


Inheritance, Compositions, Aggregations, and Mixins

A comparative approach.

Week 7

Inheritance Review



What is Inheritance (Is-A Relation)?

- A mechanism that allows us to inherit all the properties from another class.
- The class from which the properties and functionalities are utilized is called the parent class (also called as Base Class).
- The class which uses the properties from another class is called as Child Class (also known as Derived class). Inheritance is also called an Is-A Relation.

Composition

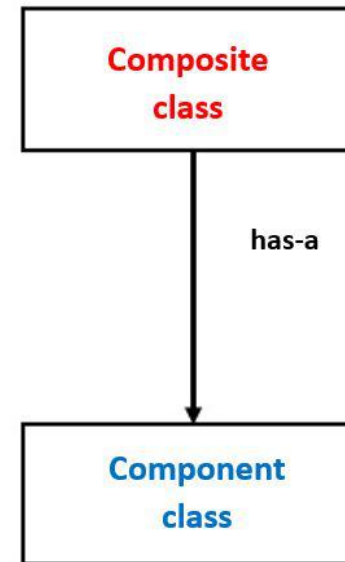
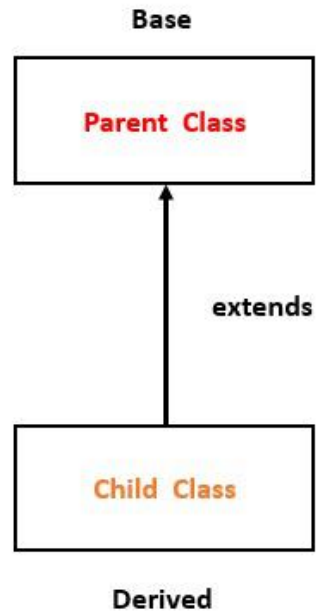


What is Composition (Has-A Relation)?

- It is one of the fundamental concepts of Object-Oriented Programming.
- In this concept, we will describe a class that references to one or more objects of other classes as an Instance variable.
- Here, by using the class name or by creating the object we can access the members of one class inside another class. It enables creating complex types by combining objects of different classes. It means that a class Composite can contain an object of another class Component.
- This type of relationship is known as Has-A Relation.

Is-A Relation vs Has-A Relation

- Inheritance allows changes in a class's functionality or behavior.
- Composition does not allow these changes but can only provide what is already available.



Has-A Relation (Composition)

```
class A :  
  
    # variables of class A  
    # methods of class A  
    ...  
    ...  
  
class B :  
    # by using "obj" we can access member's of class A.  
    obj = A()  
  
    # variables of class B  
    # methods of class B  
  
    ...  
    ...
```

Composition Code Sample

```
Component class object created...  
Composite class object also created...  
Composite class m2() method executed...  
Component class m1() method executed...
```

```
class Component:  
  
    # composite class constructor  
    def __init__(self):  
        print('Component class object created...')  
  
    # composite class instance method  
    def m1(self):  
        print('Component class m1() method executed...')  
  
class Composite:  
  
    # composite class constructor  
    def __init__(self):  
  
        # creating object of component class  
        self.obj1 = Component()  
  
        print('Composite class object also created...')  
  
    # composite class instance method  
    def m2(self):  
  
        print('Composite class m2() method executed...')  
  
        # calling m1() method of component class  
        self.obj1.m1()  
  
# creating object of composite class  
obj2 = Composite()  
  
# calling m2() method of composite class  
obj2.m2()
```


