SOFTENG 701 Assignment 3

A modifiable Mancala implementation

Ruoyi (Zoe) Cai
Software Engineering
The University of Auckland
New Zealand
Rcai861@aucklanduni.ac.nz

Abstract—**Planning and implementing a Mancala game to maximise modifiability, using change cases.**

Keywords—modifiability, change cases

I. Introduction

This report describes a Mancala implementation in Java designed to support modifiability in terms of change cases. It includes justifications for design decisions and evaluation of strengths and weaknesses of the design.

II. Modifiability in Terms of Change Cases

The Mancala implementation was designed to maximise modifiability using change cases. Potential new requirements or modifications to the game were identified and the likelihood of each potential modification was estimated. The most likely and the most critical modifications were identified, and the game was implemented to accommodate those potential change cases.

The change cases were identified by analysing each sentence of the requirements and picking out the variables or rules that could be changed. The following list consists of the potential changes identified and their likelihood (10 being the most likely, and 0 being impossible).

TABLE I. TABLE 1 CHANGE CASES

Change case	Likelihood
The display of the board might need to be modified. E.g., a new requirement might be switching the perspective of the board at each turn according to which player is currently picking, or a GUI is needed instead of ASCII text in console	9
The number of players could be changed	7
The number of houses and stores could be changed	6
The number of initial seeds in the houses and stores could be changed	10
The number of seeds that are moved or deposited at each turn could be changed	5
The order of the houses and the stores could be changed. Perhaps stores are ordered before houses	4
The order of the houses and the stores could be changed between players, e.g., order of houses could be mixed up such that one player's houses are no longer next to each other	4
The direction in which the seeds are sown could be reversed to clockwise	5
Rule one could be modified. E.g., if the house with the last seed sown contained a certain number of seeds, the player would get another turn	5
Rule two could be modified. E.g., if the last seed were sown in the player's store, player would have to have a specific number (an even number, perhaps) to get another turn	5
Rule three could be modified:	
If the last seed were sown in the player's own house, the house would have to contain a certain number of seeds for capture	5
If the last seed were sown in the player's own house, the opposite house would have to have a certain number of seeds before being captured	5
Score calculation could be modified, e.g., a seed in the store is worth two points whereas a seed in a house is worth one	8
The entire game could be changed such that the board is no longer used for Mancala, but instead for another game	1

III. Classes

Kalah is the main game class that creates a Board object and manages turns. It contains the main() method which creates a new instance of Kalah and calls play() to start the game.

Board represents the actual Kalah board, and contains all logic for modifications of the board and retrieval of information from the board. It keeps the houses and stores of all players in a CircularArrayList.

