Intelligent Office - no-TDD Group

Goal

The goal of this task is to develop an intelligent office system, which allows the user to manage the light and air quality level inside the office.

The office is square in shape and it is divided into four quadrants of equal dimension; on the ceiling of each quadrant lies an **infrared distance sensor** to detect the presence of a worker in that quadrant.

The office has a wide window on one side of the upper left quadrant, equipped with a **servo motor** to open/close the blinds.

Based on a **Real Time Clock** (**RTC**), the intelligent office system opens/closes the blinds each working day.

A **photoresistor**, used to measure the light level inside the office, is placed on the ceiling. Based on the measured light level, the intelligent office system turns on/off a (ceiling-mounted) **smart light bulb**.

Finally, the intelligent office system also monitors the air quality in the office through a **carbon dioxide** (**CO2**) **sensor** and then regulates the air quality by controlling the **switch** of an exhaust fan.

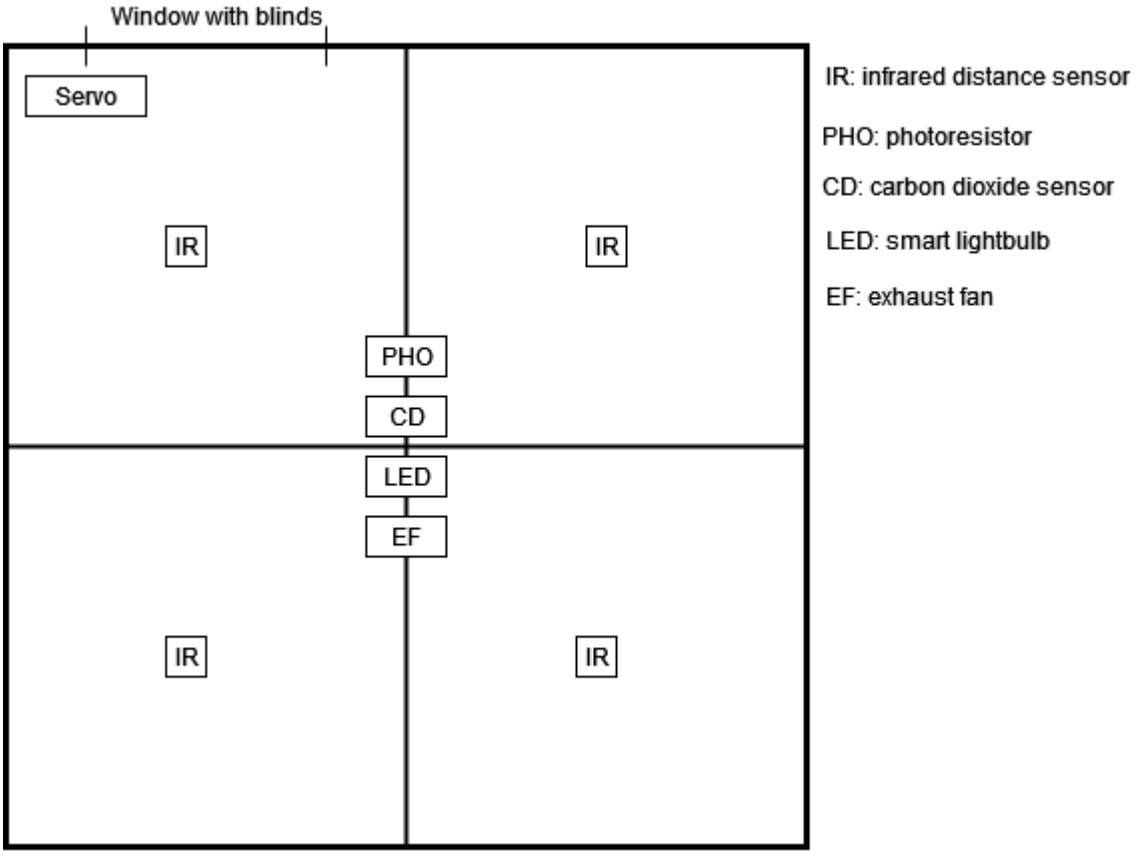
To recap, the following sensors and actuators are present:

- Four infrared distance sensors, one in each quadrant of the office.
- An RTC to handle time operations.
- A servo motor to open/close the blinds of the office window.
- A photoresistor sensor to measure the light level inside the office.
- A smart light bulb.
- A carbon dioxide sensor, used to measure the CO2 levels inside the office.
- A switch to control an exhaust fan mounted on the ceiling.

