A modifiable design of 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 good a quality. This report will use a popular game named Kalah as a demonstration, firstly defining my understanding of modifiability and then, proposing a design of Kalah, which satisfies the modifiability most.**

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. As a sub-characteristic of maintainability, it can localise changes, prevent ripple effects and defer binding time. To practice and show my understanding of modifiability, in this report, I use a simple but popular game named Kalah, which has multiple different ways to play, as an example. My design will be explained in detial with UML diagrams in the next section.

Shaoqing Yu
5 May, 2016

II. Modifiablity Defination

III. Design Explanation

To reach a better modifiability, usually some design patterns and OOP approaches (abstraction, encapsulation, polymorphism and inheritance) should be properly applied. My detailed design of Kalah game is showed as below. There are totally 6 packages in my implementation, each of them contains a group of classes which are tightly interrelated.

A. kalah

As showed in Figure 1, there are only 2 class in this package: Kalah and Game. By inheriting from Game, which is the abstraction of games in the real world, the Kalah only needs to (also has to) implement the logics in Idle(), AddPlayers() and AddActions().

Modifiablity: With the inheritance from Game, developers can define not only Kalah but also other games by simply creating a class and implementing the abstract functions defined in Game (Idle, AddPlayers, AddActions).

TABLE I: The overall comparison of 4 designs.

	Design A	**Design B**	**Design S1**	**Design syu702**
Number of Package	1			
Number of Classes	4	20	8	28
Number of Exception Classes	0	0	0	0
Number of Interfaces	0	6	1	8
Number of Enums	1	2	2	0
Number of Types				

TABLE II: Metrics of design A.

PACKAGE	**TYPE**	**MOC**	**DIT**	**NOC**	**WMC**	**NOSM**
kalah	**Kalah**		1	0	5	1
kalah	**Player**		1	0	2	0
kalah	**Board**		1	0	31	1
kalah	**GamePlay**		1	0	10	0
kalah	**MoveResult**		0	0	0	0
	Total		4	0	48	2

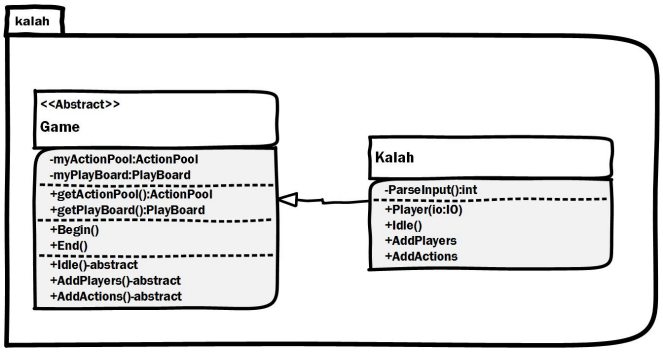

Fig. 1: The design of kalah package.

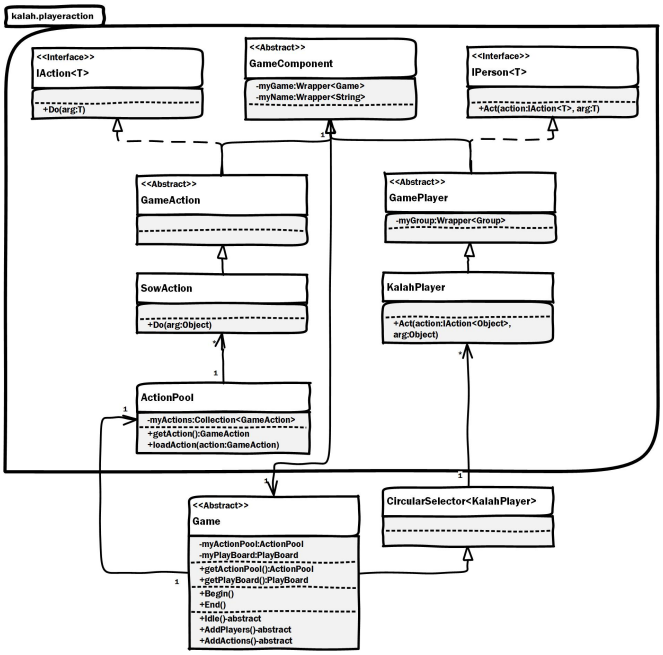

Fig. 2: The design of kalah.playeraction package.

B. kalah.playeraction

This package defines the most important 2 parts in a Game: the players and their actions.

