9126 Quality Model

Request for developing a standard to assess software quality attributes encouraged the ISO standards committee to create such a standard. In this standard, six main attributes are defined and the set of attributes corresponding to each one is determined for measurement [23]. One of its shortages is its lack of guidance for measuring quality aspects; while it is beneficial as it separates external and internal attributes of software. Defined layers in this model are as follows:

- Quality attributes
- Quality sub-characterisitcs
- Metrics

2.1.4. Other Quality Models

In addition to the aforementioned models, we may mention some other quality models. For example, +FURPS/FURPS used in IBM Rational software [17, 24], Dromey model, which focuses on the relation between quality attributes and sub-attributes [11, 10], SPICE model, which is prepared for evaluating software procedures (ISO/IEC 15504 standards), and some IEEE standards such as Std 730-1998.

3. A Quality Model for Maintainability

Surveying existing quality models provided us with proper perspective for choosing a suitable set of attributes, including different aspects of maintainability. We compared well-known models and chose the most prominent attributes. Table 1 is a comparative table between different models, which help us choose important attributes. Based on comparative table (Table 1), six generic attributes are chosen to describe maintainability in the hierarchical quality model. These include *analyzability*, *changeability*, *stability*, *testability*, *understandability* and *portability*. To clarify the definition of these attributes we explain each of them in this section.

- **Analyzability:** Ability to detect deficiency or source of failure in software or detecting its parts which should be modified [23]. In other words, it is a characteristic demonstrating how much effort is needed to detect deficiency, the source of failure and the part which should be modified [38].
- **Changeability:** Software product ability to support preparations for a specific change in it [23]. To rephrase, software characteristic which shows how much effort is needed to remove faults or to adapt to environment changes [38].
- **Stability:** Attributes of software concerning avoiding unexpected effects caused by modifications [23]. It shows the risk of unpredicted effects caused by modifications [38].
- **Testability:** Capability of software which enables us to validate it after modifications [23]. Besides, it shows the amount of effort needed to test software so that we can make sure it fulfills its defined requirements [37, 14]. Another definition of testability is as follows: simplicity of building and synthesizing successful test on a component or particular software [12].

- **Understandability:** This attribute is based on user recognition of software applications and how it should be used for a specific application and in specific circumstances [23]. That is to say, it states the amount of effort which is needed to recognize logical concepts of software and its applicability [38].
- **Portability:** Simplicity of migrating a software from one environment to another (for example, from Linux OS to Windows OS) [32].

Table 1. Comparison between 3 well-known quality models from maintainabilty perspective

Attribute in our model	ISO/IEC 9126-1 i [23]	McCall's quality model [31]	Boehm's Model [5]
Analysability	Analysability	-Consistency -Self-documentation -Modularity	-
Changeability	Changeability	Expandability	Modifiability
Stability	Stability	-	-
Testability	Testability	-Simplicity -Self-descriptiveness -Instrumentation	Testability
Understandability	-	-Conciseness -Simplicity	Understandability
Portability	-	Portability	-

Notice: In McCall model testability is considered as an independent quality attribute.

3.1. Architecture-Level Maintainability Quality Model

Similar to other quality attributes, software maintainability cannot be measured directly. Quality models have been created as a result of this property. Therefore, quality attributes are divided into quality sub-attributes and these sub-attributes are measured through metrics grouped in some factors. Dividing attributes to sub-attributes and defining factors to measure sub-attributes lead to a hierarchical structure. Figure 1 illustrates an example of hierarchy developed for this study. We want to measure software maintainability in architecture-level using this model.

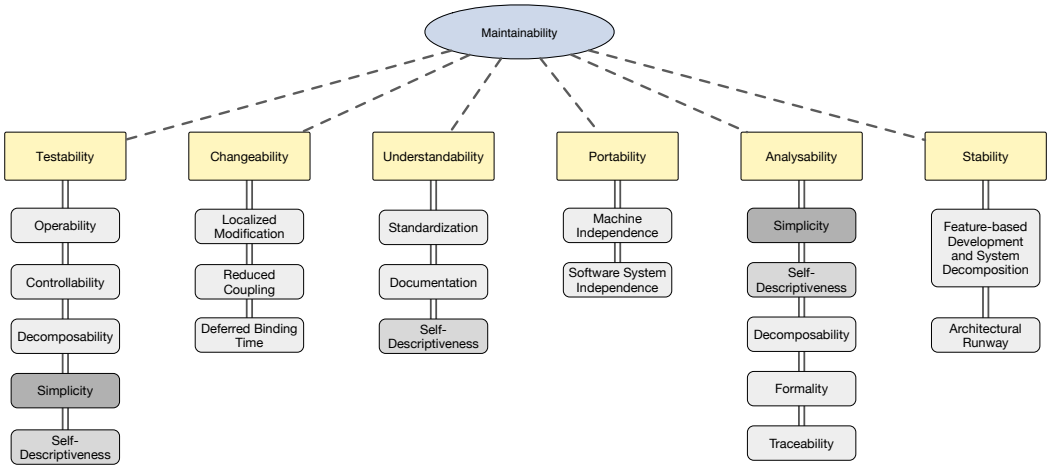

Fig 1. A Quality Model for Architecture Maintainability

3.2. Maintainability Factors

We discussed the most important attributes related to maintainability in previous section. One of the objectives of our study is to connect architectural patterns to maintainability quality attribute and its related-attributes (analyzability, changeability, stability, testability, understandability, and portability). A direct connection between architectural patterns and maintainability enables us to design highly maintainable software. The relation between

patterns and quality attributes will help the architect meet particular requirements. The challenge is that it is difficult to connect patterns to quality attributes directly.

The solution for relating an architectural pattern to the maintainability attribute in our work is using factors. We use maintainability factors to define the relationship between architectural patterns and maintainability related quality attributes. The factor measures the amount of improvements achieved in maintaining software code using a specific pattern. Each factor can belong to some quality sub-attributes. Improvement in the amount of factors, shows an improvement in their related sub-attributes.

Maintainability patterns are also helpful in detecting and describing maintainability requirements. It is hard to define attributes related to maintainability without defining requirements of this field. For instance, in analyzing requirements of a customer, we may conclude that system should have high changeability. How is this requirement defined? (For example, Modifications must be carried out in 10 hours). Will this definition help architecture design? Can architect detect related patterns based on these types of requirements? How is it possible to determine the time needed for modifications in a system which is not implemented?

We have no tenacious metrics to measure some of factors we introduce in our model. However, the value for some of the factor is fuzzy and subjective. In this study, we present a series of factors for each maintainability-related quality attributes in Tables 2 to 7. We did our best to choose the most essential and influential factors.

Table 2. Testability Factors

#	Factor	Definitions
T1	Operability	The more optimum system operation is, the more simple testing is achieved
T2	Controllability	As our control over software increases, we are able to do more tests.
T3	Decomposability	The problem can be isolated by controlling test scope. Consequently smarter tests may be accomplished to detect problem.
T4	Simplicity	The more simple software is the faster test is performed. Because number of tests is reduced.
T5	Self-Descriptiveness	Adding comments, process pre-explanations, valid and meaningful names and so on, will increase understandability of artifact. Furthermore, test process becomes simple as the artifact and its IO's (input/outputs) are better understood.

Table 3. Changeability Factors

#	Factor	Definitions
C1	Localized Modification (Increased Cohesion)	Narrowing modification scope reduces its cost [25].
C2	Reduced Coupling	Avoiding coupling decreases modification range and prevent it from affecting subsystems and non-related components.
C3	Deferred Binding Time	Deferring binding time enables users to modify software and it also provides possibility of late decision making about part of system.

