OOP Project

Smart Home System

A modular Java-based smart home system with automation, energy management, and security

Presented By:
Islem Smiai
Yesmine Srairi
Yassine Antit
Yasmine Akik
Houssine Khlif
Yasmine Yazidi

Supervisor
DR.Ameni Azzouz

Tunis Business Scool

Why Object-Oriented Design

Managing Complexity

Smart homes bring together many interconnected components such as lights, thermostats, sensors, and alarms. Object-Oriented Programming allows us to organize this complexity by modeling each real-world component as a separate object with clear responsibilities.

Modularity & Scalability

As smart homes evolve, new devices, rules, and features are constantly added.
OOP enables a modular design where components can be extended or replaced without affecting the rest of the system, making the solution scalable and future-proof.

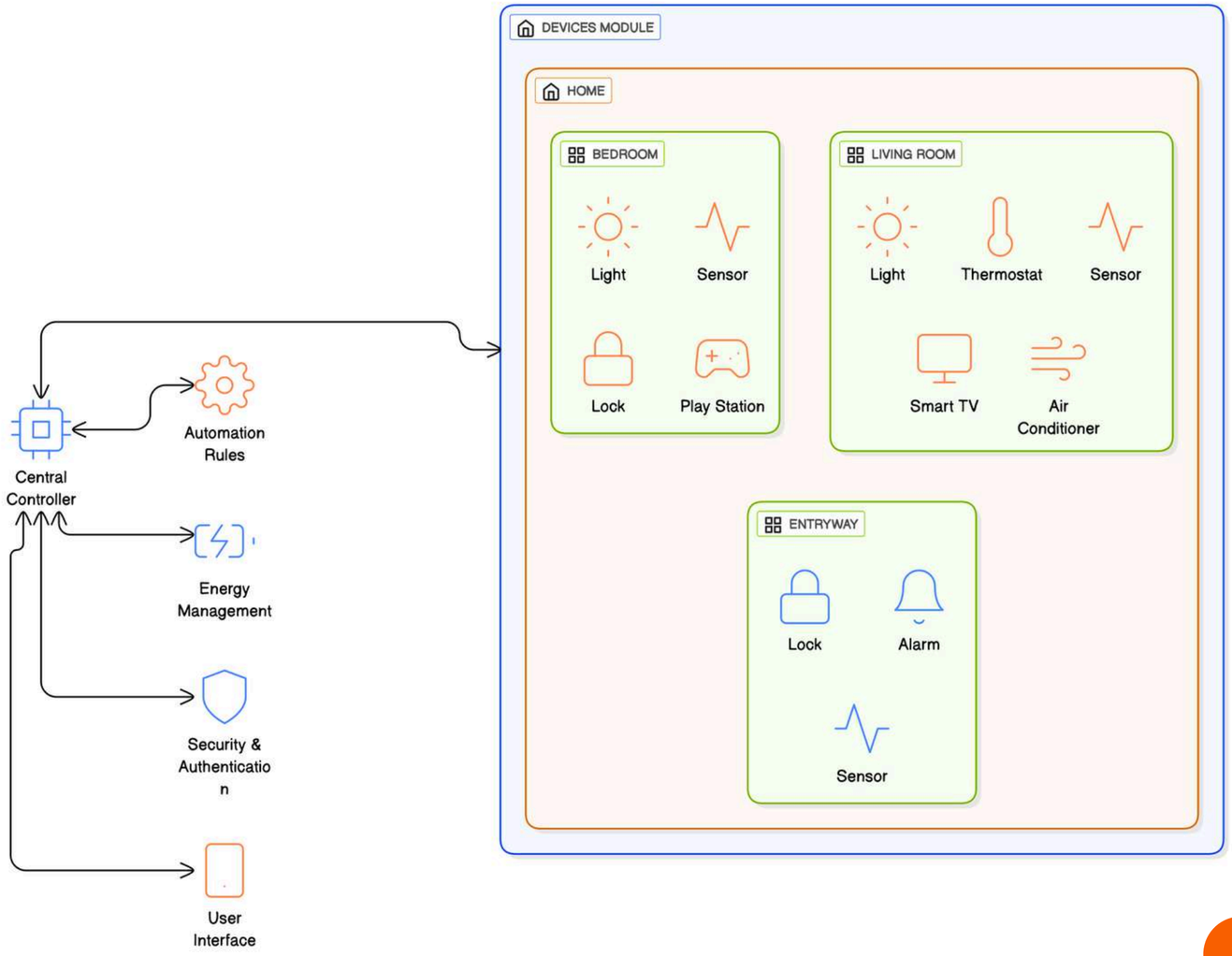
Real-Time Coordination

Smart home systems require devices, users, and automation rules to interact in real time.
With OOP, shared behaviors, abstraction, and polymorphism allow different components to communicate seamlessly, enabling responsive control and intelligent automation.

Our Project

What our project provides:

- A modular smart home system modeled using OOP
- Centralized control of connected devices
- Automation rules for intelligent behavior
- Integrated energy and security management
- A functional control UI and a visual simulation UI



Problem Modeling

The smart home is modeled as a set of objects representing devices, users, and system managers.

Energy-aware automation:

The system monitors energy sources and adapts device behavior based on battery levels.

Role-based access control:

Users have permissions defining which devices they can control.