As Figure 2 indicates, each action (for example SowAction) implementing IAction interface has its own Do functionality, in which the logics of this action should be; each player (for example KalahPlayer) implementing IPerson has its own Act functionality, which receives a instance of IAction and then handles it.

Modifiablity: With this design, developer can define different users executing different actions by inheriting GameUser and GameAction and implementing the detialed logics in their Do() and Act() respectively.

C. kalah.structure

This package defines the data structure of Kalah game. As we see from Figure 3, basically, there are 2 kinds of

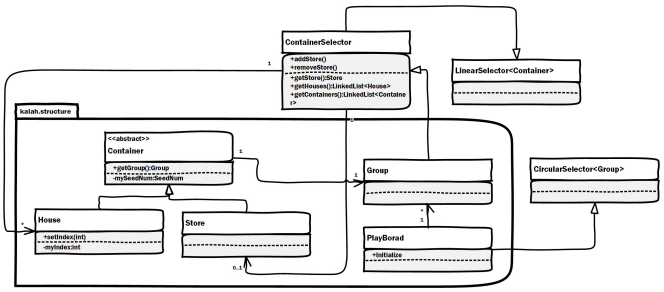

Fig. 3: The design of kalah.structure package.

Container, House and Store. They all have a Group which they belong to, meanwhile, each House instance holds its own index.

A Group is a ContainerSelector which can select elements in its collection according to their type and itorate them linearly.

A PlayBoard is a CircularSelector of Group which can iterate elements in its collection circularly.

Modifiablity: With this design, the virtual layout of play board can be easily changed by using different components (Store, House and Group). For example, we can have a new play board which support 5 or more players, and each of them has Store and House one by one in a Group.

D. kalah.rule

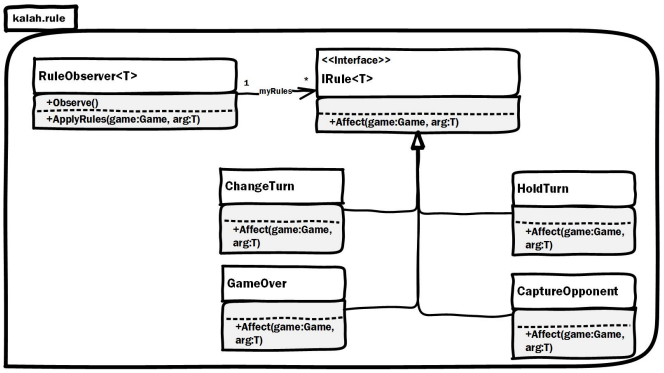

Fig. 4: The design of kalah.rule package.

After players' action, the rules of the game will have effects. They are all defined as Figure 4 shows. Here I use a design pattern called "observer".

If the RuleObserver observe a rule (for example CaptureOpponent) after player's action, the rule will affect the game automatically.

Modifiablity: With this design, the Game could have more rules by creating new rule classes which implement the IRule interface. They will bring more joy and complexity to this game.

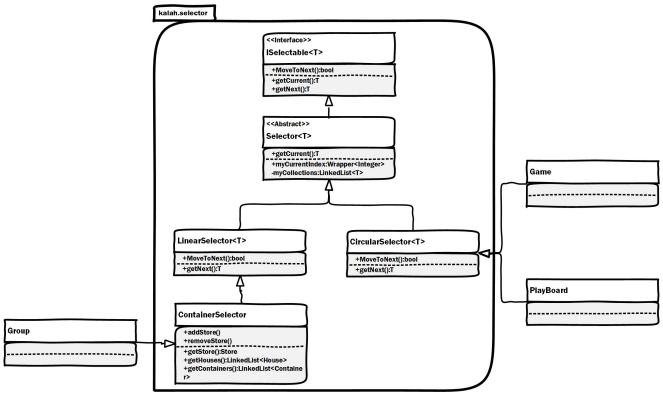

Fig. 5: The design of kalah.selector package.

E. kalah.selector

The purpose of this package is to provide multiple ways to loop through and get certain element in a collection, details are showed in Figure 5.

The ISelectable interface defines the basic API of a selector.

The Selector class is abstract, which has a collection of elements and is able to return the element which is pointed by current index.

The LinearSelector and CircularSelector implement 2 ways to iterate each elements in its collection respectively.