Table 4. Understandability Factors

#	Factor	Definitions
U1	Standardization	Using standards in each artifact reduces learning time and increases understandability.
U2	Documentation	Using peripheral documents, dictionary and etc, increases understandability of software artifacts.
U3	Self-Descriptiveness	As the software includes more comments and inside artifact explanations, it is more understandable.

Table 5. Portability Factors

#	Factor	Definitions
P1	Machine Independence	Hardware abstraction level increases portability. This technique uses standard middleware.
P2	Software System Independence	Syntehsizing system based on environment independent methods increase portability. For instance using COBRA, JAVA Beans.

Table 6. Analyzability Factors

#	Factor	Definitions
A1	Simplicity	More simple architecture provides higher analyzing capability. A sophisticated architecture leads to complicated development and lower analyzability.
A2	Self-Descriptiveness	Detailed explanations inside architecture documentations result in better understanding of architecture. Accordingly it would be possible to more properly investigate and analyze the architecture. Obviously, suitable recognition of architecture leads to increase in analyzability of software developed based on that architecture.
A3	Decomposability	Modular architecture design of software modifies its analyzability.
A4	Formality	The more formal architecture is defined, the more analyzability is achieved [6].
A5	Traceability	When traceability of architecture is high, more analysis can be performed [6].

Table 7. Stability Factors

#	Factor	Definitions
S1	Feature-based Development and System Decomposition	Feature-based development reduces risks of unexpected fault effects [1].
S2	Architectural Runway	Architectural runway will help predicting next phases of development and detecting its architecture. This case is managed by agile software development methods and software product line.

4. Maintainability-Supporting Architectural Patterns

4.1. Patterns

Patterns are the reusable solutions for reoccurring problems [43]. They might be used at low level development phase such as implementation; they can be utilized at intermediate levels, e.g. design patterns, and they may also be used at higher levels, i.e. at the architecture level. Architectural styles are a specific sort of patterns applied at architecture level. They include an approach for defining architecture components and restrictions regarding the relationship between architecture components. (Note: in this study, patterns and styles are considered equivalent).

4.2. Architecturally Sensitive Maintainability Patterns

The concept of architecturally sensitive patterns is derived from the relation between architectural patterns and maintainability. By defining maintainability patterns, we aim to overcome a series of obstacles at architecture level and to facilitate these activities. Architecturally sensitive maintainability patterns refer to a series of patterns, which need changes at the architecture level for their implementation. In other words, implementing these kinds of patterns is difficult or even impossible without modifying software architecture. These patterns are supposed to help designers satisfy requirements concerning the maintainability aspect of software and avoid reworking in the development procedure.

An example of architecturally sensitive maintainability patterns is multi-layered architecture. Imagine that we want to increase portability which in turn increases maintainability in a software system. In these circumstances, it is not practical to solve the problem by merely relying on the patterns applicable on low-levels artifacts such as code level or design level patterns. There is no choice but to change software architecture and to use proper styles. Portability is an architecturally sensitive attribute, and we need applying architectural level patterns to achieve it. One of the patterns fulfilling this requirement is multi-layered architecture.

4.3. Maintainability Hexagon Radar

In this section we survey in a series of the most employed architectural patterns based on detected factors for each maintainability-related attribute. The result of the survey is presented as a hexagon radar diagram which illustrates the effectiveness of desired pattern on maintainability related-attributes. Numbers in a diagram represent the amount of effectiveness on the corresponding sub-attribute. Effectiveness is interpreted as follows:

- 0, large reduction of sub-attribute quality level
- 1, relative reduction of sub-attribute quality level
- 2, neutral (ineffective on quality sub-attributes)
- 3, relative increase in sub-attribute quality level
- 4, large increase in sub-attribute quality level

It should be noted that architecture patterns are preventing rather than preserving in maintainability aspects. If the radar diagram of a specific pattern shows it has a low degree of testability, it should be considered as a serious precaution regarding system testability. However, if the radar diagram of a specific pattern shows high testability, it cannot be reassured that the system based upon this pattern will necessarily be testable. We may say that, testability is guaranteed at the architectural level (obeying all rules regarding the pattern); however, there is no guarantee that testability is not violated in lower levels. We will investigate this issue in two experiences mentioned in the following.

4.3.1. Multi-layered Pattern

Multi-Layered Pattern Characteristics

Pattern Name	Multi-Layered
Pattern characteristics	System is divided into layers and each layer may communicate with adjacent layers.
Examples	OSI, Common Operating Systems such as Linux
Related references	[16, 26, 2, 40]

Multi-layered pattern is a common pattern in developing complicated systems. Its main property is restricting communication between different layers. Each layer might be connected to one upper and one lower layer. Attributes in each layer are intimately related to each other.

Testability (large increase): Multi-layered pattern increases the testability of the system. It increases *decomposability* (T3) at the system level. It is easier to test a system which is divided into layers in comparison to a system which does not use such a pattern. The force for using interfaces in this pattern increases *controllability* (T2) of the software system and layers.

Changeability (large increase): Since related components are located in nearby layers, multi-layered pattern increases *cohesion* (C1) in system components. On the other hand, multi-layered pattern mechanism results in the *reduction of coupling* (C2) in the system as layers are bound to use adjacent layers. Decrease of coupling leads to the reduction of ripple effect when we modify software artifacts. Layers are independent, so changing a layer will not cause another layer to change (layers are usually connected via interfaces).

Understandability (neutral): Multi-layered pattern has a negligible effect on system understandability. Increase or decrease of understandability in applying this pattern depends on documentation level and layering method. Clearly, a system might be layered from different points of view. MVC is a multi-layered pattern based on dividing a system into three layers; view, control and model. Evidently, understandability of this system is much more than an ordinary system without such a structure. All input events are delivered to control layer. Subsequently, this layer will respond to all requests by referring to model layer. In plain english, understandability attribute in this pattern is not architecturally sensitive. That is to say, multi-layered pattern does not lonely affect this attribute.

Portability (large increase): A distinctive advantage of multi-layered pattern is increasing portability. Responsibilities are separated into different layers which can be substituted by a novel implementation without the need for changing other layers; greatly enhancing portability. To transfer a system from one hardware to another or from one operating system to another, it is merely needed to change the middle and upper layer of the software system. Only the lowest layer has a dependency to the machine. As a result the *machine and software dependency* in this pattern is low (P1 & P2).

Stability (relative increase): Employing Multi-Layered pattern (by dividing duties between different layers) decreases risk of unexpected faults which, consequently, increases stability. Classifying layers based on tasks provides us with reusability and improves system stability for further developments. By the way, this benefit does not directly result from multi-layered pattern yet it is the consequence of using such style. Indeed, if this style is used to develop a system with completely new layers and attributes, which are not developed before, the risks of unexpected faults are still a challenge. It only provides better understanding of the system as it is divided into layers. In the framework based development programing, e.g. .NET, the framework may be considered as underneath layer and other layers can be constructed based on that. Obviously, exploiting a layer with high reusability and maturity decreases issues regarding system stability.

Analyzability (relative increase): It heavily depends on the implementation scheme of layers. A prominent point concerning multi-layered pattern which is not the case in other styles is that each layer can be replaced with corresponding stubs. In other words, each layer can be substituted by stubs simulating its performance. In this situation just one layer has been tested instead of analyzing the whole system. The analysis procedure might be conducted either upward or downward. Limiting the analysis ranges to a part of software, simplifies analysis procedure and facilitates finding deficiencies. By utilizing multi-layered pattern, *simplicity* (A1) and *decomposability* (A3) of the software system are increased. The multi-layered pattern increases analyzability.

