Changeability Design Considerations behind Kalah in Java

William Shin (wshi593)
SOFTENG701: Advanced Software Engineering
Development Methods
Department of Software Engineering
University of Auckland
wshi593@aucklanduni.ac.nz

Design Considerations

A. Model View Controller (MVC) Design Pattern

The Model View Controller (MVC) design pattern was employed to decouple the system and increase the separation of concerns between the model, view and controller. Low coupling in object-oriented programming (OOP) allows for a system specification to be changed, without needing to modify unrelated modules. For example, if the method of displaying information (the view) needs to be changed, the related code in the view package can be modified without needing to affect any of the system's models.

The following figure (figure 1) shows the relationships between all entities in *Kalah*. The *Pit*, *House*, *Store*, *Board*, *CircularBoard* and *Score* classes belong to the model package. The *ScoreController* and *GameController* belong to the controller package and *KalahView*, *AsciiKalahView* and *AsciiUtil* belong to the view package.

Notably, there is a high amount of cohesion within each package and a low amount of coupling between classes from different packages. The high amount of cohesion within each package shows that each module is focused and follows the Single-Responsibility Principle. Also, an attempt was made to follow the Dependency Inversion Principle which is based on the concept that high level modules should not depend on the implementation details of low level modules, and should depend on abstractions instead. Here, the *Kalah* class merely depends on the *GameController*, instantiating a *CircularBoard*, and an implementation of a *KalahView*, the *AsciiKalahView*. This allows the *Kalah* class to not depend on the lower level implementation details, and therefore improves the modifiability (and changeability) of the system.

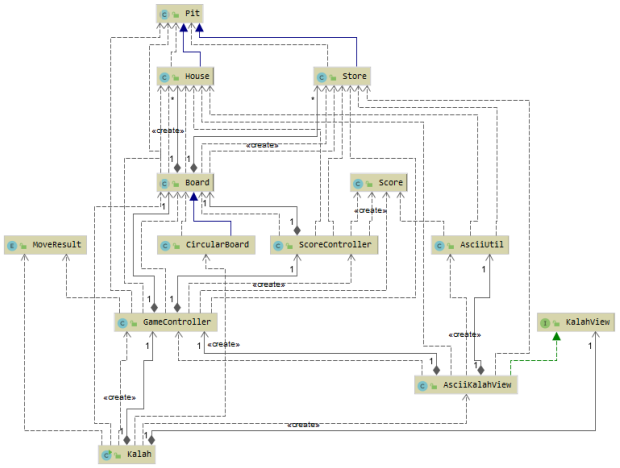

Fig. 1. UML Class Diagram of Kalah

B. KalahView Interface

The view package contains the *KalahView* interface, and an implementation *AsciiKalahView* and a util class, *AsciiUtil* which is used to produce and format *Strings* so that they can be displayed to the user.

The *KalahView* interface consists of five methods:

- *String promptMove();*
- *void printGameOver();*
- *void printEmptyHouse();*
- *void printBoard();*
- *void printResult();*

By using an interface, alternate implementations can be instantiated in the *Kalah* class. This is an example of the Open/Closed Principle being followed. The Open/Closed principle is based on the concept that software entities should be open to extension but closed to modification. Here, the system's behaviour in displaying information can be changed by creating another implementation of the *KalahView* interface rather than modifying the *AsciiKalahView* class.

The Open/Closed Principle improves modifiability as very few changes to the existing system would be necessary to display information differently. As mentioned in lectures, the Changeability Index (CI) does not take new lines of code into consideration, and therefore indicates that the view package's design has a high degree of changeability.

The five methods mentioned above were carefully selected to only include the five different ways information is displayed to the user. In the future, if more ways of presenting information are needed, the *KalahView* interface can be extended to follow the Interface Segregation Principle. Doing so would allow the system to maintain its changeability.

C. Board Abstract Class

For similar reasons as the *KalahView* interface, an abstract class was used to represent a Kalah board. The *Board* abstract class and its extension *CircularBoard* can be found in the board package within the model package.

Most of the implementation details related to the rules of Kalah are placed in the *Board* abstract class. The constructor parameters allow the number of players, initial seeds and houses to be set in the *Kalah* class. If the game's rules were to be extended to change any of these parameters, then this can be accomplished by changing the parameters when instantiating a concrete extension of the *Board* class, which only requires one line of code to change.

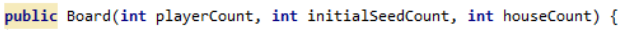

Fig. 2. Constructor of *Board* class

In addition, the *Board* abstract class contains an abstract method, *initializeBoard()*, which a concrete subclass, such as *CircularBoard*, must override. The method allows the subclass to decide how the pits are arranged. Again, this means that if the system's specifications related to how the Kalah board is arranged is changed, another subclass of *Board* can be created to satisfy the new requirements. By creating a new subclass, the existing code base not need to be modified a significant amount- only the type of subclass that is instantiated would need to be changed in the *Kalah* class.

ACKNOWLEDGMENT

I would like to thank Dr. Ewan Tempero for lecturing the SOFTENG701 course and dedicating their time and energy to ensure that their students are actively engaged in learning about software development. I would also like to thank any teaching assistants who are involved in the marking of this assignment.