
Lecture 2 - Python Object Oriented Programming & Use Case Modelling

Lecture Information

Master's Degree	Digital Automation Engineering (D.M.270/04)
------------------------	---

Curriculum	Digital Infrastructure
-------------------	------------------------

Lecture Title	Python & Object-Oriented Programming (OOP)
----------------------	--

Author	Prof. Marco Picone (marco.picone@unimore.it)
---------------	--

Academic Year	2025/2026
----------------------	-----------

Last Update	September 2025
--------------------	----------------

License	Creative Commons Attribution 4.0
----------------	----------------------------------

Table of Contents

- 2.1 Object Oriented Programming (OOP) Introduction
 - 2.1.1 What Is Object-Oriented Programming ?
 - 2.1.2 Procedural Programming vs. Object-Oriented Programming
 - * 2.1.2.1 Example: Procedural vs. Object-Oriented Approach
 - * 2.1.2.2 Key Points
 - 2.1.3 Limitation of “Simple” Data Structures
 - 2.1.4 Introducing Classes, Objects and Instances
- 2.2 Defining a Class in Python
 - 2.2.1 The `__init__` Method
 - 2.2.2 Creating an Instance of a Class
 - 2.2.3 Multiple Instances of the Same Class
 - 2.2.3 Adding Attributes to the `__init__` Method
 - 2.2.4 Understanding `self`
 - 2.2.5 Creating Instances with Attributes
 - 2.2.6 Instances Without Parameters
 - 2.2.7 Access Instance Attributes
 - 2.2.8 Change Instance Attributes
 - 2.2.9 Classes & Instance Methods

-
- 2.2.10 Pythonic Class Print Method
 - 2.2.10 Pythonic Dunder Methods
 - 2.3 Class Inheritance
 - 2.3.1 Method Overriding
 - 2.4 Classes & Comments
 - 2.5 Exception Management
 - 2.5.1 Exception Management in Python
 - 2.5.2 Else & Finally
 - 2.5.3 Custom Exceptions
 - 2.6 Object Oriented Programming Smart Home Example (in Python)
 - 2.6.1 Which are the Entities in the Project ?
 - 2.6.2 Sensors & Actuators Characteristics
 - 2.6.3 Open “Issues” and Model Improvements
 - 2.6.4 Updated Modeling with Inheritance - Device Class
 - 2.6.5 Updated Modeling with Inheritance - Sensor Class
 - 2.6.6 Updated Modeling with Inheritance - Actuator Class
 - 2.6.7 From Sensor Abstraction to TemperatureSensor & HumiditySensor
 - 2.6.8 From Actuator Abstraction to SmartLight
 - 2.6.9 Final Overall Design and Modeling with Inheritance
 - 2.7 Smart Home and Data Manager
 - 2.8 Implementing the Smart Home Class and its Behaviors

2.1 Object Oriented Programming (OOP) Introduction

2.1.1 What Is Object-Oriented Programming ?

Object-Oriented Programming (OOP) is a powerful programming paradigm that organizes software design around data, or **objects**, rather than functions and logic. OOP enables developers to model real-world entities and their interactions, making code more modular, reusable, and easier to maintain.

Key Points of OOP:

- **Objects:**
 - Represent real-world entities (e.g., a person, a car, an email).
 - Bundle together **properties** (attributes/data) and **behaviors** (methods/functions).
- **Properties (Attributes):**

-
- Define the characteristics of an object.
 - * *Example (Person)*: name, age, address
 - * *Example (Email)*: recipient_list, subject, body

- **Behaviors (Methods):**

- Define what an object can do or how it interacts.
 - * *Example (Person)*: walk(), talk(), breathe(), run()
 - * *Example (Email)*: add_attachment(), send()

- **Encapsulation:**

- Combines data and methods that operate on that data within a single unit (object).
- Helps protect the internal state of an object from unintended interference.

- **Abstraction:**

- Hides complex implementation details and exposes only the necessary features of an object.

- **Inheritance:**

- Allows new classes (objects) to acquire the properties and behaviors of existing ones, promoting code reuse.

- **Polymorphism:**

- Enables objects to be treated as instances of their parent class rather than their actual class, allowing for flexible and interchangeable code.

Why Use OOP?

- Encourages **modular** and **organized** code structure.
- Facilitates **code reuse** through inheritance and composition.
- Makes it easier to **Maintain** and **Extend** software systems.
- Models complex systems more naturally by mirroring real-world relationships.

Objective:

This course will introduce you to the fundamental tools and design principles of OOP, empowering you to structure your code effectively and laying the groundwork for deeper exploration of advanced object-oriented concepts in the future.

2.1.2 Procedural Programming vs. Object-Oriented Programming

Object-oriented programming (OOP) is a programming paradigm that models both tangible real-world entities (like cars, sensors, or users) and abstract relationships (such as companies and employees, or students and teachers) as **software objects**. Each object encapsulates both **data** (attributes) and **behavior** (methods), allowing you to represent complex systems in a modular and intuitive way.

In contrast, **procedural programming** organizes code as a sequence of instructions or function calls, much like following a **recipe**: you define a set of steps that the program executes in order to accomplish a task. While this approach works well for simple or linear problems, it can become unwieldy as systems grow in complexity.

2.1.2.1 Example: Procedural vs. Object-Oriented Approach

Procedural Programming Example:

```
# Representing a car using procedural programming
car_make = "Tesla"
car_model = "Model 3"
car_year = 2023

def start_car(make, model):
    print(f"{make} {model} is starting...")

start_car(car_make, car_model)
```

Object-Oriented Programming Example:

```
# Representing a car using object-oriented programming
class Car:
    def __init__(self, make, model, year):
        self.make = make
        self.model = model
        self.year = year

    def start(self):
        print(f"{self.make} {self.model} is starting...")

my_car = Car("Tesla", "Model 3", 2023)
my_car.start()
```

2.1.2.2 Key Points

- **OOP models both data and behavior together** in objects, making code more modular and easier to maintain.
- **Procedural programming** separates data and functions, which can lead to less organized code as complexity increases.
- **Objects are central in OOP:** they represent both the structure (data) and the capabilities (methods) of entities in your program.
- **OOP mirrors real-world relationships**, making it easier to design and reason about complex systems.

Fundamental Takeaways:

- **Encapsulation:** Objects bundle data and behavior, hiding internal details.
- **Modularity:** Code is organized into reusable, self-contained objects.
- **Abstraction:** Complex systems are simplified by modeling only relevant features.
- **Reusability:** Objects and classes can be reused and extended, reducing duplication.
- **Maintainability:** Changes are easier to manage due to clear structure and separation of concerns.

By adopting OOP, you can build software that is more robust, flexible, and aligned with real-world concepts.

2.1.3 Limitation of “Simple” Data Structures

Traditional data types and data structures in Python—such as **numbers**, **strings**, and **lists**—are well-suited for representing simple pieces of information. For example, you might use a number to store the cost of an apple, a string for the name of a poem, or a list for your favorite colors.

However, when you need to represent **more complex entities**—such as IoT devices in a deployment—these basic structures quickly become limiting. Imagine you want to track several IoT devices, each with attributes like an **ID**, **manufacturer**, **software version**, **latitude**, and **longitude**.

A common but problematic approach is to use lists to store this information:

```
temperature_sensor = ["device-0001", "acme-inc", "v1.0.0", 10.44121341,  
↪ 44.2132112]
```

```

humidity_sensor = ["device-0002", "acme-inc", "v0.2-rc", 10.54121341,
                   ↵ 44.3132112]
light_switch_actuator = ["device-0003", "acme-inc", "v0.1.1-rc",
                           ↵ 10.64121341, 45.7132112]

```

While this method works for small examples, it introduces several **limitations**:

- **Lack of clarity:** It's not obvious what each element in the list represents without referring to documentation or comments.
- **Error-prone:** Mixing up the order of elements or forgetting an attribute can lead to subtle bugs.
- **Difficult to extend:** Adding new attributes (e.g., device status or last communication time) requires updating every list and all related code.
- **No behavior:** Lists can only store data; they cannot encapsulate behaviors or operations related to the device.

Key Takeaways:

- Basic data structures are insufficient for modeling complex, real-world entities.
- Lists and tuples lack explicit meaning for each attribute, making code harder to read and maintain.
- As complexity grows, maintaining and extending such representations becomes increasingly difficult.
- There is no way to associate behaviors (methods) with the data, limiting code organization and reusability.

To effectively model complex entities like IoT devices, you need a way to group related data and behaviors together in a clear, maintainable, and extensible manner. This is where **Object-Oriented Programming (OOP)** becomes essential.

2.1.4 Introducing Classes, Objects and Instances

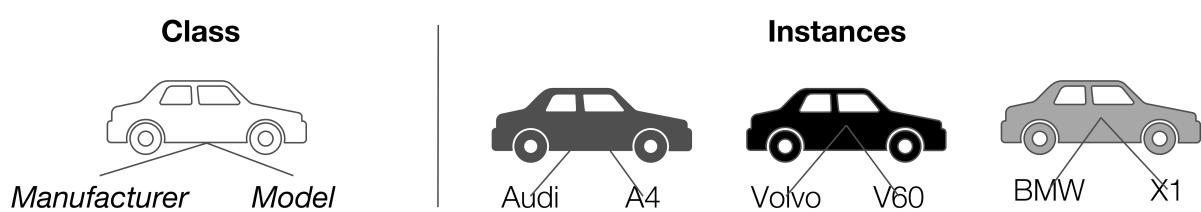


Figure 2.1: Simple representation of the relationship between Classes and Objects (Instances) for a Car.

Classes are fundamental building blocks in object-oriented programming. A **class** serves as a **blueprint** or **template** for creating user-defined data structures. It defines the **attributes** (data) and **methods** (behaviors) that characterize a particular type of object, but it does **not** hold any actual data itself.

- **Class:**

- Specifies what data (attributes) and behaviors (methods) an object should have.
- Acts as a **blueprint**—it describes *how* something should be structured, but does not represent any specific item.

For example, a Car class might define that every car has a manufacturer and a model, and can perform actions like start() or drive(). However, the class itself does **not** represent any particular car.

- **Object (Instance):**

- An **object** (or **instance**) is a **concrete realization** of a class.
- When you create an object from a class, you are building an **instance** that contains **actual data**.
- Each instance has its own unique values for the attributes defined by the class.

For example, if Car is the class, then my_car = Car("Audi", "A4") creates an **instance** of the Car class. This instance represents a specific car with the manufacturer "Audi" and model "A4".

