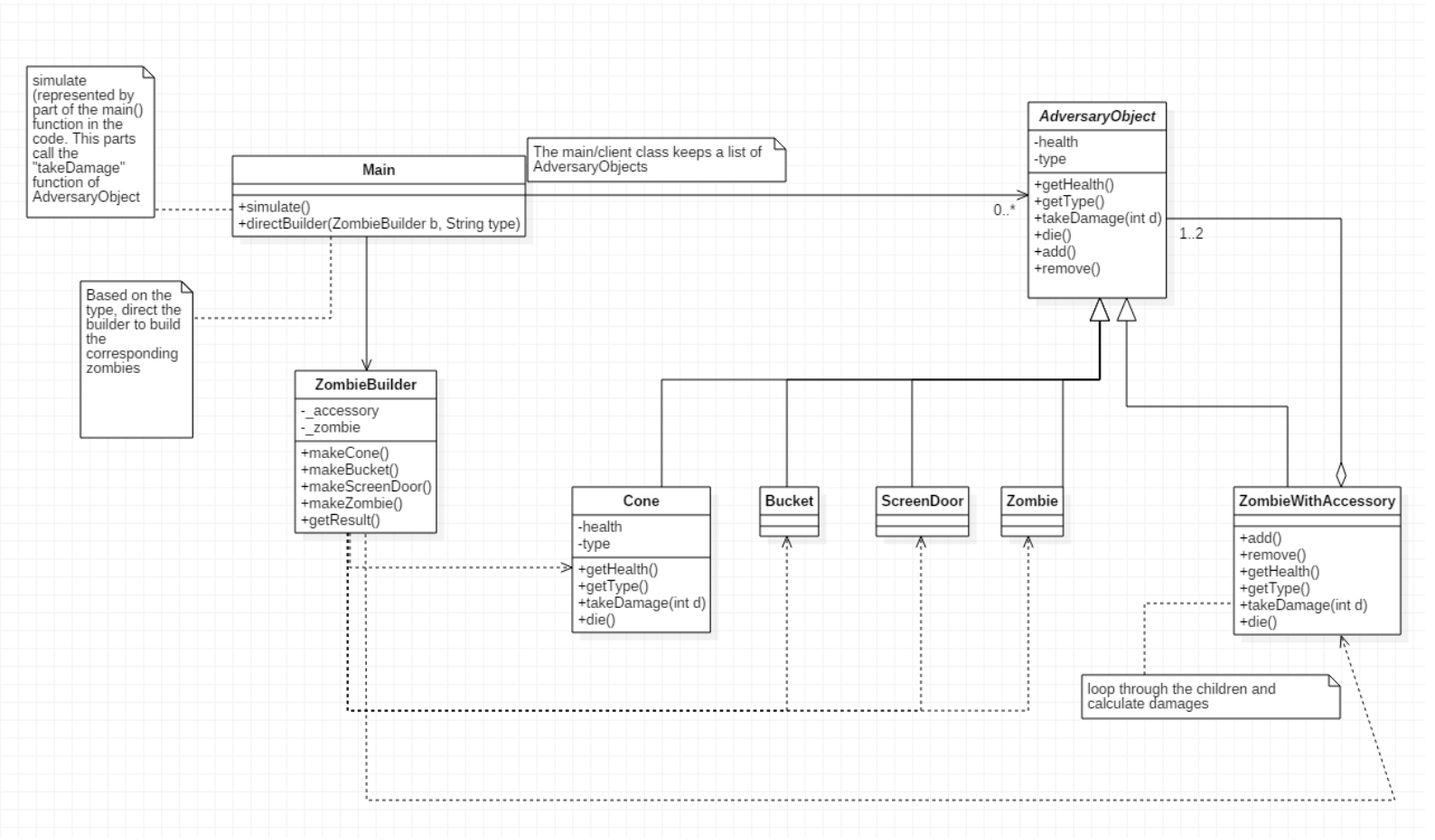
CptS 487 Software Design and Architecture

Solution

1. **[10pts]**

See the diagram below. Also available in midterm1.mdj



In my solution, I choose to use Builder as the creator pattern. In this case, the abstract *Builder* class and the ConcreteBuilder class are merged into one, as we only need one Builder to build the "Zombies".