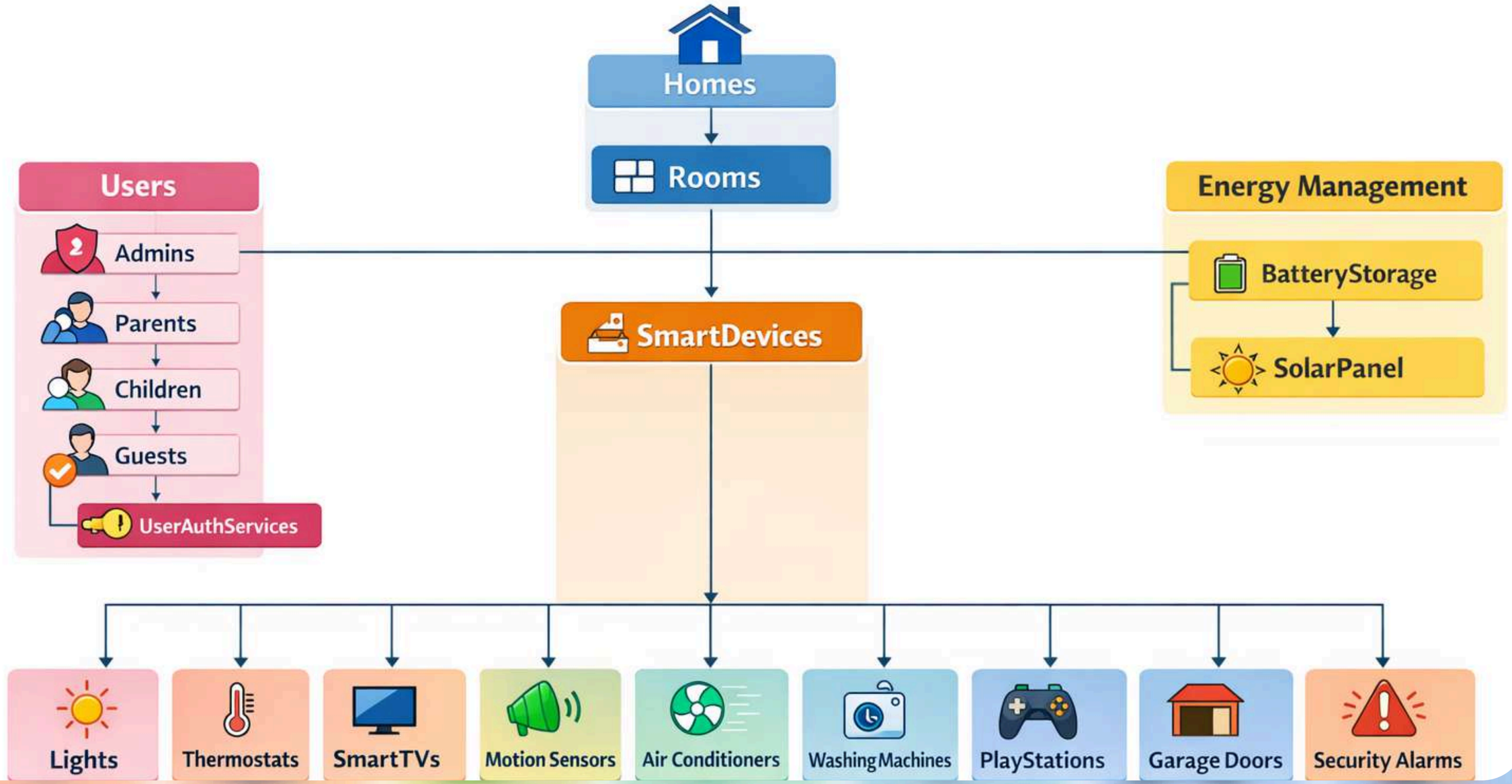
Automation rules:

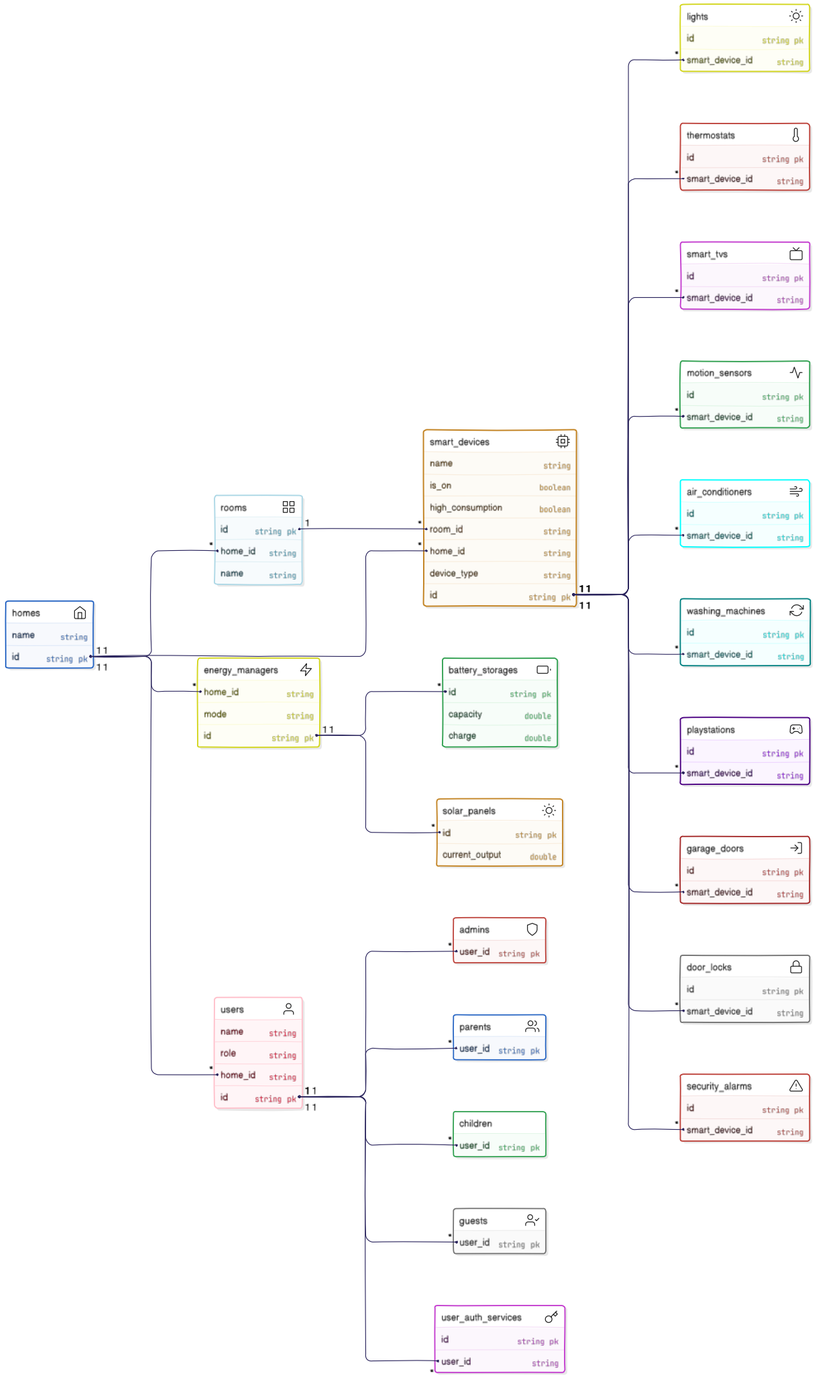
Encapsulated as objects that trigger actions under specific conditions.

