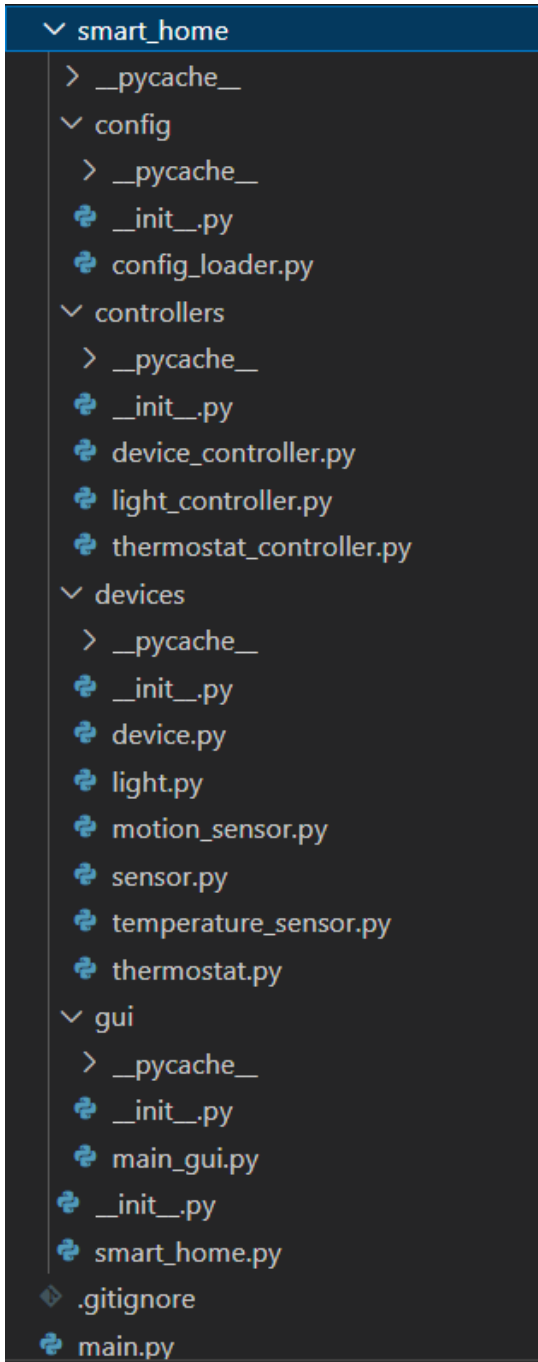


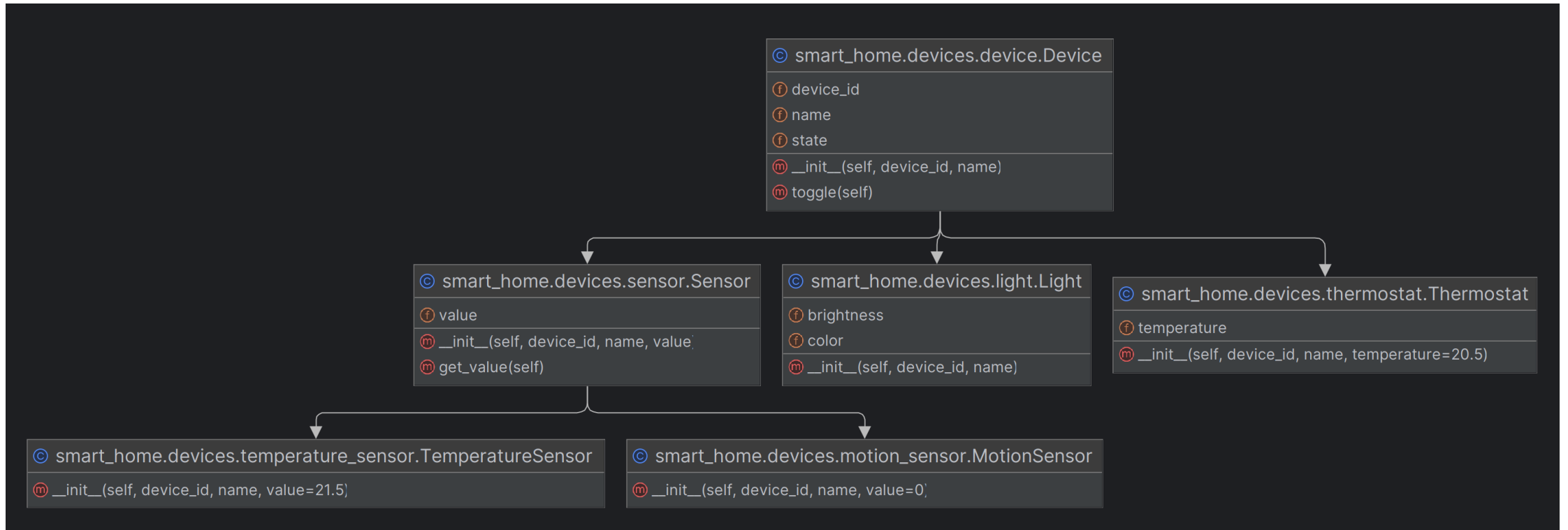


# Py **Smart** Home – Teil 1

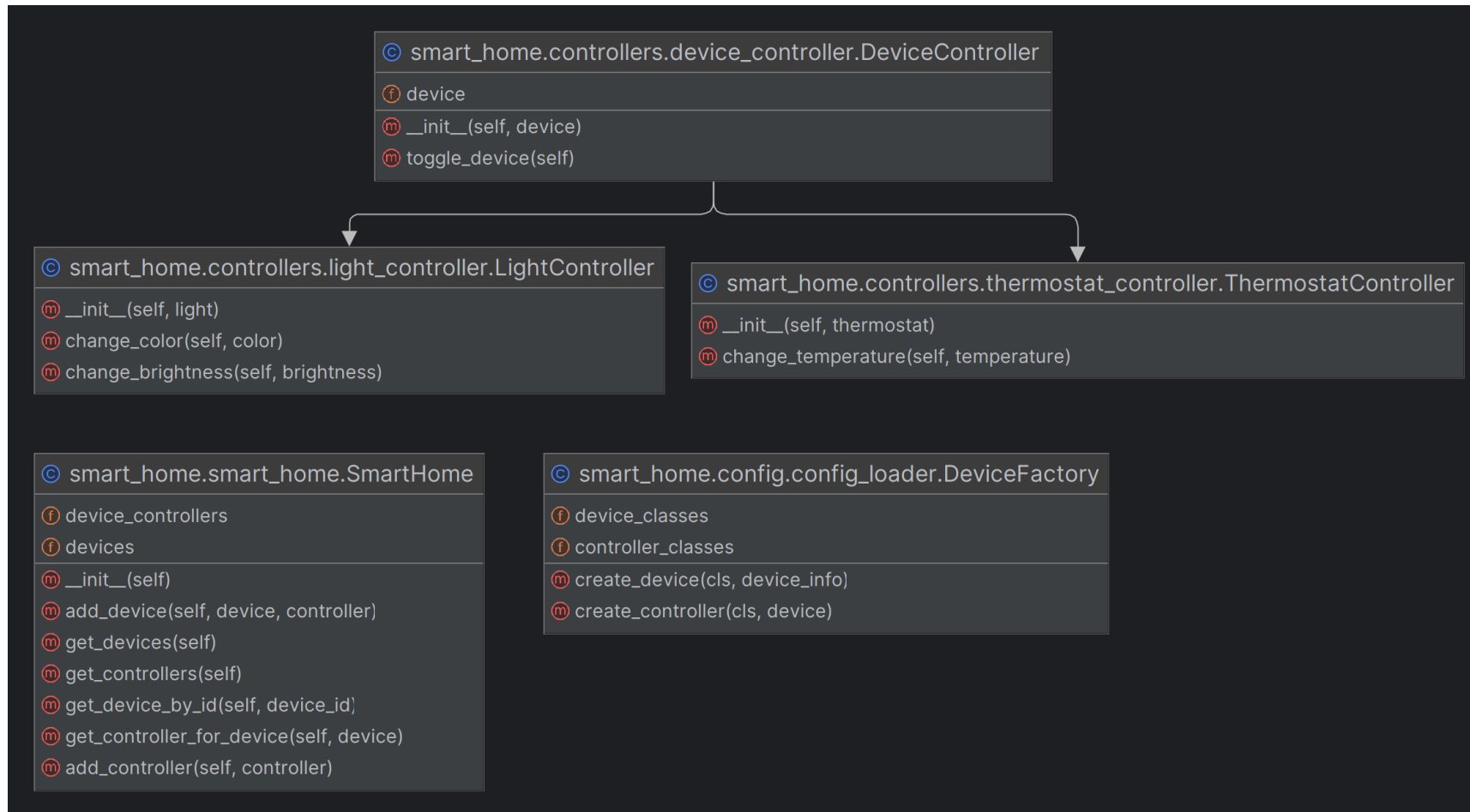
# Package Struktur



# UML



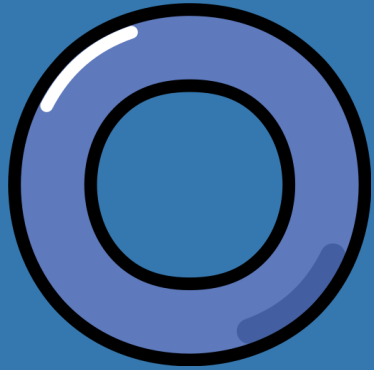
# UML



A large, stylized letter 'S' in a light teal color with a thick black outline and a slight 3D effect.

