AJIT-e: Online Academic Journal of Information Technology
2018- Cilt/Vol: 9-Sayı/Num: 34
DOI: 10.5824/1309-1581.2018.4.001.x
http://www.ajit-e.org/?menu=pages&p=details_of_article&id=344

Received : 03.04.2018 Editorial Process Begin: 28.04.2018 Published: 03.10.2018

An Experimental Evaluation of the Effect of SOLID Principles to Microsoft VS Code Metrics

Osman TURAN, Ankara University Graduate School of Natural and Applied Sciences, osmanturan@gmail.com

Ömer Özgür TANRIÖVER, Faculty of Engineering Computer Engineering Department Ankara Turkey, Asst. Prof Dr. Özgür Tanrıöver, tanriover@ankara.edu.tr

ABSTRACT *Software maintenance is necessary for reasons such as changes in user needs, changes in the operating conditions of the system due to changes in the infrastructure, the occurrence of unforeseen errors. The suitability of the software for maintenance operations is a significant influence in reducing the cost. Using only basic object oriented programming concepts do not show that we are writing maintainable code in our applications. Object oriented design principles such SOLID are about reducing dependencies and increasing maintainability. ISO/IEC 9126 is about maintainability but ISO/IEC 9126 is not clear about whether all inputs to measurement should be used together in conjunction or whether they should be used as appropriate or available. Indeed, ISO/IEC 9126 provides no guidance, heuristics, rules of thumb, or any other means to show how to trade off measures, how to weight measures or even how to simply collate them. In this study each sub-characteristic of ISO/IEC maintainability with help of Visual Studio (VS) code metric tool is assessed. The focus of this assessment is on maintainability and its sub-characteristics like analyzability, testability, changeability and stability. Before doing an analysis, each sub-characteristics of maintainability part of ISO/IEC 9126 standard are mapped to five VS code metrics for measurement of characteristics. This work shows the effect of object oriented design principles (SOLID) to the maintainability, complexity and flexibility of the code while associating ISO/IEC, VS code metric and SOLID.*

Keywords: *Object Oriented Design Principles, SOLID, ISO/IEC 9126, Code Metrics.*

SOLID İlkelerinin Microsoft VS Code Metriğine Etkisinin Deneysel Olarak Değerlendirilmesi

ÖZ *Yazılımın bakımı, kullanıcı ihtiyaçlarındaki değişiklikler, altyapıda meydana gelen değişiklikler, sistemin çalışma koşullarındaki değişiklikler, öngörülemeyen hataların ortaya çıkması gibi nedenlerle gereklidir. Yazılımın bakım işlemleri için uygunluğu maliyeti düşürmede önemli bir etkendir. Sadece temel nesne tabanlı programlama kavramlarını kullanmak, uygulamalarımızda sürdürülebilir kod yazdığımızı göstermez. SOLID gibi nesneye yönelik tasarım prensipleri bağımlılıkları azaltmak ve yazılım bakımını artırmak ile ilgilidir. ISO/IEC 9126 bakım yapılabilirlikle ilgilidir fakat ISO/IEC 9126 ölçüme ilişkin tim girdilerin bir arada mı yoksa ayrı olarak mı kullanılmaları gerektiği konusunda net değildir. Nitekim, ISO/IEC 9126 pratik olarak veya deneysel tarzda yazılım ölçümlerinin nasıl yapılacağı, bu ölçümlerin nasıl basitçe toplanacağı, ölçümlerin nasıl değiştirilebileceği konusunda rehberlik sağlamaz. Bu çalışmada,*

DOI: 10.5824/1309-1581.2018.4.001.x

Visual Studio (VS) kod metrik aracı yardımıyla ISO / IEC bakım yapılabilirliğin her alt-özelliği değerlendirilmiştir. Bu değerlendirmenin odağı sürdürülebilirlik ve analiz edilebilirlik, test edilebilirlik, değiştirilebilirlik ve kararlılık gibi alt özellikler üzerine odaklanmaktadır. Bir analiz yapmadan önce, ISO / IEC 9126 standardının bakım yapılabilirlik bölümünün her bir alt-karakteristiği özelliklerin ölçümü için beş VS kod metriğine eşlenmiştir. Bu çalışma, nesneye yönelik tasarım ilkelerinin (SOLID) ISO / IEC, VS kod metriği ve SOLID'i ilişkilendirerek kodun bakım yapılabilirliği, karmaşıklığı ve esnekliği üzerindeki etkisini gösterir.

Anahtar Kelimeler: *Nesne Yönelimli Programlama Prensipleri, SOLID, ISO/IEC 9126, Kod Metrikleri.*

1. Introduction

Software-related post-works hold an important place in IT departments. A software system that does not need change over time is unthinkable. Software maintenance is necessary for reasons such as changes in user needs, changes in the operating conditions of the system due to changes in the infrastructure, the occurrence of unforeseen errors. According to the literature, maintenance typically consumes about 40 to 80 percent (60 percent average) of software costs. [1]. Therefore, it is probably the most important life cycle phase.

The suitability of the software for maintenance operations is a significant influence in reducing the cost. Quality and maintenance have an interesting relationship. Trying to improve one quality attribute often degrades another. For example, attempts to improve efficiency often degrade modifiability. [1]. But object oriented design principles can overcome of this problem. Using only basic object oriented programming concepts do not show that we are writing maintainable code in our applications. So any architect, developer, or information technology (IT) professional who designs, builds, or operates applications and services should know how to implement object oriented programming system (OOPS) and use them in right manner, that is where five object oriented principles (also called as SOLID Principles) comes to mind. SOLID is an acronym for the first five object oriented design principles (Single responsibility, Open-closed, Liskov substitution, Interface segregation, Dependency inversion) introduced by Robert C. Martin [2]. These principles, when combined together, make it easy for a programmer to develop software that are easy to maintain and extend over time. [3]. Metric changes on the code are measured by Microsoft Visual Studio (VS) Code Metrics tool. Code metrics in Visual Studio is a tool for measuring the quality and complexity of our code. It provides us various metrics whose values validate our code. [18][23]. VS code metrics are used because we did all the code enhancements on VS.

