Good practices 2: Fundamentals of Object Orientation

Javier Nogueras
jnog@unizar.es

TWEED Project, SC1

Contents

1. Introduction to object orientation

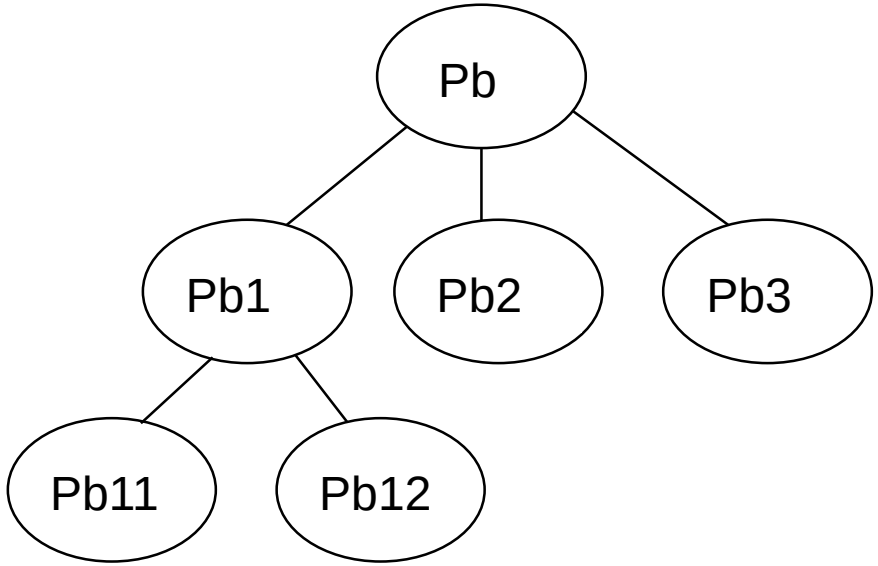
- The goal of Software Engineering (SE) is to find ways to build software with quality
  - "SE = Coordinated application of techniques, methodologies and tools to produce high-quality software, within a given budget and by a given date"

Software quality criteria

- Attributes that define software quality:
  - Functionality: The degree of certainty that the system processes the information accurately and completely.
  - Flexibility: The degree to which the user may introduce extensions or modifications to the information system without changing the software itself.
  - Maintainability: The ease of adapting the information system to new demands from the user, to changing external environments, or in order to correct defects.
  - Reusability: The degree to which parts of the information system, or the design, can be reused for the development of other applications.
  - Other: efficiency, user-friendliness, portability ...

Modularity

- Modularity (divide and conquer) is one of the most direct mechanisms to achieve the objectives of flexibility, maintainability, and reusability:



Object Orientation

- Methodology: collection of methods, notations, processes and tools applied throughout the development process (analysis, design, implementation, testing, ...)
- Object-Oriented methodologies:
  - They use an object model as an abstraction of the system to be developed
  - The system is seen as a collection of interacting objects
  - An object is a concept, abstraction or physical entity with well-defined boundaries and meaning in the problem domain
    - Example: *Donald Trump*, *Google*, or a *table* are objects
  - Relationships are established between objects that reflect the structure of the real world

Object model

- The object model facilitates the modularity
  - Objects are small modules that encapsulate state and behaviour
  - The state represents the current values of the properties (attributes and bindings) of the object.
    - The state of the object evolves over time
    - The state of the object recalls the effect of the operations on itself
  - Behavior is the set of operations of the object
    - Behavior explains the object's competences: the actions and reactions it can perform

Gradual transition between analysis, design and implementation

- During analysis (what does it do?) and design (how is it solved?) UML is commonly used
  - UML = Unified Modeling Language
  - Widely used notation for object-oriented modelling
  - Includes graphical notations for constructing diagrams to represent specific aspects of the model
    - The static structure of the system by means of class diagrams, package diagrams, deployment diagrams, ...
    - The behaviour of the system through use case diagrams, interaction (sequence) diagrams, ...
- Object-oriented programming languages are used for the implementation, which directly support the concepts of the object model

Key concepts inherent in the object model

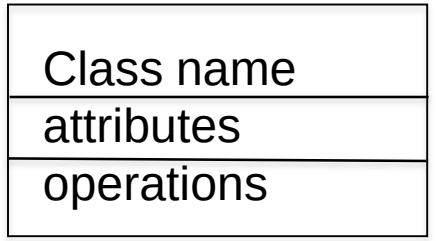
- Abstraction: focusing on the essentials
- Encapsulation: hiding implementation details
- Hierarchy: ordering or classification of abstractions
- Polymorphism: sending syntactically the same messages to objects of different types
- Modularity: division into cohesive and loosely coupled modules

