OOP - Advanced

Instructors **Battista Biggio**, **Angelo Sotgiu** and **Leonardo Regano**

M.Sc. in Computer Engineering, Cybersecurity and Artificial Intelligence
University of Cagliari, Italy

Polymorphism

Polymorphism is the provision of a single interface to entities of different types[1]. Polymorphism enables a programming language to dynamically determine which method to use at runtime, depending on the parameters sent to the method.

```
# Example: the implementation of `f()` depends on the object class.
obj = A()  # B()
obj.f()
```

---

[1] https://www.stroustrup.com/glossary.html#Gpolymorphism

Dynamic dispatch is the process of selecting which implementation of a polymorphic operation (method or function) to invoke at run time.
The decision on which version of a method to call can be based either on a single object/parameter (single dispatch), or on a combination of objects (multiple dispatch).

- **Single dispatch** is a type of polymorphism where only one parameter is used to determine the call. Typically, this parameter is the object itself (**self**, or **this**).
- **Multiple dispatch** is a form of polymorphism where multiple parameters are employed in determining which method to call.

Example

```
a.divide(b)   # `a/b`
```

Single dispatch: the implementation will be chosen based only on `a` type (rational, floating point, matrix,..), disregarding the type or value of divisor `b`.

Multiple dispatch: the implementation will be chosen based on the combination of operands. The `dividend` and `divisor` types determine which kind of `divide` operation will be performed.

Exercise

Write the classic `rock-paper-scissor` [2] game.

Rock, paper and scissor are instances of the classes `Rock`, `Paper`, `Scissor`, respectively (but this is not the only possibility).
The three classes are subclasses of the abstract class `Weapon`.

---

[2] https://en.wikipedia.org/wiki/Rock_paper_scissors

Define the classes in a way that objects respond to the `fight_against` method as follows:

```
list_of_weapons = [Scissor(), Paper(), Rock()]
for w1 in list_of_weapons:
  for w2 in list_of_weapons:
    print(w1, "vs", w2, "->", w1.fight_against(w2))
```

```
Output
--------------------
Scissor vs Scissor -> TIE
Scissor vs Paper -> Scissor
Scissor vs Rock -> Rock
Paper vs Scissor -> Scissor
Paper vs Paper -> TIE
Paper vs Rock -> Paper
Rock vs Scissor -> Rock
Rock vs Paper -> Paper
Rock vs Rock -> TIE
```

We can start from the classic implementation using a conditional structure in the `fight_against` method

```
from abc import ABC, abstractmethod


class Weapon(ABC):

  @abstractmethod
  def fight_against(self, other_weapon):
    pass

  def __str__(self):
    return self.__class__.__name__


class Scissor(Weapon):
  def fight_against(self, other_weapon):
    if isinstance(other_weapon, Paper):
      return "Scissor"
    elif isinstance(other_weapon, Rock):
```

Consider extending the game to incorporate additional weapons, such as in "Rock Paper Scissors Spock Lizard"[3].

The 'IF' based solution has several limitations:

- **Conditional Structure Extension**: If new classes are added, the conditional structure must be extended by adding new branches. This leads to code that is tightly coupled and requires modifications whenever new weapons are introduced.
- **Code Duplication**: The addition of new branches in the conditional structure may result in portions of code that are similar but not exactly identical. This redundancy complicates code maintenance and can lead to inconsistencies.
- **Dependency on Implementation Order**: The order in which conditions are checked in the 'IF' structure can be crucial. Changing the order may alter the behavior of the code.

---

[3] https://en.wikipedia.org/wiki/Rock_paper_scissors#Additional_weapons

To avoid the limitations and adhere to the **Open/Closed Principle (OCP)**[4], it is possible to use the 'double dispatch' approach.

The behavior of the method `fight_against` is determined dynamically at runtime based on the weapon types, allowing for more flexibility and scalability.

