The Modifiability Evaluation of 4 Designs of the Kalah Game

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 modifiability has long been viewed as one of the important matrices of a software design with a good quality. However, there exists no obvious metrics to evaluate design modifiability. Even the understandings of modifiability are not widely agreed. This report will compare 4 different designs, and then discuss the meaning of modifiability as well as the assessments.**

***Index Terms*—Modifiability, metrics, quantity, quality, empirical relationship.**

I. Introduction

The modifiability of design, which means that to how much degree the design can be changed when requirements vary, is one of the important attributes when software designers evaluate the quality of their designs. To clarify the definition and effective metrics of design modifiability, this report will compare 4 designs from both quantitative and qualitative perspectives, and then by analysing the empirical relationship between them, conclude the metrics to effectively measure design modifiability.

II. Modifiability Classification

Literally, the modifiability of design means the capability of the software to be modified. The modification discussed here is the changes happens on design rather than software behaviours. Under a certain change case, a valid modification means to the largest possibility to avoid changes on existing code, but satisfies the changing requirements [1].

It is a concept widely discussed in Object-Oriented (OO) software design methodology, since, more often than not, the modifiability is reached through OO design principles, such as abstraction, encapsulation, inheritance, polymorphism [2]. It means, to some extends, the modifiability of a design can be reflected by the quantity of applying OO design paradigm. However, we cannot simply claim that the more OO design principles are applied, the more modifiability a design will has, since the OO design principles could be misused so that have no actual effects at all [3]. To determine the effectiveness of OO design paradigm, the quality of design should also be concerned and tested against some change cases.

In the coming sections, this report will compare the OO design metrics of 4 designs quantitatively, and then evaluate the design quality against several change cases.

III. Quantitative Comparison

The modifiability can be reflected by some metrics of OO design paradigm statistically. This results from the fact that the goal of OO design paradigm is to better support changes than what came before. Without properly applying sufficient OO design principles, the design will have very limited changeability.

To measure the usability of the OO design principles within a design, the Number of Types (NoT), which stands for the scale of the abstraction of this design, is the first metric need to be concerned. It is hard to believe a simple design will have too much modifiability, as there is not too much can be modified really. The Number of Packages (NoP) reflects the maintainability of semantic coherence. The Number of Interfaces (NoI), which is the abstraction of common services, can limit the communication protocols between objects. To keep the expected changes on enums at the same place, the Number of Enums (NoE) should also be recorded here. The Number of Exception Classes (NoEC) will be excluded and listed separately from the Number of Classes (NoC), which is the direct result of applying OO design paradigm, as the exception capturing have no impact on functionality modifications.

Besides these, specifically to each design, the Coupling between Object Classes (CBO) [4], includes all the dependencies and associations between class, should be not high if a better modifiability is expected. The Depth of Inheritance Tree (DIT) and Number of Children (NOC) [4] are the straight reflections of the using of the inheritance principle. The Weighted Methods Per Class (WMC) [4], which implies the count of methods per class, repeals the changeability since a class with many methods is likely to be more application specific, limiting the possibility of reusing and modifying. The static methods are effective at decoupling since they provide the global accessibility. However, too much using of it could destroy the encapsulation principle, hence it is also worth listing Number of Static Methods (NOSM) here.

In the next section, the report will firstly provide an overview of statistics of 4 designs, and then analyse them separately.

A. Overall

This section will compare the 4 design quantitatively by statistics of OO design metrics, specifically, from the numbers of types, packages, interfaces, enums, classes, and exception classes respectively.

TABLE I: The overall comparison of 4 designs.

