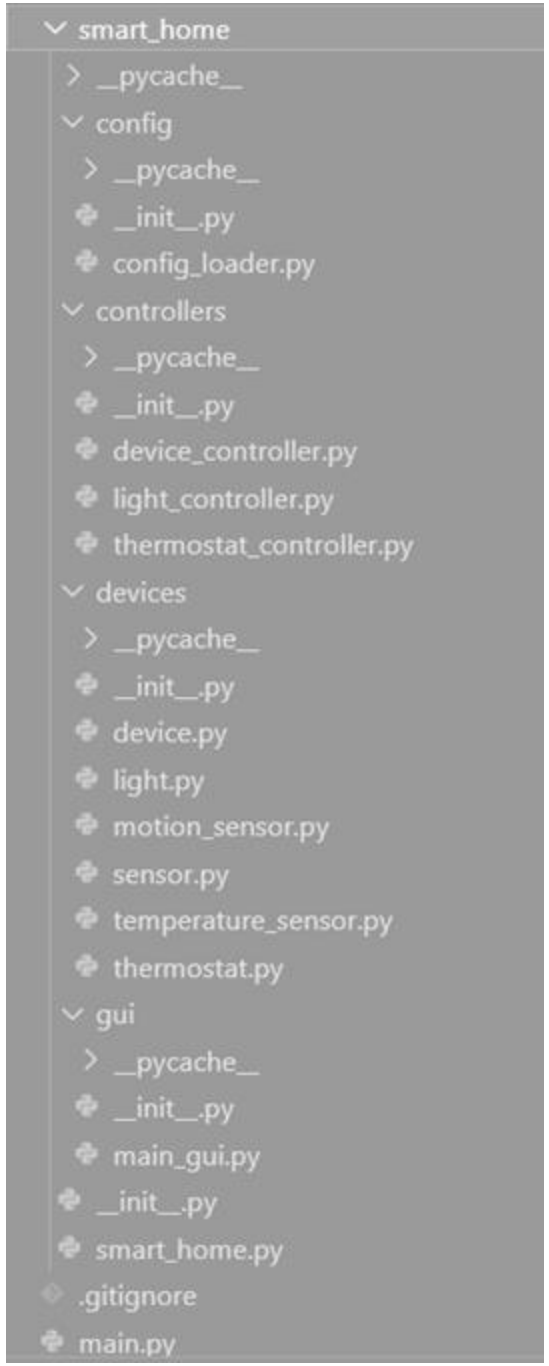
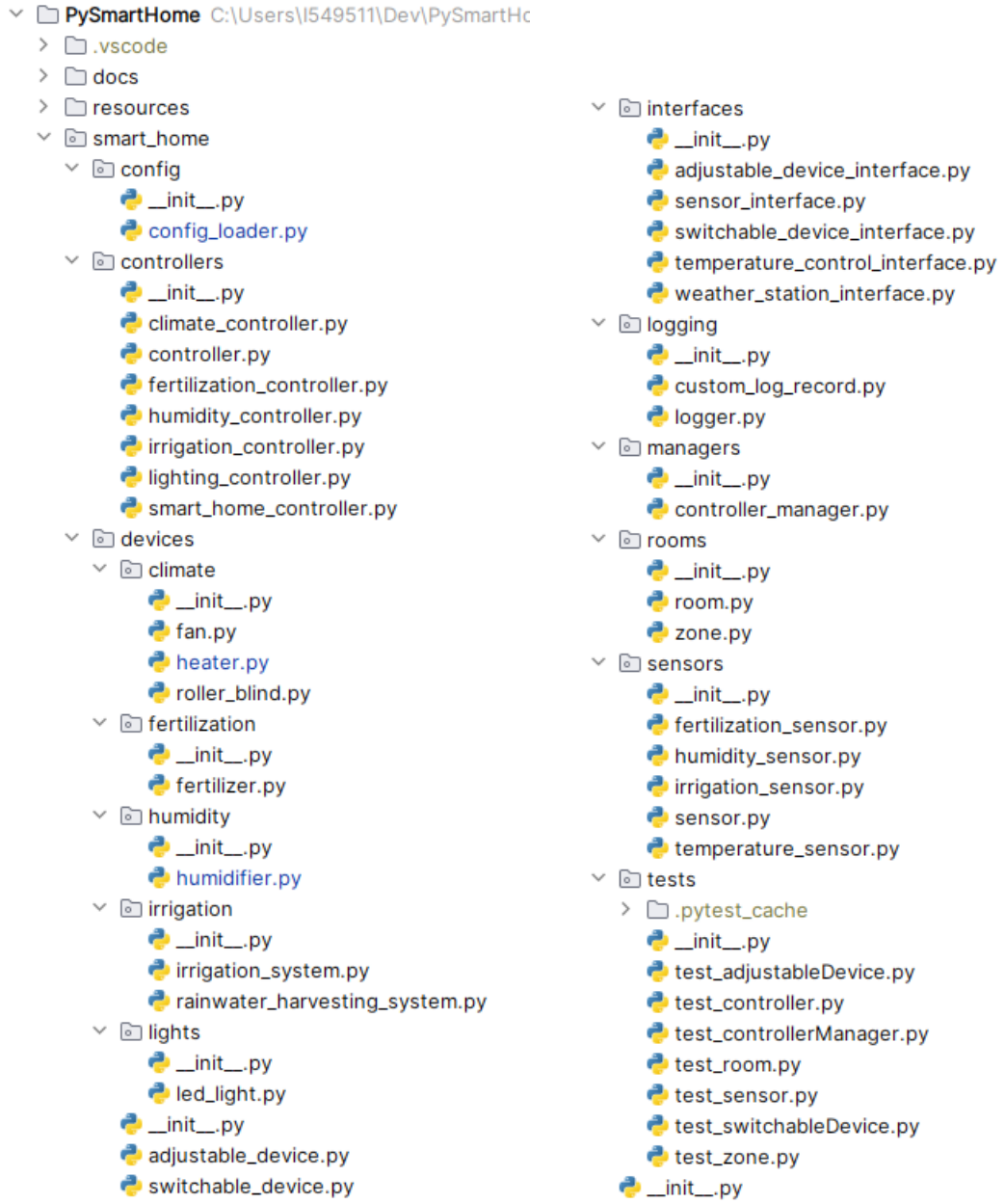
Py Smart Home – Teil 2

