A Modifiability-Improved Design by Applying SOLID Principles

Shaoqing Yu (syu702)
Student in
Department of Electrical and
Computer Engineering
The University of Auckland
Auckland, New Zealand
Email: syu702@aucklanduni.ac.nz

***Abstract*—The SOLID principles have been highly spoken of and widely used in the software development areas since its birth, as it is viewed to be able to increase the design modifiability. This report will provide a comparison between my previous design of kalah game and the new one which has 2 of the SOLID principles applied in. A discussion about the improvements brought by using SOLID will also be included at the end.**

I. Introduction

The modifiability of code, which means the capability of the software product to be modified, is one of the important attributes when software designers evaluate the quality of their designs. There are more than one way that are widely believed to be able to increase the design modifiability, and the SOLID principles, is one of the most popular. This report will firstly give a brief introduction of modifiability and the SOLID principles to clarify the depth and scope of the topic. Then after applying 2 of SOLID in the previous design, the evaluations about the improvements of modifiability will be described in detail.

II. Modifiablity Defination

The modifiability, as its name implies, means to how much degree that the design can be modified without introducing defects or degrading existing product quality. Usually, more modifiability could be achieved through localizing changes, preventing ripple effects and deferring binding time. It is a abstract concept that cannot be measured purely by any statistic metrics, as its actual meaning may vary in different change cases. It indicates that it would be more clear to talk about modifiability in some specific situations where the requirements have been changed and some results are under expecting.

III. SOLID Principles Clarification

TABLE I: The SOLID principles definition.

	Principle	Description
S	Single Responsibility Principle (SRP)	A class should have one, and only one, reason to change.
O	Open-Closed Principle (OCP)	Software entities should be open for extension, but closed for modification.
L	Liskov Substitution Principle (LSP)	Child classes must be substitutable for their parent classes.
I	Interface Segregation Principle (ISP)	Make ne grained interfaces that are client specific.
D	Dependency Inversion Principle (DIP)	Depend on abstractions, not on concretions.

The term ”SOLID” comes from the first letters of the five principles, namely, the Single Responsibility Principle (SRP), the Open-Closed Principle (OCP), the Liskov Substitution Principle (LSP), the Interface Segregation Principle (ISP), and the Dependency Inversion Principle (DIP) [1]. More details are listed in Table I.

Due to the limited time and restricted report length, this report will focus on the OCP and DIP in application in the coming sections.

IV. Application of OCP

The OCP is the fundamental one of all 5 principles. It means software entities should be open for extension, but closed for modification.[1] The 2 commonly used techniques to meet OCP are polymorphism and generic classes.[1]

Based on the analysis above, applying this principle actually means to abstract the stable part of your design into interfaces or generic classes, and meanwhile distribute the variations of the implementation details to different sub classes.

A. Changes on Original Design

Whenever the implementation may vary, the OCP should be considered to be applied. To be specific, in this game, the rules, the players, the actions and the outputs could be changed in different situations. One solution is to modify the original design as Appendix A.

With the inheritances coloured red in Appendix A, the sub types will share the methods or the stable part of its class with the parent type. A new specific variation of implementation could be achieved by creating a new class deriving from the same parent class.

Another approach is to utilize some generic classes as the blue blocks showed in Appendix A, which have the commonly used and stable processes being abstracted in.

B. Modifiability Improvements

The importance of the OCP is to inspire the developers' awareness of distinguishing the stable and changeable parts in the design. Giving enough freedom to change the changeable parts, but avoiding the changes on the stable, that is what "open" and "close" mean in OCP. With telling the differences, when new changes are required, the developers only need to modify the changeable parts and reuse the stable. Compared with the unnecessary changes on stable parts in some situation where the OCP is ignored or failed to be applied, this is a kind of improvements in modifiability. There are 2 ways to apply OCP in designs, namely, polymorphism and generic-class.[1]

For the polymorphism approach, such as the red inheritances in package kalah.rule, the changes of logics in rules are open, but the common protocols among them are defined in parent type IRule, which is close to modifications. It can defer the time of type determining to runtime. With this, adding a new rule is just to create a new class sharing the same protocols defined in IRule. Otherwise, besides implementing how the rule affects the game, the usages of this rule is also need to be defined again apart from the methods in IRule.

For the generic-class approach, such as the blue generic classes Selector, the changes of operating objects are open, but the process of handling and class structure are close to be modified. It can defer the time of type determining to compiling. With this, a new selector of a collection of certain types can be used directly by injecting the target type into the generic template. Otherwise, the code manageability issues could be raised from the duplications of the similar class structures and data dealing processes.

V. Application of DIP

To properly apply the DIP, it is important to define "dependency" first. Here the "dependency" means that in one 2-element-pair, changes to the definition of one element may cause changes to the other.[1] In practice, one class depends on the other if the other one is passed as a method parameter or used as a local variable.