Following the Composite pattern, the *AdversaryObject* defines an abstract class ("*Component*"), with the universal interfaces defined (getHealth(), getType(), takeDamage() and die()), as well as the add/remove function for the "Composite" element.

You might also notice that die() function returns a boolean value in the code: this is so that the Main class could learn of a zombie's death, and remove it from the list of enemies accordingly. It would not be appropriate for the zombies to somehow remove themselves from an array

that they are not aware of. **Of course, this implementation could/should be changed when the demands are different, for instance, when you will learn in the future about the Observer pattern.**

Cone/Bucket/ScreenDoor/Zombie are the "Leaf" elements (details in Bucket/ScreenDoor/Zombie are skipped because they are identical with the Cone class), while ZombieWithAccessory is the "Composite" element. The "leaves" are created by the factory methods in the Builder, while we use the getResult() function to put together the ZombieWithAccessory.

Finally, the main class servers as the Client in the Composite Pattern, as well as a merge of Client and Director in the Builder pattern. For details, please see the source code.

There are many variations to this solution of course. First of all, for the Creational Pattern, Builder is best suited, as Composite involves putting together the Leaves. Factory might still work, and the TA will grade it based on your implementation. AbstractFactory, however, doesn't really work in this scenario, as it doesn't fit the condition that would require an AbstractFacotry, i.e., a "family of products" and/or multiple sets of products. There is one exception: if you consider Zombie and Armor different products (ProductA and ProductB) in the Abstract Factory pattern, then it might make sense.

Secondly, for the Composite Pattern part, in this solution, I use "ZombieWithAccessory" to represent all the "zombies". You may define each individual composite (ZombieWithCone, ZombieWithBucket, etc.). And, RegularZombie is also represented by ZombieWithAccessory, but you might also simply use the Zombie "leaf" class to represent it.

2. **[10pts]** The matching relationships are:
   For Builder:
   Main – Director
   ZombieBuilder – Both *Builder* and ConcreteBuilder, for simplicity purpose
   Everything Zombie Related – Product

   For Composite:
   AdversaryObject – Component
   Zombie/Cone/Bucket/ScreenDoor – Leaf
   ZombieWithAccessory – Composite
   Main - Client

3. **[40pts]**

   See attached solution code and comments in Java. Note that add/remove code has to become empty place holder in the leaves – another option is to keep the add/remove operations in the Composite only.

Also, note that operations in *AdversaryObject* must be overridden in each leaf class, which causes a little redundancy in the code. However, this is good for future extension purposes.

To test the code, choose 1 to create Zombies first, then choose 2 and set the damage value. The program will automatically simulate the whole process until all the Zombies are dead.

4. **[20pts]**

When the damage is 25, the health of zombies/accessories are reduced cleanly every time. If you increase it to 40, when the cone/bucket/door fells off, the remaining damage of the "bullet" is carried over to the "Zombie" part of the Composite.

You could take advantage of the Pattern by making the takeDamage function return an int value to represent the leftover damage, instead of the void return that you might have planned initially. Meanwhile, this damage is not carried over among the zombies.

5. **[20pts]**

To handle the Watermelon attack, my solution is to split the takeDamage(int d) function in to two: takeDamageFromFront() and takeDamageFromAbove(). The client would decide which one to call when it decides which plant is doing the attack. Note that, by doing so, both functions are counterparts to the "operation()" stated in the Composite pattern structure. (As a matter of fact, getHealth() and getType() are also counterparts to the "operation()")

Then, in Cone/Bucket/Zombie classes, the takeDamageFromAbove functions exactly the same as takeDamageFromFront: they take the damage from both front and above in the same way.

However, in the ScreenDoor class, takeDamageFromAbove deals no damage to itself. We simply use the return int value defined in Question 3, and make it carried over on to the "Zombie" part directly. So that the damage is applied on the Zombie directly.

And, we would need to change when the ZombieWithAccessory "die()" in this situation. No longer does every component has to be dead for the ZombieWithAccessory to be removed, but once the Zombie part is killed, the "whole ZombieWithAccessory" should be considered dead. Hence, this is another place that needs to be modified.

In general, the Composite pattern would mostly still work in this case. But as stated, it requires duplicate the takeDamage function in other classes that do not have to deal with it.