2. Object Oriented Programming in Python

- We will introduce the key concepts of object-orientation and its implementation in Python
  - It reduces the complexity of programs and improves its maintainability
  - As opposed to primitive data structures (lists, tuples, dictionaries), users can define their own data structures to naturally model the reality of the problem they want to solve
- The examples integrated in the following slides are available in this GitHub repository:
  - https://github.com/javierni/oop-examples

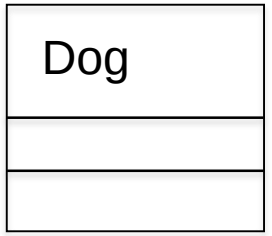
2.1. Abstraction

- It allows you to focus on the inherent essentials of an entity, and ignore its incidental properties
- The main task in the abstraction process is the identification of objects
  - The object model should reflect the problem domain.
- A class describes a group of objects that share properties (attributes and operations), constraints, relationships with others and have common semantics
  - Person, Company or Table are classes
- Representation of classes in UML: name, attributes, operations



Defining classes in Python

Class diagram in UML:



- Definition of a class

  `class <Class name>:`

- Instantiation: creation of objects

  `<Class name>()`

```
# dog1.py

class Dog:
    pass
    #pass allows error-free code execution
```

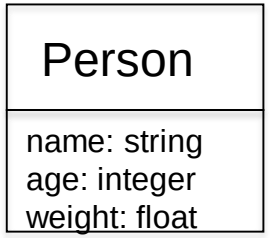
Code

```
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
```

Python Terminal

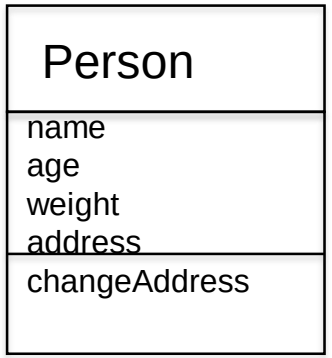
Attributes

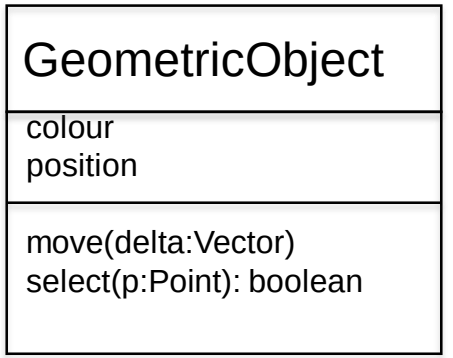
- *name, age* and *weight* are attributes of *Person*
  - Describe the properties of the objects
- Each attribute has a value in each instance
- Instantiate is the operation of creating objects of a class
  - Instance = object belonging to a class
- When attributes have the same value in all instances, they are called *class attributes*
- The name of the attribute is unique within the class
- You can specify the type of the values of an attribute



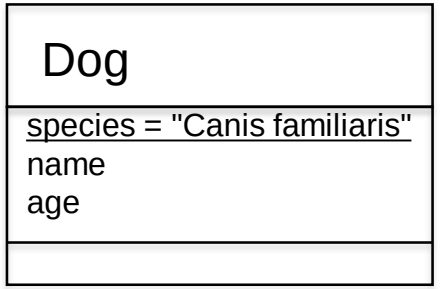
Operations and methods

- An operation is a transformation that can be applied to/by the objects of a class
  - *hire, fire, pay_dividends* can be operations of a *Company* class
  - *sing, laugh* and *jump* can be operations of a *Person* class
- Objects of a class share the same operations
- Operations have arguments
  - The number of arguments, their type and the return type of an operation is called *signature*
- A method is the implementation of an operation for a class
  - Constructors: methods that implement instantiation operations





Definition of class attributes and instance attributes in Python

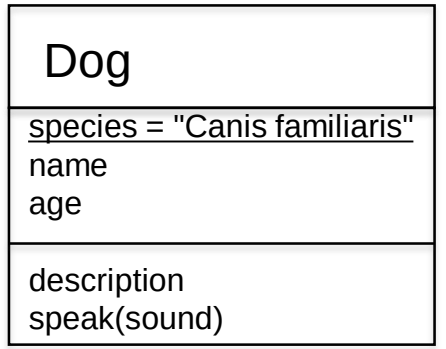


- Instance attributes are defined within the *__init__()* constructor
  - Can have any number of parameters
  - First parameter: reference to the instance just created (Python sends it automatically during instantiation)
    - By convention it is usually called *self* but another identifier could be used
- Class attributes are defined outside the *__init__* method and an initial value must be assigned
- Notation *.<attribute name>* for accessing the attribute value

```
# dog2.py
class Dog:
    # Identation to indicate that an
attribute/method belongs to the class
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

