

DESIGN RATIONALE

FIT2099

BENJAMIN CHEE-WEN RAJAN | 30170397 | braj0006

LIOW GIAN HAO | 30666910 | glio0001

***THIS DESIGN RATIONALE IS ONLY INCLUSIVE OF ASSIGNMENT 3 FEATURES BUT
CHANGES MADE TO ASSIGNMENT 2 ARE DOCUMENTED AT THE BOTTOM***

Design principles are indicated in BOLD and [number] as a reference.

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

New Maps

Added Classes

- MapCreator

Added ENUM

- MapVariants {START, TOWN}

With the additions of maps, the Application driver class will start to get flooded, and changes/additions the classes will be harder to keep track, this is due to the introduction of **even more literals [1]**. In some instances, it may also introduce **meaningless variables [6]**.

Our approach to the new town map was not to just add the new map and fill the map with actors and items in the applications, instead we created a new class that would specifically create maps and fill the maps with units based on the map variant chosen. This method is done with a MapCreator class that creates a map and to choose the type of meant for creation, you would access the ENUM MapVariants. This helps us **contain map creation/modification [5]** within a class itself, and the use of ENUM **avoids excessive use of literals [1]**. It also enables us to **declare map creation within a specific tighter scope [5]**. Within the methods in MapCreator, creation, filling of the map takes an ENUM parameter, and selects the map based on this, this is to **avoid code repetition [2]** where one method is able to do the trick instead of multiple different public methods for map creation.

With this solution, the application will only be required to create a world object and call MapCreation when maps are meant to be added or filled. It also presents a much tidier code. If future updates or additions are required, the developers are only required to refer to MapCreator for any map related changes.

Shotguns and Snipers

Added Classes

- Shotgun
- ShotgunAction
- ShotgunSelectionAction
- Sniper
- SniperAction
- SniperShotAction
- SniperTurnAction

Added ENUM

- Directions {NORTH, SOUTH, EAST, WEST}
- ItemCapability added {IS_SHOTGUN_AMMO, IS_SNIPER_AMMO}

Shotgun

As per **Liskov Substitution Principle [7]**, the Shotgun class is an extension of a WeaponItem, and will abide by its functionalities and properties. To **ensure classes contain their own functionality [3]**, ShotgunAction will be the action that will initially be called and provide a submenu, once selected, it calls ShotgunSelectionAction, where the effects of the shotgun takes place. Shotgun will contain an allowableAction of ShotgunAction. Within ShotgunAction, a submenu containing 4 different ShotgunSelectionAction appears for the 4 directions. The chosen action will execute().

This way, the ShotgunAction doesn't need to know how the damage of a shotgun is done, and instead will only require to call ShotgunSelectionAction, where the effects are done AFTER the direction is known, allowing us to **reduce dependencies [4]**. All necessary attributes are also stored as class level attributes to **avoid literals [1]** but attributes not required to be global are declared **within the method's scope [5]**. With the use of Directions as ENUM, we avoid **meaningless variables [6]** and also **reduce use of literals [1]** as ENUM values are fixed. The class ShotgunSelectionAction was created to **reduce code duplication [2]**, where now a single class is able to process the 4 directions.

Sniper

Similar to Shotgun, Sniper also abides the **Liskov Substitution Principle [7]**, where its functionality is kept similar to its superclass. The sniper action starts from the SniperAction, which is an allowableAction. By calling this, the SniperAction makes use of the map's number range method and creates a loop that finds all the possible targets possible. After the loop, an ArrayList of Actor objects is created, the actors will also be of UNDEAD ZombieCapability. To **keep code repetition as low as possible [2]**, we loop the targets and create a SniperTurnAction for each of the targets. The actions are compiled, and a submenu is called to present all the possible targets as choices.

By selecting a target, an action known as nextTurn will store a SniperTurnAction, which can be called by getNextAction() in the SniperAction. Now, during the player's turn, they will be presented an aiming option, it will bring up a submenu, allowing them to choose between shooting or aiming. The submenu is controlled by SniperTurnAction, which also contains conditional statements where if aiming has been 3 rounds, you can only shoot. The actions

stated above will always keep track of the number of turns aimed, when the choice to shoot is chosen, SniperShotAction is called, when aim is chosen, getNextAction() is manipulated to return SniperTurnAction, where the round is now +1. In SniperShotAction, conditional statements present will determine the damage and the miss chance, which will then perform the hurt action.