Figure 2 shows the radar diagram of the multi-layered pattern.

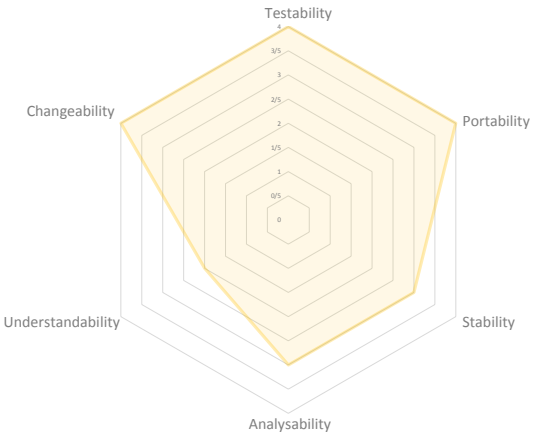

Fig 2. Multi-Layered Pattern Architecture-Level Maintainability Radar

4.3.2. Pipe & Filter Pattern

Pipes & Filters Pattern Characteristics

Pattern Name	Pipes and Filters Pattern
Pattern characteristics	System is divided into a few processes called filters. Processes are connected to each other via pipes. Processes are unaware of source and destination of data. This style facilitates evolutionary computing in a system.
Examples	Linux command line, output of a command can be input of another one. Architecture of P&F compilers
Related references	[33, 7, 2, 8]

Pipes and filters pattern consists of two separate components. Filters are code sections conducting a process on data and pipes are connection channels between these processes. Filters can be reused and new filters can be obtained combining existing ones. These can be mentioned as benefits of this style. Moreover, flexibility of the systems which is implemented based on this pattern is very high. Furthermore, filters can be arranged in such a way that facilitates parallel processing; however, proper management of faults is impossible and there are difficulties with data and process overloads. These accounted for disadvantages of this kind of pattern.

Testability (relative reduction): Testability of pipes and filters pattern is very high in filter levels. Each filter can be tested independent of other ones. Its behavior can be tested through watching its inputs and outputs, i.e. controllability in filter level is high, which implies higher testability. System-level testing is difficult due to evolutionary processing and limited control on data flow. Hence, testability is low in system level. The *controllability* of the pipes and filters pattern in system-level is so low (T2).

Changeability (large increase): As mentioned before this style shows great flexibility and its architecture can be changed easily. Modifying each filter does not impact adjacent filters, and they operate independently (C1). The coupling between filters is low (C2). As a result, the level of the changeability attribute increases by using this pattern.

Understandability (large reduction): One of the main weak points of this pattern is its low understandability. Filter interactions, type of communication and transferred data, which are not standards in some cases due to lack of control on filters operation, cause poor understandability. Assume that input of a specific filter is a list of people. It is probable that the list is sorted in previous filters. In order to avoid adding redundant comparison and sorting overload to the system, we should add a quantitative data to the system which shows whether it is sorted or not. Series of this data is added to the list as the process makes progress and results in complexity and decreases understandability.

Portability (relative increase): Filters have great portability as various filters can be merged using different methods. The filters and pipes design *machine independence* (P1).

Stability (neutral): This pattern does not affect system stability. Namely, stability is not sensitive to this architecture pattern. In case of using mature filters, system stability will probably increase. Since pipes have an enormous impact on system operation, we used the word "probably". In other words, even utilizing perfectly mature filters do not guarantee system stability.

Analyzability (large reduction): One of the most essential shortcomings of pipes and filters pattern is analyzing code and error sources. The more filters we have, the more analyses we need to figure out error sources. We may even need to analyze all filters to find failure source. Ordinarily, it is not practical to easily define analysis range and use suitable stubs. Analyzability of this system is extremely low. The level of *traceability* (A5) factor in this pattern is very low.

Figure 3 shows the radar diagram of the pipes and filters pattern.

4.3.3. Batch Sequential Pattern

It is exactly the same as pipes and filters pattern except some limitations regarding type of connection between filters [40].

1. Parallelism does not exist in this pattern
2. There is no interaction with user during execution
3. Filters are coarse-grained
4. Delay is considerable in this pattern

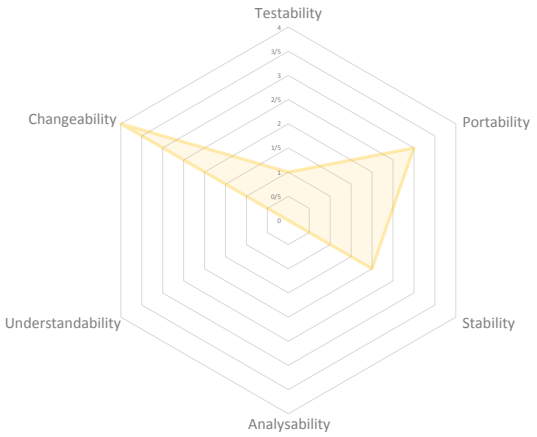

Fig 3. Pipes & Filters Pattern Architecture-Level Maintainability Radar

Concerning our study there is not any difference between these two models with respect to maintainability and the major differences are related to efficiency.

4.3.4. Blackboard Pattern

Blackboard Pattern Characteristics

Pattern Name	Blackboard
Pattern characteristics	A processing framework for coordinating large and heterogeneous modules in an environment with indefinite control. Each component can write or read knowledge on the base. The only mechanism for communication between components is this knowledge base. Components can not directly communicate.
Examples	The system of robot which does various tasks in an office. It can gain knowledge from environment. It performs three tasks continuously: - Gaining environment information to compute motion and receive human commands - Analyzing gained information and solving related issues - Activity and changing environment
Related references	[41, 42, 7, 19]

Blackboard pattern is a suitable approach to solving uncertain problems. In this pattern, components do not communicate to each other directly. Each component changes some of the variables on the blackboard; other components respond to these changes by analyzing updated information and according to their defined tasks. It is possible to call multiple components simultaneously by using this pattern.

Testability (large reduction): Testability of the systems implemented by using blackboard pattern is very low. Uncertainty of circumstances prevents simple testing of code sections. *Controllability* (T2) of components is extremely low, and it is impossible to guess outputs by determining input data. The communications among components in this pattern are complicated and as a result the *simplicity* attribute of the software is low (T4).

Changeability (large increase): The level of changeability in this pattern is very high. Each component can be edited without interrupting responsibility of the other components (Localized modification or C1). It is simple to add new components to the system as they are independent of the others (Low coupling or C2). Additionally, one component may be omitted without influencing other ones.

Understandability (large reduction): Understandability in component level relies on internal style of com-

ponent (as internal style of components may be a pattern which is different from the general system pattern) and component documentation level. Overall, understandability in component level is not architectural sensitive. Particular method of interaction between components makes it hard to understand the system. The understandability in architecture-level is very low.

Portability (neutral): Portability is independent of architecture in this pattern. It separately depends on the internal structure of each component.

Stability (neutral): It is also architectural insensitive and depends on the internal structure of components.

Analyzability (large reduction): This style has a considerably low analyzability. Uncertain conditions oblige us to test large number of components so that we can find failures and deficiencies. The level of *simplicity* (A1) and *traceability* (A5) factors in this pattern is very low.

Figure 4 shows the blackboard pattern maintainability diagram.

