CptS 487 Software Design and Architecture

Assignment 6

We are continuing with the Plants vs. Zombies example. For this assignment, as usual, ignore the actual game, but follow the assumptions below closely.

1. The plants we need to consider are:
   a) Peashooters.  They shoot bullets which hit the zombies from the front.
   b) Watermelons.  They catapult a watermelon which hit the zombies from above, bypassing accessories such as the screen doors.
   c) Magnet-shrooms.  They remove the metal accessories (bucket, screen-door) immediately from the zombies when they are near.
2. The zombies we need to consider are the same 4 types of zombies, three of which carries an accessory. Only the bucket and screen-door are considered metallic. We are also sticking with the 50/75/150/75 health values for them.




3. The demo game play shall extend on the functionalities in Assignment 4, allowing user to choose from three different attacks by 1) a Peashooter 2) a Watermelon and 3) a Magnet-shroom.
4. For simplicity purpose, we assume the Peashooter's damage is 25 each time it attacks, and a Watermelon's damage is 40. The leftover damage of the watermelon should be carried over from the accessory to the Zombie, while the magnet-shroom only affects the metal accessories. Every time a Zombie gets damaged, its health is reduced by the value of the damage. Once the health of the Zombie is <= 0, the Zombie, with or without its accessory, "dies".
5. Also, if the Zombie has an accessory, the damage will be applied to the "accessory" of the zombie first. Once the "accessory" is "destroyed", the accessory will fall and the "Zombie with an accessory" should become a "Regular Zombie".

There are two exceptions: 1) When a watermelon attacks the Screen Door Zombie, it damages the Zombie directly – and once the Zombie's health is <= 0, the whole Zombie dies. 2) When a Magnet-shroom attacks a Zombie with a bucket/screen-door, it removes the metal accessory immediately regardless of how much health the accessory has remaining, turning the whole zombie into a Regular Zombie. The Magnet-shroom has no effect on Regular Zombies or Cone Zombies.

Read the assumptions carefully and complete the tasks below.

More importantly: Absolutely no plagiarism allowed, just as all other assignments and submissions.

1. **[10pts]** Choose a Creational Pattern we have covered (Factory/Abstract Factory/Builder) to create the 4 types of "Zombies". This time, use Decorator pattern to model the "Zombie"-related classes. Draw a class diagram for your solution using StarUML or other UML tools. The diagram should connect the Creational pattern part and the Decorator pattern part. Also, include necessary attributes and operations in your diagram.

   Hint: the "Zombie"-related classes MUST have takeDamage(int d) and its necessary variations; and there should be a die() method as well. A bool isMetal attribute and/or a bool getMetalStatus() operation *may* be necessary too depending on your implementation. Refer to the solution to Assignment 4 for ideas.

   Once again, include the "match-up" pairs between your classes and the participants of the patterns' structures, for both your Creational Pattern and Decorator Pattern.

   Note: your diagram does not need to include the GameEventManager or GameObjectManager down below.

2. **[80pts]** Write an executable demo program that follows your design above. The program should provide a simple command line interface of several commands. A sample command line interface may look like this:

```
1. Create zombies?
2. Demo game play?

1
Which kind?
1. Regular
2. Cone
3. Bucket
4. ScreenDoor
```

   a) **[10pts]** The creation part is the same as what we required in Assignment 4, except that this time your Zombie-related classes should be organized following the Decorator pattern. Pick Factory or Builder pattern to do the creation; Abstract Factory is not a good fit here as we no longer have the Composite that needs to enforce certain rules (i.e.

ConeZombie *must be* Cone + Zombie, etc).

b) **[30pts]** Next, the user can simulate a plant attacking the zombie by selecting the "Demo game play" option. The zombies are attacked one at a time from left to right, same as in Assignment 4.

Use the command line output to simulate the attack process: each time the plant attacks, print out an updated array of the Zombies and their health values. Remember, the Zombies with an accessory might change type. See c) below on the requirement of how this array should be updated.

The user should be able to manually select one of the three plants to perform the attack every time he/she wants to simulate an attack. Follow the assumption above on how the different attacks work. For instance, the user may select "Peashooter" for the first attack, and then select "Watermelon" for the second, the "Magnet-shrooms" for the third, and "Watermelon" again for the fourth, and so forth.

c) **[10pts]** Here are two pieces of code snippets from two classes "GameObjectManager" (GOM) and "GameEventManager" (GEM). You are **required** to use these two classes and the skeleton code in your submission (with necessary translation to C# if you're using C#).