PARTS OF THE FACE



Understand it easily

- Composition can easily indicate a relationship like having a Face with eyes, mouth, ears, teeth, etc.
- From the name itself, "COMPOSITION," a face is composed of
- Face class was used as most of its parts are originally dependent and direct to human's face. You don't really swap them.

```

class Hair:
    def __init__(self, hair_type=""):
        self.hair_type = hair_type

class Eyebrows:
    def __init__(self, eyebrow_shape=""):
        self.eyebrow_shape = eyebrow_shape

class Forehead:
    def __init__(self, forehead_texture=""):
        self.forehead_texture = forehead_texture

class Eye:
    def __init__(self, eye_color=""):
        self.eye_color = eye_color

class Cheek:
    def __init__(self, cheek_color=""):
        self.cheek_color = cheek_color

class Ear:
    def __init__(self, ear_shape=""):
        self.ear_shape = ear_shape

class Nose:
    def __init__(self, nose_size=""):
        self.nose_size = nose_size

class Mouth:
    def __init__(self, teeth=None, tongue=None):
        self.teeth = teeth or ["incisor", "canine", "premolar", "molar"]
        self.tongue = tongue or ["taste buds", "muscles"]

    def speak(self, message):
        print(f"The mouth is speaking: {message}")

    def chew(self, food):
        print(f"The mouth is chewing: {food}")

class Chin:
    def __init__(self, chin_shape=""):
        self.chin_shape = chin_shape

```

```

class Face:
    def __init__(self):
        self.hair = Hair()
        self.eyebrows = Eyebrows()
        self.forehead = Forehead()
        self.eye = Eye()
        self.cheek = Cheek()
        self.ear = Ear()
        self.nose = Nose()
        self.mouth = Mouth()
        self.chin = Chin()

```

```

my_face = Face() print("My face has:")
print(f"Hair: {my_face.hair.hair_type}")
print(f"Eyebrows: {my_face.eyebrows.eyebrow_shape}")
print(f"Forehead: {my_face.forehead.forehead_texture}")
print(f"Eye: {my_face.eye.eye_color}")
print(f"Cheek: {my_face.cheek.cheek_color}")
print(f"Ear: {my_face.ear.ear_shape}")
print(f"Nose: {my_face.nose.nose_size}")
print(f"Chin: {my_face.chin.chin_shape}")
print(f"Mouth: Teeth - {my_face.mouth.teeth}, Tongue - {my_face.mouth.tongue}") my_face.mouth.speak("Hello, world!") my_face.mouth.chew("pizza")

```

Advantages over Inheritance of Composition

- Flexibility
- Encapsulation
- Reduced Coupling
- Avoidance of Inheritance Hierarchy Problem
- Better Code Reuse



Coupling Problem

Suppose we have a base class called Animal, which has some basic attributes and methods like name, age, and speak().

We also have two subclasses called Dog and Cat, which inherit from Animal and add some additional attributes and methods specific to those animals.

Here's what the code might look like:

```
class Animal:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def speak(self):
        pass

class Dog(Animal):
    def __init__(self, name, age, breed):
        super().__init__(name, age)
        self.breed = breed

    def speak(self):
        return "Woof!"

class Cat(Animal):
    def __init__(self, name, age, color):
        super().__init__(name, age)
        self.color = color

    def speak(self):
        return "Meow!"
```

Common Types of Coupling

- Tight Coupling
- Inappropriate Dependency
- Circular Dependencies

Tight Coupling

When one class directly depends on the implementation details of another class, it's tightly coupled.

This makes the classes difficult to change independently.

In the example, class A is tightly coupled with class B because it directly instantiates B and calls its method.

```
class A:  
    def method(self):  
        b = B()  
        b.do_something()  
  
class B:  
    def do_something(self):  
        # Code
```

Inappropriate Dependency

Classes should depend on abstractions, not on concrete implementations.

If a class depends on specific methods or attributes of another class, it creates a strong dependency.

Here, **PaymentProcessor** depends directly on the **PayPalGateway** class, which makes it **hard to switch to another payment gateway without modifying PaymentProcessor**.

```
class PaymentProcessor:
    def process_payment(self, amount):
        gateway = PayPalGateway() # Inappropriate dependency
        gateway.process(amount)

class PayPalGateway:
    def process(self, amount):
        # Process payment
```

Circular Dependencies

Circular dependencies occur when two or more modules depend on each other. This makes it challenging to understand and maintain the codebase.

In this example, **module1** depends on **module2** and vice versa, creating a circular dependency.

To mitigate coupling problems, it's essential to follow principles like the Dependency Inversion Principle (DIP) and the Dependency Injection pattern.