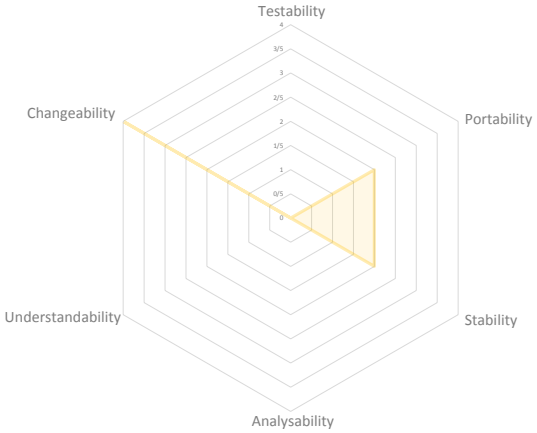

Fig 4. Blackboard Pattern Architecture-Level Maintainability Radar

4.3.5. Object Oriented Pattern

Object Oriented Pattern Characteristics

Pattern Name	Object Oriented
Pattern characteristics	System is divided into objects and messages transferred between them. Characteristics of objects are not editable. There is inheritance property between objects. Changing one object without affecting others.
Examples	A wide range of programs written in JAVA.
Related references	[33, 16]

It is vastly applied and common. The system is defined by objects and their relations. Objects can inherit each other in different levels, which includes objects attributes and characteristics. In object-oriented pattern, direct access to characteristics of an object is restricted for other objects. The relationship between objects might be absolutely sophisticated in this pattern.

Testability (relative reduction): By controlling inputs of each object and analyzing its outputs various tests can be designed. There is an important point regarding testability, which is the dependency of different objects to each other. Imagine that object A is dependent on object B and object B is dependent on object C. So in case of a change in object C proper operation of A and B might be violated. This effect is known as ripple effect. Testing object-oriented systems are difficult and sophisticated. Their testability depends on interaction method and usage of objects. The level of the *controllability* (T2) and *simplicity* (T4) factors in this pattern are low.

Changeability (relative reduction): Each object in object-oriented pattern can be modified. This will influence the object itself and other objects which have interactions with that object. That is to say, editing one object may force us to edit a wide range of objects. Therefore, changeability in object-oriented pattern depends on the kind of relationship between the object and its adjacent objects. The level of the *coupling* factor in OO systems is higher than other patterns (C2).

Understandability (relative reduction): As two previous attributes, this attribute completely depends on the relationship between objects, and how much it spreads. A large number of connections and multi level inheritance dramatically decrease understandability.

Portability (large reduction): This pattern suffers from extremely poor portability. It is complicated for the system to migrate from one environment to another one. The levels of the software and machine independence factors in this pattern are low (P1 & P2).

Stability (relative reduction): System stability may decrease drastically according to relation between objects. If there is a circular relation between some objects, changing one of them will lead to a vast change in dependent objects. If there are not lots of circular dependencies and inheritance levels, stability of the system will improve gradually, and it will become matured.

Analyzability (relative reduction): This attribute completely depends on relationship between objects. The more complicated and widespread is relation between objects, the less is analyzability. The level of the *traceability* factor in this pattern is low (A5).

Figure 5 shows the maintainability radar diagram of the object oriented pattern.

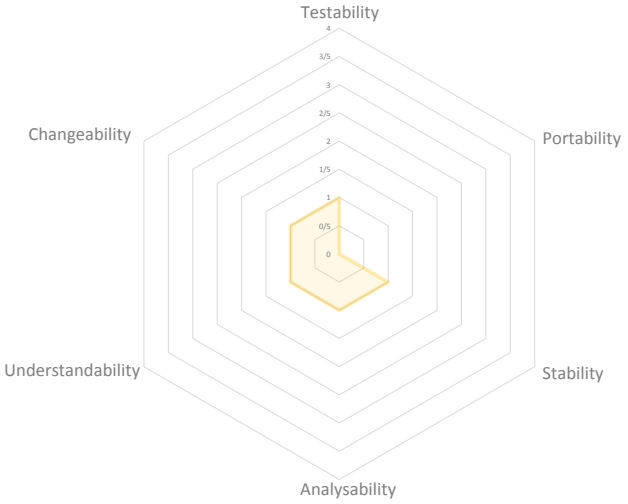

Fig 5. Object Oriented Pattern Architecture-Level Maintainability Radar

4.3.6. Interpreter Pattern

Interpreter Pattern Characteristics

Pattern Name	Interpreter
Pattern characteristics	It consists of an interpreter motor, peripheral memory for saving required information and state of system. State includes two types, interpreter motor state and the state of program which is being interpreted.
Examples	PHP language Interpreter
Related references	[41, 39, 2, 43]

This pattern is frequently applied in environments where comparing and choosing best language or machine for implementation is not practical. A good example is a speaker who is going to give a speech in various countries. In this situation the speaker does not speak in different languages, but he speaks in his mother tong, and a translator accompanies him to translate his language. The most crucial property of this style (can be inferred from the example) is its high portability between different platforms. Unfortunately, this pattern does not provide ideal efficiency (remember delay of translation).

Testability (neutral): This pattern neither improves nor worsens any of factors mentioned in Table 2. As a matter of fact, this pattern cannot increase or decrease testability of the software system on its own. Testability depends on pattern used for writing script of the executable program. Thus, testability is not architectural sensitive in this pattern.

Changeability (large increase): Interpreting quiddity of this architectural pattern *defers binding time* (C3). As explained in Table 3, deferred binding time is one of the methods employed for increasing changeability. So changeability will increase by applying this pattern.

Understandability (neutral): Understandability is not architectural sensitive in this pattern.

Portability (large increase): This pattern provides highest portability level comparing to the other patterns. Independency of the designed system to the machine it runs on, assist in achieving the highest degree of portability in this pattern (P1 & P2). Written scripts merely have to obey language syntax and do not concern execution environment.

Stability (neutral): Stability is not architectural sensitive by using this pattern.

Analyzability (neutral): It is not architectural sensitive as well.

Figure 6 shows the maintainability radar diagram of the interpreter pattern.

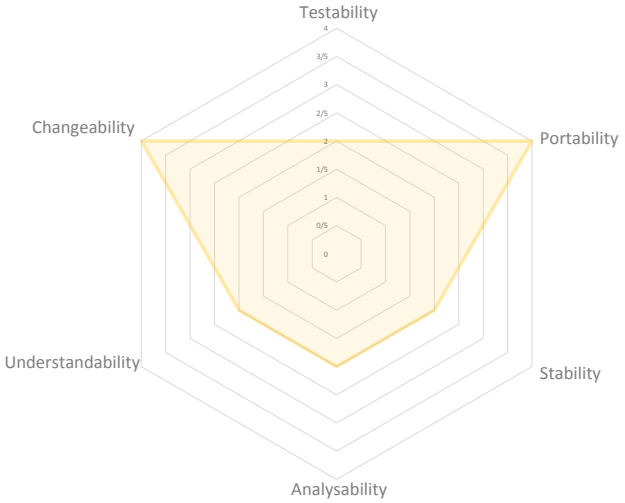

Fig 6. Interpreter Pattern Architecture-Level Maintainability Radar

4.3.7. Virtual Machine Pattern

This pattern is similar to interpreter pattern [15]. Contrary to the previous pattern, scripts are interpreted to an intermediate language called byte-code, which optimizes interpretation procedure in virtual machine. Another difference is changeability of this pattern. Since the code are compiled to an intermediate one, deferred binding time does not exist anymore, and this attribute will not be architectural sensitive. In plain english, *changeability* in this pattern depends on programing method, patterns inside components and so on. It is not directly dependent on architecture.

