The SOLID Principles and their affect on Modifiability

Carson Lourenco
Department of Electrical and Computer Engineering
University of Auckland
clou241@aucklanduni.ac.nz

I. Introduction

The SOLID principles have the potential to improve the modifiability of a system if implemented correctly. This report outlines an analysis of five implementations (two of which were implemented by the author, three of which were supplied) of the game Kalah with regards to their use of the SOLID principles and how modifiable they were with regards to certain change cases. It was then determined whether or not the SOLID principles affected these assessments of modifiability in any way.

II. SOLID Principles in developed implementations

A. Assignment 4

Generally speaking it is difficult to infer from inspecting code whether or not SOLID principles were considered when the code was written. In this case however, it is known that the implementation wasn't written with the SOLID principles in mind.

The only of example of a situation that unintentionally makes use of a SOLID principle is the definition of the KalahBoardInfoRetriever interface as input to the KalahIO class to print out the Board class (which implements the interface). This would be an example of the Interface Segregation Principle (ISP) as the KalahIO will only be able to get information about the board and not manipulate the board as it only has access to the methods it needs from the board object.

B. Assignment 5

In an attempt to improve the modifiability of the Assignment 4 implementation outlined in section 2A, the code was re-factored with the SOLID principles in mind. The SOLID principles that were applied were the Single Responsibility Principle (SRP) and the Dependency Inversion Principle (DIP). These principles were applied specifically to improve the modifiability of the program with regards to the following change cases

- Adding a robot player
- Changing the number of houses
- Changing the direction seeds are sewn

the previous version of this program (Assignment 4) already allowed for the changing of houses per player by changing a single static variable, so the code didn't need to be re-factored with regards to this change case. The other change cases however, weren't straight forward so the code was refactored to simplify adding these changes.

With regards to changing the direction that seeds are sewn, SRP was applied to remove the part of the game logic that gets the next place to sow seeds. In order to achieve this, the entire game logic was re-factored and simplified by separating it from the board initially with the idea that this was the "single responsibility" that would exist in the single method of a new class.

If a developer wished to change the direction that the seeds we're sewn, they could change the implementation of that method. However, after further consideration it was seen that when considering the definition of SRP from the perspective of "one reason to change", there were multiple responsibilities in the logic e.g. capture rules, determining next player etc. So with the change case of "changing direction of seeds sown" in mind, the part of the game logic that determines the next place to sow seeds was separated into an AClockwiseTraversal object. This class has one method getNextTraversalState() defined and returns a TraversalState object which the game logic uses to distribute seeds. This class only has one responsibility and one reason to change - that is to change how the next place to sow is determined - thus is a valid use of the SRP principle.

The AClockwiseTraversal class implements the KalahTraversable interface which defines the getNextTraversalState() method. The reason for defining the class using an interface is so that Dependency Inversion can be applied as well (DIP). The game logic class (KalahStandardLogic) takes in a KalahTraversable object in the constructor so that the traversal object that defines how the the next place to sow is found can be defined via dependency injection (which could be a new Traversal object that defines the Clockwise traversal). Polymorphism will allow for the client class of KalahTraversable to work. As the logic depends on an abstract definition it is a valid application of the DIP.

With regards to the "Adding a robot player" change case, this followed a similar approach. It was seen in the previous implementation that the only thing the user ever did was pick a house to take seeds from. This logic was separated into its own class HumanPlayer with one method getPlayerInput() and returns an String representing the house selection for the player or the 'quit game' input. This is another example of SRP in action as the class only has one reason to change i.e. change the way the player input is specified.

This class was defined using an interface KalahPlayerInput and is depended upon by the Game Instance class (specified in constructor). In other words, concrete implementation of KalahPlayerInput is only decided at run time via dependency injection and polymorphism will allow for the code to still work. Should a user wish to have a robot player, they would create a new class implementing the interface and change concrete definition that gets injected via the constructor. This is an example of DIP as the GameInstance class depends on the abstract definition of the KalahPlayerInput.

