TDD Exercise

Goal

The goal is to develop a smart system to manage a small parking garage with only three spots.

The following sensors and actuators are present:

- Three infrared distance sensors, one for each parking spot.
- A servo motor to open/close the garage door.
- A Real Time Clock (RTC) to handle time operations.
- A smart lightbulb.

Communication between the main boards and the other components happens with GPIO pins; GPIO communication is configured in BOARD 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 `ParkingGarageError` exception.

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

Instructions

Fork the repository at https://github.com/Espher5/TDD_exercise, then clone it and import the project into PyCharm.

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

- `ParkingGarage`: you will implement your methods here.
- `ParkingGarageError`: exception that you will raise to handle errors.
- `ParkingGarageTest`: you will implement your tests here.
- `mock.GPIO`: contains the mocked methods for GPIO functionalities.
- `mock.RTC`: contains the mocked methods for RTC functionalities.

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.

COMMIT your code at the end of each TDD phase (i.e., RED, GREEN, and REFACTOR).

At the end of this programming task, **PUSH** all your code into your forked repository and then fill out the form at https://forms.gle/ciEYG8hW5ne9JZhv7.

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.

User Stories

1. Car Detection

Three infrared distance sensors, one in each parking spot, are used to determine whether a car is parked in that spot.

Each sensor has a data pin connected to the board, used by the system in order to receive the measurements; more specifically, the three sensors are connected to pin 11, 12, and 13 respectively (BOARD mode).

Communication with the sensor happens via the GPIO input function. The pins have already been set up in the constructor of the `ParkingGarage` class.

The output of the sensor is an analog signal which changes according to the distance between the sensor and the object (i.e., 2.5V when an object is 50 cm away and ~0V when the object is out of the max range of the sensor). For this exercise let's assume the input can be classified into just these two categories:

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

Requirement:

- Implement `ParkingGarage.check_occupancy(pin: int) -> bool` to verify whether or not a specific spot has a parked car over it.

2. Occupancy Display

When requested, the system must be able to provide the number of occupied parking spots in the garage.

Hint for testing: remember the `side_effect` property.

Requirement:

- Implement `ParkingGarage.get_occupied_spots() -> int` to retrieve the number of occupied parking spots.

3. Parking Fee

At the exit of the garage, there is a machine that, once the customer inserts a ticket, calculates the price they must pay before leaving. The machine is equipped with a RTC to keep track of the current time.

The RTC is connected on pin 15 (BOARD mode).

For each hour spent in the garage, there is a flat cost of 2.50 €; additionally, during the weekend (Saturday and Sunday) an additional 25% fee is applied to the total of the parking ticket. Even when a customer does not exceed a full hour, he/she will still pay 2.50 € (see examples below).

Finally, for simplicity, let's assume that the two times are always on the same day (i.e., entry time cannot be greater than exit time).

The RTC pin has already been set up in the constructor of the `ParkingGarage` class. Use the methods of `mock.RTC` to implement this user story.

Requirement:

- Implement `ParkingGarage.calculate_parking_fee(entry_time: str) -> float` to calculate the price of a parking ticket, given the current time. Assume the entry time is provided as a string in the format `"hh:mm:ss"`.

Examples:

- If the entry time is `"12:30:15"`, the exit time (returned by the RTC) is `"15:24:54",` and today is Monday; then the total price to pay will be 7.50€, even if the last hour is not full.
- If the entry time is `"10:15:08"`, the exit time is `"18:12:28",` and today is Saturday (+25%); the total price to pay will be 25€.

4. Open Exit Garage Door

At the exit of the garage, there is an automatic door controlled by a servo motor. When a car approaches this exit, the system fully opens the door; on the other hand, once the car has passed, the system fully closes the door.

A servo motor 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.

Let's assume our servo is connected on pin 16 (BOARD mode), operates at 50hz frequency (see below) and a rotation of 0 degrees corresponds to the garage door being closed, while a rotation of 180 degrees corresponds to the garage door being fully open.

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

duty cycle = (angle / 18) + 2

The servo motor has already been configured and can be controlled by passing the duty cycle (see the formula above) corresponding to the desired angle to the `change_servo_angle(duty_cycle: float)` method in the `ParkingGarage` class.

Finally, since at this stage of development there is no way to determine the state of the physical servo motor, use a boolean instance variable (i.e., `self.servo_status`) to keep track of its state.

Requirement:

- Implement `ParkingGarage.open_garage_door() -> None` and `ParkingGarage.close_garage_door() -> None` to handle the servo motor behavior.

5. Smart lightbulb

In the middle of the garage there is a smart lightbulb; when a car is parked, the system turns on the lightbulb, assuming it is not already on. On the other hand, if the garage is empty, the system turns off the light.

Consider the lightbulb as a LED, connected to the main board via pin 18 (BOARD mode).

Communication with the LED happens via the GPIO output function. The pin has already been set up in the constructor of the `ParkingGarage` class.

Finally, since at this stage of development there is no way to determine the state of the physical lightbulb, use a boolean instance variable (i.e., `self.light_on`) to keep track of its state.

Requirement:

- Implement `ParkingGarage.turn_light_on() -> None` and `ParkingGarage.turn_light_off() -> None` to implement the behavior related to the smart lightbulb.