4.3.8. Broker Pattern

Broker Pattern Characteristics

Pattern Name	Broker
Pattern characteristics	Servers and clients are independent, Data is transferred as soon as it is ready, Various clients, Changing the number of clients during run time, Relation between components via an interface called broker
Examples	Subscribing to a newspaper or magazine.
Related references	[8, 2, 44, 7]

In this pattern[1] client and server components operate independently. As a result, server will not be blocked in case of not existence of a client, which has its pros and cons. As an example of this pattern, consider a newspaper printing organization. Printing each newspaper imposes costs on newspaper office. If the newspaper is not sold,

[1]Broker pattern is a specialized form of Mediator [28]. Maintainability radar of Mediator is the same as Broker's diagram.

this cost will be wasted (financial loss). It is the same in this pattern, if data is generated and there is no client means that it is burdened by overload.

Testability (large increase): Testability of this pattern is high due to increase in several factors including: separation of clients and servers (T3); *simplicity* of play/restore of test cases (T4); components *controllability* (T2); dividing the system into separate components (T3) and less complicated subsystems (T4).

Changeability (large increase): Changeability is also considerably high because of *deferred binding time* (C3), *reduction of coupling* between client and server (C2) and *increases in cohesion* in the client and server components (C1).

Understandability (neutral): As relationship between components is standard in this pattern (It is necessary to define interfaces and standards for communication), understandability in this system is much more than ordinary systems. However, understandability of generated codes depends on coding standard and prepared documentations and it is independent of this pattern.

Portability (relative increase): Using this pattern results in independence of server component and execution environment of client component. When clients are independent of servers' environment and vice versa, portability of the whole system will increase (increase in P2). It is possible to use an environment different from clients' environment to generate data (increase in P1). Indeed, we may have different clients from various environments that interact with one server through brokers.

Stability (neutral): Stability is not architectural sensitive in this pattern. This pattern affects stability neither positively nor negatively.

Analyzability (large increase): The system is divided into independent components (increase in A3 & A1). The amount of *traceability* (A5) among components are moderate. This pattern largely increases analyzability of the software.

Figure 7 shows the maintainability radar diagram of this pattern.

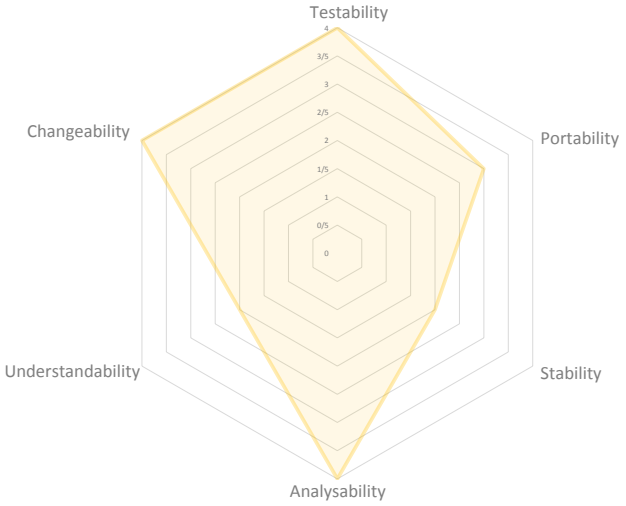

Fig 7. Broker Pattern Architecture-Level Maintainability Radar

4.3.9. MVC Pattern

MVC Pattern Characteristics

Pattern Name	MVC
Pattern characteristics	System is divided into three parts; model, view and controller. Controller fetches proper model and renders corresponding view. Finally, the response is transmitted.
Examples	Lots of web applications in present layer use this pattern.
Related references	[9]

This pattern is a common used pattern in the web development domain. It is usually used in the presentation component and the server side. It is a specific version of multi-layered pattern. Figure 8 shows a MVC style

which is mapped to multi-layered pattern. In this pattern, model is responsible for providing data information and communicating with a data source. Controller is supposed to interpret received requests and fetch the suitable view. View provides view information to show a specific model. We expect this pattern to present maintainability quality attributes similar to multi-layered pattern. We should pay attention to one prominent point:

- Controller is separated from view and model. There are some recommendations based on this pattern:
  - Operations regarding changing data should take place in models.
  - Operations concerning display should be performed in view.
  - Controller is responsible for interpreting requests, fetching data from models, providing data for views and controlling accesses.

Testability (large increase): Like multi-layered pattern.
Changeability (large increase): Like multi-layered pattern.
Understandability (relative increase): Since there exist a standard for developing controllers, views and models in this pattern, the developers had a common understanding about the concepts used in this pattern. As a result, its understandability is higher than the multi-layered pattern.
Portability (large increase): This pattern does not impact on the portability attribute, and the portability is not architectural sensitive in this pattern.
Stability (neutral): Like multi-layered pattern.
Analyzability (relative increase): Like multi-layered pattern.
Radar diagram of this pattern is illustrated in Figure 9.

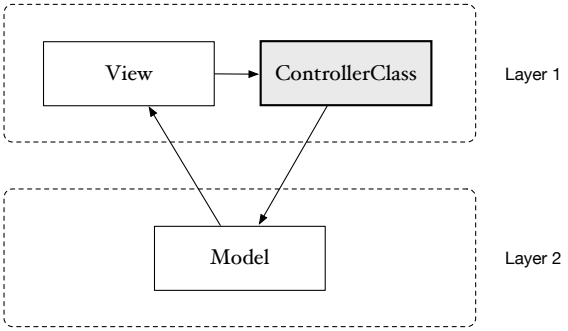

Fig 8. Mapping MVC Pattern to Layer Pattern

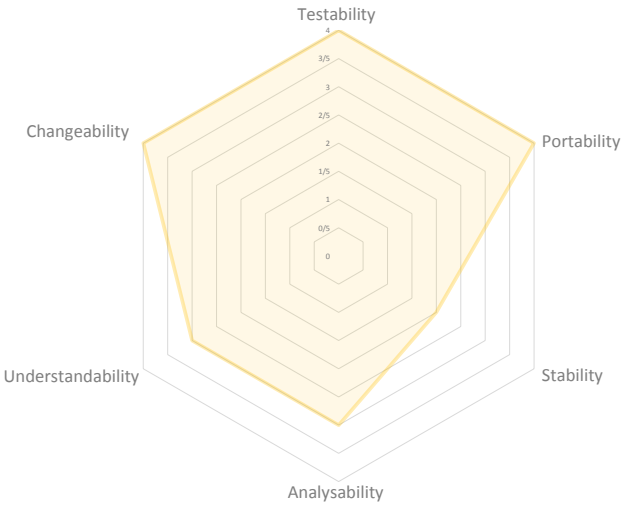

Fig 9. MVC Pattern Architecture-Level Maintainability Radar

5. Case Study

One of the goals of this study is creating a mechanism for the selection of proper patterns according to the requirements of software stakeholders. Paying attention to influence of architecture patterns on each quality attribute

during architecture design phase leads to the reduction of the development cost in next phases such as the maintenance.

In this section, we survey a special usage of our model, which is *architecture refactoring*. The aim of architecture refactoring is to increase maintainability of the software system without changing external visible functionality of the system [13].