Key Points:

- A **class** is a **template**; an **object** (or **instance**) is a **real, usable entity** created from that template.
 - You can create multiple objects from the same class, each with different attribute values.
 - Classes define **what** data and behaviors objects will have; objects hold the **actual data** and can perform the defined behaviors.
-

2.2 Defining a Class in Python

All **class definitions** in Python begin with the `class` keyword, followed by the **class name** (written in **CapitalizedWords** notation by convention) and a colon. Any code that is **indented** beneath the class definition becomes part of the class body.

Here's a simple example of a Car class:

```
class Car:  
    pass
```

- The body of the Car class contains only the pass statement for now just to illustrate the syntax of a class definition.
- pass acts as a **placeholder**, indicating where code will be added in the future. It allows the class to be defined without causing errors, even if it currently has no attributes or methods.

Note:

By convention, Python class names use **CapitalizedWords** notation (also known as PascalCase or UpperCamelCase). For example, a class representing a generic sensor device should be named SensorDevice.

2.2.1 The `__init__` Method

The Car class isn't very useful yet because it doesn't define any properties or behaviors. To make it more meaningful, we can add **attributes** such as manufacturer and model (keeping it simple for now).

These attributes are defined and initialized in a special method called `__init__()`. The `__init__()` method is known as the **constructor** and is automatically called whenever a new object (instance) of the class is created. It sets up the initial state of the object by assigning values to its properties.

Key aspects of the `__init__()` method:

- The first parameter of `__init__()` is always `self`, which refers to the instance being created.
- You can add additional parameters to `__init__()` to initialize object attributes.
- The `__init__()` method allows each object to have its own unique data.

Example: Defining an empty `__init__()` method in the Car class

```
class Car:  
    def __init__(self):  
        pass # Placeholder for future initialization code
```

The pass statement is used here as a placeholder, indicating that the method doesn't do anything yet but will be expanded later.

2.2.2 Creating an Instance of a Class

To create an instance of a class, you simply call the class as if it were a function. This invokes the `__init__()` method and creates a new object.

```
my_car = Car()
```

In this example, `my_car` is an instance of the `Car` class. The `__init__()` method is called automatically when the instance is created.

2.2.3 Multiple Instances of the Same Class

Let's see what happens when you create **multiple instances** of the same class:

```
car_1 = Car()
car_2 = Car()

print(car_1)
print(car_2)
print(car_1 == car_2)
```

Output:

```
<__main__.Car object at 0x106702d30>
<__main__.Car object at 0x108702d40>
False
```

Explanation:

- `car_1` and `car_2` are two **separate objects** (instances) of the `Car` class.
- When you print each object, Python displays their **memory addresses**, showing that they occupy different locations in memory.
- Comparing `car_1 == car_2` returns **False** because, by default, two different instances are considered **not equal**, even if they are created from the same class and have the same (empty) content.

Key Points:

- **Each call to the class creates a new, independent object.**
- **Instances have unique memory addresses**—they are distinct entities in memory.

-
- **Default equality (==) checks if two variables refer to the same object, not if their contents are the same.**
 - **Objects created from the same class can behave differently if their attributes are set differently.**

This demonstrates that **classes are blueprints**, and each time you instantiate a class, you get a new, unique object—even if no attributes have been set yet.

Note: Dedicated methods can be implemented to customize how instances are compared for equality based on their attributes, but by default, they are compared by their identity (memory address). In Python, this can be achieved by overriding the `__eq__` method in the class definition. This method allows you to define custom logic for comparing two instances of the class based on their attributes rather than their memory addresses.

2.2.3 Adding Attributes to the `__init__` Method

The `__init__()` method in a Python class acts as a constructor and is designed to initialize the attributes of each new object (instance) you create. By defining **parameters** in the `__init__()` method, you can **require that specific information be provided when an object is instantiated**.

Inside the method, you assign these parameter values to instance attributes using the `self` keyword, which ensures that each object maintains its own unique data. This approach allows you to create flexible and reusable classes, where each instance can have different attribute values based on the arguments passed during creation.

For example, by adding `manufacturer` and `model` as parameters to the `__init__()` method of the `Car` class, you ensure that every `Car` object is initialized with its own manufacturer and model information.

```
class Car:  
    def __init__(self, manufacturer, model):  
        self.manufacturer = manufacturer  
        self.model = model
```

Key Aspects:

- The `__init__()` method is the **constructor** of the class and is called automatically when a new `Car` object is created.
- The method's **signature** is indented by four spaces, and its **body** is indented by eight spaces. This indentation is crucial in Python to indicate code blocks and class/method structure.

-
- The first parameter, `self`, refers to the **instance** being created.
 - Inside `__init__()`, attributes are defined using `self.attribute_name = value`. For example:
 - `self.manufacturer = manufacturer` creates an attribute `manufacturer` and assigns it the value passed to the constructor.
 - `self.model = model` creates an attribute `model` and assigns it the corresponding value.
 - These attributes are **unique to each instance** of the class.
-

2.2.4 Understanding `self`

In Python, `self` is a conventional name used to refer to the instance of the class within its methods. It acts as a reference to the current object, allowing you to access its attributes and methods.

When you define a method in a class, the first parameter must always be `self`, which represents the instance that the method is being called on. This allows you to differentiate between instance variables (attributes) and local variables within the method. Here's a breakdown of how `self` works:

- **Instance Reference:**

- `self` refers to the specific instance of the class that is being manipulated. It allows you to access and modify the instance's attributes and call its methods.
- For example, in the `__init__()` method of the `Car` class, `self.manufacturer` refers to the `manufacturer` attribute of the specific `Car` instance being created.
- When you create an instance of the class, Python automatically passes the instance as the first argument to the method, which is why you don't need to provide it explicitly when calling the method.

```
class Car:  
    def __init__(self, manufacturer, model):  
        self.manufacturer = manufacturer # 'self' refers to the instance being  
        # created  
        self.model = model  
  
my_car = Car("Toyota", "Corolla") # 'my_car' is passed as 'self' to  
# __init__()  


---


```

2.2.5 Creating Instances with Attributes

Now that we have defined the `Car` class with attributes, let's create instances of this class with specific values for `manufacturer` and `model`:

```
car_1 = Car("Toyota", "Corolla")
car_2 = Car("Honda", "Civic")

print(car_1.manufacturer, car_1.model)
print(car_2.manufacturer, car_2.model)
```

Output:

```
Toyota Corolla
Honda Civic
```

Explanation:

- When creating `car_1`, we pass "Toyota" and "Corolla" as arguments, which are assigned to the `manufacturer` and `model` attributes, respectively.
 - Similarly, `car_2` is created with its own `manufacturer` and `model`.
 - Each instance maintains its own state, allowing us to access the specific values assigned during creation.
 - As you can see the `self` is visible only inside the class methods, and it is not used when creating instances or accessing attributes from outside the class. You just pass the required arguments to the class constructor (in this case, `manufacturer` and `model`) without mentioning `self`.
-

2.2.6 Instances Without Parameters

In Python, you can create instances of a class without passing any parameters to the constructor. This is possible when the `__init__()` method is defined without any additional parameters (besides `self`). In such cases, the instance will be created with default values or uninitialized attributes. On the other hand, if the `__init__()` method requires parameters, you must provide those arguments when creating an instance.

```
# Attempting to create a Car instance without required arguments
car_1 = Car() # This will cause an error!
```

Output:

```
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    Car()
TypeError: __init__() missing 2 required positional arguments: 'manufacturer' and 'model'
```

Explanation:

- The Car class requires two arguments (manufacturer and model) when creating a new instance.
- If you try to instantiate a Car without providing these arguments, Python raises a TypeError indicating that the required positional arguments are missing.
- Always ensure you supply all required arguments when calling the constructor of a class that defines them in its __init__() method.

Tip:

If you want to allow creating instances without specifying all attributes, you can provide default values in the __init__() method parameters.

```
class Car:
    def __init__(self, manufacturer="Unknown", model="Unknown"):
        self.manufacturer = manufacturer
        self.model = model
```

Now you can create instances without arguments

```
car_1 = Car() # Uses default values
car_2 = Car("Honda", "Civic") # Provides specific values
```

2.2.7 Access Instance Attributes

After you create instances of the Car class, you can **access their attributes** using **dot notation**. This allows you to retrieve or modify the values stored in each object's attributes.

```
car1 = Car("Audi", "A4")
print(f"Car 1 -> Manufacturer: {car1.manufacturer} Model: {car1.model}")
```

Output:

```
Car 1 -> Manufacturer: Audi Model: A4
```

Or for example you can save the attributes in variables

```
car2 = Car("BMW", "X3")
manufacturer = car2.manufacturer
model = car2.model
print(f"Car 2 -> Manufacturer: {manufacturer} Model: {model}")
```

Key Aspects:

- **Dot notation** (`object.attribute`) is used to access or modify instance attributes.
- All instances of the `Car` class are **guaranteed to have the defined attributes** (e.g., `.manufacturer`, `.model`).
- You can **confidently use these attributes** in your code, knowing they will always be present and hold the values assigned during object creation.
- This approach **improves code reliability and readability** compared to using basic data structures like lists or dictionaries.

By organizing data with classes, you ensure that each object has a consistent structure, making your code more robust and maintainable.

2.2.8 Change Instance Attributes

Although instance attributes are **guaranteed to exist** after initialization, their values can be **changed dynamically** at any time. This flexibility allows you to update the state of an object as your program runs.

Example: Modifying Instance Attributes

```
car1 = Car("Audi", "A4")
print(f"Car 1 -> Manufacturer: {car1.manufacturer} Model: {car1.model}")

# Change the model attribute
car1.model = "Q3"
print(f"Car 1 -> Manufacturer: {car1.manufacturer} Model: {car1.model}")
```

Output:

```
Car 1 -> Manufacturer: Audi Model: A4
Car 1 -> Manufacturer: Audi Model: Q3
```

In this example:

-
- We first create a Car object with the model "A4".
 - We then update the model attribute to "Q3" using dot notation.
 - The change is immediately reflected when we print the object's attributes again.

Key Aspects: - **Instance attributes can be modified at any time** after object creation. - **Dot notation** (object.attribute = new_value) is used to update attribute values. - **Each instance maintains its own state**—changing one object's attributes does not affect others. - **Dynamic updates** allow objects to reflect changes in real-world scenarios (e.g., updating a car's model after an upgrade).

This ability to modify attributes is a core feature of object-oriented programming, enabling your objects to evolve and respond to changes throughout your program.