Security:

The system ensures safe operations through authentication and device protection,

Smart Home Automation Data Model



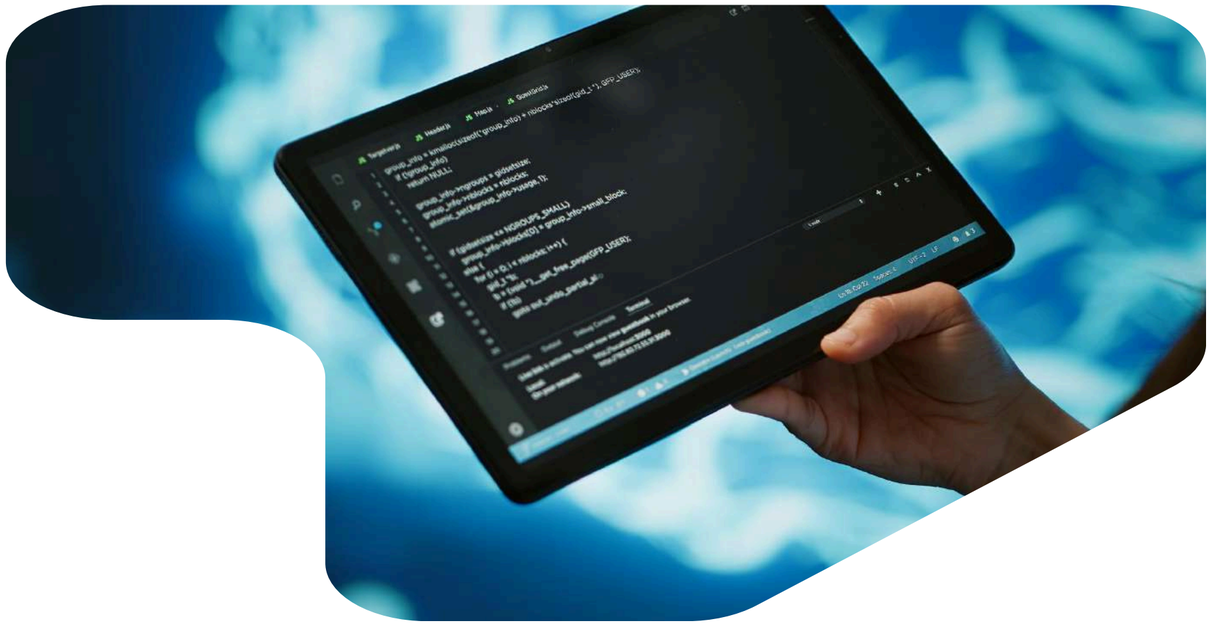


Class Diagram

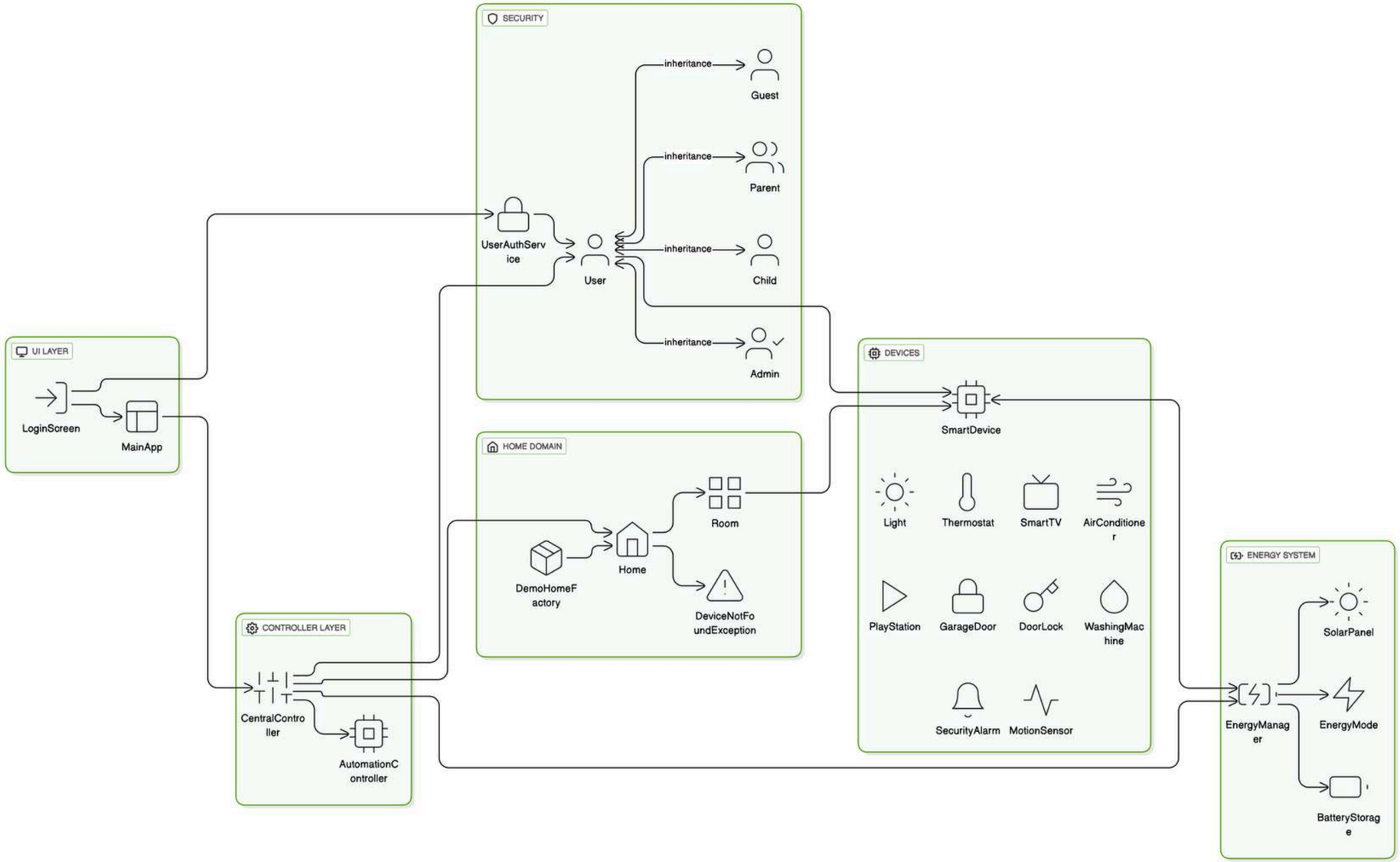
- Our system is structured modularly to separate concerns and simplify maintenance.
- Devices, energy management, home modeling, automation, security, and UI are handled in distinct components.
- Automation rules and energy optimization logic are centralized for consistent behavior.
- Object-oriented design allows easy addition of new devices, user roles, or energy sources.
- Ensures safe, efficient, and extendable smart-home simulation.

