SWE316 (Software Design and Construction)

Homework # 2

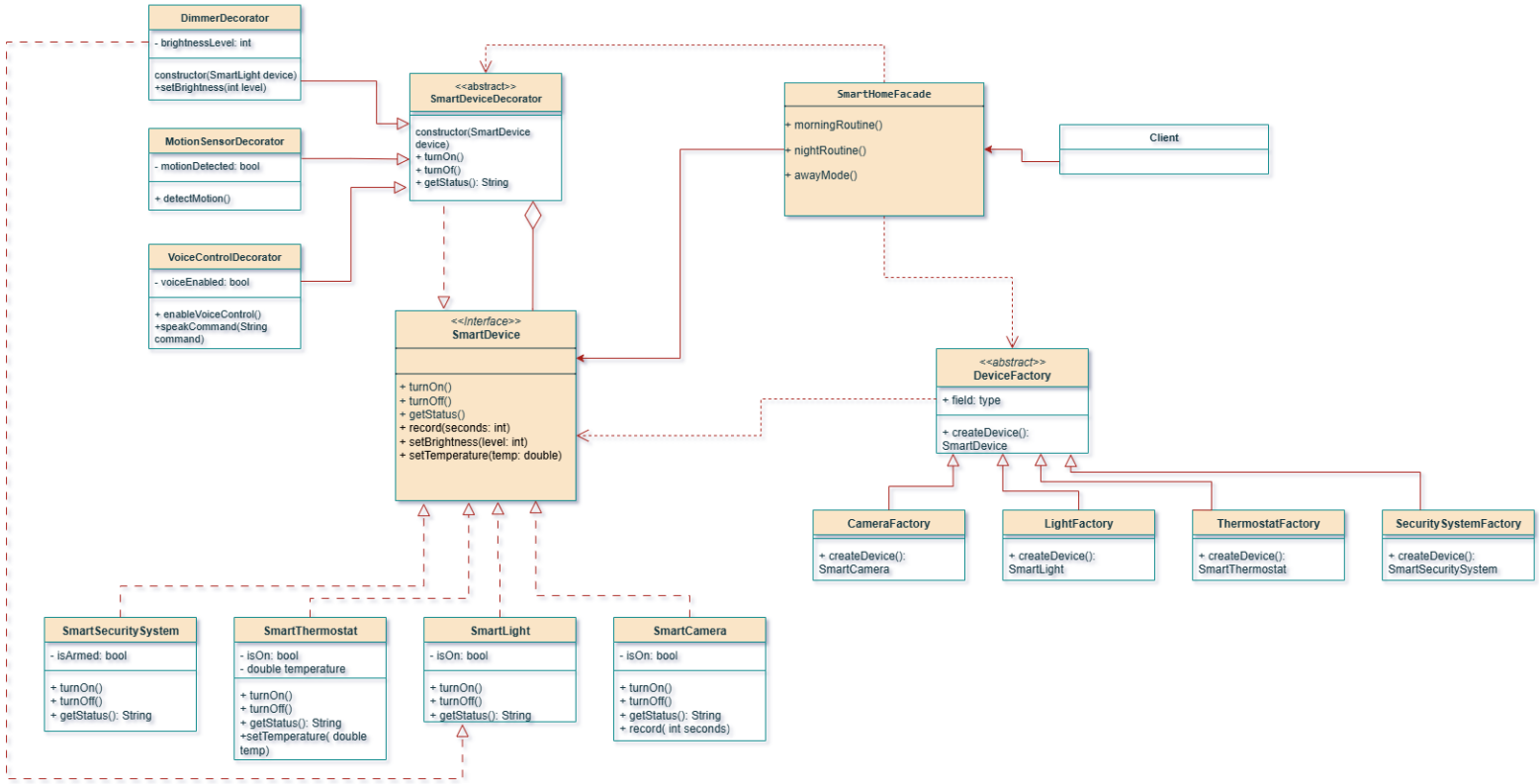
Date of submission: 12/9/2025

Name: Ibrahim Alshayea

ID: 202176470

The UML Diagram:

For High Resolution View, Click Here



3.1 Design Rationale

1. Design Pattern: Factory Method

1.1 Problem Solved

The Client code had to be hardcoded with the exact class names of every device it controlled. If the system requirements changed. For example, replacing the standard SmartCamera with a new camera with more features, say SmartCamera2, the developer would be forced to open the Client source code, find every instance of new SmartCamera(), and manually rewrite it with SmartCamera2() instead of just calling the factory method.