The ContainerSelector has to be a LinearSelector which holds a collection of Cotainer instances. This class can dynamically add and remove Store instance. It is also able to return subsets of its collection according to the type of elements.

Modifiablity: With this design, the new iteration way (for example skip a house in each step of sowing) can be achieved easily by adding a new selector to the game, if required.

F. kalah.dump

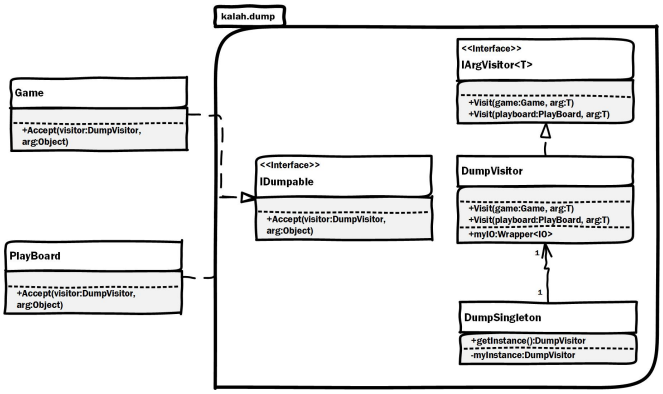

Fig. 6: The design of kalah.dump package.

As Figure 6 demonstrates, the IDumpable interface replaces the dump processing functionality by accept(). Each class who would like to access MockIO and dispaly anything should inherit it.

The DumpVisitor lets you define a new displaying operation without changing the classes of the elements on which it operates.

The DumpSingleton class holds a static instance of DumpVisitor, providing global access to DumpVisitor.Visit().

Modifiablity: With this design, the classes who want to print anything to MockIO merely need to implement the methods defined in IDumpable interface. The DumpVisitor lets you define a new dump operation without changing the classes of the elements on which it operates. The DumpVisitor's global accessibility is guaranteed by DumpSingleton.

IV. Design Result

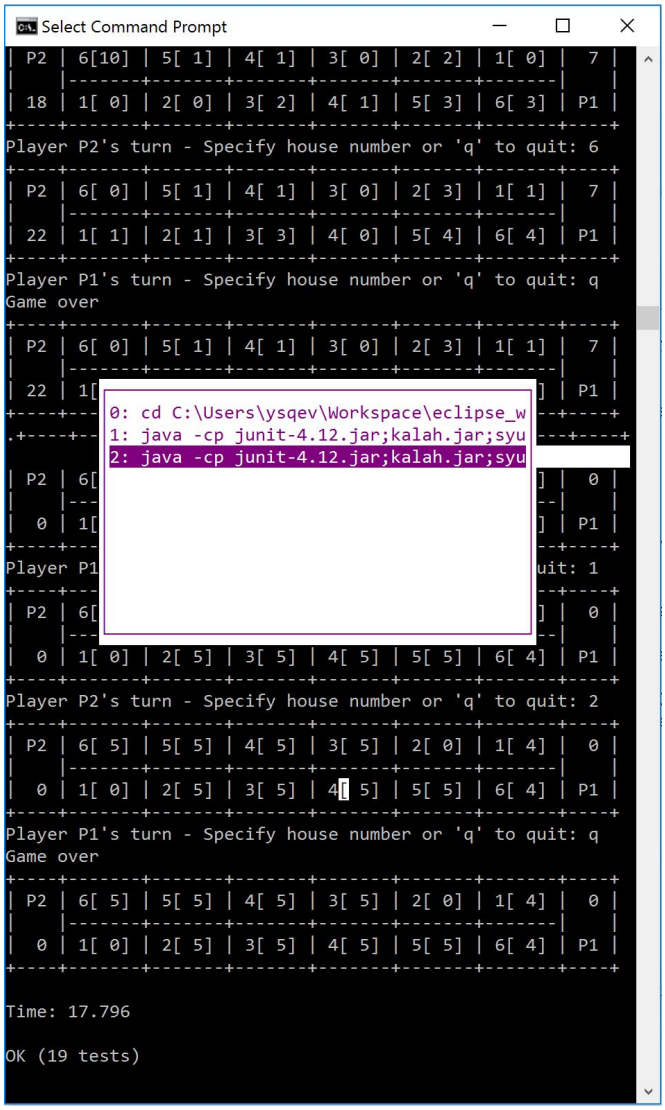

Fig. 7: The execution result of my design.