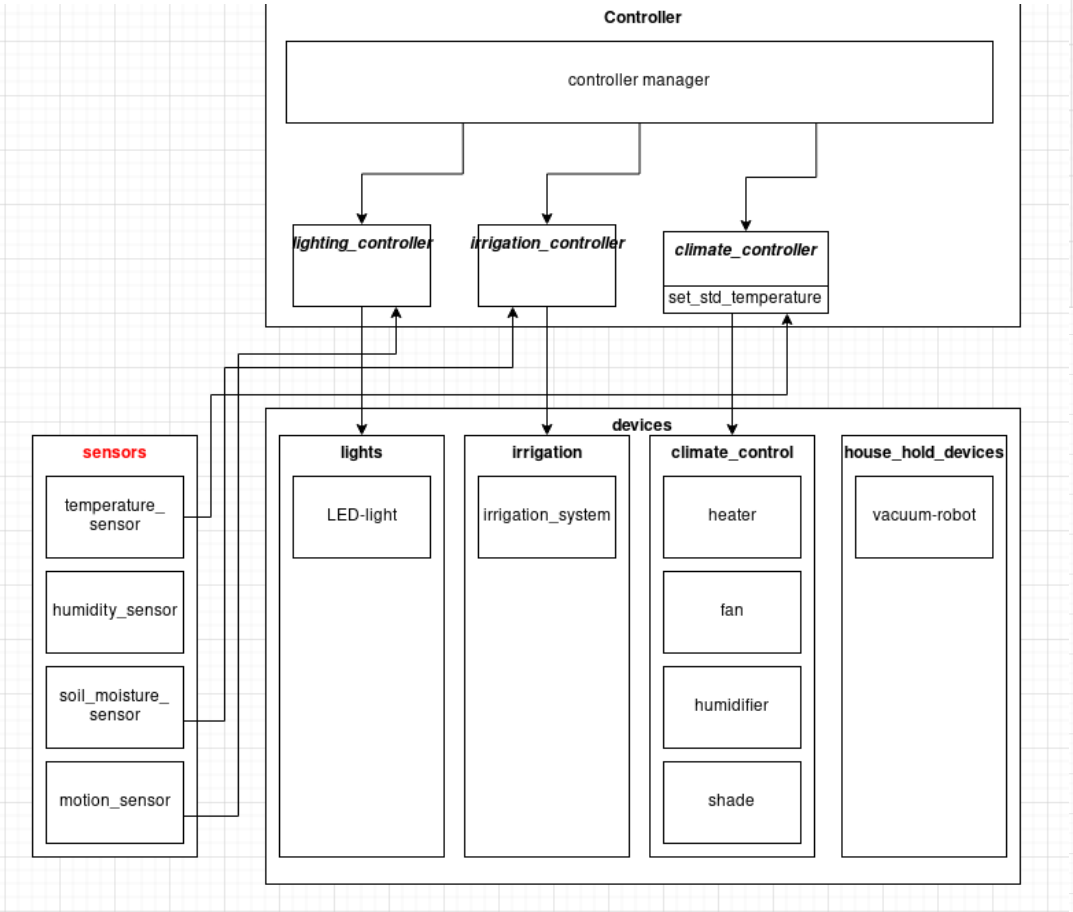
Package Struktur

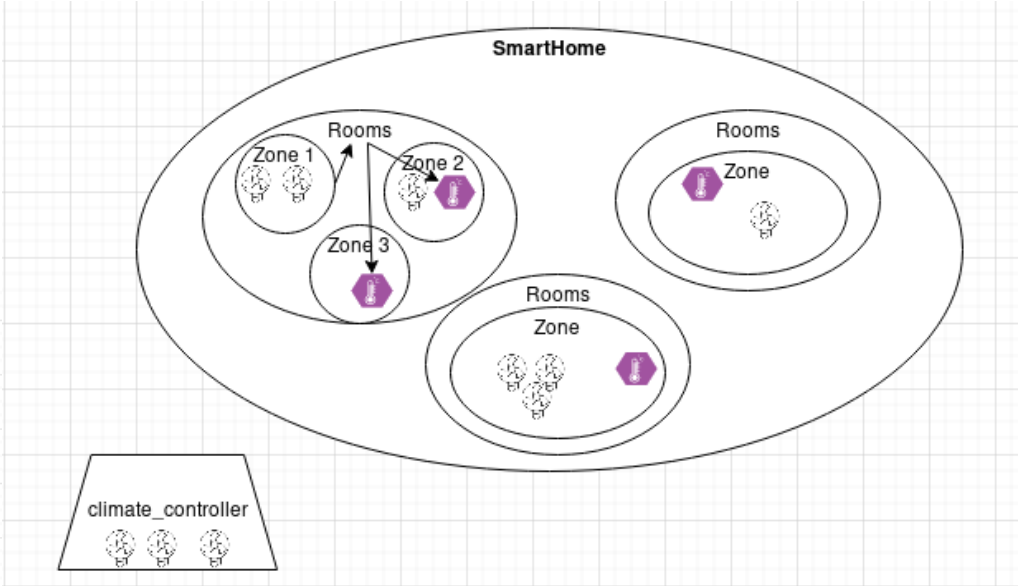


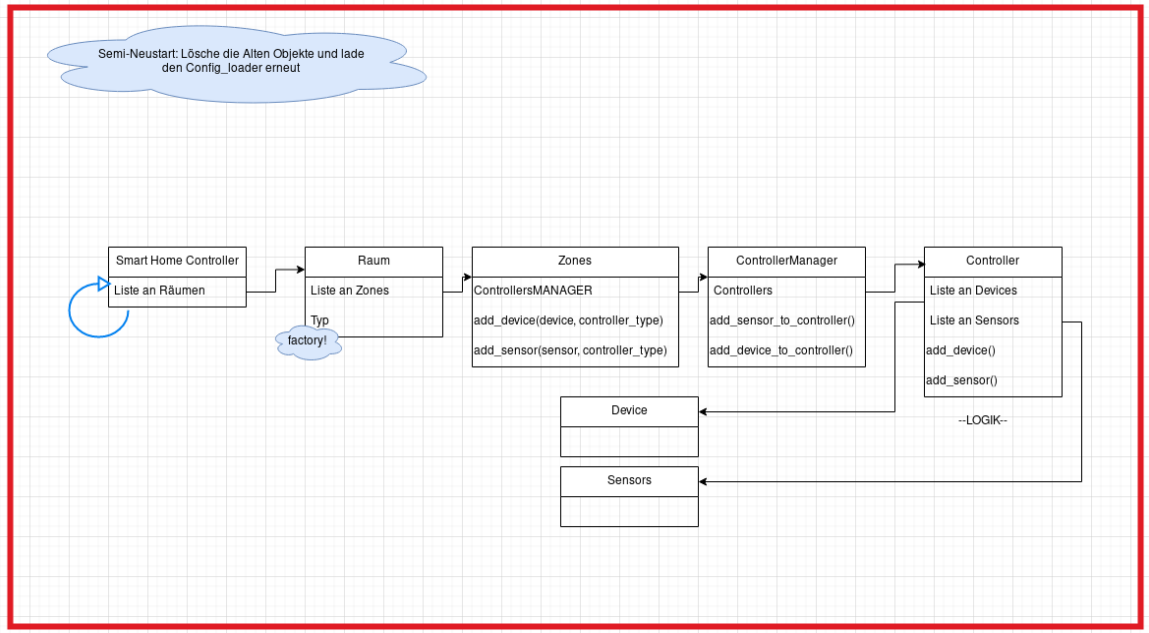
Package Struktur



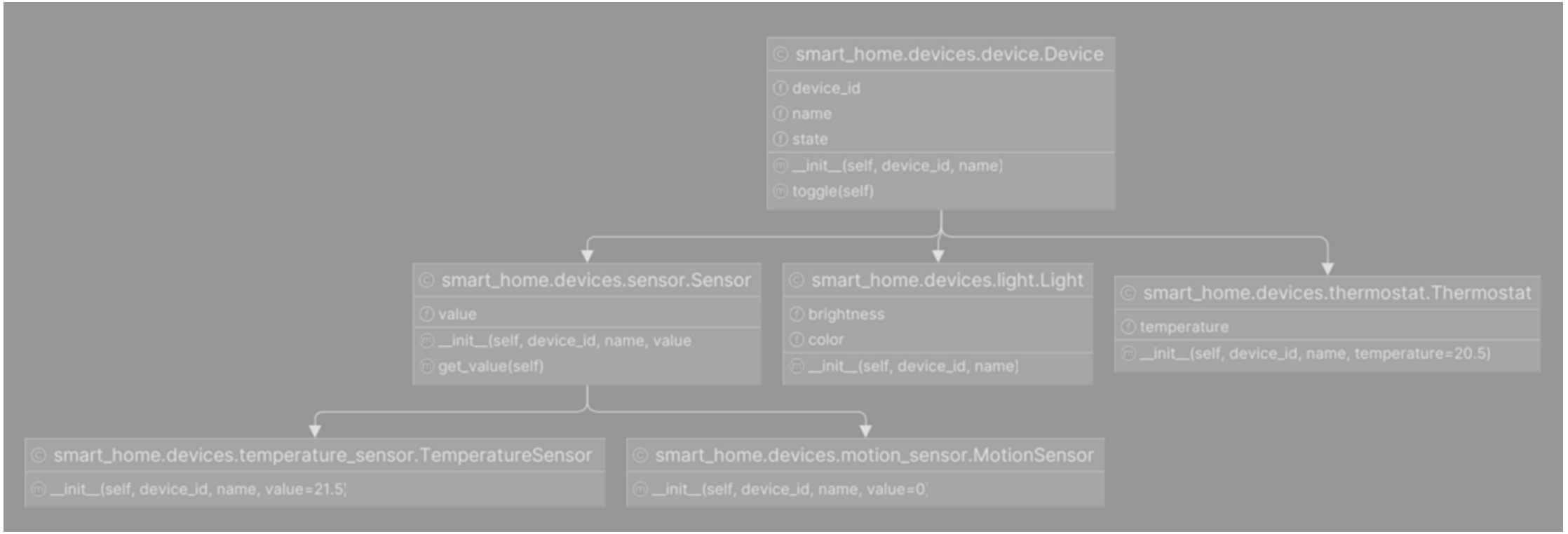
Idee


Controller
controller manager
lighting_controller
irrigation_controller
climate_controller
set_std_temperature
devices
sensors
temperature_ sensor
humidity_sensor
soil_moisture_ sensor
motion_sensor
lights
LED-light
irrigation
irrigation_system
climate_control
heater
fan
humidifier
shade
house_hold_devices
vacuum-robot