	Design A	Design B	Design S1	Design syu702
NoT	5	28	11	36
NoP	1	6	4	7
NoI	0	6	1	8
NoE	1	2	2	0
NoC	4	20	8	28
NoEC	0	0	0	0

According to the overall statistics of each design as Table I shows, we can see that either of design B and syu702 has more than 30 types totally to abstract and encapsulate different functionalities of the Kalah game. By contrast, the total NoT in either design B or S1 is around 10, which means the using of OO design principles within design is very limited. The smaller NoI in design A and S1 means to lower level of common service abstraction. Meanwhile, compared with design B and syu702, the NoC in design A and S1 is insufficient. If we assume the original requirements are the same and all will be fully met by each design, the densities of functionality implementation of each class in design A and S1 are much higher than design B and syu702. It is acknowledged that the class with high functional implementation density is hard to maintain and reuse.

B. Design A

TABLE II: Metrics of design A.

PACKAGE	TYPE	CBO	DIT	NOC	WMC	NOSM
kalah	**Kalah**	1	1	0	5	1
kalah	**Player**	0	1	0	2	0
kalah	**Board**	0	1	0	31	1
kalah	**GamePlay**	2	1	0	10	0
kalah	**MoveResult**	0	0	0	0	0

As the Table I indicates, the design A has totally 5 types including 4 classes, 1 enum and no interface. The detailed metrics of each type are listed in Table II. The highest CBO in GamePlay is 2, which looks noraml. However, considering there are totally 5 types, the modifications on this class should be done carefully. There exists no inheritance within this design as each class has neither parent (the maximum DIT is 1) nor child (the maximum NOC is 0). The high WMC of class Board implies that this class is too heavy, having some functions which could be distributed.

C. Design B

TABLE III: Metrics of design B.

PACKAGE	TYPE	CBO	DIT	NOC	WMC	NOSM
kalah	**Kalah**	3	1	0	5	1
kalah	**Game**	3	1	0	10	0
kalah	**GameState**	4	1	0	9	0
kalah	**Gam..ory**	3	1	0	11	5
.player	**Human**	1	2	1	3	0
.player	**Player**	1	1	0	6	0
.board	**Board**	1	1	0	8	0
.board	**StdToplogy**	0	1	0	22	0
.board	**Pit**	0	1	0	20	0
.board	**PitToplogy**	0	0	0	0	0
.board	**Side**	0	0	0	0	0
.display	**Std..try**	1	1	0	8	0
.display	**Display**	2	1	0	3	0
.display	**Boa..try**	0	0	0	0	0
.move	**Move**	1	1	0	5	0
.move	**Mov..ons**	1	1	0	4	0
.move	**Kal..Mov..sor**	4	1	0	3	0
.move	**Mov..tor**	0	0	0	0	0
.move	**MoveStatus**	0	0	0	0	0
.rules	**Kal..Cap..rty**	0	1	0	4	0
.rules	**Kal..Cap..sor**	0	1	0	1	0
.rules	**NoMov..rty**	0	1	0	2	0
.rules	**NoEmp..rty**	0	1	0	3	1
.rules	**Kal..Sco..sor**	0	1	0	4	0
.rules	**Kal..Rul..Set**	1	1	0	7	0
.rules	**Boa..Sta..sor**	3	0	0	0	0
.rules	**RuleSet**	2	0	0	0	0
.rules	**Boa..Sta..rty**	3	0	0	0	0

[1]The package name which starts with "." means it is a sub-package of "kalah", for example, .player=kalah.player.

[2]".." is used to reduce the length of the type name.

Although there are 20 classes, 6 interfaces and 2 enums in design B according to Table I, it has only one effective inheritance between class Human and Player according to DIT and NOC. The CBO ranges from 0 to 4, which is not too high compared with the total NoT. It is worth noting that the NOSM in GameFactory effectively decouple the dependences with other classes. Generally, the methods' weight are distributed, but the WMC of Pit and StdToplogy are still a little bit high compared with others.

D. Design S1

TABLE IV: Metrics of design S1.