5.1. Architecture-Level Refactoring

An advantage of knowing the effect of different architectural patterns on specific maintainability attributes is its utility in developing an architectural refactoring procedure. For this purpose, we need to devise a framework to make it possible to detect the requirements concerning system maintenance, and current architecture pattern, and to choose a suitable architecture pattern into which to refactor. Architectural level refactoring steps are as follows:

1. *Developing maintainability requirements model:* In this study, we used radar diagrams to demonstrate the relationship between each pattern and six maintainability-related-attributes. We can use this radar diagram to describe maintainability requirements. The output of this step is a radar diagram showing the minimum acceptable level of each maintainability-related-attributes.
2. *Determining current architectural pattern:* An essential step toward refactoring is the understanding of the current system. There are some suitable methods such as methods based on scenario profiles [4] which can be used to recognize architectural patterns in the system at hand. It should be noted that architectural patterns are heterogeneous in real systems, meaning that one frequently sees a particular pattern used at one level, with very different patterns used at other (lower) levels.
3. *Comparing radar diagrams:* Comparing the requirement radar diagram, the radar diagram corresponding to the architectural pattern of the current system, and the radar diagrams of various other architectural patterns and choosing the desired pattern based on minimal fulfillment of requirements, the incurred costs (monetary, time, etc.), and the amount of improvements achieved in maintainability.

5.2. Two Experiences Regarding Architecture-Level Refactoring

In this section, we investigated two real-world refactoring experiences (one successful and one not so). The first one is a system in an academic environment which has been designed 15 years ago. The second one is a commercial content management system designed three years ago. We will consider various issues concerning these two experiences.

5.2.1. The first experience: Successfully implemented refactoring leading to noticeably increased maintainability

rms.ilam.ac.ir is a website used for collecting research information regarding professors at Ilam University. The primary version of the website was designed about 15 years ago using Visual Basic language. The authors of this article are currently responsible for the maintenance of this website. During the past four years, maintainability of the system has been arduously difficult and time consuming. This led to us deciding to move the system to a new architecture which helps solve the system's problems regarding maintainability. In the rest of this section, we first introduce the system's previous architecture, and then investigate the new architecture, concluding by comparing the statistics corresponding to maintainability for the two architectures.

The old system architecture. This system was designed using proxy architectural pattern (which is similar to broker pattern). All requests are required to be transmit to a central controller (proxy server). Based on the type of request one of the existing classes would be selected by proxy server and compiled and sent as a response.

The major problem which we encountered using this system was the long time needed to understand code purpose and classes related to each request. The system did not have particular external or internal documentations. Adding a new functionality required changing the proxy system, adding classes and recompiling the whole project. As one can see in broker pattern radar diagram, it suffers from low understandability. Proxy is connected to a large number of views and it fetches one of them based on the type of request. Here lies the root of the problem, adding a new view means that the proxy should be modified. Similarity between requests will inevitably result in sending request for an old view. All old views should be checked, and new views should not be added unless they are checked to avoid any interference.

Novel architecture. After investigating previous architecture and its shortcomings our developers team, decided to modify architecture pattern and approaches through which requests are handled. The closest patterns to the current system pattern were surveyed, and the MVC pattern was selected to meet the requirements [35]. As a result, the proxy component was divided into 19 controller classes. Corresponding views were built for each model. Previous views were still usable and they were added into the system with minor modifications.

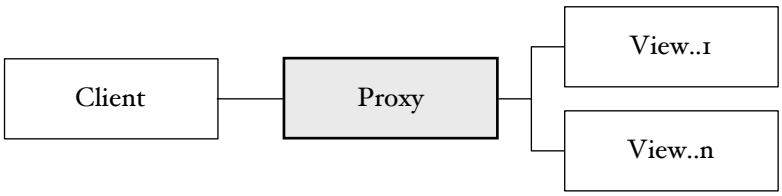

Fig 10. Profs System's Old Architecture

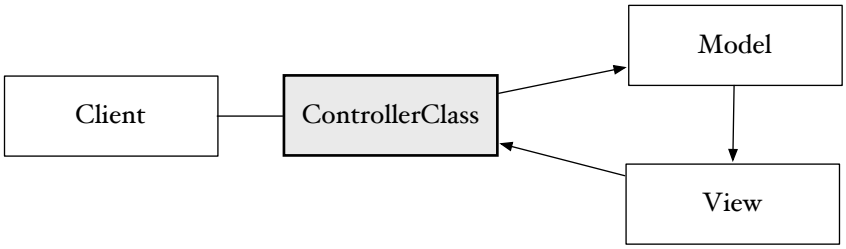

Fig 11. Profs System's Novel Architecture

Maintainability in old and novel model. As both models were available, we decided to compare systems maintainability with respect to man-hour and considering three scenarios performed during last year. The modifications requested by administration of the Ilam University during the last year were categorised as three scenarios:

- Scenario 1, Adding a new account type to the system: In accordance with requirements of audit staff, we needed to add a type of user who can log on to the system and make a report on professors' research activities.
- Scenario 2, changing journals and conferences scoring methods: According to a new decision of the university research council, we needed to change the approach through which articles are scored.
- Scenario 3, Adding affiliation to articles submitted by professors: As new research institutes are established, and faculty staff may be a member of two places, it should be possible to change affiliation of each article and make a report with respect to the place where it is presented in.

One of the difficulties of maintaining the old system was forgetting the logic behind the program after a short time. We implemented the aforementioned scenarios in three-time spans, when architecture was being changed (when our understanding of architecture was on its highest degree), three months after changing architecture and six months after changing it. Afterwards, we compared results. It should be mentioned that time is rounded to hours, and they are approximate. As presented in Table 10, the average improvement achieved during performing is at least 50 percent. This improvement has increased by 12 and 14 percent, respectively, in the next months owing to the reduction of time needed for understanding and changing the code. We may state general reasons for the improvement achieved using the MVC style as follows:

- Developer team was familiar with MVC pattern due to their previous experiences.
- Separating the controller: instead of changing a huge class called proxy, we have 19 medium controller classes (scenario 1).
- Separating models: We need to change only one class to modify models and change database (scenario 2 and 3).
- Smaller code and simple testing (scenario 2).
- Higher controllability of MVC pattern.

In reporting time spans for the MVC pattern, we have considered total time (man-hour). In MVC pattern, lots of modifications needed in scenario 1 can be achieved in parallel using multiple teams. It should be noted that our experiment was not a controlled one. It was merely a simple experience in a work environment. Perhaps some other factors have affected results as well. On the other hand, maintainability improvement in this project is undeniable and sensible. The developers team have felt this characteristic during realizing users' requirements and system maintenance. This improvement did not exist before, and it has been observed only after changing the architectural pattern.