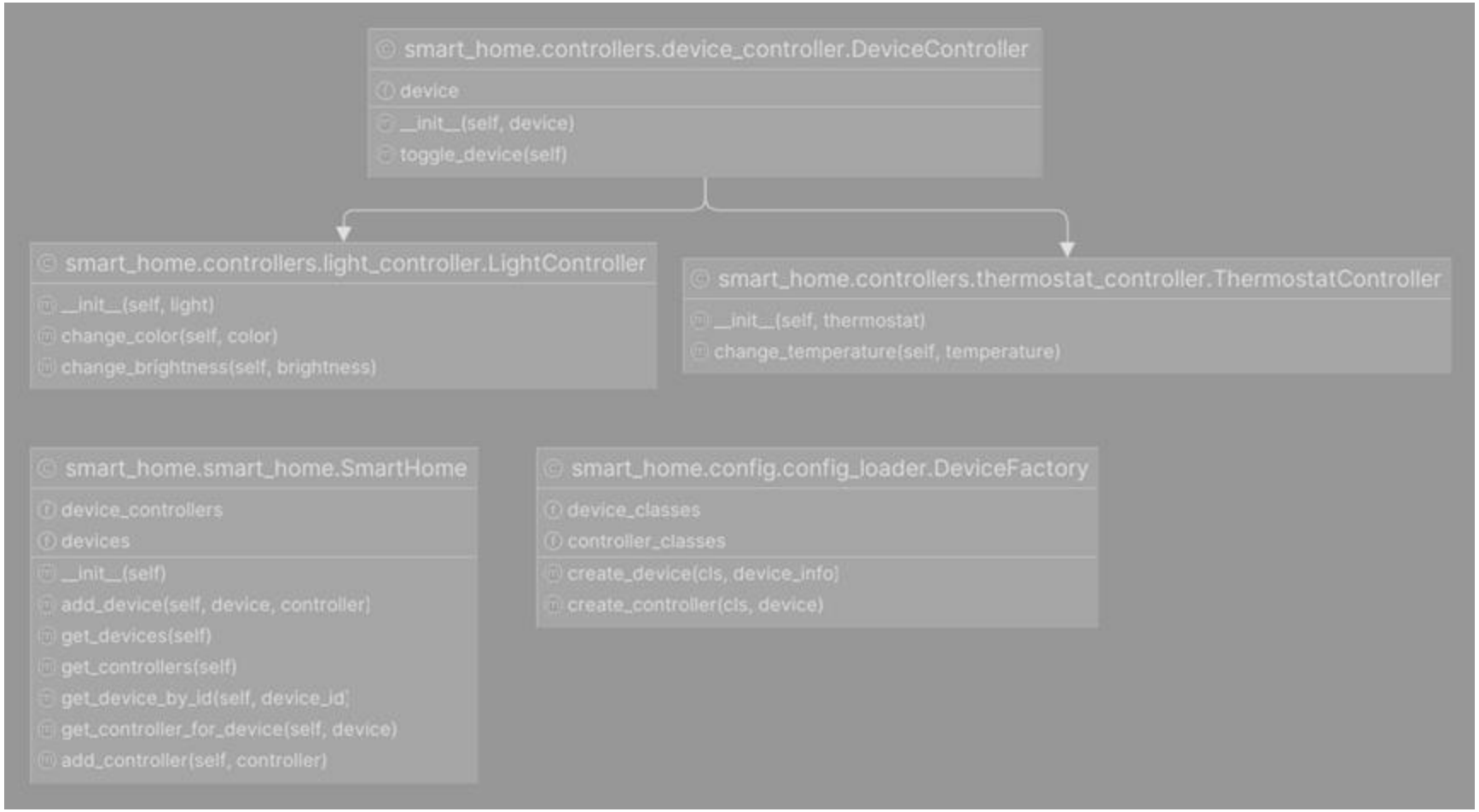

SmartHome
Rooms
Zone 1
Zone 2
Zone 3
Rooms
Zone
Rooms
Zone
climate_controller


Semi-Neustart: Lösche die Alten Objekte und lade den Config_loader erneut
Smart Home Controller
Liste an Räumen
Raum
Liste an Zones
Typ
factory!
Zones
ControllersMANAGER
add_device(device, controller_type)
add_sensor(sensor, controller_type)
ControllerManager
Controllers
add_sensor_to_controller()
add_device_to_controller()
Controller
Liste an Devices
Liste an Sensors
add_device()
add_sensor()
--LOGIK--
Device
Sensors

