LABORATORY 8
UML class diagrams

Objectives

The main objective of this laboratory is to introduce UML class diagrams. These diagrams provide a visual representation of a software system's structure, showing the classes, their attributes, operations, and the relationships between them, facilitating understanding and communication among developers and stakeholders.

Theoretical concepts

UML (Unified Modeling Language) is a modeling language that allows the visualization of the design of a system. UML provides two main types of diagrams: *structural diagrams* – such as class diagrams, component diagrams, deployment diagrams, etc. – that describe the way in which the system is built, and *behavioral diagrams* – such as use case diagrams, sequence diagrams, activity diagrams, etc. – that describe the way in which the components of the system interact.

This laboratory will focus on class diagrams. A class diagram describes the structure of a system by showing the classes, their members (attributes and operations), as well as the relationships among objects.

In a class diagram, classes are represented with rectangles composed of three parts:

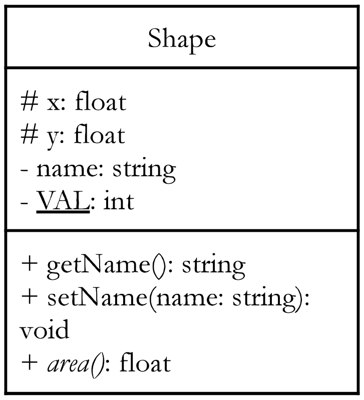
1. the upper part contains the name of the class;
2. the middle part contains the attributes of the class;
3. the bottom part contains the methods of the class.

For the class members (attributes and methods) the access modifier must be specified as follows: + (public), # (protected), – (private) and ~ (packet – in other languages such as Java).

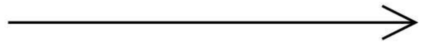
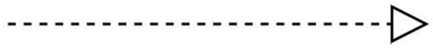
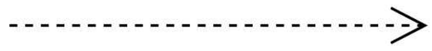
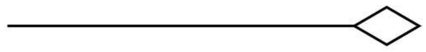
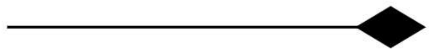
Static members are <u>underlined</u>. Abstract methods are represented in *italics*.

A derived property is shown with its name preceded by a forward slash "/".

The diagram below represents a Shape class with two protected attributes *x* and *y* (floating point values), a private attribute name (*string* value), public getters and setters for the name variable, a public abstract method area, and a private static variable VAL (integer value).



A class diagram also depicts the relationship between classes and how they interact. There are 6 main types of relationships that can be represented in a UML diagram as depicted in the table below:

Relationship name	**Representation**
Association	
Realization/Implementation	
Dependency	
Aggregation	
Composition	

Relationships in UML class diagrams.

For class relationship the multiplicity of the relationship (i.e. the number of objects that participate in the association from the perspective of the other end) can be represented, as illustrated in the table below:

Multiplicity	**Meaning**
0..1	No instance or exactly one instance
1 or 1..1	Exactly one instance
m..n	*m* to *n* instances
* or 0..*	Zero or more instances
1..*	One or more instances

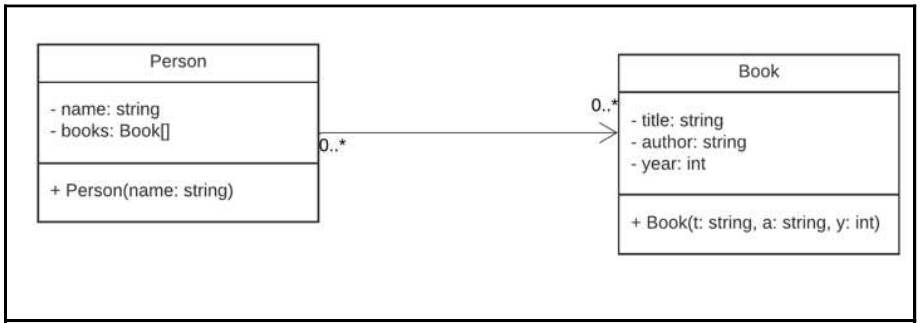
The remainder of this laboratory work illustrates, with code examples, the six common relationships in UML class diagrams.

Association

Association represents the ability of one object to send a message to another object instance to execute an action for it.

This relationship is represented with a solid line. Associations can be unidirectional or bidirectional. The first example depicts unidirectional association: a *Person* stores an array of *Book*s that the *Person* owns.

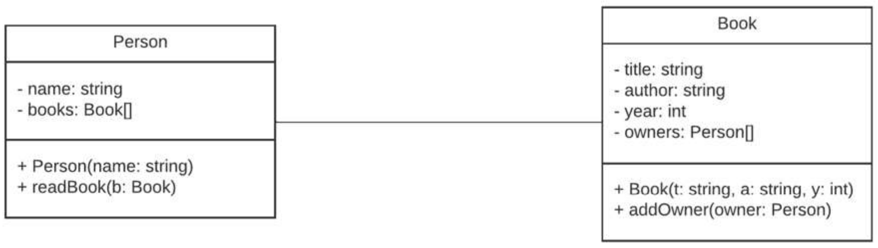
The multiplicity of the relationship (0..*'s in the figure) indicates that a book might be owned by any number of people and that a person can own any number of books.



```
#define MAX_BOOKS 1000
class Book{
public:
  Book(string t, string a, int y);
private:
   string title;
   string author;
   int year;
};

class Person{
public:
   Person(string name);
private:
   string name;
   Book books[MAX_BOOKS];
};
```

The second example illustrates a bidirectional association in which *Person* and *Book* objects store each other in class attributes. As opposed to the previous example, in addition to a *Person* object knowing all the books that the person owns, a *Book* object also stores all the people that own it.

