|  |  |  |  |
| --- | --- | --- | --- |
|  | **Course Name: Introduction to Web Technology - Lab** | **Experiment No.1** | |
| **Course Code: 23CP306P**  **Faculty: Bhaumik Thakkar** | **Branch: CSE** | **Semester: V** |
| **Submitted by: Chetan Gadhiya**  **Roll no: 23BCP182** | | | |

**Objective :**

*Explain the factory design pattern and write a program using any object-oriented programming language to demonstrate the working of a \*Smart Home Automation System\* using the \*Factory Pattern\*.*

**Description :**

*The* ***Factory Design Pattern*** *simplifies object creation by using a factory method to instantiate devices in the Smart Home system. Instead of manually creating new device objects, the system uses a factory to generate specific device types like lights, fans, or thermostats. This promotes* ***flexibility*** *and* ***decouples*** *the object creation process from the main system, making it easier to add new device types in the future.*

**Program :**

from abc import ABC, abstractmethod

# Abstract base class for Smart Devices

class SmartDevice(ABC):

    @abstractmethod

    def turn\_on(self):

        pass

    @abstractmethod

    def turn\_off(self):

        pass

# Concrete class for Light Devices

class LightDevice(SmartDevice):

    def turn\_on(self):

        print("Light is turned ON.")

    def turn\_off(self):

        print("Light is turned OFF.")

# Concrete class for Thermostat Devices

class ThermostatDevice(SmartDevice):

    def turn\_on(self):

        print("Thermostat is turned ON.")

    def turn\_off(self):

        print("Thermostat is turned OFF.")

# Factory class for creating devices

class DeviceFactory:

    @staticmethod

    def create\_device(device\_type: str) -> SmartDevice:

        if device\_type == "light":

            return LightDevice()

        elif device\_type == "thermostat":

            return ThermostatDevice()

        else:

            raise ValueError(f"Unknown device type: {device\_type}")

# Example usage of Factory Pattern

print("Factory Pattern Example:")

light = DeviceFactory.create\_device("light")

light.turn\_on()

light.turn\_off()

thermostat = DeviceFactory.create\_device("thermostat")

thermostat.turn\_on()

thermostat.turn\_off()

# Abstract factory for Smart Home devices

class SmartHomeFactory(ABC):

    @abstractmethod

    def create\_light(self) -> SmartDevice:

        pass

    @abstractmethod

    def create\_thermostat(self) -> SmartDevice:

        pass

# Concrete factory implementation

class ConcreteSmartHomeFactory(SmartHomeFactory):

    def create\_light(self) -> SmartDevice:

        return LightDevice()

    def create\_thermostat(self) -> SmartDevice:

        return ThermostatDevice()

# Example usage of Abstract Factory Pattern

print("\nAbstract Factory Pattern Example:")

factory = ConcreteSmartHomeFactory()

# Create a Light Device

light = factory.create\_light()

light.turn\_on()

light.turn\_off()

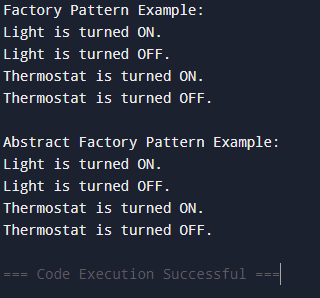
# Create a Thermostat Device

thermostat = factory.create\_thermostat()

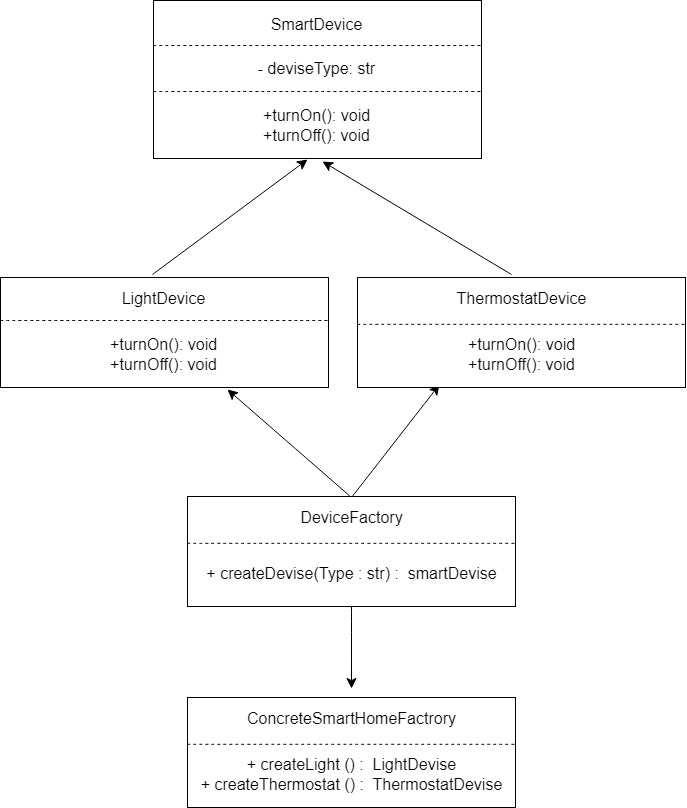
thermostat.turn\_on()

thermostat.turn\_off()

**Output :**

****

**Class Diagram:**



**Conclusion :**

The implementation of the Factory Pattern and Abstract Factory Pattern for a **Smart Home Automation System** demonstrates how these design patterns help achieve:

1. **Scalability**: Easily extendable to include more smart devices.

2. **Code Reusability**: Centralized object creation logic avoids redundant code.

3. **Decoupling**: Separates object instantiation from client code, promoting flexibility.

4. **Maintainability**: Simplified updates or modifications to the system as new features are introduced.