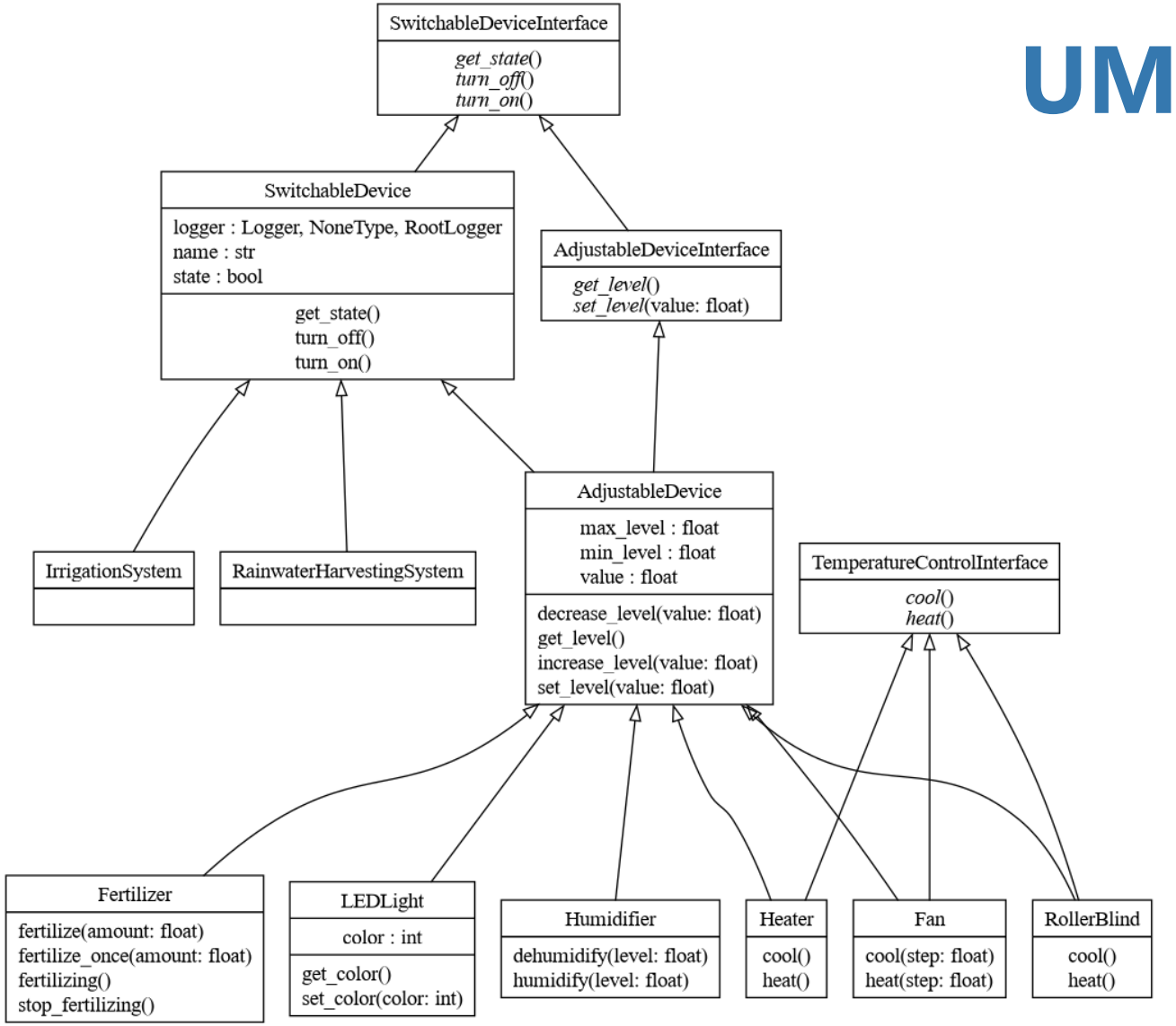
UML

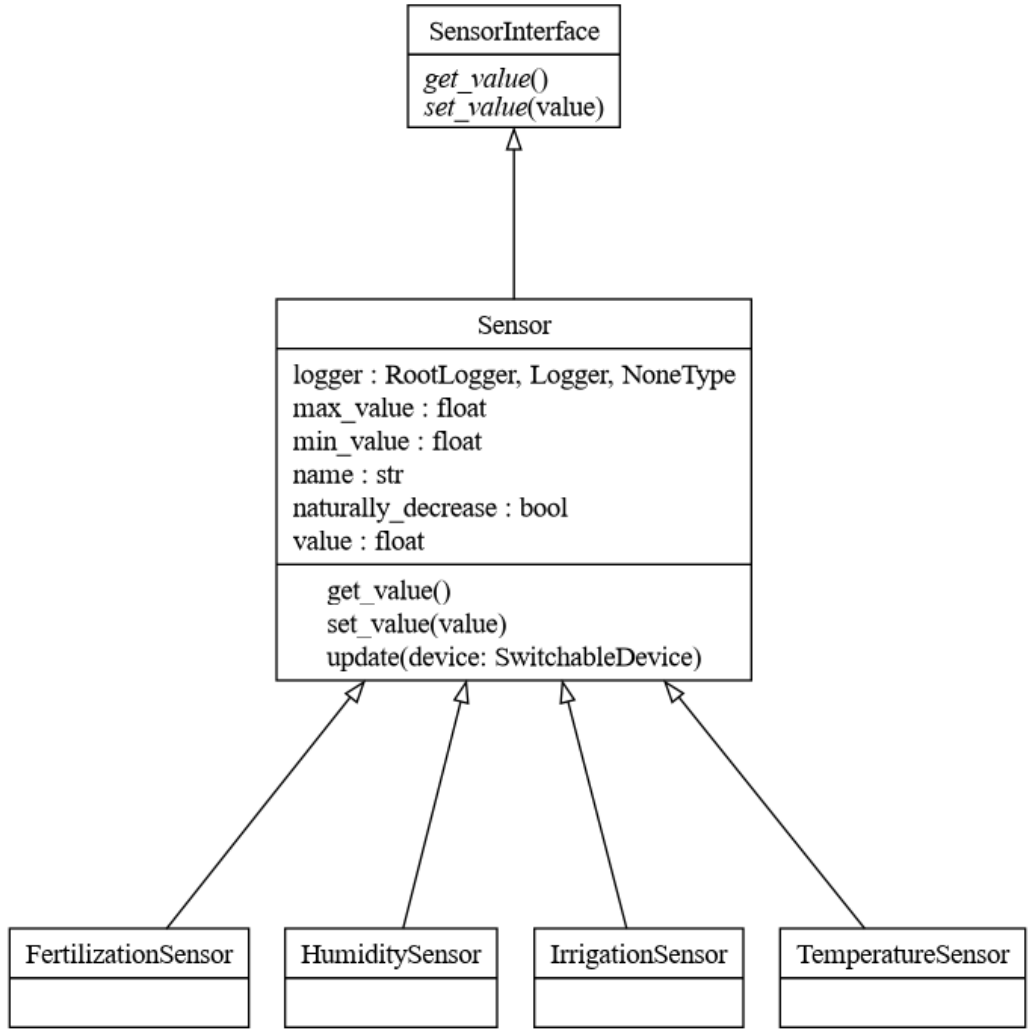

smart_home.devices.device.Device
device_id
name
state
__init__(self, device_id, name)
toggle(self)
smart_home.devices.sensor.Sensor
value
__init__(self, device_id, name, value
get_value(self)
smart_home.devices.light.Light
brightness
color
__init__(self, device_id, name)
smart_home.devices.thermostat.Thermostat
temperature
__init__(self, device_id, name, temperature=20.5)
smart_home.devices.temperature_sensor.TemperatureSensor
__init__(self, device_id, name, value=21.5
smart_home.devices.motion_sensor.MotionSensor
__init__(self, device_id, name, value=0

UML

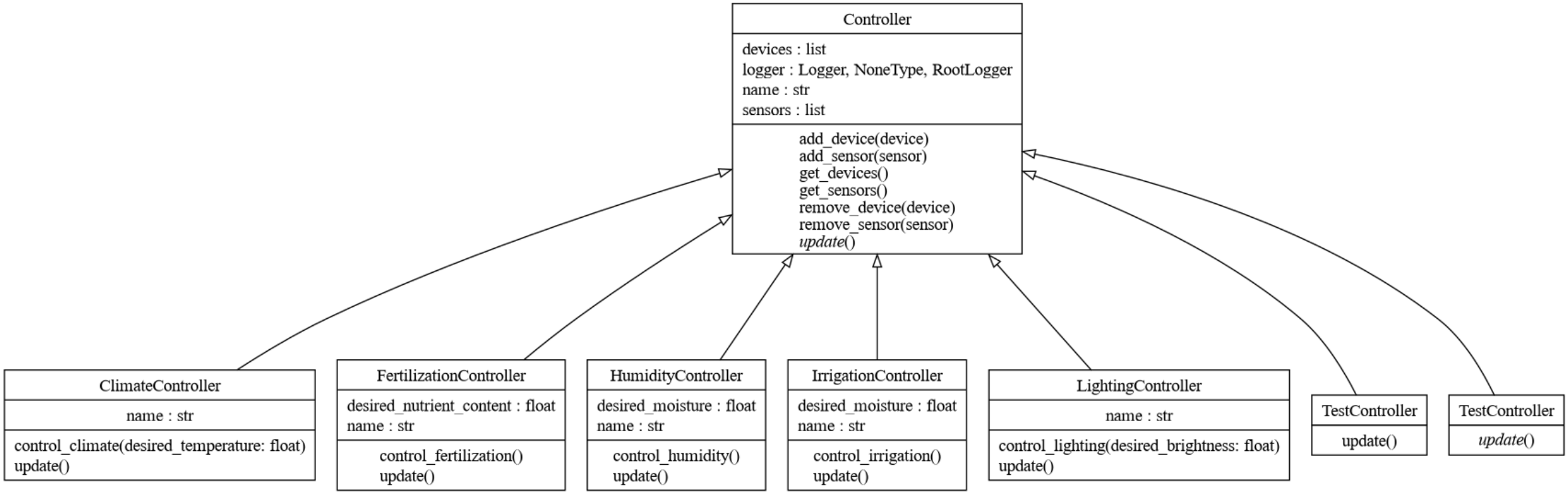

smart_home.controllers.device_controller.DeviceController
device
__init__(self, device)
toggle_device(self)
smart_home.controllers.light_controller.LightController
__init__(self, light)
change_color(self, color)
change_brightness(self, brightness)
smart_home.controllers.thermostat_controller.ThermostatController
__init__(self, thermostat)
change_temperature(self, temperature)
smart_home.smart_home.SmartHome
device_controllers
devices
__init__(self)
add_device(self, device, controller)
get_devices(self)
get_controllers(self)
get_device_by_id(self, device_id)
get_controller_for_device(self, device)
add_controller(self, controller)
smart_home.config.config_loader.DeviceFactory
device_classes
controller_classes
create_device(cls, device_info)
create_controller(cls, device)