The implementation took multiple classes, which may seem a lot but necessary due to to ensure **everyone is in charge of their own properties and functions [3]**. Attributes are also **declared tightly [5]**, where the rounds are controlled by SniperTurnAction, the miss chance is under SniperShotAction, and searching for targets is done initially by SniperAction. Like Shotgun, attributes are declared as class variables to **avoid literals [1]**. The use of 3 actions helped **reduce dependencies too [4]**. SniperTurnAction will add itself onto its getNextAction, allowing us to **not repeat ourselves [2]**.

Ammo

Ammo is an extension of an Item class, represented by an 'o' and 'i' for shotgun ammo and sniper ammo respectively. It contains a capability IS_SHOTGUN_AMMO and IS_SNIPER_AMMO to enable other events to check the properties of the item without the use of instanceof (code smells).

For ammo checking, it is both done within the initial ShotgunAction and SniperAction. Within execute, before any of the previously mentioned code runs, it runs a conditional statement where it reads the player's inventory and checks if there exists an item with capability IS_SHOTGUN_AMMO or IS_SNIPER_AMMO, which are both enum values. If there exists such item, the item is removed from the inventory and the code can proceed. If there is no such ammo, it will call a return statement which would end the entire function right there.

By keeping the ammo checking within the action, it further emphasizes **on classes and their own responsibilities [3]**. And with the use of enums, **literals are avoided [1]**.

Mambo Marie

Added classes

- VoodooZombie
- VoodooGround
- VoodooBehaviour
- VoodooVanishAction
- ChantAction
- CreateZombieAction

MamboMarie classes have all been named within the Voodoo naming scope. This is because we assumed that MamboMarie represented the name of the new unit, all related behaviours and actions were mostly written under the same naming scope of Voodoo.

An important property of MamboMarie is that it does not appear initially, and every turn, a 5% chance to appear. We concluded that to **reduce dependencies [4]** as much as possible, we would implement a special ground object in the map that would be responsible for the spawning of the Voodoo priestess. The ground object known as VoodooGround, is not penetrable and is denoted by a lowercase 'm'. We utilized the tick() method to determine if a VoodooZombie were to spawn. As **classes should be responsible for their own attributes [3]**, we implemented the spawn chance attribute within the class as well.

To spawn, it goes through a conditional statement, and when it is true, it replaces the existing VoodooGround with a Dirt object and adds a new VoodooZombie with the name "MamboMarie" into the map. Mambo Marie always spawns at the top left of the map because the VoodooGround will always exist approximately at the top left of the map as well.

In VoodooZombie, numerous attributes are declared (chanting duration, vanish ticker etc..) at the beginning of the class to **avoid literals [1]** later on. The VoodooZombie is denoted by an uppercase 'M'. For its special skills, it will know two different behaviours, VoodooBehaviour and the basic WanderBehaviour. To ensure **classes responsible for their own properties [3]**, the VoodooBehaviour will keep track of the cooldown, the chanting duration and also the number of zombies spawned. If it can start chanting, it will call upon a ChantAction that tracks the chanting duration and also returns appropriate menu description. If it is ready to spawn zombies, it will call CreateZombieAction. If the special ability is still on cooldown, it will return null, where in the behaviour loop, it will tell the program to skip this behaviour and go to the next behaviour's action. As listed above, every action is **declared in a tight scope [5]**, making it easy to differentiate where each event is performed.

Mambo Marie is also capable of vanishing. This calls upon a VoodooVanishAction which will remove the current VoodooZombie with a VoodooGround object and the cycle continues. The tracker and action are both within the playTurn() method.

The design has considered flexibility, if in the future the chanting duration is considered to be more than one turn, or cooldown is required to be adjusted, it can be found in VoodooZombie which will direct it towards VoodooBehaviour's parameters.

