Code Evaluation for an Asteroids Game clone

1- Code quality:

- **Class Intelligence distribution and single responsibility design:** Class hierarchy is generally good. Most classes follow the single responsibility design with except to the Game Controller class. The game controller class is the largest class in this project. It includes tasks that should be moved to other classes e.g. spawning power-ups should be moved to the powerup class, asteroid management should be moved to the asteroid class.The coders note the need to do some of these changes in commented TODOs in the code, so they appear to be aware of that. In addition, objects of this class can be accused of being God objects as the intelligence of the game is somehow centered inside them. As an observation, the class size is more than double of the second largest class, 4 times larger than all remaining classes, imports almost all other classes and manages objects of them. In conclusion, intelligence (which indirectly reflects responsibilities) must be moved to other classes to allow for a more equal distribution.
- **Class packages:** one of the things that can be quickly noticed after viewing the project is that the name of some packages was actually the name of the implemented design patterns, which is a quite bad practice. In addition, as some package names were incorrect, classes under those same packages weren't homogeneous in the sense that allows them to be in the same package e.g. under ObserverPattern package, some classes were related to behaviour (power up classes) while others were related to objects (asteroids, bullet, enemy classes). The package structure should be edited and improved.
- **Class structure:** The general structure appears to be very good. Interfaces were used to model changing behaviours (shooting, power-up, and movement behaviours). An abstract GameObject class was used to create a universal abstraction for all game objects (objects that appear on the screen during the game play). This was then extended to create individual abstract classes for each family of game objects (e.g. Asteroid, Enemy, PowerUp), or to create normal classes (e.g. SpaceShip). The former were finally extended to create more individual game objects (e.g. big/small asteroids, normal/bos enemies). This is good when it packs things shared with all objects (like rendering), but here, some methods were included in the original GameObject class although they are intended to be used only by some inheritors (e.g. SlowDown only for the SpaceShip class). These methods express specific behaviours, and are better to be moved so they only exist in their related classes. Aside from that, other structure-related things are in good order.
- **Implementation and use of Design Patterns:** From a high level perspective, 4 design patterns appear to be used in this project. Namely: Strategy, Iterator, Factory, and Observer.
  - Strategy is used to interchange encapsulated spaceship power up behaviour. Specifically, change through Speed powerup and NoPowerUp at runtime.
  - Iterator was used to store all instances of game objects to iterate through it. However, even after 30 minutes of manual and automated search throughout the code to understand where the iterator has been used, no results were found, so it appears to be structured but not used (e.g. no usage of the .hasNext() method anywhere).
  - Factory design pattern was used to spawn objects like the enemies, bullets, asteroids, and so on.
  - Observer design pattern was implemented having the spaceship as a subject and enemy entities as observers. The notification happens when a spaceship moves, so all enemy entities get to know the new location of the spaceship.

  In general, use of design patterns is good with the exception of Iterator, that seems written in the code but not found to be used (although a really really deep search was done). Up to this point, 4 of the 7 design patterns learned in lectures DP1 and DP2 must be available in the code. However, only three of those exists (no clues on the use of Singleton, Adapted, Composite or Decorator). I have no clue on the existence of the 4th required one.
- **General code analysis:** IntelliJ Analyze tools show no cyclic dependencies within this project. Extra code inspection was done using the Code Inspection tool under the IntelliJ set of Analyze tools. All results only contained warnings to optional improvements e.g. fields that can be local, unnecessary modifiers, etc.. Aside from these trivial things, everything looks good.

Reflection on the code quality: All negative issues related to code quality were discussed above. Based on this, this project would score 7 or 6 on the rubric because of the existence of lower quality parts.

2- Code Readability:

- **General:** It appears to be quite clean, intact, easy to read, and organized. No commented code exists within the project.
- **Formatting:**
  Coders were mostly consistent in their writing e.g. all functions that involve one line of code were written in one line only, all longer functions were written as a chunk of code. Some small consistency violations involve the existence of empty lines of code. It is true that placing some empty lines in some places would improve readability, but placement criteria should be consistent e.g. it appears that one team member favors putting an empty line before and after an *if* statement, while others don't. Almost everything follows the most popular Java conventions [1, 2] with the exception of things previously mentioned. All functions were in an easy-to-chunk size.
- **Naming:** All observed names reflects the objects related to them correctly. In addition, naming follows the java language conventions [1, 2]. However, a small confusion may be caused with class naming. There exists 2 classes with almost identical names: GameObject, which is an abstract class (explained earlier), and GameObject**s**, which implements the Iterable interface. Renaming would solve possible confusions.
- **Comments:** Some parts are very well commented (especially large vital classes), other parts are poorly commented, or include only auto-generated JavaDoc that aside from specifying parameters and returns, doesn't really reflect on the task of that specific part of code. Good naming and organization compensates this. However, this aspect can should improved.

Reflection on the code readability: This project scores high in readability, with a score of 8. Adding more comments would raise this score more.

3- Testing:

- **Evaluating written tests:** All written tests are meaningful. 295 total tests were run, and they all build run successfully. It looks like the project members are trying to apply parallel inheritance to the test code, which would be quite effective especially considering the class structure of their code. Many commented test code lines exists within the tests. A big downside to all test classes is that they almost entirely lack meaningful comments, so understanding the goals of the tests was quite challenging.

Reflection on testing: Since no coverage analysis tool was used included with the given code, the test code coverage couldn't be determined. However, it looks in the range of 30-40%, which is not sufficient according to the rubric. Score for this section would be 4.

4- Tooling: Not enough file were provided with the source code to judge tooling; we assume its evaluation is not required.

References:
[1] Google's Java style Guide: https://google.github.io/styleguide/javaguide.html#s4-formatting
[2] Oracle's Java Style Guide: http://www.oracle.com/technetwork/java/codeconventions-150003.pdf