UML

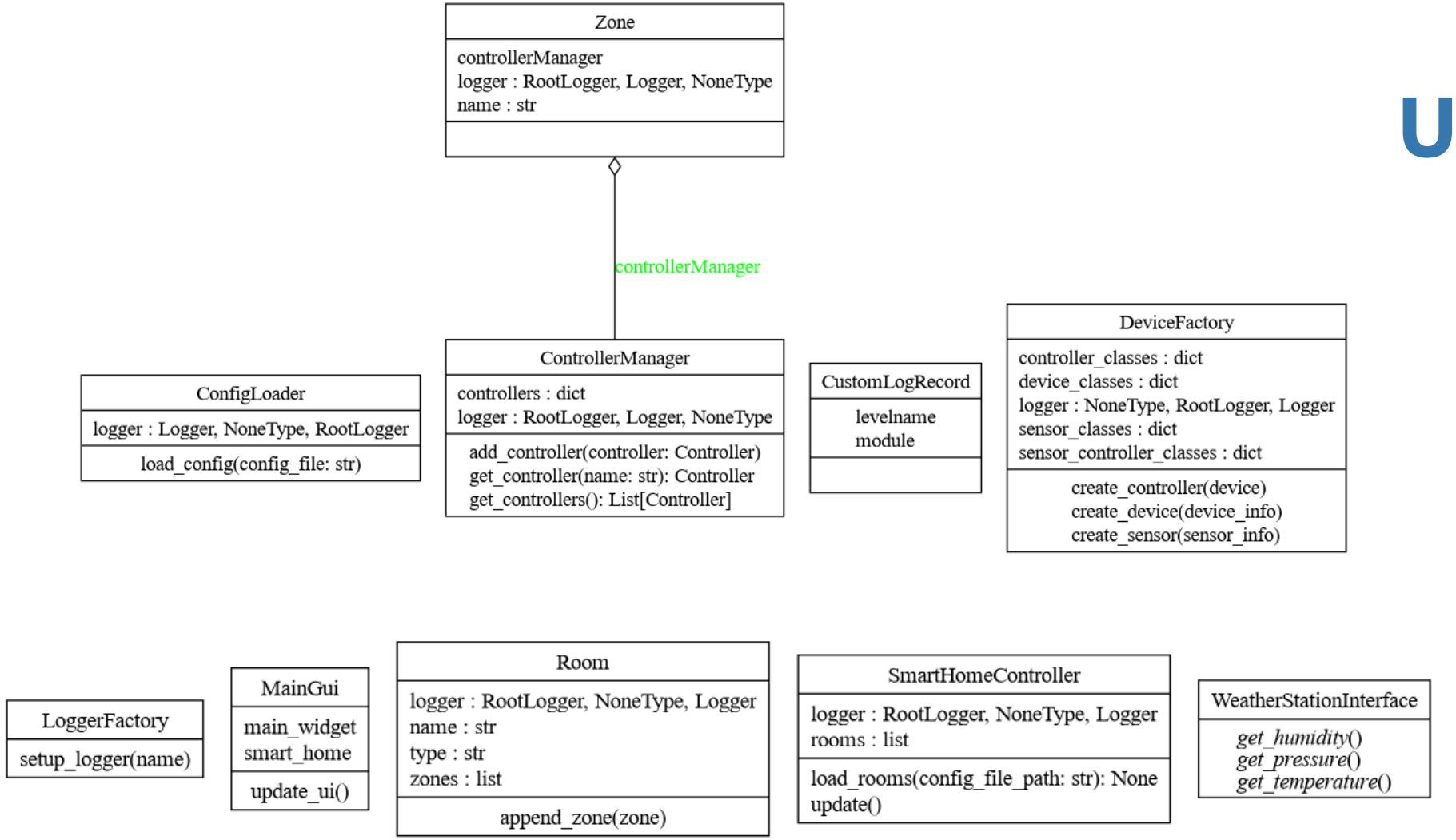

SwitchableDeviceInterface
get_state()
turn_off()
turn_on()
SwitchableDevice
logger : Logger, NoneType, RootLogger
name : str
state : bool
get_state()
turn_off()
turn_on()
AdjustableDeviceInterface
get_level()
set_level(value: float)
IrrigationSystem
RainwaterHarvestingSystem
AdjustableDevice
max_level : float
min_level : float
value : float
decrease_level(value: float)
get_level()
increase_level(value: float)
set_level(value: float)
TemperatureControlInterface
cool()
heat()
Fertilizer
fertilize(amount: float)
fertilize_once(amount: float)
fertilizing()
stop_fertilizing()
LEDLight
color : int
get_color()
set_color(color: int)
Humidifier
dehumidify(level: float)
humidify(level: float)
Heater
cool()
heat()
Fan
cool(step: float)
heat(step: float)
RollerBlind
cool()
heat()

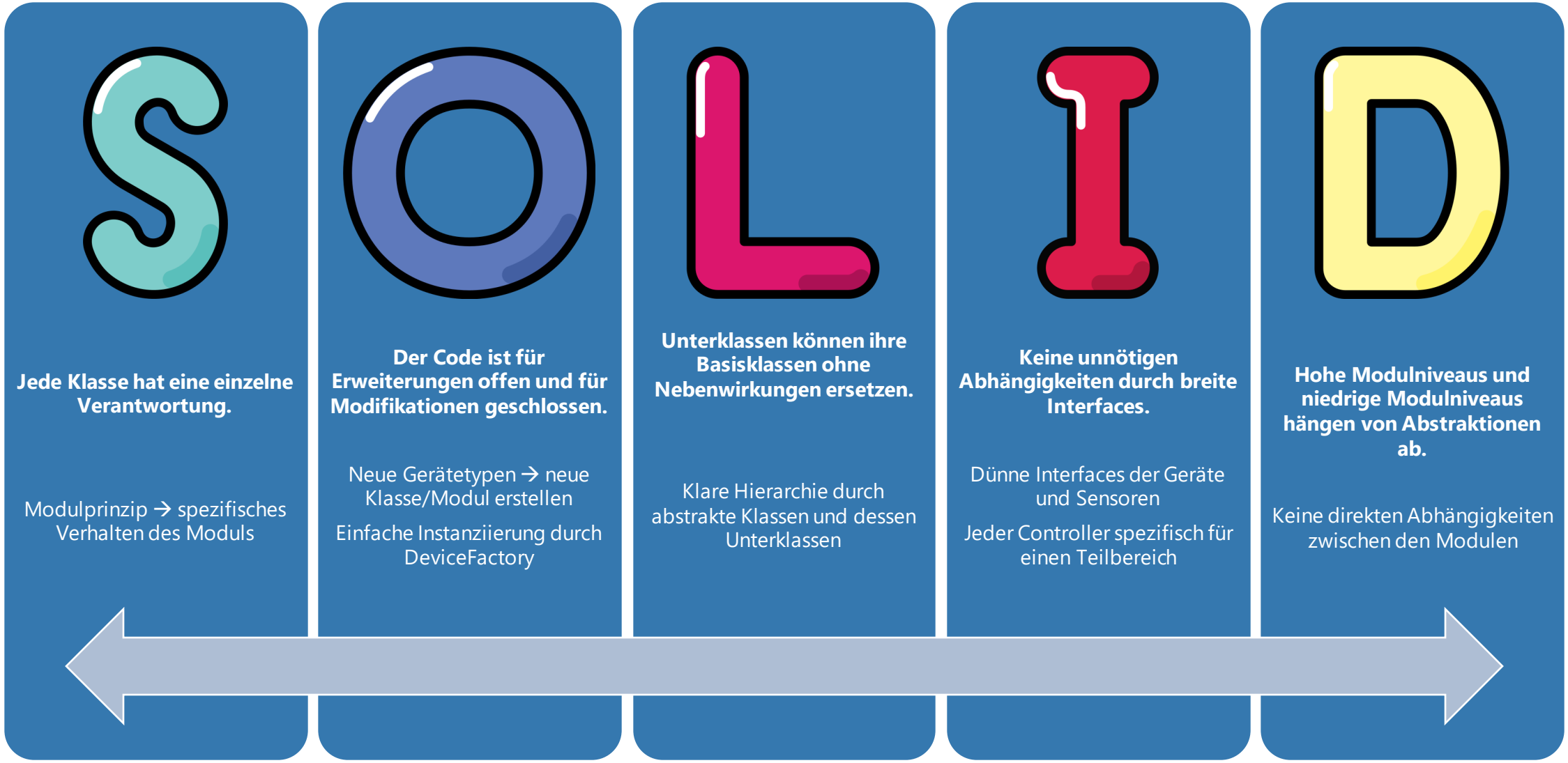

SensorInterface
get_value()
set_value(value)
Sensor
logger : RootLogger, Logger, NoneType
max_value : float
min_value : float
name : str
naturally_decrease : bool
value : float
get_value()
set_value(value)
update(device: SwitchableDevice)
FertilizationSensor
HumiditySensor
IrrigationSensor
TemperatureSensor