Our code is organized modularly for clarity, maintainability, and scalability.

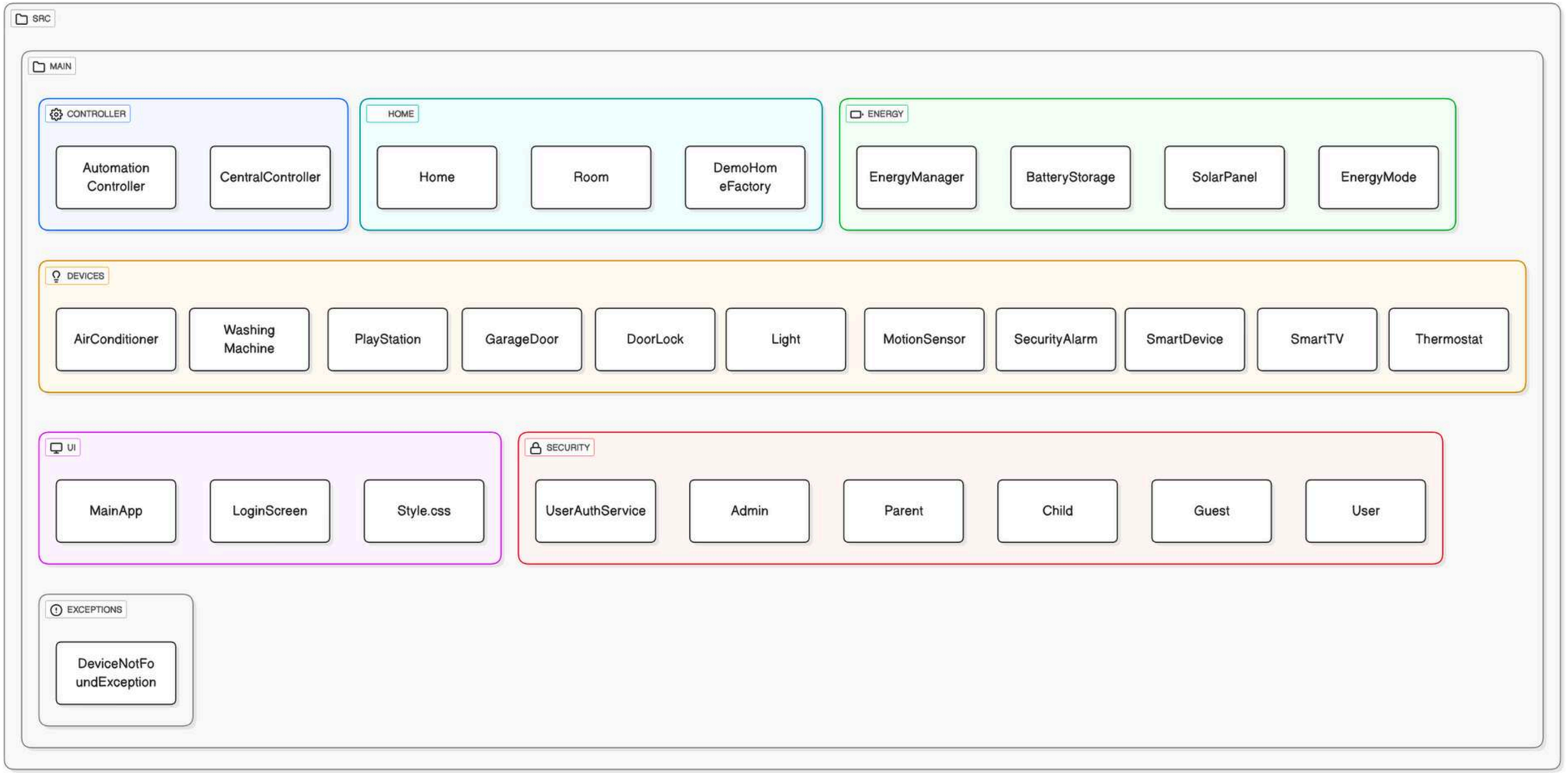
System Architecture



Our System's Architecture

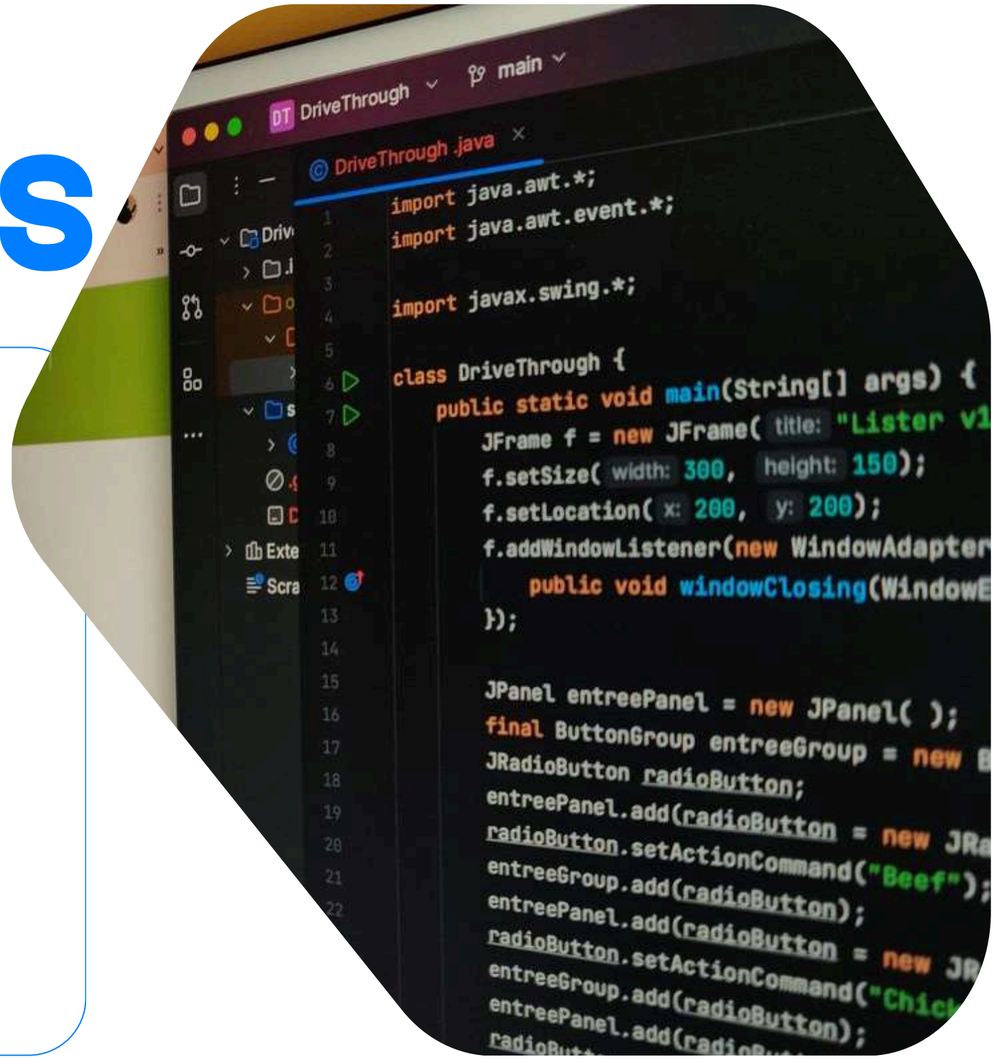


Code Organization Overview



OOP Principles

- Encapsulation
- Abstraction
- Inheritance
- Polymorphism



Encapsulation

Data protection & Controls access :

The internal details of our classes are hidden

Each class has private attributes

```
private double capacity;
private double charge;
private String name
private List<Room> rooms
```

Only necessary features are exposed through public methods

getters and setters

```
public void getChargePercent() {