When invoking `w1.fight_against(w2)`, it triggers a method in `w2` to carry out the task.
The specific `w2` method is dynamically determined at runtime based on the weapon classes, eliminating the need for a conditional structure.
This is possible because both `w1` and `w2` are aware of their own classes.

This way, the behavior of the game can be extended without modifying the existing code.
New weapons can be introduced by adding new classes, with a more modular and maintainable codebase.

---

[4] Code should be **open for extension**, but **closed for modification**

A possible solution

```
from abc import ABC, abstractmethod


class Weapon(ABC):

  def __str__(self):
    return self.__class__.__name__

  @abstractmethod
    def fight_against(self, other_weapon):
      pass

  # second dispatch
  @abstractmethod
  def _fight_against_scissor(self):
    pass

  @abstractmethod
```

An alternative solution using methods overloading

```
from abc import ABC, abstractmethod
from plum import dispatch


class Weapon(ABC):

  def __str__(self):
    return self.__class__.__name__

  @abstractmethod
  @dispatch
  def fight_against(self, other_weapon: "Scissor"):
    pass

  @abstractmethod
  @dispatch
  def fight_against(self, other_weapon: "Rock"):
    pass

  @abstractmethod
```

Comment

This problem is very simple, and double dispatch was adopted only to illustrate how it works. Given the simplicity of the problem, the conditional structure is appropriate.

One alternative is to use a single class, employing a string to identify the type of weapon and a look-up table to determine the winner. This approach could simplify the code structure and eliminate the need for multiple classes.

Exercise - Double dispatch

R is the class of real number, and **C** is the class of the complex number[5].

```
class MathEntity(ABC):
  ...

class R(MathEntity):

  def __init__(self, re):
    self.re = re

class C(MathEntity):
  """represents re + i img """

  def __init__(self, re, img=0):
    self.re = re
    self.img = img
```

[5] This is a toy problem - real and complex are already implemented in python.

Write methods to sum instances of `R` and `C` using the 'double dispatch' approach. (Try even other solutions, like 'lookup-table' and conditional structure with explicit type checking. Discuss the differences.)

```
values = [R(2), R(-3), C(2,3), C(3,-3)]
for op1 in values:
  for op2 in values:
    print(op1 , "+" , op2 ,"=", op1 + op2)
```

```
Output
--------------------
2 + 2 = 4
2 + -3 = -1
2 + 2+i3 = 4+i3
2 + 3-i3 = 5-i3
-3 + 2 = -1
-3 + -3 = -6
-3 + 2+i3 = -1+i3
-3 + 3-i3 = 0-i3
2+i3 + 2 = 4+i3
2+i3 + -3 = -1+i3
2+i3 + 2+i3 = 4+i6
2+i3 + 3-i3 = 5
3-i3 + 2 = 5-i3
3-i3 + -3 = 0-i3
3-i3 + 2+i3 = 5
3-i3 + 3-i3 = 6-i6
```

Solution (naive double dispatch)

```
class MathEntity(ABC):
  # define abstractmethods __init__, __repr__, __add__


class R(MathEntity):

  def __init__(self, re):
    self.re = re

  def __repr__(self):
    return str(self.re)

  def __add__(self, op2):
    return op2._add_real(self)

  def _add_real(self, op1):
    return R (self.re + op1.re)

  def _add_complex(self, op1):
    return C (self.re + op1.re, op1.img)
```