UML


Controller
devices : list
logger : Logger, NoneType, RootLogger
name : str
sensors : list
add_device(device)
add_sensor(sensor)
get_devices()
get_sensors()
remove_device(device)
remove_sensor(sensor)
update()
ClimateController
name : str
control_climate(desired_temperature: float)
update()
FertilizationController
desired_nutrient_content : float
name : str
control_fertilization()
update()
HumidityController
desired_moisture : float
name : str
control_humidity()
update()
IrrigationController
desired_moisture : float
name : str
control_irrigation()
update()
LightingController
name : str
control_lighting(desired_brightness: float)
update()
TestController
update()
TestController
update()

UML


Zone
controllerManager
logger : RootLogger, Logger, NoneType
name : str
controllerManager
ControllerManager
controllers : dict
logger : RootLogger, Logger, NoneType
add_controller(controller: Controller)
get_controller(name: str): Controller
get_controllers(): List[Controller]
ConfigLoader
logger : Logger, NoneType, RootLogger
load_config(config_file: str)
CustomLogRecord
levelname
module
DeviceFactory
controller_classes : dict
device_classes : dict
logger : NoneType, RootLogger, Logger
sensor_classes : dict
sensor_controller_classes : dict
create_controller(device)
create_device(device_info)
create_sensor(sensor_info)
LoggerFactory
setup_logger(name)
MainGui
main_widget
smart_home
update_ui()
Room
logger : RootLogger, NoneType, Logger
name : str
type : str
zones : list
append_zone(zone)
SmartHomeController
logger : RootLogger, NoneType, Logger
rooms : list
load_rooms(config_file_path: str): None
update()
WeatherStationInterface
get_humidity()
get_pressure()
get_temperature()


S
O
L
I
D
Jede Klasse hat eine einzelne Verantwortung.
Modulprinzip → spezifisches Verhalten des Moduls
Der Code ist für Erweiterungen offen und für Modifikationen geschlossen.
Neue Gerätetypen → neue Klasse/Modul erstellen
Einfache Instanziierung durch DeviceFactory
Unterklassen können ihre Basisklassen ohne Nebenwirkungen ersetzen.
Klare Hierarchie durch abstrakte Klassen und dessen Unterklassen
Keine unnötigen Abhängigkeiten durch breite Interfaces.
Dünne Interfaces der Geräte und Sensoren
Jeder Controller spezifisch für einen Teilbereich
Hohe Modulniveaus und niedrige Modulniveaus hängen von Abstraktionen ab.
Keine direkten Abhängigkeiten zwischen den Modulen

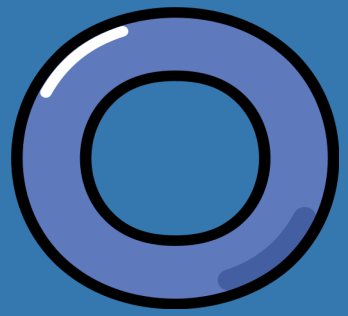
Jede Klasse hat eine einzelne Verantwortung.

Modulprinzip → spezifisches Verhalten des Moduls

- devices
  - climate
    - __init__.py
    - fan.py
    - heater.py
    - roller_blind.py
  - fertilization
    - __init__.py
    - fertilizer.py
  - humidity
    - __init__.py
    - humidifier.py
  - irrigation
    - __init__.py
    - irrigation_system.py
    - rainwater_harvesting_system.py
  - lights
  - __init__.py
  - adjustable_device.py
  - switchable_device.py

```python
class Heater(AdjustableDevice, TemperatureControlInterface):

    930C +1
    def __init__(self, name: str, initial_level: float = 1.0):
        super().__init__(name, initial_level)
        self.logger.info(f'Heater {name} created with initial level {initial_level}')

    930C +1
    def heat(self):
        self.logger.info(f'Heating {self.name} with step 0.1')
        self.increase_level(0.1)

    930C +1
    def cool(self):
        self.logger.info(f'Cooling {self.name} with step 0.1')
        self.decrease_level(0.1)
```

```python
class Humidifier(AdjustableDevice):
    johannadke +1
    def __init__(self, name: str, initial_level: float = 1.0):
        super().__init__(name, initial_level, -1.0, 1.0)
        self.logger.info(f'Humidifier {name} created with initial level {initial_level}')

    1 usage (1 dynamic)  johannadke +1
    def humidify(self, level: float = 0.5):
        self.logger.info(f'Humidifying {self.name} with {level} amount')
        self.set_level(level)
        if not self.get_state():
            self.turn_on()

    1 usage (1 dynamic)  johannadke +1
    def dehumidify(self, level: float = -0.5):
        self.logger.info(f'Dehumidifying {self.name} with {level} amount')
        self.set_level(level)
        if not self.get_state():
            self.turn_on()
```

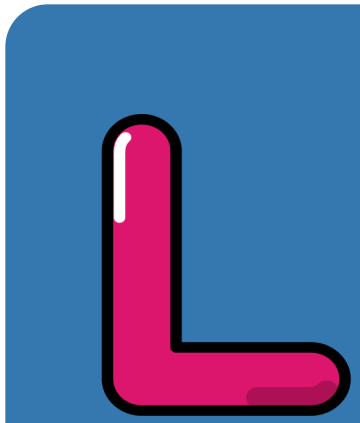


Der Code ist für Erweiterungen offen und für Modifikationen geschlossen.

Neue Gerätetypen → neue Klasse/Modul erstellen
Einfache Instanziierung durch DeviceFactory

