Тринадцатое практическое занятие

Разработка клиент-серверного приложения

Продолжим изучать начатую в рамках прошлого занятия тему управления периферийным оборудованием с внешнего устройства.

Вспомним, на чем мы остановились!

На стороне микроконтроллера (сервера) мы запускали код:

```
import sys

from led import Led

LED_PIN = 16

led = Led(led_pin=LED_PIN)

while True:
    command = sys.stdin.readline().strip().lower()
    if command == "on":
        led.on()
        sys.stdout.write("Диод включен!\n".encode("utf-8"))
    elif command == "off":
        led.off()
        sys.stdout.write("Диод выключен!\n".encode("utf-8"))
    elif command[0] == "b":
        try:
            brightness = int(command[1:])
            led.turn_to_brightness(brightness)
            sys.stdout.write(f"Диод включен на яркость
{brightness/255*100:.2f}% от максимальной!\n".encode("utf-8"))
        except Exception as e:
            sys.stdout.write(f"Неверная команда!\n".encode("utf-8"))
    else:
        sys.stdout.write("Неверная команда!\n".encode("utf-8"))
```

Этот код определял логику работы светодиода, исходя из принимаемых через серийное соединение команд от клиента.

На стороне клиента мы должны были сформировать необходимые команды в виде строк, закодировать их в массив байт и отправить через серийное соединение на микроконтроллер. После чего получить от него ответ со статусом выполнения

команды. В самом простом виде "клиентский" код, запускаемый на компьютере выглядел так:

```
import serial

from CONFIG import RP_PORT

s = serial.Serial(port=RP_PORT, baudrate=115200)

while True:
    command = input("Command: ")
    command = f"{command}\n".encode("UTF-8")
    s.write(command)
    answer = s.readline().strip().decode("UTF-8")
    print(answer)
```

В чем тут проблема? Как всегда почти во всем! По факту наш код сейчас полностью "процедурен" и практически непригоден к нормальному расширению. Что это значит? Это значит то, что пока мы светим одной лампочкой все вроде просто и понятно, но, что произойдет, если потребуется управлять еще одной? А что, если еще и еще? А если еще и другими устройствами?

Наш "серверный" код будет прирастать все новыми и новыми условными выражениями пока не превратится в бесконечное перечисление из всех возможных комбинаций для всех возможных устройств, требуя все больше усилий для поддержки своей работоспособности.

Еще одна проблема со стороны "сервера" в том, что создание устройств жестко записано в код программы, а это значит, что изменение пинов к которым мы подключаем устройство однозначно и не может быть изменено без корректировки и перезаписи на микроконтроллер программы.

Резюмируя: у нас полностью нарушен второй принцип SOLID - наш код полностью открыт к изменениям и полностью закрыт к расширению... Исправим это!

Как же все это сделать? Конечно с помощью ООП и SOLID принципов!

Но рассмотрим сначала немного теоретическую часть. Мы уже неоднократно в тексте использовали слова "клиент" и "сервер"... Разберемся в их значении!

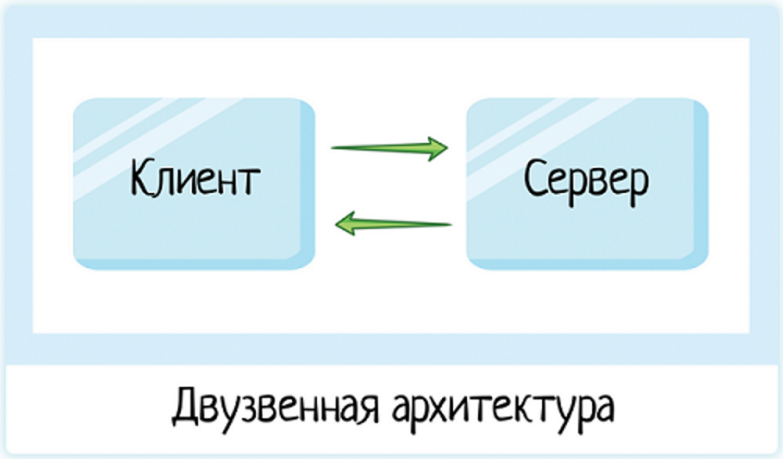
Сервер (Server) – это часть приложения, которая обрабатывает запросы от клиентов, выполняет бизнес-логику и управляет данными, то есть делает то, что ждет от нее пользователь.

Клиент - это часть программы, которая отправляет запросы на сервер и принимает ответы от него.

Клиент-серверное приложение — это архитектура программного обеспечения, определяющая взаимодействие этих частей (например, через интернет или локальную сеть) для выполнения необходимых задач.

Каждая часть выполняет свои функции, что позволяет распределить нагрузку и упростить разработку и поддержку приложения.

В общем виде мы можем представить структуру работы нашей программы в следующем виде:



В нашем случае:

- Клиент - это программа, запущенная на компьютере.
- Сервер - это микроконтроллер, реагирующий на запросы клиента.
- А стрелочки между ними это сообщения, которыми они между собой обмениваются.

На этом теории достаточно. Самое время пожелать себе удачи и реализовать все это руководствуясь SOLID принципами проектирования объектно-ориентированных программ!

ⓘ Ссылки на все базовые примеры кода в конце методички!

Серверная часть

Сервер

> Последний принцип SOLID - "D" - Dependency inversion principle (Принцип инверсии зависимостей) говорит нам о том, что высокоуровневый код не должен зависеть от низкоуровневого!

В самом низу - у нас код, который определяет непосредственное управление подключаемыми устройствами (светодиодами, кнопками, лентами и пр.).

В самом верху - код сервера, который будет взаимодействовать с внешним миром, получать команды и отчитываться "клиенту" о результатах.

Руководствуясь описанным принципом инверсии зависимостей, сервер не должен зависеть от лампочек и кнопок, ему фактически вообще не нужно о них знать, так как это выходит за пределы области его единственной ответственности - общения с "клиентом"

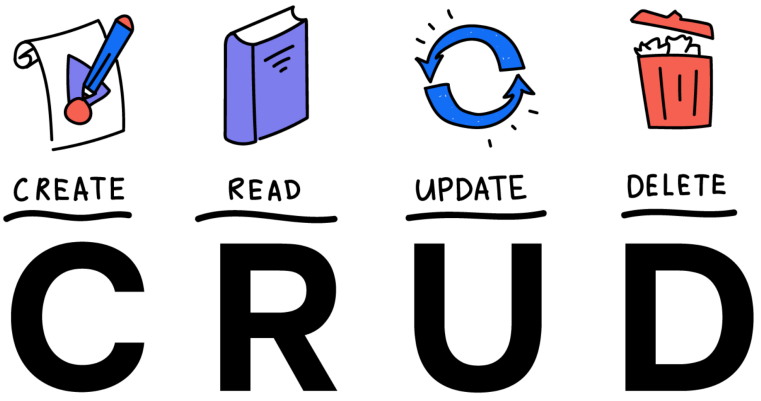
> Привет, первый принцип "S" - Single responsibility principle (Принцип единственной ответственности)!

Но, прежде чем начать писать код, нужно определиться, а какие конкретно требования может предъявить клиент?