The endgame

Added classes

- QuitAction
- WorldZombieGame

Modified methods

- playTurn() in Player

Ending the game was a task that required modification to the world object. If we were to do it in any other class, it would conflict with **classes being responsible for own properties [5]**. However, due to the world class being an engine class, we decided it would be best to extend it, and from our application, we will use our own extended world class. Following the **Liskov Substitution Principle [7]**, subclass will contain methods and functions similar so that they behave the same way. Within the run() method, while process the turn of every actor, a counter will be used to indicate the amount of actors of ZombieCapability UNDEAD or ALIVE. After all actor's turns are processed, a conditional statement is present to determine whether the game should end, if there are no zombies left, the player wins, if there is only one human left (the player), the player loses! When it ends, it removes the player from the map and prints a String statement. The String statement is obtained through a protected getter instead of a class level variable to **reduce literals [1]** and **declaring things in the tightest possible scope [5]**.

To quit the game, a new QuitAction is created. This is because quitting is required to be a selectable option for the player. Within the player's playTurn() method, it will always add a new QuitAction into all of the actions presented to the player. QuitAction is also denoted with a 'q' hotkey, and its execute function will remove the player from the map, which automatically ends the game.

Reviewed Design From Assignment 2

Modified attributes

- behaviours[] array in Human class

Added classes

- Farmer extends Human
- Food extends Item
- FarmingBehaviour extends Behaviours
- ConsumeBehaviour extends Behaviours
- SowAction extends Action
- HarvestAction extends Action
- FertilizeAction extends Action
- ConsumeAction extends Action

We'll be upfront, we realized some major design flaws and unnecessary downcasting or poor design principles in general from Assignment 2. We took the time to tidy it up, and we are hoping this may grant us some pity marks or some form of compensation for lost marks in Assignment 2. We believe that being able to learn from our mistakes is also one of the core **learning principles** as well :D. Please?

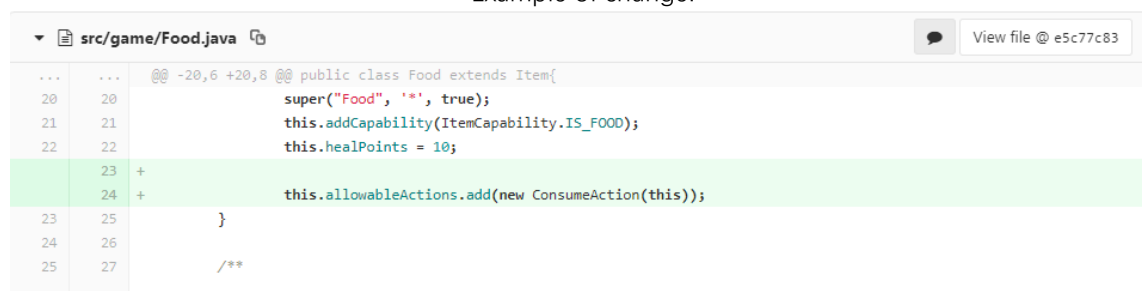
☹️. All jokes aside, here is our justification / correction.

The first part comes from playTurn() in Player as seen in the screenshot below.

```
36         for (Item item: inventory) {
37             if(item.hasCapability(ItemUpgradeCapability.CLUB) ||
38                item.hasCapability(ItemUpgradeCapability.MACE)) {
39                 actions.add(new UpgradeItemAction(item));
40             }
41             if(item.hasCapability(ItemCapability.IS_FOOD)) {
42                 Food food = (Food)item;
43                 actions.add(new ConsumeAction(food));
44             }
45             if(item.hasCapability(WeaponCapability.IS_SNIPER)) {
46                 actions.add(new WeaponAction(item));
47             }
48             if(item.hasCapability(WeaponCapability.IS_SHOTGUN)) {
49                 actions.add(new WeaponAction(item));
50             }
51         }
```

A major design flaw was the redundancy of adding an inventory checker and then adding actions in playTurn(), it **adds dependencies [4]** and it breaks our attempt at **declaring everything in the tightest scope [5]** as well as **classes being responsible for their own properties [3]**. We asked ourselves, why would the player keep track of his inventory and actions when it has already been done in processActorTurn() ? We then realized an extremely easy fix was to just add the actions into the allowableActions in the item classes instead... (silly us). LegWeapon, ArmWeapon, Food, Crops and such classes has been edited to the more appropriate design.

Example of change:



```
src/game/Food.java
@@ -20,6 +20,8 @@ public class Food extends Item{
20      super("Food", '*', true);
21      this.addCapability(ItemCapability.IS_FOOD);
22      this.healPoints = 10;
23  +
24  +      this.allowableActions.add(new ConsumeAction(this));
25  }
26
27  /**
```