    return (charge / capacity)* 100.0;
}
public void setCharge(double charge) {
...
}
```

Abstraction

Hiding complexity by exposing only relevant functionality

Abstract classes

```
public abstract class
SmartDevice {
...
}
public abstract class User {
...
}
```

Interfaces

```
public interface
AutomationController {
...
}
```

Inheritance

parent-child relationships to reduce redundancy

Abstract classes

- Devices inherit from **SmartDevice**
- Users inherits from **User**
- Exceptions inherit from **Exception**

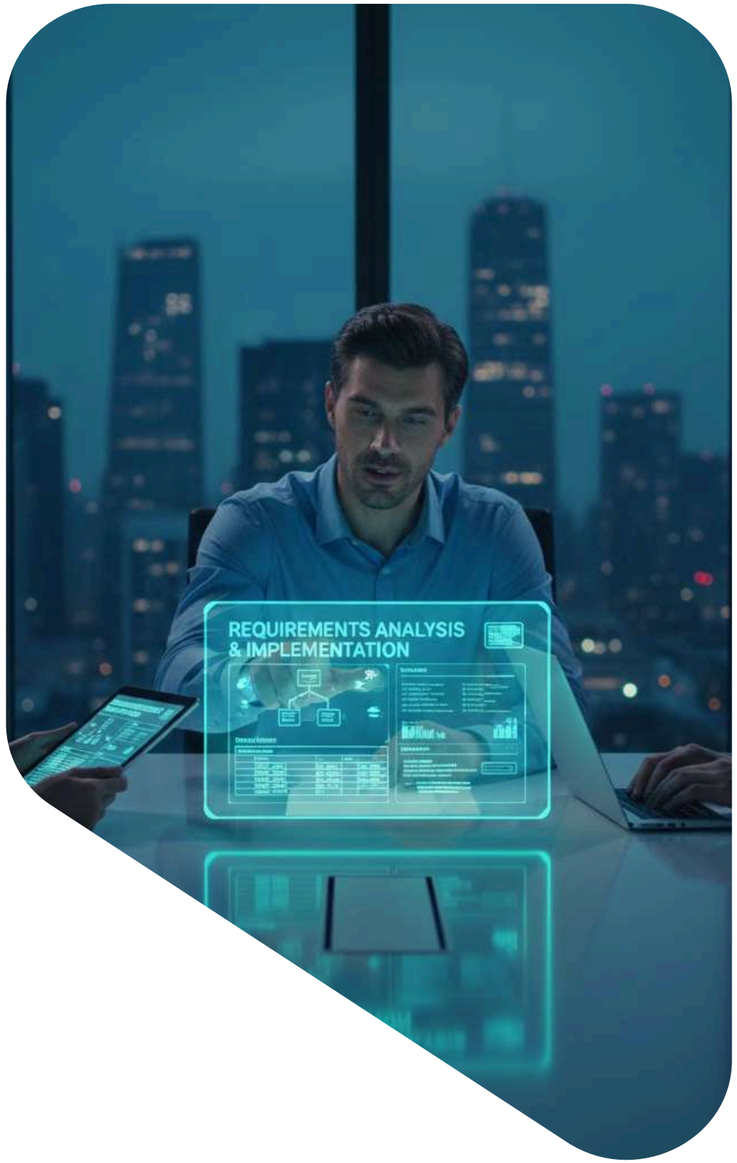
```
public class Parent extends
User {
...
}
public class GarageDoor
extends SmartDevice{
...
}
```