В целом, ответ на этот вопрос уже давно выкристаллизовался при работах с интернет-запросами и базами данных.

Принципиально мы можем запросить **создание** новой сущности, ее **удаление**, **изменение** или **получение** от нее информации.

Все это лаконично образует аббревиатуру **CRUD**:



Практически любую задачу можно отнести к одному из этих четырех процессов.

Однако простого постулированния требуемой операции будет, очевидно, недостаточно. Для корректной работы и нам потребуются дополнительные параметры, определяющие выполнение поставленной задачи (что создать, включить светодиод или изменить его цвет и т.п.).

Исходя из этого, представим наш протокол обмена сообщениями между клиентом и сервером как списки пар:

```
["CREATE", {}]
["READ", {}]
["UPDATE", {}]
["DELETE", {}]
```

Где первый элемент всегда строка, указывающая на тип выполняемой операции, а вторая - словарь со всеми необходимыми параметрами для ее корректного выполнения.

> Отметим, что такая структура данных легко сериализуется и десериализуется через JSON, что позволит не думать о том, как привести стандартные типы данных к исходному типу после передачи.

Исходя из всего этого код серверного класса будет:

```
class ServerApp:

    def __init__(self, command_handler_factory=CommandHandlerFactory()):
        self._in_stream = sys.stdin
        self._out_stream = sys.stdout
        self._command_handler_factory = command_handler_factory

    def _read_message(self):
        message = self._in_stream.readline().strip()
        try:
            command_type, command = json.loads(message)
        except Exception as e:
            command_type, command = "404", {}
        return command_type, command

    def _write_message(self, message):
        self._out_stream.write(f"{message}\n".encode("utf-8"))

    def run(self):
        command_type, command = self._read_message()
        command_handler =
self._command_handler_factory.get_command_handler(command_type=command_typ
e)
        if command_handler is not None:
            answer = command_handler.execute_command(command=command)
            if bool(answer) is True and answer != -1:
                self._write_message(answer)
```

Принципиально объект класса `ServerApp` может только запускаться `run()`. При этом на каждом цикле запуска сервер будет ожидать очередное сообщение от клиента:

```
self._read_message()
```

Это сообщение будет распаковываться в тип команды `command_type` и саму команду - `command`.

После чего тип команды будет передаваться "помощнику" - `self._command_handler_factory`, который вернет необходимого "исполнителя" команды, если такой существует.

```
command_handler =
self._command_handler_factory.get_command_handler(command_type=command_typ
e)
```

Именно конкретный исполнитель `command_handler` будет выполнять уже команду `execute_command` по полученному в самом начале словарю с параметрами команды:

```
if command_handler is not None:
    answer = command_handler.execute_command(command=command)
```

Если в результате выполнения команды будет получен существенный ответ, он будет передан обратно клиенту через метод `_write_message`:

```
if bool(answer) is True and answer != -1:
    self._write_message(answer)
```

Принципиально все должно быть понятно кроме того, что это за "помощник" и "исполнитель".

Тут необходимо снова вспомнить о **принципе единственной ответственности**: функции сервера заканчиваются на приеме и отправке сообщений.

Проще будет разобраться на конкретном примере, код "помощника":

```
from devices import DeviceFactory


class CommandHandlerFactory:
    def __init__(self, device_factory=DeviceFactory()):
        self._device_factory = device_factory

    def get_command_handler(self, command_type):
        if command_type == "CREATE":
            return
CreateCommandHandler(device_factory=self._device_factory)
        elif command_type == "READ":
            return ReadCommandHandler(device_factory=self._device_factory)
        elif command_type == "UPDATE":
            return
UpdateCommandHandler(device_factory=self._device_factory)
        elif command_type == "DELETE":
            return
```

```
DeleteCommandHandler(device_factory=self._device_factory)
        else:
            return None
```

И все? Всего лишь один метод с перечислением возможных типов команд?

Почему не вставить его сразу в класс `ServerApp`? Правильно - **принцип единственной ответственности**: если появится новый тип команды, мы будем менять код "помощника", а не "начальника", а если изменится логика обмена сообщениями, то изменим только "начальника".

Другими словами:

> У класса должна быть только одна причина для изменения!

Но что возвращает наш помощник? Правильно - конкретных исполнителей!

Что их объединяет (кроме похожих названий), так это то, что каждый из них должен быть в состоянии выполнить собственную уникальную задачу!

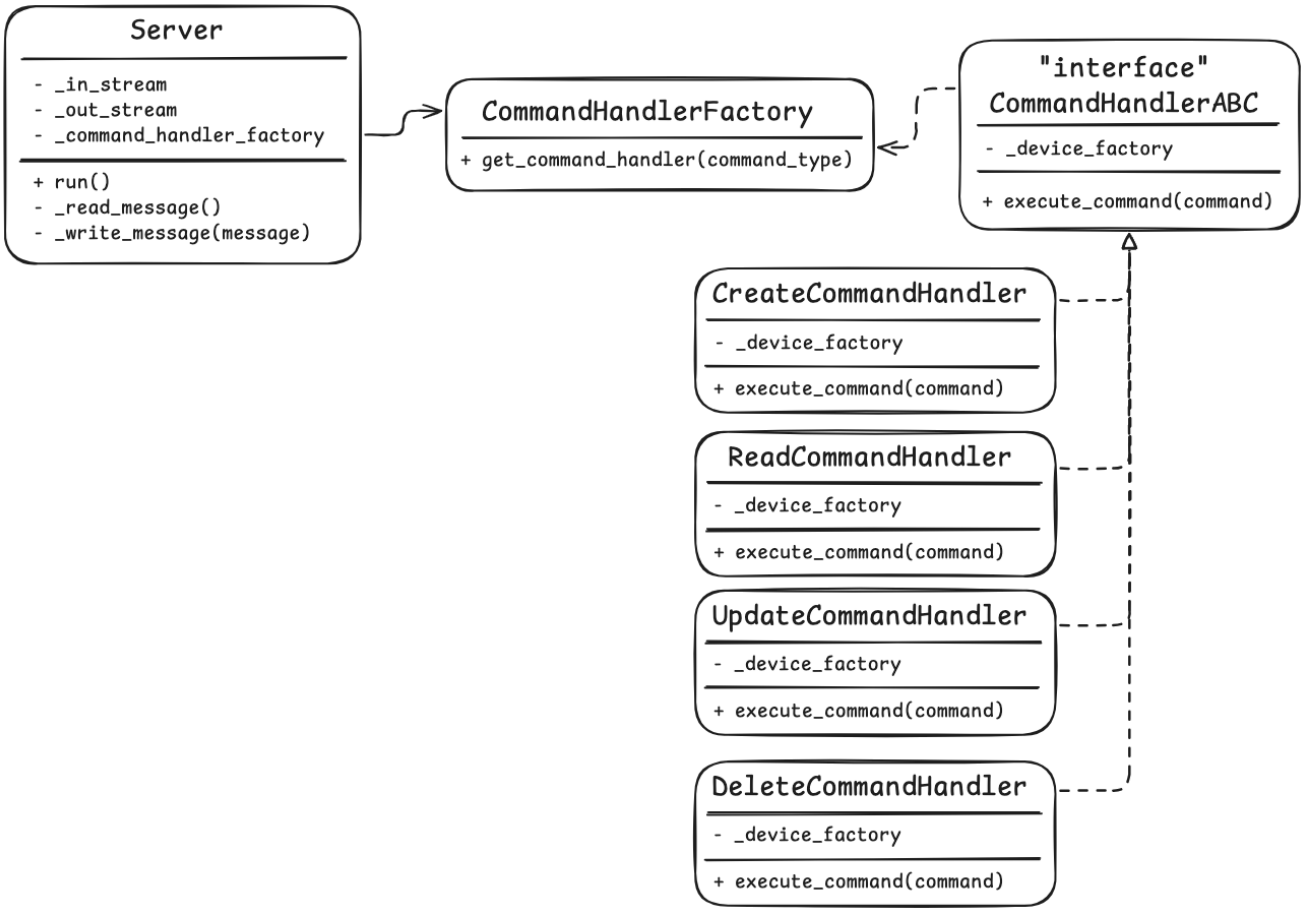
Другими словами выполнить метод:

```
command_handler.execute_command(command=command)
```