2.2.9 Classes & Instance Methods

Instance Methods are functions defined within a class that operate on individual instances of that class. Like the `__init__()` constructor, the first parameter of every instance method is always `self`, which refers to the specific object the method is called on.

```
class Car:
```

```
    def __init__(self, manufacturer, model):
        self.manufacturer = manufacturer
        self.model = model

        # Instance method returning a description of the car
    def description(self):
        return f"Manufacturer: {self.manufacturer} Model: {self.model}"
```

Key Points: - **Instance methods** are called on objects (instances) of a class, not on the class itself. - The `self` parameter allows the method to access and modify the instance's attributes. - Instance methods can perform operations using the data stored in the object. - You invoke an instance method using dot notation: `car1.description()`. - Instance methods help encapsulate behavior that is specific to each object, making your code modular and organized.

2.2.10 Pythonic Class Print Method

The `__str__()` method in Python is a special instance method that defines how an object is represented as a string. By default, printing an instance of a class displays its memory address, which is not informative. Implementing the `__str__()` method allows you to customize the string output, making it more meaningful and user-friendly.

Replacing a custom `.description()` method with `__str__()` is considered more Pythonic, as it integrates seamlessly with built-in functions like `print()` and `str()`. This approach enhances code readability and usability, especially when debugging or logging object information.

You can change what gets printed by defining a special instance method called `str()`:

```
# Replace description with __str__() method
def __str__(self):
    return f"Manufacturer: {self.manufacturer} Model: {self.model}"
```

Key Points: - `__str__()` gives a readable string for an object. - **Overrides** the default memory address output for instances. - **Preferred** over custom methods for string representation. - **Enhances** code clarity and debugging. - **Called automatically** by `print()` and `str()`.

2.2.10 Pythonic Dunder Methods

Dunder Methods (short for “double underscore methods”) are special methods in Python that begin and end with double underscores, such as `__init__()` and `__str__()`. These methods are also known as **magic methods** or **special methods**. They allow you to customize the behavior of your classes and objects, enabling integration with Python’s built-in functions and operators.

Dunder methods are automatically invoked by Python in specific situations. For example, `__init__()` is called when an object is created, while `__str__()` is called when you print an object or convert it to a string. There are many other dunder methods that let you define how your objects behave when compared, added, indexed, iterated over, or used in other common operations.

Mastering dunder methods is essential for advanced object-oriented programming in Python, as they provide powerful ways to make your classes behave more like built-in types and interact seamlessly with Python’s language features.

For a comprehensive list and detailed documentation, refer to the Python Data Model documentation.

-
- Key Points:** - **Dunder methods** start and end with double underscores (e.g., `__init__`, `__str__`).
- They **customize class behavior** for built-in operations and functions. - Examples include **object creation** (`__init__`), **string representation** (`__str__`), **comparison** (`__eq__`), **addition** (`__add__`), and more.
- Dunder methods are **automatically called** by Python in specific contexts.
- Understanding and using dunder methods is **crucial for advanced OOP** and creating Pythonic, robust classes.
-

2.3 Class Inheritance

Inheritance is a fundamental concept in object-oriented programming that allows one class (the **child class**) to acquire the attributes and methods of another class (the **parent class**). This mechanism promotes **code reuse**, **modularity**, and **extensibility** by enabling you to build new classes based on existing ones.

When you create a child class, it automatically inherits all the properties and behaviors defined in its parent class. However, child classes are not limited to what they inherit—they can also **override** existing methods or **extend** the parent class by adding new attributes and methods that are unique to themselves. This flexibility allows you to customize and specialize behavior for different types of objects while maintaining a consistent structure.

For example, you might have a generic `Vehicle` class that defines common attributes like `manufacturer` and `model`, and methods such as `start()`. You can then create a `Car` class that inherits from `Vehicle`, gaining all its features, but also adding specific attributes (e.g., `number_of_doors`) or overriding methods to provide specialized behavior.

To inspect the type of an object and its relationship to classes, Python provides built-in functions:

- Use `type(object)` to determine the exact class of an object:

```
type(car)
# Output: <class '__main__.Car'>
```

- Use `isinstance(object, ClassName)` to check if an object is an instance of a specific class (including parent or child classes):

```
isinstance(car1, Car)
# Output: True
```

Here's an improved example of class inheritance in Python:

```
class ElectricCar(Car):
    """
    ElectricCar inherits from Car and adds electric-specific attributes.
    """

    def __init__(self, manufacturer, model, battery_capacity_kwh):
        # Initialize attributes from the parent Car class
        super().__init__(manufacturer, model)
        # Add new attributes specific to ElectricCar
        self.battery_capacity_kwh = battery_capacity_kwh
        self.battery_level = 100 # Battery level as a percentage
```

Explanation: - ElectricCar inherits all attributes and methods from Car using super(). - Adds new attributes: battery_capacity_kwh and battery_level. - Includes a custom charge() method and a Pythonic __str__() for readable output.

Key Points: - **Inheritance** enables child classes to reuse and extend the functionality of parent classes. - **Child classes** inherit all attributes and methods from their parent, but can also define their own or override inherited ones. - **Code reuse** and **modularity** are enhanced by organizing related classes in hierarchies. - Use **type()** to check the exact class of an object. - Use **isinstance()** to verify if an object is an instance of a particular class or its subclasses. - Inheritance supports the creation of flexible and maintainable software architectures.

2.3.1 Method Overriding

Method overriding is a core feature of object-oriented programming that allows a child class to provide a specific implementation for a method that is already defined in its parent class. This enables you to customize or extend the behavior of inherited methods to suit the needs of the child class.

To override a method, simply define a method in the child class with the **same name** and **signature** as the one in the parent class. When you call this method on an instance of the child class, Python will use the child class's version, effectively replacing the parent's implementation for that object.

Below is an “improved” example of the ElectricCar class, which inherits from the Car class and overrides several methods:

```
import random

class ElectricCar(Car):
    def __init__(self, manufacturer, model, kwh):
        # Call the parent class constructor to initialize common attributes
```

```

super().__init__(manufacturer, model)
# Add new attributes specific to ElectricCar
self.kwh = kwh
self.battery_level = 100 # Battery level as a percentage

# Override the __str__ method to provide a custom string representation
def __str__(self):
    # Use the parent class's __str__ and add electric-specific info
    return f"{super().__str__()} - Kwh: {self.kwh}"

# Override or add new methods specific to ElectricCar
def estimate_air_pollution(self, path_km_value):
    # Electric cars produce zero direct air pollution
    return 0

def measure_battery_level(self):
    # Simulate measuring the battery level with a random value
    self.battery_level = random.randint(10, 100)
    return self.battery_level

```

Explanation

- The `__init__` method uses `super()` to call the parent class constructor, ensuring that common attributes (`manufacturer`, `model`) are properly initialized.
- The `__str__` method is **overridden** to provide a more informative string representation, including the electric car's battery capacity (`kwh`). It also calls the parent's `__str__` using `super()` to avoid code duplication.
- The `estimate_air_pollution` method is overridden to reflect that electric cars produce zero direct air pollution.
- The `measure_battery_level` method is unique to `ElectricCar`, simulating battery measurement with a random value.

Key Points

- **Method overriding** allows child classes to customize or replace inherited behavior.
 - Use `super()` to access parent class methods and avoid code duplication.
 - Overridden methods in the child class take precedence when called on child class instances.
 - Child classes can introduce **new attributes and methods** in addition to overriding existing ones.
 - Overriding supports **polymorphism**, enabling flexible and extensible code design.
-

2.4 Classes & Comments

Proper commenting is **essential** for writing clear, maintainable, and professional Python code. Comments help you and others understand the logic, intent, and structure of your programs, making collaboration and future updates much easier.

In Python, there are two main ways to add comments:

Single-Line Comments

Single-line comments begin with the # symbol. Everything after # on the same line is ignored by the Python interpreter. Use single-line comments for brief explanations, clarifying complex code, or leaving notes for future reference.

```
# Calculate the area of a circle
radius = 5
area = 3.14 * radius ** 2 # Area formula: πr2
```

Multi-Line Docstrings

A **docstring** is a string literal placed immediately after the definition of a function, method, class, or module. Docstrings are enclosed in triple quotes ("""" ... """ or ''' ... ''') and are used to document the purpose, behavior, and usage of code blocks. Docstrings can be accessed programmatically via the `.__doc__` attribute and are essential for generating documentation.

```
def calculate_area(radius):
    """
    Calculate the area of a circle given its radius.

    Parameters:
        radius (float): The radius of the circle.

    Returns:
        float: The calculated area.
    """
    return 3.14 * radius ** 2
```

Or for classes:

```
class Car:
    """
    Represents a car with manufacturer and model attributes.

    Attributes:
        manufacturer (str): The name of the car manufacturer.
        model (str): The model of the car.
    
```

```
"""
def __init__(self, manufacturer, model):
    self.manufacturer = manufacturer
    self.model = model
```

Class docstrings offer a concise summary of the class, describing its purpose, main attributes, and often including a usage example.

Method docstrings give specific details about the method, explaining its functionality, input parameters, return values, and any exceptions it might raise.

Key Points:

- **Single-line comments** (#) are ideal for short notes and clarifications.
 - **Docstrings** provide structured documentation for functions, classes, and modules.
 - Well-written comments and docstrings improve **code readability, collaboration, and maintenance**.
 - Docstrings are accessible via the `.__doc__` attribute and support automated documentation tools.
 - Use comments to explain **why** something is done, not just **what** is done.
-

2.5 Exception Management

Exception management in Python is a crucial aspect of writing robust and reliable programs. It enables developers to **handle errors gracefully**, preventing unexpected crashes and allowing the program to recover or provide meaningful feedback to users. Exception handling is accomplished using **try-except blocks**, which catch and respond to exceptions that occur during code execution.

An **exception** is an event that interrupts the normal flow of a program's instructions. Unlike **syntax errors**, which prevent code from running at all, exceptions are raised by the Python interpreter when an error occurs while the program is running. When an exception is triggered, Python displays a message starting with `Traceback (most recent call last):`, indicating where the error occurred and what type of exception was raised.

Common exception types include: - **ZeroDivisionError**: Raised when dividing by zero. - **FileNotFoundException**: Raised when trying to access a file that does not exist. - **ValueError**: Raised when a function receives an argument of the correct type but inappropriate value. - **IndexError**: Raised when trying to access an index that is out of range in a sequence (like a list). - **NotImplementedError**: Raised when an abstract method that should be implemented is not.

