ISSN 2229-5518

Effect of SOLID Design Principles on Quality of Software: An Empirical Assessment

Harmeet Singh, Syed Imtiyaz Hassan

Abstract— Design is one of the important phases of software development life cycle, which has an impact on entire life cycle of the project. If the design is good then all other phases of Software development life cycle like coding, maintenance and support will be stress and hassle-free. The availability of lot of data has proved that designing has a significant role and it directly impacts the quality and performance requirements. The world of Agile Development today has led the developer to compete with the latest features and technologies in the market. However for the naïve users it is difficult to maintain the design quality if there are no standard guidelines available. The design quality is sometime depend on developer expertise and experience only, so we must have standard and proved design guidances to work. If the software design is in proper accordance with the principles and patterns then it can increase the software re-usability, maintainability and scalability. This research paper focuses on empirical analysis to prove the SOLID design principles guidelines by using a working prototype, applying the design principles to it and then evaluating the prototype by using different metrics.

Index Terms— Design Principles, CKJM metrics, Software design, SOLID Principles, Software quality.

---

1 Introduction

Software design helps to imagine the overall system and reduces the cost involved in developing and supporting the project [1]. Since it is not an easy task to identify the feasibility of the actual requirements at the very start of the project, therefore the design should support scalability which will allow the induction of the new requirements into the software architecture. To support scalability there are a few important factors [2] on which the designer should concentrate while thinking about the software design so as to avoid redesigning. These factors include rigidity, fragility, immobility and viscosity. Rigidity specifies the difficulty measure of changing the software. Fragility is the tendency of the software to break every time it is changed. Immobility is the inability to reuse software from the other projects or parts of software from the same project. Viscosity is the inability to preserve the design of the system which can degrade if a proper solution is not incorporated pertaining to any changes in the system requirement.

The presence of these four factors results in a poor architecture. Any application that demonstrates these factors is actually suffering from a bad design. To handle these factors we have some set of guidelines introduced by Robert Martin [2] called Design Principles. This paper contains an empilical assessment of the effect of SOLID principles on the software quality using a small project. This project is named as Payroll System. We are going to implement this system with two different designs, without and with solid principles. We captured the violation of these principles in the first design and the improvements and benefits obtained from the implementation of these principles in the second design. We compare the results by generating the CKJM (Chidamber and Kemerer Java Metrics) for both the cases and proved these design guidance.

- *Harmeet Singh, Department of Computer Science, Jamia Hamdard, New Delhi, India, singh.meet@gmail.com*
- *Syed Imtiyaz Hassan, Department of Computer Science, Jamia Hamdard, New Delhi, India, s.imtiyaz@gmai.com*

2 Overview of SOLID Design Principles

2.1 S-The Single Responsibility Principle

"There Should Never Be More Than One Reason for a Class to Change [2]." The Single Responsibility Principle (SRP) is considered to be one reason for change. 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.

2.2 O-The Open Close Principle

"Software Entities like classes, modules and functions should be open for extension, but closed for modification [2]." When a single change to a program results in a cascade of changes to dependent modules, that program exhibits the undesirable attributes that we have come to associate with “bad” design. The program becomes fragile, rigid, unpredictable and unreusable. The openclosed principle attacks this in a very straightforward way. It says that you should design modules that never change. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works.

2.3 L-The Liskov Substitution Principle

"Derived type must fully support the substitution of their base types. [2]" Functions that use pointers or references to base classes must be able to use objects of the derived without knowing it. This is related to the substitution property. [3] The importance of this principle becomes obvious when you consider the consequences of violating it. If there is a function which does not conform to the LSP, then that function using a pointer or reference to a base class must know about all the derivatives of that base class.

2.4 I-The Interface Segregation Principle

"Clients Should Not Be Forced to Depend Upon Interfaces

ISSN 2229-5518

That They Do Not Use [2]." A change in an unrelated Interface can result into an inadvertent change in the client code. This results in an inadvertent coupling between all the clients. ISP suggests that clients should not know about them as a single class. Instead, clients should know about abstract base classes that have cohesive interfaces.

2.5 D-The Dependency Inversion Principle

"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]." We should decouple high level modules from low level modules, introducing an abstraction layer between the high level classes and low level classes. To conform to the principle of dependency inversion, we must isolate this abstraction from the details of the problem.