Заметим, что само название метода `execute_command` (выполнить команду) никак не связано с типом и внутренней логикой выполнения программы и позволяет "начальнику" просто говорить любому "исполнителю": "Иди работай!" и гарантировано не встретить возражений, вызываемых разными названиями в методах конкретных исполнителях.

> Другими словами все исполнители **реализуют один общий интерфейс**!

На UML диаграмме все это можно представить так:



Взглянем на интерфейс исполнителей:

```
class CommandHandlerABC:

    def __init__(self, device_factory):
        self._device_factory = device_factory

    def execute_command(self, command):
        raise NotImplementedError
```

Он определяет то, что каждый конкретный исполнитель будет содержать некую "фабрику устройств" (`self._device_factory`), являющуюся экземпляром класса `DeviceFactory`, который мы рассмотрим далее.

Важнее то, что каждый наследник класса `CommandHandlerABC` вынужден реализовать по своему метод `execute_command(self, command)`, иначе при вызове этой команды главным начальником программа будет "сломана" исключением `NotImplementedError`, а наш исполнитель - опозорен.

> ⓘ Такой синтаксис интерфейса (через выбрасывания исключения `NotImplementedError`) - вынужденный компромисс, так как в `micropython` нет встроенного в стандартный `python` пакета `abc`, определяющего функционал абстрактных классов. В полноценном синтаксисе интерфейс выглядел бы как:

```python
from abc import ABC, abstractmethod
from devices import DeviceFactory

class CommandHandlerABC(ABC):
    def __init__(self, device_factory):
        self._device_factory = device_factory

    @abstractmethod
    def execute_command(self, command):
        pass
```

Но вернемся к реализации наших "исполнителей":

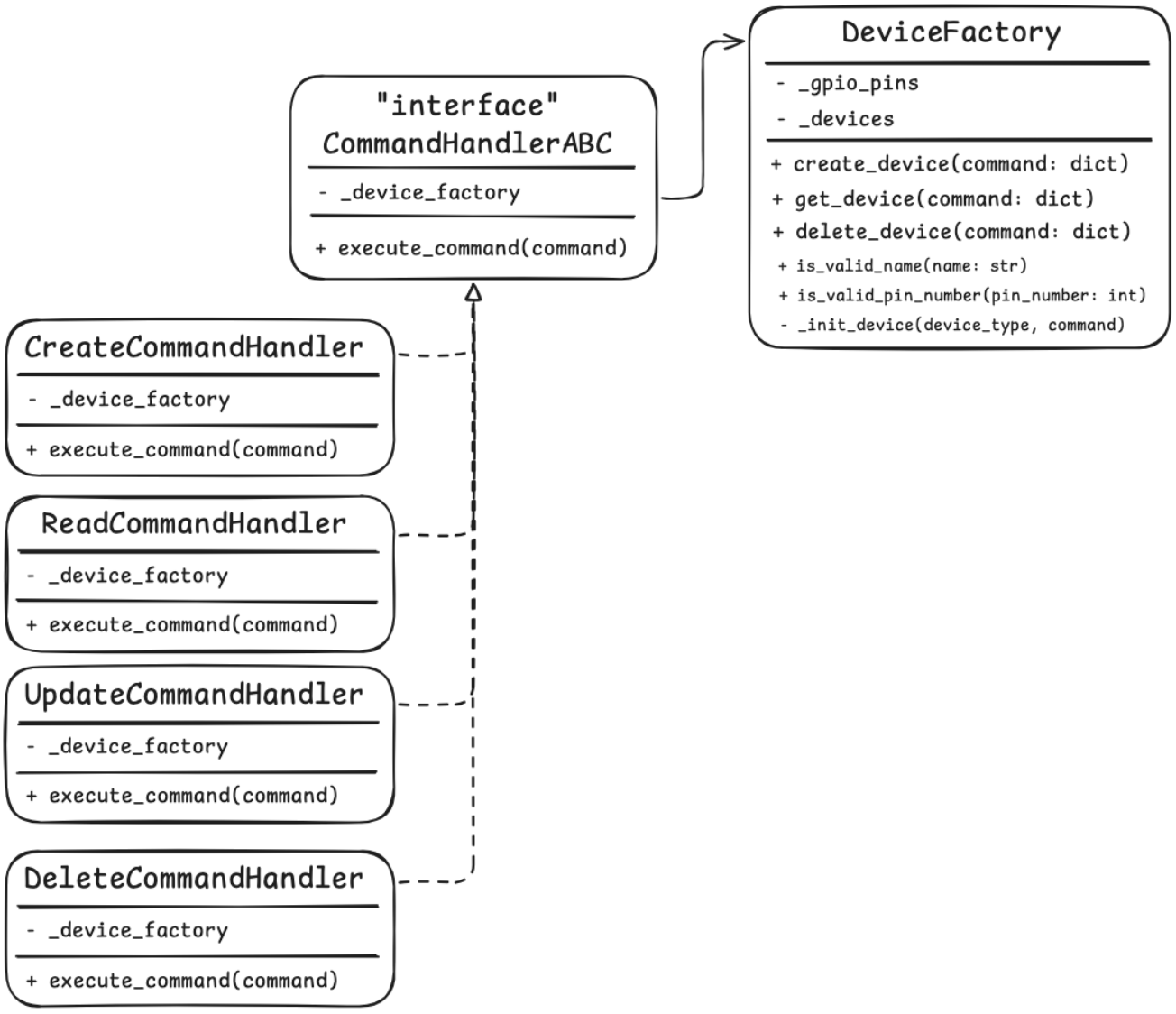
```python
class CreateCommandHandler(CommandHandlerABC):
    def execute_command(self, command: dict):
        status = self._device_factory.create_device(command=command)
        return status

class ReadCommandHandler(CommandHandlerABC):
    def execute_command(self, command: dict):
        device = self._device_factory.get_device(command=command)
        if device is None:
            return f"Устройства {command.get("device_name")} нет!"
        status = device.status
        return status

class UpdateCommandHandler(CommandHandlerABC):
    def execute_command(self, command: dict):
        device = self._device_factory.get_device(command=command)
        if device is not None:
            status = device.execute_command(command=command)
            return status
        return -1

class DeleteCommandHandler(CommandHandlerABC):
    def execute_command(self, command: dict):
        status = self._device_factory.delete_device(command=command)
        return status
```

Что мы видим? То, что каждый "исполнитель" не такой уж и "**реальный** исполнитель"! У каждого из них есть эта "фабрика устройств (`_device_factory`)", которая в реальности делает за них всю основную работу!