Polymorphism

Allows objects to take many forms while performing actions differently

```
public abstract class
SmartDevice {

    public abstract double
getPowerUsage();

}
```

```
@override
public double
getPowerUsage() {
    return isOn ? 3.0 :
0.0;
}
```



Requirements Analysis & Implementation

Project Requirements Coverage Overview

- Abstract SmartDevice Class with at least 3 subclasses
- Interfaces
- Home Structure & Composition
- Central Controller & Automation Rules
- Custom Exception Handling
- Java Collections

Use of Concrete Subclasses

Inheritance & Polymorphism

Requirement:

“Use at least 3 concrete subclasses of an abstract class.”

Our Implementation:

- SmartDevice is abstract
- It cannot be instantiated directly
- It is implemented by many concrete device classes

Concrete subclasses in our project:

- SecurityAlarm
- GarageDoor
- PlayStation
- MotionSensor
- DoorLock

- Light
- Thermostat
- SmartTV
- AirConditioner
- WashingMachine

SmartDevice:

```
public abstract class SmartDevice {
    protected abstract void onTurnOn();
    protected abstract void onTurnOff();
```

Light:

```
public class Light extends SmartDevice {
  @Override
  protected void onTurnOn() {
    System.out.println(name + " is turned ON.");
```

> Subclass

Thermostat:

```
public class Thermostat extends SmartDevice {
  @Override
  protected void onTurnOn() {
    System.out.println(name + " is activated.");
```

> Subclass

Interfaces

Requirement:

“Use interfaces in the project.”

Our Implementation:

- Created AutomationController interface
- Defines automation rule execution
- Implemented by CentralController

Concrete Interface In Our Project:

Automation Controller

AutomationController (Interface):

```
public interface AutomationController {
 void tick(LocalTime time, User user);
}
```

CentralController:

```
public class CentralController implements AutomationController {

 @Override
  public void tick(LocalTime systemTime, User currentUser) {
 // Automation rules execution
 }
}
```

Home Structure & Composition

Requirement:

“A Home contains Rooms, and Rooms contain SmartDevices.”

Our Design:

- Home contains multiple Room objects
- Home also contains global devices (Alarm, DoorLock)
- Each Room contains a list of SmartDevice

Home:

```
// Home.java
private List<Room> rooms = new ArrayList<>();
private List<SmartDevice> globalDevices = new ArrayList<>();
```

Room:

```
// Room.java
private List<SmartDevice> devices = new ArrayList<>();

// Adding devices
room.addDevice(new Light("GuestRoom Light"));
home.addGlobalDevice(new SecurityAlarm("SecurityAlarm"));
```

Central Controller & Automation Rules

Requirement:

“Central controller manages devices and automation rules.”

Our Implementation:

- CentralController implements AutomationController
- Automation rules executed every system tick

Example Of Rules Implemented:

Turn off lights at 10 PM

Block PlayStation after 8 PM

ECO mode when battery is low

Central Controller:

```
public class CentralController
 implements AutomationController {

 @Override
 public void tick(LocalTime time, User user) {
 // Automation rules
 }
}
```

Rule Example:

```
// Example rule
if (time.getHour() >= 20 &&
user.getName().equals("Child")) {
 device.turnOff();
}
```

Custom Exception Handling

Requirement:

“Use custom exceptions.”

Our Implementation:

- Created a custom exception: DeviceNotFoundException
- Used when searching for a device that does not exist

Concrete Example:

- Interface defines what must be done
- Controller defines how it is done

DeviceNotFoundException:

```
public class DeviceNotFoundException extends Exception {
    public DeviceNotFoundException(String message) {
        super(message);
    }
}
```

DeviceNotFoundException Usage Example:

```
public SmartDevice getDeviceByName(String name)
        throws DeviceNotFoundException {
    for (SmartDevice device : getAllDevices()) {
        if (device.getName().equalsIgnoreCase(name)) {
            return device;
        }
    }
    throw new DeviceNotFoundException(
        "Device with name '" + name + "' not found."
    );
}
try {home.getDeviceByName("Kitchen Light");}
catch (DeviceNotFoundException e) {
 System.out.println(e.getMessage());}
```

Java Collections

Requirement:

“Use Java Collections (ArrayList, HashMap, etc.)..”

