4. Domain Modelling and OOD

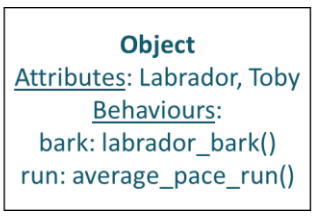
COMP1531 • KC Notes

4.1 Domain Models

- **Domain Model**: a system of **abstractions** to describe and represent the problem domain. It decomposes the domain into **key concepts or objects** and identifies **relationships between objects**
    - Requirements analysis determines **external behaviour**, how the user interacts with the system
        - E.g. in our use case analysis, the system is a black box
    - Domain modelling determines **internal behaviour**, how the system interacts to produce the external behaviour
        - The system becomes transparent, we look at concepts/objects within
- **Benefits of Domain Modelling**
    - High level discussions of what is central to the problem
    - In Agile, **helps understand the scope of an epic**
    - Identifies relationships between sub-domains
    - Ensures system reflects the problem domain
    - There is a common language, so there is a shared understanding among everyone – in agile, refined by the team **to understand impact of epics and features on system**

4.2 Object Oriented Design

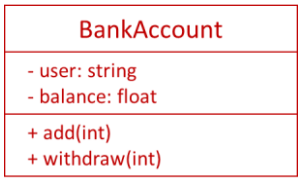
- **Object**: real-world entity, e.g. car, phone, pet, account, time
    - **Attributes**: characteristics and properties of the object (e.g. weight, colour, breed)
    - **Behaviour**: methods, what the object can do (e.g. bark, jump, withdraw, increment)
    - Objects have some **state** and has its own **identity**
    - Methods invoked on the object **is the definition of the public interface**
- **Class**: a **blueprint** of an object that *defines* the attributes and methods
- **Objects are instantiated from a class. A class does not have a state** (breed, name are not defined).



4.3 OOD Properties: Encapsulation

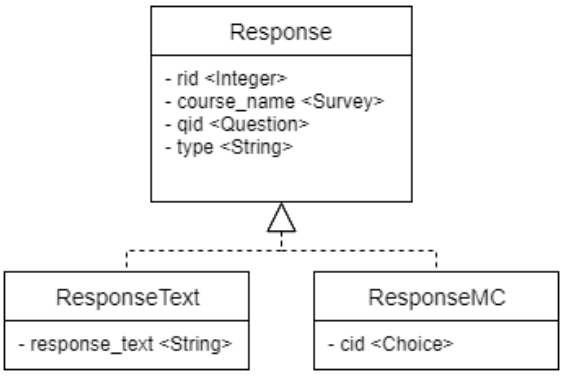
- **Encapsulation**: **hide** object states so that they can only be modified using methods
    - BankAccount's balance **should not be directly modified –** access should be restricted to that variable. Instead, users should be interacting with *add* and *withdraw*
- UML defines private attributes with – and public attributes with +
    - You should generally **not have any public attributes**
- **Benefits:**
    - Object remains in a **consistent state**
        - Set limitations within *withdraw* function, e.g. max withdraw is the balance
        - Modifying *balance* directly could violate certain constraints
    - Encapsulation **increases usability**
        - Only exposing public methods makes the class's role clearer
    - **Abstracts implementation and reduces dependencies**
        - Separates interface and implementation
        - Changing implementation of balance affects only the functions
        - Abstracts away *how* the class works, keeps *what* the class can do

BankAccount **Class**
Attributes:
user
balance
Behaviour:
add(amount)
withdraw(amount)



4.4 OOD Properties: Inheritance

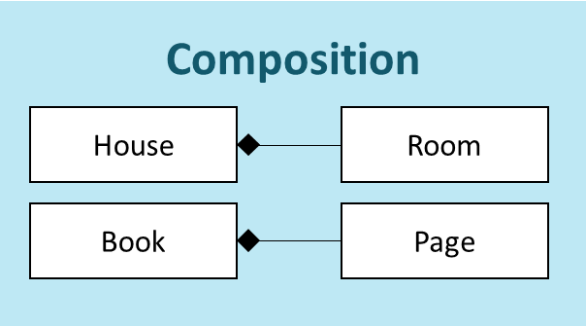
- **Inheritance**: a child class *derives* properties from a parent class
    - "***is-a***" relationship, e.g. a **MCQuestion** *is a* type of **Question**
- UML says a child class ***extends*** properties from the parent class

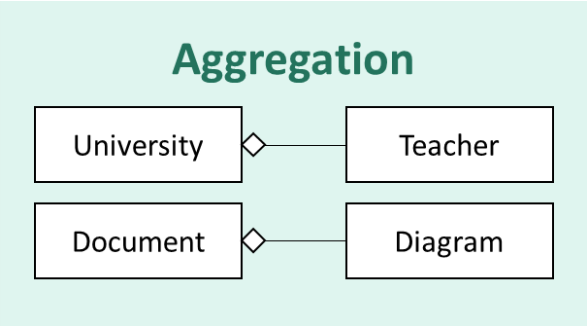


- It is possible for the child class to **override** properties with more specific behaviour
- Children may possibly have **multiple parents** – multiple inheritance
- Other examples:
    - **Car** and **Truck** inherit from the base class **Vehicle**
        - They share common attributes *make*, *model* and common methods *drive*

4.5 OOD Properties: Association

- **Association**: a class *contains* another class
  - "***has-a***" relationship, e.g. a **Survey** *has a* list of **Questions**
- **Composition**: the part/contained item is an **integral part of the containing item**, the item cannot exist without the contained item – will the object "die"?
  - "A room belongs to only one house, there is no independent life of a room. When you delete a house, you delete the room"
  - "When you delete a survey, all responses related to that survey should also be deleted"
  - UML: filled diamond
- **Aggregation**: the **part/contained item can exist on its own**, separate from containing item
  - "A diagram can exist even if we destroy the document"
  - UML: open diamond





4.6 Domain Modelling Techniques

- **Noun and Verb Phrase Identification**: identify nouns in a textual description (e.g. A **customer** goes to an **ATM machine**, *enters* a **PIN** and *withdraws money* from their **account**)
- **CRC Cards**: useful for separating **role** with a set of **responsibilities**
  - **Class**: collection of similar objects
  - **Responsibility**: what the class knows or does
  - **Collaborator**: another class the class interacts with

Admin	
Creates a Survey Has a zID Has a password	Survey

4.7 OOP in Python

```
# Vehicle is a base class
class Vehicle:

    _number_of_vehicles = 0     # underscore means 'private'

    def __init__(self, year, valuation):
        self._year = year
        self._valuation = valuation

    # Getters
    def get_year(self):
        return self._year

    def get_valuation(self):
        return self._valuation

    def get_num_vehicles(self):
        return Vehicle._number_of_vehicles

    # Setters
    def set_year(self, year):
        self._year = year

    def set_valuation(self, valuation):
        # Put constraints in setters
        if valuation < 0:
            raise ValuationTooLow
        else:
            self._valuation = valuation

    # Base classes may have base functions
    def reduce_valuation(self):
        self._valuation = self._valuation * 0.95


# Car inherits properties from Vehicle
class Car(Vehicle):

    def __init__(self, year, valuation, manufacturer):
        self._manufacturer = manufacturer

    # also put getters and setters

# Running the code:
if __name__ == "__main__":
    vehicle = Vehicle(2003, 40000)
    my_car = Car(2004, 50000, "Holden")
    print(vehicle.get_year())
    my_car.reduce_valuation()
```