Final Project Report

1. Name of project and names of all team members

Developer: Samiksha Patil

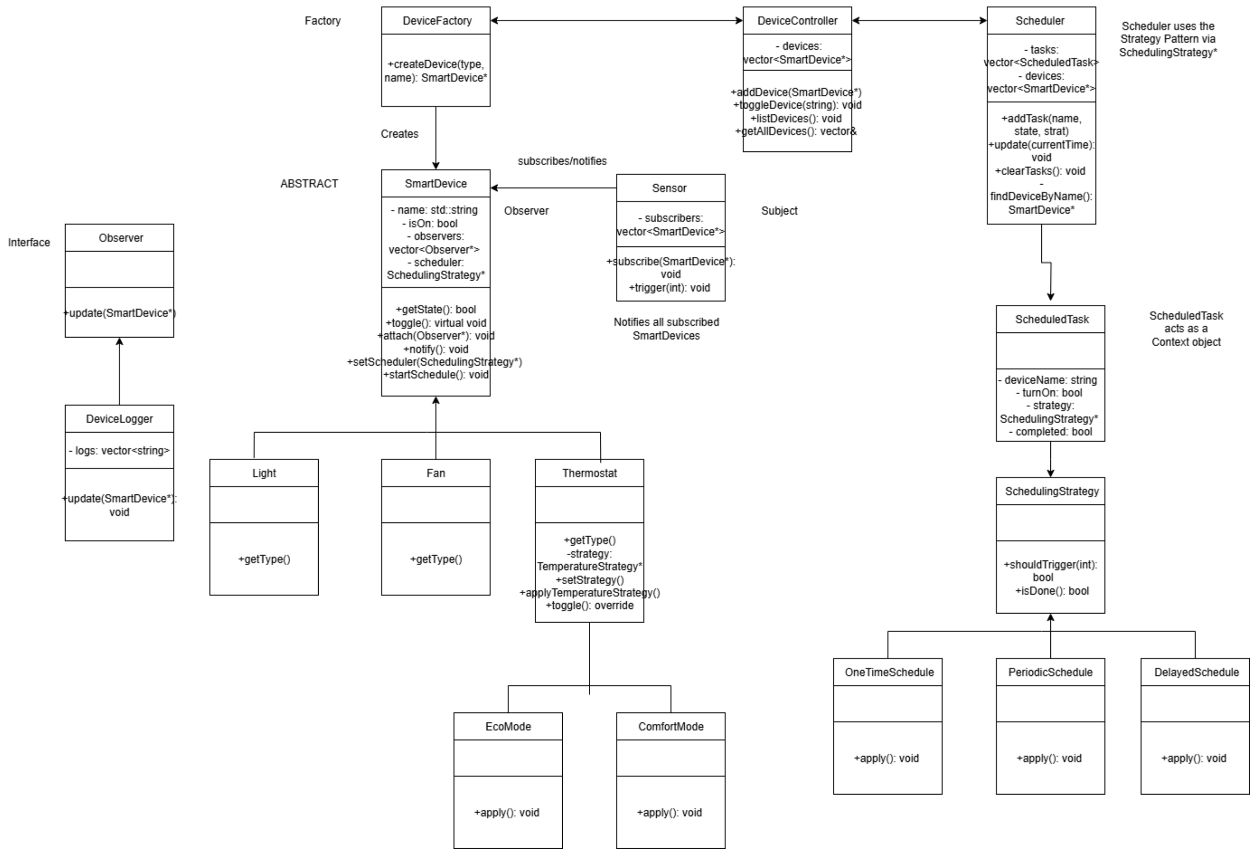
Project Title: SmartHomeSim: C++ Simulator for Smart Device Control

2. Final State of System Statement

The final state of SmartHomeSim system reflects a fully functional, object-oriented simulation of smart home device control. All core features proposed in Project 5 were successfully implemented, including device creation via a factory pattern, sensor-based triggering, multiple scheduling strategies (One-Time, Delayed, Periodic), and command-line user interactions to add, control, and simulate devices. Enhancements from Project 6 were also completed: I integrated the Observer pattern to allow sensors to notify subscribed devices, Strategy pattern for runtime switching of thermostat behavior (Eco vs Comfort mode), and polished the CLI for improved usability with a structured menu. Project 7 added several advanced features, including support for device action scheduling, persistent device logs via the Logger observer, and JUnit-style unit tests using assertions to verify major components.