```
>>> miles = Dog("Miles", 4)
>>> buddy = Dog("Buddy", 9)
>>> miles.name
'Miles'
>>> buddy.name
'Buddy'
>>> buddy.species
'Canis familiaris'
>>> Dog.species = "Perro"
>>> buddy.species
'Perro'
```

Definition of instance methods in Python



- Methods defined within a class using identation
  - The first parameter is the *self* reference (no need to pass it in the invocation)
- Notation .*<method name>* for accessing the method of an object

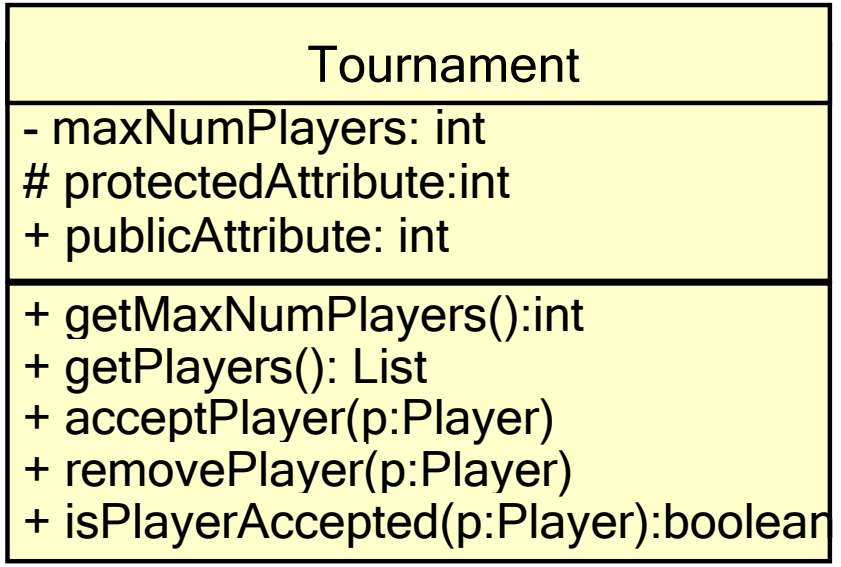
```
# dog3.py
class Dog:
    species = "Canis familiaris"
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def description(self):
        return f"{self.name} is {self.age} years old"
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

```
>>> miles = Dog("Miles",4)
>>> miles.description()
'Miles is 4 years old'
>>> miles.speak("Woof Woof")
'Miles says Woof Woof'
```

2.2. Encapsulation

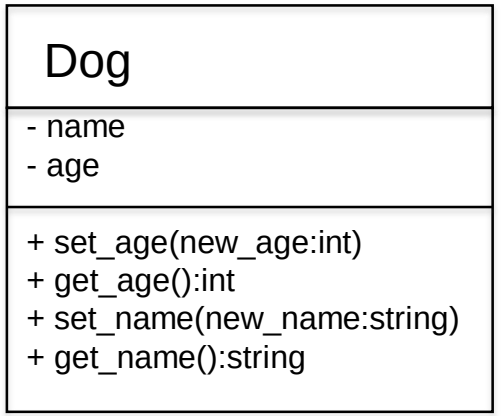
- It’s the process of hiding all details of an object that do not contribute to its essential characteristics
- Hiding information about the implementation of an object means that no part of a complex system depends on the internal details of particular objects
- Encapsulation and abstraction go hand in hand:
  - abstraction focuses on the external view
  - and encapsulation on the internal view

Levels of visibility



- UML defines 3 levels of visibility for the members of a class:
  - Private (for the implementers of the class):
    - A private attribute can only be accessed by the class where it is defined
    - A private operation can only be invoked by the class where it is defined
    - Private attributes and operations are not accessible by subclasses or any other class
  - Protected (for extension/subclass creators):
    - A protected attribute or operation can be accessed by the class where it is defined and any descendant of the class
  - Public (for users of the class):
    - A public attribute or operation can be accessed by any class

Encapsulation in Python



- There are no specific modifiers to distinguish between *private, protected* and *public*
- By convention, non-public members use an identifier starting with an underscore (_), but this does not prevent direct access

```
# dog4.py
class Dog:
    def __init__(self, name, age):
        self._name = name # Private attribute
        self._age = age # Private attribute
    def set_age(self, new_age):
        self._age = new_age
    def get_age(self):
        return self._age
    ...
```

```
>>> miles = Dog("Miles",4)
>>> miles._age = 6 # Not recommended
>>> miles._age # Not recommended
6
>>> miles.set_age(7) # Recommended
>>> miles.get_age() # Recommended
7
```