PACKAGE	TYPE	CBO	DIT	NOC	WMC	NOSM
container	**House**	1	2	0	1	0
container	**Store**	0	2	0	1	0
container	**Container**	0	1	2	7	0
kalah	**Board**	2	1	0	54	0
kalah	**Printer**	2	1	0	35	0
kalah	**Kalah**	3	1	0	13	1
user	**Player**	2	1	0	9	0
user	**User**	0	0	0	0	0
util	**Gam..les**	0	1	0	0	0
util	**Command**	0	0	0	0	0
util	**Order**	0	0	0	0	0

For design S1, there are 8 classes, 1 interface and and 2 enums recorded in Table I. Despite House and Store are subclasses of Container, the inheritance is not effective here since on Table IV, the WMC of them are low. This results in 2 ”god classes”, Board and Printer, which do complicated tasks that should have been distributed to other classes properly. What’s worse, the classes with higher WMC also have more CBO than others.

E. Design syu702

TABLE V: Metrics of design syu702.

PACKAGE	TYPE	CBO	DIT	NOC	WMC	NOSM
kalah	**Kalah**	0	4	0	9	1
kalah	**Game**	2	3	1	11	0
.structure	**Group**	0	4	0	4	0
.structure	**PlayBoard**	1	3	0	5	0
.structure	**House**	0	2	0	3	0
.structure	**Store**	0	2	0	2	0
.structure	**Container**	2	1	2	4	0
.pla..ion	**KalahPlayer**	1	3	0	6	0
.pla..ion	**SowAction**	1	3	0	10	0
.pla..ion	**GamePlayer**	0	2	1	5	0
.pla..ion	**GameAction**	0	2	1	3	0
.pla..ion	**Gam..ent**	1	1	2	3	0
.pla..ion	**ActionPool**	1	1	0	5	0
.pla..ion	**IPerson**	1	0	0	0	0
.pla..ion	**IAction**	0	0	0	0	0
.selector	**Con..tor**	0	3	1	9	0
.selector	**Cir..tor**	2	2	2	2	0
.selector	**Lin..tor**	1	2	1	3	0
.selector	**Selector**	0	1	2	3	0
.selector	**ISe..ble**	0	0	0	0	0
.wrapper	**SeedNum**	0	2	0	3	0
.wrapper	**Wrapper**	0	1	1	4	0
.wrapper	**IGetter**	0	0	0	0	0
.wrapper	**ISetter**	0	0	0	0	0
.dump	**DumpVisitor**	0	1	0	9	0
.dump	**Dum..ton**	1	1	0	2	1
.dump	**IArgVisitor**	2	0	0	0	0
.dump	**IDumpable**	1	0	0	0	0
.rule	**GameOver**	1	1	0	2	0
.rule	**Rul..ton**	1	1	0	2	1
.rule	**EmptyHouse**	1	1	0	2	0
.rule	**HoldTurn**	1	1	0	2	0
.rule	**RuleObserver**	1	1	0	4	0
.rule	**ChangeTurn**	1	1	0	2	0
.rule	**Cap..ent**	1	1	0	3	0
.rule	**IRule**	0	0	0	0	0

[1]The package name which starts with ”.” means it is a sub-package of ”kalah”, for example, .player=kalah.player.

[2]”..” is used to reduce the length of the type name.

The detailed metric statistics about design syu702, which has 28 classes and 8 interfaces in Table I, are listed on Table V. The lower CBO reflects the better code modularity and encapsulation. In this design, the inheritance is effectively used more than one time, which reduce the code duplication. The WMC of each class ranges from 2 to 11, which means there is no ”god class”. Even completely rewriting some classes will not affect a lot implementations due to the low WMC.

F. summary

From the statistical results of the OO design metrics, design B and design syu702 have more modifiability than other 2. To be specific, the design syu702 is the best among them.

IV. Qualitative Comparison

To be complementary, the quality of design, which involves the adaptations to change cases and some design patterns, should also be discussed, as sometimes the OO design principles could be inappropriately applied. To check the real effects of applying the OO design paradigm, the 4 designs with their UML diagram will be evaluated against the change cases listed in Table VI.

TABLE VI: Changes cases to be considered during evaluation.