After the definition of DIP is clarified, the applying of this principle is just as simple as making sure that every dependency of the various implementations in the design should target an interface, or an abstract class, rather than a concrete class.

A. Changes on Original Design

Since the decoupling is concerned initially, only after the several modifications listed below have been done, the design has been changed to what showed in Appendix B, which satisfies DIP.

Game - ActionPool Game depends on ActionPool instead of KalahActionPool.

Game - PlayBoard Game depends on PlayBoard instead of KalahPlayBoard.

IArgVisitor - Game IArgVisitor depends on Game instead of Kalah.

IArgVisitor - PlayBoard IArgVisitor depends on PlayBoard instead of KalahPlayBoard.

IDumpable - IArgVisitor IDumpable depends on IArgVisitor instead of DumpVisitor.

All dependencies in design are coloured out in Appendix B. The dependencies with red colour rely on abstract classes or interfaces, which imply DIP. The 2 purple dependencies involve concrete classes, but they are decoupled by the singleton design pattern, which provides global accessibility. The Container composites SeedNum, which is also a concrete class. However, The SeedNum is a wrapper of Integer, which is solidly dependable. Overall, the DIP is fully applied.

B. Modifiability Improvements

Generally, the DIP can improve modifiability, as the dependency relies on the stable abstractions instead of implementation details, which may vary. Let me take the "Game - PlayBoard" dependency for example, if the layout of the play board changes for some reasons, in the previous design, we have to not only change the code in PlayBoard(concrete), but also change other classes which are associated, such as Game. However, after applying DIP, we just need to simply create a new play board class deriving from PlayBoard(abstract), and inject its instance to Kalah through the constructor. The code in other associated classes don't need to be changed because the methods defined in PlayBoard(abstract) are stable.

VI. Design Result

The new design has been tested under JRE 1.8 and JUnit 4.12. As what demonstrated in Figure 1, it is able to pass all of the 19 test cases. This means at least the existing design functionalities has not been degraded by applying OCP and DIP.

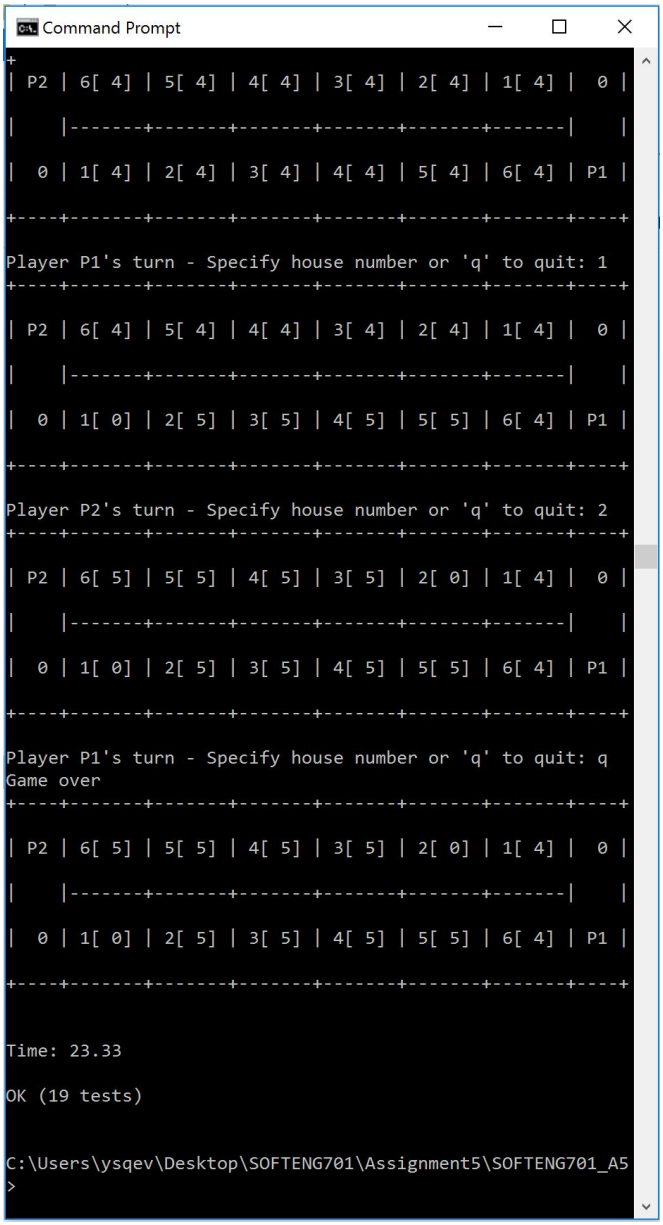

Fig. 1: The execution result of my design.

