Kalah Implementation Report

Dylan Fu
Department of Electrical, Computer
and Software Engineering
University of Auckland
Auckland, New Zealand
dfu987@aucklanduni.ac.nz

I. Object-Oriented Design

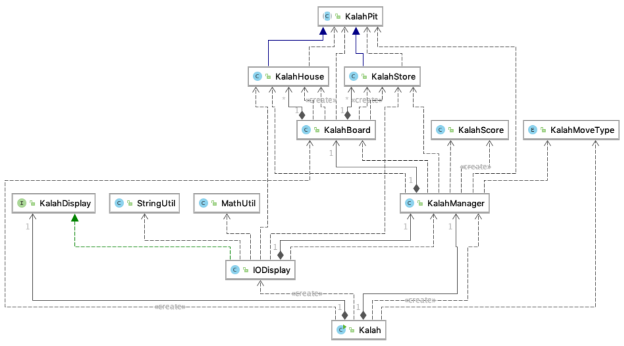

Figure 1: Class Diagram of Kalah Implementation

A. Classes and Objects

A class in Java is an abstract and user-defined data type, consisting of several functions and variables. The main purpose of a class is to store data and information, this is useful when. We want to encapsulate similar data together in a class. This class can be reused through instantiation of class objects avoiding code duplication of the same data type. This can be seen in my Kalah implementation (see Figure 1) where I have split up the model components of Kalah into: KalahBoard, KalahPit, KalahHouse, KalahStore and KalahScore. Then I have a view class called ConsoleDisplay that implements KalahDisplay, and then I have a KalahManager class for managing the game and a controller class called Kalah. These components of Kalah have been separated modularly to allow for code reuse and encapsulation to ensure high cohesion within a class, but I also made sure each class only contained methods relating to the intention of the class i.e. without including any unnecessary getter/setter functions that could be abused by many functions as a way of bypassing the service or controller layer. By ensuring low coupling and high cohesion between and within my classes, it allows for my implementation to be easily modified, thus lending my design to meet the changeability quality attribute.

B. Abstraction: Interfaces and Abstract Class

Abstraction in Java is the hiding of internal implementation details and only showing the functionality to the user i.e. What is hidden is how the feature works and what is shown through abstraction is what features are available. Both interfaces and abstract classes are used for the purpose of abstraction. Firstly, interfaces are used for total abstraction and it can achieve multiple inheritance, same as abstract classes but interface methods are always abstract. This is useful when the subclasses all implement the inherited methods differently. I have used an interface for the KalahDisplay class which declares four methods: promptNextMove, quitGame, gameOver, invalidHouse. These methods in the view layer are critical for letting the user know and important events. Consequently, any implementations of the KalahDisplay must implement these four methods. For this assignment, it required that the user interface be through keyboard input and ASCII console output. Therefore, I made a console implementation called ConsoleDisplay. However, if a different user interface was implemented as a replacement, the new implementation would be required to implement these four abstract methods. This would make sure that if a GUI is implemented, no other classes would need to be changed, as the new GUI would conform to the interface. Therefore, the application would have good changeability and no other classes would need to be modified. An abstract class was used for KalahPit and the subclasses KalahHouse and KalahStore inherited functionality from KalahPit as abstract classes allow the declaration of non-abstract methods. I used an abstract class because the subclasses KalahHouse and KalahStore have many common methods. Therefore, to avoid duplicating code I had code shared by the subclasses in the abstract class KalahPit i.e. sow methods are included in KalahPit as the logic for sowing is the same for both KalahHouse and KalahStore. By avoiding code duplication in multiple classes, we reduce the number of classes that need to be modified, thus promoting good changeability.

C. Architecture – Model View Controller Service Pattern

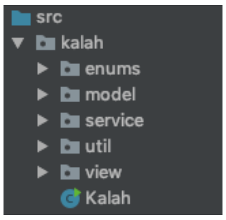

Figure 2: Package Structure showcasing the MVCS Pattern

I also followed the MVCS design pattern which is based on the common MVC pattern with an extra service layer. I used this pattern for separation of concerns, as it separates the components of application into distinct layers, where the model layer contains the business logic, the view layer will present the data to the user, the controller layer acts as an interface between the model and view layers, and the service layer is between the controller and model that can be called by many controllers. The pattern helps promote low coupling between classes. Subsequently, MVCS also promotes good changeability.

II. Future Application Extensions

A. Change of Rules or New Feature Extensions

1) Rule Variations

Currently, the application allows for variation in the number of houses and initial seeds in each pit. This can easily be done by just changing two input parameters in the controller class named Kalah. Thus, promoting changeability.

2) New Features

As the Kalah implementation has a good object-oriented design that promotes changeability, new features can easily be extended on top of the current application through the use of classes.

B. Implementing a Web Application

The current architectural design lends itself to being very easily modifiable. If in the future the user may want to play the game through a web app, then the current design wouldn't necessarily need to be modified, as a web service and a web controller could easily extend the current application without modifying the existing code. The front-end could just easily replace the current ConsoleDisplay class. Therefore, my design promotes good changeability in this case, where one would a web application.