Board Game Area - Game Planner

This assignment is focused on allowing the client build a list of games they want to try out on Board Game Arena. Most of the application was already built by the previous intern who left the company, and you are being asked to finish the application. In the end you will have a working program that can help you figure out what board games to play on Board Game Arena.

Learning Objectives

- Designing with inheritance and polymorphism in java
- Practicing Test Driven Development
- Implementing junit tests for all methods
- Making use of java collections/lists
- Making use of equality and sorting (especially sorting)
- Explore the use of streams in Java

Instructions

For this assignment, you will be implementing the IGameList.java interface. This interface is used to represent a list of games that a user wants to play on Board Game Arena. You will also be implementing the IPlanner.java interface which is used to help filter the BGArena games into smaller lists, so they can be added to the GameList. Both of them have classes already associated with them, GameList.java and Planner.java respectively. This was to make sure you used the correct constructors.

> Important

> Just because we provided the base class does NOT mean all the functions you need are there. Just the public ones that are part of the interface. In our solution, we added multiple additional classes for this program as support / utility classes, and multiple private methods to each class!
>
> You **cannot** modify the interfaces IGameList or IPlanner or the constructor types for GameList or Planner. Everything else you can modify as needed. The autograder unit tests will be based on GameList and Planner, thus the reason you cannot modify them.

Your goal is to implement those two interfaces, so you have a working program. You will find that you will need to additional classes to make your life simpler. While you are only implementing a small portion of this program, that focuses on equality and sorting of data structures - don't let that fool you. It can take a lot to think about the logistics of the ordering, and how to break it down into smaller more manageable parts. Take your time to do that.

We suggest that for Module 05 - you implement the GameList, and some of the Planner (mainly the filter portions), and also finish the technical questions in Report.md. Then in Module 06, you can add the sorts, and probably update GameList now that you have more tools to work with. This will help spread out a deceptively small assignment, as it isn't small at all once you start working on all the options.

> Tip
>
> We have included a sample completed program you can try out. No promise that it is 100% bullet proof, and probably some typos, but you can get the idea of how the final program works. If you go into the sample_working directory, you can run the program with the following command:
>
> ```
> bin/bg_arena_planner
>    or if windows
> bin\bg_arena_planner.bat
> ```
>
> Both commands will execute the .jar file in the lib directory while properly setting the path. For macOS and linux, you may need to add the execute permission to the file (`chmod +x bin/bg_arena_planner`).

🔥 Task 1: Design

Before you start writing, it is important to think about design. You DO NOT have to be perfect in your design, so we will come back to this step a few times.

1. First, become a detective and read through the files provided. Take notes on what you are seeing. This is a common skill in software engineering, and you will need to do this often as you work with other people's code. Report.md has specific questions on the code that may help you.

2. Go to DesignDocument.md and fill out (ONLY) the initial design sections. We have broken them up into sections to help you think about the design in smaller parts.

> Tip
>
> You are free to use mermaid or any other UML tools you want, just make sure if you are using another UML tool, you include the image in your submission. For a reminder on how to use mermaid markdown, look here.

🔥 Task 2: Implement by Test Driven Development

After your initial design, you should seek to follow the TDD process. This means you should write tests first, and then implement the code to pass those tests. Or better stated, you should write *ONE* test first, implement, and repeat until you have written all your tests.

1. Figure out a number of tests by brainstorming (done in design)
2. Write **one** test
3. Write **just enough** code to make that test pass
4. Refactor/update as you go along
5. Repeat steps 2-4 until you have all the tests passing/fully built program

Note: you often don't know all the tests as you write. As such, it is alright to continue to expand your list. This is where people get stuck on TDD. They think they have to know **all** the tests before they start. You don't. You just need to know the next test, and then at the end you double check you have covered all code paths and have full coverage.

> Caution
>
> While we may not be using Github Classroom, you may still be choosing to use a github repository. If so, make sure to commit as you development. The bare minimum commits would be after every test, but you probably will have additional commits especially at the beginning.

