1. Introduction

The Smart Home Controller application is a Java-based simulation that allows users to manage lights, thermostats, doors and security systems both individually and by room. It maintains a detailed log of every action—such as "Living Room Light turned ON"—and supports automation modes like Night Mode, which can lock doors and switch off all lights in one command. This report explains how five classic design patterns were woven into the system to achieve flexibility, maintainability and extensibility, summarises the central classes, and reflects on the principal development challenges encountered.

2. Design Patterns Used

The system employs the Singleton, Factory, Observer, Decorator and Strategy patterns to ensure a clean architecture and easy evolution of features. The Singleton pattern governs the `SmartHomeController` class, guaranteeing that only one controller instance exists; this single instance coordinates all device interactions and centralises logging. The Factory pattern handles device creation through a `DeviceFactory` interface and its concrete implementation, which encapsulates the logic for instantiating Lights, Thermostats, Doors and other device types based on a simple enumeration; this approach decouples client code from the details of object construction and makes it trivial to introduce new device kinds in the future. Whenever a device's state changes—say, a thermostat is turned off or a security camera is activated—the Observer pattern ensures that every registered `SystemObserver` (for example, a logger or console display component) is notified without the controller needing to know the specifics of each observer. The Decorator pattern wraps core `Device` objects in dynamic feature-adding classes such as motion-sensor or timer decorators; this allows new behaviours to be layered onto any device instance at runtime without modifying the original classes or creating a combinatorial explosion of subclasses. Finally, the Strategy pattern encapsulates automation routines—Night Mode, Morning Mode, Vacation Mode—in separate classes implementing a

common `AutomationStrategy` interface; the controller can switch strategies on the fly, invoking each strategy's `execute()` method to carry out complex sequences of device commands without embedding conditional logic in the controller itself.

3. Key Classes

At the heart of the application lies the `SmartHomeController`, implemented as a Singleton, which provides methods for turning devices on and off, registering and notifying observers, and applying automation strategies. Devices themselves share an abstract `Device` superclass that defines a uniform interface—methods like `turnOn()` and `turnOff()`—and concrete subclasses (`Light`, `Thermostat`, `Door`, `SecurityCamera`) implement device-specific behaviour. The `DeviceFactory` and its `ConcreteDeviceFactory` implementation centralise object creation, while the `SystemObserver` interface and its implementations (`SystemLogger`, `ConsoleDisplayObserver`) handle event reporting. Decorators derive from an abstract `DeviceDecorator` class that holds a reference to the wrapped `Device` and overrides core methods to add functionality before or after delegating calls. Automation strategies each implement `AutomationStrategy`, defining an `execute(controller)` method that orchestrates multi-device actions for a particular mode.

4. Challenges Faced

Throughout development, striking the right balance between flexibility and complexity was the foremost challenge. Leveraging five design patterns made the architecture highly modular and extensible, but without careful planning of class relationships, it would have become unwieldy; I mapped out dependencies in UML and iteratively refined package boundaries to keep coupling low. Implementing the Decorator pattern introduced its difficulties: each decorator had to faithfully forward every method of the wrapped `Device` and inject new behaviour, and missing even a single method led to silent inconsistencies that were only caught through testing. Observer management likewise required special attention—naïvely broadcasting every state

change resulted in redundant notifications and occasional circular loops, so I augmented my notification logic to tag events with their source and suppress duplicates. Finally, integrating patterns end-to-end demanded close coordination—for example, when a Strategy invoked a series of device commands, those actions needed to trigger Observer updates without breaking the Decorator chains; resolving these sequencing issues meant adding explicit hooks in the controller to orchestrate pattern interactions.

5. Conclusion

The Smart Home Controller successfully demonstrates how the Singleton, Factory, Observer, Decorator and Strategy patterns can be combined to produce a maintainable, extensible application. Each pattern addresses a distinct design concern—centralised control, dynamic object creation, decoupled event notification, runtime behaviour extension and interchangeable automation routines—together; they form a cohesive architecture that meets the project's functional requirements. Although integrating multiple patterns introduced complexity and required careful testing and refinement, the resulting system is robust and adaptable: new device types, decorators or automation modes can be added with minimal impact on existing code. This project thus illustrates the power of design patterns to structure complex software in a clear, scalable way.

[Refer to the README.md file in the source code for more technical details]