Peer review of And I OOPP

RAD SDD

The RAD and SSD both look good progression wise. Both documents seem to have been worked on efficiently throughout the project.

The user stories sometimes seem a bit abstract or a bit too concrete and detailed.

- Is the game running? *"As a gamer, I want the game to run, so that I can play it."*

Enormous and abstract, can be used by every project ever and is more or less required to work. Maybe change it to something like "is the game window generating properly?"

- Rules. "*As a gamer, I want the game to have rules, so that I can follow them and learn the game.*"

Written as something that should be worked through beforehand not something that will be worked on during the project. Vague, unclear if there is a need for tasks. It seems like a task. Maybe change it to something like "is the game implementing the *described* game rules?"

- Attacks. "*As a gamer I don't want Mario to be able to attack enemies all the time and over the whole playing field*."

Is poorly worded and hard to understand. Specifies a specific tower and gives it specific rules instead of explaining how towers in general are supposed to work. Can be changed to describe all towers, "I want the towers to have specific, reasonable and (maybe) unique effects".

Apart from the user stories the documents are well structured and written. It's easy to understand and quickly get a grasp of what kind of project that has been developed. It describes the game well, which means that even if you're unfamiliar with these kinds of games you will most likely understand the purpose of it.

Code

Do the design and implementation follow design principles?

Does the project use a consistent coding style?

Looking through most classes it becomes clear that all the programmers have assorted to using a similar coding style. It is difficult if not impossible to find code that seems out of place or weird compared to the rest. Names are coherent, classes are compact and small and most if not all code is smell free.

Is the code reusable?

It's reusable due to high abstractions implemented with interfaces and generic typing. The classes are small and modular, which satisfies the Single Responsibility Pattern, as well as the Interface Segregation Principle. Some examples of this include the view object classes which implements a generic rendering method. The code does rarely depend on concrete implementations (if, switches etc).

Is it easy to maintain?

Due to the previously mentioned, the abstract code makes it easy to maintain. Most changes will only affect a small piece of the code and as it almost has no concrete methods it allows changes to be made without requiring to change methods, only parameters.

Can we easily add/remove functionality?

Again because of the abstract codebase, it would be easy to add functionality to the application. As an example, the view interface makes it easy to add another visual part to the program with whatever part of the model that would be wanted. It is also possible to change the game's more concrete parts, like the amount of lanes. This would be done in the Model class and would also scale the towers. The only problem is that the enemies don't exist automatically on the lanes that are added, but this could be implemented later.

Are design patterns used?

- Quite a few design patterns are implemented. These are some examples:
    - Facade pattern is used when creating the world, which abstracts the implementation.
    - Observer pattern is used at DragAndDropService, MouseInputService etc.
    - MVC pattern is used for the overall structure. Here the view does not change the model, the controllers do that and listens to the views for input. The model also does not know about the other components in the MVC. This is a good design from a OO perspective which, as previously mentioned, makes the design extensible and modular.
    - Factory pattern is used when creating sprites, lanes etc.

Is the code documented?

The documentation of the code is not consistent and is only present in some files. Of those files only some have complete and clear documentation with explanations and instructions. This should be implemented more.

Are proper names used?

The names are specific and descriptive about their usage/task.

Is the design modular? Are there any unnecessary dependencies?

Overall the code is modular and does not use unnecessary dependencies. It’s so modular that the underlying display framework JavaFX could be changed to another without changing the view.

Does the code use proper abstractions?

The code seems to follow the Dependency Inversion Principle fairly well and depend on abstractions rather than the details.

Is the code well tested?

The code has some testing, but only for some classes. The only class that is well tested is ViewPort. World has some testing but could probably have some more and the rest of the classes are not being tested at all.

Are there any performance issues?

The game runs fine on both mac and windows. Nothing to complain about.

Is the code easy to understand?

The code is clean and well structured, however it takes a while to understand the “full picture” of the code. A more detailed Class diagram or more code documentation would have been helpful. There are a lot of classes and understanding each and everyone's purpose is difficult without proper documentation. Making it so that a new person has to take a long time to first “dissect” the code to get to grips with its purpose.
A good improvement would be to add a flowchart that describes how the programs make the game run, which classes and methods need to be called first and what effect they have on the game window. Do they spawn the map, set up waves etc. It makes it easier to move around the code beginning to end and understand what is essential for the game to run.

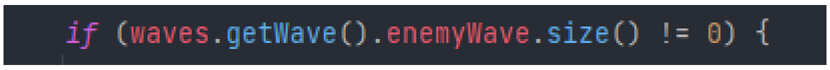
Does it have an MVC structure, and is the model isolated from the other parts?

There is a clear MVC structure as described in the SDD. The model is isolated and does not communicate with the other parts of the MVC. The controller controls the model and gets input from the view component.

Can the design or code be improved? Are there better solutions?

First and foremost the overall structure of the codebase is clean and well thought-out, but it could be confusing to understand, as stated before. Therefore documentation is one area that could be improved a lot. Considering the simplicity of the game, the overall structure could also be considered overworked since it is a bit complicated, but that is obviously a subjective point. Testing is another point that could be improved, especially since such a big chunk of the code is complete. This is especially true considering the vertical slice method that has been presented during the course.

There is also a bit of method-chaining present, which could violate the law of Demeter. As an example, the Model class's update method does chain methods:



```
    if (waves.getWave().enemyWave.size() != 0) {
```

In the same class there is also a createWorld method that sets the map, more specifically the lanes and cells which defines the map. It probably makes more sense that these values, which define a map, could be set higher up in the hierarchy. This is especially true if the game should be extensible and include multiple maps.

When using PMD to review the code there are some minor problems. Imports that are not used and many instances where ArrayList is used where List had sufficed. Maybe not vital to the success of the project but could be a nice clean up.

Summary

The project is overall well done and adapts well to the course assignment of object oriented programming. The code is extensible and abstract which makes it easy to add other modules to it. The code is not easy to understand at first hindsight and takes some time to get a good grasp over, but once you've managed that it seems fairly easy to maintain the code even for someone who has not been part of the project from the start.

The user stories could use some work, some of them seem too abstract and others are too narrow. Some user stories are obvious and redundant, one could argue that they shouldn't be part of the user stories at all.
The testing and documentation needs more work but are on the right path.

Well done And I OOP, keep up the good work!