Table 8. Required effort to do scenarios based on changed class and person-hour in old architecture (First Experience)

Scenario	Changed Class	After Architecture Change	After 3 Month from Architecture Change	After 6 Month from Architecture Change
Scenario 1	11	5h	12h	11h
Scenario 2	8	10h	17h	16h
Scenario 3	12	20h	26h	27h

Table 9. Required effort to do scenarios based on changed class and person-hour in novel architecture (First Experience)

Scenario	Changed Class	After Architecture Change	After 3 Month from Architecture Change	After 6 Month from Architecture Change
Scenario 1	10	3h	5h	5h
Scenario 2	2	7h	8h	8h
Scenario 3	2	10h	12h	11h

Table 10. Improvement from old architecture to novel architecture in percent (First Experience)

Scenario	Changed Class	Time Improvement		
		After Architecture Change	After 3 Month from Architecture Change	After 6 Month from Architecture Change
Scenario 1	9%	40%	58%	58%
Scenario 2	75%	60%	76%	75%
Scenario 3	83%	50%	53%	59%
Average	55%	50%	62%	64%

Table 11. Required effort to do scenarios based on changed class and person-hour in old architecture (Second Experience)

Scenario	Old architecture		Novel architecture	
	Changed Class	Time	Changed Class	Time
Scenario 1	1	9h	1	10h
Scenario 2	17	4h	17	5h
Scenario 3	2	10h	5	12h
Scenario 4	19	93h	23	84h

Table 12. Improvement from old architecture to novel architecture in percent (Second Experience)

Scenario	Changed Class	Time
Scenario 1	0%	-10%
Scenario 2	0%	-20%
Scenario 3	-133%	-20%
Scenario 4	-21%	+10%
Average	-38%	-10%

5.2.2. The second experience: unsuccessful refactoring experience

Chavir.ir is an online content management system which facilitates maintaining and editing information concerning informative websites such as scientific labs, research institutes and so on. The system was designed in 2010. The primary design did not follow any architectural pattern. It consisted of a large number of php files, producing responses based on URLs sent by users (rrc.tums.ac.ir website uses this version). The developers team decided to use the MVC style to take advantage of its benefits. In this section, we first introduce the previous architecture, and then we investigate the new architecture, and finally comparing the statistics corresponding to maintainability for the two architectures.

Old system architecture. As mentioned before, the original system did not conform to any specific architectural pattern. It included a bunch of files, which were responsible for managing part of data such as news, slides and etc. It had 17 main php files, which were responsible for adding and deleting data, and making data editing possible. Each file used to perform operations on one or a few database tables. New capability could be added into the system either by adding a php file with a structure similar to the previous ones or by modifying existing files.

New system architecture. Based on the successful experience of the research system, we decided to change the architectural pattern to the MVC style and benefit from its advantages, including high maintainability. Thus, 17 php files were converted to 17 controller classes, 76 view classes and 40 model classes. The large number of views created is due to the necessity to use separate classes for index, create, update and detail. In other words, four view classes were built for each controller class on average. The number of model classes equals to the number of tables inside the database.

Converting classes and moving them to the new pattern took three months. Based on our estimation it consumed about 400 man-hours for implementation and test. New code was designed based upon the *Yii Framework*, and it perfectly follows MVC pattern. Manesht.ir is a website which uses the new version of this system.

Maintainability in old and new models. Similar to the research system experience, here we review the scenarios in which we engaged. The following four scenarios are considered:

- Scenario 1: Iranian banks were obliged to change their electronic payment ports from their own domains to a centralized port called shaparak.ir. This resulted in the change of web services and electronic payment procedures. Hence we had to change the old and the new systems.

- Scenario 2: There was a problem related to SQL-Injection in system forms. Thus, all controllers which used to receive their information based on POST method had to be changed.

- Scenario 3: The system should be able to show advertisements at pre-defined places inside the website format and according to a pre-determined size. This was necessitated by one of the clients.

- Scenario 4: Accessibility control was simple and used to be performed by defining a few defaults. Clients' requests necessitated adding dynamic access control. So we needed to provide facilities for defining account groups, and for controlling accesses in all pages of website administration, which had to happen prior to each action.

Table 10 shows that improvement is achieved only in one scenario and in scenarios 1-3 not only there is no improvement, but also the time needed for software maintenance is increased. Cost increase in the new architecture can be justified on following grounds:

- The increase in the number of classes needing to be modified, which results in increased cost and testing time (scenario 3),

- The time needed to get familiar with the MVC architecture and relearning it in the group (since previous architecture was very simple and elementary)

Anyway, looking at the results, we may infer that the original and the new architectures are suitable for minor and major changes, respectively (Scenario 4).

Comparing the time consumed and modified classes in both architectures, it can be concluded that refactoring has not considerably affected (like it was the case in the first example) maintainability. It is worth mentioning that architectural level patterns can either open or close the way toward software maintainability. Undoubtedly, an improper pattern decreases maintainability. Nevertheless, using a proper pattern does not necessarily increase maintainability [22]. In fact, maintainability patterns together with correct decisions at lower levels may lead to a maintainable software.

6. Related Work

In 1980, Lientz and Swanson divided maintainability into three sections: changeability, understandability and testability [29]. Bengtsson et al. had a similar idea. They defined changeability as follows [4]: "changeability is the simplicity of carrying out modifications with respect to changes in environment, requirements or features". Bass et al. hypothesized about achieving maintainability via changing architecture [2]. Heitlager et al. preset practical model for measuring maintainability in their work [21]. They provide some metric for measuring ISO 9126 maintainability sub-attributes at code-level. Hegedus et al. showed that by utilizing design patterns one can improve the maintainability of the software code [20]. They survey more than 300 revisions of JHotDraw program to conclude about their findings.

Bushmann et al. introduced a distinct point of view. They contended that maintainability includes expandability, portability, refactoring, and omitting unwanted features [7]. They divided software changes into three levels; non-local changes, local changes and architectural changes. They believed that the most simple changes are local changes and architecture should be designed in such a way that these changes could be easily carried out (it should not be preventive with respect to maintainability). One of the studies about the relationship between design patterns and maintainability was conducted by Prechelt et al. [36]. Their study compared two systems with controlled conditions. It revealed that using a specific design pattern does not necessarily increase maintainability.

There are some major works done on the relationship between maintainability and architecture. Bengtsson et al. [3] proposed a series of metrics to measure maintainability of architecture. They also worked on a method for analayzing maintainability at the architecture level, which is called ALMA [4]. They assessed their method in a few specific cases [4]. Muthanna et al. presented a series of metrics to measure maintainability at design level [34]. Loetta et al. investigated maintainability of two systems with SOA and non-SOA architecture in a postal system [27]. Liu et al. proposed a set of metrics to measure architecture maintainability [30]. Their study measures maintainability at architecture level and ignores the code level.

7. Conclusion and Future Work

In this study, we chose and developed a quality model for maintainability based on known quality models. Some of the well-known patterns were surveyed using the proposed model, and they were compared based on the attributes of it. This study is a part of a larger one which aims to conduct refactoring at the architecture level. In this study, we obeyed an up-down scheme, and the main goal was to detect the relationship between architectural patterns and maintainability quality attribute. In the next efforts, we are going to develop a methodology for detecting the relation between clients' requests and architectural patterns. In this methodology, we will use a tool known as scenario profiles. They will classify clients' requirements and map them to maintainability sub-attributes. In this case, it would be possible to find suitable patterns for refactoring by checking developed hierarchy. There is still a problem regarding architectural level refactoring which can be considered as future work. This problem is estimating the cost and time needed for refactoring the current pattern to desired one. For instance, suppose that by considering the customer requirements, we are led to three suitable patterns, and we have to choose the final pattern by estimating the cost and implementation time incurred by each of the three patterns. Conventional methods for cost estimation regarding software maintenance include the number of links, number of components, migration of features and size of components, may not be suitable to estimate the refactoring cost.

References

[1] F. Bachmann, R. L. Nord, and I. Ozkaya, *Architectural tactics to support rapid and agile stability*, CrossTalk (SEI), (2012).

[2] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2 ed., 2003.

[3] P. Bengtsson, *Towards maintainability metrics on software architecture: An adaptation of object-oriented metrics*, in First Nordic Workshop on Software Architecture (NOSA'98), Ronneby, 1998.

[4] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, *Architecture-level modifiability analysis (alma)*, Journal of Systems and Software, 69 (2004), pp. 129–147.