Example: IndexError (List Index Out of Range)

```
numbers = [1, 2, 3]
print(numbers[5]) # Attempting to access an index that does not exist
```

Output:

```
Traceback (most recent call last):
  File "example.py", line 2, in <module>
    print(numbers[5])
IndexError: list index out of range
```

Key Points: - **Exception management** prevents program crashes and enables error recovery. - Use **try-except blocks** to catch and handle exceptions. - **Exceptions** are runtime errors that disrupt normal program flow. - **Syntax errors** stop code before execution; **exceptions** occur during execution. - Python provides detailed **tracebacks** to help locate and diagnose errors. - Handling exceptions improves **program reliability** and **user experience**. - Always anticipate and manage possible exceptions in your code for better maintainability.

Note: Exceptions are handled in a stack where if an exception is not caught in the current function, it propagates up to the caller function, and so on, until it is either caught or reaches the top level of the program, which will terminate the program if unhandled. Exceptions and errors are normal part of programming and should be expected and handled properly.

2.5.1 Exception Management in Python

Python uses **exception handling** to manage errors that occur during program execution, allowing your code to respond gracefully rather than crashing. The primary mechanism for this is the **try-except** block, which lets you specify code that might raise an exception and define how to handle different error types.

Basic Structure

```
try:
    # Code that may raise an exception
except ExceptionType:
    # Code to handle the specific exception
```

You can also catch **all exceptions** by omitting the exception type, or handle **multiple exception types** using multiple **except** clauses.

Example 1: Handling a Specific Exception

```
try:  
    result = 10 / 0  
except ZeroDivisionError:  
    print("Error: Division by zero is not allowed.")
```

Example 2: Catching All Exceptions (Generic Handler)

```
try:  
    print(undefined_variable)  
except:  
    print("An unexpected error occurred.")
```

Example 3: Multiple Exception Clauses

```
try:  
    value = int("abc")  
    print(10 / value)  
except ValueError:  
    print("Error: Could not convert string to integer.")  
except ZeroDivisionError:  
    print("Error: Division by zero.")  
except Exception as e:  
    print(f"Other error: {e}")
```

Printing Error Messages

To display the actual error message, you can use the `as` keyword to bind the exception to a variable:

```
try:  
    print(undefined_variable)  
except NameError as error:  
    print(f"NameError occurred: {error}")
```

Key Points

- **try-except** blocks allow you to handle errors and prevent program crashes.
- Catch **specific exceptions** for targeted error handling.
- Use a **generic except** clause to catch any exception (not recommended for production code).
- Multiple **except clauses** let you handle different error types separately.
- Use **exception variables** (e.g., `except Exception as e`) to print or log detailed error messages.
- Proper exception handling improves **program reliability** and **user experience**.

2.5.2 Else & Finally

Python's `try`, `except`, `else`, and `finally` blocks work together to provide flexible error handling and control flow.

- **try block:** Contains code that may raise an exception.
- **except block:** Handles exceptions if they occur in the `try` block.
- **else block:** Runs only if no exception was raised in the `try` block.
- **finally block:** Runs no matter what—whether an exception was raised or not.

Example:

```
try:  
    value = int("42")  
    print("Conversion successful.")  
except ValueError:  
    print("Conversion failed.")  
else:  
    print("No errors occurred.")  
finally:  
    print("This always executes.")
```

Output:

```
Conversion successful.  
No errors occurred.  
This always executes.
```

If an exception occurs, the `except` block runs and the `else` block is skipped, but the `finally` block still executes:

```
try:  
    value = int("abc")  # Raises ValueError  
    print("Conversion successful.")  
except ValueError:  
    print("Conversion failed.")  
else:  
    print("No errors occurred.")  
finally:  
    print("This always executes.")
```

Output:

Conversion failed.
This always executes.

Summary: - Use `else` for code that should run only if no errors occurred. - Use `finally` for cleanup actions that must run regardless of errors (e.g., closing files or releasing resources). - The combination of these blocks makes your code robust and predictable.

2.5.3 Custom Exceptions

You can create your own custom exceptions in Python by defining a new class that inherits from the built-in `Exception` class. Custom exceptions are useful when you want to signal specific error conditions in your code that are not covered by standard exceptions.

Example: Defining and Using a Custom Exception

```
class BatteryLowError(Exception):
    """Raised when the battery level is too low for operation."""
    def __init__(self, battery_level, message="Battery level is critically
        low!"):
        self.battery_level = battery_level
        self.message = message
        super().__init__(f"{message} (Level: {battery_level}%)")
```

Then you can raise this exception in your code when a specific condition is met:

```
def operate_electric_car(battery_level):
    if battery_level < 20:
        raise BatteryLowError(battery_level)
    print("Car is operating normally.)
```

Then you can use the custom exception in a `try-except` block:

```
try:
    operate_electric_car(15)
except BatteryLowError as error:
    print(f"Custom Exception Caught: {error}")
```

Explanation: - A new exception class `BatteryLowError` is defined, inheriting from `Exception`. - The custom exception can include extra information (like `battery_level`) and a custom message. - In the function `operate_electric_car`, the exception is raised if the battery level is below a

threshold. - The exception is caught using a try-except block, and a descriptive error message is printed.

Key Points: - Custom exceptions help make your error handling more descriptive and specific. - Always inherit from Exception or one of its subclasses. - You can add custom attributes and messages to your exception class. - Use custom exceptions to signal and handle domain-specific errors in your applications.

2.6 Object Oriented Programming Smart Home Example (in Python)

The objective of this exercise is to **practically apply object-oriented programming (OOP)** principles to model a simplified **IoT system** using Python classes. You will learn how to design and implement software components that represent real-world entities in a **Smart Home** environment.

Scenario Overview

Imagine a Smart Home equipped with various **IoT devices**: - **Temperature sensors** - **Humidity sensors** - **Smart lights**

The task is to: - **Design data structures** and identify appropriate **classes** to represent these devices. - **Implement Python classes** to model device attributes and behaviors. - Create a **central system** (e.g., a `SmartHomeController` class) to **collect** and **manage data** from all devices.

Key Concepts

- Use **class** definitions to encapsulate device properties (e.g., `device_id`, `location`, `status`) and behaviors (e.g., `read_temperature()`, `switch_on()`).
- Model each device type (e.g., `TemperatureSensor`, `HumiditySensor`, `SmartLight`) as a separate **class**.
- The **central system** should aggregate device data and provide methods for monitoring and control.

Exercise Scope

- The scenario is **simplified**:
 - No network communication between devices.
 - All components run **within the same process**.
- Focus on **splitting components** into logical classes and enabling their **interaction** via method calls.

- In future lectures, you will learn how to extend this architecture to support **networked communication** and distributed systems.

Goal: Gain hands-on experience with OOP by modelling a Smart Home IoT system, preparing you for more advanced topics in distributed IoT software architecture.

2.6.1 Which are the Entities in the Project ?

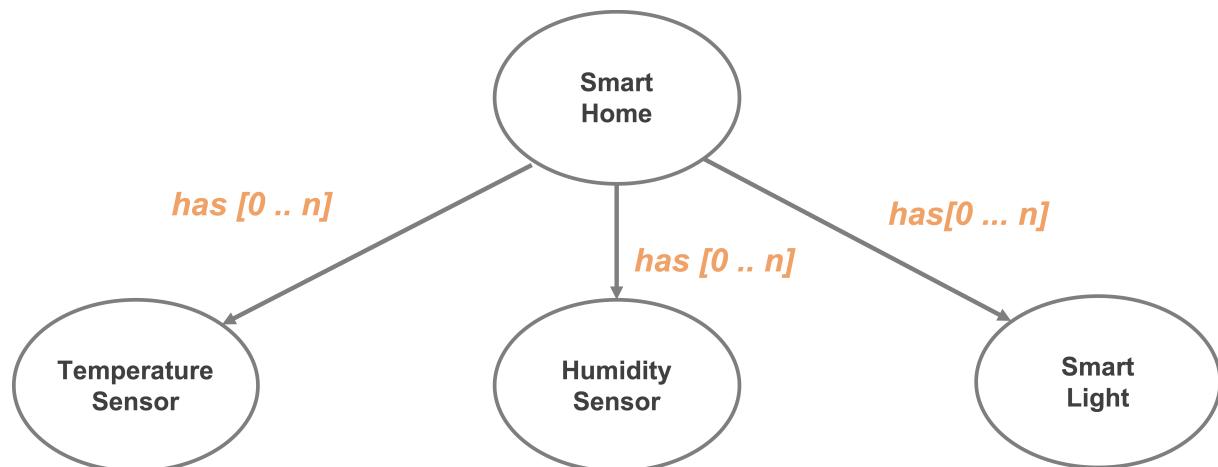


Figure 2.2: Smart Home IoT System Entities that then will be mapped to classes.

When designing a **Smart Home IoT system** using object-oriented programming, it is essential to identify the **key entities**, their **attributes** (data), and **behaviors** (methods) that reflect real-world requirements.

In our scenario, the **Smart Home** acts as the central hub, managing all connected devices. It should encapsulate:

- **Attributes:**
 - **Home ID:** Unique identifier for the smart home.
 - **Location Data:** Such as **latitude** and **longitude**.
 - **Device List:** A collection of all devices (e.g., temperature sensors, humidity sensors, smart lights) installed in the home.
- **Behaviors (Methods):**
 - **Add Device:** Attach a new sensor or light to the home.
 - **Remove Device:** Detach an existing device from the home.
 - **List Devices:** Retrieve information about all connected devices.

By using **classes** to represent both the Smart Home and its devices, you achieve:

- **Encapsulation:** Group related data and behaviors together.
- **Modularity:** Each entity is self-contained and reusable.
- **Extensibility:** Easily add new device types or features.
- **Clear Relationships:** The Smart Home maintains a list of devices and provides methods to manage them.

This modeling approach mirrors real-world systems, making your code **organized**, **maintainable**, and **scalable** for future enhancements.