Our Implementation:

ArrayList:
- Rooms in the home
- Devices in each room

HashMap:
- User authentication
- Username → User mapping

Why collections were necessary:

Dynamic number of devices

Easy iteration and filtering

Scalable system design

ArrayList – Device & Room Management

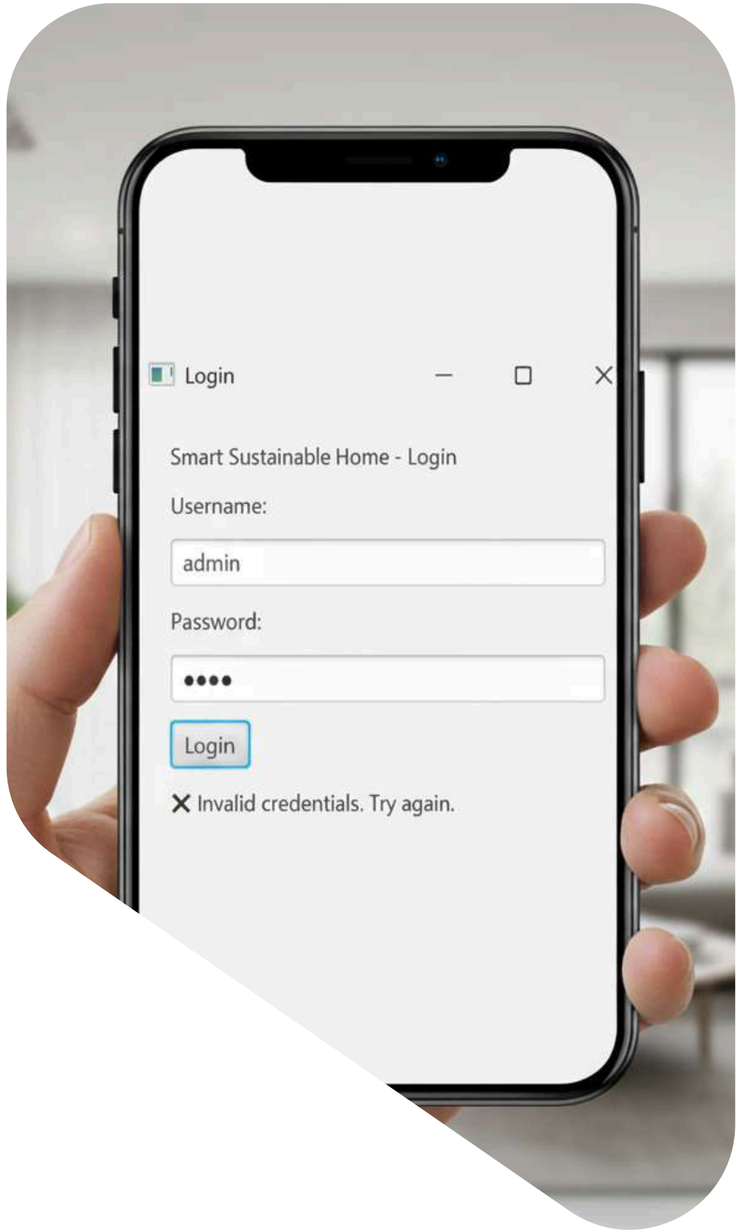
```
private List<Room> rooms = new ArrayList<>();
private List<SmartDevice> globalDevices = new ArrayList<>();

private List<SmartDevice> devices = new ArrayList<>();
```

HashMap – User Authentication

```
private static final Map<String, String> credentials = new HashMap<>();
private static final Map<String, User> users = new HashMap<>();
```

Authentication & Access Control

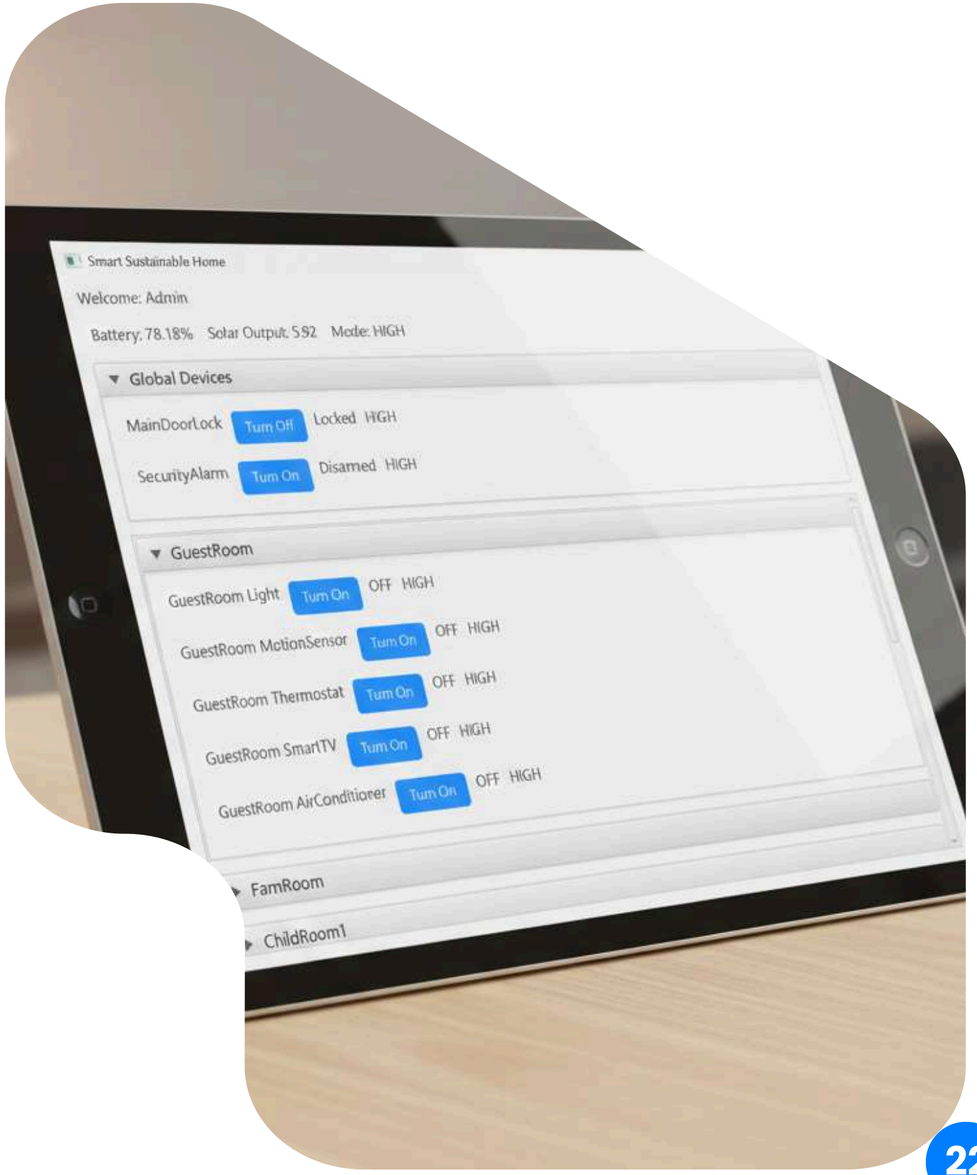
- Secure entry point to the smart home system
- Users authenticate using credentials before accessing controls
- Role-based access ensures only authorized users can perform sensitive actions
- Prevents unauthorized control of devices and security systems



