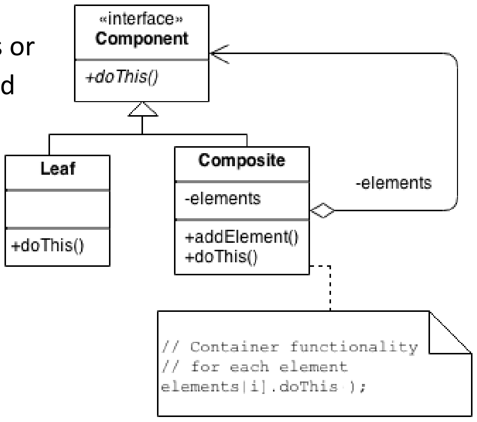
Design Patterns used in Castillo De Luna

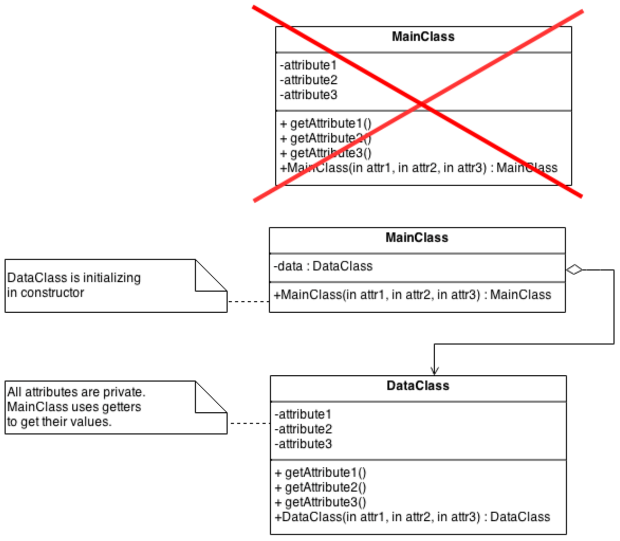
Simple Design Patterns used:

Composite Structural Pattern is a tree structure of simple and composite objects. It can have a 1-to-many "has a" relationships or 1-to-many "is a" relationships. In Castillo De Luna, we have Good Character Composite objects and a Bad Character Leaf object that both inherit from a blank character base class. They are both characters of the same type but processing of each other happens differently because they are different types of objects. The good player can add elements to its list of attribute variables, the leaf object cannot.

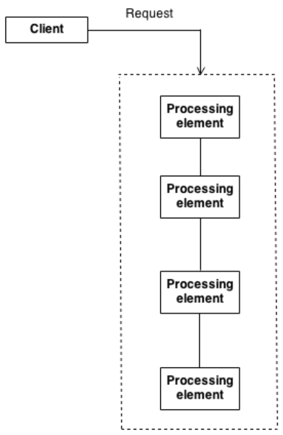


Private Class Data Structural Pattern is also used whenever objects are created and used in the game. All objects in our game are first initialized with data in private fields during construction and then they are only accessed by getter methods and only set using setter methods. This controls who can access what and when.

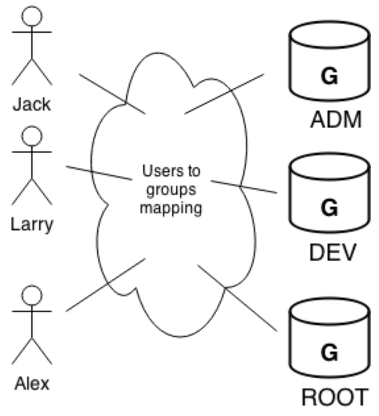
After a good character is created with a name and health and inventory of items, only getter/setter methods should be able to access and make changes to the characters health or inventory. This would prevent things like main classes from making direct changes to objects that weren't intended to be changed in that way.


MainClass
-attribute1
-attribute2
-attribute3
+ getAttribute1()
+ getAttribute2()
+ getAttribute3()
+MainClass(in attr1, in attr2, in attr3) : MainClass
MainClass
-data : DataClass
+MainClass(in attr1, in attr2, in attr3) : MainClass
DataClass is initializing in constructor
DataClass
-attribute1
-attribute2
-attribute3
+ getAttribute1()
+ getAttribute2()
+ getAttribute3()
+DataClass(in attr1, in attr2, in attr3) : DataClass
All attributes are private. MainClass uses getters to get their values.

The Chain of Responsibility Behavioural Pattern is a type of pattern that asks the object in question to handle a request by the object calling it, but if it can't be handled, it shouldn't just fail but should try to be handled by another object class, maybe even it's base class or it's super () until the request can be completed. Castillo De Luna has consumables and non consumables that both can handle object requests, but if for some reason they are unable to do something, their base class that they inherit from should be asked if it can handle it. Same concept with the good and bad characters that inherit from a base super character class.



Mediator Behavioral Design Pattern defines an object that encapsulates how sets of different objects interact. There is a loose coupling by keeping objects from referring to each other explicitly. In the game we have a good character who has access to items both consumable and non consumable, as well as rooms, other bad characters, and storyline messages. The good character doesn't interact directly with any of those objects, but more of a parent mediator that does the actual interaction. If object A needs to do something with object B, it should talk to the mediator to get the work done, and not just talk directly to the other object.



Antipatterns found in Castillo De Luna:

Spaghetti Code Software Development Antipattern is a software structure that is made difficult to extend and optimize because of how it is coded. Frequent code refactoring can improve software structure, support software maintenance, and enable iterative development. This code smell has created a massive main method in our game and some functions or parts of the games have huge implementations instead of 1 liners or simple 2-3 steps and be done. We are working on refactoring and making these long procedural spaghetti methods into smaller chunks that are more manageable. We are also creating more small functions in adjacent object classes instead of doing more work from the main.

The Blob Software Development Antipattern is a procedural-style design that leads to one object with a lion's share of the responsibilities, while most other objects only hold data or execute simple processes. The solution includes refactoring the design to distribute responsibilities more uniformly and isolating the effect of changes. In our game, after the main player chooses what they want to do, there was a huge blob that figured out if the user inputted the correct command, parsed the command, then started doing interactions on objects and rooms throughout the game. This blob was a dangerous blob, and it was only fixed by creating more smaller getter/setter functionality with the object classes and let them do their own data processing instead of our massive game loop blob. Now, we send things to be done to the object, and they return the result of their actions. The only way to fix the blob is to refactor, refactor, and refactor.

Poltergeists Software Development Antipattern are classes with very limited roles and effective life cycles. They often start processes for other objects. They have a very brief lifecycle that clutter the designs and create unnecessary abstractions. The refactored solution includes a reallocation of responsibilities to longer-lived objects that eliminate the Poltergeists. Our game had some poltergeists in our code with short temporary single operation classes from far off subclasses. We combined their functionality with larger super classes that were the bases of the small poltergeists. This was a better object-oriented way of setting up our code.