Mancala Modifiability

A story of modifiable software.

Abby Thompson
of *Software Engineering*
UoA
Auckland, New Zealand

I. Introduction

The following will describe the thought behind a terminal based implementation of Kalah. The game Kalah, is more popularly known as Mancala. The program is designed for modifiability in mind. It will look at how modifiability is understood and measured. Leading onto how modifiability was implemented into the code for mancala.

II. Design

My main thoughts on the design was to have a interface almost type class and a game logic class as the foundation for the program. The interface type class would use the user's input to manipulate the game through the game logic and then output the results to give the user their board visualisation. The game logic class is Game.java and connects all parts of mancala together. It would bring the players, pits and stones etc together to implement the game's rules.

I've implemented these features to the best that I could, to a design that I thought was modifiable and to pass all the required tests. The bulk of the code is within Game.java and Kalah.java, where the objects for use are all defined respectively.

The design is quite straightforward, due to the straightforward nature of the game. Figure 1 gives an easy outline of the program, not all the methods are included but it shows the skeleton of the design.

III. Modifiability

Before going into the justification of the design, I shall define modifiability. There are a multitude of ways to define this Quality Attribute. The point, I think, is that you can't just define it without having a percentage of leniency.
As a take on of the definition in lecture 2, slide 7, I believe to believe that a well modifiable program can be altered with ease, without decreasing the program's existing quality. It's easy to be said. It's harder to measure and even harder to produce especially in a larger project. Modifiability in software, I would say, relates to technical debt within code. When creating software it's easy to not think towards future changes or being a time schedule for launch, when technical debt usually occurs.

Measuring this attribute is difficult and there are equations that can help this process. However, as listened to in the lectures, these aren't quite right. They make sense on a small scale of course. If you only have to alter one class then wow this is so modifiable. This isn't the case when realising that all the code could be in this class, making it quite hard to understand. I believe that the measurement is very subjective and needs to be altered to suit the design. Understanding what you're measuring and how to alter the definition of modifiability to suit is extremely important.

IV. Justification for Design

There are quite a few tactics to thinks about when looking into the design for this game. I've tried to create design that was quite simple where not all the classes would be affected by a change in one. Any change to the enumeration classes will only affect the game class for example. I've tried to separate how the game works from the code that will implement the code and create the game for the user on the terminal. So if the game ends up looking different on the terminal then there will be one place for code to be changed. I thought that naming methods, for such changes, quite obvious and related. As if you were to change the printing to the terminal my methods are called printBoard, printHouses etc.

I believe this is a very important part of code which related to modifiability. That is, making sure that the classes, comments, variables and methods are all very obvious in their

naming. I have tried to do that in this assignment. When you're modifying code, there is a part of that process that involves having to understand how it all relates and works in order to make sure that quality is upheld. Without this clarity, especially in legacy code, time is lost and I feel that that inflicts on the modifiability of code as ease and efficacy are lost.

I've kept most of the related code of what I could in this tiny program together. This is in terms of methods in classes and the flow of the relations between each class. It's quite logical and easy to understand as each fits into each. Where the game is played is the Kalah class and the main rules of the game are implemented are in the game class.

In terms of looking towards the future of this class, I tried to think of cases that would require possible change. The number of stones are somewhat preference to a user, so I made that a final variable in the kalah class. The number of pits was also an easily changeable variable. These small things can in turn make such expected changes very small.
I tried to reflect such thinking into the whole program. Expecting a change to occur in the future and preparing for it the most effective way is a quick way to increasing modifiability. However, it's the unexpected changes that can trip the design up. Say the code was to be used for another board game. It's got the skeleton for a board game with pits and houses. With my design there is no problem with multiplicity of pits and players due to abstraction but if they were to change it to an absurdly different game and expect the program to comply, that's a completely different game of modifiability. But I'd say this should cope with the smaller scale changes.

Of course there is no way that this is a full proof and entirely modifiable design. This design does have dependencies, that decrease modifiability. The design doesn't have any patterns in it, as it is possibly too small to warrant.

V. Description of Classes

The following are all classes in my source code.

A. Game.java

This class is basically where I have put the main game logic. Where the class gets the players, modes etc and calls the other classes to bring the design together.

B. Kalah.java

The Kalah class links the user to the game logic. It prints all the outputs, collects input and produces the visualisation of the game from start to finish.

C. House.java

The house is the name for the end pit next to the user. It is basically the score. Stored within a list of stones where stones get added until the end of the game.

D. Move.java

This is an enumeration class. It is the status of the game's current move. Letting the program know, if an illegal move has been played, a steal or when it's still a player's turn.

E. Pit.java

Pit.java is the class representing the pit objects. These have a list of stones that need to be removed when a player chooses to move from it. Stones can also be added during a move.

F. Player.java

This class uses the pit and house classes to create the player's representation of the board. All the manipulation to the player's pits and houses happen through the player.

G. Status.java

This is also an enumeration class. The class represents the status of the entire game. Who's turn it is and when it is finished.

H. Stone.java

This is a constructor for a stone object to be represented within the pits and houses of each player. Very simple class.

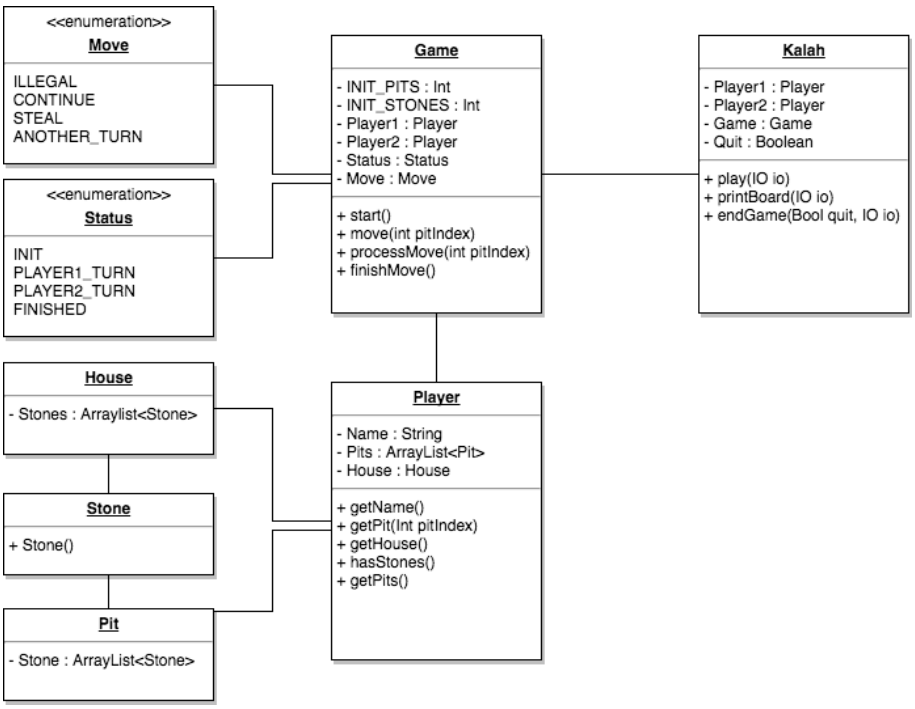

Fig. 1. *UML Diagram for Mancala design.*