While maintainability index can give an opinion for determining the maintainability of the source code of a system, it is hard using the maintainability index to the desired effect. Because computed value of the maintainability index does not provide clues about characteristics of maintainability or it is not give clue about taking an action to improve this value. The maintainability index has been proposed objectively determine the maintainability of software systems based on the status of the corresponding source code. In this study each sub-characteristics of ISO/IEC maintainability with help of Visual Studio (VS) code metric tool is assessed. The evaluation was made by associating the metrics with the VS code metric results

for each maintainability characteristic. Before doing an analysis, each sub-characteristics of maintainability part of ISO/IEC 9126 standard are mapped to five VS code metrics for measurement of characteristics.

Specifically, this study contains an assessment of the effect of SOLID principles on the Visual Studio code metrics using a human resource management system project and named as HRS. The system developed with two different ways, without and with solid principles. We captured the code metrics of HRS in the default design and after the implementation of these principles in the second design. We compare the results with the context of the improvements and benefits obtained from the implementation. At the same time although the ISO/IEC 9126 has some usefulness about counting and assessing metrics [20], the results have been assessed within the scope of ISO/IEC 9126 [19], which proposes six main factors that determine overall quality are maintainability, usability, efficiency, portability, functionality and reliability. The focus of this assessment is on maintainability and its sub-characteristics like analyzability, testability, changeability and stability.

The paper is organized as follows. Section 2 provides literature analysis on SOLID principles and code metrics. Section 3 present a brief overview of the SOLID principles and VS code metrics. Section 4 and its sub-sections recapitulate the ISO/IEC 9126 standard for software product quality, focusing on the characteristics of maintainability and provide the application method of design principles to classes and application results of code metrics. Section 5 compares and discusses with related works. The last section summarizes the main findings.

2. Related Work

Although separately each of SOLID design principles as Object Oriented Design Principles have been investigated widely such as effect of quality on software, rules and techniques in object-oriented programming, contribution to maintenance cost etc. There are not much published papers include all SOLID principles and addressing all of these principles which deal with the software effect with Visual Studio code metrics. In paper [4] Al-Ahmad contribute a framework for conceptual modelling and focuses on the conceptual modelling facet of inheritance and suggests better support for it in object oriented programming. He has examined the influence of the Liskov Substitution Principle, interfaces, separate type, and class hierarchies on conceptual modeling. There are some papers mentioned that Liskov Substitution Principle in such papers as [7], [9], [11]. In [5] Zotos presents object-oriented design principles to solve the software crisis between mathematics and computer science. He used all of the design principles contained in this paper. These principles show the right direction of designing and helps in avoiding costly mistakes at the designing stage. In order to write quality code, it is needed to understand the principles and methodologies behind the language.

Deligiannis, Shepperd, Roumeliotis and Stamelos made an empirical investigation of object-oriented design heuristic for maintainability [6]. They aim two goals. First, to investigate the impact of a design heuristic on the maintainability of object-oriented designs. The second goal is to investigate the relationship between OO design heuristic and metrics. A good design

DOI: 10.5824/1309-1581.2018.4.001.x

allows us to easily plug-in new functionality in terms of new classes and new methods without a need to re-implement the results of the previous iteration cycles. In paper [8] Bavota, De Lucia and Oliveto try identifying extract class refactoring opportunities using structural and semantic cohesion measures. They propose an Extract Class refactoring method based on graph theory that exploits structural and semantic relationships between methods. They summarize that during software development the classes of a system undergo continuous modifications making the source code more complex and drifting away from its original design. In particular, due to strict deadlines programmers do not always have a bunch of time to make sure everything conforms to object oriented programming (OOP) guidelines. When the added responsibility grows and breeds, the class becomes too complex and its quality deteriorates. Paper [10] presents an observational study on students' ability to understand and apply design patterns and used Object-Oriented Design Principles, such as Open-Closed, Single Responsibility, Dependency Inversion, Interface Segregation and Liskov Substitution principles. Paper show that the majority of students correctly identified maintenance problems as the main symptom of a poor architecture that according to the general belief that design patterns solve maintenance issues.

Paper [12] introduce an algorithm for the discovery of refactoring and assess Dependency Inversion Principle use Liskov's Substitution Principle and Design by Contract requirements on class contract preservation during sub- classing to become clearer of implementation inheritance. Context aware mobile patient monitoring framework development issue is discussed in [13]. As the paper, design patterns can be used as a method to document application frameworks and design principles are good ideas help software developers to build better design. Design patterns are used as tools for applying the design principles. Five design principles that takes place in this paper support reusability and extensibility. Paper [14] makes models for predicting extract subclass refactoring using object oriented quality metrics. Talk about refactoring that it has several benefits such as enhancing the code's understandability, maintainability, testability. Therefore, design principles provide these properties. Paper [15] try to identify and apply of extract class refactoring in object oriented systems. It talks about a class that should implement only one concept and should only change when the concept it encapsulates evolves.

3. Definition of Solid Design Principles and Used VS Code Metrics

The Single Responsibility Principle – S means that there should never be more than one reason for a class to change [2]. If there is more than one motive for changing a class, then that class is assumed to have more than one responsibility, which results as high coupling. This kind of coupling leads to fragile designs that can break in unexpected ways for any change requirements [16].