III. SOLID Principles in supplied implementations

The following section describes the results of analysing supplied implementations of the game Kalah for the potential application of the SOLID principles.

A. Implementation 1003

SRP: SRP is applied with the Store class as its only reason to change is when the number of seeds in a pit is changed.

OCP: OCP is a likely candidate for a SOLID principle being applied due to the IDisplay, IGame and Pit interfaces which are all closed for modification (due to being depended upon in other classes, a change would likely cause faults in other areas of the code) and open for extension when classes implement the interfaces and override their behaviours.

LSP: LSP is present as interfaces are defined in the Kalah class (IDisplay and IGame) and concrete definitions are assigned i.e. the ”child” class can be substituted for the base Interface and it makes sense to do so.

Furthermore, the House and Store objects implement the Pit interface to allow them both to exist in an ArrayList of Pit objects (seen in the Game class) via polymorphism.

ISP: Interface segregation is not present at all in the project as all concrete definitions implement at most one interface and share all of the methods of the interface (and don't present any more). There is no way to separate the interfaces presented in concrete types.

DIP: Dependency inversion is present in the Kalah class where concrete definitions are stored in fields of the interface type. For example the concrete Game class is store in a field of the interface type that it implements. The rest of the logic in the class then depends on the methods that are in the interface. As the logic depends on an abstract definition this could be considered a form of dependency inversion.

B. Implementation 1024

SRP: The HumanPlayer class has a single responsibility and that is to get the house choice of a player in the game. The only reason to change this class is to change how the player input is determined therefore is a valid use of the single responsibility principle.

OCP: The abstract Player class is an example of where OCP is applied. Player is open for extension - classes can extend the class and override the behaviour of the abstract method getChoice() - and closed for modification as classes depend on abstract type and changes could cause faults to occur e.g. if the getChoice() method is removed then the Kalah class which depends on Player will also have to change.

LSP: There are two abstract definitions in the project, that is the Player abstract class and the BoardPrinter interface.

In the Kalah class the logic in the class depends on the BoardPrinter field defined which has a KalahBoardPrinter assigned to it (allowed through polymorphism as KalahBoard-Printer implements BoardPrinter). The same can be said for the player fields in the class which are defined as the Player type however have the concrete HumanPlayer assigned to it via polymorphism (as HumanPlayer ”extends” Player).

ISP: There is no interface segregation possible as all classes extend at most one abstract definition (either abstract class or interface) and in the situations where this is the case, interface segregation isn't possible as the concrete definitions share the same interfaces as the abstract definitions and don't present any more functionality than what is defined in by the abstracted type.

DIP: The setup method in the Kalah class assigns concrete definitions to the abstracted fields (interfaces and abstract classes) that the rest of the class depends on (e.g. the logic in the class depends on the abstract Player class and the Board-Printer interface fields which are assigned the HumanPlayer and KalahBoardPrinter objects respectively).

As the logic in the class depends on abstracted types, this is a form of dependency inversion.

C. Implementation 1037

SRP: The main example of SRP being used would be in all of the classes in the rule package that implement the IRule interface. They all override the single method defined in the interface and present no extra methods to clients.

The DumpSingleton class is another example of the SRP being used in this implementation. Its sole responsibility is to maintain a DumpVisitor singleton instance. The only reason to change would be to change the object that the class maintains a singleton for.

It could be argued that the LinearSelector class has one responsibility and that is to return the next valid iteration in a collection. However this responsibility is defined with two methods MoveToNext() which sets the next index, and getNext() which gets a variable based on the current index in a collection. There is a possibility that this may not be valid use of SRP

OCP: As interfaces were part of the implementation and all examples of classes implementing these interfaces had some form of overriding the methods defined in the interfaces, the open close principle is being applied. This can especially be seen with the classes in the rule package which depend on the IRule interface. The IRule interface is closed for modification as other classes in the same package depend on it and would break if the Affect() method definition were to change. It is open for extension as any new class implementing the IRule interface can extend its behaviour.