VII. Conclusion

The new modified design has been proved to have more modifiability by using OCP and DIP since some specific changes cases are easier to meet than before. Based on these, it is reasonable to believe that the SOLID is effective at improving design modifiability, even only 2 of them are verified in this report. Chasing modifiability is like keeping the balance between stability and changeability in design. The balance point is hard to locate, not only because the variations of change cases, but also due to the difficulties in conceptual understandings. Based on my experience, the OCP actually means to abstract the stable parts into shared or commonly used interfaces, and then implement detailed logics accordingly to changes. Since the abstractions are more stable than implementations, which could be changed in different cases, they are more dependable than concrete classes due to the avoidance of unnecessary ripple effects. That is the mainly contribution brought by DIP to the design. The software design is an experience-based process, some of the design principles like SOLID might have already been applied in the daily development unintentionally. This assignment provides me an opportunity to critically think about its meaning and function, then examine them by myself. This experience is really first-hand and valuable, which would guide my software design in the future.

APPENDIX A
UML DIAGRAM OF APPLYING OCP

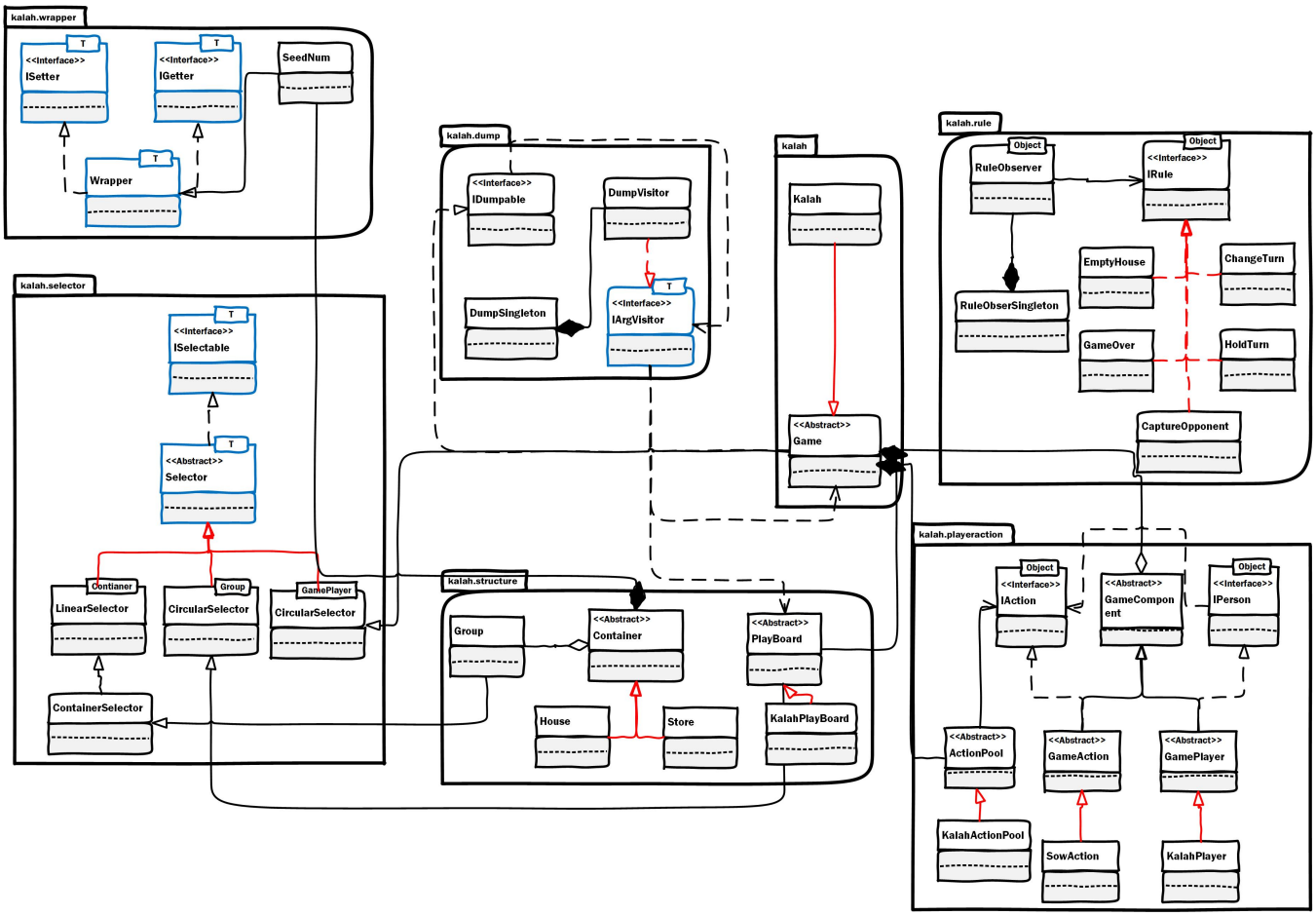

Fig. 2: The Changes of Applying OCP.

