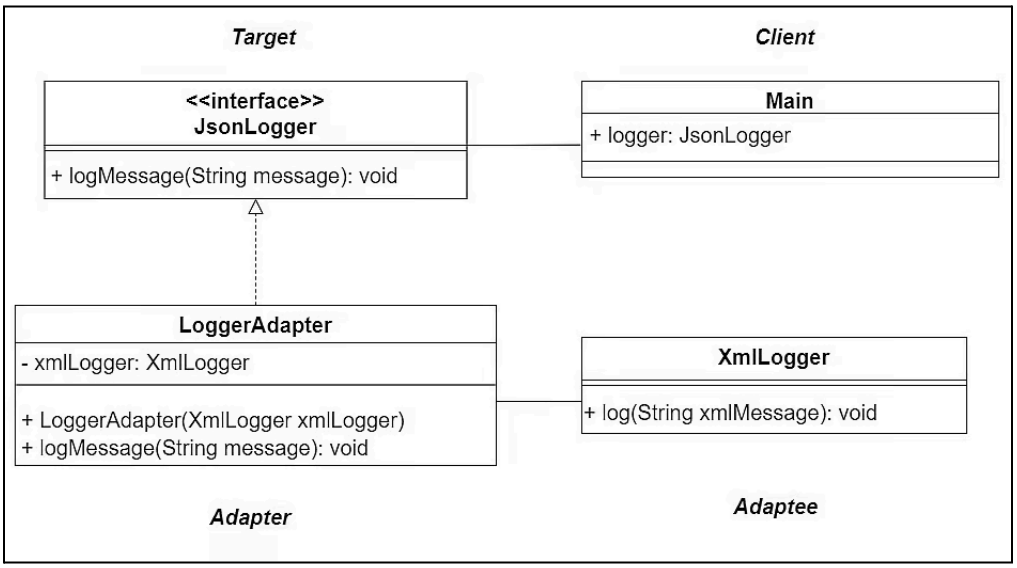
Adapter

Resolves compatibility problems
Motivation: Might not be able to merge codebases, need to bridge gap between systems.

Solution:

1. Target Interface: the interface that new system adheres to and what the client code is written against
2. Adaptee: existing component that does not conform to the Target Interface (Note: Functionality remains unaltered)
3. Adapter: the class bridging the gap between the Target and the Adaptee. It implements the Target interface, providing **compatibility** between the new and old systems. It translates calls from the Target Interface to the Adaptee
4. Client: works with objects of the Target type, unaware of Adapter or Adaptee, seems like it is simply interacting with Target Interface

Design + Implementation:
UML class diagram of our adapter pattern



Limitation + Pitfalls

1. Performance Overhead - extra layer of complexity, can add additional method calls and complex data conversion processes
2. Complexity and Maintainability - any updates to the legacy system must reflect in their respective adapters

Closing Notes:

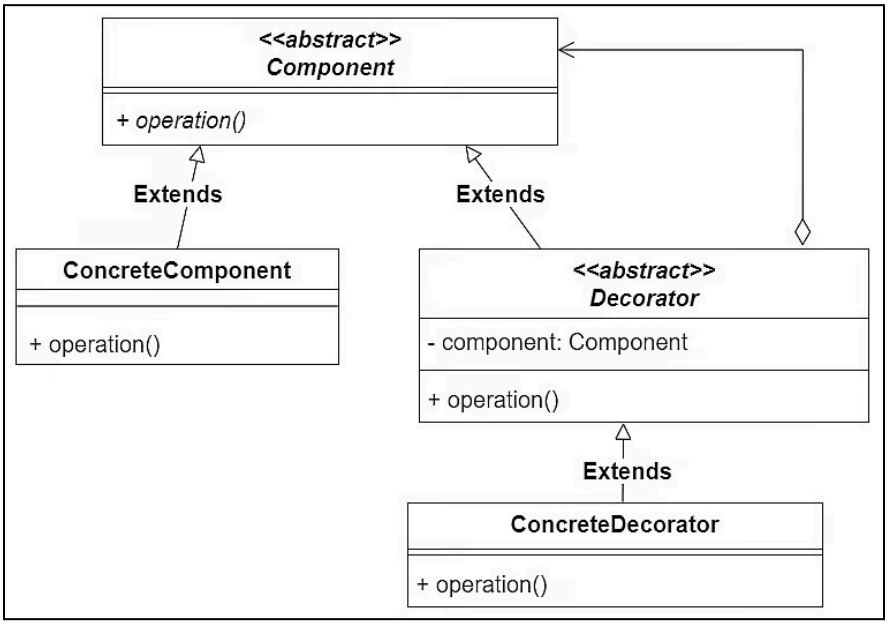
1. Promotes Single Responsibility Principle
2. Ensures Open/Closed Principle (open for extension, closed for modification)
3. Facilitates loose coupling