Case	Description
DIR	Change the direction of sowing seeds
UND	Provide an Undo move
S/L	Provide Save/Load for partially completed games
EPY	Change the capture rule to, a capture can take place even when the opposite house is empty
TKC	Change the capture rule to, a capture takes place if the last seed falls into an opponent's house and that house has an even number of seeds, then all seeds in that house are captured
RBT	Allow the possibility that the second player is played by the computer, that is, is a robot
PRE	Change in presentation of playing board, for example, using "*" for each seed

A. Design A

The UML diagram of design A is demonstrated at appendix A.

Considering there are very little abstraction of objects and solid encapsulation, but neither inheritance nor polymorphism to enable modification in design A, the modifiability is very limited. It means no matter what kind of modifications the developer want to bring to any class in design A, he has to not only edit the target class, but also modify other classes associated. None of the change cases will be easily met in this case.

B. Design B

The UML diagram of design B is demonstrated at appendix B.

The interface PitTopology makes changing the direction of sowing not difficult. The package kalah.move brings more scalability to adding a new move such as "UND" to the game. There is a design pattern named "simple factory" applied in GameFactory. With the class Game, GameState and GameFactory, the "S/L" change case can be easily implemented by saving or loading a instance of GameState in GameFactory. To change the capture rule to "EPY" or "TKC", the interface BoardStateProcessor and BoardStateProperty should be realizated like KalahStdCaptureProcessor and KalahStdCaptureProperty, but with different logics. The abstract class Player can be extended to create a new robot player, just similar with the class Player. The only problem of this design regarding to these change cases is to change the IO output. If any new output, such as "*" to represent seeds on playing board, is required, the developers have to seek through all the classes outputting something to the IO, and change the code by hands.

C. Design S1

The UML diagram of design S1 is demonstrated at appendix C.

The design S1 has a typical tree structure. A kalah game has a Board and a Printer, a Board has several Players, each Player holds some Houses and one Store. Although there exists a inheritance, the only one, between Container and House and Store, which means it is easy to add a new kind of place to store and sow the seeds, this inheritance is not effective since it brings no changeability to the tree structure we mentioned before. The interface User makes adding a new user, such as a robot, possible, but the actual player actions are required to be implemented in class Board. The enums Command and GameVariables bring more ease on adding new moves such as "UND", but this changeability is behaviour-based rather than design-based, which means the associated code is still expected in class Kalah. Therefore, actually this design is just a little bit better than design A on modifiability, even some OO design principles are used. Despite more classes and enums are used in this design compared with design A, which means more design abstraction and object categorization, the inappropriate associations between classes, such as "a player has some houses and a store" and "a board has players", reduce the quality of design. The class Printer concentrates on IO displaying, which narrowed the range of code to change under "PRE" change cases. However, the classes with IO displaying functions has to call the instance of Printer somehow. This leads to high potential CBO on class printer if many classes have IO displaying requirements.

D. Design syu702

The UML diagram of design syu702 is demonstrated at appendix D.

The design of package kalah.selector is similar as a design pattern named "iterator", but not the typical one. As the name implies, the typical iterator provides a structure by which the developers can easily iterate elements in a collection. Besides the iteration function which is the same as a iterator, a selector focuses on returning the expected elements. With this design, the change case "DIR" can be easily implemented by creating a new selector named "ReversingSelector" and then letting other classes who expect this functionality inherit from it.

The design of package kalah.rule makes the kalah game more scalable. Here a design pattern named ”observer” is applied to decouple the rules and their effects on the kalah game. With this design, the implementations of change cases ”EPY” and ”TKC”, or even other more new rules, be met as simple as to create a new class to realizate the IRule interface and implement how to affect the game in the method Affect. The instance of this class should be properly observed by RuleObserver at where it occurs after the player’s certain action, for example, the SowAction.

The design of package kalah.playeraction aims to define game players and their different actions. The class ActionPool can load different actions and index them by literals. With the help of ActionPool, the change case UND can be achieved by creating a new class which inherits from GameAction. The instance of this class should be registered at th beginning of this game in ActionPool and acted by game players when it is required. The robot player change case ”RBT” can be implemented by adding a new class inherited from GamePlayer and overriding the method named Act.

