- Exits should keep track of their own name and their own hotkey instead of putting literals in GameMap. (make use of enum class or smthg)

  Information/implementation hiding could improve. For example, the return-type of getAllowableActions in Items should be collections instead of list. This exposes the data structure used.

  ```
  public List<Action> getAllowableActions() {
          return allowableActions.getUnmodifiableActionList();
  }
  ```

Adding more getters with proper encapsulation will make it easier test.

If you can't think of anything bad about the engine, then write a justification of your positive opinion with reference to the design principles you've learned in this unit.

Good point
Engine code overrides a lot. That makes it easier to add on codes.(Maintainability). It also helps reduce repeated codes, DRY

Minimize dependencies that cross encapsulation boundaries
By making method/instance variable private/protected. Looking through all the classes, all the instance variables are either private or protected. This minimizes the dependencies that cross encapsulation boundaries.

Encapsulation to ensure that there are no privacy leak is relatively good. The getAllowableActions in Item returns an Unmodifiable list. This can help avoid privacy leaks.

There are some examples of the code in the engine packages that follows the Declare things in the tightest possible scope principle. For example, createMapFromStrings(GroundFactory groundFactory, List<String> lines) is declared private. This is because this method served no purpose for other classes to use it and should not be open for other classes to use it if they dont have to.

There are also a lot of abstract classes in the engine package. For example, Action is an abstract class. Other classes like PickUpItemAction, DoNothingAction and DropItemAction inherits from the Action and methods like getNextAction() is not present in PickUpItemAction, DoNothingAction and DropItemAction while methods like hotkey() is present in DoNothingAction. This gives flexibility to the other classes whether they want to implement this methods or not. It makes the code more maintainable and flexible. It also follows a very important concept called Don't Repeat Youself(DRY).

Fail Fast. The engine packages follows the principle of Fail Fast makes it easy for us to correct ourselves. For example, ActorLocation class's method, add(Actor actor, Location location) and move(Actor actor, Location newLocation). Sometimes the system will fail because we add player at an invalid location. It makes it super helpful for debugging.

The engine code follows the principle: Avoid variables with hidden meanings. There are barely any of them which makes the code easier to understand and less overwhelming.
Another good thing of the engine code is that it is filled with Java Docs. It shorten the time for us to read the code class by class, just the understand it. It quicken the process a lot and was super helpful.

Based on how my perspective, most if not all of a method is located in the wrong class. This is very important as it can reduce unnecessary connascence.

Another way the engine code reduces connascence, in this case, connascence of execution is by throwing exception. For example, in the World class, if player == null, throw exception.

The interfaces(GroundFactory,printable,Weapon,Capable) are also relatively small. It is good as doesn't force us to implement all of the methods.