Name mangling

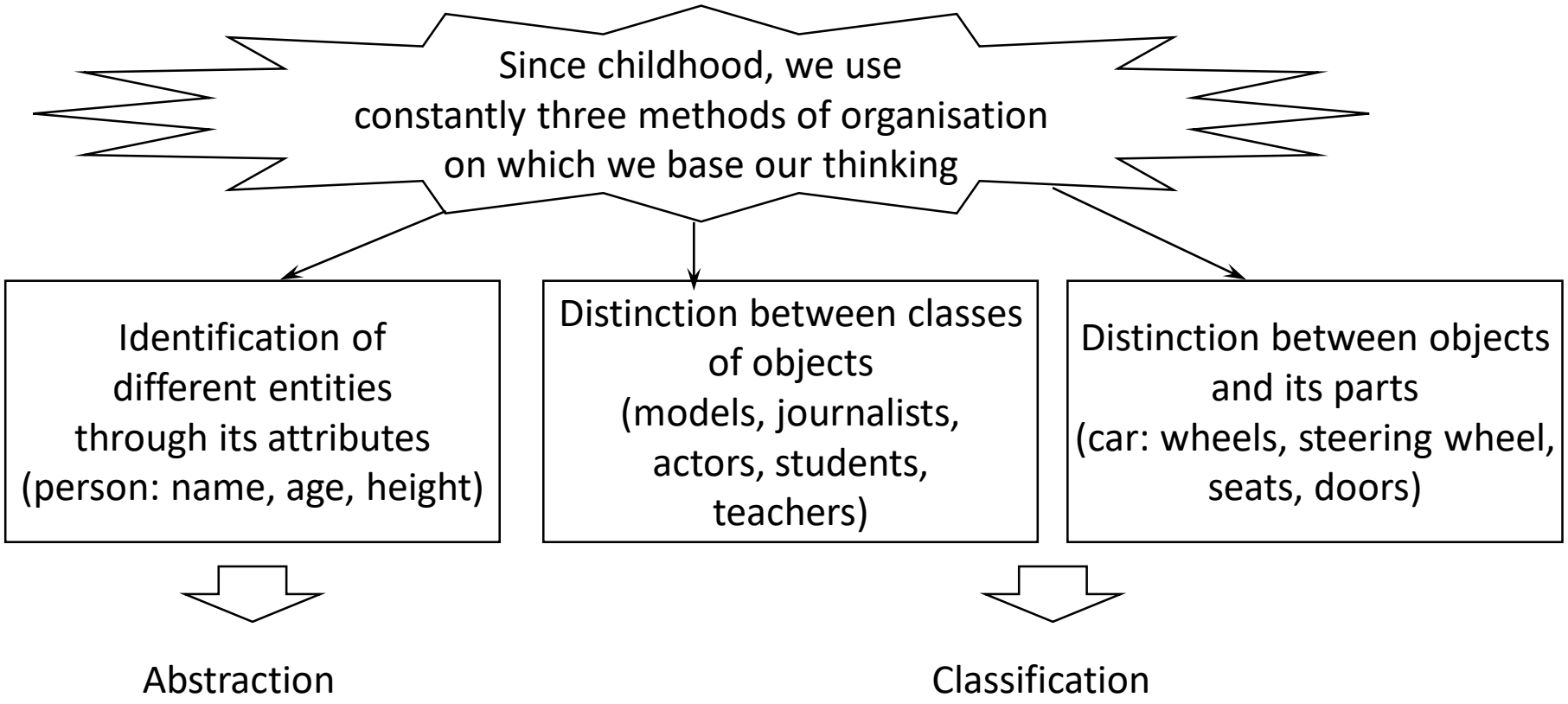
- To reinforce the encapsulation, a double underscore (__) can be used as a prefix
  - This causes automatic renaming (name mangling) and the identifier becomes _*<Class name>*__*<member name>*
  - Direct accesses with __*<member name>* cause an error

```
# dog5.py
class Dog:
    def __init__(self, name, age):
        self.__name = name # Private attribute
        self.__age = age # Private attribute
    def set_age(self, new_age):
        self.__age = new_age
    def get_age(self):
        return self.__age
    ...
```

```
>>> miles = Dog("Miles",4)
>>> miles.__age
 Traceback (most recent call last):
   File "<stdin>", line 1, in <module>
AttributeError: 'Dog' object has no
attribute '__age'
>>> miles._Dog__age
4
```