These help in reducing dependencies between classes and making the codebase more flexible and maintainable.

```
# module1.py
from module2 import B
```

```
class A:
    def method(self):
        b = B()
        b.do_something()
```

```
# module2.py
from module1 import A
```

```
class B:
    def do_something(self):
        a = A()
        a.method()
```


Coupling Problem in Inheritance

Now suppose we want to add a new feature to both Dog and Cat classes that allows them to swim. We could create a new mixin called Swimmable, like so:

We can then add Swimmable to both Dog and Cat, like so:

Now both DogSwim and CatSwim have the ability to swim, but notice that adding this new feature has created a coupling between the Dog and Cat classes and the Swimmable mixin. If we ever need to modify the Swimmable mixin, it could potentially affect the behavior of both Dog and Cat classes, which may not be what we want.

In this case, the coupling is relatively low, since the Swimmable mixin is a fairly simple and self-contained unit of functionality. However, in more complex cases where multiple mixins are used together, the coupling can become more significant and lead to a more tightly-coupled and difficult-to-maintain codebase.

```
class Swimmable:
    def swim(self):
        return f"{self.name} is swimming!"
```

```
class DogSwim(Dog, Swimmable):
    pass

class CatSwim(Cat, Swimmable):
    pass
```

Flexibility with Composition

```
class Engine:
    def start(self):
        raise NotImplementedError

class GasEngine(Engine):
    def start(self):
        print("Starting gas engine")

class ElectricEngine(Engine):
    def start(self):
        print("Starting electric engine")

class Car:
    def __init__(self, engine):
        self.engine = engine

    def start(self):
        self.engine.start()

# Create a car with a gas engine
gas_engine = GasEngine()
car = Car(gas_engine)
car.start() # Output: Starting gas engine

# Now let's swap out the engine with an electric engine
electric_engine = ElectricEngine()
car.engine = electric_engine
car.start() # Output: Starting electric engine
```

- Composition allows you to easily switch out components with different implementations.
- For example, if you have a Car class that has an Engine component, you could easily swap out a gas engine for an electric engine without changing the Car class.
- In contrast, inheritance can make it harder to change the behavior of a class, especially if the class has many subclasses that depend on its implementation.

Encapsulation with Composition

```
class Engine:
    def start(self):
        raise NotImplementedError

class GasEngine(Engine):
    def start(self):
        print("Starting gas engine")

class Car:
    def __init__(self, engine):
        self.engine = engine

    def start(self):
        self.engine.start()

car = Car(GasEngine())
car.start() # Output: Starting gas engine

# Let's change the implementation of GasEngine without affecting Car
class ImprovedGasEngine(Engine):
    def start(self):
        print("Starting improved gas engine")

car.engine = ImprovedGasEngine()
car.start() # Output: Starting improved gas engine
```

- Composition promotes encapsulation because components are hidden behind their interface, and their implementation is not exposed to the user of the class.
- This makes it easier to maintain and refactor code because changes to a component do not affect the rest of the class or its users.

Reduced Coupling with Composition

- Composition can reduce the coupling between classes because the components are only connected to the class through their interface.
- Inheritance can create a tight coupling between the superclass and its subclasses, making it harder to modify or replace the superclass without affecting its subclasses.

```
class Engine:
    def start(self):
        raise NotImplementedError

class GasEngine(Engine):
    def start(self):
        print("Starting gas engine")

class Car:
    def __init__(self, engine):
        self.engine = engine

    def start(self):
        self.engine.start()

car = Car(GasEngine())
car.start() # Output: Starting gas engine

# Let's add a new component to the Car class
class Transmission:
    def shift_up(self):
        print("Shifting up")

car.transmission = Transmission()
car.transmission.shift_up() # Output: Shifting up

# Let's add a new component to the GasEngine class
class ImprovedGasEngine(GasEngine):
    def start(self):
        super().start()
        print("Revving up")

car.engine = ImprovedGasEngine()
car.start() # Output: Starting gas engine, Revving up
car.engine.shift_up() # AttributeError: 'ImprovedGasEngine'
```

```
class Animal:
    def __init__(self, name):
        self.name = name

    def make_sound(self):
        pass

class Dog(Animal):
    def make_sound(self):
        print("Woof")

class Cat(Animal):
    def make_sound(self):
        print("Meow")

class Lion(Animal):
    def make_sound(self):
        print("Roar")

class Zoo:
    def __init__(self, animals):
        self.animals = animals

    def make_sounds(self):
        for animal in self.animals:
            animal.make_sound()

zoo = Zoo([Dog("Fido"), Cat("Fluffy"), Lion("Simba")])
zoo.make_sounds() # Output: Woof, Meow, Roar
```