1.2 Why This Pattern Fits

The Factory Method defines an interface for creating an object (DeviceFactory) but lets subclasses (LightFactory, CameraFactory) decide which class to instantiate. This allows the Client to work with the abstract concept of a Factory rather than specific device constructors, so in our last example, the client will just call the factory method and the factory subclass will decide to return SamrtCamera2.

1.3 Implementation Analysis

- Abstraction: I defined a high-level DeviceFactory abstract class that declares the createDevice() method. The SmartHomeFacade interacts only with this abstraction, never needing to know which specific factory subclass is currently active.
- Dependency Management: The Client no longer depends on concrete classes like SmartLight. Instead, both the Client and SmartLight depend on the SmartDevice interface, which can be created using the factory.

1.4 SOLID Principle Demonstration: Open/Closed Principle (OCP)

- Open for Extension: We can extend the system by adding a new class, SmartLocks, and a corresponding LockFactory.
- Closed for Modification: The SmartHomeFacade (which uses the factory) does not need to be recompiled or modified to support this new SmartLocks device. Its logic remains generic: calling factory.createDevice(), regardless of what that device actually is.

2. Design Pattern: Decorator

2.1 Problem Solved

Static inheritance fixes the features of an object at compile-time. If a user purchased a standard SmartLight and later decided to install a dimmer switch, the software could not represent this change without destroying the old object and creating a new DimmableLight object. Moreover, this way leads to class exploding.

2.2 Why This Pattern Fits

It treats features (dimming, motion detection) as separate layers. We can wrap a SmartLight in a DimmerDecorator, and then wrap that entire bundle in a VoiceControlDecorator. This is the best way to handle the business logic instead of just creating many classes each having combination of features.

2.3 Implementation Analysis

- Abstraction & Interface Consistency: I created an abstract SmartDeviceDecorator that implements the SmartDevice interface. To ensure true polymorphism, I promoted specific methods like setBrightness() and record() to the SmartDevice interface as default methods. This ensures that a decorated object (which is just a generic SmartDevice to the outside world) can still receive these commands without the client needing to cast it.
- Composition: The core of our implementation is the protected SmartDevice device field inside the decorator. This field holds the reference to the next item in the chain. All calls are forwarded to this wrapped object, preserving the original behavior while adding new decorations.
- Dependency Management: I enforced logical safety in the implementation of DimmerDecorator. While it is a generic decorator, its constructor public DimmerDecorator(SmartLight device) enforces that it can only wrap a Light. This prevents runtime errors (like trying to dim a camera) while still treating the final output as a generic SmartDevice.

2.4 SOLID Principle Demonstration: Open/Closed Principle (OCP)

- Open for Extension: We can add infinite new features (e.g., WiFiEnablerDecorator, ColorChangerDecorator) simply by creating new decorator classes.
- Closed for Modification: We explicitly do not modify the source code of the SmartLight or SmartCamera classes to add these features. The core device logic remains pristine and untested, ensuring high stability.

3. Design Pattern: Facade

3.1 Problem Solved

The legacy Client code violated the Principle of Least Knowledge. To execute a simple user goal like "Start Morning Routine," the Client had to know about Light, Thermostat, and SecuritySystem, instantiate them, and call distinct methods in a precise order..

3.2 Why This Pattern Fits

It The Facade Pattern fits because it provides a simplified interface to a complex subsystem. It creates a dedicated place (SmartHomeFacade) for the high-level business logic (Routines). The Client simply use the façade api and the façade handles everything else.

3.3 Implementation Analysis (Connecting Theory to Code)

- Dependency Management: The Facade isolates the dependencies. The complex imports (Decorators.*, Factories.*) are contained solely within the SmartHomeFacade.java file. The Main class (Client) remains clean, importing only the Facade.

3.4 SOLID Principle Demonstration: Single Responsibility Principle (SRP)

- Client's Responsibility: Initiation and User Intent. The Client only worries about when to trigger a routine.
- Facade's Responsibility: Subsystem Orchestration. The Facade worries about how to execute the routine.

3.2 Trade-Offs and Alternatives

1. Alternative to Factory Method: The Simple Factory Pattern

- Description of Change:

  Instead of implementing the polymorphic Factory Method (with an abstract DeviceFactory and subclasses), we could have used a Simple Factory. This would involve a single class, SimpleDeviceFactory, with a static method that accepts a string parameter like "light" and uses a large if-else block to instantiate the correct object.

- New Problems and Constraints:

  Violation of Open/Closed Principle (OCP): Every time a new device type is introduced, the SimpleDeviceFactory source code must be opened and modified to add a new if-else statement.