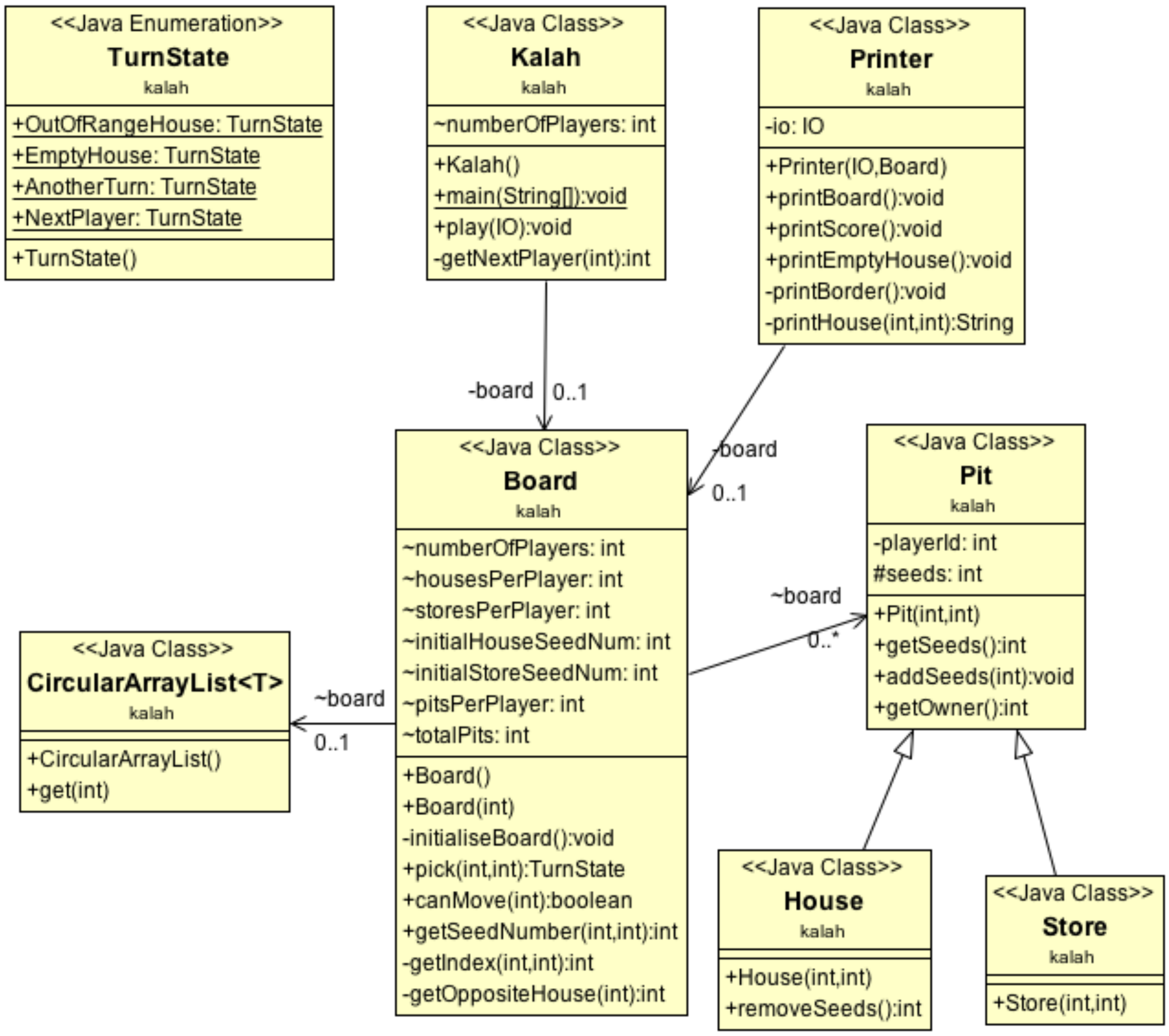

Fig. 1. Figure 1 UML class diagram of the implementation

CircularArrayList extends ArrayList, and simply ensures that given an index equal to or higher than the size of the array list, the get() method goes back to the beginning of the list and retrieves the appropriate element instead of throwing an exception.

TurnState is an enum class used by Board's pick() method to represent the outcome of a turn, such as AnotherTurn or EmptyHouse.

Pit represents a hole on a physical board. It is extended by House and Store.

House represents a player's house. It extends Pit and has an additional method to empty itself of seeds.

Store represents a player's store. It extends Pit and currently has no additional methods.

Printer handles all the printing logic, including displaying the board, asking the user for input, and displaying the game outcome and player scores.

IV. Justification

Modifiability tactics such as localise change, maintain semantic coherence, and information hiding were used to increase modifiability. Object oriented programming principles such as abstraction, encapsulation, polymorphism, and inheritance were also adhered to.

The code is well-commented, and the format and style of the code are maintained throughout to improve comprehensibility.

A. Classes

The TurnState enum was used for the outcome of each turn to improve code clarity. As several turn outcomes are possible, the alternative solution to returning the outcome would have been to return an integer that represents the outcome. However, using integers leads to ambiguity in both the return values in pick() in the Board class as well as in interpreting the returned value in the Kalah class, as a developer reading the code would not know what the integers represent without documentation. It is much easier to understand enum words. Using enum also allows the outcomes to be managed in one place and more outcomes to be added easily should the rules change.

Store and House both extend Pit. This has several advantages. As getOwner() and getSeeds() could be called on both Store and House objects, the Store and House objects can now be kept in the CircularArrayList board together, and methods can modify the objects (e.g., deposit a seed) regardless of which class the object is an instance of. This is an example of polymorphism and defers binding time.

Having a Pit superclass also enables more types of Pit to be created easily by creating a new subclass of Pit, if the rules of the game were changed to include more types of pits with special methods or properties.

The Printer class handles all of the printing logic. This maintains semantic coherence by keeping all the methods that deal with input and output together, and makes it easy for a developer to find any method related to printing that needs to be modified.

B. *Methods*

The number of players could be changed easily. In fact, the logic has already been implemented, and the only logic change required is to call a different constructor with the new player number. The initialiseBoard() method is able to initialise a board with any number of houses and stores for any number of players with any number of initial seed numbers.

The number of houses per player, stores per player, initial house seed number, and initial store seed number could also be changed easily by creating a new constructor which override the default Board field values.