Именно она создает устройства (`create_device`), удаляет - `delete_device` и возвращает конкретное устройство - `get_device`!

> Вообще именно "фабрика устройств" контролирует весь жизненный цикл конкретных устройств и только через нее возможно взаимодействие с ними

Дальше наши "исполнители" могут запросить у конкретного устройства его статус (`device.status`) или заставить выполнить непосредственную команду (`device.execute_command(command=command)`), но это уже частности.



Зачем нужны промежуточные исполнители, если они в итоге все используют "фабрику устройств"? По большому счету они унифицируют взаимодействие между фабрикой и сервером ("начальником") позволяя ему не вдаваться в частности наименования отдельных методов.

Пришло время взглянуть на эту "супер-фабрику":

```python
class DeviceFactory:
    _gpio_pins = {key: True for key in range(29)}
    _devices = {}

    def create_device(self, command: dict):
        kwargs = command.get("kwargs")
        device_type = kwargs.get("device_type")
        if kwargs is None or device_type is None:
            return -1
        device = self._init_device(device_type, command)
        if device is not None and device.status == "Created":
            self._devices[device.name] = device
            for pin in device.pins_number:
                self._gpio_pins[pin] = False
            return 0
        return -1

    def _init_device(self, device_type, command):
        if device_type == "led":
            device = LedDevice(device_factory=self, command=command)
        elif device_type == "rgb_led":
            device = RgbLedDevice(device_factory=self, command=command)
        else:
            device = None
        return device

    def get_device(self, command: dict):
        device_name = command.get("device_name")
        if device_name is not None:
            device = self._devices.get(device_name)
            return device
        return -1
```

```
    def delete_device(self, command: dict):
        device_name = command.get("device_name")
        device = self._devices.pop(device_name, None)
        if isinstance(device, Device):
            if isinstance(device, Switchable):
                device.off()
            for pin in device.pins_number:
                self._gpio_pins[pin] = True
            return 0
        return -1

    def is_valid_name(self, name: str):
        if name is None:
            return False
        return name not in self._devices.keys()

    def is_valid_pin_number(self, pin_number: int):
        return bool(self._gpio_pins.get(pin_number, None))
```

Что мы можем увидеть:

Во-первых, в ней всего два статичных атрибута:

```
    _gpio_pins = {key: True for key in range(29)}
    _devices = {}
```

Оба не для публичного использования (фабрика внимательно следит за своим содержимым) и не позволит никому другому вмешаться в логику своей работы.

- Переменная `_gpio_pins` - это словарь, контролирующий свободен ли пин микроконтроллера для подключения. Всего в RP Pico 28 пинов, изначально (пока устройства не созданы) все значения свободны (значение `True`).
- `_devices` - словарь, содержащий в себе созданные устройства, ключом к которым будут выступать уникальное для каждого из устройств имя.

Так как эти переменные инкапсулированы, фабрика содержит в себе два метода `is_valid_name` и `is_valid_pin_number`, позволяющие внешнему объекту узнать существует ли уже такое имя и есть ли подключение к конкретному пину.

Методы: `create_device`, `get_device`, `delete_device` говорят сами за себя и определяют логику создания, получения и удаления устройств.

> Отметим, что все эти методы никак не зависят от конкретного типа устройства и защищены тем самым от риска потенциальных изменений (за исключением метода `delete_device` связанного с интерфейсом `Switchable` (Выключаемый),

> позволяющим выключить устройство во время удаления, после которого доступ к нему перестанет быть возможен).

Конкретный выбор создаваемого устройства выполняется в служебном методе:

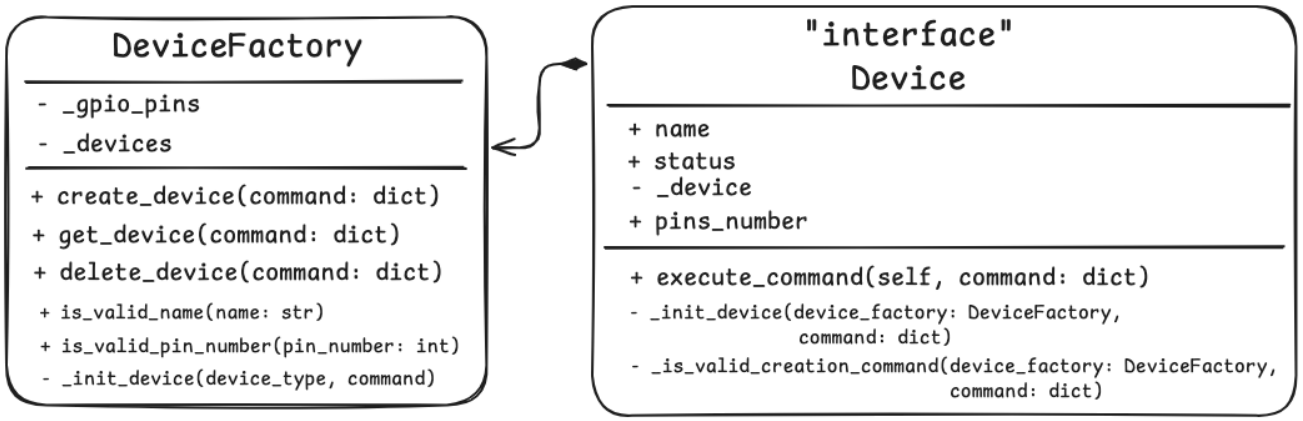
```
    def _init_device(self, device_type, command):
        if device_type == "led":
            device = LedDevice(device_factory=self, command=command)
        elif device_type == "rgb_led":
            device = RgbLedDevice(device_factory=self, command=command)
        else:
            device = None
        return device
```

В дальнейшем (в случае появления новых типов устройств) все возможные изменения не выйдут за пределы этого метода, что снизит риск ошибок в перспективе развития программы.

> Можно вообще избежать изменений! Реализовав класс новой фабрики устройств (наследника `DeviceFactory`) и переопределив за счет полиморфизма метод `_init_device`

```
class NewDeviceFactory(DeviceFactory):

    def _init_device(self, device_type, command):
        device = super()._init_device(device_type, command)
        if device is not None:
            return device
        if device_type == "new_type":
            device = NewDevice(device_factory=self, command=command)
        else:
            device = None
        return device
```

Но что же это за устройства? По большому счету, это для фабрики не важно, ведь они все имплементируют общий интерфейс "устройств".



Что это значит? То, что устройством мы считаем то, у чего есть имя `name`, статус `status`, список пинов `pins_number`, к которым мы его подключили. Про имя мы уже говорили - любое уникальное имя (для того чтобы точно знать какое конкретное устройство нам необходимо). Статус - сохраняет в себе текущее состояние устройства (его как раз и возвращает исполнитель `ReadCommandHandler`).