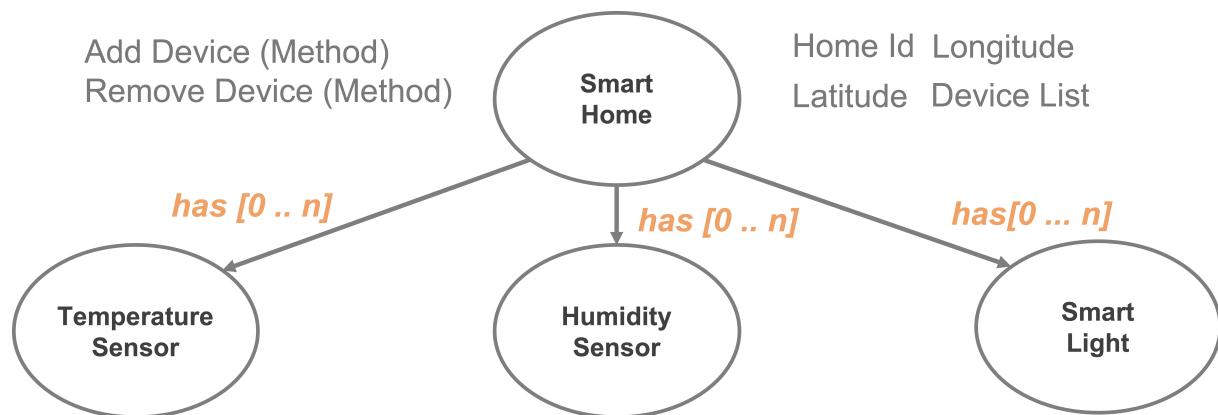


Figure 2.3: Specifications of the Smart Home Entities with their attributes and behaviors.

2.6.2 Sensors & Actuators Characteristics

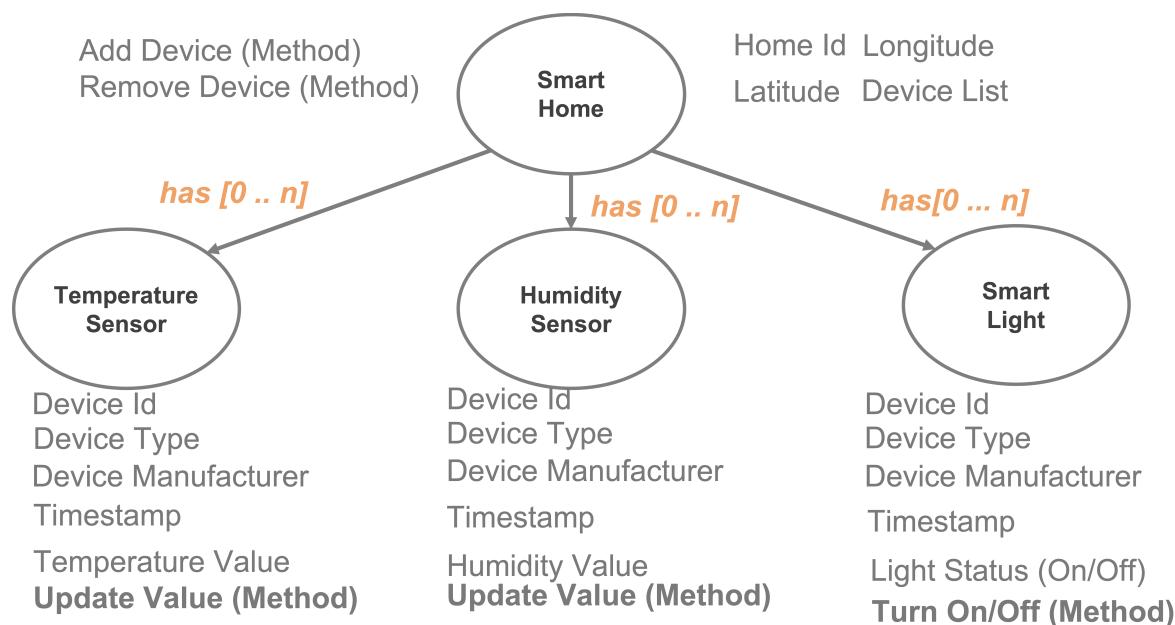


Figure 2.4: Specifications of the Sensors & Actuators with their attributes and behaviors.

When modeling **Smart Home IoT devices** using object-oriented programming, it is crucial to identify the **core attributes** and **behaviors** for each entity. This ensures your classes accurately represent real-world devices and support extensibility. In our scenario, we have three primary device types: **Temperature Sensor**, **Humidity Sensor**, and **Smart Light** (an actuator). Each device type should be encapsulated in its own class with specific attributes and methods.

A **Temperature Sensor** class should encapsulate:

- **Attributes:**
 - **ID:** Unique identifier for the sensor.
 - **Type:** Specifies the kind of sensor (e.g., temperature).
 - **Manufacturer:** The company that produced the sensor.
 - **Last Measurement Timestamp:** The time when the last reading was taken.
 - **Last Measurement Value:** The most recent temperature value (e.g., 25 °C).
- **Behavior (Method):**
 - **Update Value:** Reads a new measurement from the sensor and updates both the value and timestamp.

A **Humidity Sensor** class should include:

- **Attributes:**
 - **ID:** Unique identifier for the sensor.
 - **Type:** Specifies the kind of sensor (e.g., humidity).
 - **Manufacturer:** The company that produced the sensor.
 - **Last Measurement Timestamp:** The time when the last humidity reading was taken.
 - **Last Measurement Value:** The most recent humidity value (e.g., 90%).
- **Behavior (Method):**
 - **Update Value:** Reads a new humidity measurement and updates the value and timestamp.

A **Smart Light** class, representing an actuator, should model:

- **Attributes:**
 - **ID:** Unique identifier for the light.
 - **Type:** Specifies the device type (e.g., smart light).
 - **Manufacturer:** The company that produced the light.

- **Last Value Update Timestamp:** The time when the light's status was last changed.
 - **Status:** Current state of the light (e.g., ON or OFF).
- **Behavior (Method):**
- **Change Status:** Triggers a change in the light's status (e.g., switches between ON and OFF), updating the timestamp accordingly.

By defining these **attributes** and **methods** in your classes, you achieve **encapsulation** of device data and behavior, making your code **modular**, **maintainable**, and ready for future expansion (such as adding new device types or features).

2.6.3 Open “Issues” and Model Improvements

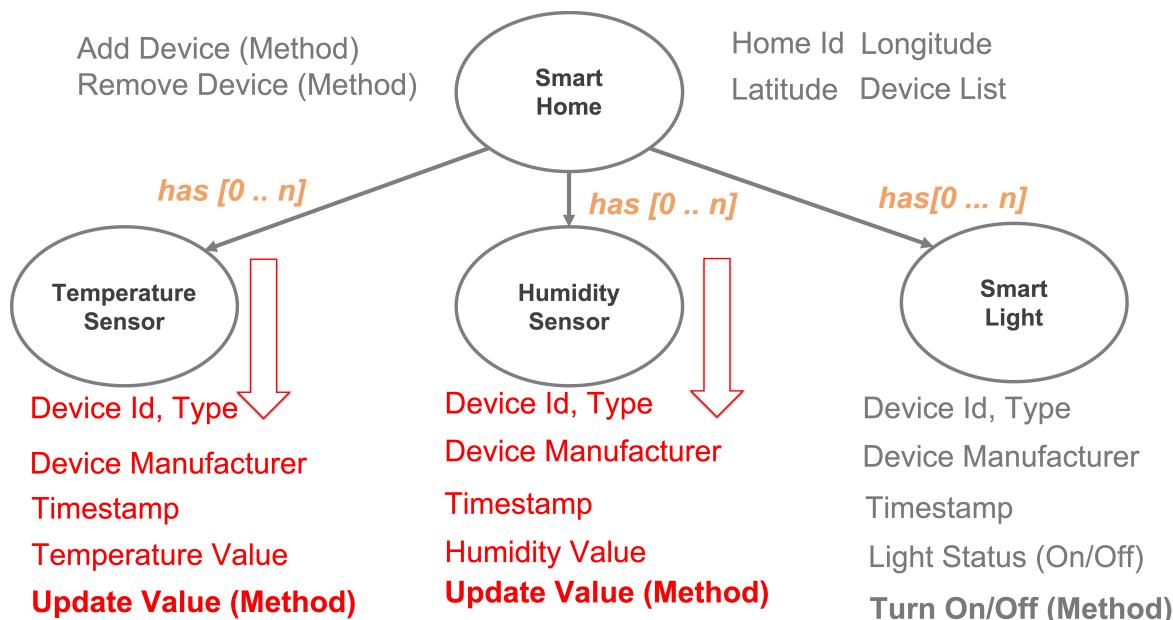


Figure 2.5: Sensors design issues due to replicated fields and methods.

When designing the **Temperature Sensor** and **Humidity Sensor** classes, you'll notice that they share **many common attributes**—such as `id`, `type`, `manufacturer`, `last_measurement_timestamp`, and `last_measurement_value`. They also both implement a **method** for updating their current sensor value (e.g., `update_value()`).

This repetition suggests an opportunity for **modeling improvement** using **inheritance** and **abstraction**. By creating a **generic Sensor base class** that encapsulates the shared attributes and behaviors, you can:

- **Reduce code duplication** by defining common fields and methods only once.
- **Increase maintainability** and **scalability** as new sensor types can inherit from the base class.
- **Clarify relationships** between different sensor types, making your code more organized and extensible.

For example, both `TemperatureSensor` and `HumiditySensor` can inherit from a `Sensor` superclass, which defines the shared structure and provides a generic `update_value()` method that can be specialized as needed.

Key Takeaway:

Use **inheritance** to model shared characteristics and behaviors, ensuring your code is modular, reusable, and easy to extend as your IoT system evolves.

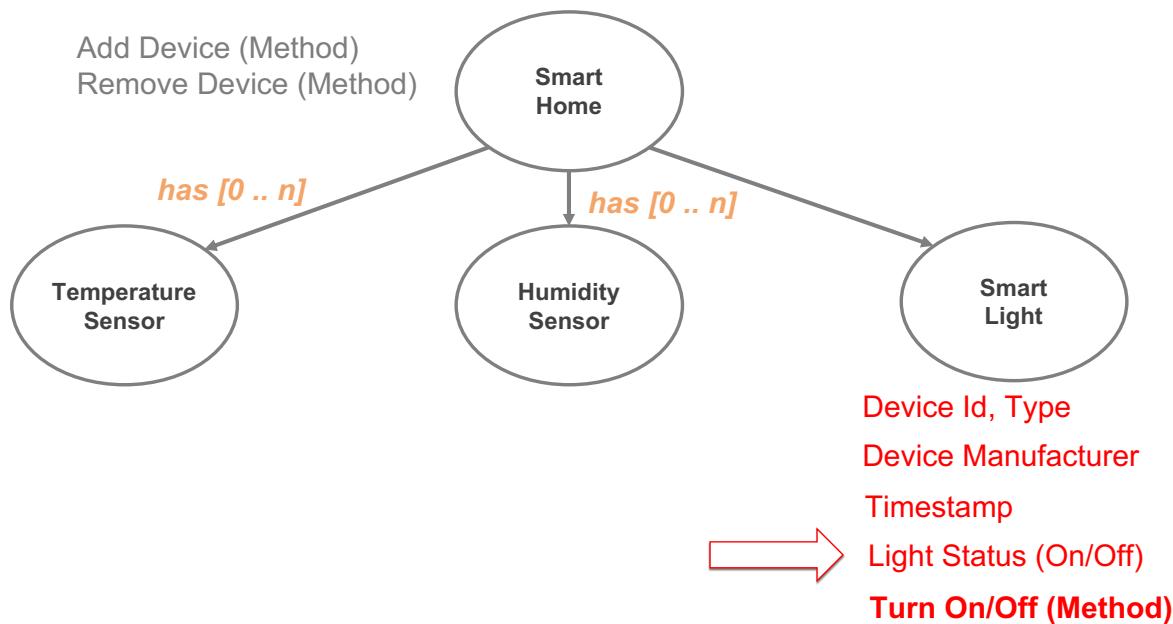


Figure 2.6: Actuators design issues due to replicated fields and methods.