Avoidance of the inheritance hierarchy problem with Composition

- Inheritance can lead to a deep and complex hierarchy, making it harder to understand and maintain code. Composition avoids this problem by using a simpler structure of objects connected through their interface.

```
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self, engine):
        self.engine = engine

    def start(self):
        self.engine.start()
        print("Car started")

class Truck:
    def __init__(self, engine):
        self.engine = engine

    def start(self):
        self.engine.start()
        print("Truck started")

engine = Engine()
car = Car(engine)
truck = Truck(engine)

car.start()    # Output: Engine started, Car started
truck.start()  # Output: Engine started, Truck started
```

Better code reuse with Composition

- Composition promotes code reuse by allowing you to reuse components across different classes.
- In contrast, inheritance can create code duplication because subclasses may inherit functionality that is not relevant to them.

Aggregation

What is Class Aggregation?

- Refers to the process of **building a new class by combining the functionalities of multiple existing classes.**
- It involves **creating a new class that can access the methods and attributes of other classes and use them to define its own behavior.**
- Instead of duplicating code in multiple classes, you can define a separate class that contains the shared functionality and use it as a component in other classes, making your code more efficient, easier to read, and maintain.

```
class A:
    def method_a(self):
        print("Method A")

class B:
    def method_b(self):
        print("Method B")

class C:
    def __init__(self):
        self.a = A()
        self.b = B()

    def method_c(self):
        self.a.method_a()
        self.b.method_b()

c = C()
c.method_c()  # Output: "Method A\nMethod B"
```


Concept of Aggregation

- Aggregation is a **concept in which an object of one class can own or access another independent object of another class.**
- It represents **Has-A's relationship.**
- It is a **unidirectional association** i.e. a **one-way relationship.**
- **For example, a parent can have many children but a child cannot have many parents.**
- In Aggregation, both the entries can survive individually which means ending one entity will not affect the other entity.

Aggregation Example

In this example, the Parent class contains a list of Child instances, establishing a **uni-directional relationship** where a parent can have multiple children.

However, a **child does not have any reference to its parent**, maintaining the uni-directional nature of the relationship.

```
class Parent:
    def __init__(self, name):
        self.name = name
        self.children = []

    def add_child(self, child):
        self.children.append(child)

class Child:
    def __init__(self, name):
        self.name = name

# Creating instances
parent = Parent("John")
child1 = Child("Alice")
child2 = Child("Bob")

# Adding children to the parent
parent.add_child(child1)
parent.add_child(child2)

# Accessing children of the parent
for child in parent.children:
    print(f'{parent.name}'s child: {child.name}')
```

Composition vs Aggregation

Composition vs Aggregation

From the given code, we are not creating an object of the Salary class inside the EmployeeOne class, rather than that we are creating an object of the Salary class outside and passing it as a parameter of EmployeeOne class.

121500

```
# Code to demonstrate Composition

# Class Salary in which we are
# declaring a public method annual salary
class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return (self.pay*12)+self.bonus

# Class EmployeeOne which does not
# inherit the class Salary yet we will
# use the method annual salary using
# Composition
class EmployeeOne:
    def __init__(self, name, age, pay, bonus):
        self.name = name
        self.age = age

        # Making an object in which we are
        # calling the Salary class
        # with proper arguments.
        self.obj_salary = Salary(pay, bonus) # composition

    # Method which calculates the total salary
    # with the help of annual_salary() method
    # declared in the Salary class
    def total_sal(self):
        return self.obj_salary.annual_salary()

# Making an object of the class EmployeeOne
# and providing necessary arguments
emp = EmployeeOne('Geek', 25, 10000, 1500)

# calling the total_sal method using
# the emp object
print(emp.total_sal())
```

```
# Code to demonstrate Aggregation

# Salary class with the public method
# annual_salary()
class Salary:
    def __init__(self, pay, bonus):
        self.pay = pay
        self.bonus = bonus

    def annual_salary(self):
        return (self.pay*12)+self.bonus

# EmployeeOne class with public method
# total_sal()
class EmployeeOne:

    # Here the salary parameter reflects
    # upon the object of Salary class we
    # will pass as parameter later
    def __init__(self, name, age, sal):
        self.name = name
        self.age = age

        # initializing the sal parameter
        self.agg_salary = sal # Aggregation

    def total_sal(self):
        return self.agg_salary.annual_salary()

# Here we are creating an object
# of the Salary class
# in which we are passing the
# required parameters
salary = Salary(10000, 1500)

# Now we are passing the same
# salary object we created
# earlier as a parameter to
# EmployeeOne class
emp = EmployeeOne('Geek', 25, salary)

print(emp.total_sal())
```

