Smart Home - TDD Group

Constraints

Your project must be free of syntactic errors. Please make sure that your code actually runs in PyCharm.

Goal

The goal of this task is to develop an intelligent system to manage a smart room in a house. First of all, an **infrared distance sensor** is placed on the ceiling of the room and is used to determine whether the user is inside the room or not.

Based on the occupancy of the room and on the light measurements obtained by a **photoresistor**, the smart home system turns on/off a **smart light bulb**.

Furthermore, the room has a window on one side, equipped with a **servo motor** to open/close it based on the delta between the temperatures measured by two temperature sensors, one indoor and one outdoor (i.e., inside and outside the room).

Finally, the smart home system also monitors the gas level in the air inside the room through an **air quality sensor** and then triggers an **active buzzer** when too many gas particles are detected.

To recap, the following sensors and actuators are present:

- An infrared distance sensor located on the ceiling to the room.
- A smart light bulb.
- A photoresistor sensor to measure the light level inside the room.
- A servo motor to open/close the window.
- Two temperature sensors, one indoor and one outdoor, used in combination with the servo motor to open/close the window.
- An air quality sensor, used to measure the gas level inside the room.
- An active buzzer that is triggered when a certain level of gas particles in the air is detected.

The communication between the main board and the other components happens with GPIO pins; GPIO communication is configured in BCM mode. For further details on how to use the GPIO library refer to the **mock.GPIO** class in the source code.

Handle any error situation that you may encounter by throwing the `SmartHomeError` exception.

For now, you don't need to know more; further details will be provided in the User Stories below.

Instructions

Depending on your preference, either clone the repository at https://github.com/Espher5/smart_home or download the source files as a ZIP archive; afterwards, import the project into PyCharm.

Take a look at the provided project, which contains the following classes:

- `SmartHome`: you will implement your methods here.
- `SmartHomeError`: exception that you will raise to handle errors.
- `SmartHomeTest`: you will write your tests here.
- `mock.GPIO`: contains the mocked methods for GPIO functionalities.
- `mock.adafruit_dht`: mocked library to control the temperature sensors.

Remember, you are **NOT ALLOWED** to modify the provided API in any way (i.e., class names, method names, parameter types, return types). You can however add fields, methods, or even classes (including other test classes), as long as you comply with the provided API.

Use **TDD** to implement this software system.

The requirements of the software system to be implemented are divided into a set of **USER STORIES**, which serve as a to-do list; you should be able to incrementally develop the software system, without an upfront comprehension of all the requirements. **DO NOT** read ahead and handle the requirements (i.e., specified in the user stories) one at a time in the order provided.

When a story is **IMPLEMENTED**, move on to the **NEXT** one. A story is implemented when you are confident that your software system correctly implements all the functionality stipulated by the story's requirement. This implies that all your **TESTS** for that story and all the tests for the previous stories **PASS**. You may need to review your software system as you progress towards more advanced requirements.

How to Deliver Your Project

1. Reserve a slot for the exam in the following calendar: https://doodle.com/meeting/participate/id/aQn47p0b/vote.

   Each student must reserve a single slot among those still available. In other words, please reserve a single slot among those still available, and make sure to not select a slot already reserved by someone else.

2. Deliver your project by sending an email containing a link to your project (either to a GitHub repository or to a sharing site) to: gscanniello@unisa.it, siromano@unisa.it, and m.esposito253@studenti.unisa.it.

   Keep in mind that the last possible date to deliver your project is January 8th at 23:59.

3. Please wait for a confirmation email after we receive your project.

User Stories

Remember to use TDD to implement the following user stories.

1. User detection

An infrared distance sensor, placed on the ceiling of the room is used to determine whether or not a user is inside the room.

This sensor has a data pin connected to the board and used by the system in order to receive the measurements; more specifically, the data pin is connected to pin GPIO25 (BCM mode).

The communication with the sensors happens via the GPIO input function. The pins have already been set up in the constructor of the `SmartHome` class.

The output of the infrared sensor is an analog signal which increases intensity according to the distance between the sensor and the object (i.e., 0V when an object is very close to the sensor and >3V when the object is out of the max range of the sensor).

Please note that this behavior of this sensor is the opposite of the one seen in the previous exercises.

For this exercise, let's assume the input can be classified into these two categories:

- Non-zero value: nothing is detected in front of the sensor.
- Zero value: it indicates that an object is present in front of the sensor (i.e., a person).

Requirement:

- Implement `SmartHome.check_room_occupancy() -> bool` to verify whether the room has someone inside of it by using the infrared distance sensor.

2. Manage smart light bulb based on occupancy

When the user is inside the room, the smart home system turns on the smart light bulb. On the other hand, the smart home system turns off the light bulb when the user leaves the room. The infrared distance sensor is used to determine whether someone is inside the room (see the previous user story).

The smart light bulb is represented by a LED, connected to the main board via pin GPIO26 (BCM mode). Communication with the LED happens via the GPIO output function. The pin has already been set up in the constructor of the `SmartHome` class.

Finally, since at the first stage of development there is no way to determine the state of the physical light bulb, use the boolean instance variable `self.light_on`, defined in the constructor of the `SmartHome` class, to keep track of the state of the light bulb.