2.3. Hierarchy

- It’s the ordering or classification of abstractions

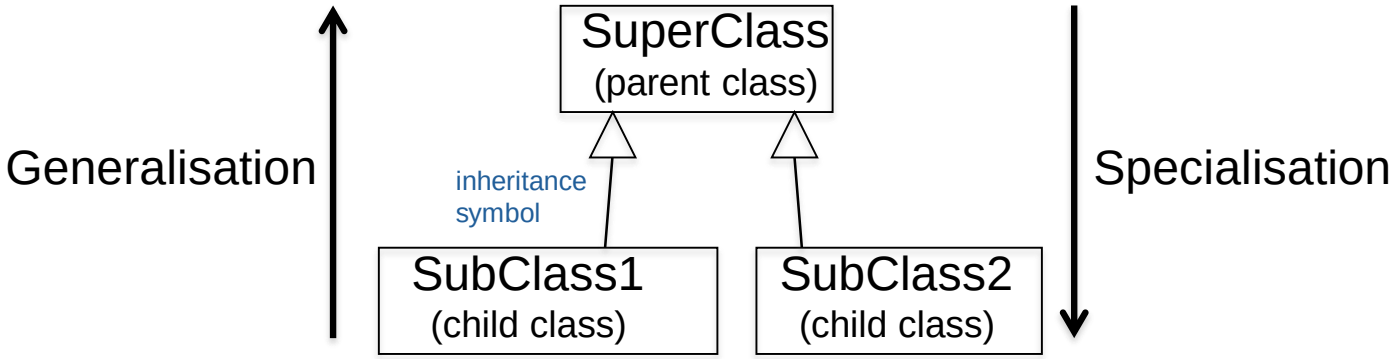


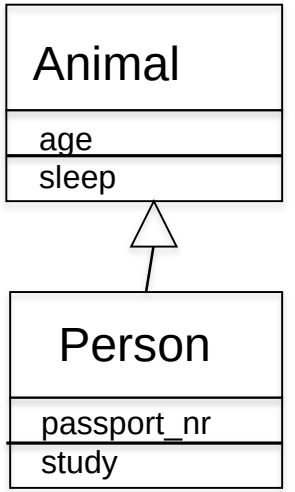
Types of hierarchies

- When we model a system we realise that very few classes are isolated
  - Most of them collaborate with others in various ways
- Main types of relationships/hierarchies between objects
  - Inheritance
    - It’s a generalisation/specialisation relationship
  - Association/aggregation/composition
    - It’s a structural relationship

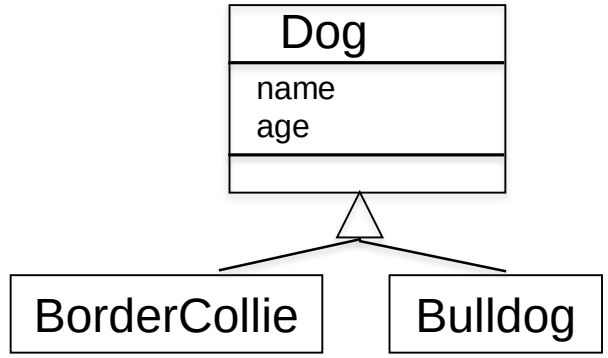
Inheritance

- It’s an abstraction to share similarities between classes while maintaining their differences
- It’s also known as the “is-a” relationship
- Each instance of a subclass is an instance of the superclass
- Subclasses inherit attributes and methods from the superclass and can add their own
  - Inherited or derived part
  - Emergent or incremental part





Inheritance in Python

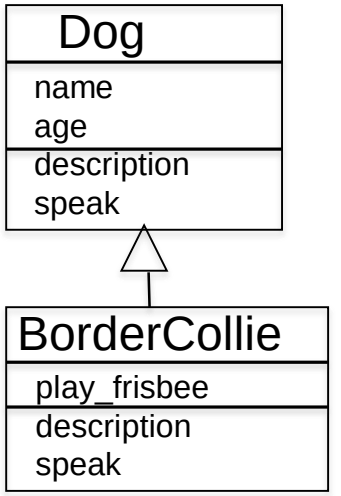


- When defining the subclass, the name of the parent class is included in parentheses

```
# dog6.py
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    # __str__: special method returning a string representation
    def __str__(self):
        return f"{self.name} is {self.age} years old"
class BorderCollie(Dog):
    pass
class Bulldog(Dog):
    pass
```

```
>>> robert = BorderCollie("Robert",8)
>>> print(robert)
Robert is 8 years old
>>> type(robert)
<class '__main__.BorderCollie'>
>>> isinstance(robert,BorderCollie)
True
```

New members and redefinition of methods in subclasses



- Apart from redefining the behavior, it's possible to reuse the behavior of the parent class with the reserved word *super*

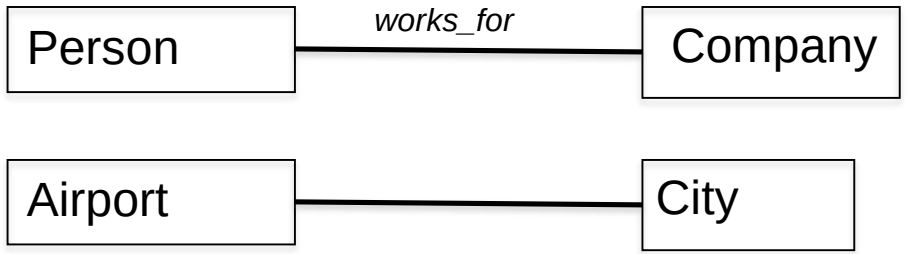
```
# dog7.py
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def description(self):
        return f"{self.name} is {self.age} years old"
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

```
>>> robert = BorderCollie("Robert",8,True)
>>> robert.description()
'Robert is 8 years old and plays frisbee'
>>> robert.speak() 'Robert says Wow'
```