Central Control Interface

- Centralized dashboard for controlling all smart devices
- Allows turning devices on/off and monitoring their states in real time
- Displays energy levels and system status

- Acts as the main interaction point between the user and the smart home



Future Improvements

Evolution from Smart Home to Intelligent Ecosystem

1 Distributed Microservices Architecture

- Monolithic CentralController handles everything
- Single point of failure

- SecurityController (locks, alarms, cameras)
- EnergyController (solar, battery, optimization)
- ClimateController (thermostats, AC, heating)
- EntertainmentController (TV, PlayStation, media)

Easier maintenance

- Update one service without touching others
- Fault isolation
- One service fails others keep running

2 Predictive Maintenance via Anomaly Detection

- Devices fail unexpectedly
- User frustration and downtime

- Monitor power usage patterns over time
- Detect statistical anomalies

Schedule repairs during low usage times

- Extend device lifespan
- Save $$$ per avoided emergency repair

System Scalability

COMPUTATIONAL SCALABILITY

Rule Engine Capacity

Stream-based filtering on device collections
Each rule uses *.stream().filter()* which is O(n): linear time (Time Complexity which is KEY for Scalability)
Example: if you have 11 devices it takes 11 operations
All devices query the SAME energy mode, ensuring consistency

Linear scaling = predictable performance
Works with 6 lights today, works with 600 lights tomorrow

USER SCALABILITY

Role-Based Access Control

Abstract User class with *canAccessDevice(SmartDevice)* method
4 concrete classes: Admin, Parent, Child, Guest

Add unlimited users without modifying core controller

DEVICE SCALABILITY

Device Ecosystem Growth

Abstract *SmartDevice* base class enables plug-and-play device integration

14 rooms + global devices managed through unified interface

11 concrete device types implemented:

Add new device categories by extending *SmartDevice* with ZERO changes to CentralController

Protecting Privacy

Privacy Protection
Managing user data and device activity must be handled responsibly to prevent misuse and ensure user trust. Access control and restricted permissions are essential to protect sensitive household information

Energy Management Complexity

Balancing comfort and sustainability requires careful control of energy-consuming devices. Designing automation rules that reduce consumption without affecting usability remains a key challenge.

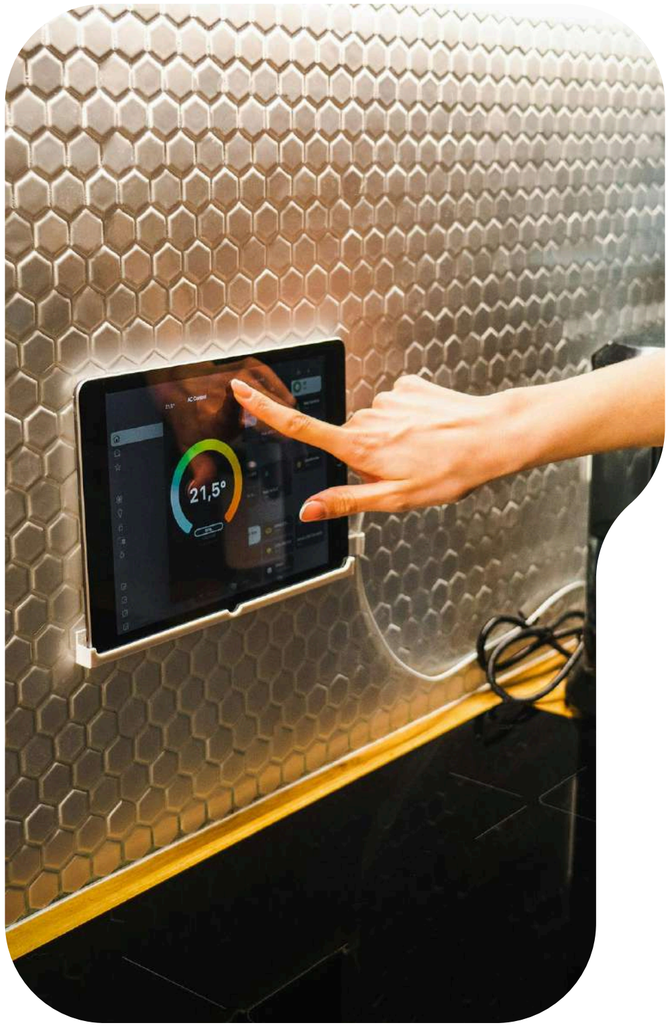
System Scalability

As the number of devices and rooms increases, maintaining system performance and organization becomes more complex. The system must remain efficient and manageable as it scales.

Security and Access Control

Ensuring that only authorized users can control specific devices is critical. Implementing role-based access while maintaining usability requires precise and consistent logic.

challenges & considerations



Conclusion

This project demonstrates how object-oriented programming can be used to build a modular, scalable, and intelligent smart home system. By combining device management, energy optimization, user roles, and automation rules, the system improves efficiency, security, and user experience.

Overall, the project highlights how OOP can power smarter, safer, and more sustainable homes, with strong potential for future expansion through intelligent analytics.

thank you


Energy Saving Mode: ON
Daily Energy Consumption
1.2
Optimize Schedule

Project Resources

GitHub Repository

Contains full Java source code ,commit history and the project documentation

Eraser Workspace:

Contains the system architecture diagrams, class diagram
and the project structure diagrams

PDF Report

Contains the detailed project description , Design decisions and the limitations and future improvements