Assume that they work in the following way:
GameObjectManager keeps track of all game objects: plants, zombies. For this assignment, we only focus on the list of all zombies, which the user created earlier. The Zombie you created should be put inside the list of GOM.

```
class GameObjectManager
{
    //Keep track of all the enemies.
    //Accessed by GameEventManger when calculating collision.
    //The List here is filled when the user creates the Zombies.
    List<Enemy> enemies;

    //TODO: Observer Pattern related attributes and methods.
}
```

GameEventManager is the controller class that handles collision detection and corresponding game logics. For this assignment, the collision detection is simulated by the user's choosing of an attack. I.e., when the user selects a type of attack, GameEventManager calls the respective methods of the Enemy.

To simulate the attack, do not include specific plant classes in your program. Every time the user selects a type of the attack, use a switch-case in "simulateCollisionDetection()" in GEM to call the corresponding damage function in the Zombie being attacked.

- When user selects "Peashooter attack", the program calls "simulateCollisionDetection(1)", then case 1 calls: "doDamage(25, e)", which calls the "takeDamage(25)" on the enemy e;

- When user selects "Watermelon attack", the program calls "simulateCollisionDetection(2)", then case 2 calls: "doDamageFromAbove(40, e)", which calls the "takeDamageFromAbove(40)" on the enemy e;

- When user selects "Magnet-shroom attack", the program calls "simulateCollisionDetection(3)", then case 3 calls: "applyMagnetForce(e)", which calls the corresponding method about magnet on the enemy e. You may need to do some kind of type check (but not a must) to implement the magnetic attack. It's up to you to decide how.

d) **[30pts]** Obviously, the collision detection method described above in GameEventManager needs to access the enemy-list (and plants/bullets list for a real game) inside GameObjectManager. Therefore, the list in GameObjectManager needs to be up to date as the game proceeds. Assume that the list **only needs to be updated** (Crucial Hint!) when an enemy within the list "dies".

You must adopt the Observer pattern to implement this feature. You **may need** to update GEM/GOM as you see fit as well. For instance, changes of a return type, and/or the parameter list might be necessary. The bottom line is, you need to implement the Observer pattern properly.

Remember to include the "match-ups" of your classes and the structure of the Observer Pattern too: which class is the "Observer", which class is the "Observable/Subject", etc. Hint: 1) GEM is NOT part of the Observer Pattern – it's an external client of the Observer Pattern. 2) Think about which function(s) in which class(es) would work as the "notify()" method in the Observer Pattern template? 3) Make sure your implementation correctly follows the behavior sequence described in the lecture.

```
class GameEventManager
{
    //Called when collision is detected between a Bullet b and an Enemy e
    //In a real implementation, the operation will get the damage from the
    //Bullet b. For instance:
    //public void doDamage(Bullet b, Enemy e)
    //{
    //    int damage = b.getDamage();
    //    e.takeDamage(damage);
    //}

    //For this test, simply pass in the damage of the Peashooter/Watermelon
    //to the following operation (25 or 30, specifically)
    public void doDamage(int d, Enemy e)
    {
         e.takeDamage(d);
    }

    //Called when "collision" is detected between
    //a magnet-shroom and an Enemy e
    //i.e, when the user select the magnet-shroom attack.
    public void applyMagnetForce(Enemy e)
    {
        //If it's a metal accessory, we need to remove the “metal” accessory
        //from e. How? It’s up to you

        //TODO: complete this method.
    }

    //To separate the responsibilities, the above methods should not
    //be called directly from your code handling user-interaction.
    //Instead, it should be done in this “hub” operation in the control
    //class. Since we are simulating, pass an “int” to represent the plant.
    public void simulateCollisionDetection(int plant)
    {
        //The method gets access to the “enemies” list in GameObjectManager
        //and finds the first Enemy to be the one to collide with.
        //Then, it passes e to one of the functions above.

        //TODO: complete this method.
    }
}
```

3. **[10pts]** Compare Composite and Decorator patterns. In your opinion, which one works better for this particular example? Explain your choice.

- **Submission Details**
  1. Submit your class diagram and short answers in a well-formatted PDF file, with your name and student ID included. The diagram must be drawn with StarUML and show correct usage of the UML notations, as well as sufficient details for specified requirements.
  2. For the coding part, write your program in either C# or Java **only**. Other languages are not allowed. Include a Readme.txt file explains how to run and test your program. For this exam, you might assume I will not test on illegal inputs. However, you should include a Readme.txt file that details how to compile, run and test your program. The key here is to test on your understanding on the design patterns.
  3. Your code need to also correspond to your design of class diagram.
  4. Zip both the PDF and your entire coding project (include all source codes and files necessary to run on a Windows 10 machine with Visual Studio/Eclipse installed, and the Readme.txt), name it "ID-FirstnameLastname-487Assignment6.zip", and submit it on to the Blackboard dropbox before the due date.
  5. Finally, do not plagiarize.