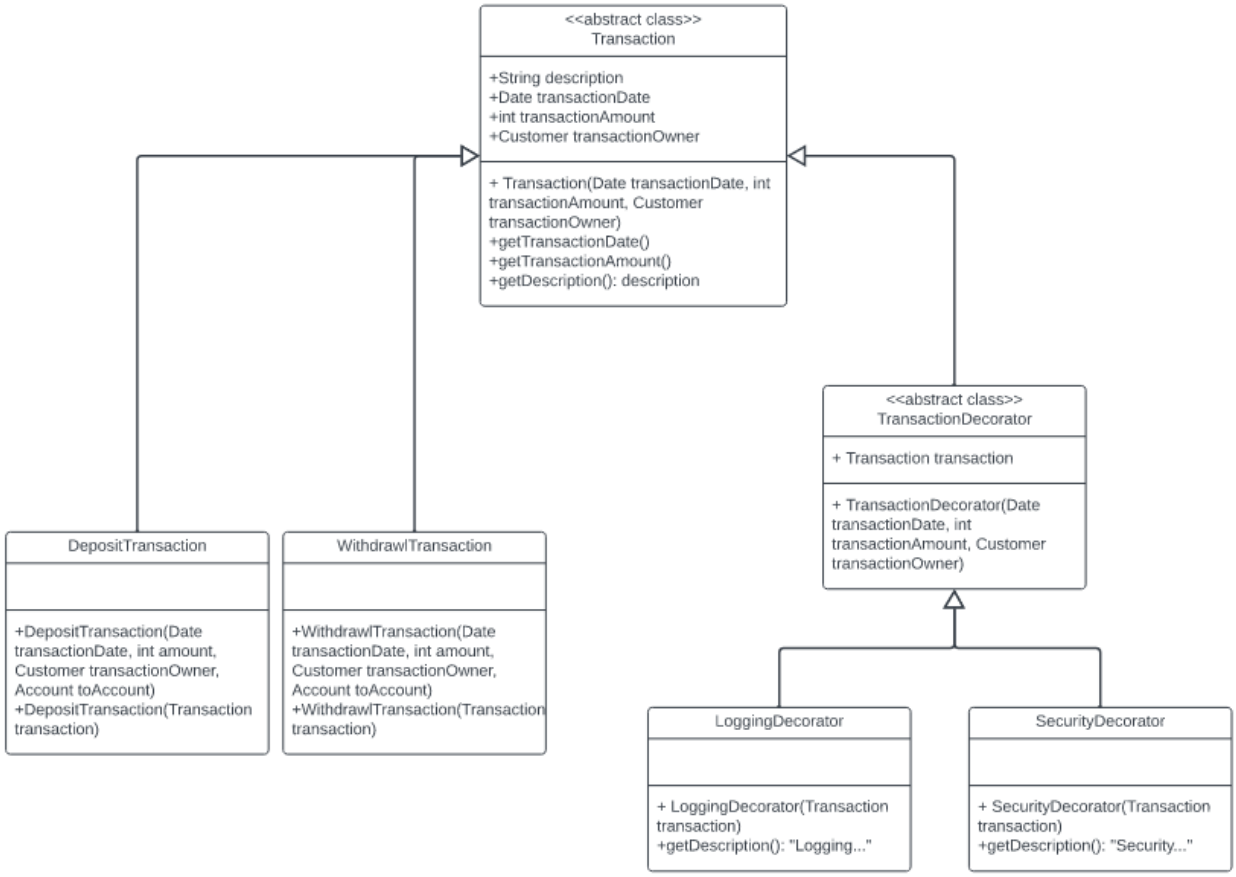
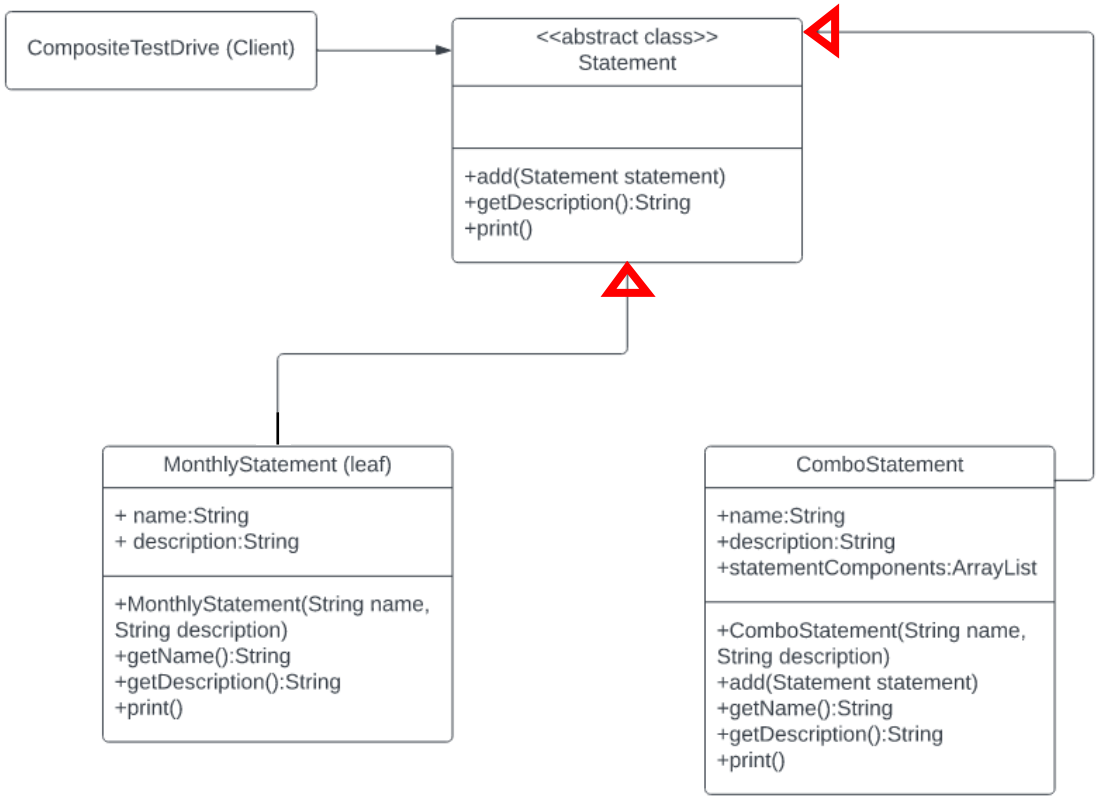
Marisa Paone HW 4 CS 665