The communication between the main board 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 **IntelligentOfficeError** exception.

The image below recaps the layout of the sensors and actuators in the office.



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/intelligent_office 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:

- **`IntelligengOffice`**: you will implement your methods here.
- **`IntelligengOfficeError`**: exception that you will raise to handle errors.
- **`IntelligengOfficeTest`**: you can write 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 **no-TDD** to implement this software system (i.e., use the development approach you prefer but not TDD).

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. You may need to review your software system as you progress towards more advanced requirements.

At the end of the task, fill out the post-questionnaire where, among other things, you will be asked to share your project (either as a link to a GitHub repository or as a sharing link to a ZIP file). The post-questionnaire is available at: https://forms.gle/TJTRqTmaR6rLpzuS6.

User Stories

Remember to use **no-TDD** to implement the following user stories.

1. Office worker detection

Four infrared distance sensors, one in each office quadrant, are used to determine whether someone is currently in that quadrant.

Each sensor has a data pin connected to the board, used by the system in order to receive the measurements; more specifically, the four sensors are connected to pin 11, 12, 13, and 15, respectively (BOARD mode).

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

The output of the infrared sensor is an analog signal which changes intensity 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 worker).
- Zero value: nothing is detected in front of the sensor.

Requirement:

- Implement `IntelligentOffice.check_quadrant_occupancy(pin: int) -> bool` to verify whether a specific quadrant has someone inside of it.

2. Open/close blinds based on time

Regardless of the presence of workers in the office, the intelligent office system fully opens the blinds at 8:00 and fully closes them at 20:00 each day except for Saturday and Sunday.

The system gets the current time and day from the RTC module connected on pin 16 (BOARD mode) which has already been set up in the constructor of the `IntelligentOffice` class. Use the instance variable `self.rtc` and the methods of `mock.RTC` to retrieve these values.

To open/close the blinds, 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 is connected on pin 18 (BOARD mode), and operates at 50hz frequency. Furthermore, let's assume the blinds 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:

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) -> None` method in the `IntelligentOffice` class.

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

Requirement:

- Implement `IntelligentOffice.manage_blinds_based_on_time() -> None` to control the behavior of the blinds.

3. Light level management

The intelligent office system allows setting a minimum and a maximum target light level in the office. The former is set to 500 lux, the latter to 550 lux.

To meet the above-mentioned target light levels, the system turns on/off a smart light bulb. In particular, if the actual light level is lower than 500 lux, the system turns on the smart light bulb. On the other hand, if the actual light level is greater than 550 lux, the system turns off the smart light bulb.

The actual light level is measured by the (ceiling-mounted) photoresistor connected on pin 22 (BOARD 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. The pin has already been set up in the constructor of the `IntelligentOffice` class.

The smart light bulb is represented by a LED, connected to the main board via pin 29 (BOARD mode). Communication with the LED happens via the GPIO output function. The pin has already been set up in the constructor of the `IntelligentOffice` class.

Finally, since at this 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 `IntelligentOffice` class, to keep track of the state of the light bulb.

Requirement:

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

4. Manage smart light bulb based on occupancy

When the last worker leaves the office (i.e., the office is now vacant), the intelligent office system stops regulating the light level in the office and then turns off the smart light bulb.

On the other hand, the intelligent office system resumes regulating the light level when the first worker goes back into the office.

Requirement:

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

5. Monitor air quality level

A carbon dioxide sensor is used to monitor the CO2 levels inside the office. If the amount of detected CO2 is greater than or equal to 800 PPM, the system turns on the switch of the exhaust fan until the amount of CO2 is lower than 500 PPM.

The carbon dioxide sensor is connected on pin 31 (BOARD 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 PPM.

The switch to the exhaust fan is connected on pin 32 (BOARD mode). The communication with the sensor happens via the GPIO output function.

Both the pin for the CO2 sensor and the one for the exhaust fan have already been set up in the constructor of the `IntelligentOffice` class.

Finally, since at this stage of development there is no way to determine the state of the physical exhaust fan switch, use the boolean instance variable `self.fan_switch_on`, defined in the constructor of the `IntelligentOffice` class, to keep track of the state of the fan switch.

Requirement:

- Implement `IntelligentOffice.monitor_air_quality() -> None` to control the behavior of CO2 sensor and the exhaust fan switch.