Improvements on engine

Done by:
Sutulova Tatiana 30806151
Elaf Abdullah Saleh Alhaddad 31063977

1. Improvements on Display class:

1.1 Problems it solve

Display class is responsible for managing the Input/Output interactions within the system. While the output is properly handled by providing several methods that allow printing with terminating the line afterwards, printing without termination or just terminating the line, the input functionality is not as wide and it includes only one method that reads in the first character of the inputted word. This functionality is useful for the small applications where I/O is mainly based on the small menus where the user chooses either the numeric value or an alphabet. However, for the apps where the user is required to input the entire integer value or a String word, the current **Display** functionality is not complete.

1.2 Design proposed

Methods to add in the Display class:

- **readInt()** , which reads in the entire number from console and returns it.
- **readString()**, which reads in the String from console and returns it

1.3 Advantages and disadvantages

The main advantage of this change is that instead of creating an entire subclass for implementing such a small functionality, it can be just added as methods to **Display**.

2. Duplicated distance() method

2.1 Problems it solve

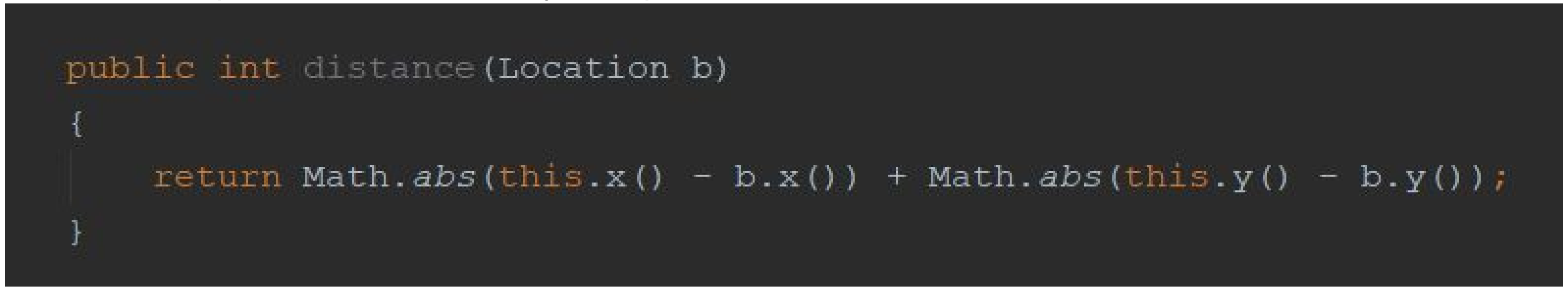
The method **distance()**, which is responsible for computing the Manhattan distance between two locations is private and it is used in several classes such as **FollowBehaviour, ThirstyBehaviour** and **HungryBehaviour**. Keeping this method private

violates Single Responsibility Principle, because SRP states that each class should be responsible for only one thing. Implementation of private methods in a class is a sign that the class is exceeding its functionality scope.
Since there is no abstract **Behaviour** class, which all the behaviours would be inheriting from, we want to suggest another way to reduce the duplication of this method.

2.2 Design proposed

This method may be added inside the **Location** class and it will be implemented taking into consideration that **Location a** will be an actual instance of the class the method is called on. So instead of passing **Location a** as a parameter, **this** keyword will be used, whereas **Location b** will be still passed to the method as a parameter.

The draft example how the method may be implemented inside the **Location**:



```java
public int distance(Location b)
{
    return Math.abs(this.x() - b.x()) + Math.abs(this.y() - b.y());
}
```

Example of the method call from outside of the **Location** class:
Location here = map.locationOf(dinosaur);
Location there = map.locationOf(player);
here.distance(there);

2.3 Advantages and disadvantages

Allows to reduce the repetition of the code and makes the code follow DRY principles, since instead of implementing the code privately in three different classes it will be implemented only once. Moreover, classes that make use of it will not be violating SRP principles and their code will not have smell anymore.