My implementation has been tested under JRE 1.8 and JUnit 4.12. As we can see from Figure 7, it is able to pass all of the 19 test cases.

V. Conclusion

APPENDIX A
UML DIAGRAM OF DESIGN A

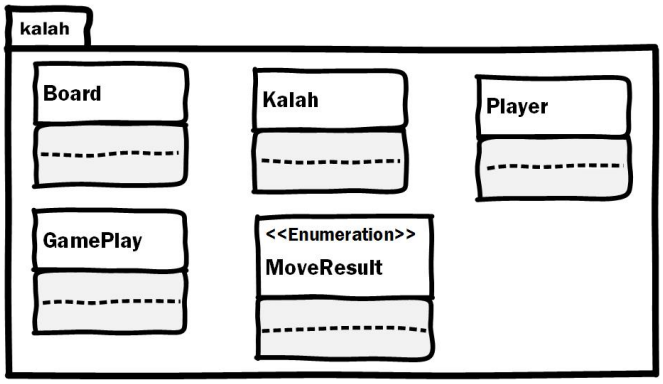

Fig. 8: The architecture of design A.

APPENDIX B
UML DIAGRAM OF DESIGN B

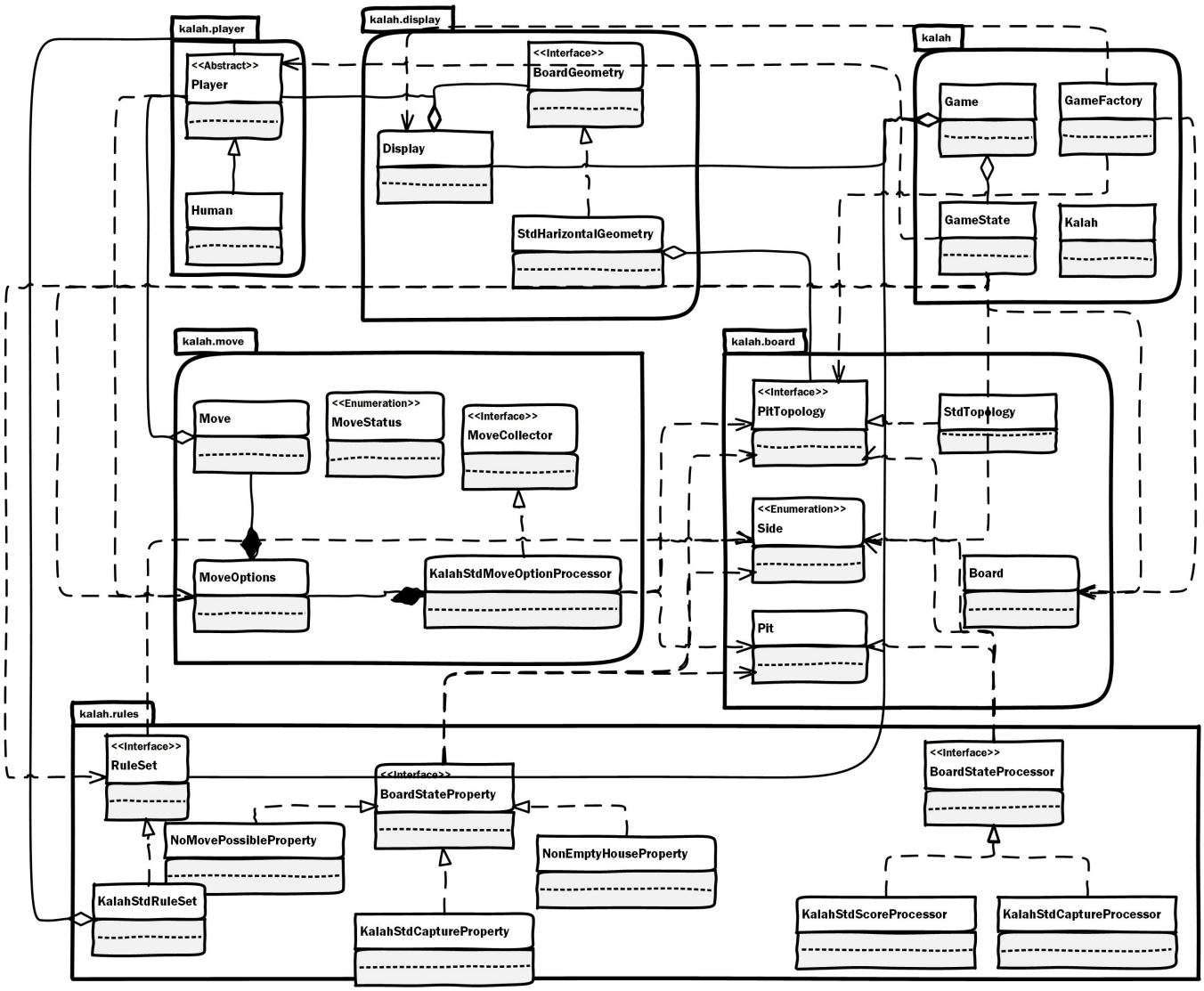

Fig. 9: The architecture of design B.

APPENDIX C
UML DIAGRAM OF DESIGN S1

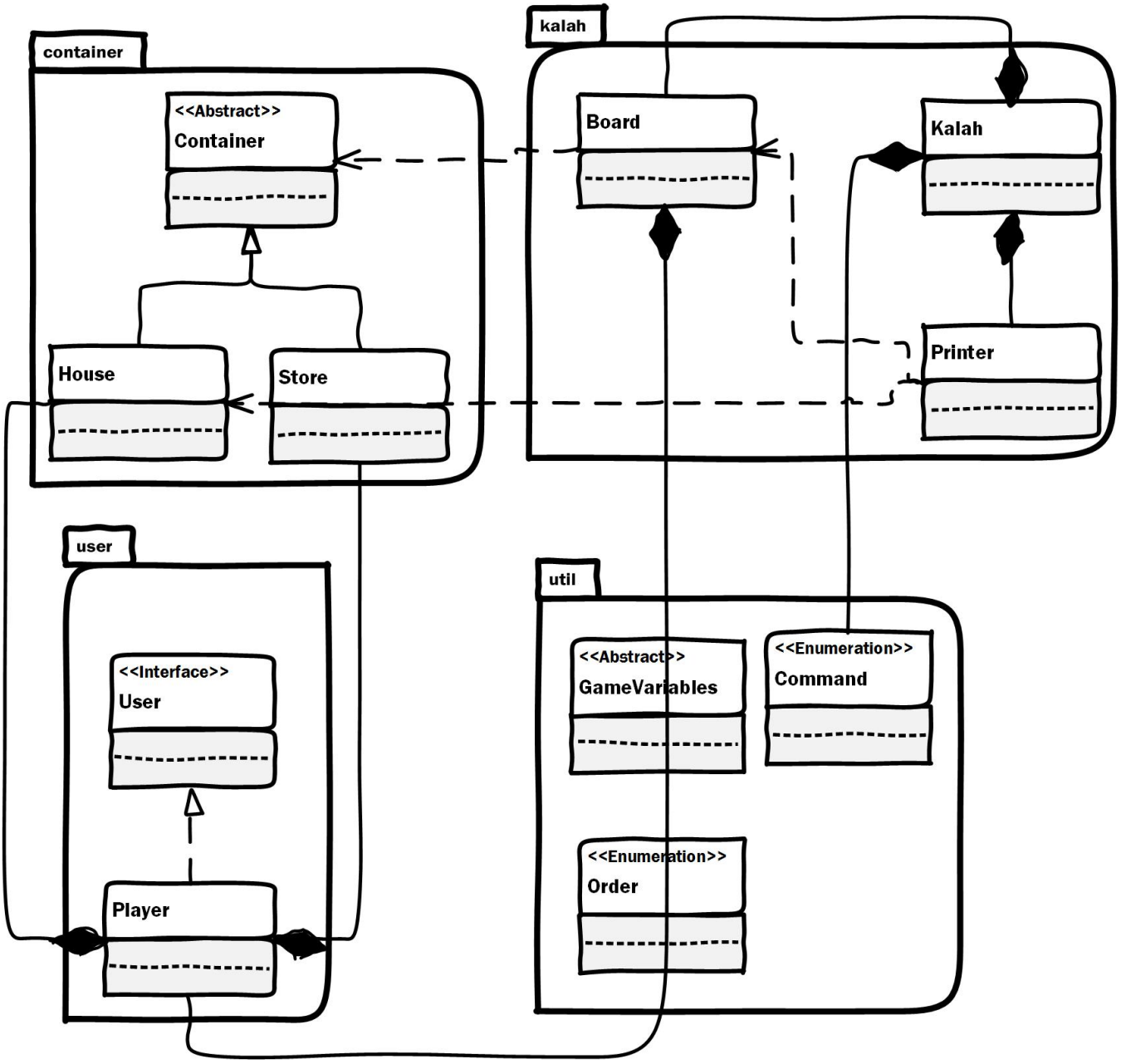

Fig. 10: The architecture of design S1.