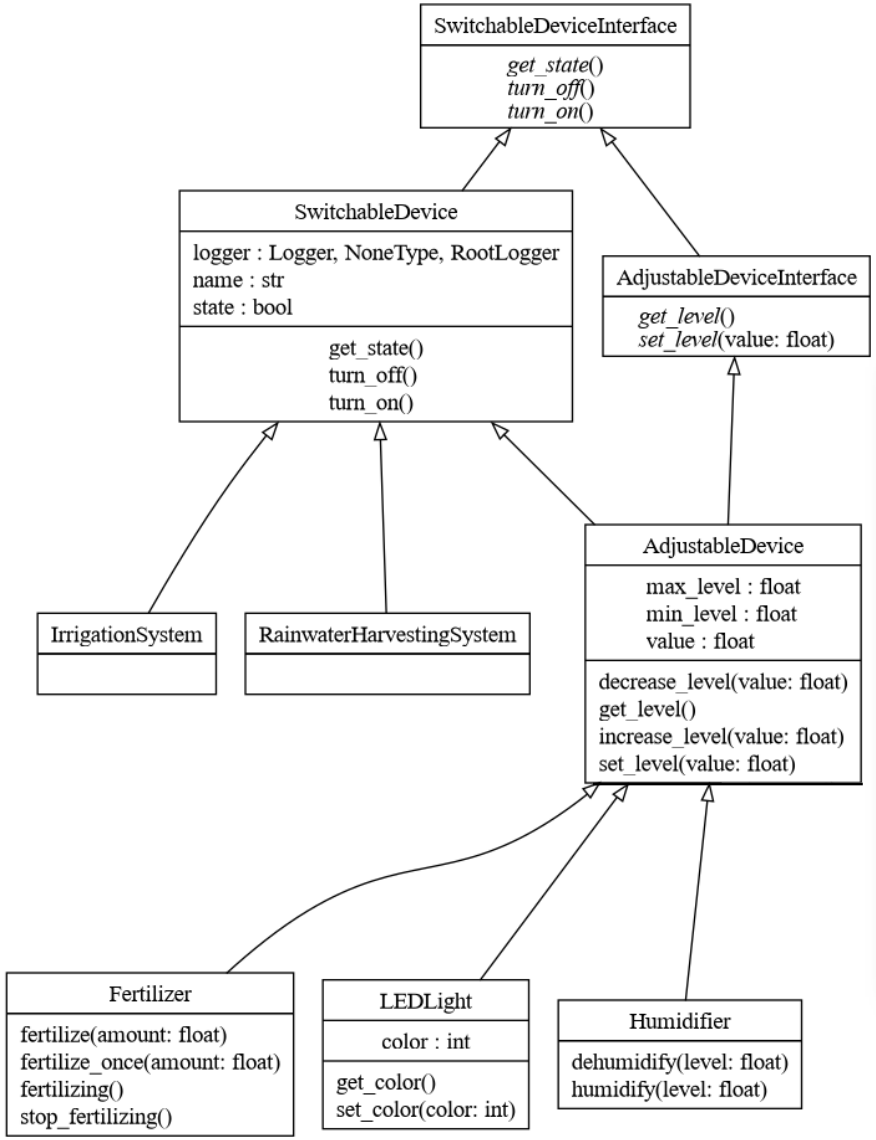
```python
class DeviceFactory:
    logger = LoggerFactory.setup_logger('DeviceFactory')

    device_classes = {
        'led_light': LEDLight,
        'fan': Fan,
        'heater': Heater,
        'humidifier': Humidifier,
        'roller_blind': RollerBlind,
        'fertilizer': Fertilizer,
        'irrigation_system': IrrigationSystem,
        'rainwater_harvesting_system': RainwaterHarvestingSystem
    }

    sensor_classes = {
        'temperature_sensor': TemperatureSensor,
        'fertilization_sensor': FertilizationSensor,
        "humidity_sensor": HumiditySensor,
        "irrigation_sensor": IrrigationSensor,
    }
```

```python
sensor_controller_classes = {
    TemperatureSensor: ClimateController,
    FertilizationSensor: FertilizationController,
    HumiditySensor: HumidityController,
    IrrigationSensor: IrrigationController,
}

controller_classes = {
    Fan: ClimateController,
    LEDLight: LightingController,
    Heater: ClimateController,
    Humidifier: HumidityController,
    RollerBlind: ClimateController,
    Fertilizer: FertilizationController,
    IrrigationSystem: IrrigationController,
    RainwaterHarvestingSystem: IrrigationController,
}
```



Unterklassen können ihre Basisklassen ohne Nebenwirkungen ersetzen.

Klare Hierarchie durch abstrakte Klassen und dessen Unterklassen




```python
class Humidifier(AdjustableDevice):
    johannadke +1
    def __init__(self, name: str, initial_level: float = 1.0):
        super().__init__(name, initial_level, -1.0, 1.0)
        self.logger.info(f'Humidifier {name} created with initial level {initial_level}')

    1 usage (1 dynamic)  johannadke +1
    def humidify(self, level: float = 0.5):
        self.logger.info(f'Humidifying {self.name} with {level} amount')
        self.set_level(level)
        if not self.get_state():
            self.turn_on()

    1 usage (1 dynamic)  johannadke +1
    def dehumidify(self, level: float = -0.5):
        self.logger.info(f'Dehumidifying {self.name} with {level} amount')
        self.set_level(level)
        if not self.get_state():
            self.turn_on()
```

Keine unnötigen Abhängigkeiten durch breite Interfaces.

Dünne Interfaces der Geräte und Sensoren

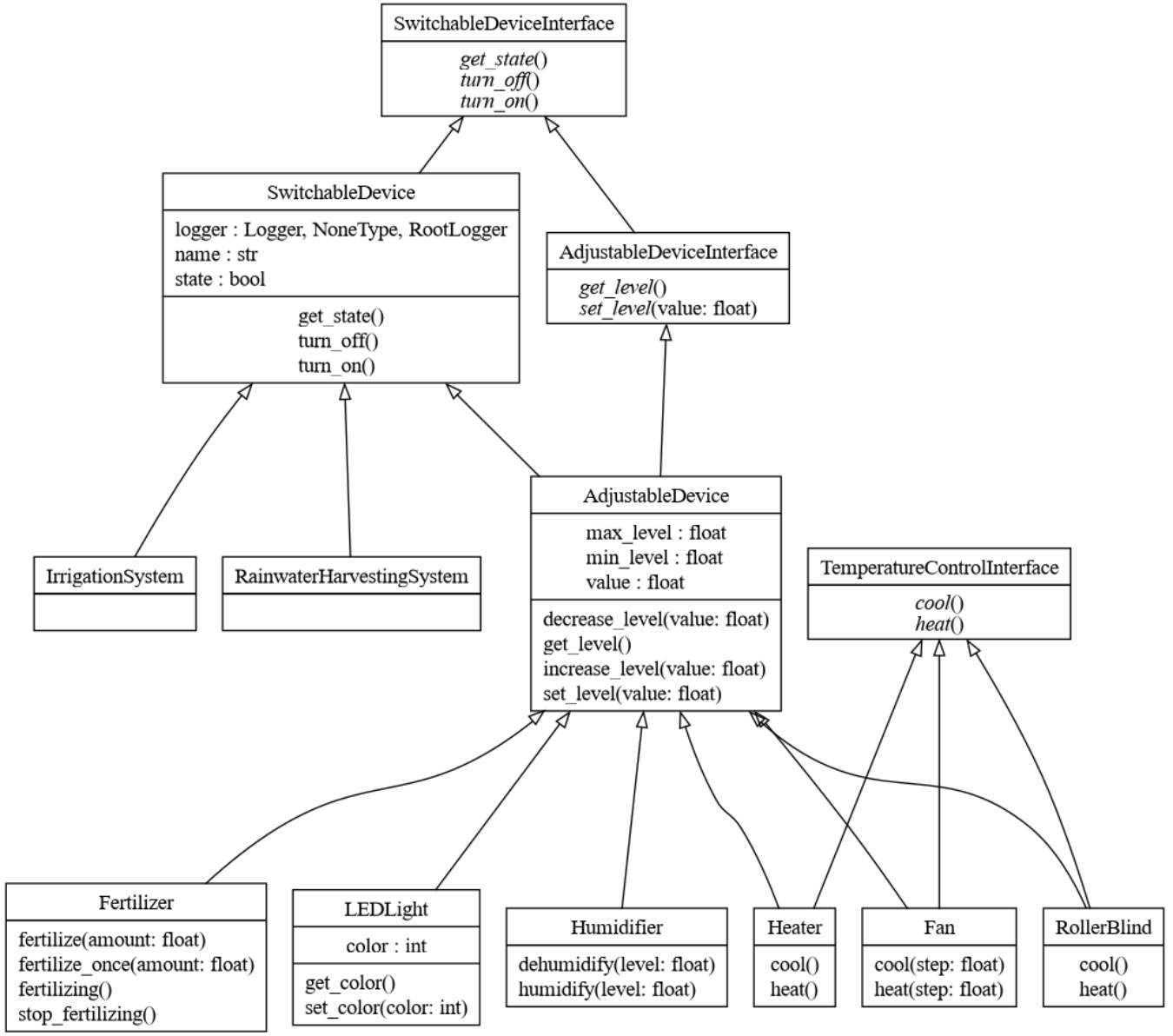
Jeder Controller spezifisch für einen Teilbereich

```python
class HumidityController(Controller):
    name = 'HumidityController'

    johannadke +1
    def __init__(self, desired_moisture: float = 60):
        super().__init__()
        self.desired_moisture = desired_moisture
        self.logger.info(f'Created controller {self.name} with desired moisture {self.desired_moisture}')

    1 usage  johannadke +1
    def control_humidity(self):
        self.logger.info(f'Controlling humidity with desired moisture {self.desired_moisture}')
        for sensor in self.sensors:
            if sensor.get_value() < self.desired_moisture:
                for device in self.devices:
                    device.humidify()
                    sensor.update(device)
            elif sensor.get_value() > self.desired_moisture:
                for device in self.devices:
                    device.dehumidify()
                    sensor.update(device)
            else:
                for device in self.devices:
                    if device.get_state():
                        device.turn_off()
                    sensor.update(device)

    johannadke +1
    def update(self):
        self.logger.info(f'Updating {self.name}..')
        self.control_humidity()
```

```python
class AdjustableDeviceInterface(SwitchableDeviceInterface):

    930C
    @abstractmethod
    def set_level(self, value: float):  # -> float 0.0 - 1.0
        pass

    930C
    @abstractmethod
    def get_level(self):
        pass
```

Hohe Modulniveaus und niedrige Modulniveaus hängen von Abstraktionen ab.

