ENGINE REVIEW
FIT2099

BENJAMIN CHEE-WEN RAJAN | 30170397 | braj0006
LIOW GIAN HAO | 30666910 | glio0001

Design Principles Utilized

[1] Avoid excessive use of literals
[2] Don’t Repeat Yourself
[3] Classes should be responsible for their own properties
[4] Reduce dependencies as much as possible
[5] Declare things in the tightest possible scope
[6] Avoid variables with hidden meanings
[7] Liskov Substitution Principle
[8] Command-query separation

Actors and their locations

Issues faced

Actors are unable to access their own location except from the playTurn() method.

Our solution

A private Location attribute as a class level variable in Actor class.

Analysis

The playTurn() method already contains a weak dependency to access to the map thus the location, which would make sense for when they are supposed to make a move. If actors are able to access their location from their own class, it would adhere to classes being responsible for their own properties [3].

However, it would not reduce dependencies [4], and actually increase it. It can be argued that why does an Actor need to know any kind of information about the map? To counter this, we do not us GameMap, instead we only store the Location object, which is an attempt at reducing unnecessary dependencies [4] as now the GameMap exists only as a weak dependency again.

This implementation idea would introduce more flexibility for the next suggestion below.

Actions and Reactions

Issues faced

Certain actions may be more suited to be contained within the object responsible for it [3]. Should losing a limb and dropping a weapon be done within AttackAction? Should a human creating a corpse be an AttackAction's responsibility?

Our solution

Implement a Reaction class. The superclass of Actor, Ground, Item will contain a attribute which is a list of reactions and also contain add, setters, getters for it to ensure Liskov Substitution Principle [7] is followed when subclasses extends them. It is similar to allowableActions.

Analysis

The idea of reaction classes are for actions that are fully dependent on the object and nothing else.

With reaction classes, when a zombie is hurt, all the events such as losing limbs, dropping the limb as a weapon are all part of a Reaction object of getting hurt as a zombie. This way, we declare things in the tightest scope possible [5] as AttackAction is now relieve of the responsibility for a Zombie's reaction to an event [3]. Now when a human dies, it can place it's own corpse on the map as well! The access for the location is from the previous engine suggestion, where we suggest that actors should be given access to where they're at.

The issues that comes with this idea is that repeating code [2] would occur. A reaction where the zombie drops item would work almost identically to the action of dropping an item. Also, it may contradict dependency reduction [4], where some reactions may end up requiring more information.

However, we still believe that a reaction concept, if implemented well, will allow much more flexibility later on as more and more upgrades are to be added.

Weapons and their attributes

Issue faced

We felt like weapons do not contain enough of their own information. A weapon should contain all necessary information and also have it be accessible from other classes as this adheres to classes being responsible for their own properties [3].

For example, certain weapons contain miss chances. To achieve this, we were required to check the instanceOf or check the verb of said item to determine it's miss chance, and the miss chance will then be stored in the AttackAction. This causes the code to smell due to more literals [1] introduced, variables with hidden meaning [6] are introduced and the weapon is not in charge of their own [3] miss chance anymore.

Our solution

The WeaponItem superclass should contain a miss attribute, which is an int/double type and can be accessed. This allows any subclass to also contain such attributes.

For example, in future AttackAction implementation, after obtaining the weapon, we will then check for its miss chance through a getter, and then determine whether the attack was successful or not.

Analysis

In majority of games of such genre would usually not have a miss chance, and since we are looking at a general case, it may be argued that if we were to add a miss chance for all weapon types, but not utilizing it completely may cause the entire miss feature to be redundant.

However, the value can be defaulted if it is not utilized in certain implementation with the use of multiple constructors, one that requires miss chance input and one that does not.

This way, the game engine is able to provide a more diverse set of features and ease the extension of future improvements without compromising core design principles. Adding on to that, different options may also be implemented such as weapons that provides healing to the user, weapons that deals more damage with a chance etc.., and it would only require adding a few extra constructors.

World and processActorTurn

Issues faced

The World class being in the engine made us feel limited on the kind of changes we could apply to it. Furthermore, the method processActorTurn() in world seems redundant for non-player characters, as non-player characters perform actions strictly through behaviours. It may seem that this method contradicts declaring things in the tightest scope [5] or repeating code [2].

Our solution
The engine containing a more abstract World class. And an extension of World in the game package similar to the current World in engine. Rename processActorTurn() to process processPlayerTurn() and processActorTurn() will be exclusive for non-players.

Analysis
A world class in the game package allows us to add more interesting features and makes it flexible for change. It can be argued that we could just extend and create our own World in engine, but then we would either be calling the superclass methods or copy pasting the same code. If we were to call superclass methods, we lose the flexibility to change the inside details of method functions, and copy pasting the same method would just be code repetition [2].

With two separate process, we can clearly differentiate the two different entities in the game, a player-controlled and an automated. This allows declaring things as tight as possible [5]. However, it would break the Liskov Substitution Principle [7], where a player is meant to be a sub class but instead starts to perform vastly differently from usual actors. To add on to the negatives, the world is contained in the engine package, while the player contained in game. However, this would even more justify the decision to have World in the game class.

Action Hotkey

Issues faced
Action contains a hotkey() method that returns a String.

Our solution
Personally, we felt like it would have been more appropriate as a Char datatype instead.