- Justification for Final Design (Robustness):

  The Factory Method was chosen because it decouples the creator from the concrete products. By using inheritance (LightFactory extends DeviceFactory), we ensure that adding a new device does not change the source code but add to it. We simply create a new factory class without touching the existing DeviceFactory or SmartHomeFacade code. This distributed responsibility makes the system significantly more robust and scalable for a growing smart home ecosystem.

2. Alternative to Decorator: Static Inheritance

- Description of Change:

Instead of using object composition to add features like dimming or motion detection, we could have used Static Inheritance. This would involve creating explicit subclasses for every feature combination, such as DimmableLight, MotionSensingCamera, or complex combinations like VoiceControlledDimmableLight.

- New Problems and Constraints:

  - Class Explosion: Using inheritance to add new features requires creating a new class that combines these features, this approach leads to a combinatorial explosion of classes.

    - Static Rigidity: Inheritance is defined at compile-time. If a user purchases a standard SmartLight and later decides to screw it into a dimmer socket, the inheritance model fails. We cannot convert an instance of SmartLight into DimmableLight at runtime; we would have to destroy the old object and create a new one, losing its current state.

- Justification for Final Design (Robustness):

The Decorator Pattern is superior because it allows for runtime extension. We can take an existing SmartLight object and wrap it in a DimmerDecorator instantly. Furthermore, it adheres to the Single Responsibility Principle; a DimmerDecorator contains only the logic for dimming, whereas a VoiceControlledDimmableLight subclass would inevitably duplicate logic found in other voice-controlled classes.

3. Alternative to Facade: Direct Client Control

- Description of Change:

We could have omitted the SmartHomeFacade entirely and allowed the Client to directly instantiate factories, create devices, apply decorators, and call specific methods. The Client would manually execute the logic

- 2. New Problems and Constraints:
    - High Coupling: The Client becomes tightly coupled to the entire subsystem.
    - Loss of Encapsulation: The internal structure of the smart home (factories, decorators) is exposed to the outside world, making it difficult to refactor the internal system without breaking the Client code.
- 3. Justification for Final Design (Robustness):

The Facade Pattern was essential to strictly enforce the Law of Demeter. The Client interacts with a single entry point (homeSystem.morningRoutine()). This decouples the User Intent from the System Implementation. It ensures that the Client code remains stable even if we completely rewrite how the devices operate internally, providing maximum maintainability.

3.3 Predict and Explain

1. What happens if a new decorator (EnergySaverDecorator) wraps the Thermostat?

Prediction: The system will accommodate this new feature seamlessly without breaking any existing code. The new decorator will successfully intercept temperature commands and enforce energy-saving logic before passing them to the actual thermostat.

- Implementation: We would create a new class EnergySaverDecorator extending SmartDeviceDecorator. We would override the setTemperature(double temp) method. Inside this method, we would add logic to check if the requested temperature is within a specific range, like (20°C–24°C). If it is outside this range, the decorator could clamp the value or ignore the request before calling super.setTemperature(temp).
- Impact: Because the SmartHomeFacade interacts with the generic SmartDevice interface, it does not care that the thermostat is now wrapped. The SmartThermostat class itself remains untouched. We have successfully extended the behavior of the thermostat without modifying its source code.

2. Predict: If the SmartHomeFacade directly instantiates concrete devices instead of using the factory?

Prediction: The SmartHomeFacade will become tightly coupled to the specific implementations of the devices.

Explanation: This violates the Open/Closed Principle (OCP).

- The Violation: By writing new SmartLight() directly inside the Facade, the class is not closed for modification. If I later want to use a different device instead of SmartLight, I'll be forced to open the Facade class and modify the code.
- The Risk: Every time you modify existing, working code (the Facade) just to add a new feature, you risk breaking it. Using the Factory allows you to extend the system (by creating a new IoTLightFactory) without ever touching or breaking the Facade code.

3. Predict: Suppose we remove the Facade entirely and let the client call devices individually.

Prediction: The system will suffer from High Coupling and Low Cohesion.

Explanation:

- High Coupling: Without the Facade, the Client must know the details of every single class in the subsystem (LightFactory, SmartLight, DimmerDecorator, Thermostat, etc.). If I change anything inside the subsystem (like renaming a method in SmartLight), the Client code breaks.
- Low Cohesion: By forcing the client class to also manage the complex steps of the the system, like morning routine, the client is doing two unrelated jobs. This low Cohesion makes the code hard to maintain. The Facade fixes this by keeping the Client focused and simple.