The design patterns ”visitor” and ”singleton” are used in package kalah.dump. The class DumpSingleton makes DumpVisitor globally accessible. The class DumpVisitor wraps the IO and centralizes all the dumping functions. Any class who wants to display into IO can easily inherits from interface IDumpable and puts its dumping function in DumpVisitor. With the design in kalah.dump package, the change case PRE is easy to be matched by implementing interface IDumpable and add new dumping function in DumpVisitor.

The ”S/L” change case can be met by adding a serializer which enables to serialize and de-serialize the instance of Game to a local file. However, honestly speaking it is out of the original design.

E. Summary

Based on the the analysis of each design against the change cases, design B and syu702 can satisfy the same number of changes cases. As regarding to design A and S1, design S1 is a little bit better than A since the designer starts to utilize some OO design methods and put associated code together, however, the applying is sometimes not effective from the result.

V. Empirical Relationship

From the previous sections, we figure out that the design A, which has the fewest number of classes, has the fewest modifiability so that no change cases can be easily satisfied. The S1 have a larger quantity of classes and interfaces, however, they are not effective in decoupling and functionality distributing. The modifiability of design S1 is close to design A, but higher since the awareness of using interface and enums. Both of design B and syu702 have sufficient quantity of abstraction and inheritance applied and can satisfy the equal number of the change cases with limited modifications. However, considering the outstanding performance the design syu702 has played in the quantitative comparison, the design syu702 wins design B slightly on more modifiability. Based on the analysis previous, if the M(x) stands for the modifiability of design x, the relationship of modifiability between each design is as Equation 1 implies:

$$M(syu702) > M(B) > M(S1) > M(A) \tag{1}$$

VI. Conclusion

From the comparison of the 4 designs, we can learn that the modifiability is such a abstract concept that varies on different environments. Even though it can be measured by some OO design metrics quantitatively like what are demonstrated in the quantitative comparison section, the real effects are still required to be tested against specific change cases because there is no guarantee between the statistic of metrics and the actual outcomes. Overall, the general term ”modifiability” only becomes meaningful in specific change cases.

APPENDIX A
UML DIAGRAM OF DESIGN A

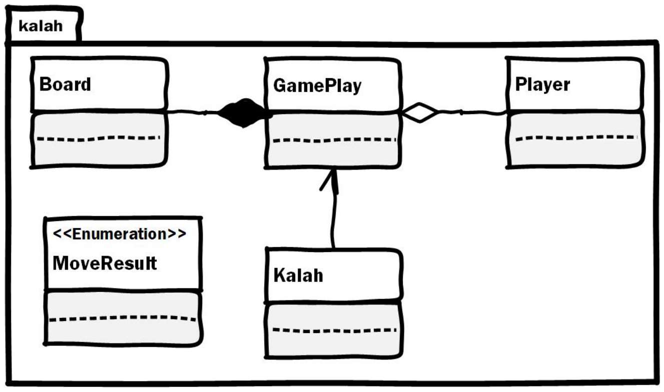

Fig. 1: The architecture of design A.