Although the `Smart Light` class is an **actuator** and not a sensor, it shares several **common attributes** with the `Temperature Sensor` and `Humidity Sensor` classes—such as `id`, `type`, and `manufacturer`. The main difference lies in their **behavior**: sensors use an `update_value()` method to record measurements, while actuators like `Smart Light` use a method such as `change_status()` to perform actions (e.g., turning on or off).

This observation highlights the importance of **generalization** in object-oriented modeling. By identifying shared characteristics, you can create a **generic base class** (e.g., `Device`) that encapsulates common attributes and provides a foundation for both **sensors** and **actuators**. Then, you can define

specialized subclasses—such as Sensor and Actuator—that extend the base class and implement device-specific behaviors.

Such a design offers several advantages: - **Reduces code duplication** by centralizing shared fields. - **Improves maintainability** and **scalability** as new device types (e.g., additional actuators) can be added easily. - **Clarifies relationships** between entities, making the system architecture more organized and extensible.

Key Takeaway:

Use **inheritance** and **abstraction** to model both sensors and actuators under a unified structure, allowing your IoT system to evolve and accommodate new device types efficiently over time.

2.6.4 Updated Modeling with Inheritance - Device Class

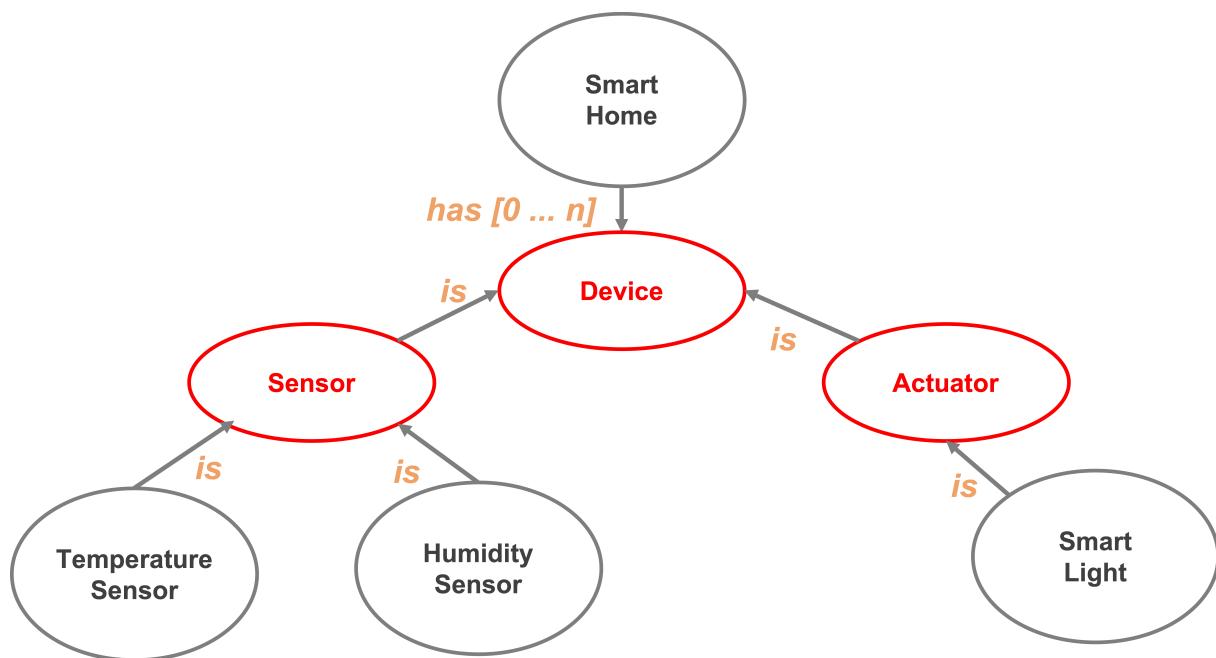


Figure 2.7: Updated Smart Home IoT System Model with Inheritance.

The Device class serves as the **foundation** for modeling all devices in the Smart Home IoT system. It encapsulates the **core attributes**—id, type, and manufacturer—that are **common to every device**, whether it is a **sensor** or an **actuator**. By defining these shared properties in a single **base class**, you achieve **abstraction** and **code reuse**, making your design more **modular** and **extensible**.

This class is intentionally kept **simple**, containing only **data attributes** and **no methods or behaviors**. Its primary purpose is to provide a **consistent structure** for all device types, allowing specialized sub-

classes (such as Sensor and Actuator) to **inherit** these attributes and then implement their own unique behaviors. This approach ensures that any new device added to the system will automatically have the essential identifying information, streamlining future development and maintenance.

Key Points: - **Base class** for all devices (sensors and actuators) - Defines **shared attributes**: id, type, manufacturer - Promotes **abstraction, code reuse**, and **extensibility** - Contains **no behaviors**—specialized functionality is added in subclasses

This modeling strategy lays the groundwork for a scalable and organized IoT architecture, where each device type builds upon a common set of attributes.

An example of the Device class is shown below:

```
class Device:  
    """Base class for all devices in the Smart Home IoT system."""  
  
    def __init__(self, id, type, manufacturer):  
        """Initialize the Device with basic attributes."""  
        self.id = id  
        self.type = type  
        self.manufacturer = manufacturer
```

2.6.5 Updated Modeling with Inheritance - Sensor Class

The **Sensor** class serves as a **base class** for all sensor types in the Smart Home IoT system. Its primary purpose is to **define the common structure and expected behavior** for any sensor you implement. By inheriting from the **Device** class, it automatically includes essential attributes such as **id**, **type**, and **manufacturer**.

In addition to these inherited attributes, the Sensor class introduces: - **last_measurement_timestamp**: Records when the most recent measurement was taken. - **last_measurement_value**: Stores the value of the latest sensor reading. - An **update_value() method**: Intended to refresh the sensor's measurement and timestamp.

The **update_value() method** is deliberately left **empty** (often called an “abstract” or “placeholder” method) in the base class. This design signals that every sensor subclass (e.g., TemperatureSensor, HumiditySensor) must provide its own specific implementation of how measurements are updated. This approach enforces a **consistent interface** and ensures that all sensors in the system share a common set of attributes and behaviors, while allowing for specialized functionality in each sensor type.

Then the different implementation of the Sensor class can define their own update_value() method to handle the specifics of how they obtain and process their measurements. For example a TemperatureSensor implemented with a Raspberry Pi could read from a connected temperature sensor, while a HumiditySensor might interface with a different hardware component. The abstract update_value() method ensures that all sensor types adhere to a common protocol for updating their readings, promoting consistency and reliability across the system.

Modeling Rationale: - Promotes **code reuse** and **modularity** by centralizing shared sensor features. - Supports **extensibility**—new sensor types can be added easily by extending the base class. - Enforces a **standard interface** for updating sensor values, improving maintainability and reliability.

This modeling strategy is a key principle of **object-oriented programming**, enabling you to build scalable and organized systems where each sensor type is both consistent and customizable.

An example of the Sensor class is shown below:

```
class Sensor(Device):
    """Base class for all sensors in the Smart Home IoT system."""

    def __init__(self, id, type, manufacturer):
        """Initialize the Sensor with inherited and specific attributes."""
        super().__init__(id, type, manufacturer)

        # Specific attributes for sensors set to None initially
        # Subclasses will update these values
        self.last_measurement_timestamp = None
        self.last_measurement_value = None

    def update_value(self):
        """Abstract method to update the sensor's measurement."""
        # Subclasses must implement this method, so we raise an error
        raise NotImplementedError("Subclasses must implement this method.")
```

The main characteristics of the above Sensor class are: - Inherits from the Device class, gaining access to common attributes like id, type, and manufacturer. - The constructor (__init__ method) calls the parent class constructor using super(), ensuring proper initialization of inherited attributes. - Introduces sensor-specific attributes: last_measurement_timestamp and last_measurement_value, initialized to None. - Defines an **abstract method** update_value(), which raises a NotImplementedError. This indicates that any subclass must provide its own implementation of this method to handle the specifics of updating sensor measurements. - Promotes a consistent interface for all sensor types while allowing for specialized behavior in subclasses.

2.6.6 Updated Modeling with Inheritance - Actuator Class

The **Actuator** class serves as a **base class** for all actuators in the Smart Home IoT system. Its primary purpose is to **define the common structure and expected behavior** for any actuator device you implement. By inheriting from the **Device** class, it automatically includes essential attributes such as **id**, **type**, and **manufacturer**.

In addition to these inherited attributes, the Actuator class introduces: - **last_status_change_timestamp**: Records when the actuator's status was last changed. - **status**: Stores the current state of the actuator (e.g., ON or OFF). - An **invoke_action() method**: Intended to trigger a specific action on the actuator.

The **invoke_action() method** is deliberately left **empty** (often called an “abstract” or “placeholder” method) in the base class. This design enforces that every actuator subclass (such as SmartLight or SmartLock) must provide its own specific implementation of how actions are invoked. This approach ensures a **consistent interface** for all actuators, while allowing for specialized functionality in each actuator type.

Modeling Rationale: - Promotes **code reuse** and **modularity** by centralizing shared actuator features. - Supports **extensibility**—new actuator types can be added easily by extending the base class. - Enforces a **standard interface** for invoking actions, improving maintainability and reliability.

This modeling strategy is a key principle of **object-oriented programming**, enabling you to build scalable and organized systems where each actuator type is both consistent and customizable.