```
class BorderCollie(Dog):
    def __init__(self, name, age, play_frisbee):
        super().__init__(name,age)
        self.play_frisbee = play_frisbee
    def description(self):
        if self.play_frisbee:
            return f"{self.name} is {self.age} years old and plays frisbee"
        else:
            return f"{self.name} is {self.age} years old"
    def speak(self, sound="Wow"):
       return super().speak(sound)
```

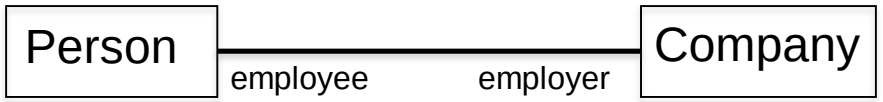
Association

- An association is a relationship that indicates that there is a physical or conceptual connection between objects
- Basic concepts
  - Name of the association: the association can have a name that describes the nature of the relationship
  - Roles (see below)
  - Multiplicity (see below)



Roles

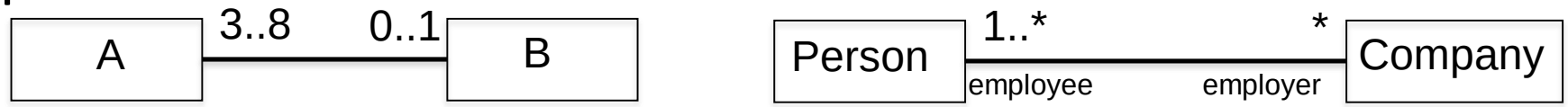
- The role that a class plays in the association can be explicitly stated
- The role is the face that one class presents to the class at the other end of the association



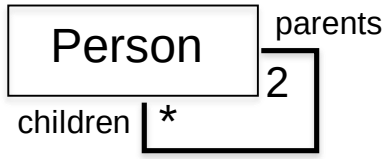
- A class can play the same or different roles in other associations
- Roles can be seen as an alternative to the name of the association
- Roles can coexist with the name of the association

Multiplicity

- It indicates how many objects of one class relate to a single instance of the other class in the association.
- In short: Given an object of class A, how many objects of class B does it relate to at least and at most?
- Representation: *MinValue .. MaxValue*
- Examples:

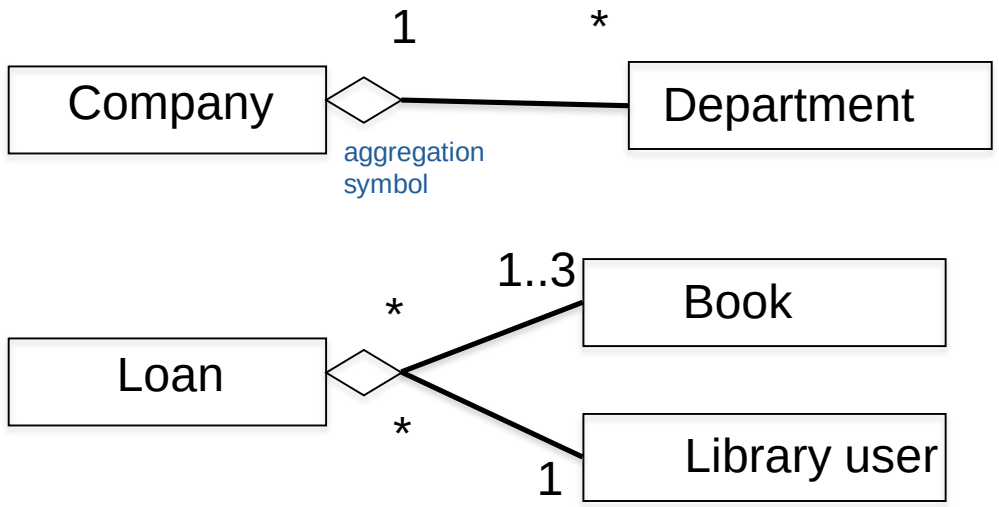


  - If nothing is stated, the multiplicity is 1..1
  - 1 is equivalent to 1..1
  - * is equivalent to 0..*



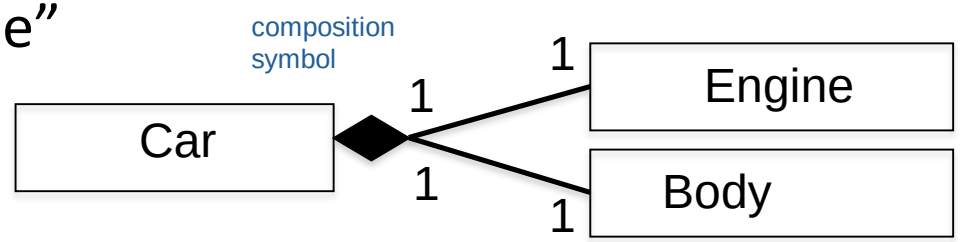
Aggregation

- An aggregation is a special form of association
- It models a “whole/part” or “has a” relationship
- A class (the “whole”) consists of smaller elements (the “parts”)



Composition

- A composition is a more restrictive, stronger form of aggregation
- An aggregation does not link the lives of the “whole” and the “parts”
- An aggregation only distinguishes the “whole” from the “parts”
- Composition:
  - It’s a relationship of belonging and coinciding lives of the “part” with the “whole”
  - The “parts” can be created after the “whole”, but once created they live and die with the “whole”
  - The “whole” manages the creation and destruction of the “parts”
  - The “parts” have no independent existence

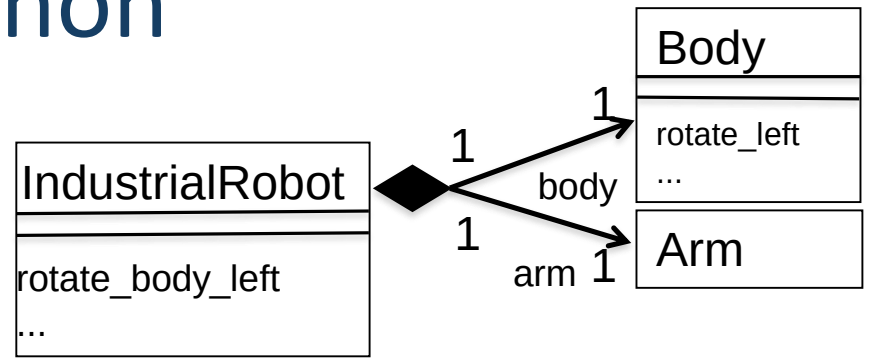


Structural relationships in Python

- The most common relationships are binary relationships (between 2 classes)
- The roles at the extremes of the association/composition/aggregation are translated into instance attributes
- If the maximum multiplicity is 1, these attributes are initialised with an instance of the related class
- If the maximum multiplicity is *, we need to manage the objects associated with a linear data structure (list, set, ...)

Example of 1:1 composition in Python

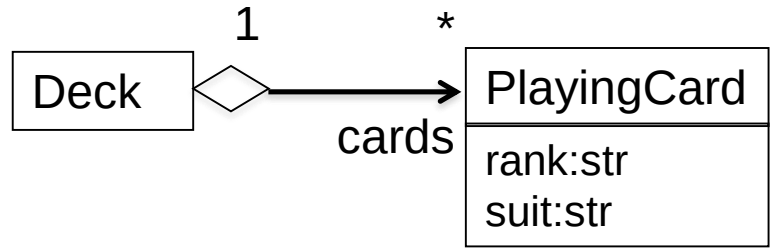
```
# robot.py
class IndustrialRobot:
    def __init__(self):
        self.body = Body()
        self.arm = Arm()
    def rotate_body_left(self, degrees=10):
        self.body.rotate_left(degrees)
    def rotate_body_right(self, degrees=10):
        self.body.rotate_right(degrees)
    def move_arm_up(self, distance=10):
        self.arm.move_up(distance)
    def move_arm_down(self, distance=10):
        self.arm.move_down(distance)
    def weld(self):
        self.arm.weld()
```




