Group: Group OOD
Group Members: Hee Weng Sheng, Isaac Lee Kian Min

Changes to Engine Package Recommendations

World Class

The World class is by no means a badly designed class, however there are certain methods where overall **readability** can be improved. This is evident in the *processActorTurn* method, where it is considered a bloater as it has an overly **long method**. This would undermine our ability to **understand** the source code, making it difficult for **extensions** or **maintainance**. Besides that, longer methods will ultimately lead to a **larger class**, again repeating this vicious cycle.

With that being said, we should use the concept of extracting bigger methods into smaller, manageable methods, **modularising** it in the process. Here are a few advantages and disadvantages to this suggestion:

Evaluation of extracting bigger methods ito smaller methods:

1) Advantage:
    a) Improves overall **readability** of source code.
    b) With that, we can better **maintain** and **extend** the intended behaviour of the original method (processActorTurn) through its smaller individual methods.
    c) In the event of an inheritance relationship of this class, we can override these smaller methods to implement other behaviours, thus reducing the possibilities of **dependencies**.
2) Disadvantage:
    a) Honestly, we could not seem to identify a disadvantage in modularising code. However, this may indirectly increase complexity of code as it needs to call more methods, but this is almost outweighed by the ease of readability.

Moving on, while there are no problems in terms of correctness with the implementation of the addPlayer method, it would have been better to have it check for a **precondition**. Given the context, it should have checked for a *NullPointerException*, which can potentially cause the program to **crash**. With that, we could have used the *Objects.requireNonNull* static method of the Objects class to execute this check, where it is used in classes such as the Location and GameMap class. Again, here are a few advantages and disadvantages to this suggestion:

Evaluation of checking for preconditions:

1) Advantage:
    a) *NullPointerExceptions* are **unchecked** exceptions thrown during runtime. Doing so allows the program to **Fail Fast** (FF), which helps programmers to **debug** the problem **earlier** on.
    b) **Reuse** of existing implemented methods through the use of the *requireNonNull* method.

2) Disadvantage:
    a) There is already an existing explicit check through the *run* method, which may cause **duplication** of codes.
    b) Might cause a **dependency** relationship with the Objects class.

Menu Class

With reference to the Menu class, the method showMenu is not well implemented as there is a limitation whereby 26 options are only available for the user to choose from. Anything more than that, it will be ignored. If there are more than 26 options, the user may not be able to choose their desired options as there are insufficient characters to display the available options.

Having said that, we suggest providing a submenu if the numbers of options for the user to choose is more than 26. The submenu will have the remaining options that are available for the user.

Evaluation of having a submenu when the numbers of options are more than 26:

1) Advantage
    a) Enables the user to choose every possible option without losing any of the options.
    b) May promote the reuse of code if implemented properly (e.g. provide an option to display the remaining options it there are more than 26 of them)

Engine Package Good Practices

One good practice maintained by code was the check for preconditions. In this case, we mean the checking for a null object, which would cause the compiler to throw a NullPointerException that causes the program to crash. This is evident in both the Location and GameMap classes, where such checks were done through the *Objects.requireNonNull(object)* static method of the Objects class. This practice fulfills the **"Fail Fast"** principle of good design and good coding. This principle shows that the system should fail immediately and visibly when something goes wrong. With that being said, here are the advantages of the "Fail Fast" principle:

Advantages:

1) It is easy to find and fix when errors occurred.
2) Resultant program will be more robust in the end.
3) Faster to stabilize the software.
4) The cost of failure and bugs are greatly reduced.

Besides that, we foresee that all the variable names are named **meaningfully**. The variable is not **stretched** over two jobs (e.g. damage variable of positive value indicates the damage dealt to the player, thus a negative damage should not indicate healing the player). This is also known as "**hybrid coupling**". This shows a good practice of one of the design principles

which is **“Avoid variables with hidden meanings**”. Simply put, a variable should be used for only one purpose, which can be understood better through good naming practices.