An example of the Actuator class is shown below:

```
class Actuator(Device):
    """Base class for all actuators in the Smart Home IoT system."""
    def __init__(self, id, type, manufacturer):
        """Initialize the Actuator with inherited and specific attributes."""
        super().__init__(id, type, manufacturer)

        # Specific attributes for actuators set to None initially
        # Subclasses will update these values
        self.last_status_change_timestamp = None
        self.status = None # e.g., "ON" or "OFF"

    def invoke_action(self, action_type, payload):
        """Abstract method to invoke an action on the actuator."""
        # Subclasses must implement this method, so we raise an error
```

```
raise NotImplementedError("Subclasses must implement this method.")
```

The main characteristics of the above Actuator class are:

- Inherits from the Device class, gaining access to common attributes like id, type, and manufacturer.
- The constructor (`__init__` method) calls the parent class constructor using `super()`, ensuring proper initialization of inherited attributes.
- Introduces actuator-specific attributes: `last_status_change_timestamp` and `status`, initialized to None.
- Defines an **abstract method** `invoke_action()`, which raises a `NotImplementedError`. This indicates that any subclass must provide its own implementation of this method to handle the specifics of invoking actions on the actuator.
- The `invoke_action()` method takes parameters `action_type` and `payload`, allowing subclasses to define various actions with associated data where the type identifies the action (e.g., “turn_on”, “set_brightness”) and the payload contains any necessary information (e.g., brightness level).
- Promotes a consistent interface for all actuator types while allowing for specialized behavior in subclasses.

2.6.7 From Sensor Abstraction to TemperatureSensor & HumiditySensor

When modeling **TemperatureSensor** and **HumiditySensor** classes, both are designed to **inherit** from the generic **Sensor** base class. This means they automatically acquire all the **shared attributes** of a Sensor—such as `last_measurement_timestamp` and `last_measurement_value`—as well as the foundational device attributes (id, type, and manufacturer) from the **Device** superclass.

Each sensor type then provides its own **implementation** of the `update_value()` method, which is responsible for updating both the **measurement value** and the **timestamp** according to the specific nature of the sensor (temperature or humidity). This approach ensures that:

- **Common structure and behavior** are centralized in the base classes, promoting **code reuse** and **consistency**.
- **Specialized functionality** is handled in the subclasses, allowing each sensor to define how it obtains and processes its measurements.
- The system remains **extensible**—new sensor types can be added easily by inheriting from the Sensor class and implementing their own update logic.

By following this modeling strategy, you achieve a clear separation between **shared features** and **device-specific behavior**, making your codebase more **organized, maintainable**, and **scalable**.

In our simple example a **TemperatureSensor** can be implemented as follows:

```

import random
import time

class TemperatureSensor(Sensor):
    """Temperature sensor class."""

    def __init__(self, id, initial_temperature_value=25.0):
        """Initialize the TemperatureSensor with inherited and specific
           attributes."""
        # Call the parent constructor to initialize common attributes where
        # type and manufacturer are fixed (to be associated for example to
        # a reference model)
        super().__init__(id, "TemperatureSensor", "GenericManufacturer")

        # Initialize Value to a default or provided temperature
        self.last_measurement_value = initial_temperature_value

        # Initialize the timestamp to the current time in milliseconds
        self.last_measurement_timestamp = time.time() * 1000 # Current time
        # in milliseconds

    def update_value(self):
        """Update the temperature measurement."""

        # Simulate temperature reading
        self.last_measurement_value = random.uniform(15.0, 30.0)

        # Update timestamp to current time in milliseconds
        self.last_measurement_timestamp = time.time() * 1000

```

Similarly, a HumiditySensor can be implemented as follows:

```

import random
import time

class HumiditySensor(Sensor):
    """Humidity sensor class."""

    def __init__(self, id, initial_humidity_value=50.0):
        """Initialize the HumiditySensor with inherited and specific
           attributes."""

```

```

# Call the parent constructor to initialize common attributes where
# type and manufacturer are fixed (to be associated for example to
# a reference model)
super().__init__(id, "HumiditySensor", "GenericManufacturer")

# Initialize Value to a default or provided humidity
self.last_measurement_value = initial_humidity_value

# Initialize the timestamp to the current time in milliseconds
self.last_measurement_timestamp = time.time() * 1000 # Current time
# in milliseconds

def update_value(self):
    """Update the humidity measurement."""

    # Simulate humidity reading
    self.last_measurement_value = random.uniform(30.0, 90.0)

    # Update timestamp to current time in milliseconds
    self.last_measurement_timestamp = time.time() * 1000

```

2.6.8 From Actuator Abstraction to SmartLight

When modeling the **SmartLight** class, it is designed to **inherit** from the generic **Actuator** base class. This means it automatically acquires all the **shared attributes** of an Actuator—such as `last_status_change_timestamp` and `status`—as well as the foundational device attributes (`id`, `type`, and `manufacturer`) from the **Device** superclass.

The SmartLight class then provides its own **implementation** of the `invoke_action()` method, which is responsible for changing the light's status (e.g., turning it ON or OFF) and updating the timestamp accordingly. This approach ensures that:

- **Common structure and behavior** are centralized in the base classes, promoting **code reuse** and **consistency**.
- **Specialized functionality** is handled in the subclass, allowing the SmartLight to define how it manages its status.
- The system remains **extensible**—new actuator types can be added easily by inheriting from the Actuator class and implementing their own action logic.

An example of the SmartLight class is shown below:

```

import time
class SmartLight(Actuator):

```

```
"""Smart light actuator class."""

def __init__(self, id, initial_status="OFF"):
    """Initialize the SmartLight with inherited and specific
       attributes."""
    # Call the parent constructor to initialize common attributes where
    # type and manufacturer are fixed (to be associated for example to
    # a reference model)
    super().__init__(id, "SmartLight", "GenericManufacturer")

    # Initialize status to a default or provided value
    self.status = initial_status

    # Initialize the timestamp to the current time in milliseconds
    self.last_status_change_timestamp = time.time() * 1000 # Current
    # time in milliseconds

def invoke_action(self, action_type, payload=None):
    """Invoke an action on the smart light."""
    if action_type == "turn_on":
        self.status = "ON"
    elif action_type == "turn_off":
        self.status = "OFF"
    else:
        raise ValueError(f"Unknown action type: {action_type}")

    # Update timestamp to current time in milliseconds
    self.last_status_change_timestamp = time.time() * 1000
```

2.6.9 Final Overall Design and Modeling with Inheritance

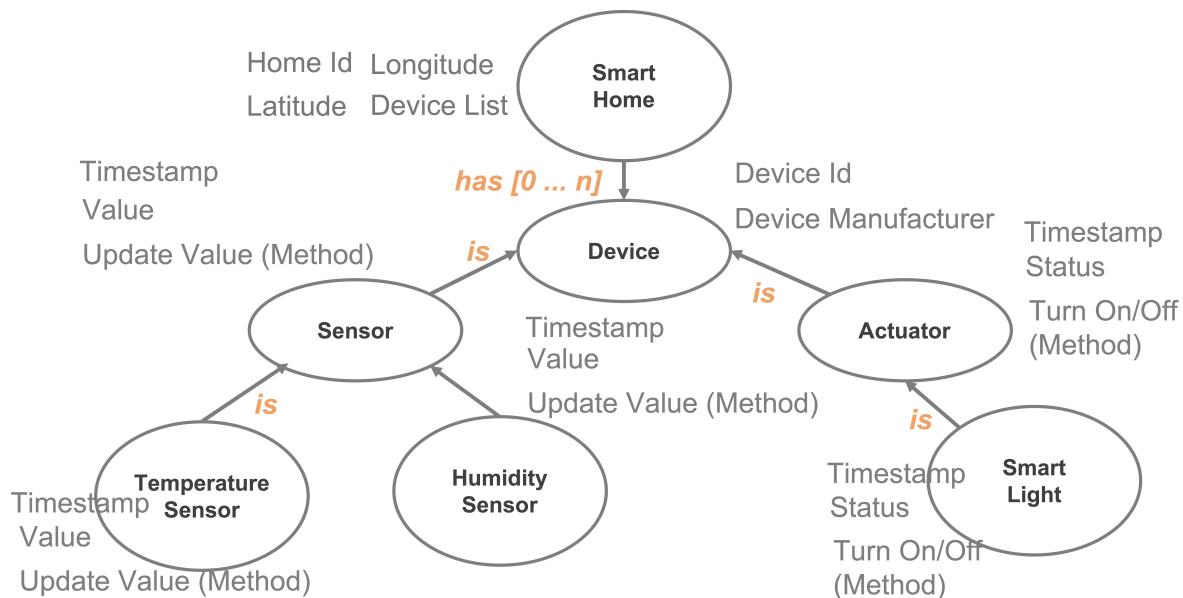


Figure 2.7: Overall Smart Home IoT System Model with Inheritance.

The **final design** of the Smart Home IoT system incorporates a well-structured hierarchy of classes that leverage **inheritance** to promote code reuse, modularity, and extensibility. At the top of the hierarchy is the **Device** class, which encapsulates the fundamental attributes shared by all devices in the system, such as `id`, `type`, and `manufacturer`. From the **Device** class, two specialized subclasses are derived: **Sensor** and **Actuator**. The **Sensor** class adds attributes specific to sensors, including `last_measurement_timestamp` and `last_measurement_value`, along with an abstract method `update_value()` that must be implemented by all sensor types. Similarly, the **Actuator** class introduces attributes like `last_status_change_timestamp` and `status`, along with an abstract method `invoke_action()` for performing actions on actuators. Building upon the **Sensor** class, two concrete sensor types are defined: **TemperatureSensor** and **HumiditySensor**. Each of these classes inherits the common attributes and behaviors from the **Sensor** class while providing their own specific implementations of the `update_value()` method to handle temperature and humidity measurements, respectively. And then **SmartLight** class inherits from the **Actuator** class and implements the `invoke_action()` method to manage the light's status (e.g., turning it ON or OFF).

This hierarchical structure ensures that all devices in the Smart Home system share a consistent set of core attributes while allowing for specialized functionality in each device type. The use of inheritance not only reduces code duplication but also makes it easier to maintain and extend the system as new device types are added in the future.