LSP: The RuleObserver class maintains a list of IRule objects and the Observe() method adds IRule objects to this

list. The clients of this class add the various concrete classes that implement the IRule interface to this list. The base definition in these cases (IRule) are being substituted with the classes that implement the interface and is therefore an example of LSP

ISP: There is the potential for ISP to be used in this implementation however it isn't exploited anywhere in the code. SeedNum extends Wrapper which implements IGetter and ISetter. A potential example where this could be used is the IGetter interface could be defined in the input to a method; SeedNum could be fed as input to that method and the method will only be able to perform the methods on SeedNum defined by IGetter. The same could be said for ISetter. There is however no evidence that this interface segregation is used anywhere even in places where it could've been used. A more specific example would be in the Container class. A SeedNum is initialised and the get() method defined in IGetter is the only method that gets invoked on the object. An IGetter object could have been specified instead of the concrete SeedNum object in this case so that the method is unable to perform the set() method available.

DIP: In the RuleObserver class, the Observe method defines an abstract definition as input. This implies that dependency injection is occurring where clients specify a concrete definition of the IRule interface (a class that implements the interface). As the method depends on an abstract definition, this is an example of dependency inversion.

IV. Modifiability Assessment

The modifiability of each implementation in this case will be assessed on two criteria with regards to two change cases. The two criteria are:

- changes involve the fewest possible number of distinct elements (lines of code)
- A subjective judgment on how simple it would be to implement the change

The following change cases are considered

- Adding a robot player
- Changing the number of houses
- Changing the direction seeds are sewn

Furthermore, each implementation will be analysed to see if any of the SOLID principles affected its modifiability.

A. Assignment 4

In order to add a robot player to the implementation, the way user input is retrieved will have to be changed. This involves replacing the method that assigns this variable to accommodate for an additional robot player object which would be approximately ¡10 lines of code to be changed. Implementing the change will be fairly simple in the client class of the player objects however will be difficult with regards to adding a robot player(depending on how complex the logic is which could involve hundreds of lines)

With regards to the changing the number of houses, this functionality is already built into the system and requires a change to one line of code so simple.

In order to change the direction that seeds are changed, the entire playerTurn() method needs to be re-factored in order to effect the change, this is in the range of greater than a hundred lines of code and will be reasonably difficult as the logic in that class is dependent on the direction that seeds are being sown

The example outlined in section 2A for a potential unintentional use of SOLID is not related to these change cases so had no effect on the modifiability of the design.

B. Assignment 5

To allow for a robot player, one line will need to be changed to specify a different class that Implements KalahPlayerInput. The new class would have the board as input (which is available to clients of KalahInputPlayer) and would override the getPlayerInput() method to determine the player's next house selection which could be difficult (potentially could involve hundreds of lines) depending on how complex the robot players logic is.

Changing the number of houses remains the same as in the previous implementation so involves a change to one line of code and is very simple.

In order to allow for the direction to be change, the developer will need to change the input to the GameInstance class (one line) to a new class which implements KalahTraversable. The interface defines one method which takes in information about the current traversal state of the board while distributing seeds and returns the next traversal state. This is a relatively simple addition to the code in the range of approximately twenty lines of code (assuming the implementation will be similar to the anticlockwise traversal variant).

The SOLID principles applied, that is SRP and DIP improved modifiability in both the first and third change cases from the assignment 4 implementation. Separating the responsibilities of getting user input, and getting the next traversal state into their own classes reduces the amount of cognitive load on the developer as they have less to think about (when defining a new class). Defining these classes with interfaces allows for classes that implement the interfaces to be easily swapped out. Existing clients of the interfaces (not including the single lines where the concrete definitions are created) don't have to be changed. This is less error prone as context reuse is being allowed through polymorphism. changes to existing code is therefore minimised improving modifiability.

