Peer Review of Chans

Group GoAyo

Design principles and implementation

A notable design principle this codebase follows is the Single Responsibility Principle. The model package has a good separation for conceptual responsibilities. It is clear that they have thought thoroughly about their model classes' purposes and responsibilities. However, the controller classes hold multiple responsibilities by being both the view and controller which fails to achieve SRP, more on this later.

It is however a bit concerning that some classes have enormous if-statements in their methods. This breaks against OCP since it would not be possible to add more features without having to modify the existing condition checks. It is also worthy of mention that the model package does not use any polymorphism which might explain the usages of giant if-statements since it disables their ability to use a behavioural pattern like the State pattern. This topic will be touched on later.

The lack of abstraction and giant if-statements make the codebase difficult to maintain and require extra effort to remove/add functionality. For instance, if a new game phase were to be added then all the conditionals that check the phases would have to be modified which usually is not the ideal way to implement a feature.

Code documentation, readability, and naming

Most of the public classes in the model are well-documented, however, Java-documentation is not necessary for private methods as they aren't accessible for the end user. For instance, *resetSelectedSpace*() is missing Java documentation, even if the method name might be explicit. There should also be some comments inside the very large methods (see figure 1, appendix) that explain the more obscure codes.

Regarding the subject of coding style, there seem to be some inconsistencies. For instance, the indentation of curly brackets should be following the same style, even if both styles are valid. Another example would be the abundance of new lines that are superfluous. Lastly, most methods and classes have appropriate names that are inherently intuitive. However, there are a few variables that are incomprehensible. Particularly the *selectedSpace* and *selectedSpace2* in *Board* class as they aren't distinctly implying their context. Additional documentation would clear this up.

Code modularity, structure, dependencies, and abstractions

In terms of modularity, the model is completely independent of other packages, furthering the portability and re-deployability to other platforms. This is one of the most important rules with MVC. However, there are parts of the pattern implementation that could be looked over and improved. The controller concept is coupled in with the view concept in a way that seems unnatural. Each controller is also an anchor pane and acts as a view more or less, whilst also containing logic for handling different interactions with itself. The expected approach would be to have the logic in the controllers, perhaps as listeners to view elements placed in the view. More of this in the improvements section of this document.

The model has very few outgoing, non-Java-Class-Library dependencies. This, as previously mentioned, can make the model re-deployable to different platforms. The one problem, however, is that the *Player* and the *Board* class is dependent on the JavaFX library. While the model is still re-

deployable to other platforms supporting JavaFX, it's not at all reusable in environments where JavaFX is inapplicable. The developer would first have to handle the colour issue and modify the model to make use of it.

Little can be said about the use of abstractions, as there is no inheritance in the model. The model uses collection classes to handle sets of entities, and the highest abstraction applicable is always used, which is very good. I.e. the *List* type is used instead of the concrete *ArrayList*.

Testing, performance, and security

A few classes are thoroughly tested, albeit some other test classes are empty. A good way to ensure the appropriate classes and their functionality is being tested thoroughly is to make sure the test coverage includes the critical functionality in the model package.

There are some complications in running the game on Windows computers with a display scaling set above 100%. This could perhaps be fixed by using JavaFX's built-in methods *setMaximized()* or *setFullScreen()*. Other than that, the application runs well.

Most classes in the model are package-private but there are some inconsistencies (*Player* and *Space*), which perhaps should be set to package-private. Although these are minor concerns, they yield substantial security and assure that no other packages are dependent on the internal functionality of the model package. Consider adding an interface for accessing internal model components to allow change of the actual implementation without affecting the other components' dependencies.

Attributes and corresponding setters are marked private or package-private which is great. However, in some instances, the getters are returning direct references to attributes which may result in alias problems where the private-marked attribute is circumvented by the public getter. Consideration for defensive copying is highly relevant as well as a general notion for immutability.

Suggested improvements

A simple improvement to apply to the controller and view package would be to split every controller up in two different classes. One class would have the JavaFX components and the other class would have the responsibility of controlling these components. The classes would be put in the respective packages' view and controller. The relationship could then be such that a controller has initialised the components in the view class and adds listeners to the components. The controller would then call the model instance and execute various commands depending on which action occurred in the view. See figure 2 in the appendix for a concrete example.

Another potential improvement is to, instead of the enum *Phase*, utilize a State pattern where each state class corresponds to the current phases, e.g. *AttackState*, *DefenceState*. The *Round* class would have an attribute of a general state interface the aforementioned classes implements. The current *startPhase*(...)-method would be replaced with a general method that delegates through the attribute. This would reduce the currently massive conditionals and instead use polymorphism through a general interface. Furthermore, this allows for easy extension of the current phases, as well as the implementation of new phases of the round.

Appendix:

```
boolean startPhase(Space selectedSpace, Space selectedSpace2, Player currentPlayer, int units)
{
    if(selectedSpace != null)
    {
        if(currentPhase == Phase.DEPLOY)
        {
            return Deployment.startDeployment(selectedSpace, currentPlayer,units);
        }
        else if(selectedSpace2 != null){
            switch (currentPhase)
            {
                case ATTACK:
                    if(Attack.DeclareAttack(selectedSpace, selectedSpace2, selectedSpace.getUnits()) ) {
                        dices = Attack.calculateAttack(selectedSpace, selectedSpace2);
                        return true;
                    }
                    return false;
                case MOVE:
                    return Movement.MoveUnits(selectedSpace,selectedSpace2, units);
                default:
                    return false;
            }
        }
    }
    return false;
}
```

Figure 1: Giant if-statements makes it difficult to maintain.

```
public class ViewClass extends AnchorPane {
    Button button;

    public ViewClass() { button = new Button(); }

    //No logic what so ever. Just a dumb view.
}
```

```
public class ControllerClass {
    ViewClass viewClass;
    Model model;

    public ControllerClass(ViewClass viewClass){
        this.viewClass = viewClass;
        model = Model.getInstance();
        initViewButtons();
    }

    private void initViewButtons() {
        viewClass.button.setOnAction(new EventHandler<ActionEvent>() {
            @Override
            public void handle(ActionEvent actionEvent) {
                model.shutDown(); // Or some other model command.
            }
        });
    }
}
```

Figure 2: Example of how the relationship between a controller and a view could look like, to separate the responsibilities of the view and the controller.