🙋 Implementation Tips

- Make sure to focus on how to keep things generalized, and broken up into other classes.
    - For example, we had a Filter.class that had stringFilter and numberFilter - which handled multiple types of filters (name, max/min players and time, rating and difficulty, etc).
        - Switch statements are really helpful inside the filters based on the Operation.
- We found it useful to create an Operator Enum similar to the GameData or ConsoleText enum. We added an additional useful function to help with filtering, which is as follows:

```
public static Operations getOperatorFromStr(String str) {
     if (str.contains(">=")) {
         return Operations.GREATER_THAN_EQUALS;
     } else if (str.contains("<=")) {
         return Operations.LESS_THAN_EQUALS;
     } else if (str.contains(">")) ...
     // rest omitted for brevity - but helps drive the point
```

By using this enum, we never had to type ">=" or "<=" in our code, and could just use the enum. This made it easier to change the operator if needed, but also made it easier to check the filter and run various methods based on the operator returned. It would be possible to accomplish the same thing with a number of `public static final String` variables, and a few other supporting methods. The important part is try to break up your code.

- A sorts strategy similar to the one presented at the end of Module 06 team activity is **important!** It made it easier to handle all the sorting options.
- Make sure you follow TDD. There area lot of paths, but you can continue to add to your code as you develop tests - this especially helps in parsing the strings
  - For example, you have a test for a filter - you write the string parser just for that test
  - You have a test that handles multiple filters, you add to the string parser to handle multiple filters
  - Then you have a test for a sort, you add to the string parser to handle the sorts
  - Without that, the struggle of trying to visualize all the conditions gets very overwhelming!

> ⚠ Warning
>
> If you modify one of our files, you need to add tests for it. If you do not modify a provided file, you don't need to add tests. This can be done without modifying our provided code, but you are welcome to modify it (except for the caveat about IGameList and IPlanner) if it fits your overall design better.

🔥 Task 3: Finish Design Document

By this point, your design has probably changed (very few people have perfect designs the first iteration). Update your design document with the final design in the "final design" section. We want to see the history of your first design to your final design. That is a good thing.

🔥 Task 4: Finish Report.md

Inside of Report.md you will need to answer a series of questions about your program, and about the learning objectives for the module in general. Fill it out.

> ❗ Important

> A primary purpose of this activity is to get you working through a process in addition to writing code. In software engineering the process you follow is often just as important as the code you write. This is because the process is what allows you to work with others, and to be able to maintain and update code over time. It may seem tedious right now, but it is a skill that will pay off in the long run.

Submission

When you are completed, you need to submit your code to gradescope. Go back to Canvas, and click through the link that takes you to the Gradescope assignment. When you submit, you should be able to upload directly from your computer, but I have left github repository uploading available as well should you choose.

📝 Grading Rubric

1. Learning (AG)
    - Code compiles without issue
    - Code passes all tests
2. Approaching (AG)
    - Passes the style check.
3. Meets (MG)
    - README.md is filled out (name, etc)
    - DesignDocument (INITIAL) sections are filled out
    - All methods are tested with JUnit tests
    - Method contain proper javadoc comments (not just javadoc notation but proper wording in the comment)
    - Report.md technical questions are questions answered correctly.
4. Exceeds (MG)
    - Code is DRY (Don't Repeat Yourself)
        - Including making use of helping/utility classes to reduce duplication.
    - Student uses proper inheritance without duplication
    - Methods include tests for edge cases in addition to happy path
    - Proper use of sorts and sort strategy
    - Design document (FINAL) sections are filled out
        - The notation needs to be correct, and the TAs will double check the final design matches the final implementation.
    - Report.md Deeper Thinking question filled out
        - Includes at least two references/citations

Legend:

- AG - Auto-graded
- MG - Manually graded

Submission Reminder 🚨

For manually graded elements, we only guarantee time to submit for a regrade IF you submit by the DUE DATE. Submitting late may mean it isn't possible for the MG to be graded before the AVAILABLE BY DATE, removing any windows for you to resubmit in time. While it will be graded, it is always best to submit by the due date, so you have full opportunity to improve your grade.

If you need a reminder about the grading policy, please review the syllabus and the canvas page on 'formative/summative' grading. This class uses a unique grading system that will allow you to be flexible with due dates and multiple resubmissions (if you submit with time for TAs to give feedback), but we also ask that you continue to work on the assignment until you get a full grade.

Autograder Limitation

Currently the autograder is limited in how it can test. As such, when it comes across an error it just stops. This means that if you have multiple errors in your code, you may only see the first one. You will need to fix the first error, and then rerun the tests to see the next error. Eventually, if every test passes, you will get the single point. It also may give you points for valid style, while errors exist in the code - so really assume the first 2 points are done together.