Design For Modularity In Kalah

SOFTENG 701
Advance Software Engineering
Development methods
Wong Chong (wcho400)
wcho400@aucklanduni.ac.nz

I. Design Considerations

A. Interface Usage

I have designed *Kalah* such that all non-singleton concrete classes to only communicate with interfaces. As seen in the following figure (figure 1), each concrete class implements the associated interface with the same name, with an 'I' to indicate the class being an interface. By communicating with interfaces, each concrete class can be assured of the effects of whatever method they have called, as it follows a known contract. Because of the usage of interfaces, it is also easy to swap out a class with another which also implements the interface. This provides the additional benefit of making the system much more modular and modifiable, as changes to reflect new design decisions can be done on the project.

If interfaces were not used, this would require each concrete class to communicate with each other. Although functionally, it would not show any differences, it makes the project more coupled. Furthermore, were any classes were needed to be changed, the assurance of the usage of each function is lost. This reduces the ease of modifiability as each class becomes less modular.

Although it could be argued that there should be an interface controller between the *Kalah* and *Board class*, I had not decided to do so. I had considered the only value in such an implement to be to allow variables to be modified. However, as Kalah is simply my implementation of the game, it would be more appropriate for such a interface controller to act upon *Kalah* itself. In such a case, I would create a *IKalah* interface for *Kalah to implement* with a method such as:

IKalah makeKalah(int players, int stalls, int seeds);

This would then allow the user to indicate the variables they would like to change, were they choose to modify the starting conditions of the game.

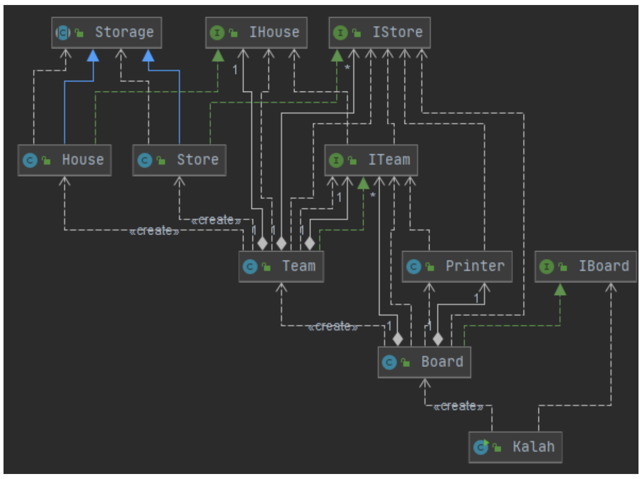

Figure 1: UML diagram of Kalah

B. Object Ownership

Each class is designed to only communicate with objects found within itself. By grouping each object with smaller components, any possible modifications on each class will only possibly affect the classes in one direction. As seen on the previous figure (figure 1), dependencies only appear in one direction. By this design, each class does not have cyclic dependencies. This further reduces the chances of having large amounts of coupling, as enforcing object ownership stops the object from depending on external classes. This limits the scope of checking which would be required to ensure a proper implementation is working. This in turn reduces the amount of coupling, as the scope of checks is reduced. Similar to the benefits of using an interface, having small amount of unidirectional dependencies allows classes to be swapped out easier, as the class itself only actually is dependent on another, while the classes it depends on simple extend the functionality of the parent class.

If these objects were not grouped into its associated groups and instead was all instantiated as objects in *Board*, it is likely to result in cyclic dependencies. Without any design enforcing restrictions on what a class can interact with, future modifications will likely result in classes having more dependencies and inadvertently create cyclic dependencies, resulting in stronger coupling of classes.

C. Printer Singleton

I had decided to create the singleton class *Printer* to display the output of the game. This decision was because the *Printer* class itself does not perform any other functionality apart from formatting. There is little risk for little reward for using some form of interface to join a non-singleton *Printer*. This decision to create a *Printer* module is to centralize the location for formatting the display, which allows an obvious location if a need to change the display arises.

If I had not made *Printer* a singleton, this would have resulted in having printing done on the Board class. This in turn goes against Single Responsibility Principle, for no obvious benefit. In this current hypothetical, we have only considered the possibility of having display functionality in *Board*. There is little reason why future modification would continue this trend, causing this single responsibility to be spread across multiple classes. Furthermore, if I had not made *Printer* a Singleton, it brings along the potential downfall of having multiple instances of *Printer*, where each instance does the exact same thing.