Keine direkten Abhängigkeiten zwischen den Modulen



Patterns

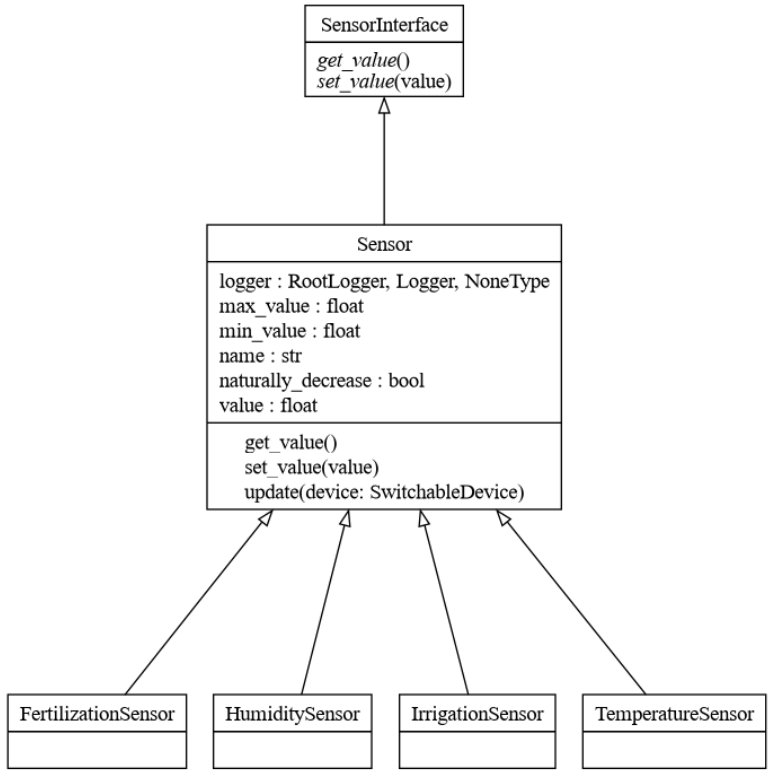
Factory Pattern

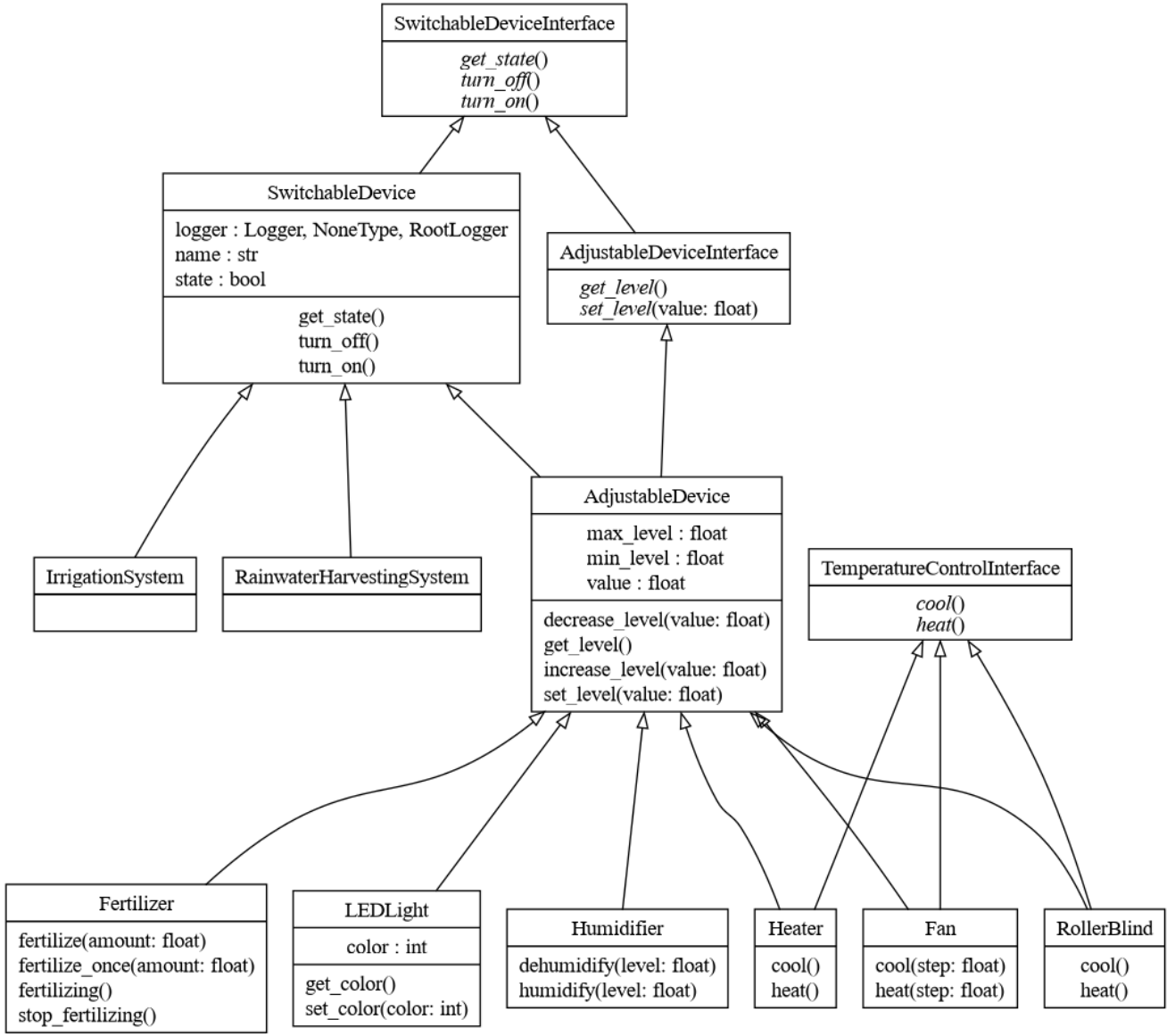
```
sensor_classes = {
    'temperature_sensor': TemperatureSensor,
    'fertilization_sensor': FertilizationSensor,
    "humidity_sensor": HumiditySensor,
    "irrigation_sensor": IrrigationSensor,
}

sensor_controller_classes = {
    TemperatureSensor: ClimateController,
    FertilizationSensor: FertilizationController,
    HumiditySensor: HumidityController,
    IrrigationSensor: IrrigationController,
}
```

```
@classmethod
def create_sensor(cls, sensor_info):
    DeviceFactory.logger.info(f'Creating sensor {sensor_info["name"]}')
    yaml_sensor_type = sensor_info.get('type')
    sensor_class = cls.sensor_classes.get(yaml_sensor_type)

    if sensor_class is not None:
        return sensor_class(sensor_info['name'])

    ConfigLoader.logger.info(f'Invalid device type: {yaml_sensor_type}')
    raise ValueError(f'Invalid device type: {yaml_sensor_type}')
```

Patterns

Template Method Pattern





Patterns

Command Pattern

```
def control_fertilization(self):
    self.logger.info(f"Controlling fertilization with desired nutrient content "
                     f"{self.desired_nutrient_content}")
    for sensor in self.sensors:
        if sensor.get_value() < self.desired_nutrient_content:
            # TODO: Wenn KI-Daten da je nach dessen Ansage düngen
            for device in self.devices:
                device.fertilize(1)
                sensor.update(device)
        elif sensor.get_value() >= self.desired_nutrient_content:
            for device in self.devices:
                if device.get_state():
                    device.stop_fertilizing()
                sensor.update(device)
```

Ausblick