Design decisions for increasing modularity in Kalah

SOFTENG 701
Advance Software Engineering
Development Methods
David Huang (zhua687)

I. USE OF INTERFACES

By requiring classes that call another class to have an interface (i.e. high cohesion classes), they will be guaranteed to have the necessary methods and, as a result, will be bound by a contract. These interfaces are 'IDisplay,' 'IGame' and 'IPlayerProp.' Two other interfaces are made which are 'IGetID' & 'IGetScore' since three of my low cohesive classes share the same methods, the interface can use the classes interchangeably. These shared interfaces therefore increase the modularity of the project.

At first, the project started without any interfaces. Although there were no changes to the project's functionalities when interfaces were introduced, this would mean the classes will not be insured they have the needed methods and shared methods would not be communicated with shared interfaces. Furthermore, errors in one class will cause errors in other classes due to the dependencies between classes. Overall, having interfaces increases the modularity of the project.

II. NON-PASSING CLASS OBJECTS

The objects created by each class will not be passed around and is only modified within the class itself. By making each class object's non-passing, the modules will be loosely coupled.

Initially, the objects created in the 'PlayerProp' class is passed to the 'Game' class and to the 'Display' class and then back to 'PlayerProp' class. A bidirectional relationship is created between the classes as a result where although the objects are not modified by other classes, it enables the classes to have the ability to modify the passing objects which makes the modules more dependent on each other. Furthermore, due to the nature of the bidirectional relationship, circular dependency is created between the classes which causes high coupling as classes must be recompiled every time they are modified. In addition, it prevents static linking since 'Game' class cannot work without its dependence on 'PlayerProp'. Finally, the bidirectional nature of the code makes it more confusing and difficult to interpret its functions. This becomes a bigger issue if the project is modified in the future, as the modules will most likely become more interdependent where locating bugs will be a major challenge. Overall, preventing objects being passed around increases the modularity of the code and makes further modifications less complicated.

III. CLASS HAVE STATED PURPOSE

Each class in the project has a specified goal and will only carry out that purpose. For example, the main class called 'Game' deals with the game's progression. The 'PlayerProp' class deals with the user's properties such as moving the player. Finally, the 'Display' class will only print the output of the game and is the only class that io (from kalah class) is passed into. This prevents the io from being moved around from class to class because it should only be used by the class that does the printing.

Initially, I did not have a 'Display' class, so all the printing was done in the 'Game' class instead. The problem with this was it was harder to specify the goals of the class which makes the modules less modular as a result. Additionally, finding the location of the print statements is easier when the file name is called 'Display.'

It could be argued that adding new print statements is more convenient since only one line of code is added to the 'Game' class rather than adding additional methods to the 'Display' class and then calling it from the 'Game' class. However, due to blocks of code performing tasks other than printing in the 'Game' class, locating a printing error will be far more difficult due to less modular code. Overall, by having each class have a specific goal, the project will become more modular and increase the ease of understanding the characteristics of the modules.