While most features were implemented, a few optional ideas such as a GUI-based interface or user account management were deferred to prioritize stability, modularity, and clean code. Compared to the initial designs in Project 5, the system evolved significantly in modularity and design clarity, adopting more design patterns and focusing on extensibility and real-time simulation capabilities. Overall, the project successfully demonstrates strong software architecture principles in a simulated smart environment.

Final UML Class Diagram -



Final class diagram incorporates a well-structured, extensible design using multiple object-oriented design patterns:

- Factory Pattern: DeviceFactory creates Light, Fan, and Thermostat dynamically.
- Observer Pattern: Sensor notifies subscribed SmartDevices on trigger events; DeviceLogger logs all state changes.
- Strategy Pattern (x2):

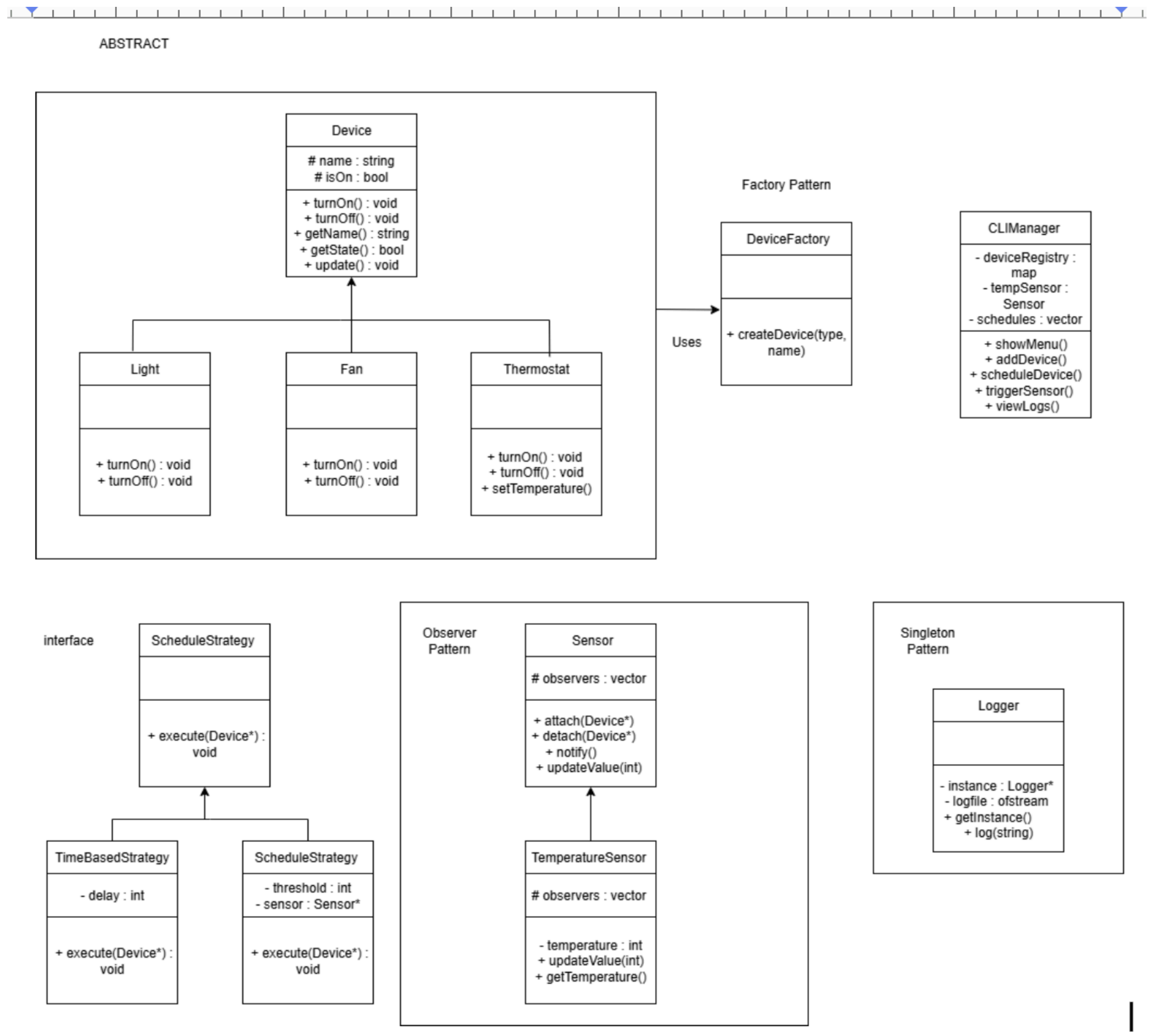
  Thermostat delegates behavior to TemperatureStrategy (EcoMode/ComfortMode).