The Open Close Principle – O requires software entities like classes, modules and functions should be open for extension, but closed for modification [2]. An entity can allow its behavior to be extended without modifying its source code or a class should be easily extendable without modifying the class itself. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

The Liskov Substitution Principle – L requires derived type must fully support the substitution of their base types. [2] Every subclass/derived class should be substitutable for its base/parent class. If any module is using a Base class then the reference to that Base class can be replaced with a Derived class without affecting the functionality of the module. While implementing derived classes, derived classes just extend the functionality of base classes without replacing the functionality of base classes.

The Interface Segregation Principle – I requires clients should not be forced to depend upon interfaces that they do not use [2]. When a client depends upon a class that contains interfaces that the client does not use, but that other clients do use, then that client will be affected by the changes that those other clients force upon the class.

The Dependency Inversion Principle – D requires High Level Modules should not depend upon Low Level Modules. Both should depend upon abstractions. Abstractions should not depend upon details. However, details should depend upon abstractions [2]. Entities must depend on abstractions not on concretions. It states that the high-level module must not depend on the low-level module, but they should depend on abstractions.

Then how important are these principle? Is one more important than the other is or are they all equally? In this experiment we will to address these questions.

On the other hand, code complexity deals with the lack rate and robustness of the application. Complex code is difficult to test and it is difficult to maintain. When a developer writes a code, developer must adhere boundary values of metrics to ensure the code is well written, understandable and maintainable. Code Metrics is an important measure that let us understand the complexity and maintainability of the code. These metrics are specified that estimation how error prone the program source code is due to its complexity or which are most likely to cause problems in the future. Developer can understand which classes, which methods, which module should be reworked or refactored. Visual Studio uses five code metrics to help users understand their code better [18] [23]. They are maintainability index, cyclomatic complexity, the depth of inheritance, class coupling and the line of code.

Maintainability Index (MI) is a metric aimed at assessing software maintainability. The Maintainability Index was introduced at the International Conference on Software Maintenance in 1992 [17]. MI has evolved into numerous variants. It has been successfully applied to a number of industrial strength software systems. It is based on three code metrics: Namely the Halstead Volume, the Cyclomatic Complexity and Lines of Code. It is based on the following formula [18]:

Maintainability Index (MI) =
MAX (0, (171 - 5.2 * ln (Halstead Volume)
- 0.23 * Cyclomatic Complexity
- 16.2 * ln (Lines of Code)) * 100 / 171)

DOI: 10.5824/1309-1581.2018.4.001.x

Maintainability Index (MI) is a composite metric that incorporates a number of traditional source code metrics into a single number that indicates relative maintainability. The MI is comprised of weighted Halstead Volume (HV), McCabe's cyclomatic complexity (CC) and Lines of Code (LOC). MI calculates an index value between 0 and 100 that represents the relative ease of maintaining the code. A high value means better maintainability. As can be seen from the formula increasing of the cyclomatic complexity or line of code reduces the value of maintainability index. As pointed by Van der Meulen and M.A Revilla [25], there are very strong connections between LOC and HV, LOC and CC. The study provides an approximate expression that have been used in our study for MI value.

The Cyclomatic Complexity (CC) measures the structural complexity of the code. It is created by calculating the number of different code paths in the flow of the program. Depends on how many different control flow of your code can execute depending on various inputs. A program that has complex control flow will require more tests to achieve good code coverage and will be less maintainable. The cyclomatic complexity definitely reveals a code smell.

The Depth of Inheritance indicates the number of class definitions that extend to the root of the class hierarchy. The deeper the hierarchy the more difficult it might be to understand where particular methods and fields are defined or redefined. The idea is that if more types exist in an inheritance hierarchy, the code will likely be more difficult to maintain as a result. However, a high depth of inheritance can also indicate a greater level of code reuse. This means that it is difficult to say what a good depth is. Remark that, (Microsoft) MS Visual Studio does include a code analysis rule, which generates a warning when an inheritance hierarchy is more than four levels deep.

The Class Coupling measures the coupling to unique classes through parameters, local variables, return types, method calls, generic or template instantiations, base classes, interface implementations, fields defined on external types, and attribute decoration. Good software design dictates that types and methods should have high cohesion and low coupling. High coupling indicates a design that is difficult to reuse and maintain because of its many interdependencies on other types. If we have a class that does not reference other class then its class coupling will be zero whereas if we refer to various classes in our class (like creating complex type properties) then it will increase class coupling.

The Lines of Code indicates the approximate number of lines in the code. The count is based on the intermediate language code and is therefore not the exact number of lines in the source code file. A very high count might indicate that a type or method is trying to too much work and it should be split up. It might also indicate that the type or method might be hard to maintain.

4. Mapping of VS Metrics to ISO/IEC 9126 Software Product Quality

ISO/IEC 9126 defines a quality model that comprises 6 characteristics and 27 sub characteristics of software product quality. ISO/IEC 9126 also defines one or more metrics to measure each of its sub characteristics [24]. For example, the quality level of a software product's maintainability can be represented by measured values of its sub characteristics. The

ISO/IEC 9126 standard is divided into four parts. Quality model, internal metrics, external metrics and quality in use metrics. The first three parts are concerned with describing and measuring the quality of the software product, the fourth part evaluates the product from the user point of view. Internal quality is believed to impact external quality, which in turn affects quality in use.

Internal quality is assessed based on four characteristics (functionality, efficiency, maintainability, portability) and their respective sub-characteristics. These are evaluated by employing a set of metrics. For instance, the quality level for maintainability takes into account the measured values of four sub-characteristics. The above quality characteristics are abstract concepts and therefore not directly measurable and observable. Each of them is characterized by a set of sub-characteristics.