3 Case A: Design Without Solid Principles

The first design implementation is random and does not include the SOLID principles. The SalaryCalculator class [Fig.1] is responsible to calculate the salary for different employees in the system. And the contract for this class is defined in ICalculate interface [Fig.1]. There is a separate class TaxCalculator[Fig.1] to calculate the tax on salary, interface for this class is same as ICalculate. The API calculateTax(double salary)[Fig.1] is called by each Employee class. For e.g. every instance of the Professor class is using an instance of TaxCalculator to calculate the tax on its salary

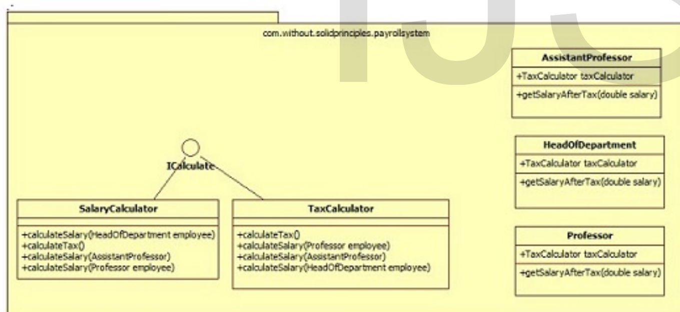


Figure 1: Design without SOLID Principles

4 Problem In The Design

The problems in the design occur if the requirements are changed. Suppose we change the requirement as follows-

- There should be 1% tax for those employees whose salary is more than 50000 or,
- We need to add one more Employee "LabAssistant" who will get 35000 per month.

This will require changes in code –

- Add one more class for the LabAssistant employee.
- Add another method in ICalculate interface and SalaryCalculator class.
- Add logic to check whether the salary is more than 50000 in TaxCalculator.

5 Violation Of Solid Principles

Implementing changes in the code to accommodate the new requirements indicates that there could be possibility of bad design.

5.1 Violation of SRP

- SalaryCalculator class should not be concerned about the type of employee. Instead, it should be responsible for calculating the salary only rather than working with the employee details.
- Employees should not be responsible to calculate their tax on the salary.

5.2 Violation of OCP

- Both the class and the interface code for SalaryCalculator need to be changed while adding a new Employee in the system.

5.3 Violation of LSP

- Since the base class SalaryCalculator is tightly coupled with the employee types, we cannot use this logic for any other type of employees. Such base class behaviour does not make any sense for the derived class and thus we cannot use derived class object to a base class reference in the current system.

5.4 Violation of ISP

- The ICalculate interface contains the different contractual API in a single interface. Instead, the calculation of salary and tax should be segregated in different contracts.

5.5 Violation of DIP

- The Employee classes are tightly coupled with TaxCalculator dependency since all Employee classes are creating a new instance of TaxCalculator class. This any property change in TaxCalculator class requires a change in all classes where the dependency has been tightly coupled.

6 Case B: Design With Solid Principles

The second approach has been designed by inducing SOLID design principles resulting in the addition of few interfaces and classes. In the new design there is an IEmployee interface, which defines the contract for any type of employee in the system. And each employee is now responsible to provide its

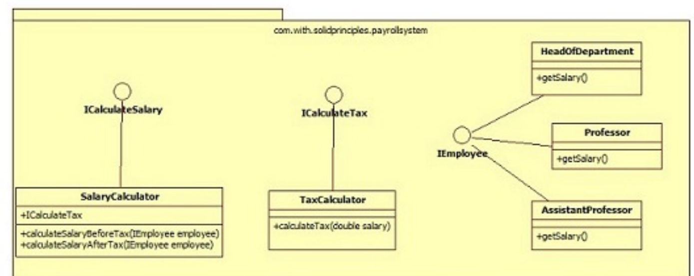


Figure 2: Design with SOLID Principle

IJSER © 2015

present salary to SalaryCalculator. The SalaryCalculator can now calculate the salary by applying the appropriate rules.
There are different interfaces now defining the contracts for salaryCalculator and TaxCalculator. The TaxCalculator can be used by SalaryCalculator only to calculate the salary after tax.

7 Reflection of Solid Principles

With the use of design principles, there would be minimal impact on the product to accommodate the new requirements in the system.

7.1 Reflection of SRP

- Every class in the system has its separate responsibilities. SalaryCalculator class is only responsible for salary calculation without being concerned about the type of employee.

