Module Overview

Purpose
This module encapsulates the **Smart Home System**, offering functionality to manage various devices, schedules, and automated triggers. It leverages the Observer and Proxy design patterns for efficient control and monitoring.

Patterns Used
Observer Pattern: Used to notify devices about system changes.
Proxy Pattern: Employed in the DeviceProxy class to control access to devices.

Classes and Methods

1. **Device** (Abstract Base Class)
   Purpose: An abstract base class defining the interface for smart home devices.

Methods:
turn_on: Abstract method to turn the device on.
turn_off: Abstract method to turn the device off.
get_status: Abstract method to get the status of the device.

Code:

```
from abc import ABC, abstractmethod

class Device(ABC):
    @abstractmethod
    def turn_on(self):
        pass

    @abstractmethod
    def turn_off(self):
        pass

    @abstractmethod
    def get_status(self):
        pass
```

2. **Light** (Concrete Class)
   Purpose: Represents a concrete implementation of a light device.

Methods:
turn_on: Turns the light on.
turn_off: Turns the light off.
get_status: Returns the status of the light.

Code:

```
class Light(Device):
    def __init__(self, device_id):
        self.device_id = device_id
        self.status = 'off'

    def turn_on(self):
        self.status = 'on'

    def turn_off(self):
        self.status = 'off'

    def get_status(self):
        return f"Light {self.device_id} is {self.status}."
```

3. **Thermostat** (Concrete Class)
   Purpose: Represents a concrete implementation of a thermostat device.

Methods:
turn_on: Not applicable for a thermostat.
turn_off: Not applicable for a thermostat.
get_status: Returns the temperature set on the thermostat.

Code:

```
class Thermostat(Device):
    def __init__(self, device_id, temperature):
        self.device_id = device_id
        self.temperature = temperature

    def get_status(self):
        return f"Thermostat is set to {self.temperature} degrees."
```

4. **DoorLock** (Concrete Class)
   Purpose: Represents a concrete implementation of a door lock device.

Methods:
turn_on: Locks the door.
turn_off: Unlock the door.
get_status: Returns the status of the door lock.

Code:

```
class DoorLock(Device):
    def __init__(self, device_id):
        self.device_id = device_id
        self.status = 'locked'

    def turn_on(self):
        self.status = 'locked'

    def turn_off(self):
        self.status = 'unlocked'

    def get_status(self):
        return f"Door is {self.status}."
```

Design Patterns Used

1. **Observer Pattern**

   Description: Used in the SmartHomeHub class to notify devices of changes in the system.

Explanation:

When a device is added or removed, all devices are notified to stay in sync with the system changes.

Methods Using Observer Pattern:
add_device: Notifies devices about the addition.
remove_device: Notifies devices about the removal.

2. **Proxy Pattern**

   Description: Used in the DeviceProxy class to control access to devices.

Explanation:
The proxy pattern ensures that device access is controlled through the proxy, allowing additional functionality to be added.

Methods Using Proxy Pattern:
turn_on, **turn_off**, **get_status**: Control access to the real device.

Test Cases (test_smart_home_system.py)

Test Case Overview
Purpose: This test suite ensures the correct functionality of the Smart Home System classes and methods.

Tested Classes: **SmartHomeHub**, **ConcreteDeviceFactory**, **Light**, **Thermostat**, **DoorLock**, **DeviceProxy**.

Test Cases

1. **Test Case**: test_turn_on_device
   Purpose: Verifies that the turn_on_device method correctly turns on a device.

Steps:
Create a SmartHomeHub instance.
Add a device to the hub.
Turn on the device using turn_on_device.
Expected Outcome: The device should be in the "on" state.

Code:

```
class TestSmartHomeSystem(unittest.TestCase):
    def test_turn_on_device(self):
        # Arrange
        hub = SmartHomeHub()
        light = Light(device_id=1)
        hub.add_device(DeviceProxy(light))

        # Act
        hub.turn_on_device(1)

        # Assert
        self.assertEqual(light.get_status(), "Light 1 is on.")
```

2. **Test Case**: test_turn_off_device
   Purpose: Verifies that the turn_off_device method correctly turns off a device.

Steps:
Create a SmartHomeHub instance.
Add a device to the hub.
Turn off the device using turn_off_device.
Expected Outcome: The device should be in the "off" state.

Code:

```
class TestSmartHomeSystem(unittest.TestCase):
    def test_turn_off_device(self):
        # Arrange
        hub = SmartHomeHub()
        light = Light(device_id=1)
        hub.add_device(DeviceProxy(light))

        # Act
        hub.turn_off_device(1)

        # Assert
        self.assertEqual(light.get_status(), "Light 1 is off.")
```

3. **Test Case**: **test_set_schedule**
   Purpose: Verifies that the set_schedule method correctly schedules a task for a device.

