Recommendations for change to the game engine

The strengths

The `engine` package maintains abstraction by packaging the engine and game seperately. This ensures that it is maintainable, by grouping related classes together into subsystem of packages. Furthermore, appropriate private/protected visibility modifiers are applied as well for abstraction and encapsulation.

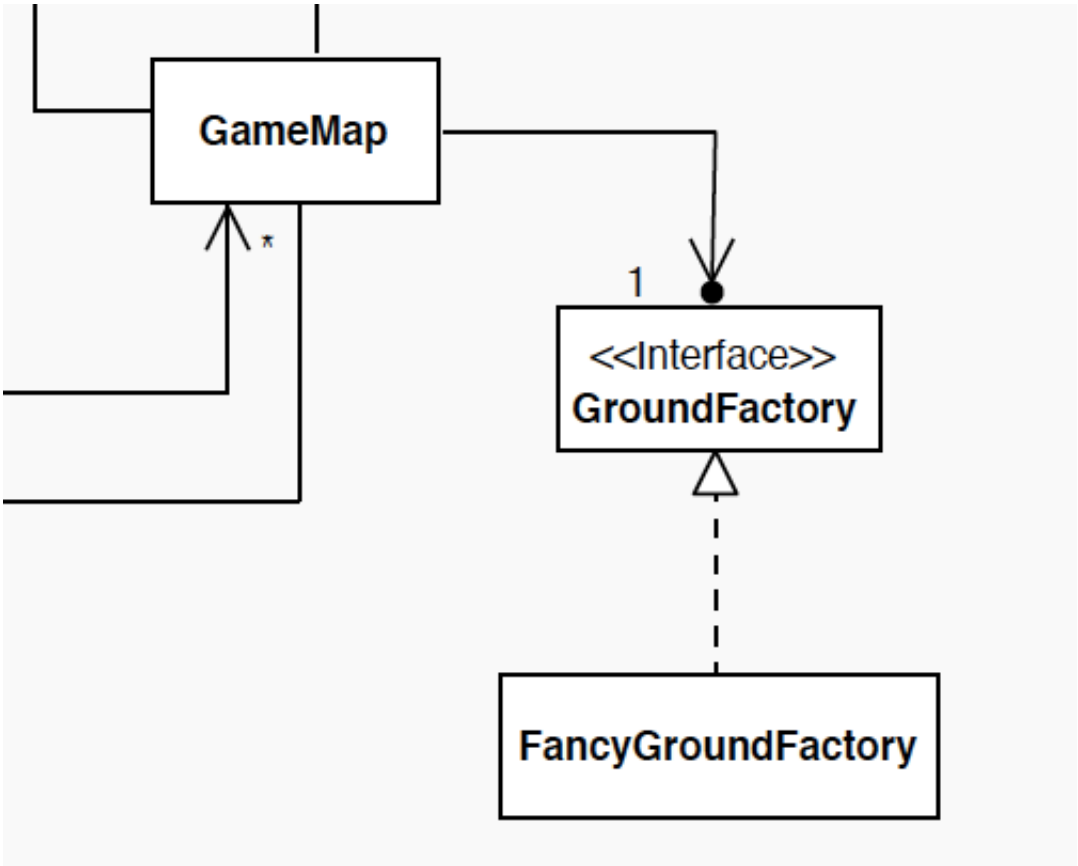
All the classes implementing an interface has usable methods provided by the interface. For instance, the `Action` class has methods which is suitable and appropriate for other action extending that superclass, such as `DoNothingAction`, `MoveActorAction`, `PickUpItemAction` and `DropItemAction` classes. In other words, most of the interfaces in the engine package is suited to the class. Thus, from these observations, the Interface Segregation Principle is applied.

Interfaces such as `Weapon`, `Capable` and `GroundFactory` promotes the Open/Closed Principle, and this introduces an additional level of abstraction which enables loose coupling, where the interfaces are closed for modifications, and open for extension such as its subclasses.

Liskov Substitution Principle is also applied in the relationship between `Action` base class and other action classes such as `DoNothingAction`, `MoveActorAction`, `PickUpItemAction` and `DropItemAction` classes. This means that the subclasses behave the same way as the superclass. For instance, the following statement is valid:

```
Action action = new DoNothingAction();
```

The Dependency Inversion Principle is also applied here, along with hinge points. For example, the relationship between `GameMap`, `GroundFactory` and `FancyGroundFactory` classes, where free changes can be made to `GameMap` and `FancyGroundFactory` provided both respect the interface, therefore giving a hinge design. The following UML diagram from the `engine` class diagram shows this principle being applied:



The code also adheres to the DRY principle, where there is no code duplication, and common functionalities are abstracted into methods.

Documentation is also provided, thus promoting usability to the code.

From observing the getter methods, there is no privacy leaks are detected in the code. All getter methods are appropriately returned a copy of the attributes.

There is also proper error handling, such as within `ActorLocations`, methods such as `add` throws an `IllegalArgumentException` if Actor is already placed or there is already an Actor at the target Location, thus if a precondition is violated (where the contract is breached) then the exception handling will indicate there is a bug. This also adheres to the Fail Fast Principle, where if there is a bug, fail as soon as a problem is detected with a message for debugging.

The flaws

The `World` class lacks getter and setter methods for its attributes, for instance, what if the client wants to get the total maps in the world? The `World` class should provide getter and setter methods so that information hiding is enforced, where class attributes are made inaccessible from the outside and by providing getter and/or setter methods for attributes that shall be readable or updatable by other classes.

However, the flaw of this approach is that maintainability may be reduced, because it exposes the implementation of the class and thus this makes the code harder to maintain. Therefore, only provide getter and setter methods only if neccessary, as too much getters and setters is not advisable. Note that the approach I am providing is used where the getter method is most commonly needed and used.

Another thing with error handling is that a proper message should be provided if a precondition is not met. Again taking the `ActorLocations` example, the `add` method should throw an `IllegalArgumentException` **with** a message that specifies the problem, and currently it does not provide any message. This will help in debugging the problem and finding the root cause. Therefore to fix this, simply add the following messages to the `IllegalArgumentException` argument in `add` method:

```
if(actorToLocation.containsKey(actor))
      throw new IllegalArgumentException("Actor is already placed or there is
already an Actor at the target Location");
    if(locationToActor.containsKey(location))
      throw new IllegalArgumentException("Actor is already placed or there is
already an Actor at the target Location");
```

Another problem with the `engine` package is that some classes are tightly coupled and handling too much responsibilities at once. For example, take the `Location` class. Currently the `Location` class is handling character representation, terrain type, and other game data. Unless they are cohesive, where things related are grouped together, then it violates the Single Responsibility Principle.

A solution is to split the task into classes each handling different responsibilites such as `LocationExit` class to keep track of exits in the location, `LocationTerrain` class to identify the location terrain, and `LocationItem` to identify the items at that particular location. All of these classes can then extend the `Location` class, which is made abstract (It can also be made as an interface, which can be use for decoupling). This approach can lead to loose coupling, which promotes the purpose of single-responsibility and seperation of concerns. However, loose coupling can also bring about inconsistency if there is too much subdivision.