  SmartDevice schedules are governed by a SchedulingStrategy (OneTime, Periodic, Delayed).
- Command-Like Scheduling: Scheduler manages scheduled tasks using ScheduledTask context objects.

- Abstraction: Core abstract classes (SmartDevice, SchedulingStrategy) allow polymorphic behavior.

This final diagram reflects the integration of all key system features and a clear separation of responsibilities, supporting open-closed and single-responsibility principles.

Class diagram submitted in Project 5 :



Comparison with Project 5 Class Diagram :

In the Project 5 class diagram, the foundation of the system was established with a basic object-oriented structure. The Device class was the core abstraction, with Light, Fan, and

Thermostat as concrete subclasses. Each device had simple methods like turnOn(), turnOff(), and update(). However, as the system evolved, this structure was significantly enhanced in the final version. The Device class was renamed to SmartDevice, and its responsibilities were expanded to support dynamic scheduling and observer subscriptions. This change allowed devices to respond to both time-based schedules and sensor inputs, increasing system flexibility.

The scheduling system also matured. In Project 5, scheduling was handled through a simple interface (ScheduleStrategy) with two concrete implementations. In the final version, this was replaced by a fully realized SchedulingStrategy hierarchy, coordinated through a new Scheduler class and ScheduledTask context objects. This shift improved modularity and allowed multiple scheduling types (One-Time, Periodic, Delayed) to be managed uniformly.

Sensor logic saw a similar refinement. Initially, a separate TemperatureSensor subclass handled updates and thresholds. In the final design, all sensor logic was unified into a single Sensor class, which can notify subscribed devices through a clean observer pattern. This change simplified the sensor system while making it more extensible.

The Logger class in Project 5 followed the Singleton Pattern, but in the final system it was replaced by DeviceLogger, a proper observer that subscribes to devices and logs their actions automatically. This change not only removed the need for a global logger instance but also integrated logging more naturally into the observer framework.

Additionally, Thermostat behavior evolved from direct temperature setting to supporting runtime behavior changes via the Strategy Pattern. This was achieved by introducing a TemperatureStrategy interface with EcoMode and ComfortMode implementations, giving thermostats dynamic, swappable behavior.

In summary, the system design matured significantly from Project 5 to the final version.

4. Third-Party code vs. Original code Statement :

No external libraries or frameworks were used in this project. However, occasional reference was made to online tutorials and documentation for conceptual clarity, particularly for:

General C++ syntax and STL usage: cplusplus.com and geeksforgeeks

Design pattern overviews: Refactoring Guru

Additionally, OpenAI's ChatGPT was used as a learning tool for brainstorming class structure ideas, clarifying design pattern implementation details, and reviewing/refining written documentation (e.g., this statement). All actual code was implemented and debugged independently.

No third-party code was directly copied or reused in any part of the implementation.

5. Statement on the OOAD process for your overall Semester Project :

Throughout the development of the SmartHomeSim system, I experienced several key elements of the Object-Oriented Analysis and Design (OOAD) process that shaped both design decisions and overall project growth.

- Challenges in Pattern Integration:
  A significant challenge I faced was integrating multiple design patterns without overcomplicating the system. For example, implementing both the Strategy Pattern for thermostat behavior and the Observer Pattern for sensor-driven control required careful coordination of dependencies.

- Trade-Off Between Simulation Accuracy and Design Scope:
  At one point, I considered implementing an automatic timer mechanism to simulate real-time behavior without requiring manual ticks. However, this approach would have required introducing multithreading and concurrency, which - while technically feasible - would have added complexity beyond the scope of the project and its focus on object-oriented design. Ultimately, I decided to keep the time advancement manual to stay aligned with the course objectives and maintain the integrity of the OOP-focused simulation. This decision reinforced the importance of aligning technical choices with design goals and project constraints.

- Importance of Iterative Design:
  One of the most positive aspects of the process was how iterative refinement improved the system over time. The initial class diagram in Project 5 was functional but rigid. Through Projects 6 and 7, I continuously revisited the design, introducing clearer abstractions, splitting responsibilities (e.g., separating scheduling from device control), and integrating design patterns that improved modularity and scalability.

Overall, the OOAD process taught me the value of planning, adapting, and grounding the architecture in both sound design principles and practical usability.