Assignment 5

Jaedyn Damms
Department of Computer Engineering
University of Auckland
Auckland, New Zealand
Email: jdam534@aucklanduni.ac.nz

I. SOLID Principles

Two S.O.L.I.D principles [1] my design follows are:

- Single-responsibility principle (SRP)
- Interface segregation principle (ISP)

SRP states *"A class should have one and only one reason to change, meaning that a class should have only one job"*.
ISP states *"A client should never be forced to implement an interface that it doesn't use or clients shouldn't be forced to depend on methods they do not use."*.

II. Changes to Assignment 3 Submission

In Assignment 3 I had **five** classes:

1) *Building* - Parent class for store and house
2) *Store* - Class for a players store
3) *House* - Class for a players house
4) *Board* - Control movement of seeds on board, and acts as a game controller
5) *Kalah* - Acts as an I/O manager, game controller, initializer and sets up the game board

In Assignment 5 I now have **ten** classes:

1) *Board Manager* - Control movement of seeds on board
2) *Building* - Parent class for store and house
3) *Store* - Class for a players store
4) *House* - Class for a players house
5) *Game Controller* - Control game flow
6) *Kalah* - Creates the game components, starts the game (main)
7) *Input Manager* - Manages inputs
8) *Output Manager* - Manages outputs
9) *Kalah Constants* - Contains constants of the system (e.g. Number of houses, number of players, print statements etc.)
10) Turn Manager - Manages the players turns

The only classes that remain **unchanged** are:

1) *Building*
2) *Store*
3) *House*

Also, I have created **five** interfaces that help define what a the Kalah game is.

1) *KalahBoardLogic*
2) *KalahBoardInput*
3) *KalahBoardDisplay*
4) *KalahBoardRules*
5) *KalahGameRules*

The *KalahBoardLogic* interface describes the functionalities that a Kalah board should have. The *KalahBoardInput* interface describes the inputs that the Kalah game relies on. The *KalahBoardDisplay* interface helps describe how the Kalah game is to be displayed. The *KalahBoardRules* interface defines the Kalah board rules (*capture check*). The *KalahGameRules* interface defines the Kalah game rules (*end game rule*).

III. How Design Reflects Use of SOLID

A. SRP

Each of my classes has been re-factored so they are only responsible for one of the functionalities provided by the software, each of my classes responsibilities have been listed above in section II. Lisiting a few: the board manager class will only need to change if the movements of the seeds on the board need to change. Store/house class will need to change if the pits on the board need additional functionalities. Game controller will only need to change if the game logic needs to change. This is the same for the rest of the classes, they each only need have one reason to change, again, as seen in the list in section II.

B. ISP

I have created five interfaces: *KalahBoardLogic*, *KalahBoardInput*, *KalahBoardDisplay*, *KalahBoardRules* and *KalahGameRules* to help define the functionalities of Kalah. The *KalahBoardLogic* interface describes the functionalities that a Kalah board should have, my *Board Manage*r class implements this interface to ensure that it provides the logical functionalities that the Kalah game requires. The *KalahBoardInput* interface describes the inputs that the Kalah game relies on, my *Input Manager* class implements this interface to ensure that it provides the inputs that the Kalah game requires. The *KalahBoardDisplay* interface helps describe how the Kalah game is to be displayed, my *Output Manager* class implements this interface to ensure that the game's display follows that of which is specified by the Kalah game. The *KalahBoardRules* interface defines rules which the board should follow, my *Board Manager* class implements this interface to ensure that the game's board rules are followed, in this case the *capture check*.The *KalahGameRules* interface defines rules which the game should follow, my *Game Controller* class implements

this interface to ensure that the game rules are followed, in this case the *end game rule*.

IV. Impact on Quality Attribute

The use of SRP and ISP has improved the *maintainability* of my code design. *Maintainability* is the ease of which the system is able to implement changes, this could effect the time taken or effort required add/change components in the system or to restore the system to an operational status following a failure. Improving the *maintainability* of a system can increase availability and reduce run-time defects. According to Microsoft's documentation on quality attributes [2] *"an applications maintainability is often a function of its overall quality attributes but there a number of key issues that can directly affect maintainability"* . From the key issues mentioned in Microsoft's documentation on quality attributes [2] the use of SRP and ISP has helped my design solve the following issues:

1) Logic code of components and segments is not cohesive
2) Code base is large, unmanageable, fragile, or over complex
3) Excessive dependencies between components/modules
4) Reliance on custom implementations/features - DONE (maybe expand)

Following SRP and ISP meant that each of my classes only had one responsibility and a good level of abstraction, this makes my classes more cohesive and less complex. This also makes my design more manageable, due to it being easier to manage classes that have a single responsibility with a good level of abstraction rather than classes that have more than one responsibility, as the code structure/design is more clear and defined. According to the list of issues with maintainability, decreasing complexity and increasing cohesion and management of my code base will consequently increase the maintainability of my design.

Due to functional responsibilities being held by a single class instead of that functional responsibility being shared by more than one class, SRP helped reduce the dependencies between classes. Meaning that variables and values could be maintained internally within the class rather than having to be passed through to classes (via parameters etc.) in order to fulfill the functional responsibilities of the design. ISP also helped reduce cross-layer (between classes) dependencies through using interfaces. The *KalahBoardLogic* interface declares the logical/functional requirements of a Kalah board, the *KalahBoardInput* and *KalahBoardDisplay* interfaces declare the inputs and outputs of the Kalah game and lastly the *KalahBoardRules* and *KalahGameRules* interfaces declare the rules that the Kalah game should implement, all these requirements (dependencies) need to be satisfied in order for the design to function properly.

Also, following SRP and ISP consequently led to good refactoring and abstraction of the design, this resulted in my design becoming less application specific as the design became more modulated. This helped solve one of the above issues by reduced the amount of custom implementations that my design had, thus increasing the overall maintainability of the design. According to Microsoft's documentation on quality attributes [2], improving on each of these key issues listed above will increase the maintainability of my design.

References

[1] "S.O.L.I.D: The First 5 Principles of Object Oriented Design", Scotch, 2018. [Online]. Available: https://scotch.io/bar-talk/s-o-l-i-d-the-first-five-principles-of-object-oriented-design. [Accessed: 01- May- 2018].

[2] "Chapter 16: Quality Attributes", Docs.microsoft.com, 2018. [Online]. Available: https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ee658094(v=pandp.10). [Accessed: 04- Jun- 2018].