Drawbacks of Composition

- As we saw in the previous snippet, we are creating an object of the Salary class **inside** EmployeeOne class which **has no relation** to it.
- So **from the outside**, if we **delete** the **object** of EmployeeOne class i.e emp in this case, then **the object of Salary class i.e obj_salary will also be deleted** because **it completely depends** upon the EmployeeOne class and its objects
- To solve this dependency problem, Aggregation came into the picture.

When to use Aggregation instead of Composition?

- Use **aggregation** when **one object** is made up of **multiple, independent** objects that **can exist on their own**.
- The objects in **aggregation** have a **weak, independent** relationship where **one can exist without** the other.
- Aggregation is **useful when creating objects that use common functionalities or resources** that can be **shared** among **multiple instances**.
- Aggregation provides greater **flexibility** and **reusability** by allowing the shared object to be passed around to other objects.

Example of using Aggregation over Composition

In this example, we have two classes: Book and Library. The Book class represents a book and has two attributes: title and author.

The Library class represents a library and has an instance variable books that is a list of Book objects. When a Library object is created, it creates an empty list to hold the Book objects using aggregation.

The Library class has three methods: add_book(), remove_book(), and get_books(). These methods allow us to add, remove, and retrieve Book objects from the books list, respectively.

In this example, we use aggregation because the **Book objects are independent of the Library object**. The Library object only holds a reference to the Book objects using the books list. The Book objects can exist on their own without the Library object.

Therefore, using aggregation allows us to create a loosely coupled relationship between the Library and Book objects, which is appropriate in this scenario.

```
class Book:
    def __init__(self, title, author):
        self.title = title
        self.author = author

class Library:
    def __init__(self):
        self.books = []

    def add_book(self, book):
        self.books.append(book)

    def remove_book(self, book):
        self.books.remove(book)

    def get_books(self):
        return self.books
```

When to use **Composition** instead of Aggregation?

- Use composition when one object is a fundamental part of another object, and their lifecycles are closely tied together.
- The objects in composition have a strong, interdependent relationship where one cannot exist without the other.
- Composition is useful when creating complex objects that are composed of smaller, reusable parts.
- Composition provides stronger encapsulation and more control over the behavior of the composed object.

Example of using Composition over Aggregation.

In this example, we have two classes: Engine and Car. The Engine class represents the engine of a car, and it has two methods: start() and stop() that start and stop the engine, respectively.

The Car class represents a car and has an instance variable engine of the Engine class. When a Car object is created, it also creates an Engine object using composition.

The Car class has two methods: start() and stop(). These methods call the corresponding methods of the Engine object, which was initialized in the constructor using composition.

In this example, we use composition because the Engine object is a fundamental part of the Car object, and its lifecycle is closely tied to that of the Car object. Without the Engine object, the Car object cannot function properly.

Therefore, using composition allows us to create a tightly coupled relationship between the Car and Engine objects, which is appropriate in this scenario.

```
class Engine:
    def start(self):
        print("Engine started.")

    def stop(self):
        print("Engine stopped.")

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        self.engine.start()

    def stop(self):
        self.engine.stop()
```

Summary of Composition vs Aggregation

In summary:

- Use **composition** when you need to create complex objects that have a strong, interdependent relationship between their parts.
- Use **aggregation** when you need to create objects that have a weak, independent relationship between their parts, and can share common functionalities or resources.

Python Mixins

What is a Mixin?