[5] B. W. Boehm, J. R. Brown, and M. Lipow, *Quantitative evaluation of software quality*, in Proceedings of the 2nd international conference on Software engineering, IEEE Computer Society Press, 1976, pp. 592–605.

[6] S. Bohner, *Impact analysis in the software change process: a year 2000 perspective*, in Software Maintenance 1996, Proceedings., International Conference on, Nov 1996, pp. 42–51.

[7] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-oriented Software Architecture: A System of Patterns*, John Wiley & Sons, Inc., New York, NY, USA, 1996.

[8] P. Clements, D. Garlan, L. Bass, J. Stafford, R. Nord, J. Ivers, and R. Little, *Documenting software architectures: views and beyond*, Pearson Education, 2002.

[9] J. Deacon, *Model-view-controller (mvc) architecture*, Online Source: `http://www.jdl.co.uk/briefings/MVC.pdf`, (2009).

[10] R. Dromey, *Cornering the chimera [software quality]*, Software, IEEE, 13 (1996), pp. 33–43.

[11] R. G. Dromey, *A model for software product quality*, IEEE Transactions Software Engineering, 21 (1995), pp. 146–162.

[12] D. G. Firesmith, *Common concepts underlying safety security and survivability engineering*, tech. rep., Carnegie Mellon Software Engineering Institute - Technical Note CMU/SEI-2003-TN-033, 2003.

[13] M. Fowler, *Refactoring: improving the design of existing code*, Addison-Wesley Professional, 1999.

[14] J. E. Gaffney Jr, *Metrics in software quality assurance*, in Proceedings of the ACM'81 conference, ACM, 1981, pp. 126–130.

[15] J. García-Martín and M. Sutil-Martin, *Virtual machines and abstract compilers-towards a compiler pattern language*, in In Proceeding of EuroPlop 2000, Irsee, Citeseer, 2000.

[16] D. Garlan and M. Shaw, *An introduction to software architecture*, Advances in software engineering and knowledge engineering, 1 (1993), pp. 1–40.

[17] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[18] P. A. Grubb and A. A. Takang, *Software maintenance: concepts and practice*, World Scientific, 2003.

[19] B. Hayes-Roth, *A blackboard architecture for control*, Artificial intelligence, 26 (1985), pp. 251–321.

[20] P. Hegedűs, D. Bán, R. Ferenc, and T. Gyimóthy, *Myth or reality? analyzing the effect of design patterns on software maintainability*, in Computer Applications for Software Engineering, Disaster Recovery, and Business Continuity, Springer, 2012, pp. 138–145.

[21] I. Heitlager, T. Kuipers, and J. Visser, *A practical model for measuring maintainability*, in Quality of Information and Communications Technology, 2007. QUATIC 2007. 6th International Conference on the, IEEE, 2007, pp. 30–39.

[22] F. Hoffman, *Architectural software patterns and maintainability: A case study*, PhD thesis, University of Skövde, 2001.

[23] ISO/IEC, *Iso standard 9126: Software engineering - product quality, parts 1, 2 and 3*, 2001 (part 1), 2003 (parts 2 and 3).

[24] I. Jacobson, G. Booch, and J. Rumbaugh, *The Unified Software Development Process*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[25] H. Kabaili, R. K. Keller, and F. Lustman, *Cohesion as changeability indicator in object-oriented systems*, in Proceedings of the Fifth European Conference on Software Maintenance and Reengineering, CSMR '01, Washington, DC, USA, 2001, IEEE Computer Society, pp. 39–.

[26] P. Kruchten, *Architectural blueprints - the "4+1" view model of software architecture*, Tutorial Proceedings of Tri-Ada, 95 (1995), pp. 540–555.

[27] M. Leotta, F. Ricca, G. Reggio, and E. Astesiano, *Comparing the maintainability of two alternative architectures of a postal system: Soa vs. non-soa*, in Software Maintenance and Reengineering (CSMR), 2011 15th European Conference on, IEEE, 2011, pp. 317–320.

[28] N. Lévy, F. Losavio, and A. Matteo, *Comparing architectural styles: Broker specializes mediator*, in Proceedings of the Third International Workshop on Software Architecture, ISAW '98, New York, NY, USA, 1998, ACM, pp. 93–96.

[29] B. P. Lientz and E. B. Swanson, *Software Maintenance Management*, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1980.

[30] L. Liu, X.-D. Zhu, and X.-L. Hao, *Maintainability metrics of software architecture based on symbol connector*, in Quality, Reliability, Risk, Maintenance, and Safety Engineering (QR2MSE), 2013 International Conference on, IEEE Computer Society Press, 2013, pp. 1564–1567.

[31] J. A. McCall, P. K. Richards, and G. F. Walters, *Factors in software quality. volume-iii. preliminary handbook on software quality for an acquisiton manager*, tech. rep., DTIC Document, 1977.

[32] S. McConnell, *Code complete*, O'Reilly Media, Inc., 2004.

[33] R. T. Monroe, A. Kompanek, R. Melton, and D. Garlan, *Architectural styles, design patterns, and objects*, Software, IEEE, 14 (1997), pp. 43–52.

[34] S. Muthanna, K. Kontogiannis, K. Ponnambalam, and B. Stacey, *A maintainability model for industrial software systems using design level metrics*, in Reverse Engineering, 2000. Proceedings. Seventh Working Conference on, IEEE Computer Society Press, 2000, pp. 248–256.

[35] Y. Ping, K. Kontogiannis, and T. C. Lau, *Transforming legacy web applications to the mvc architecture*, in Software Technology and Engineering Practice, 2003. Eleventh Annual International Workshop on, IEEE, 2003, pp. 133–142.

[36] L. Prechelt, B. Unger, W. F. Tichy, P. Brössler, and L. G. Votta, *A controlled experiment in maintenance comparing design patterns to simpler solutions*, IEEE Trans. Softw. Eng., 27 (2001), pp. 1134–1144.

[37] R. Pressman, *Software Engineering: A Practitioner's Approach*, McGraw-Hill, Inc., New York, NY, USA, 6 ed., 2005.

[38] J. Sanders and E. Curran, *Software Quality: A Framework for Success in Software Development and Support*, ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1994.

[39] M. Shaw, *Some patterns for software architectures*, Pattern languages of program design, 2 (1996), pp. 255–269.

[40] M. Shaw and P. Clements, *A field guide to boxology: Preliminary classification of architectural styles for software systems*, in Computer Software and Applications Conference, 1997. COMPSAC'97. Proceedings., The Twenty-First Annual International, IEEE, 1997, pp. 6–13.

[41] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*, Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[42] O. Silva, A. Garcia, and C. Lucena, *The reflective blackboard pattern: Architecting large multi-agent systems*, in Software engineering for large-scale multi-agent systems, Springer, 2003, pp. 73–93.

[43] J. Vlissides, R. Helm, R. Johnson, and E. Gamma, *Design patterns: Elements of reusable object-oriented software*, Reading: Addison-Wesley, 49 (1995), p. 120.

[44] M. Völter, M. Kircher, and U. Zdun, *Remoting Patterns: Foundations of Enterprise, Internet and Realtime Distributed Object Middleware*, Wiley. com, 2005.

Please cite this article using:

Zahed Rahmati, Mohammad Tanhaei, Ensuring software maintainability at software architecture level using architectural patterns,AUT J. Math. Com., 2(1) (2021) 81-102
DOI: 10.22060/ajmc.2021.19232.1044