Appendix B
UML Diagram of Design B

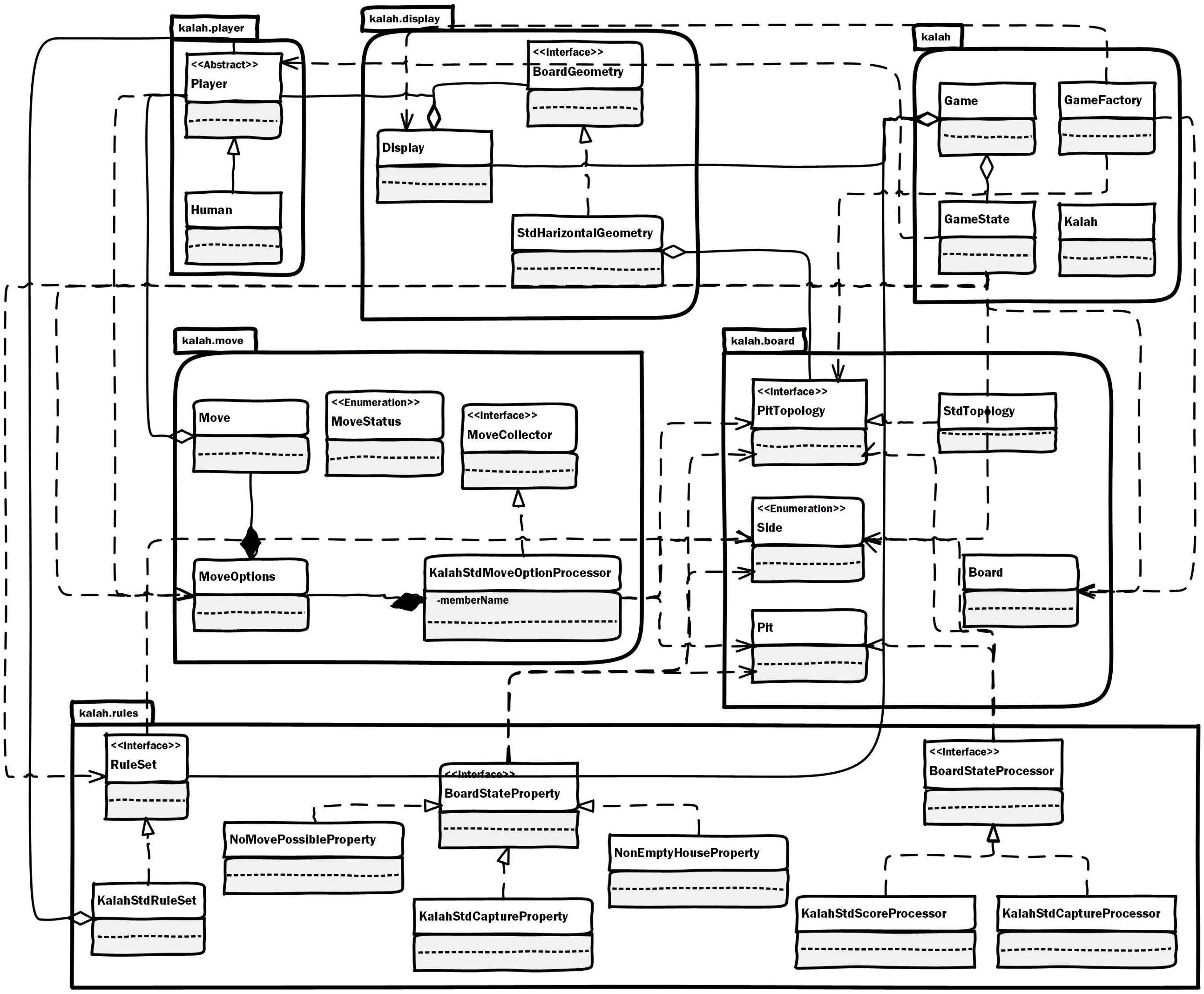

Fig. 2: The architecture of design B.

