Group:17E
Date: 2021-10-17

Peer review

MVC Pattern

The code is structured around the Model-View-Control design pattern. Each part of the MVC is divided into its own package. The model is well isolated and through the use of interfaces it is easy to grant access only to the necessary parts of the model. A drawback of the current design is that the model exposes many important direct references, creating a risk for aliasing issues. This could be improved by re-designing getters in the Model class to pass copies of objects instead of direct references. The controller part is certainly thin as it chains calls to the model and does basically nothing else. As for the view, it contains some logic, however since it is all related to rendering, it's fine. The view classes are large, however that is inevitable since all GUI objects are created in code.

Path Class

- Abstract class path, good for extendability.
- Method createMapCollision in Path very big, use functional decomposition. Currently it is very hard to grasp.
- Usage of the Factory pattern in Path Factory, good for extensibility and easy to add new paths if needed in the future.

Projectiles

Abstract class projectiles consist of a lot of instance variables that only has getters but no setters, variables should be set to final if they are not meant to be changed once they are created.

Projectiles has a boolean variable “canRemove” that only gets used when projectiles are added to a list with objects that should be removed from the map. Instead of setting the variable to “true” the method could chain the object directly to the list to skip the boolean.

Towers

Method for changing target modes is confusing and not extensible. Would make more sense to use a target mode as a parameter. Solution would also make choosing target mode easier in game, with a dedicated button for each mode instead of buttons to scroll through the modes. Target mode could also follow the state-pattern to simplify its usage and changes.

Another solution to changing target mode would be to implement the state pattern and let each tower have a state and let each target mode be a different state. Implement it in the way

that each state knows what comes before and after and each time a new state is needed simply create a new instance of the state.

Stats for towers are set in factory class, which might be more confusing than having them be set in the actual class for each tower.

Spawning Viruses

The matrix “SpawnInfo” that contains the spawn information for viruses in each level is “hard coded” and therefore not very reusable. Good practice would probably be some form of algorithm to calculate what/when to spawn viruses.

Utilities

The Utilities package is located outside the model package although some classes in the utilities package are not just “helper classes” but classes on their own such as GameTimer. Generally if you want to have utility classes they should be stateless and only consist of static methods, a good example is their Calculate class which is simply a helper class to the model. that they should consider being in the model. Suggestion is to move the instantiable classes such as GameTimer to a part of the model since it is just not a “helper class” but an important part of the application.

Unit tests

The model has great coverage in tests, and the tests connected to it are well made. There are also tests connected to the view which isn’t optimal for these types of tests and when they are run some errors are created but the tests pass non the less.

Coding style and documentation

The code is written with a consistent style and each class is well structured with related fields mostly placed close together, improving readability. Variables, methods and classes have names that make sense. Most of the code is covered by JavaDocs, and the JavaDocs comments are generally well written. However, the Model class, which has a lot of important public methods, completely lacks documentation.

Structure and inheritance

Inheritance is well used throughout the code, which leads to minimal code duplication. Both interfaces and abstract classes contribute to making extending the program easy, improving code reuse and creating good encapsulation. Over all, the code is well encapsulated through the use of private/package private fields, and dependencies are mostly abstract thanks to interfaces and abstract classes.

Group:17E
Date: 2021-10-17

Platform independence

We were not able to run the application on both Linux(Arch btw) and Mac following the specified instructions. This might imply that there are other steps necessary to running the application on different platforms and instructions for these steps were missing.