APPENDIX D
UML DIAGRAM OF DESIGN SYU702

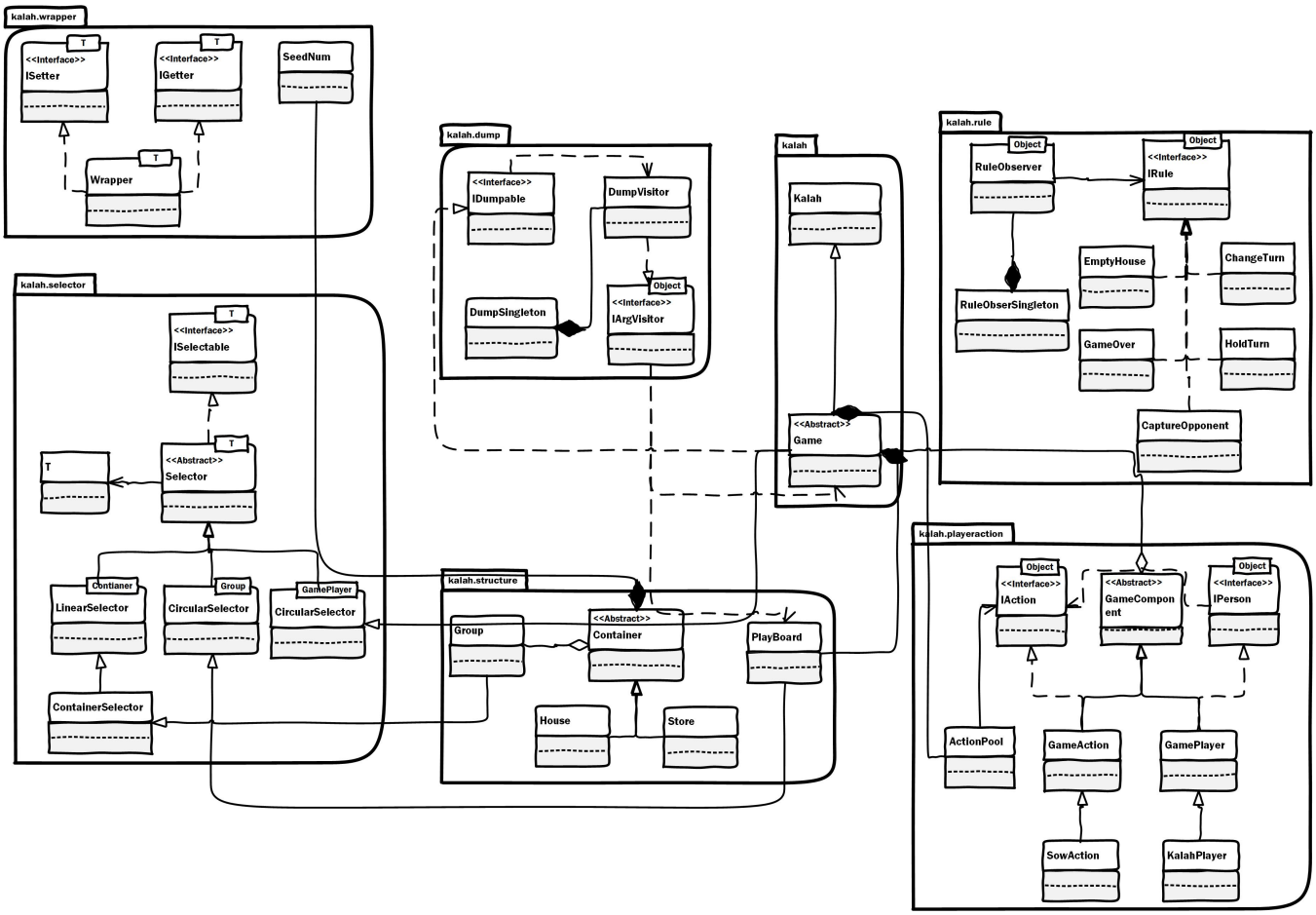

Fig. 11: The architecture of design syu702.

ACKNOWLEDGMENT

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