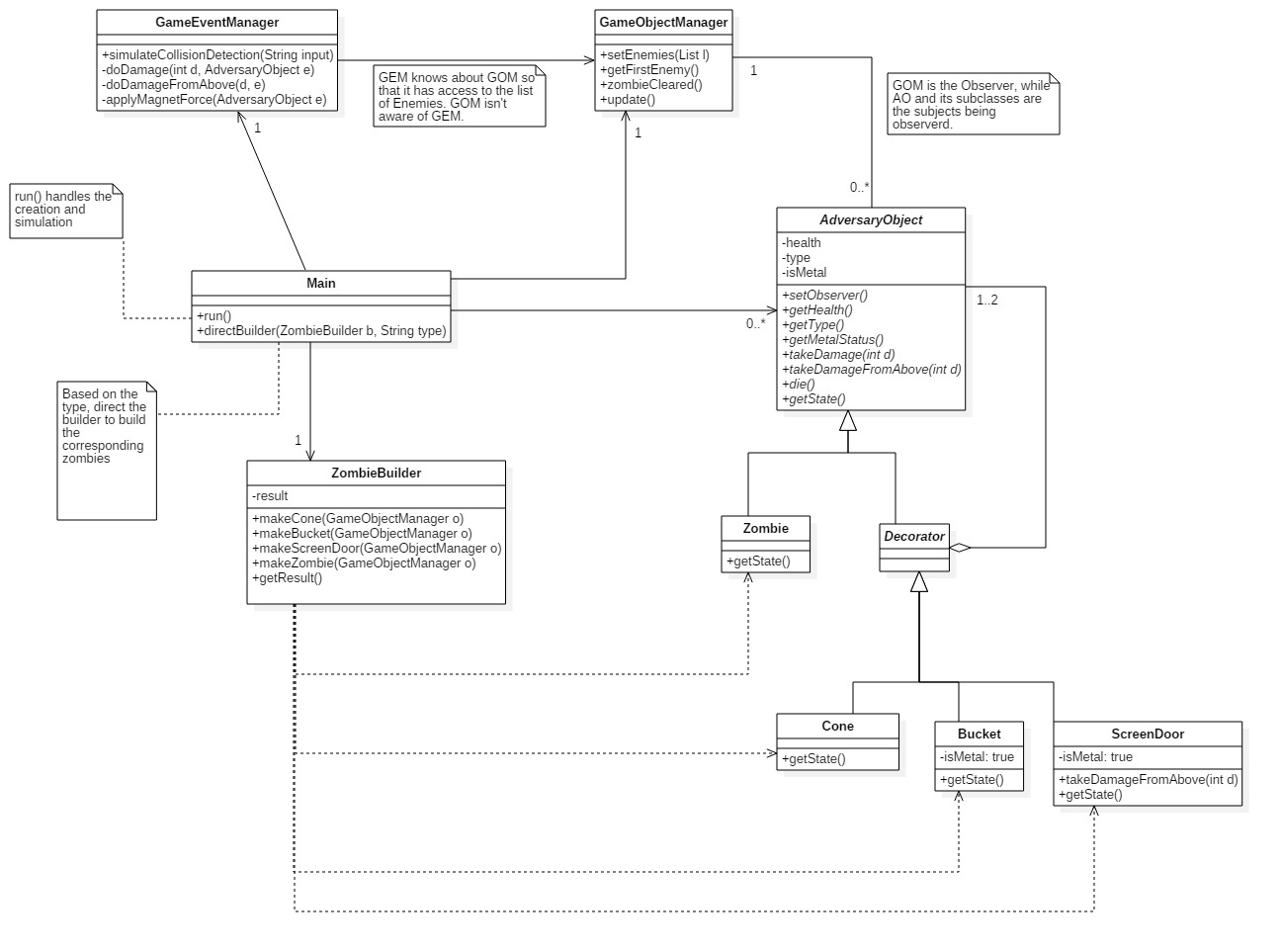
## CptS 487 Software Design and Architecture

## Assignment 6 Solution

1. **[10pts]**

See the diagram below. Also available in Assignment6-solution.mdj/.jpg.



You don’t need to include GameEventManager (GEM) or GameObjectManager (GOM).

1. The Decorator Pattern section:

Match-ups:

Component – AdversaryObject

ConcreteComponent – Zombie

Decorator – Decorator

ConcreteDecorator – Cone/Bucket/ScreenDoor

Note that the structure is the same as in HW4. In this case, the decorator is used to wrap around other *Component* inside it, and “shed” itself when it dies.

In my solution, I did not include “addedBehavior/addedState” in the “Decorator” section of the class diagram, mostly because the new requirements can be implemented by the existing functions. The type check is performed once in GameEventManager before applyMagnetForce().

1. The Builder Pattern section:

Because GOM is the Observer to the AdversaryObjects, we need to configure the Observer to when creating the Zombies and Accessories. Therefore a *unique* GOM is passed into the make() functions in the Builder – this might be missed in the implementation.

1. The Observer Pattern section:

As specified above, GOM is the Observer, and AdversaryObject and its subclasses are subject. There should be arrows pointing from the ConcreteSubject classes (Zombie/Cone/Bucket/Door) to the GOM due to the “die()” notification, but I’m skipping them for clarity.

1. Others:

GEM only functions as a controller that works as a bridge between user input and game logic. It separates the responsibility from the Main class.

**Even more flexibility is available in this case. All reasonable strategies will be accepted depending on how well you utilized the patterns.**

1. **[80pts]** Write an executable demo program that follows your design above.

See attached solution code and comments in Java. Changes to the code template are permitted because they are necessary. In my solution, for instance, the applyMagnetForce() simply calls the die() function directly on the metal objects.

The match-ups for the Observer pattern are:

Observer/ConcreteObserver – GameObjectManager (they are merged as we don’t have multiple concreteObservers)

Subject – AdversaryObject

ConcreteSubject – Zombie/Cone/ScreenDoor/Bucket

See explanation after the class diagram in Q1 too. Remember that in Observer pattern, the Subject must have an Observer(s) “attached” to the subject. In my code, this happen in the Builder/Factory stage which makes the most sense to me.

Functions that are crucial to patterns should be correctly utilized. For instance, an update() in GOM, using die() in the AdversaryObjects as “notify()”, and implement getState() in the Zombie/Decorators should be the proper way of implementing the Observer pattern.

1. **[10pts]** Compare Composite and Decorator patterns. In your opinion, which one works better for this particular example? Explain your choice.

My personal opinion is that Decorator is much better. As you can see from my code, every function has only a minimum of lines, and the objects are much more light-weighted and flexible.

But your argument is what mattered here.