APPENDIX B
UML DIAGRAM OF APPLYING DIP

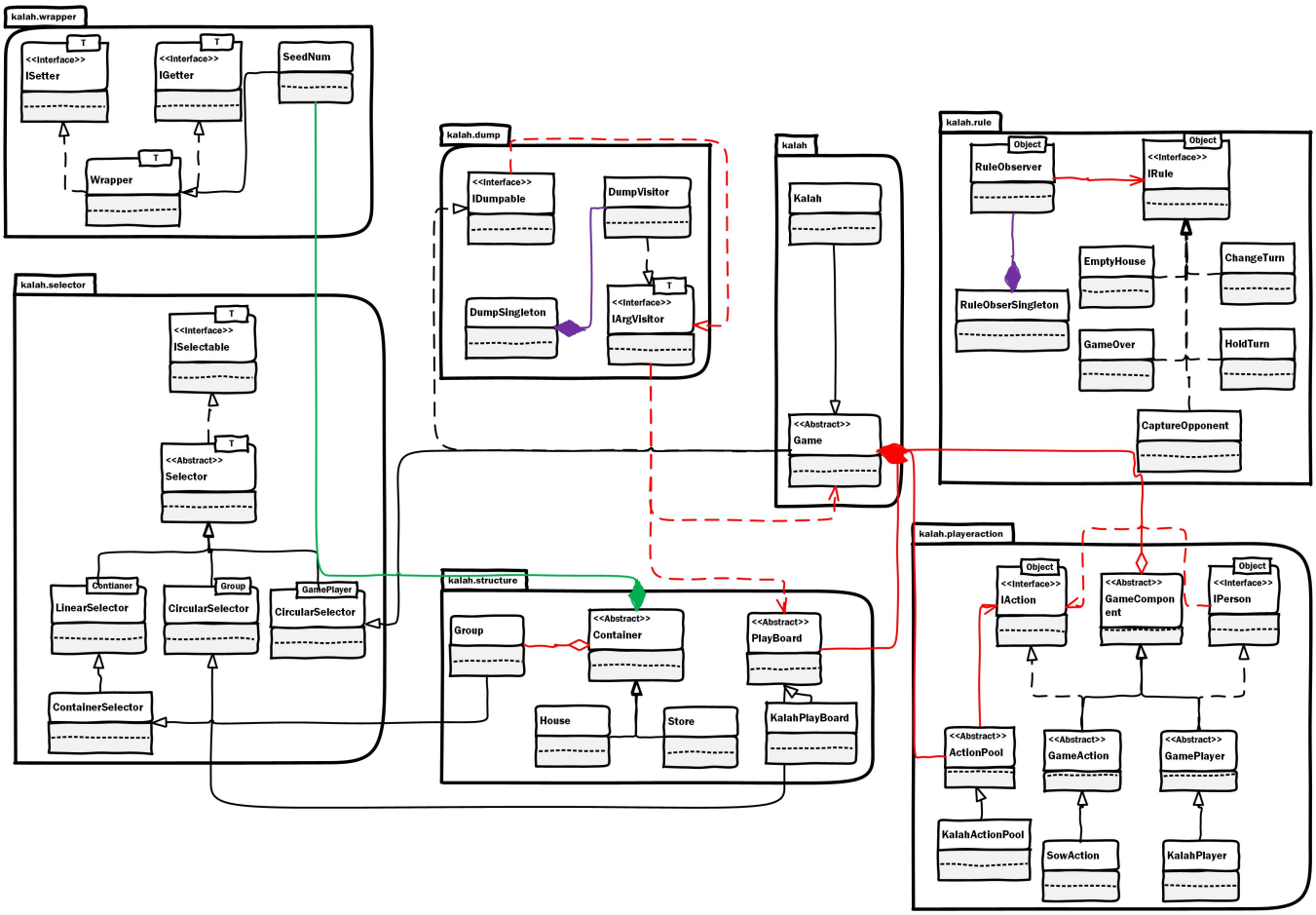

Fig. 3: The Changes of Applying DIP.

ACKNOWLEDGMENT

I would like to express my special thanks of gratitude to my lecturer Ewan, who bears my poor grammar and limited expression of understandings. The marker also deserves my thanks, considering without his hard working it is impossible for us to get a fair mark on our reports. Happy winter holidays!

ACRONYMS

DIP Dependency Inversion Principle. 1–3

ISP Interface Segregation Principle. 1

LSP Liskov Substitution Principle. 1

OCP Open-Closed Principle. 1–3

SRP Single Responsibility Principle. 1

REFERENCES

[1] R. C. Martin, “Design principles and design patterns,” 2000.