In this study, we focused on the maintainability characteristics that sub-characterized:

- Analyzability: Degree to which the software product can be diagnosed for deficiencies or causes of failures in the software, or for the parts to be modified to be identified.
- Changeability: Degree to which the software product enables a specified modification to be implemented or the ease with which a software product can be modified.
- Stability: Degree to which the software product can avoid unexpected effects from modifications of the software.
- Testability: Degree to which the software product enables modified software to be validated.

However, in new version of ISO/IEC, modularity and reusability are added to sub-characteristics [21].

- Modularity: Degree to which a system or computer program is composed of discrete components such that a change to one component has minimal impact on other components.
- Reusability: Degree to which an asset can be used in more than one software system or in building other assets.

ISO/IEC 9126 is not clear about whether all inputs to measurement should be used together in conjunction or whether they should be used as appropriate or available. Indeed, ISO/IEC 9126 provides no guidance, heuristics, rules of thumb, or any other means to show how to trade off measures, how to weight measures or even how to simply collate them [20].

Since our main aim was to evaluate maintainability coupled with MS VS standard environment, each sub-characteristics of maintainability part of ISO/IEC 9126 standard are mapped to five VS code metrics for measurement of characteristics. The changeability characteristic of a system is linked to properties such as complexity of the source code. Source

code complexity is measured in terms of cyclomatic complexity. The analyzability characteristic of a system is effected from lines of code (LOC) and complexity attributes. The testability characteristic of a system is effected from complexity and LOC attributes. Stability is effected from coupling. A larger system requires, in general, a larger effort to maintain. Higher size causes lower analyzability and it is hard to understand the system. The complexity property of source code refers to the degree of internal disorder of the source code. Large code units are complex. In addition, complex units are difficult to analyze and difficult to test. İf there is duplication in the source code then it is difficult to maintain it. Excessive duplication makes a system larger than it needs to be. In addition, it effects the analyzability and changeability. VS code metrics and the mapping of system characteristics onto these properties is shown in Table 1 [22].

Table 1. Mapping system characteristics onto code metrics

Maintainability Sub-Characteristics	Code Metrics Value
Analyzability	1. Lines of Code (LOC) 2. Cyclomatic Complexity (CC) 3. Number of Method & Weighted Methods in Class (WMC)
Changeability	1. LOC 2. CC 3. Depth of Inheritance (DIT)
Stability	1. Coupling
Testability	1. LOC 2. CC
Modularity	1. Coupling 2. DIT
Reusability	1. Coupling 2. WMC

4. The Effect of Application of SOLID Design Principles

The project is a Human Resource Management program. It is working on n-tier architecture. The project has modules about employee which employee data management, personnel tracking, accounting and payroll system, reporting etc. Changes made in the project were made in business and UI layer in the architecture. When we take the class diagram in the Microsoft Visual Studio, we see that the software has 48 class in working layer. It is indicated Figure 10.

In the first phase of work, Visual Studio (VS) code metric tool started and default metric values of the whole project received before making any change. It is shown on the Table 2. In table 2, Personnel refers to the whole solution. General, Report and Payroll represent a project in the solution. ListUpdate, takeFormData and dataSave indicates a method. The modules to be modified are selected within the range of low MI values. İn the first stage, only one method was modified according to the SOLID design principles. The changes were made in order. Modified method is about subsistence money calculation. The task of method is to get form data and assign these data to list object. The method does checks about journal control when doing these operations. There are several if blocks in the method. Code metric values recalculated after every change made.

Table 2. VS Code Metrics Result

	MI	CC	DIT	Coupling	LOC
Personnel	73	1593	5	271	5632
Personnel. General	71	97	5	50	428
Personnel. Report	68	109	5	93	365
Personnel. Payroll	73	452	5	79	1835
listUpdate	49	5		15	24
takeFormData	40	6		21	46
dataSave	49	5		15	24

Single Responsibility Principle: To solve a problem, find the sub problems in the domain that working in. Divide each sub problem into sub-sub-problems until reaching the point where such a mini problem has just one single task left. Then solve each of these mini problems in its own class. Initially, we had a method that used to retrieve form data and bind them to list items. It is shown in the Figure 1. In addition, there were "if blocks" in the method for controlling data. Controlling data is for assurance of input validation.

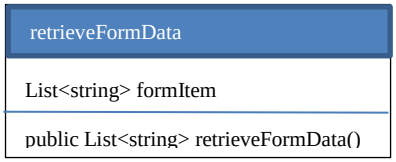


Figure 1. Initial version on SRP

To implement this principle within the method, list items are declared in another class. It is invoked from there. All controls such as steps for form control and assignment of data to list items (controlListData) which exist in the single incohesive large method is separated to different cohesive methods. Each new mehod is simple and has just one single responsibility. Result classes after applying SRP is shown in Fiure 2. At the end of single responsibility principle refactor, Visual Studio code metric tool was run again. Maintainability index increased by 7 percent. In addition, class-coupling value decreased. On the other hand, according to ISO/IEC 9126 system characteristics stability, modularity and reusability have increased. Because coupling value has decreased.

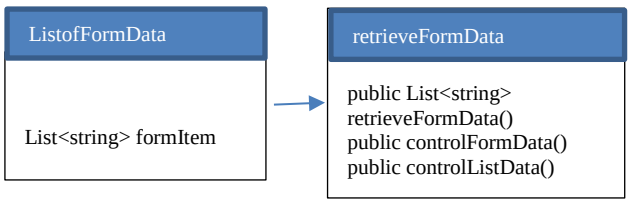


Figure 2. After applying refactoring on SRP

DOI: 10.5824/1309-1581.2018.4.001.x

Open Closed Principle: An entity can allow its behavior to be modified without altering its source code. Modules that adhere to open-closed principle have two primary attributes. First is open for extension that it is possible to extend the behavior of the module as the requirements of the application change. Second is closed for modification that extending the behavior of the module does not result in the changing of the source code or binary code of the module itself. There are controls about detecting journal entries and filtering operations about type of journal data in the modified class.