By extracting the logic of initialising the board from the constructors into initialiseBoard(), code is reused and ripple effect is prevented if the board layout were to be changed in any way.

getNextPlayer() in the Kalah class allows the turn order to be modified easily, as only the code inside this method needs to be changed if the turn order were, for example, reversed to clockwise.

canMove() in the Board class takes in a player ID and returns whether the player has any moves left. This allows the game over logic to be changed easily.

getOppositeHouse() takes in a house index and returns the index of the house that is opposite to the input house. This allows more players to be added easily as the logic for calculating the opposite house position is kept in this one method instead of being hard-coded into the pick() method.

getIndex() takes in a player ID and the house number, and returns the index of house or store in the board list. This allows the layout of the board to be changed easily, as this is the only method with knowledge of how the pits are laid out on the board. If for example the stores were placed before the houses in the new layout, the only method that would need to be changed is getIndex(), and other methods calling getIndex(), such as getSeedNumber(), can still get the pit index with only the player ID and the pit number.

printScore() in Printer could be used for any number of players.

C. *Variables*

The variable ‘playerId’ in Pit is private and has no setters, so it can’t be modified once it is set, which adheres to encapsulation. Most other variables in other classes, such as numberOfPlayers and initialHouseSeedNum in Board are also set to private. The ‘board’ list itself is also kept private, and can only be modified via methods in the Board class. This restricts communication paths and limits any changes to the board layout to the methods inside the Board class.

The variable ‘seeds’ in Pit is protected so that it can be accessed by Pit’s subclasses.

Variables are used for default values in the game, such as numberOfPlayers and the starting player ID, and are named so that anyone reading the code could understand instantly what the variables represent and change the values if necessary.

The variables currentPit and lastPit in the pick() method in Board are used to store a pit that might be read multiple times in the lines of code following. They are used so that the code is easier to understand.

Similarly, inside the method getOppositeHouse() — which calculates the index of the house opposite to the input house — has variables oppositePlayerIndex, indexOfFirstOppositeHouse, oppositeHouseIndex, which represent and store the results of separate steps in the calculation. Although the calculation could be done in one step, these variables are used to improve code clarity and comprehensibility which in turn improves modifiability.

D. *Weaknesses*

In the Printer class, the printBoard() method has hard-coded values for a two-player game. However, as the printing logic is kept together in this one method, if more players were to be added, only the content of this method would need to be changed.

The number of seeds deposited is currently hard-coded as 1.

The score calculation logic is inside the Printer class as it is simple at the moment and is only performed at the end of a game when Printer prints the score. However it could easily be moved out if score calculation becomes more complicated or is used more elsewhere.

V. Evaluation

Modifiability is evaluated below based on how much change is required to achieve the change case. 10 represents extremely easy to modify, and 1 represents extremely difficult to modify.

VI. Conclusion

This implementation of Mancala is modifiability in terms of change cases, as the modifiability of the change cases correspond to their likelihood.

Change case	Likelihood	Change Required	Modifiability
The display of the board might need to be modified. E.g., a new requirement might be switching the perspective of the board at each turn according to which player is currently picking, or a GUI is needed instead of ASCII text in console	9	printBoard() method inside the Printer class	8
The number of players could be changed	7	Call a different constructor	10
The number of houses and stores could be changed	6	Create a new constructor that overrides the default values stored in fields.	9
The number of initial seeds in the houses and stores could be changed	10	Change the values stored in the field inside the Board class	10
The number of seeds that are moved or deposited at each turn could be changed	5	Modify the pick() method logic inside the Board class	5
The order of the houses and the stores could be changed. Perhaps stores are ordered before houses	4	Modify initialiseBoard(), getIndex(), and getOppositeHouse() inside the Board class	5
The order of the houses and the stores could be changed between players, e.g., order of houses could be mixed up such that one player's houses are no longer next to each other	4	Modify initialiseBoard(), getIndex(), and getOppositeHouse() inside the Board class	5
The direction in which the seeds are sown could be reversed to clockwise	5	Modify getNextPlayer() inside Kalah	8
Rule one could be modified. E.g., if the house with the last seed sown contained a certain number of seeds, the player would get another turn	5	Add logic to pick() and create new enum if necessary	7
Rule two could be modified. E.g., if the last seed were sown in the player's store, player would have to have a specific number (an even number, perhaps) to get another turn	5	Add logic to pick()	7
Rule three could be modified:			
If the last seed were sown in the player's own house, the house would have to contain a certain number of seeds for capture	5	Add logic to pick()	7
If the last seed were sown in the player's own house, the opposite house would have to have a certain number of seeds before being captured	5	Add logic to pick()	7
Score calculation could be modified, e.g., a seed in the store is worth two points whereas a seed in a house is worth one	8	Modify printScore()	7
The entire game could be changed such that the board is no longer used for Mancala, but instead for another game	1	Modify everything	1

Fig. 1. Figure 2 Change case modifiability evaluation