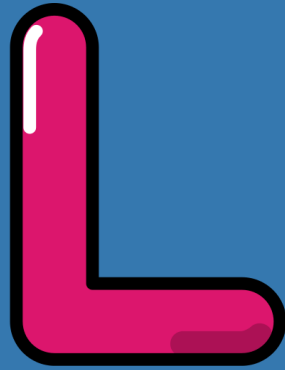
Jede Klasse hat eine einzelne Verantwortung.

Config\_loader Modul -> Laden der Konfiguration  
Device-Unterklasse -> spezifische Verhalten eines bestimmten Gerätetyps

A large, stylized letter 'O' in a light blue color with a thick black outline and a slight 3D effect.

Der Code ist für Erweiterungen offen und für Modifikationen geschlossen.

Neue Gerätetypen -> neue Klasse erstellen + Registrierung im device\_classes Wörterbuch von DeviceFactory, keine Änderungen an der DeviceFactory-Klasse oder dem config\_loader Modul

A large, stylized letter 'L' in a magenta color with a thick black outline and a slight 3D effect.

Unterklassen können ihre Basisklassen ohne Nebenwirkungen ersetzen.

Geräte- und Sensor-Klassen sind durch Erweiterung und Anpassung der Device- und Sensor-Klassen entworfen worden

A large, stylized letter 'I' in a red color with a thick black outline and a slight 3D effect.

Keine unnötigen Abhängigkeiten durch breite Interfaces.

Jeder Controller interagiert nur mit den Methoden, die er benötigt (z.B. Toggle für DeviceController),  
Device Interface ist klein und fokussiert

A large, stylized letter 'D' in a yellow color with a thick black outline and a slight 3D effect.

Hohe Modulniveaus und niedrige Modulniveaus hängen von Abstraktionen ab.

Keine direkten Abhängigkeiten zwischen den Modulen  
DeviceController-Klasse nur von der abstrakten Device-Klasse abhängig und nicht von spezifischen Geräteklassen



# Codebeispiele für SOLID

```
#Single Responsibility Principle (SRP) -> alle Konfigurationsdaten aus einer config file
smart_home = load_config('resources/config.yaml')
```

```
#Open-Closed Principle (OCP) -> einfaches Hinzufügen von devices
```

```
class DeviceFactory:
    device_classes = {
        'device': Device,
        'light': Light,
        'thermostat': Thermostat,
        'sensor': Sensor,
        'temperature_sensor': TemperatureSensor,
        'motion_sensor': MotionSensor
    }

    controller_classes = {
        Device: DeviceController,
        Light: LightController,
        Thermostat: ThermostatController,
        Sensor: DeviceController,
        TemperatureSensor: DeviceController,
        MotionSensor: DeviceController
    }
```

```
#Liskov Substitution Principle (LSP) -> Subklasse kann Funktionalität von Superklasse überschreiben
```

```
#Single Responsibility Principle (SRP) -> Unterklasse definiert spezifisches Verhalten des devices
```

```
class Light(Device):
    def __init__(self, device_id, name):
        super().__init__(device_id, name)
        self.color = 0xFFFFFF
        self.brightness = 100
```

```
#Dependency Inversion Principle (DIP) -> Device Controller nur von Devices abhängig (geringe Abhängigkeiten)
```

```
#Interface Segregation Principle (ISP) -> jeder Controller nutzt nur benötigten Methoden (schmales Interface)
```

```
class DeviceController:
    def __init__(self, device):
        self.device = device

    def toggle_device(self):
        if isinstance(self.device, Device):
            self.device.toggle()
        else:
            raise Exception("Device does not support toggling.")
```

# Patterns

## Factory Pattern

```
class DeviceFactory:
    device_classes = {
        'device': Device,
        'light': Light,
        'thermostat': Thermostat,
        'sensor': Sensor,
        'temperature_sensor': TemperatureSensor,
        'motion_sensor': MotionSensor
    }

    controller_classes = {
        Device: DeviceController,
        Light: LightController,
        Thermostat: ThermostatController,
        Sensor: DeviceController,
        TemperatureSensor: DeviceController,
        MotionSensor: DeviceController
    }
```

## Template Method Pattern

```
def toggle(self):
    self.state = not self.state
    print(f"{self.name} is {'on' if self.state else 'off'}")
```

## Command Pattern

```
class DeviceController:
    def __init__(self, device):
        self.device = device

    def toggle_device(self):
        if isinstance(self.device, Device):
            self.device.toggle()
        else:
            raise Exception("Device does not support toggling.")
```