Но где само устройство, которое будет что-то делать? Оно закрыто от внешнего мира в переменной `_device`.

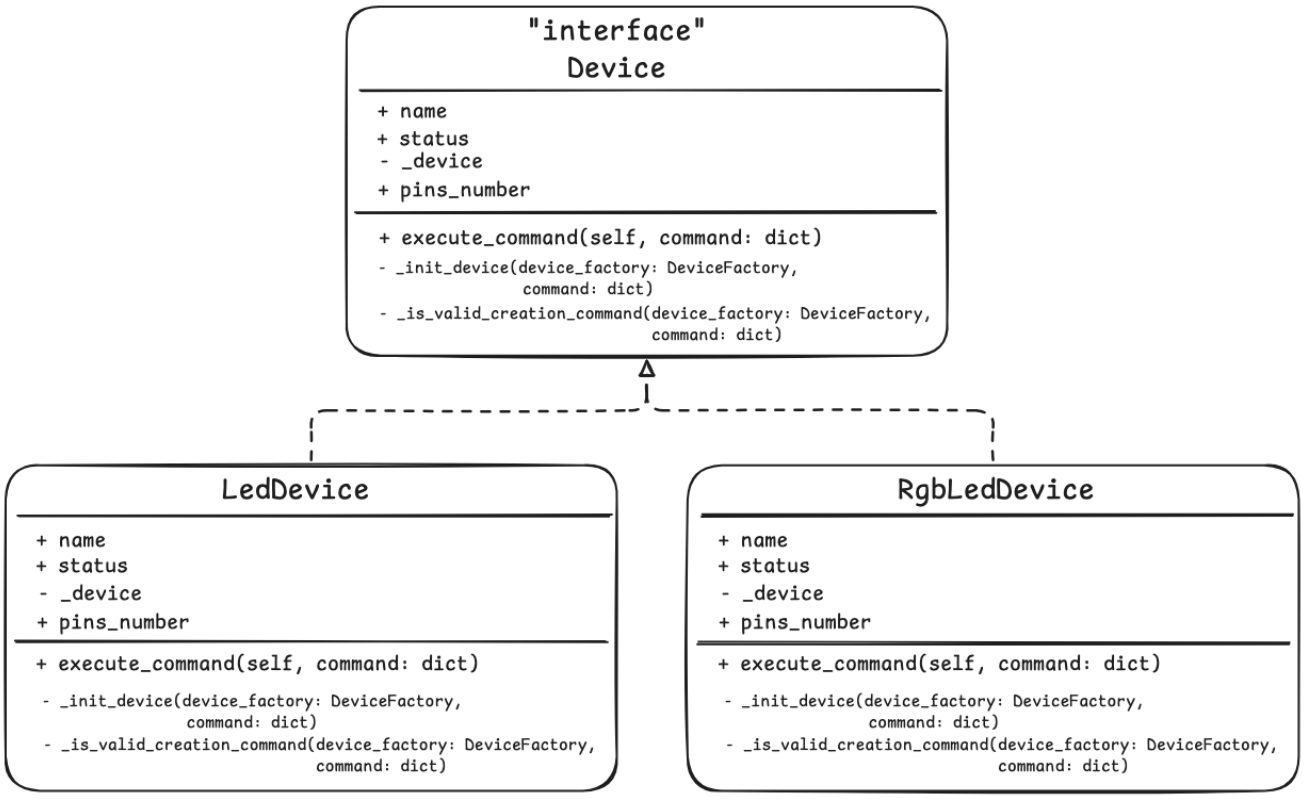
Все взаимодействие с ним будет выполняться посредством единственного открытого метода `execute_command(self, command: dict)`, внутри которого (исходя из переданного словаря с командами) будет происходить работа с конкретным объектом устройства.

```
class Device:

    def __init__(self, device_factory: DeviceFactory, command: dict):
        self.name = None
        self.status = None
        self._device = None
        self.pins_number = []
        self._init_device(device_factory, command)

    def execute_command(self, command: dict):
        raise NotImplementedError

    def _is_valid_creation_command(self, device_factory: DeviceFactory,
command: dict):
        raise NotImplementedError

    def _init_device(self, device_factory: DeviceFactory, command: dict):
        raise NotImplementedError
```

Для реализации этого интерфейса необходимо реализовать еще два служебных метода `_is_valid_creation_command` и `_init_device`, отвечающих за логику инициализации самого устройства `_device`.

Метод `_is_valid_creation_command` проверяет корректность данных для инициализации устройства, а метод `_init_device` его непосредственно создает.

Рассмотрим реализацию этого интерфейса на примере двух устройств: обычного и RGB светодиодов.




```
class LedDevice(Device):

    def __init__(self, device_factory: DeviceFactory, command: dict):
        super().__init__(device_factory, command)

    def _is_valid_creation_command(self, device_factory: DeviceFactory,
command: dict):
        kwargs = command.get("kwargs")
        led_pin = kwargs.get("led_pin")
        device_name = command.get("device_name")
        pin_status = device_factory.is_valid_pin_number(led_pin)
        name_empty = device_factory.is_valid_name(device_name)
        if pin_status is True and name_empty:
            return True
        return False
    def _init_device(self, device_factory: DeviceFactory, command: dict):
        if self._is_valid_creation_command(device_factory, command):
            kwargs = command.get("kwargs")
            led_pin = kwargs.get("led_pin")
            self._device = Led(led_pin=led_pin)
            self.pins_number.append(led_pin)
            self.name = command.get("device_name")
            self.status = "Created"
```

