RECOMMENDATION / JUSTIFICATION FOR GAME ENGINE

Issue 1: Privacy leak caused by `map` getter of Location class:

```
public GameMap map() {
            return map; }
```

The current getter returns a reference to the Location's *map* with no prevention of privacy leak. Considering *map* is a mutable object and does not enforce strong privacy protection in and of itself (all of its attributes has *protected* modifier), this method has in effect provided full access to modify *map* to any client that uses the class Location and is located within the same package or a subclass of GameMap. It would be possible for any such client code to, for example, change the value of map's *widths* and *heights* NumberRange to any illegal value whatsoever, or pass *null* Actor value into *actorLocations*, bypassing the null check in `addActor` mutator etc. Although initially declared as a private property of Location, *map*'s access restriction is now highly compromised, making it even more exposed than a *package-private* property.
(Fairly speaking, *map* is safe within the encapsulation of game engine package as change within the engine is not allowed but for subclasses of GameMap outside the package, these threats still hold applicable).

Recommendation:
This privacy leak can be addressed from either Location or GameMap class. One option is to improve `map()` getter to a safer version by having it return a deepcopy of *map*. However, this approach could be cumbersome as it requires copy constructors for GameMap and each of its reference-typed variables.
An alternative option that is less laborious is to change the visibility of GameMap's properties from *protected* to *private* and implement setter methods accordingly. No copy constructor needs to be added and `map()` getter can stay unchanged.

The advantage of both recommendations is that they provide an extra layer of data protection for *map*, preventing unauthorized modification or passing of illegal value from client codes. The resulting program is less prone to crashing and overall robustness is improved. Nonetheless, enforcing these protective measures does increase the difficulty in using GameMap and Location class as their public interfaces are reduced and modified. As a result, codes that use these classes must be adjusted accordingly, for example in World class `gameMap.actorLocations = actorLocations;` must be changed to `gameMap.setActorLocations(actorLocations);`

Issue 2: Shotgun Surgery smell between GameMap and Location caused by `makeNewLocation` method

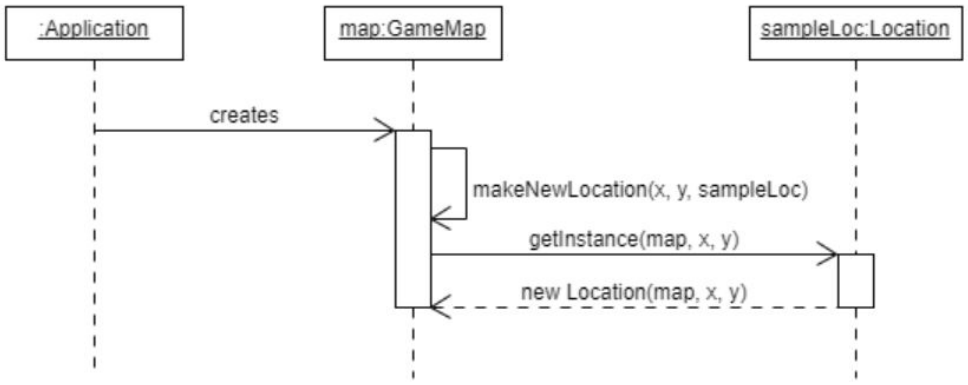
```
protected Location makeNewLocation(int x, int y) {
            return new Location(this, x, y); }
```

The current `makeNewLocation` method hardcodes the type Location by directly calling Location's constructor. By doing so, it forces all the Locations used in GameMap to be of the same type and to be of type Location. It would be impossible for the API users to subclass Location and use it on this GameMap unless they also subclass GameMap and override this method, which means for every new Location class, there must also be one new GameMap class created. Not only does this design give an apparent Shotgun Surgery code smell, it also reduces class reusability as GameMap class becomes useless once the Location is substituted. It also limits the potential to have a variation of Location types on one GameMap at the same time.

Recommendation:
Our proposed recommendation is to change how `makeNewLocation` instantiates new Location objects by overloading `makeNewLocation`. The overloading method has a Location parameter *sampleLoc* added to its signature, which will be used to retrieve new Location instances of its data type.

```
protected Location makeNewLocation(int x, int y, Location sampleLoc) {
            return sampleLoc.getInstance(this, x, y); }
```

sampleLoc could be provided at map's creation, or be made into a variable of GameMap (which also requires overloading of constructor).
For Location class and subclasses, a zero-parameter constructor and a `getInstance` method will be added.
From Application, the code will look like `new GameMap(groundFactory, lines, new LocationSubclass())`



A shortcoming of this recommendation is that it increases the number of methods in GameMap by a certain amount. However, given the fact that it does not make any modification to the public interface (other than expanding the interface) while removing code smell and giving API users more flexibility, there is a net gain from its implementation.

Issue 3: The use of unchecked exception in ActorLocations class:
The use of unchecked exception, in particular **IllegalArgumentException**, can be seen in two methods in ActorLocations class. The methods are `add` and `move`, which respectively add a new Actor to a given Location and move an existing Actor to a new Location. The recommendation we propose is to limit the use of the exception to `move` and replace it with a checked exception in `add`.

For `move` method, **IllegalArgumentException** is thrown when the provided Location is already mapped to another Actor (in `locationToActor` HashMap). The method is a part of a complicated mechanism that moves Actors around the GameMap and hence, if its Location parameter is wrongly provided, it could mean that there are also problems somewhere along the process leading to the calling of the method. (Such process entails identifying Exit points of a Location, Map's verifying if a Location is occupied by an Actor, determining the accessibility of a Location, etc.)

Using unchecked exception in this case is well-justified from **Fail Fast**'s point of view. As it can be used as pointers for underlying logic issues of the code, the principle states that it is good practice to allow such an exception to manifest itself even at the cost of crashing the program. Since unchecked exception is exempted from Catch or Specify requirement, it is the right fit for an error that is not meant to be handled or requires programmers' intervention of some sort.

On the other hand, `add` method does not face the same issue. Actor placement to the Map is a one-step process (and in fact, most of the time, it is called from Driver by API users with a hardcoded Location parameter). If any error is raised, it is much less likely to stem from program's logic but rather, from API users not made aware of the rule that two Actors can't share one Location.
The use of unchecked exception in this case leads to unnecessary crashes for a trivial easily-detectable mistake. It also appears to be time-consuming considering the extra crashing and debugging taken for (API) users to learn about the game rule, while with a checked exception, they can be notified at compilation.

Recommendation:
The proposed change is to throw an instance of **Exception** or a customized checked exception class in place of **IllegalArgumentException,** where each option comes with certain pros and cons. Using Exception would avoid the creation of an additional class and keep the changes to the codebase at minimal. Nonetheless, implementing `try... catch...` with Exception would trap all types of exceptions including the unchecked types, which might not be what it's originally intended for. Using customized checked exception class, on the contrary, means an extra class has to be added but also allows exception handling to be more specialized.