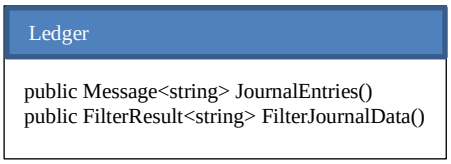
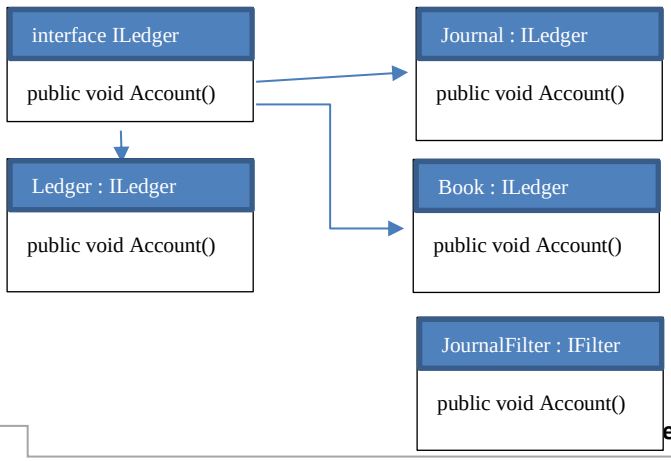


Figure 3. Initial class before applying OCP

To implement open closed principle all controls and filtering processes were reorganized. To do this, we put the implementation of filtering or implementation of controlling in another class. After applying implementation, we do not have to modify the new class for filtering or for controlling new criteria. Because the behavior of the requested operations are marshalled to the new class. Moreover, we can extend the behavior of the new class to support new criteria. Because all we simply have to do is, pass in a new class. Therefore, it is open for extension. Subclass provides extension by not putting the abstraction in codified interfaces but in over ridable behavior. It often leads to composite systems and overall realizes more opportunities for reuse. At the end of open closed principle implementation, Visual Studio code metric tool was run again. Maintainability index (MI) increased by about 4.5 percent. Cyclomatic Complexity did not change, class coupling decreased by about 6.25 percent. İn addition to MI, stability, modularity, reusability, analyzability and changeability have increased. Because some of them depend on coupling and coupling is decreased. In addition, because of the ease of adding new features or changing existing ones analyzability and changeability characteristics were positively affected.



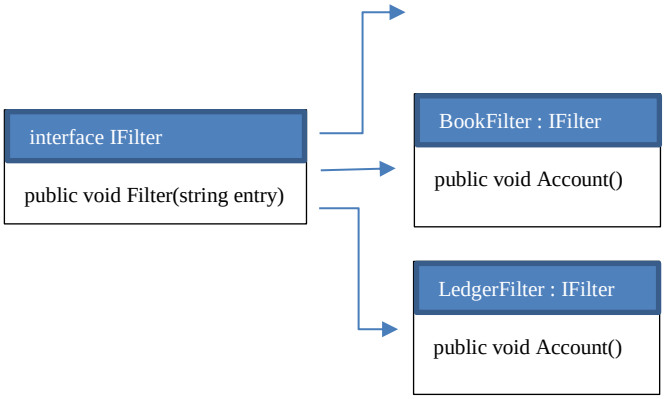


Figure 4. Diagram after applying OCP

Liskov Substitution Principle: References to base classes must be able to use objects of derived classes without knowing it. If a software has a base class and a few number of subclasses, the rest of the code should always refer to base and not to subclasses. This principle is just an extension of the Open Close Principle.

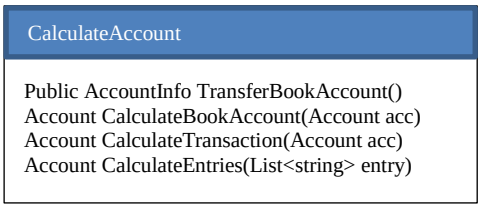
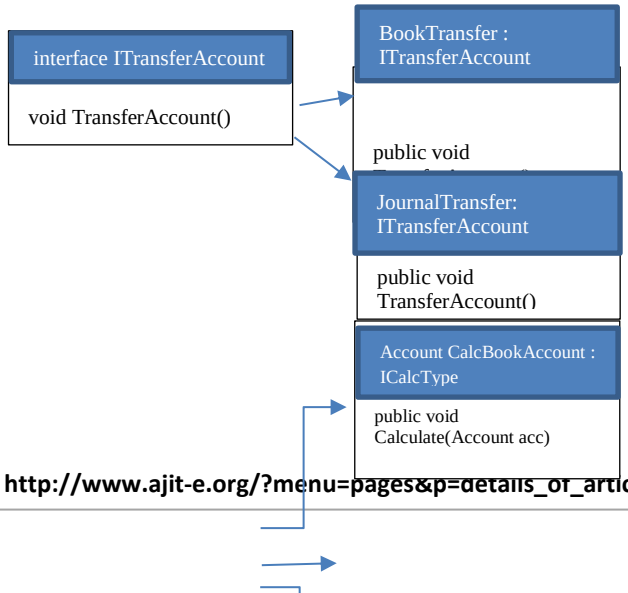


Figure 5. First class before applying LSP

Initially, we had class calculateAccount that contains methods about book of account for accounting monetary transactions. However, method of calculation can be differ between accounts. In addition, we had another class getAccount derived from calculateAccount class. In the method of getAccount class calculations are done as type of account information. Method of calculation for BookAccount, Transaction and Entries was diverging according to the account information with if blocks. For applying this principle, calculateAccount is re-written as the type of account information and calculateAccount class is derived from the related class.



