UNIVERSITY OF MIANWALI

SEN-501

DATABASE SYSTEM

Deliverable Name: Assignment

Names: Muhammad Zariyan (BSE-F23-M30)

Department: Software Engineering

Assignment Title:

Design and Implement Creational Design Patterns for a Smart Home Automation System

Scenario / Problem Statement

You are hired as a Software Architect for SmartNest Solutions, a company developing a next-generation Smart Home Automation System.

The system manages a variety of smart devices such as lights, thermostats, security cameras, speakers, sensors, and door locks.

1. The company wants the software to be:
2. Flexible to support new device types in the future
3. Efficient in creating complex device objects
4. Memory-friendly when duplicating configurations
5. Consistent by ensuring only one central controller controls all devices

To achieve this, management instructs you to use all five Creational Design Patterns:

1. Singleton
2. Factory Method
3. Abstract Factory
4. Builder
5. Prototype

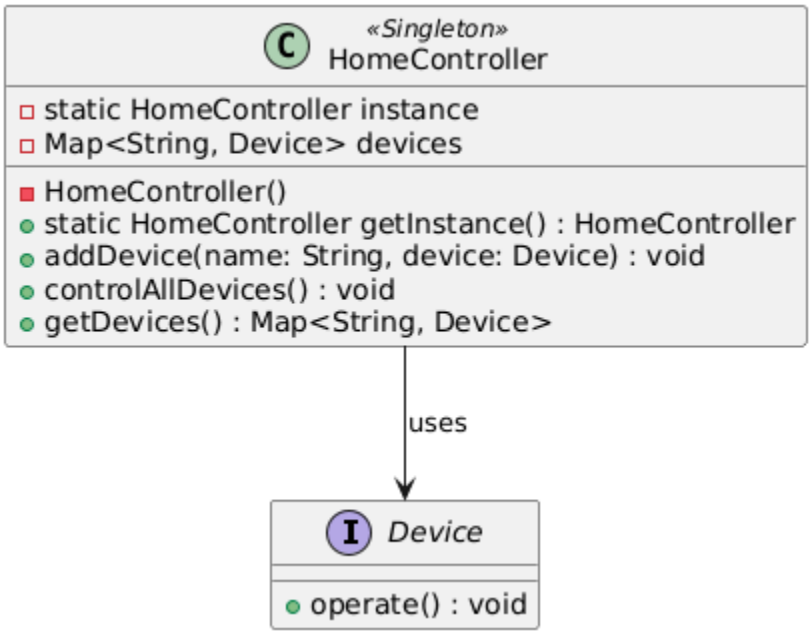
Your task is to design and implement these patterns to support different parts of the Smart Home Automation System.

Singleton Method:

1. **Purpose:** Only one HomeController exists in the system.
2. **Structure:**

   i) Private constructor → prevents creating new objects
   ii) Static instance → stores the single object
   iii) `getInstance()` method → gives access to that object

3. **Use:** Controls all devices from one place.
4. **Relation:** Manages a list/map of `Device` objects