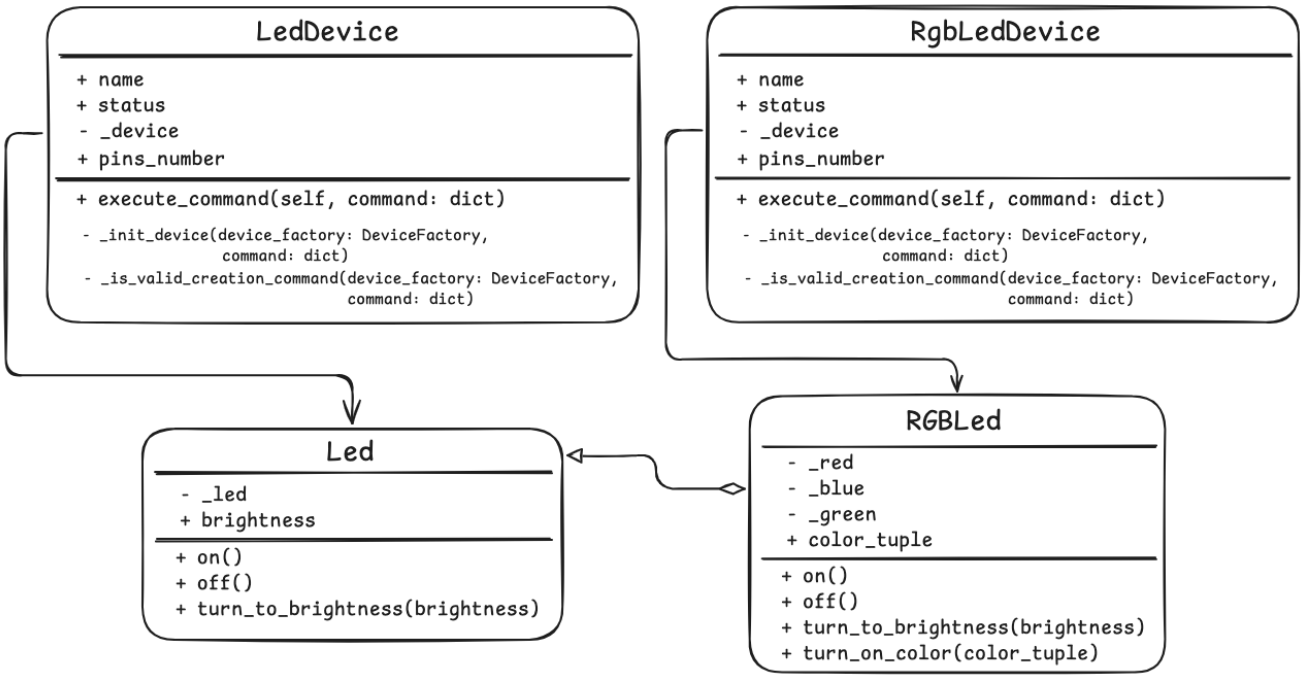
```
            return 0
        self.status = "Not created"
        return -1

    def execute_command(self, command: dict):
        try:
            kwargs = command.get("kwargs")
            base_command = kwargs.get("command")
        except Exception as e:
            return -1
        base_command = str(base_command).lower()

        if base_command == "on":
            self._device.on()
            self.status = f"Диод включен!"
        elif base_command == "off":
            self._device.off()
            self.status = f"Диод выключен!"
        elif base_command == "brightness":
            brightness = kwargs.get("brightness")
            if brightness is None:
                return -1
            self._device.turn_to_brightness(brightness)
            self.status = f"Диод включен на яркость {brightness / 255 *
100:.2f}% от максимальной!"
        else:
            self.status = "Неверная команда!"
            return -1
        return 0
```

```
class RgbLedDevice(Device):
    def __init__(self, device_factory: DeviceFactory, command: dict):
        super().__init__(device_factory, command)

    def _is_valid_creation_command(self, device_factory: DeviceFactory,
command: dict):
        kwargs = command.get("kwargs")
        if kwargs is None:
            return False
        red_pin = kwargs.get("red_pin")
        green_pin = kwargs.get("green_pin")
        blue_pin = kwargs.get("blue_pin")
        pins_status = [device_factory.is_valid_pin_number(red_pin),
                       device_factory.is_valid_pin_number(green_pin),
                       device_factory.is_valid_pin_number(blue_pin),
                       ]
        device_name = command.get("device_name")
        name_empty = device_factory.is_valid_name(device_name)
        if False not in pins_status and name_empty:
```

```
            return True
        return False
    def _init_device(self, device_factory: DeviceFactory, command: dict):
        if self._is_valid_creation_command(device_factory, command):
            kwargs = command.get("kwargs")
            red_pin = kwargs.get("red_pin")
            green_pin = kwargs.get("green_pin")
            blue_pin = kwargs.get("blue_pin")
            self._device = RGBLed(red_pin=red_pin,
                                  green_pin=green_pin,
                                  blue_pin=blue_pin)
            self.pins_number.extend([red_pin, green_pin, blue_pin])
            device_name = command.get("device_name")
            self.name = device_name
            self.status = "Created"
            return 0
        return -1

    def execute_command(self, command: dict):
        try:
            kwargs = command.get("kwargs")
            base_command = kwargs.get("command")
        except Exception as e:
            return -1
        base_command = str(base_command).lower()

        if base_command == "on":
            self._device.on()
            self.status = f"Диод выключен!"
        elif base_command == "off":
            self._device.off()
            self.status = f"Диод выключен!"
        elif base_command == "brightness":
            brightness = kwargs.get("brightness")
            if brightness is None:
                return -1
            self._device.turn_to_brightness(brightness)
            self.status = f"Диод включен на яркость {brightness / 255 *
100:.2f}% от максимальной!"
        elif base_command == "color":
            color_tuple = kwargs.get("color")
            if color_tuple is None:
                return -1
            self._device.turn_on_color(color_tuple=color_tuple)
            self.status = f"Включен RGB цвет {color_tuple}!"
        else:
            self.status = "Неверная команда!"
            return -1
        return 0
```

Оба этих класса являются по своей сути "адаптерами" между словарями с командами, поступающими через "фабрику устройств" и "исполнителей" через "сервер" от "клиента".

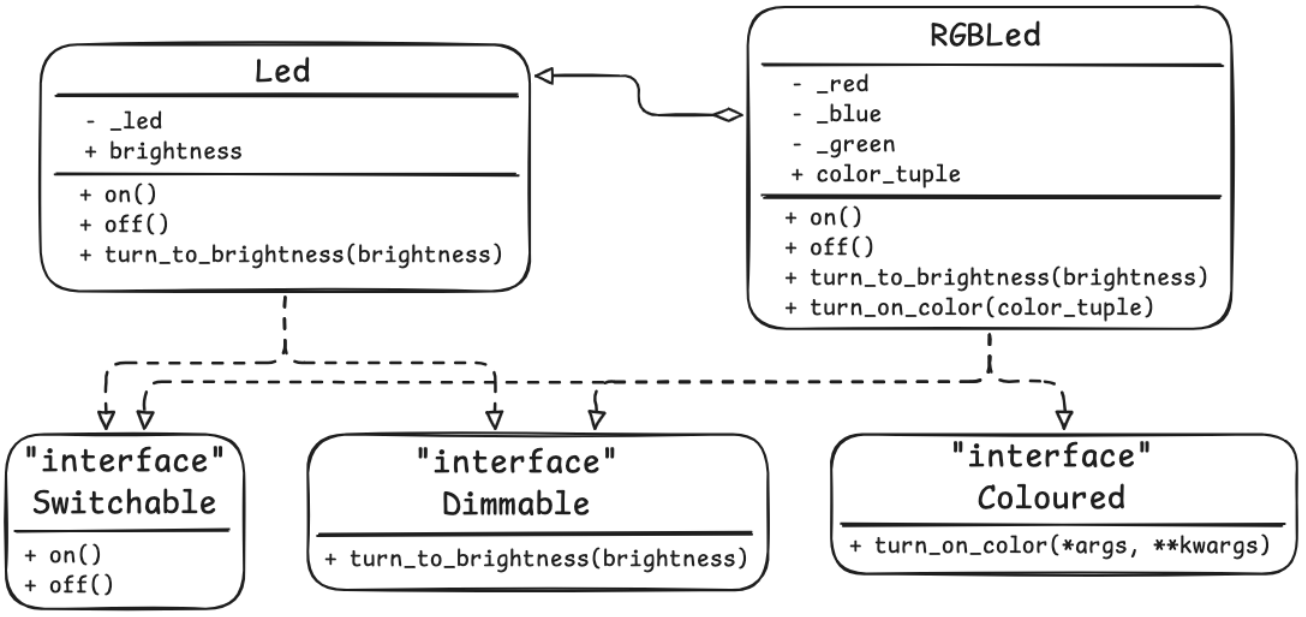
Именно эти классы агрегируют в себе те классы `Led` и `RGBLed`, разработанные нами ранее:



Технически только классы `LedDevice` и `RgbLedDevice` зависят от конкретных методов работы классов `Led` и `RGBLed` и того, что конкретно необходимо для их создания.