- A class featuring a set of attributes / methods that can be used to provide extra functionality to another class.
- Not considered to be a base class itself.
- Typically, through inheritance, we ensure that objects get the functionality of the base class they're extending.
- A mixin provides functionality without actually being included in the inheritance tree, functioning more as additions to the class, rather than proper ancestors.
- As such, the is-a relationship is no longer a thing when it comes to mixins, as we're used to in a regular case of inheritance.

Why not just add mixin functions to the base class?

- Now, you might then ask the very logical question: why don't we simply add the **mixin** functionality to the base class, so the objects extending the base class can inherit all functionality from one place? And the answer's pretty simple: because it wouldn't make sense to do so. But let's exemplify.

Mixin example

Let's assume we have a base class, called Lifeform. And three classes inheriting from it, called Horse, Eagle and Hummingbird, like so:

All three classes inherit the eat() instance method from the base class.

However, for Eagle and for Hummingbird, we may want to also define the fly() method. But we can't add that to the Lifeform base class, because that would make Horse inherit the wings attribute. And horses don't have wings, as far as we know :)

One way we could do this is to provide this functionality in each of the Eagle and Hummingbird classes, but we'd end up with duplicate code.

```
class Lifeform():  
    def eat(self):  
        print("Eating")
```

```
class Horse(Lifeform):  
    pass
```

```
class Eagle(Lifeform):  
    pass
```

```
class Hummingbird(Lifeform):  
    pass
```

Another way of using mixin

Notice how the Eagle and Hummingbird both inherit from Lifeform and from FlierMixin, but Horse only inherits from Lifeform?

This way, we managed to avoid code duplication and provide the fliers with the ability to fly. We can now call fly() only on Hummingbird and Eagle objects, but not on Horse objects:

The Horse can't fly. The way we designed that class, it can only eat. The Eagle, on the other hand, has the eat() functionality from the base class, Lifeform, but also the fly() functionality from the FlierMixin class. Without this mixin class, we would have needed to provide the same fly() functionality in each of the flier classes (Eagle and Hummingbird), potentially achieving unnecessary code duplication.

```
class Lifeform():
    def eat(self):
        print("Eating")

class FlierMixin():
    def fly(self):
        print("Flying")

class Horse(Lifeform):
    pass

class Eagle(Lifeform, FlierMixin):
    pass

class Hummingbird(Lifeform, FlierMixin):
    pass
```

```
my_eagle = Eagle()
my_eagle.eat()
my_eagle.fly()
```

```
my_horse = Horse()
my_horse.eat()
my_horse.fly()
```

Output:

Eating

Flying

Eating

Traceback (most recent call last):

...

my_horse.fly()

AttributeError: 'Horse' object has no attribute 'fly'

Mixin vs Multiple Inheritance

Mixin vs Multiple Inheritance

- There are lots of cases, like the one we just described, where the difference between the mixin design pattern this article proposes to showcase and the concept known as multiple inheritance is very subtle and quite hard to detect. After all, our example did feature a Lifeform class and a FlierMixin class. Our Eagle class inherits from both the Lifeform and FlierMixin classes and this is clearly a case of Multiple Inheritance.
- However, what differentiates the two is that, while Lifeform is a class that is meant to be instantiated, FlierMixin is not. Of course, we could choose to instantiate FlierMixin, no one is ever going to be stopping us from doing that in Python, but we don't really have much reason to, since it only provides the fly() ability to other classes and there's no real justification for using a FlierMixin object on its own.

Mixin vs Multiple Inheritance Simplified

- The main difference between multiple inheritance and mixins is that in multiple inheritance, the base classes are expected to have their own functionality that is relevant to the derived class, while in mixins, the base class only provides additional functionality to the derived class. In other words, the derived class is a combination of the functionalities of all its base classes in multiple inheritance, while in mixins, the derived class inherits only the functionality provided by the mixin.