7.2 Reflection of OCP

- The addition of a new employee does not require any changes in SalaryCalculator class except for the addition of a new class of IEmployee type.

7.3 Reflection of LSP

- If we want to use the same SalaryCalculator logic for any other type of employees (other than the Head of Department, Professor or Assistant Professor), then the current base class can be used as a pointer to the derived classes as it is not dependent on the type of Employee. Employee should only be of IEmployee interface type.

7.4 Reflection of ISP

- The interfaces have been segregated as per the responsibilities. The ICalculateSalary interface has been defined for SalaryCalculator and ITaxCalculator interface has been defined for TaxCalculator, giving a clear meaning to the contracts.

7.5 Reflection of DIP

- The classes dealing with Employees do not need to worry about the calculation of tax. The SalaryCalculator removes this dependency and uses a single instance of taxCalculator.

8 CKJM Metrics analysis for both cases

CKJM [4] is an open source command line tool that calculates CK metrics [5] for Java programs. The CKJM tool calculates object-oriented metrics [6] by processing the byte code of compiled Java files. The following six metrics proposed by Chidamber and Kemerer are calculated for each Java class. [7]

8.1 WMC - Weighted Methods per Class

- The WMC metric is simply the sum of the complexities of its methods. As a measure of complexity we can use the cyclomatic complexity, or we can abritrarily assign a complexity value of 1 to each method. By default, CKJM assigns a complexity value of 1 to each method, and therefore the value of the WMC is equal to the number of methods in the class.

8.2 DIT - Depth of Inheritance Tree

- The DIT metric provides a measure of the inheritance levels for each class. In Java the minimum value of DIT is 1 since all the classes inherit the default 'Object' class.

8.3 NOC - Number of Children

- The NOC metric simply measures the number of immediate descendants of the class.

8.4 CBO - Coupling Between Object Classes

- The CBO metric represents the number of classes coupled to a given class. This coupling can occur through method calls, field accesses, inheritance, arguments, return types, and exceptions.

8.5 RFC - Response for A Class

- The RFC metric measures the number of different methods that can be executed when an object of that class receives a message. CKJM gives a rough approximation about the response set by simply inspecting method calls within a class.

8.6 LCOM - Lack of Cohesion In Methods

- The LCOM metric counts the sets of methods in a class that are not related by the sharing of class's method(s). The original definition of this metric (the one used in CKJM) considers all the method pairs of a class. The lack of cohesion in methods is then calculated by subtracting the number of method pairs that share a field access from the number of method pairs that don't. share a field access the number of method pairs that do.

8.7 NPM - Number of Public Methods

- The NPM metric simply counts the methods in a class that are declared as public. It can be used to measure the size of an API provided by a package.

Table 1 CKJM Metrics Of WithOut Design Principle

Class Name	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM
AssistantProfessors	2	1	0	1	5	0	3	2
Professors	2	1	0	1	5	0	4	2
Main	2	1	0	2	12	1	0	2
ICalculate	5	1	0	3	5	10	2	5
SalaryCalculator	6	1	0	4	7	15	1	6
HeadOfDepartment	2	1	0	1	5	0	3	2
TaxCalculator	6	1	0	4	7	15	3	6

Table 2 CKJM Metrics Of With Design Principle

Class Name	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM
AssistantProfessors	2	1	0	1	3	1	0	2
ICalculateTax	1	1	0	0	1	0	3	1
SalaryCalculator	4	1	0	3	7	4	1	4
TaxCalculator	2	1	0	1	3	1	1	2
HeadOfDepartment	2	1	0	1	3	1	0	2
Main	2	1	0	5	14	1	0	2
IEmployee	1	1	0	0	1	0	6	1
Professors	2	1	0	1	3	1	1	2
ICalculateSalary	2	1	0	1	2	1	1	2

IJSER

9 Result And Discussion

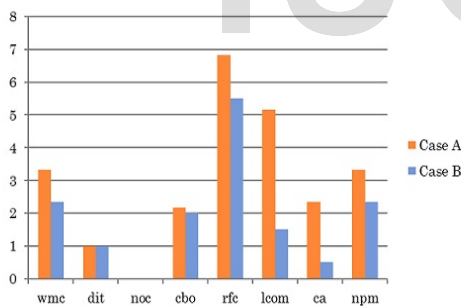
These principles are introduced to make an immortal software [8] design and validation metrics tools [9]. Different tools are used in separate applications to measure these metrics. CKJM metrics are used to assess the results for both the type of implementations. An analysis of the calculated metrics has been used to construct the software design prediction models, where we could have different combinations of design principles. Each time, a model constructed according to the data from project version i has been assessed by predicting the design in project version i+1. The analysis shows that the implementation of design principles in an application can reduce the dependency factor and can help in developing a scalable architecture. For this sample application, the quality has been improved by reducing the coupling and introducing the cohesion measure. The collected metrics and SOLID combination