DOI: 10.5824/1309-1581.2018.4.001.x

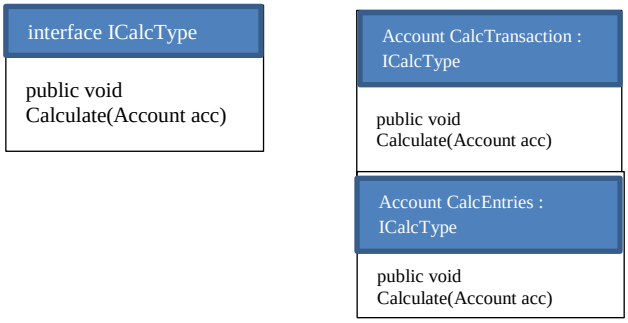


Figure 6. Diagram after applying LSP

After making changes for Liskov Substitution Principle, maintainability index of the project increased by about 1.3 percent. However, cyclomatic complexity increased by about 0.15 percent. If we assess this according to ISO/IEC 9126, base types can be reused and the derived types can be changed.

Interface Segregation Principle: No client-code-object should be forced to depend on methods it does not use. Each code object should only implement what it needs, and not be required to implement anything else. The interface segregation principle is all about reducing code objects down to their smallest possible implementation and removing dependencies the object does not need to function properly. Because of applying this principle is to have small and focused interfaces that define only what is needed by their implementations. For implementing this principle in our project, the main interfaces that keep the journal records are divided into interfaces that are smaller but contain no unnecessary objects. Initially, we had interface IAccountRecord that contains bookRecord, ledgerRecord and journalRecord methods. But every method differ from anothers in context. To apply this principle, IAccountRecord is divided to IBookRecord, ILedgerRecord and IJournalRecord interfaces and every method is derived from related interface. At the end of the interface segregation principle implementation, Visual Studio code metric tool was run again. There was no change in the expectation that the principle would increase the maintainability index. However, Cyclomatic Complexity increased by about 0.4 percent. Class coupling decreased. Therefore, the goal of this principle is helping decouple the application so that it is easier to maintain. It is improving flexibility and possibility of reuse.

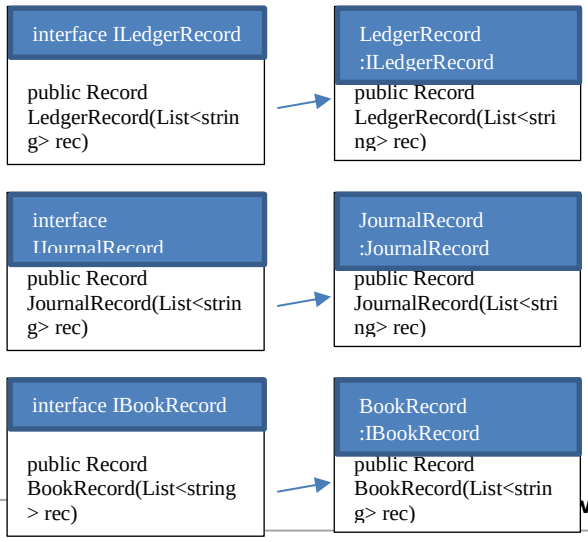
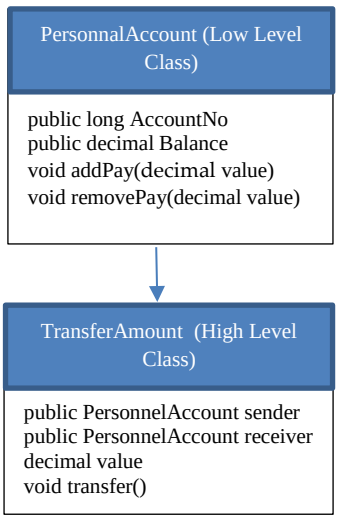


Figure 7. Diagram after applying ISP

Dependency Inversion Principle: Primarily concerned with reducing dependencies amongst the code modules. It needs the low-level objects to define contracts that the high-level objects can use without the high-level objects needing to care about the specific implementation the low-level objects provide. In the project there are classes and interfaces for reporting and notification. Reports are written in the database or in different formats. Notification was using as sms or e-mail. To implement this principle the report generation task and printing part separated to different interfaces. On the notification part, an abstraction is introduced and notification methods implement it. As a result, it is allowed that both high level and low level classes to rely on abstractions. At the end of Dependency Inversion Principle implementation, Visual Studio code metric tool was run again. Maintainability index increased as expected. Already expected that this principle be primarily concerned with reducing dependencies. As a result of interface separation, high-level policy modules and low-level detail modules were reusable and maintainable.

For dependency inversion principle, a class about worker amount and transfer to balance sheet is changed. First version is shown on Figure 8. In the first version the high level TransferAmount class is depend on the low level PersonnelAccount class. This increase the coupling. The sender and receiver references the PersonnalAccount type in the TransferAmount class. Therefore, if another account types are not taking place in the PersonnalAccount then it is impossible to use them. If we want to use for aiming only adding pay for other class, the new class have to be inherited from PersonnelAccount. However, in this situation new class would not apply the removal of pay. This violates the Liskov Substitution Principle. On the other hand, if we want to change TransferAmount class then this violates the Open-Closed Principle. If we make a change in the PersonnelAccount class then it effects the TransferAmount class. Similar problems can be arise and the software can be rigid when the software grows. Times are taken when changing or extending functionality. For these reasons, Dependency Inversion Principle is applied to software. After applying DIP, second version is shown on Figure 9.