И вот мы добрались до самого нижнего уровня нашей серверной части программы!

Последним штрихом, позволяющим в дальнейшем формализовать процесс создания новых устройств, является связь классов `Led` и `RGBLed` через общие интерфейсы:



```
class Switchable:
    def on(self):
        raise NotImplementedError

    def off(self):
        raise NotImplementedError


class Dimmable:
    def turn_to_brightness(self, brightness):
        raise NotImplementedError


class Coloured:
    def turn_on_color(self, *args, **kwargs):
        raise NotImplementedError
```

Так интерфейс `Switchable` заставит все устройства, которые можно просто включить и выключить, делать это именно через методы `on` и `off`, а не потенциальные `turn_on` и им подобные.

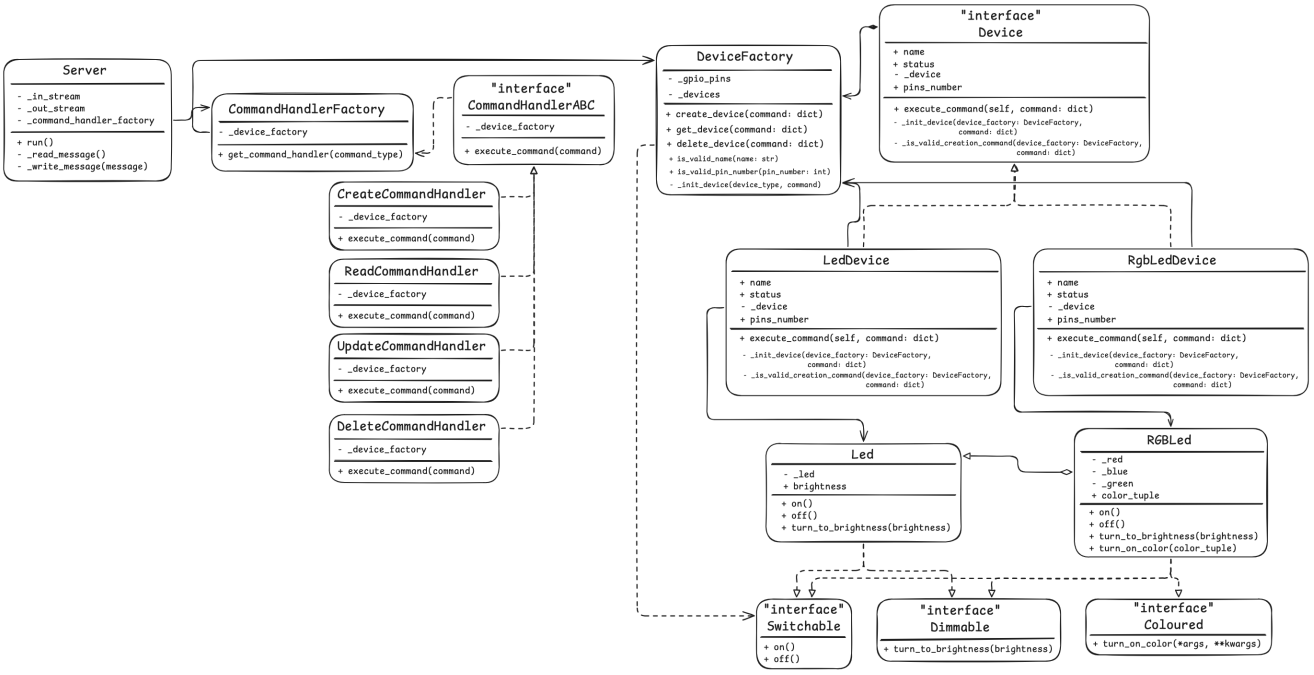
Аналогично управление яркостью будет возможно через метод `turn_to_brightness` интерфейса `Dimmable`, а цветом через `turn_on_color` интерфейса `Coloured`.

Сами классы для диодов будут выглядеть вот так:

```
class Led(Switchable, Dimmable):
    __BRIGHTNESS_SCALE = 255

    def __init__(self, led_pin):
        self._led = PWM(Pin(led_pin, Pin.OUT), freq=1000)
        self.brightness = 0

    def on(self):
        self._led.duty_u16(65535)

    def off(self):
        self._led.duty_u16(0)

    def turn_to_brightness(self, brightness):
        if brightness <= 0:
            brightness = 0
        elif brightness > self.__BRIGHTNESS_SCALE:
            brightness = self.__BRIGHTNESS_SCALE
        duty = int(65535 / self.__BRIGHTNESS_SCALE * brightness)
        self._led.duty_u16(duty)
        self.brightness = brightness
```

```
class RGBLed(Switchable, Dimmable, Coloured):

    __BRIGHTNESS_SCALE = 255

    def __init__(self, red_pin, green_pin, blue_pin):
        self._red = Led(red_pin)
        self._green = Led(green_pin)
        self._blue = Led(blue_pin)
        self.color_tuple = (0, 0, 0)

    def turn_on_color(self, color_tuple):
        self._red.turn_to_brightness(color_tuple[0])
        self._green.turn_to_brightness(color_tuple[1])
        self._blue.turn_to_brightness(color_tuple[2])
        self.color_tuple = color_tuple

    def turn_to_brightness(self, brightness):
        if brightness <= 0:
            brightness = 0
        elif brightness > self.__BRIGHTNESS_SCALE:
            brightness = self.__BRIGHTNESS_SCALE
        total_summ_color = 0
        for idx, led in enumerate([self._red, self._green, self._blue]):
            led_brightness = self.color_tuple[idx] * brightness /
self.__BRIGHTNESS_SCALE
            led.turn_to_brightness(led_brightness)
            total_summ_color += self.color_tuple[idx]
        if total_summ_color == 0:
            self.turn_on_color([brightness] * 3)

    def on(self):
        self.turn_on_color([255, 255, 255])

    def off(self):
        self.turn_on_color([0, 0, 0])
```