Steps:
Create a SmartHomeHub instance.
Add a device to the hub.
Set a schedule for the device using set_schedule.
Expected Outcome: The schedule should be added to the list of scheduled tasks.

Code:

```
class TestSmartHomeSystem(unittest.TestCase):
    def test_set_schedule(self):
        # Arrange
        hub = SmartHomeHub()
        door = DoorLock(device_id=1)
        hub.add_device(DeviceProxy(door))

        # Act
        hub.set_schedule(1, "06:00", "Turn On")

        # Assert
        self.assertEqual(hub.scheduled_tasks, [{'device': 1, 'time':
"06:00", 'command': "Turn On"}])
```

4. **Test Case**: test_add_trigger
   Purpose: Verifies that the add_trigger method correctly adds a trigger for the system.

Steps:
Create a SmartHomeHub instance.
Add a device to the hub.
Add a trigger using add_trigger.
Expected Outcome: The trigger should be added to the list of automated triggers.

Code:

```
class TestSmartHomeSystem(unittest.TestCase):
    def test_add_trigger(self):
        # Arrange
        hub = SmartHomeHub()
        light = Light(device_id=1)
        hub.add_device(DeviceProxy(light))

        # Act
        hub.add_trigger("temperature > 75", "turn_off_device(1)")

        # Assert
        self.assertEqual(hub.automated_triggers, [{'condition':
"temperature > 75", 'action': "turn_off_device(1)"}])
```

5. **Test Case**: test_add_dynamic_device
   Purpose: Verifies that the add_dynamic_device method correctly adds a dynamic device.

Steps:
Create a SmartHomeHub instance.
Add a dynamic device using add_dynamic_device.
Expected Outcome: The dynamic device should be added to the list of devices.

Code:

```
class TestSmartHomeSystem(unittest.TestCase):
    def test_add_dynamic_device(self):
        # Arrange
        hub = SmartHomeHub()
        new_device = Light(device_id=4)

        # Act
        hub.add_dynamic_device(DeviceProxy(new_device))

        # Assert
        self.assertEqual(len(hub.devices), 1)
```

6. **Test Case**: test_remove_dynamic_device
   Purpose: Verifies that the remove_dynamic_device method correctly removes a dynamic device.

Steps:
Create a SmartHomeHub instance.
Add a dynamic device to the hub.
Remove the dynamic device using remove_dynamic_device.
Expected Outcome: The dynamic device should be removed from the list of devices.

Code:

```
class TestSmartHomeSystem(unittest.TestCase):
    def test_remove_dynamic_device(self):
        # Arrange
        hub = SmartHomeHub()
        dynamic_device = Light(device_id=1)
        hub.add_device(DeviceProxy(dynamic_device))

        # Act
        hub.remove_dynamic_device(1)

        # Assert
        self.assertEqual(len(hub.devices), 0)
```

7. **Test Case**: test_dynamic_changes_after_commands
   Purpose: Verifies that dynamic changes in the system are reflected in the status report.

Steps:
Create a SmartHomeHub instance.
Add a dynamic device.
Turn on a device, set a schedule, and add a trigger.
Remove a dynamic device.
Expected Outcome: The status report should include the changes after dynamic modifications.

Code:

```
class TestSmartHomeSystem(unittest.TestCase):
    def test_dynamic_changes_after_commands(self):
        # Arrange
        hub = SmartHomeHub()
        light = Light(device_id=1)
        thermostat = Thermostat(device_id=2, temperature=70)
        hub.add_device(DeviceProxy(light))
        hub.add_device(DeviceProxy(thermostat))

        # Act
        hub.turn_on_device(1)
        hub.set_schedule(2, "06:00", "Turn On")
        hub.add_trigger("temperature > 75", "turn_off_device(1)")

        new_device = Light(device_id=4)
        hub.add_dynamic_device(DeviceProxy(new_device))
        hub.remove_dynamic_device(2)

        # Assert
        expected_status = [
            "Light 1 is on.",
            "Light 4 is off."
        ]
```

```
        status_report = [device.get_status() for device in hub.devices]
        for expected in expected_status:
            self.assertIn(expected, status_report)
```

Patterns Used in Tests

1. **Unit Testing Patterns:**
   Description: Test cases are designed to follow the Arrange-Act-Assert pattern.

   Explanation: Each test case first sets up the necessary conditions (Arrange), then performs the action being tested (Act), and finally checks the expected outcome (Assert).

2. **AAA Pattern** (Arrange-Act-Assert):
   Description: Each test case follows the Arrange-Act-Assert pattern for clear structure and readability.

   Explanation: This pattern ensures that the setup, action, and assertion phases of a test are clearly separated, making it easier to understand and maintain the test cases.