DOI: 10.5824/1309-1581.2018.4.001.x

Figure 8: First version of the classes

After applying DIP higher level classes refer to dependencies using interface or abstract classes. It decreases the coupling. Lower level class implements the interfaces or makes inheritance that inherited from abstract classes. So new classes can be used without any impact. Flexibility of software improves. Implementing this principle needs extra effort and code view can be complex but it is handy for maintainability. Independence of classes increase reusability.

Table 3: VS Code Metric Result

	MI	CC	DIT	Coupling	LOC
Personnel	79	1561	5	259	5629
Personnel. General	73	98	5	49	430
Personnel. Report	68	109	5	92	367
Personnel. Payroll	75	453	5	76	1833
listUpdate	51	5		14	23
takeFormData	43	4		16	33
dataSave	52	5		14	19

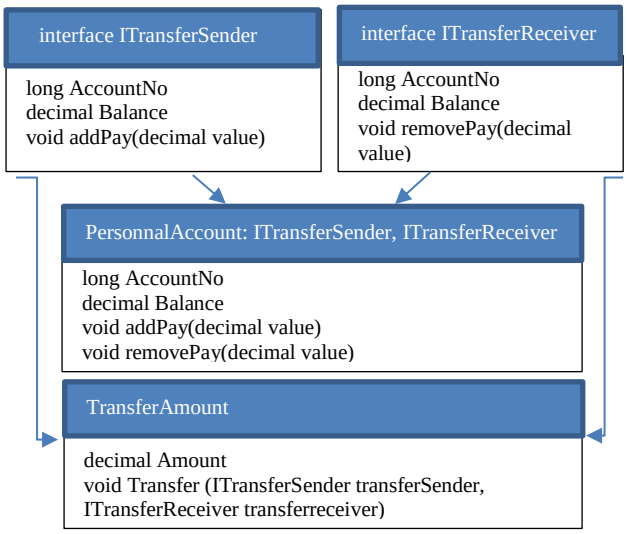


Figure 9: Second version after applying DIP

All results may vary depending on the coding technique. However, maintainability index value for all principles will increase. The metric values formed after the application of all the principles are shown on the Table 3.

Figure 10. Class Diagram of the Project

5. Discussion with Related Work

There is not much work about Single Responsibility Principle on literatur. But when we search with keyword about refactoring, god class dividing, seperation of concern then we see that there are works and papers. Researches have been made on the impact of refactoring on code quality and maintenance cost in general by considering more than one project. In [28] Hegedus and others made a study about empirical evaluation of software maintainability. The concept of refactoring is an essential part of the development process. Fowler [29] proposed that code smells should be the primary technique for identifying refactoring opportunities in the code. The paper compares the differences in maintainability and source code metrics as refactored and non-refactored source code elements. Result of the study source code elements subjected to refactorings had significantly lower maintainability than elements not affected by refactorings. Moreover, refactored elements had significantly higher size related metrics, complexity, and coupling. Also these metrics changed more significantly in the refactored elements. In our research we show that if source code is refactored as the principles then code can reach the high cohesion, low coupling, high maintainability index values. Another study [30] states that single refactorings only make a very little changes on maintainability but a whole refactoring period can significiantly increase maintainability. In [31] mention the existing literature lacks observations about the relations between metrics/code smells and refactoring activities performed by developers. But our paper indicates relation between metrics and refactoring activities. We show that code metrics depend on a good design refactoring. Other researches like [32], [33] state extract class and move method are found the most frequently considered refactoring activities. For making a good refactoring as the SOLID principles that we state, developer should make extracting class and moving method.

6. CONCLUSION

DOI: 10.5824/1309-1581.2018.4.001.x

The SOLID principle aims reducing dependencies and increasing maintainability. Every principle require additional time and effort spent to apply it during the design time and they can increase the complexity of code because of increasing number of interfaces or classes. However, they produce a flexible design, loose coupling, and higher maintainability. Code is more robust, more stable and better understandable. İn addition to these Visual Studio code metric values can give an insight about maintainability and complexity of the code. The developer can make an assessment about code with help of code metric values before beginning maintenance task or refactoring.

In the ISO 9126 and VS-SOLID mapping, coupling deals with stability and modularity. Mitigating on the technologies or evolving changes is critical for software developers to stabilize a system and preserve its design. Instable software tends to increase maintenance cost up to 75 % of the software total costs [26, 27]. Therefore, stability is very important. Applying stability early at the model level enables the developers to improve maintainable software and reduce the software cost. Stability can enhance reusability, as it focusses on providing code part that remain unchanged over time. This ensures a stable core design and thus a stable software. In order for the software to stabilize, it is important to emphasize that the coupling is low in a software. If coupling is low, then it can need making impact analysis less.

This work shows that SOLID design principles increase the maintainability of the code, generally reduce complexity of the code and reduce dependency, provide flexibility to the code. Design principles improve the separation of concern through weaker coupling and stronger cohesion. However, if these principles are applied without measure then some potentially undesirable consequences may occur. They are the proliferation of relatively small concrete classes, the proliferation of abstract classes and interfaces, increasing in the depth of the inheritance tree. As a result, Visual Studio code metrics can tell which class and which method should be studied. Moreover, code can be structured better with the help of SOLID design principles. Further study could be to investigate the SOLID effect with different code metric measurement programs by making more changes in a larger project or it could be to build a design principle compliant architecture infrastructure and force developers to code accordingly.