Mixins simplified

- A class in Python that **provide optional or additional functionality to be added to other classes.**
- Its main purpose is to **provide reusable functionality for multiple classes, helping to reduce code duplication and improve code modularity.**
- They can be thought of as **plugins or modules that can be added to a specific class, allowing it to have extended functionality without modifying the base class itself.**
- Because they are **independent units** of functionality, they can be **easily added or removed from a class without affecting the underlying class or its behavior.**
- Their independent nature **allows classes to receive new features or functionality without affecting a specific base class.** This helps to improve **code flexibility and modularity.**

Things to watch out when implementing mixins.

- Name conflicts: Because mixins introduce new methods and attributes to a class, there is a risk of name conflicts if the mixin and the class both define a method or attribute with the same name. This can make it difficult to understand the behavior of the class, and can lead to unexpected results. **ADD A MIXIN TO YOUR MIXIN NAMES!**
- Coupling: Mixins can introduce tight coupling between classes, which can make it difficult to modify or extend them in the future. If a mixin is heavily used throughout a codebase, it can be challenging to remove or replace it without affecting a large number of classes.
- Inheritance complexity: Mixins can make class inheritance more complex, especially when multiple mixins are used together. This can make it harder to understand the relationships between classes, and can make it more challenging to debug or modify the code.
- Overuse: Finally, there is a risk of overusing mixins, which can lead to code that is difficult to understand and maintain. If too many mixins are added to a class, it can become bloated and difficult to work with, especially for developers who are unfamiliar with the codebase.

Reducing Coupling

- Use composition instead of inheritance: Rather than inheriting from a base class or mixin, create a new class that contains an instance of the base class or mixin. This allows you to selectively expose only the functionality you need, without creating a tight coupling between the two classes.
- Use interfaces: Interfaces define a contract that classes must follow in order to use a particular piece of functionality. By relying on interfaces rather than concrete classes or mixins, you can reduce coupling and make your code more flexible.
- Keep mixins small and focused: Mixins should be designed to provide a specific unit of functionality, rather than trying to be a catch-all solution for a wide range of use cases. This can help keep the coupling between mixins and other classes to a minimum.
- Use dependency injection: Rather than creating instances of classes or mixins directly within a class, use a dependency injection framework or design pattern to pass instances of dependencies into the class as parameters. This can help reduce the tight coupling between classes and their dependencies.
- Keep the inheritance hierarchy shallow: The deeper the inheritance hierarchy, the more complex the relationships between classes can become. Try to keep the inheritance hierarchy as shallow as possible, and prefer composition over inheritance whenever possible.

Association

Association

Represents a relationship between two or more classes where objects of one class are related to objects of another class.

It can be a one-to-one, one-to-many, or many-to-many relationship.

Note: Association is a more generic term and can be implemented using attributes, methods, or other relationships.

```
class Person:
    def __init__(self, name):
        self.name = name

class Department:
    def __init__(self, department_name):
        self.department_name = department_name

class Employee:
    def __init__(self, person, department):
        self.person = person
        self.department = department
```

Best time to consider association

- When you want to represent relationships between classes without implying a specific level of dependency or containment.
- It provides a more flexible and generic way to model connections between objects. Here are some scenarios where association is a suitable choice:

Loose Coupling against Inheritance and Composition

```
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self, engine):
        self.engine = engine

    def start(self):
        return self.engine.start()

# Creating instances
engine = Engine()
car = Car(engine)

# Using Association
print(car.start()) # Outputs: Engine started
```

```
class Engine:
    def start(self):
        return "Engine started"

class Car(Engine):
    pass

# Using Inheritance
car = Car()
print(car.start()) # Outputs: Engine started
```

```
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        return self.engine.start()

# Using Composition
car = Car()
print(car.start()) # Outputs: Engine started
```

Association versus Aggregation

Association is a more generic term, indicating a relationship between classes without specifying the nature or strength of the relationship. It can be used for various types of connections.

Aggregation is a specific form of association that implies a stronger relationship, often involving a "whole-part" connection.

It suggests ownership, but the "part" can exist independently.

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

class Driver:
    def __init__(self, name):
        self.name = name

# Association between Car and Driver
car = Car("Sedan")
driver = Driver("John")
```

```
class Engine:
    def start(self):
        return "Engine started"

class Car:
    def __init__(self):
        self.engine = Engine()

# Aggregation between Car and Engine
car = Car()
```

Next Topics

- Encapsulation
- Polymorphism