C. Implementation 1003

To effect the first change case, the prompt method will have to be re-factored out of the Display class and the IDisplay interface (approx. five lines) into a separate class which manages getting player input. This would mean creating both a human player and a robot player class that implement an interface that defines the method for getting player input. Once this is done the Kalah class will have to be re-factored to change the way that the input variable is assigned. This would involve creating the two players and synchronising the current player in the Game class with a new current player variable in the Kalah class. This is in the range of 20 lines of code to be changed and is non-trivial in terms of difficulty especially

considering how complex the robot player has the potential to be.

To effect the second change case a single static variable will need to be changed in the Kalah class (one line of code) and is very simple.

To change the direction that seeds are sown, a single static variable ANTICLOCKWISE will need to be changed (one line of code). This is a straight forward change.

The modifiability was not affected by the use of SOLID principles in this implementation as it could not be seen in the areas that the changes would need to be implemented nor in the areas in the class hierarchy affected by the change.

D. Implementation 1024

To add a robot player, a new class will need to be created extending the Player abstract class and one line will need to be changed to specify the new class in the setup method in the Kalah class. The difficulty in making this change will be in deciding what logic will be used to determine the result of the getChoice method defined by the interface.

In order to effect the second change case, a single static variable on one line will need to be changed in the KalahBoard class. This is a straightforward change

To effect the third change case, while loop defined in sownOnly() method defined in the Kalah class will have to be rewritten. Assuming the logic is fairly similar this change would involve refactoring approx. 40 lines of code and will be fairly simple assuming the logic is similar to the current implementation.

The SOLID principles had an effect only in the first change case as DIP allowed for the setupGame method to be re-factored to use a different implementation of Player without affecting any of the logic in the class. Context reuse is being allowed by dependency inversion meaning less chance for error to occur when refactoring to include the new robot player. The lines of code being changed is therefore minimised to only the essential logic improving modifiability

E. Implementation 1037

In order to effect the first change the parseInput method would need to be re-factored to consider a new robot player object. This object would have a method within it that will be used to assign the player input variable in its client. The use of this class will need to be synchronised with whose-ever turn it currently is (this can be done with the getCurrent() method available), and will need access to both player's seed containers also available with getCurrent(). This will involve in the range of approximately 20 lines of code to re-factor the ParseInput() and Idle() methods to consider the new player. Making this change will be fairly difficult due to the code that needs to be re-factored potentially introducing faults, the potential complexity of the new robot player, and the difficulty with understanding the code.

In order to effect the second change case a single variable will need to be changed in the PlayBoard class (one line). This change is trivial in terms of difficulty.

In order to change the the direction that the seeds are sown, the LinearSelector class will have to be refactored to determine the next index in the myCollections variable which manages Houses and Stores for a players in the game. This would involve changing approx. 5 lines of code and is fairly simple. However it should be noted that to come to this conclusion, a lot of difficulty went into understanding how the system worked so that is where any extra difficulty would come from.

None of the SOLID principles applied had any affect on the modifiability of the code for this implementation as changes involved refactoring existing code. It could however be argued that if LinearSelector is in fact an example of SRP, then it would've impacted the third change case the code being changed only involves one responsibility. It is easier to think about and is therefore more modifiable

F. Summary of Modifiability

The most modifiable out of all the implementations is the Assignment 5 implementation. Minimal change needs to be made to existing elements and the elements added were minimal (21 + lines of code for robot player implementation). Furthermore, the changes were easy to make as they all focused on a single responsibility.

The rest of the implementations have problems with regards to modifiability. Some involve potentially greater than a hundred lines not including the robot player logic (assignment 4) and others are extremely difficult to understand (1037).

It is hard to say which is the least modifiable as they all have issues with regards to the certain change cases however if the number of distinct elements definition is followed, then the least modifiable would be the Assignment 4 implementation. It had the most distinct elements involved out of all the implementations. However this doesn't consider how difficult it would be to actually implement the changes.

V. Conclusion

Applying the SOLID principles in a general manner has the potential to impact the modifiability positively as seen in the analysis, however the SOLID principles are most effective at improving modifiability when the change cases are considered when applying them.