Appendix C
UML Diagram of Design S1

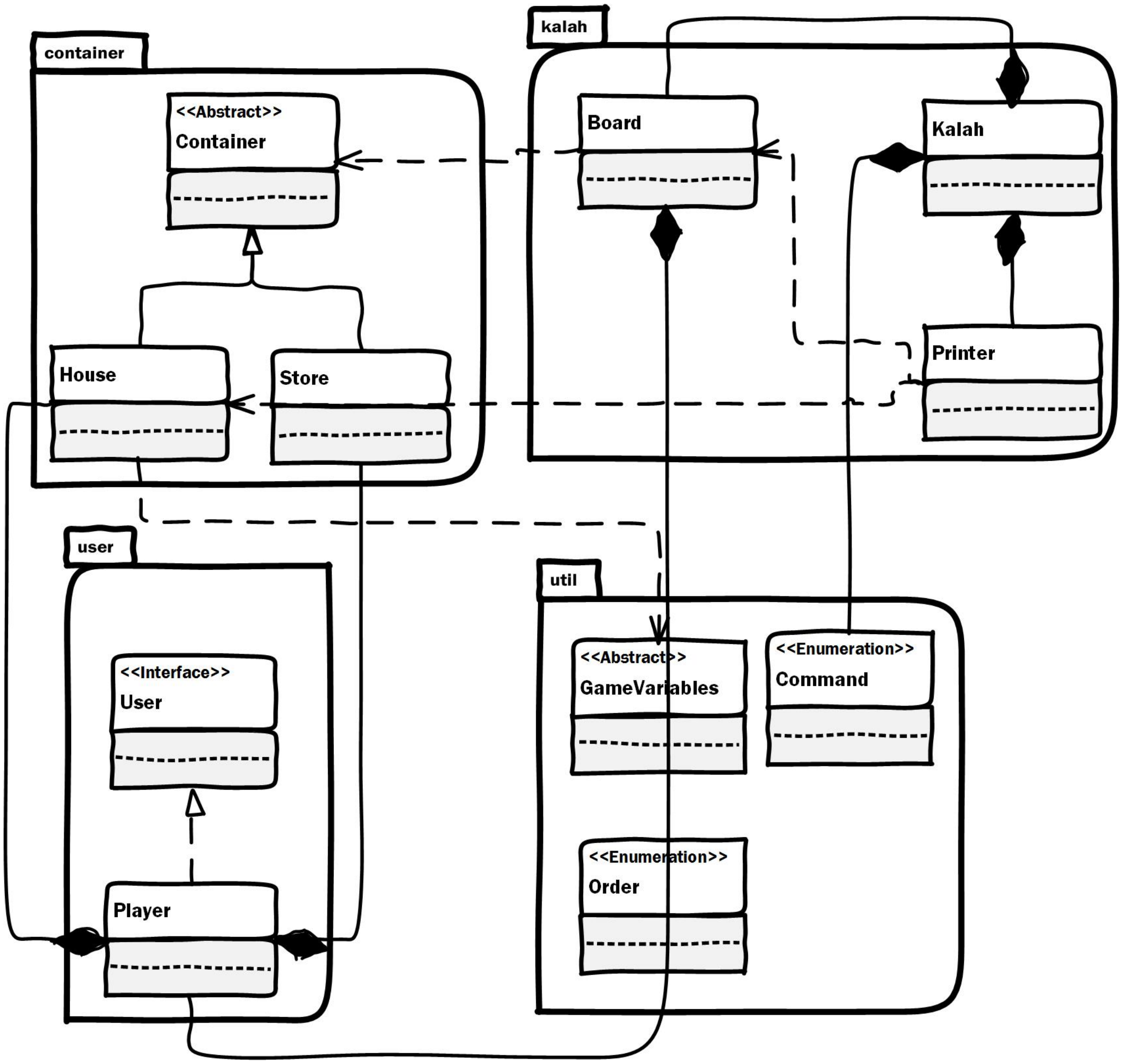

Fig. 3: The architecture of design S1.