```
>>> robot = IndustrialRobot()
>>> robot.rotate_body_left()
Rotating body 10 degrees to the left...
```

```
class Body:
    def __init__(self):
        self.rotation = 0
    def rotate_left(self, degrees=10):
        self.rotation -= degrees
        print(f"Rotating body {degrees} degrees to the left...")
    def rotate_right(self, degrees=10):
        self.rotation += degrees
        print(f"Rotating body {degrees} degrees to the right...")
```

Example of 1:* aggregation in Python



- If the main purpose of the class is to store data, data classes can be used (from Python 3.7 onwards)
  - Simplified definition of instance attributes (suggested data types), string representation, ...

```
#cards.py
from dataclasses import dataclass
from typing import List

@dataclass
class PlayingCard:
    rank: str
    suit: str
```

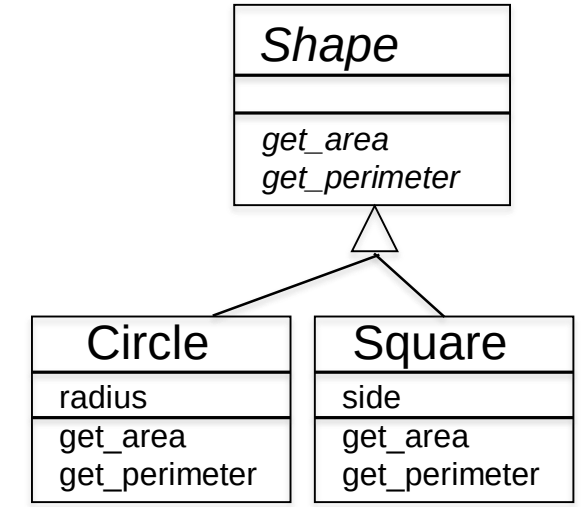
```
@dataclass
class Deck:
    cards: List[PlayingCard]
    def add_card(self,card):
        self.cards.append(card)
```