2.7 Smart Home and Data Manager

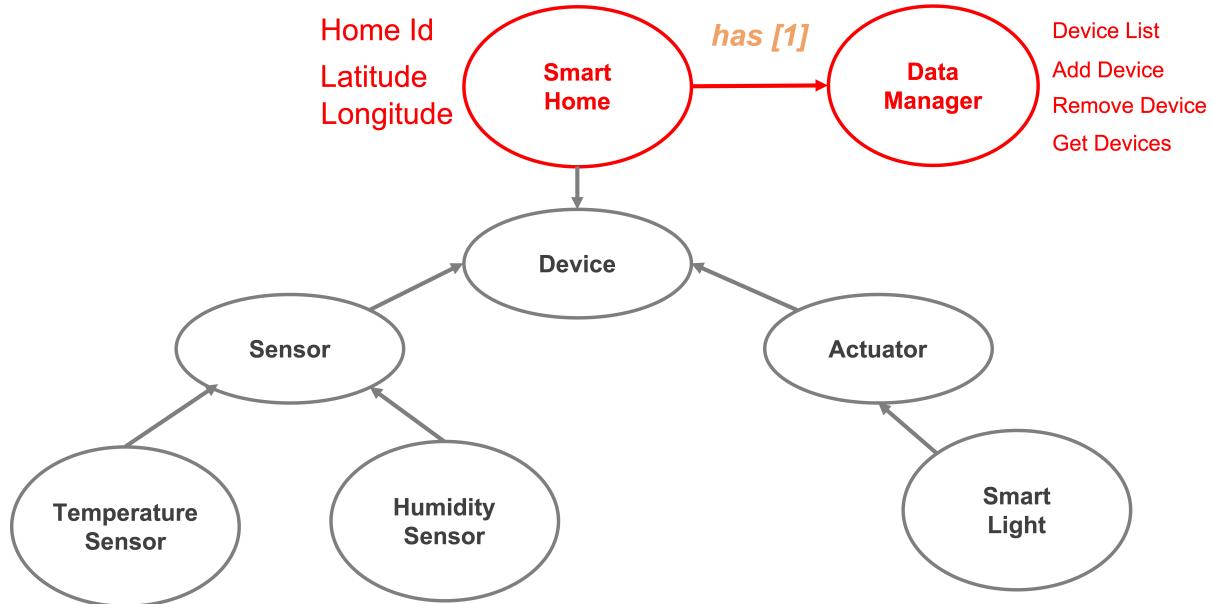


Figure 2.8: Adding a Data Manager to the Smart Home IoT System to delegate data management.

In a well-designed object-oriented system, the **Smart Home** class should focus on representing the home itself—its identity, location, and high-level behaviors—rather than directly managing how device data is stored or retrieved. The responsibility for **data management** (such as storing, updating, and retrieving device information) is best **delegated** to a dedicated class, often called a **Data Manager**.

By introducing a **Data Manager** class, you achieve several modeling benefits:

- **Separation of Concerns**: The Smart Home class is responsible for orchestrating home-level operations, while the Data Manager handles the details of device storage and access.
- **Encapsulation**: The Data Manager hides the internal details of how devices are managed (e.g., using a dictionary, CSV file, database, etc.), exposing only the necessary methods for the Smart Home to interact with devices.
- **Modularity & Extensibility**: Changes to data storage (such as switching from in-memory to persistent storage) can be made within the Data Manager without affecting the Smart Home logic.
- **Simplified Smart Home Code**: The Smart Home interacts with devices through the Data Manager's interface, making its code cleaner and easier to maintain.

Modeling Rationale:

- The **Smart Home** class should be associated with *what* the home is and *how* it is managed at a high level.
- The **Data Manager** class should be responsible for *how* device data is stored, retrieved, and maintained.
- This design pattern follows the principle of **delegation**, where specialized classes handle specific responsibilities, resulting in a more robust and maintainable architecture.

In summary, delegating device management to a **Data Manager** class allows the Smart Home to remain focused on its core responsibilities, while data handling is abstracted and encapsulated, supporting future scalability and flexibility.

An example of the DataManager class is shown below:

```
class DataManager:  
    """Class responsible for managing device data."""  
  
    def __init__(self):  
        """Initialize the DataManager with an empty device dictionary."""  
  
        # List to store devices by their ID  
        self.devices = []  
  
        # Dictionary to store sensor data by sensor ID  
        self.sensor_data = {}  
  
    def add_device(self, device):  
        """Add a new device to the manager."""  
        self.devices.append(device)  
        if isinstance(device, Sensor):  
            self.sensor_data[device.id] = {  
                "last_measurement_timestamp":  
                    device.last_measurement_timestamp,  
                "last_measurement_value": device.last_measurement_value  
            }  
        elif isinstance(device, Actuator):  
            self.sensor_data[device.id] = {  
                "last_status_change_timestamp":  
                    device.last_status_change_timestamp,  
                "status": device.status  
            }  
  
    def remove_device(self, device_id):  
        """Remove a device from the manager by its ID."""  
        self.devices = [d for d in self.devices if d.id != device_id]  
        if device_id in self.sensor_data:  
            del self.sensor_data[device_id]
```

```

    del self.sensor_data[device_id]

def get_device(self, device_id):
    """Retrieve a device by its ID."""
    for device in self.devices:
        if device.id == device_id:
            return device
    return None

def list_devices(self):
    """List all devices managed by the DataManager."""
    return self.devices

def update_sensor_data(self, sensor_id, timestamp, value):
    """Update the data for a specific sensor."""
    if sensor_id in self.sensor_data:
        self.sensor_data[sensor_id]["last_measurement_timestamp"] =
            ↵ timestamp
        self.sensor_data[sensor_id]["last_measurement_value"] = value

def update_actuator_data(self, actuator_id, timestamp, status):
    """Update the data for a specific actuator."""
    if actuator_id in self.sensor_data:
        self.sensor_data[actuator_id]["last_status_change_timestamp"] =
            ↵ timestamp
        self.sensor_data[actuator_id]["status"] = status

```

The main characteristics of the above DataManager class are:

- Responsible for managing device data, including adding, removing, and retrieving devices.
- Maintains a list of devices and a dictionary to store sensor and actuator data by their IDs.
- Provides methods to add and remove devices, retrieve a device by its ID, and list all managed devices.
- Includes methods to update sensor and actuator data, ensuring that the latest measurements and statuses are stored.
- Promotes separation of concerns by delegating data management responsibilities away from the Smart Home class, allowing for cleaner and more maintainable code.
- Supports extensibility, as changes to data storage or management can be made within the DataManager without affecting other parts of the system.

2.8 Implementing the Smart Home Class and its Behaviors

The **Smart Home** class serves as the central entity in the Smart Home IoT system, representing the home itself and managing its associated devices. It encapsulates key attributes such as `home_id`,

latitude, and longitude, which uniquely identify the home and its location. Additionally, the Smart Home class maintains a reference to a **Data Manager** instance, which is responsible for handling the storage and retrieval of device data. This delegation allows the Smart Home to focus on high-level operations while the Data Manager manages the specifics of device data.

The Smart Home class provides several important behaviors (methods) to manage its devices:

- **add_device(device)**: Adds a new device (sensor or actuator) to the home by delegating the operation to the Data Manager.
- **remove_device(device_id)**: Removes an existing device from the home using its unique ID, again delegating to the Data Manager.
- **get_device(device_id)**: Retrieves a specific device by its ID, allowing the Smart Home to access device information as needed.
- **list_devices()**: Lists all devices currently managed by the Smart Home, providing an overview of connected devices.

An example of the SmartHome class is shown below:

```
class SmartHome:  
    """Class representing a Smart Home."""  
  
    def __init__(self, home_id, latitude, longitude, data_manager):  
        """Initialize the SmartHome with its ID, location, and a DataManager  
        instance."""  
  
        # Basic attributes of the Smart Home  
        self.home_id = home_id  
        self.latitude = latitude  
        self.longitude = longitude  
  
        # Instance of DataManager to manage devices  
        self.data_manager = data_manager  
  
    def add_device(self, device):  
        """Add a new device to the smart home."""  
        self.data_manager.add_device(device)  
  
    def remove_device(self, device_id):  
        """Remove a device from the smart home by its ID."""  
        self.data_manager.remove_device(device_id)  
  
    def get_device(self, device_id):  
        """Retrieve a device by its ID."""
```

```
    return self.data_manager.get_device(device_id)

def list_devices(self):
    """List all devices in the smart home."""
    return self.data_manager.list_devices()
```

The fact that the `SmartHome` class delegates device management to the `DataManager` class is a key design choice that promotes **separation of concerns**. This means that the Smart Home focuses on representing the home and its high-level operations, while the Data Manager handles the specifics of how devices are stored and managed. In this way **we can change the internal implementation of the `DataManager` (for example using a database instead of in-memory storage) without affecting the `SmartHome` class or its interactions with devices.**

An example of how to use the `SmartHome` class along with the `DataManager` and device classes for a simple monitoring scenario is shown below:

```
import time

# Create a DataManager instance
data_manager = DataManager()

# Create a SmartHome instance
smart_home = SmartHome("home_1", 37.7749, -122.4194, data_manager)

# Create some devices
temperature_sensor = TemperatureSensor("sensor_1",
                                       initial_temperature_value=22.5)
humidity_sensor = HumiditySensor("sensor_2", initial_humidity_value=45.0)
light_bulb = LightBulb("bulb_1", initial_status="OFF")

# Add devices to the smart home
smart_home.add_device(temperature_sensor)
smart_home.add_device(humidity_sensor)
smart_home.add_device(light_bulb)

# Simulate some device readings
temperature_sensor.update_value()
humidity_sensor.update_value()
light_bulb.invoke_action("turn_on", None)

# Wait for a moment
time.sleep(1)
```

```
# Retrieve and print device information
for device in smart_home.list_devices():
    print(f"Device ID: {device.id}, Type: {device.type}, Manufacturer:
          {device.manufacturer}")
    if isinstance(device, Sensor):
        print(f"  Last Measurement: {device.last_measurement_value} at
              {device.last_measurement_timestamp}")
    elif isinstance(device, Actuator):
        print(f"  Status: {device.status} at
              {device.last_status_change_timestamp}")
```

In this example:

- A DataManager instance is created to handle device data.
- A SmartHome instance is initialized with a unique ID and location, along with the DataManager.
- Several devices (a temperature sensor, a humidity sensor, and a smart light bulb) are created and added to the Smart Home.
- The sensors update their readings, and the light bulb is turned on.
- After a brief pause, the code retrieves and prints information about all devices in the Smart Home, demonstrating how the Smart Home interacts with its devices through the Data Manager.
- Navigating through the devices and checking their types allows for appropriate handling of sensor and actuator-specific attributes.

Note: The method `isinstance()` is used to check the type of each device, allowing the code to access sensor-specific or actuator-specific attributes accordingly. This ensures that the correct information is printed based on the device type.