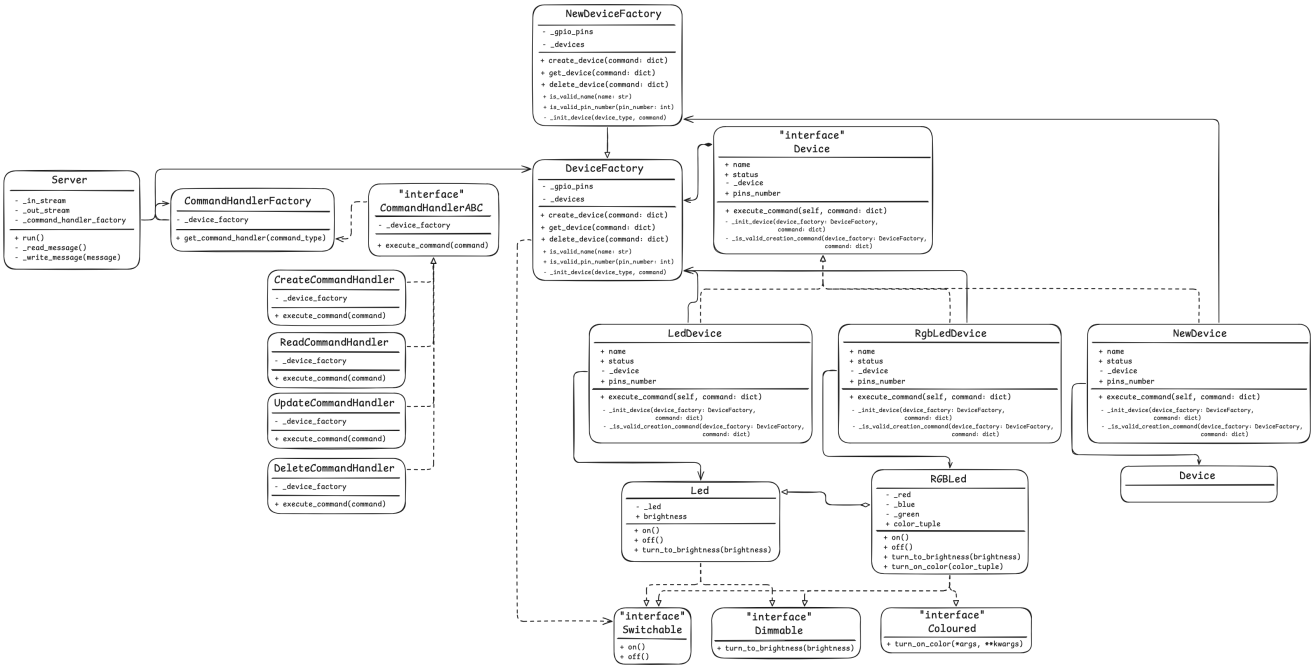
Собрав все это вместе общая UML диаграмма будет выглядеть вот так:



Остается только записать все эти классы в память микроконтроллера и запустить следующий `main.py` файл:

```
from server import ServerApp

if __name__ == "__main__":
    server = ServerApp()

    while True:
        server.run()
```

Такой код запустит работу программы по приведённой выше схеме. Если бы мы захотели бы запустить расширенный вариант программы с добавленными внешними устройствами (который мы рассматривали ранее):



Новую фабрику устройств `NewDeviceFactory` следует явно указать пир создании сервера:

```
from command_handlers import CommandHandlerFactory
from devices import NewDeviceFactory
from server import ServerApp

if __name__ == "__main__":
    server =
ServerApp(command_handler_factory=CommandHandlerFactory(device_factory=New
DeviceFactory()))

    while True:
        server.run()
```

Как видите мы только **расширили** код, а не **изменили** его (мы не вносили изменений ни в один из классов, а только написали новые).

Клиентская часть

По своей сути задача клиентской части приложения сформировать по определённому протоколу сообщение и отправить его на сервер.

В самом простом формате мы можем написать следующий код:

```
import json
import time

import serial

from CONFIG import RP_PORT

s = serial.Serial(port=RP_PORT, baudrate=115200)

def send_command(command, answer=True):
    data = json.dumps(command)
    data = f"{data}\n".encode("UTF-8")
    s.write(data)
    if answer is True:
        read_command = ("READ", {"device_name":
command[1].get("device_name")})
        data = json.dumps(read_command)
        data = f"{data}\n".encode("UTF-8")
        s.write(data)
        a = s.readline().strip().decode("UTF-8")
        print(a)
```

В этом коде функция `send_command` посылает через серийное соединение `s` переданную в нее команду `command` и в случае, если требуется ответ от сервера (`answer=True`), формирует необходимую команду и выводит полученное от сервера сообщение обратно в консоль.

```
led_create = ["CREATE", {"device_name": "led1",
                         "kwargs": {"device_type": "led",
                                    "led_pin": 16,
                                    },
                         },
              ]

send_command(led_create)
```

Так такое сообщение инициализирует обычный светодиод подключенный к 16 пину RP Pico.

Дополнительные примеры сообщений

```
led_create = ["CREATE", {"device_name": "led1",
                         "kwargs": {"device_type": "led",
                                    "led_pin": 16,
                                    },
                         },
              ]
rgb_led_create = ["CREATE", {"device_name": "rgb_led1",
                             "kwargs": {"device_type": "rgb_led",
                                        "red_pin": 13,
                                        "green_pin": 14,
                                        "blue_pin": 15,
                                        },
                             },
                  ]
led_read = ["READ", {"device_name": "led1",
                     "kwargs": {},
                     },
            ]
rgb_led_read = ["READ", {"device_name": "rgb_led1",
                         "kwargs": {},
                         },
                ]
led_on = ["UPDATE", {"device_name": "led1",
                     "kwargs": {"command": "on",
                                },
                     },
          ]
led_off = ["UPDATE", {"device_name": "led1",
```

```
                    "kwargs": {"command": "off",
                               },
                    },
            ]
led_br_50 = ["UPDATE", {"device_name": "led1",
                     "kwargs": {"command": "brightness",
                                "brightness": 125,
                                },
                     },
             ]
rgb_led_on = ["UPDATE", {"device_name": "rgb_led1",
                      "kwargs": {"command": "on",
                                 },
                      },
              ]
rgb_led_off = ["UPDATE", {"device_name": "rgb_led1",
                       "kwargs": {"command": "off",
                                  },
                       },
               ]
rgb_led_br_50 = ["UPDATE", {"device_name": "rgb_led1",
                         "kwargs": {"command": "brightness",
                                    "brightness": 125,
                                    },
                         },
                 ]
rgb_led_color_red = ["UPDATE", {"device_name": "rgb_led1",
                             "kwargs": {"command": "color",
                                        "color": (255, 0, 0),
                                        },
                             },
                     ]
rgb_led_color_green = ["UPDATE", {"device_name": "rgb_led1",
                               "kwargs": {"command": "color",
                                          "color": (0, 255, 0),
                                          },
                               },
                       ]
led_del = ["DELETE", {"device_name": "led1",
                      "kwargs": {},
                      },
           ]
rgb_led_del = ["DELETE", {"device_name": "rgb_led1",
                       "kwargs": {},
                       },
               ]
```

Вызов команд

```
send_command(led_create)
time.sleep(1)

send_command(rgb_led_create)
time.sleep(1)

while True:
    send_command(rgb_led_create)

    send_command(rgb_led_on)
    time.sleep(1)

    send_command(rgb_led_br_50)
    time.sleep(1)

    send_command(rgb_led_off)
    time.sleep(1)

    send_command(rgb_led_color_red)
    time.sleep(1)

    send_command(rgb_led_color_green)
    time.sleep(1)

    send_command(rgb_led_del)
    time.sleep(1)

    send_command(led_on)
    time.sleep(1)

    send_command(led_off)
    time.sleep(1)
```

Ссылки на файлы с кодом

Серверная часть

Ссылка на код

Клиентская часть

Ссылка на код