Table 3 Average Mean of CKJM Metrics Without Design Principles

Class Name	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM
AssistantProfessors	2	1	0	1	5	0	3	2
Professors	2	1	0	1	5	0	4	2
Main	2	1	0	2	12	1	0	2
HeadOfDepartment	2	1	0	1	5	0	3	2
SalaryCalculator	6	1	0	4	7	15	1	6
TaxCalculator	6	1	0	4	7	15	3	6
Average (Mean)	3.33	1	0	2.16	6.83	5.16	2.33	3.33

Table 4 Average Mean of CKJM Metrics With Design Principles

Class Name	WMC	DIT	NOC	CBO	RFC	LCOM	CA	NPM
AssistantProfessors	2	1	0	1	3	1	0	2
TaxCalculator	2	1	0	1	3	1	1	2
HeadOfDepartment	2	1	0	1	3	1	0	2
Main	2	1	0	5	14	1	0	2
Professors	2	1	0	1	3	1	1	2
SalaryCalculator	4	1	0	3	7	4	1	4
Average(Mean)	2.3	1	0	2	5.5	1.5	0.5	2.3



can be used in further research areas, where we would like to identify the factors and investigate whether they have statistically significant influence on all types of applications. The results of the important reproducible empirical research studies have been specified as follows.

10 Conclusion

As the demand for software has diversified during the last few years leading to a rapid development of the software application [10], the focus has now shifted on the scalability of the Software design. It is important that while working with the latest development methodologies, the software should scale itself as per the market competency. Design should be based on Principles so that it would be easy to reuse and scale the services [1].

In this research work, a comparative study on the SOLID Design Principles has been done demonstrating their use in avoiding an immoral design. It can be stated that the application of these SOLID design Principles together could lead us to create a highly maintainable and scalable system. The research demonstrates the empirical assessment of a Software application against the Design approach, and evaluates the quality of software using CKJM matrices. For our sample application we have reduced the coupling by 69% (appox.) and introduce the cohesion by 29% (approx.). Thus the case study approves and encourages the use of Design principles.

11 Future Work

This type of study motivates the users to enhance the implementation of the Design Principles in Software design by finding out the useful and vital combination of these Principles. This approach should be used to define and use different combinations of these principles in various types of applications including Desktop, Web and mobile applications.

This assessment study can be elaborated in a tool so that a software developer can analyze the impact of various combinations of design principles and choose the required and feasible approach for its application.

Apart from finding the different combinations, the reusability and quality of the software can be increased, by predicting the Software design model.

References

[1] R. Subramanyam and M. Krishnan, "Empirical analysis of ck metrics for object-oriented design complexity: Implications for software defects," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, April 2003.

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

[3] [Online]. Available: http://www.oodesign.com/design-principles.html

[4] S. R. Chidamber and C. F. Kemerer, "A metrics suite for object oriented design," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 20, pp. 476–493, June 1994.

[5] [Online].Available: http://www.spinellis.gr/sw/ckjm/doc/indexw.html

[6] R. L. Henrike Barkmann and W. L. owe, "Quantitative evaluation of software quality metrics in open-source projects."

[7] J. Bansiya and C. G. Davis, "A hierarchical model for object-oriented design quality assessment," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 28, January 2002.

[8] E. D. G. Neha Goyal, "Reusability calculation of object oriented software model by analyzing ck metric," *International Journal of Advanced Research in Computer Engineering and Technology*, vol. 3, pp. 2466–2470, July 2014.

[9] T. H. A. S. Saddam H. Ahmed and A. A. Sewisy, "A hybrid metrics suite for evaluating object-oriented design," *International Journal of Software Engineering*, vol. 6, pp. 65–82, January 2013.

[10] R. Vir and P. S. Mann, "A hybrid approach for the prediction of fault proneness in object oriented design using fuzzy logic," *Journal Academic Industrial Research*, 2013.