Appendix D
UML Diagram of Design syu702

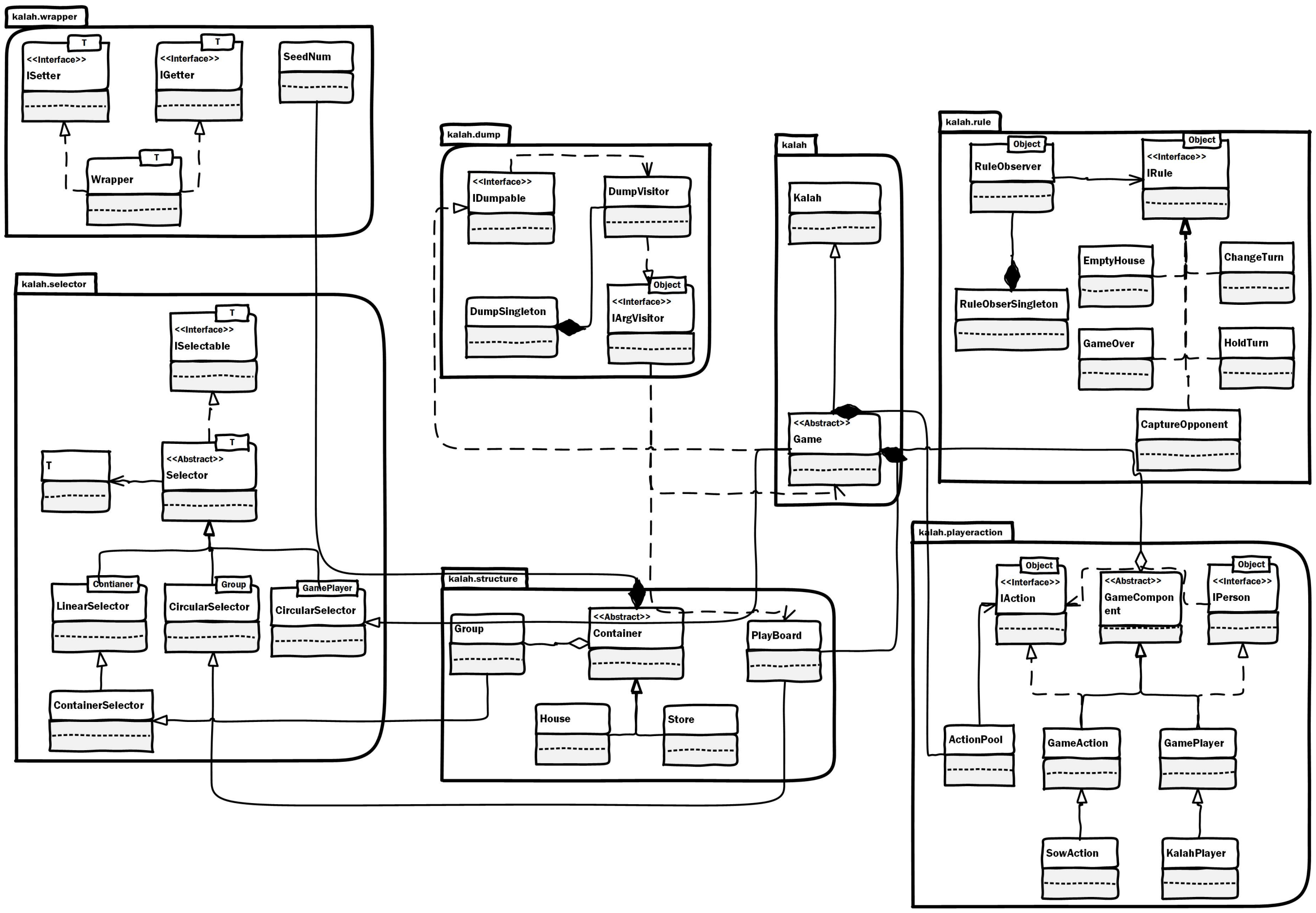

Fig. 4: The architecture of design syu702.

Acknowledgment

I would like to express my special thanks of gratitude to my lecturer Ewan, whose critical thinking skills and humours in lecturing really impress me, and also teach me a lot. If there exists any marker on this assignment, I think he/she definitely also deserves my thanks, because reading such a long report with tables and diagrams must be quite time-consuming.

References

[1] P. Oman and J. Hagemeister, “Metrics for assessing a software systems maintainability,” *Software Maintenance*, vol. volume, pp. 337–344, Nov 1992.

[2] M. Andersson and P. Vestergren, “Object-oriented design quality metrics,” 2004.

[3] P. Bengtsson, N. Lassing, J. Bosch, and H. van Vliet, “Analyzing software architectures for modifiability,” 2000.

[4] S. R. Chidamber and C. F. Kemerer, “A metrics suite for object oriented design,” *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 20, no. 6, pp. 263–265, 1994.