SOLID Principles & Modifiability

SOFTENG 701 Assignment 5

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

I. Introduction

This report demonstrates the use of two SOLID principles in a design of the game Mancala and explains how the incorporation of the principles improves the modifiability of the designs.

II. Definitions of Modifiability

Two definitions of modifiability are used to evaluate the design.

The first defines modifiability in terms of change cases. Potential new requirements or modifications to the game that are related to the parts of the code that incorporate the two SOLID principles are identified. These changes cases are used to determine the degree to which the incorporation of the two SOLID principles improves the design.

The change cases are identified by analysing each sentence of the requirements and picking out the variables or rules that could be changed. Only the change cases relevant to the parts of the design that incorporate the two SOLID principles are included in this case.

Modifiability is then evaluated based on how much change is required to achieve each change case. To evaluate modifiability of the design in terms of each change case, the second definition is used, which is the ISO 25010 definition of modifiability: "Degree to which a product or system can be effectively and efficiently modified without introducing defects or degrading existing product quality."

One way to ensure a change to a system does not introduce a defect into a particular module in the system is if the module does not need to be changed. Hence the effect of incorporating the principles to modifiability is evaluated by comparing the changes required to the current design and the changes required to a design that does not incorporate the principles.

III. Principle 1: Liskov Substituion Principle (LSP)

LSP states that child classes must be substitutable for their parent classes. The incorporation of this principle provides context reuse — the context can remain unchanged but the program bahaviour changes.

This principle is incorporated in the classes that depend on the class Pit. Pit is an abstract class that is extended by House and Store. Pit contains common implementation between House and Store, such as methods that return the number of seeds contained in the pit or the owner of the pit, as well as a method that adds seeds to the pit. The child classes Store and House then extend Pit with additional functionalities. For example, House implements an additional method that removes all the seeds inside the house.

LSP is followed by the field board inside the MancalaBoard class, which contains a CircularArrayList of Pit objects. This allows either House or Store to be substituted into the circular arraylist.

Examples that demonstrate this principle in the implementation are:

- When the method pick() is called, which deposits a seed into pits following the pit picked by the player, the addSeed() method is called regardless of whether the pit is a House or a Store type.
- When the owner of the Pit needs to be retrieved, getOwner() is called regardless of the type of the Pit.

The change cases affected by this principle are:

The layout of the board could be changed:

- A new type of Pit could be introduced with different behaviour. For example, all Houses are replaced by a new type of House with different behavior.
- The number of houses and stores could be changed.
- The order of the houses and the stores could be changed. Perhaps stores are ordered before houses.
- 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.

The rules of the game could be changed:

- The number of seeds that are moved or deposited at each turn could be changed.
- The direction in which the seeds are sown could be reversed to clockwise.

The alternative design that doesn't incorporate this principle would not have a circular arraylist containing Pit

objects. Instead it would need to store the House and the Store objects in separate lists, as the child classes House and Store would either not have the methods or the correct implementation for the methods that the parent class Pit has, and hence could not substitute the parent.

For the first type of change cases where the layout of the board is changed, it is clear that the current design is more modifiable than the alternative design, as the alternative design would not only require the two separate houses and stores lists to be changed, but also any code that keeps track of and maintains the order of the houses and stores, and calculates the index of the house or store needed.

For example, where a new type of Pit could be introduced with different behaviour, the current design would only require that a new class extends Pit with the new behaviour. If all Houses were to be replaced by a new type of House with a set of different behaviour, the circular arraylist would only need to be populated with the new class objects, and the remaining code would not need to be changed. This requires a lot less changes than the alternative design, which would require all code that refers to the House class be changed to refer to the new class. Changing less existing code in the current design makes it less likely for defects to be introduced.

For the change cases related to the rules of the game, only the logic related to the specific change case needs to be modified in the current design, which is located in one place as all houses and stores are stored in one arraylist. For example, if the direction the seeds are sown were to be reversed to clockwise, only the code that controled the traversal of the arraylist would need to be changed in the current design. In the alternative design however, where the houses and stores must be stored in separate lists, the traversal of both the houses list and the stores list would need to be changed.

IV. Principle 2: Dependency Inversion Principle (DIP)

This principle requires modules to depend on abstractions, not concretions. Following this principle means abstract classes or interfaces should only depend on other abstract classes or interfaces, and that concrete classes too should depend on abstract classes or interfaces.

This principle is incorporated using interfaces and dependency injection. The current design consists of two modules — board and display — which have interfaces that both the components inside the modules themselves and the other modules depend on.

MancalaBoard implements the PitsBoard interface, which both MancalaPrinter and Kalah classes depend on. This means that if another implementation of a board with pits were to be used instead, the MancalaBoard class could be substituted with another class that implements PitsBoard without requiring changes in MancalaPrinter and Kalah.

Similarly, MancalaPrinter implements the Printer interface, which the Kalah class depends on. If another implementation of printer is needed a new class could be used instead of MancalaPrinter, as long as it implements Printer.

The change cases related to this principle are:

- 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
- 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

A new board with different layout or behaviour is needed:

- The number of players could be changed
- The number of houses and stores could be changed
- The number of initial seeds in the houses and stores could be changed
- The number of seeds that are moved or deposited at each turn could be changed
- The order of the houses and the stores could be changed. Perhaps stores are ordered before houses
- 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
- The direction in which the seeds are sown could be reversed to clockwise
- 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
- 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
- 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
    - 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
- The entire game could be changed such that the board is no longer used for Mancala, but instead for another game

For the change cases related to the display of the board and the score calculation (which is done by the printer), if changes are significant or if another implementation is required, a new class implementing the Printer interface could substitute MancalaPrinter, which could be done by modifying only the line in the play() method of the Kalah class to instantiate another Printer implementation. Existing code calling Printer methods would not need to be changed, and hence defects are less likely to be introduced.

For the change cases related to a new board, a new class implementing PitsBoard could substitute MancalaBoard, which could be done by simply modifying the constructor of the Kalah class to instantiate the new class. Code calling the PitsBoard methods does not need to be changed.

Dependency injection is used to inject PitsBoard into Printer implementations. If a new implementation of PitsBoard is required, Printer would not need to be changed as it depends on an abstraction and not the concrete MancalaBoard class, hence avoiding changes to existing code and introduction of defects.

Alternative designs that do not incorporate this principle would require any code that instantiates and uses either the MancalaBoard or the MancalaPrinter classes to be changed which, compared to the current design, is a lot less modifiable as it is a lot more likely to introduce defects from modifying a significant amount of existing code.

V. Conclusion

The incorporation of the two SOLID principles improves modifiability of the Mancala design by preventing significant changes to existing code and reducing the probability of introducing defects to the existing code.