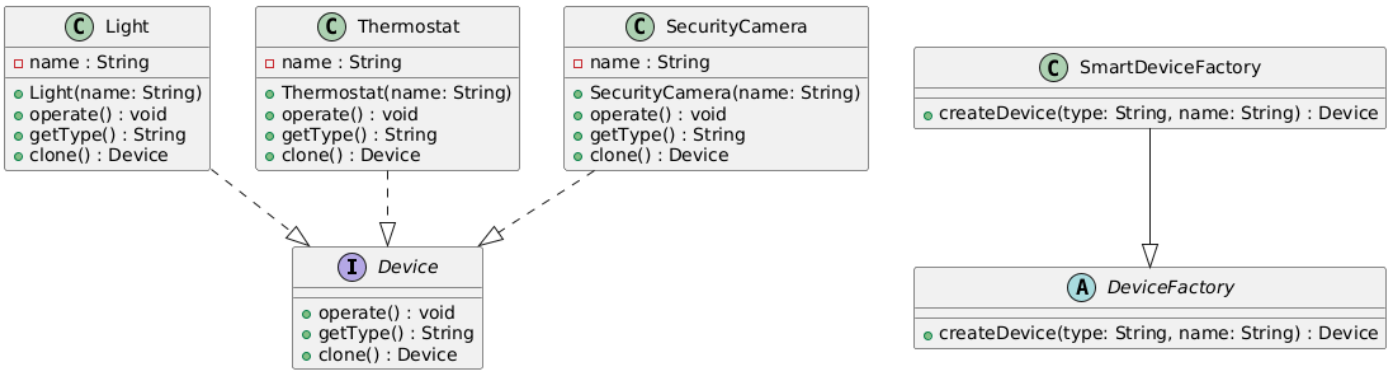
Implementation:

```
// Singleton: Ensures only one central controller exists
public class HomeController {
    private static HomeController instance;
    private Map<String, Device> devices;

    private HomeController() {
        devices = new HashMap<>();
        System.out.println("Home Controller initialized");
    }

    public static synchronized HomeController getInstance() {
        if (instance == null) {
            instance = new HomeController();
        }
        return instance;
    }

    public void addDevice(String name, Device device) {
        devices.put(name, device);
    }

    public void controlAllDevices() {
        System.out.println("=== Home Controller Managing All Devices ===");
        devices.forEach((name, device) -> {
            device.operate();
        });
    }

    public Map<String, Device> getDevices() {
        return devices;
    }
}
```

Factory Method:

Design View:

1. **Purpose:** Create devices without the client knowing the exact class.
2. **Structure:**
   - Abstract Factory → defines a `createDevice(type)` method
   - Concrete Factory → decides which device to create (Light, Thermostat, Camera)
3. **Use:** Dynamically create different devices.
4. **Relation:** Client asks factory → gets a `Device` object.



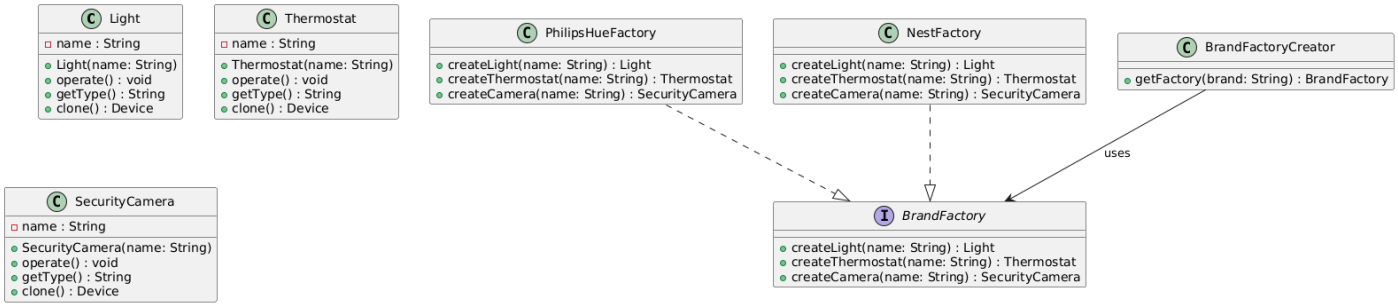
Implementation:

```
    public void operate() {
        System.out.println("☎ Security Camera '" + name + "' is monitoring");
    }

    @Override
    public String getType() {
        return "SecurityCamera";
    }

    @Override
    public Device clone() {
        return new SecurityCamera(this.name);
    }
}

// Factory Method
public abstract class DeviceFactory {
    public abstract Device createDevice(String type, String name);
}

public class SmartDeviceFactory extends DeviceFactory {
    @Override
    public Device createDevice(String type, String name) {
        switch (type.toLowerCase()) {
            case "light":
                return new Light(name);
            case "thermostat":
                return new Thermostat(name);
            case "camera":
                return new SecurityCamera(name);
            default:
                throw new IllegalArgumentException("Unknown device type: " + type);
        }
    }
}
}
```

Abstract Method:

Design View:

1. **Purpose:** Create a family of related devices (like all Philips or all Nest) without the client knowing exact classes.
2. **Structure:**
    - Abstract Factory → declares methods for each type of device
    - Concrete Factories → implement brand-specific devices
    - Factory Creator → chooses which brand factory to return
3. **Use:** Brand-specific devices creation.
4. **Relation:** Client → Factory Creator → Concrete Factory → Devices.