REFERENCES

[1] Robert L. Glass , "Frequently Forgotten Fundamental Facts about Software Engineering", An Article in IEEE Software May/June 2001

[2] R. C. Martin, “Design Principles and Design Patterns”, [Online]. Available: http://www.objectmentor.com, 2000

[3] Sandi Metz (Duke University) , “SOLID Object-Oriented Design”, Talk given at the 2009 Gotham Ruby Conference in May, 2009. Online at http://www.youtube.com/watch?v=v-2yFMzxqwU

[4] Walid Al-Ahmad, “A framework for conceptual modeling in OOP”, Journal of the Franklin Institute, 2006

[5] Kostas Zotos, “Object-oriented design principles in mathematics”, Applied Mathematics and Computation, 2006

[6] Ignatios Deligiannis, Martin Shepperd, Manos Roumeliotis, Ioannis Stamelos, "An empirical investigation of an object-oriented design heuristic for maintainability", The journal of system and software, 2001

[7] Magiel Bruntink, Arie van Deursen, "An empirical study into class testability", The Journal of System and Software, 2006

[8] Gabriele Bavota, Andrea De Lucia, Rocco Oliveto , "Identifying Extract Class refactoring opportunities using structural and semantic cohesion measures", The Journal of Systems and Software, 2011

[9] David Lievens, William Harrison, "Abstraction over implementation structure with symmetrically encapsulated multimethods", Science of Computer Programming, 2013

[10] Alexander Chatzigeorgiou, Nikolaos Tsantalis, Ignatios Deligiannis , "An empirical study on students ability to comprehend design patterns", Computers & Education, 2008

[11] Gabriale Arevalo, Stephane Ducasse, Silvia Gordillo, Oscar Nierstrasz , "Generating a catalog of unanticipated schemas in class hierarchies using Fomal Concept Analysis", Information and Software Technology, 2010

[12] Vassilis E. Zafeiris, Sotiris H. Poulias, N.A. Diamantidis, E.A. Giakoumakis, "Automated refactoring of super-class method invocations to the Template Method design pattern", Information and Software Technology, 2016

[13] Mahmood Ghaleb Al-Bashayreh, Nor Laily Hashim, Ola Taiseer Khorma, "Context- Aware Mobile Patient Monitoring Framework Development", 2013 International Conference on Electronic Engineering and Computer Science, 2013

[14] Jehad Al Dallal , "Constructing models for predicting extract subclass refactoring opportunities using object-oriented quality metrics", information and Software Technology, 2012

[15] Marios Fokaefs, Nikolaos Tsantalis, Eleni Stroulia, Alexander Chatzigeorgiou, "Identification and application of Extract Class refactorings in object-oriented systems", Journal of Systems and Software, 2012

[16] Harmeet Singh, Syed Imtiyaz Hassan , "Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment", International Journal of Scientific & Engineering Research, April-2015

[17] Paul Oman and Jack Hagemeister. "Metrics for assessing a software system's maintainability". Proceedings International Conference on Software Mainatenance (ICSM), 1992, pp 337-344.

[18] www.microsoft.com, 01.08.2017

[19] ISO 9126-1 Software Engineering - Product Quality - Part 1: Quality Model, 2001.

[20] Hiyam Al-Kilidar, Karl Cox, Barbara Kitchenham, "The Use and Usefulness of the ISO/IEC 9126 Quality Standard", International Symposium on Empirical Software Engineering, 2005

[21] ISO/IEC 25010:2011, http://www.iso.org/iso/catalogue_ detail.htm?csnumber=35733, 01.08.2017

[22] Morteza Asadi, Hassan Rashidi, "A Model for Object Oriented Software Maintainability Measurement", I.J. Intelligent Systems and Application (MECS), 2016

[23] https://docs.microsoft.com/tr-tr/visualstudio/code-quality/code-metrics-values, 01.08.2017

[24] Ho-Won Jung, Seung-Gweon Kim, Chang-Shin Chung, "Measuring Software Product Quality: A Survey of ISO/IEC 9126", IEEE Software, vol. 21, pp. 88-92, 2004

[25] Meine J.P. van der Meulen, Miguel A. Revilla, "Correlations between Internal Software Metrics and Software Dependability in a Large Population of Small C/C++ Programs", 18th IEEE International Symposium on Software Reliability Engineering, 2007

[26] Galorath, D.D., "Software total ownership costs: development is only job one" Softw. Tech. News, 23–32, 2008

[27] Chen, J.-C., Huang, S.-J., "An empirical analysis of the impact of software development problem factors on software maintainability" System Software. 82, 981–992, 2009

[28] Péter Hegedűsa, István Kádárb, Rudolf Ferenc, Tibor Gyimóthyb, "Empirical evaluation of software maintainability based on a manually validated refactoring dataset", Information and Software Technology, 2017

[29] M. Fowler, "Refactoring: Improving the Design of Existing Code", Addison-Wesley, 1999.

[30] Gábor Szoke, Gábor Antal, Csaba Nagy, Rudolf Ferenc, Tibor Gyimóthy, "Empirical study on refactoring large-scale industrial systems and its effects on maintainability", The Journal of Systems and Software, 2016

[31] Gabriele Bavotaa, Andrea De Luciab, Massimiliano Di Pentac, Rocco Olivetod, Fabio Palombab, "An experimental investigation on the innate relationship between quality and refactoring", The Journal of Systems and Software, 2015

[32] M. Gatrell, S. Counsell, "The effect of refactoring on change and fault-proneness in commercial C# Software", Science of Computer Programming, 2014

[33] Jehad Al Dallal, "Identifying refactoring oppurtunities in object oriented code: A systematic literatur review", Information and Software Technology, 2014