Requirement:

- Implement `SmartHome.manage_light_level() -> None` to control the behavior of the smart light bulb.

3. Manage smart light bulb based on light level

Before turning on the smart light bulb, the system checks how much light is inside the room (by querying the photoresistor).

If the measured light level inside the room is above or equal to the threshold of 500 lux, the smart home system does not turn on the smart light bulb even if the user is in the room (or turns the light off if it was already on); on the other hand, if the light level is below the threshold of 500 lux and the user is in the room, the system turns on the smart light bulb as usual.

The behavior when the user is not inside the room does not change (i.e., the light stays off).

The light level is measured by the photoresistor connected on pin GPIO27 (BCM mode). The communication with the sensor happens via the GPIO input function. For this sensor, the value returned by the GPIO input function is assumed to be in lux.

Requirement:

- Implementat `SmartHome.measure_lux() -> float` to query the photoresistor and measure the amount of lux in the room.
- Update the implementation of `SmartHome.manage_light_level() -> None` to handle the additional constraint of the photoresistor measurements.

4. Open/close window based on temperature

Two temperature sensors, one indoor and one outdoor are used to manage the window.

Whenever the indoor temperature is lower than the outdoor temperature minus two degrees, the system opens the window by using the servo motor; on the other hand, when the indoor temperature is greater than the outdoor temperature plus two degrees, the system closes the window.

The above behavior is only triggered when both of the sensors measure temperature in the range of 18 to 30 degrees celsius. Otherwise, the window stays closed.

The two temperature sensors are DHT11 sensors, and are connected to pins GPIO23 and GPIO24 respectively (BCM mode); they can be controlled with the `mock.adafruit_dht` library.

Both of the sensors have been initialized in the constructor of the `SmartHome` class and can be accessed by using the `self.dht_indoor` and `self.dht_outdoor` objects.

In order to retrieve the temperature use the `mock.adafruit_dht.DHT11.temperature` property (i.e., by calling `self.dht_indoor.temperature`) on the appropriate sensor object. This property returns a float value.

To open/close the window, the system commands a servo motor, which is a type of DC (Direct Current) motor that, upon receiving a signal, can rotate itself to any angle from 0 to 180 degrees. We can control it by sending a PWM (Pulse-Width Modulation) signal to its signal pin; this means sending a HIGH signal for a certain period of time (called duty cycle), followed by a LOW signal period. The duty cycle determines the angle the servo motor will rotate to.

The servo motor is an SG90 model, is connected on pin GPIO6 (BCM mode), and operates at 50Hz frequency. Furthermore, let's assume the window can be in the following states:

- **Fully closed**, corresponding to a 0 degrees rotation of the servo motor.
- **Fully open**, corresponding to a 180 degrees rotation of the servo motor.

In order to calculate the duty cycle corresponding to a certain angle, refer to the following formula:

$$\textit{duty cycle} = (\textit{angle} / 18) + 2$$

The servo motor has already been configured and can be controlled by calling the **`ChangeDutyCycle(duty_cycle: float)`** method on the **`self.servo`** object and passing it the appropriate duty cycle.

Finally, since at the first stage of development there is no way to determine the state of the physical servo motor, use the **`self.window_open`** instance variable to keep track of its state.

Requirement:

- Implement **`SmartHome.manage_window() -> None`** to control the behavior of the window based on the temperature values. In particular, please write your code inside the **`try`** block provided inside the method.

Hint:

- Mocking the **`mock.adafruit_dht.DHT11.temperature`** property is a bit more unusual compared to mocking a method. In particular you have to add the argument **`new_callable=PropertyMock`** to the **`patch`** decorator before your test method when you mock a property, like so:

```
@patch('mock.adafruit_dht.DHT11.temperature', new_callable=PropertyMock)
```

- For the return values you can use **`return_value`** and **`side_effect`** (for multiple return values) on the mock object as usual.

5. Gas leak detection

An air quality sensor is used to check for any gas leaks inside the room; the sensor is configured in a way such that if the amount of gas particles detected is below 500 PPM, the sensor returns a constant reading of 1. As soon as the gas measurement goes to 500 PPM or above, the sensor switches state and starts returning readings of 0.

To notify the user of any gas leak an active buzzer is employed; if the amount of detected gas is greater than or equal to 500 PPM, the system turns on the buzzer; otherwise, when the gas level goes below the threshold of 500 PPM, the buzzer is turned off.

The air quality sensor is connected on pin GPIO5 (BCM mode). The communication with the sensor happens via the GPIO input function.

The active buzzer is connected on pin GPIO6 (BCM mode). The communication with the buzzer happens via the GPIO output function.

Both the pin for the air quality sensor and the one for the buzzer have already been set up in the constructor of the **`SmartHome`** class.

Finally, since at the first stage of development there is no way to determine the state of the physical active buzzer, use the boolean instance variable **`self.buzzer_on`**, defined in the constructor of the **`SmartHome`** class, to keep track of the state of the buzzer.

Requirement:

- Implement **`SmartHome.monitor_air_quality() -> None`** to control the behavior of the air quality sensor and the buzzer.