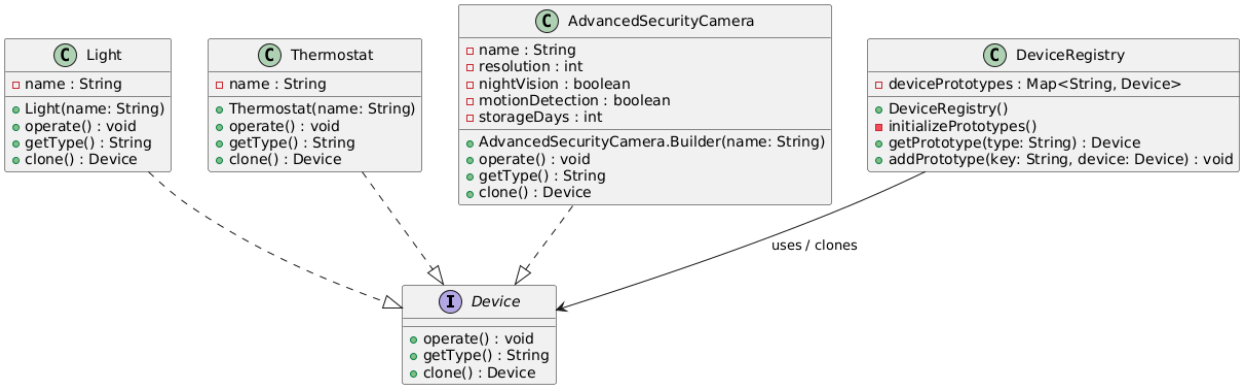
Implementation:

```
public class NestFactory implements BrandFactory {
    @Override
    public Light createLight(String name) {
        return new Light("Nest Light " + name);
    }

    @Override
    public Thermostat createThermostat(String name) {
        return new Thermostat("Nest Thermostat " + name);
    }

    @Override
    public SecurityCamera createCamera(String name) {
        return new SecurityCamera("Nest Cam " + name);
    }
}

// Brand Factory Creator
public class BrandFactoryCreator {
    public static BrandFactory getFactory(String brand) {
        switch (brand.toLowerCase()) {
            case "philips":
                return new PhilipsHueFactory();
            case "nest":
                return new NestFactory();
            default:
                throw new IllegalArgumentException("Unknown brand: " + brand);
        }
    }
}
```

Prototype Method:

Design View:

1. **Purpose:** Create new devices by copying existing ones instead of building from scratch.
2. **Structure:**
    - Registry keeps a map of prototype devices
    - `getPrototype()` → returns a copy of a device
    - `addPrototype()` → add new prototypes
3. **Use:** Quickly create devices with same configuration.
4. **Relation:** DeviceRegistry manages multiple prototypes.



Implementation:

```
// Device Registry for prototype management
public class DeviceRegistry {
    private Map<String, Device> devicePrototypes;

    public DeviceRegistry() {
        devicePrototypes = new HashMap<>();
        initializePrototypes();
    }

    private void initializePrototypes() {
        // Create prototype devices
        devicePrototypes.put("basic_light", new Light("Basic Light"));
        devicePrototypes.put("basic_thermostat", new Thermostat("Basic Thermostat"));
        devicePrototypes.put("premium_camera",
            new AdvancedSecurityCamera.Builder("Premium Camera")
                .resolution(4)
                .nightVision(true)
                .motionDetection(true)
                .storageDays(30)
                .build());
    }

    public Device getPrototype(String type) {
        Device prototype = devicePrototypes.get(type);
        if (prototype != null) {
            return prototype.clone();
        }
        throw new IllegalArgumentException("Unknown prototype type: " + type);
    }

    public void addPrototype(String key, Device device) {
        devicePrototypes.put(key, device);
    }
}
```