3. Get location of the items on the map - item locations

3.1 Problems it solve

Just like the **ActorLocations** class that is implemented in the engine folder we could add ItemLocations. In many instances, we loop through the whole map to be able to find the items that the **Actor** can interact with. Looping through the map requires us to have nested loops; one for the **xRange** of the map and one for the **yRange** of the map. If we added the **ItemLocation** class and made it an attribute in **GameMap** just like **actorLocations**, we will be able to access the items and their locations without needing to loop through the map to find them.

3.2 Design proposed

Class	Attributes	Methods
ItemLocations	● **Map<Item, List<Location>> itemToLocation**: contains the name of the item as a key and the list of locations that this type of item can be found at. ● **Map<Location, List<Item>> locationToItem**: contains the location as a key and the list of items that are at that location	● **ItemLocations()** - Default constructor, initializes the two HashMaps attributes ● **add(String ItemName, Location location)** - adds a new Item at a given location ● **remove(String ItemName, Location location)** - removes an item at a certain location ● **contains(String itemName)** - returns true if the map contains the item. ● **isItemAt(String itemName, Location location)** - returns true if the location contains the item ● **locationOf(String itemName)** - returns the list of locations that this type of item can be found at

GameMap	● **ItemLocations itemLocations**: contains the items and their corresponding location	● **addItem(Item item, Location location)** - add a new Item at the given location ● **removeItem(Item item, Location location)** - removes the item from the given location ● **containsItem(Item item)** - Is the Item on the map ● **isAnItemAt(Item item, Location location)** - Is there an item of that type in that location

3.3 Advantages and disadvantages

Allows us to find the items without needing to loop through the whole map. This reduces the run time and the complexity of the code.

4. GameMap

In **GameMap** there is an attribute called **actorLocations**. It contains the actors on the map and their corresponding locations. Adding a method called **getActorLocations()** will allow us to be able to access this attribute and in return loop through the actors in the map when we want the player to interact with other actors rather than looping through the whole map to find the actors the player would like to interact with. This allows us to eliminate the nested for loops and reduce the run time and the complexity of the code.

5. execute() in Action class

In **Action** class, the function **execute()** has two parameters: **actor** that is an instance of the **Actor** class and **map** that is an instance of the **GameMap** class. In the implementation of our program, there were several subclasses of an **Actor** class, that contained functionalities (methods) related to only a certain subclass, but not to all subclasses of **Actor**. Hence, when the **execute()** method was overridden in the **Action** subclass which could be implemented by a certain type of an **Actor** only, there were calls to this **Actor**'s methods. Since the parameter in **execute()** is an **Actor** type, we had to add an if statement

that was checking whether **actor** was an instance of a specific subclass and implement downcasting which resulted in violation of the Liskov Substitution Principle and led to a code smell. The principle defines that objects of a parent class shall be replaced with the objects of its children classes without breaking an application and violating it results in degradation of robustness and maintainability of the code.
The function can be improved by modifying the **actor** parameter in **execute()** in the **Action** class (where this method is inherited from in all the actions) to be generic instead of specifying that it is an instance of **Actor**. That way when overriding the **execute()** method in a subclass, we will be able to immediately specify what type the **actor** parameter is, which will help to remove the casting and therefore improve the code in terms of the code smell.

6. Conclusion

Generally, the code in the engine may be considered good as it is but it still requires several improvements that have been mentioned previously. Analysing the code we concluded that it mostly follows SOLID principles, for instance , most of the classes follow Single Responsibility Principle, since each class and interface has its own responsibility and purpose. An example of that is **Location** class, which represents a location on the game map and all the functionalities it handles are only related to the **Location** instances. Following SRP makes managing big programs easier since we immediately know which class to import and use for every purpose in the game. Furthermore, classes in engine follow the Open/Closed principle (OCP), which allows us to easily inherit from the classes, such as Action or Actor, without modifying code in them. Lastly, engine follows the Dependency Inversion principle, which ensures that high-level modules do not depend on low-level modules, meaning that changes that we implement in game classes do not affect the engine, ensuring its reusability.