For part 1 I chose the decorator method decorating the concrete classes of DepositTransaction and WithdrawlTransaction with a loggingDecorator and a SecurityDecorator. The decorator method allows you to “decorate” concrete classes with multiple decorators which adds functionality to concrete classes. The decorator method consists of a component, concrete components, a decorator, and concrete decorators. Each component can be used on its own or wrapped by a decorator. From my code, the DepositTransaction and WithdrawlTransaction was tested by using it by themselves (without decorators). The decorator implements the same interface or abstract class (Transaction) as the component they are going to decorate. The concrete decorators can extend the state of the component, and add new methods. Each decorator HAS-A component which means the decorator has an instance variable that holds a reference to a component.



For part 2 I chose the composite method which uses recursion to combine monthly statements (leaf class) into quarters, half yearly, and a full yearly statement. The composite pattern consists of a component, leaf, and composite class. The composite class also implements the lead related operations, but its role is to define behavior of the components having children, and to store child components. The leaf class defines the behavior for the elements in the composition. It implements the operations of the composite supports. The component class (statement) defines an interface (or in this case abstract class) for all objects in the composition: both the composite and the leaf nodes.



For part 3 the adapter pattern was used which converts the interface of a class into another interface that the client expects (due to incompatible interfaces). Here, the Iverify interface was created which has a verifyCustomer method that returns a boolean. The two “Adaptees” are ABCValidator and XYXChecker. These have their specific request methods being validate() and backgroundCheck() which has different parameters that they take in. The ABCAdapter class makes these two methods useful by “translating” them into the verifyCustomer method that the client can see with the IVerify interface. The inHouseVerifier has its own implementation of the verifyCustomer method. This pattern is useful when there is a mismatch in parameters and there is a need for “translation in code” or an “adapter”.