```
>>> queen_of_hearts = PlayingCard('Q', 'Hearts')
>>> ace_of_spades = PlayingCard('A', 'Spades')
>>> deck = Deck([queen_of_hearts, ace_of_spades])
>>> king_of_diamonds = PlayingCard('K', 'Diamonds')
>>> deck.add_card(king_of_diamonds)
>>> deck
Deck(cards=[PlayingCard(rank='Q', suit='Hearts'),
PlayingCard(rank='A', suit='Spades'),
PlayingCard(rank='K', suit='Diamonds')])
```

2.4. Polymorphism

- It’s the property by which it is possible to send syntactically equal messages to objects of different types
- Even if the message is the same, different objects can respond to it in unique and specific ways
- Combined with inheritance, it allows for more flexible and adaptable code
  - Objects of derived classes can be treated in the same way as if they were direct instances of the base class

Example with a hierarchy of figures



- *get_area* message sent to an object of unknown type
- The correct *get_area* method is executed via dynamic linking

```
# shapes.py
from abc import ABC, abstractmethod
from math import pi
# abstract class: no direct instances
class Shape(ABC):
    @abstractmethod
    def get_area(self):
        pass # abstract method to be redefined in subclass
    @abstractmethod
    def get_perimeter(self):
        pass
    def __str__(self):
       return f"object with area {self.get area()} and perimeter {self.get perimeter()}"
```

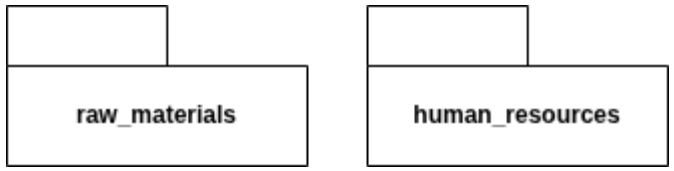
```
class Circle(Shape):
    def __init__(self, radius):
        self.radius = radius
    def get_area(self):
        return pi * self.radius ** 2
    def get_perimeter(self):
        return 2 * pi * self.radius
```

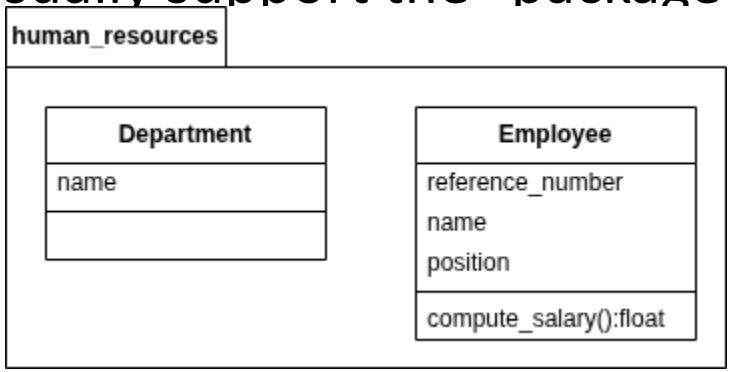
```
>>> shape_a = Circle(100)
>>> print(shape_a) object with area 31415.926535897932 and perimeter 628.3185307179587
```

2.5. Modularity

- It's the property of a system that consists of a set of "cohesive" and loosely "coupled" modules
- Apart from considering classes (objects) as small modules, packages are defined to logically group the classes of a system
  - Cohesive module: it groups together closely related classes
  - Loosely-coupled module: it attempts to minimise the dependencies between a module and the other modules of a system.
  - Package diagrams are included in UML
  - Object-oriented programming languages usually support the "package" concept

package symbol





Module management in Python

- A module is a grouping of related code (classes if we use object orientation) stored in a single .py file.
- A package is a collection of related modules
  - It’s physically stored in a directory structure
- A library is a collection of related packages and modules

- The recommendations seen in the session on "Modularisation and code style“ also apply here

3. Exercises

- Click to open the exercises
- Ex1: Definition of attributes and methods
  - Construction of a class to represent a time of day
- Ex2: Encapsulation
  - Definition of operations in a class to represent fractions
- Ex3: Inheritance
  - Define a class to represent squares in a hierarchy of figures
- Ex4: Aggregations
  - Definition of operations between polynomials expressions

4.References

- Booch G, Rumbaugh J, Jacobson I (1999). The Unified Modeling Language User Guide, 2nd edition. Addison Wesley.
- Object-Oriented Programming (OOP) in Python 3. https://realpython.com/python3-object-oriented-programming/
- Python Classes: The Power of Object-Oriented Programming. https://realpython.com/python-classes/
- Data Classes in Python 3.7+ (Guide). https://realpython.com/python-data-classes/
- The Python Tutorial. https://docs.python.org/3.10/tutorial/index.html