Decorator

Adding or changing behaviors (extending an object with a WRAPPER)
Motivation: use this when there is no possible/practical "God Class" containing all complexities and variants, and using inheritance is impractical
Solution: "favor composition over inheritance" and wrap with decorators and make decorators and components interchangeable

Design + Implementation:

1. Component (abstract class or interface) - abstract decorator and concrete component will implement and extend from this
2. Concrete Component - extends the component class + provides an implementation to all methods in the Component Class, altered by decorators
3. Decorator (abstract class) - extended by concrete decorators, consists of a has-a reference to the Component class/interface
4. Concrete Decorator - provides a concrete implementation to the base abstract decorators, can be added dynamically at runtime and they override base decorator's behavior



Limitation + Pitfalls

1. Might violate interface segregation principle - no client should be forced to depend on methods it does not use
2. Increases Complexity - Every single time we add a new decorator, we add a layer of abstraction which may be harder for new developers to understand

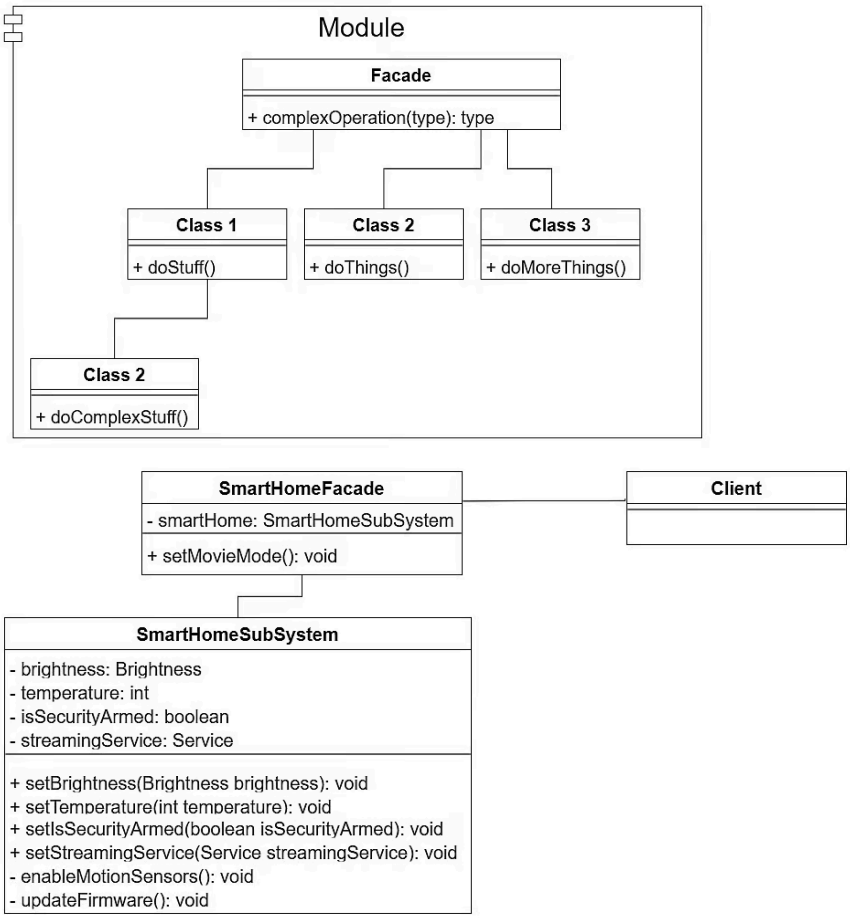
Closing Notes:

1. Open/Closed Principle
2. Composition over Inheritance
3. Single Responsibility Principle

Facade

Presents a complicated collection of classes in a simplified manner. This allows clients easy access to the functionality of these classes.
Motivation: Think of "Presets", we want to set several attributes at once with a single method
Solution: 1 class contains references to other classes and perform actions on those classes
Design + Implementation:

1. Subsystem Classes - classes the facade aims to simplify
2. Facade - the gateway offering a unified higher-level interface of the diverse interfaces within a subsystem
3. Client - interacts with the subsystem primarily through the facade, gaining streamlined and intuitive experience. Clients can focus tasks without complexities



Limitation + Pitfalls

1. Oversimplification
2. Maintenance Challenges

Closing Notes:

1. Abstraction of Complexity
2. Decoupling