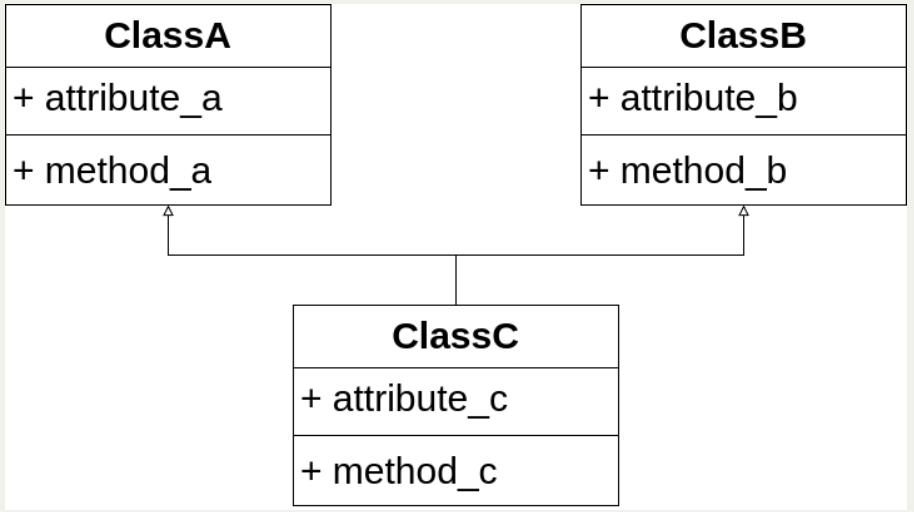
Solution (method overloading)

```python
from abc import ABC, abstractmethod
from plum import dispatch


class MathEntity(ABC):
  # define abstractmethods __init__, __repr__, __add__


class R(MathEntity):

  def __init__(self, re):
    self.re = re

  def __repr__(self):
    return str(self.re)

  @dispatch
  def __add__(self, op2: "R"):
    return R(self.re + op2.re)
```

We can use the double dispatch approach when the behavior depends on the pair of objects involved in the relationship, and we plan to extend the number of classes involved.

Multiple Inheritance

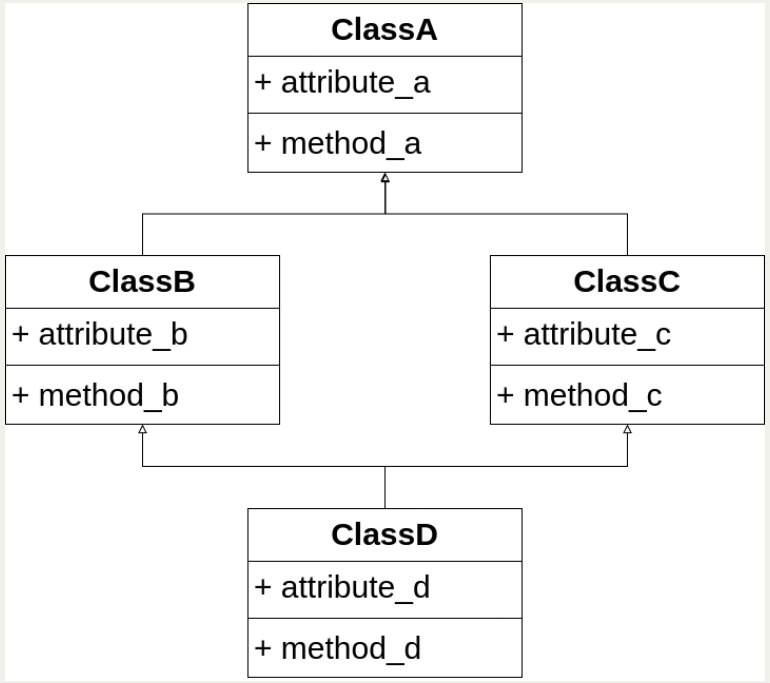
```
class A:
  ...


class B:
  ...


class C(A, B):
  ...
```



What happens if the parent classes have attributes/methods with the same names (e.g., the `__init__` method)?

The diamond problem

What happens in this case, if we want to call the `__init__` methods of all the classes when instantiating a ClassD object?



Mixin Classes

Mixin Classes are classes that does not belong to any inheritance hierarchy, and are not intended to be instantiated.

Their purpose is to contain functionalities that can be imported from other classes with inheritance, providing reuse of code without the needing to construct complex hierarchies.

They can also be used to build classes by composition.