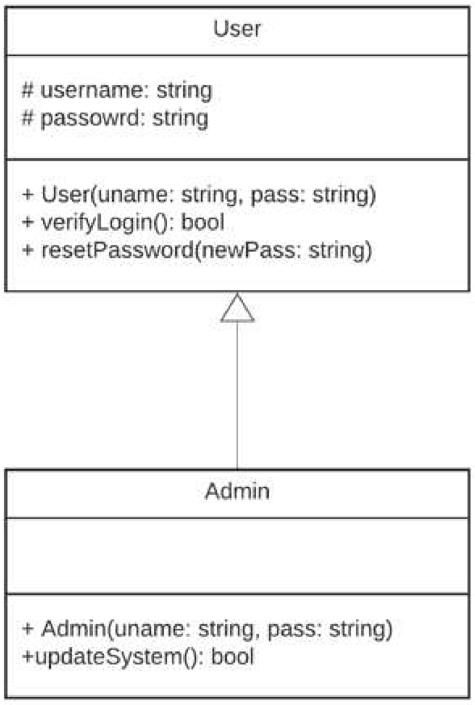


```
class Person;
#define MAX_BOOKS 1000
class Book{
public:
  Book(string t, string a, int y);
private:
   string title;
   string author;
   int year;
   Person* owners[10];
};

class Person{
public:
   Person(string name);
private:
   string name;
   Book books[MAX_BOOKS];
};
```

Inheritance

Inheritance models an "IS-A" relationship between a base (parent) class and a derived (child) class; the derived class inherits all the members of the base class. Inheritance is represented as a solid line with an empty triangle pointing from the derived class to the base class.
In the example below, the base class is a *User* class with two protected attributes (username and password) and two methods. The derived class *Admin* inherits all these members from the *User* class, but an administrator can also update the system (via the method bool *updateSystem*()). In the code, the inheritance relationship is achieved by `class Admin: public User.`

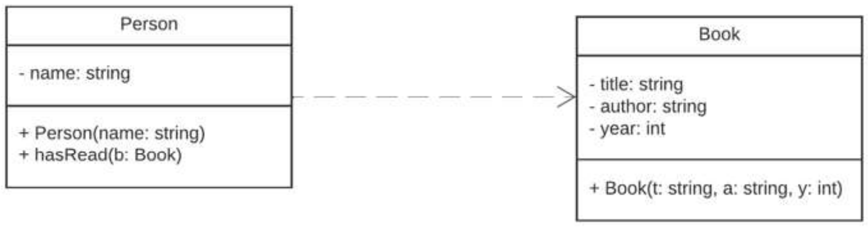

User
# username: string
# passowrd: string
+ User(uname: string, pass: string)
+ verifyLogin(): bool
+ resetPassword(newPass: string)
Admin
+ Admin(uname: string, pass: string)
+updateSystem(): bool

```
class User{
public:
    User(string uname, string pass);
    bool verifyLogin();
    void resetPassword(string newPass);
protected:
    string username;
    string password;
};

class Admin: public User{
public:
    Admin(string uname, string pass);
    bool updateSystem();
};
```

Dependency

Dependency refers to the relationship between two classes in which one element, the client, uses or depends on another element, the supplier. Dependency is represented with a dashed line with an arrowhead at the supplier side.
This relationship occurs when in the code the supplier object is not stored as a class attribute of the client, but it is passed as a parameter to a method of the client or the client creates a local supplier object (inside a method). In the example below, the *Person* class (the client) implements a *hasRead* method with a *Book* parameter (the supplier) that returns whether the person has read the book.



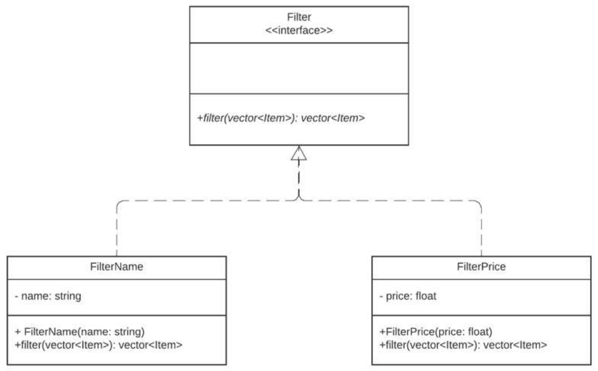
```
class Book{
public:
  Book(string t, string a, int y);
private:
    string title;
    string author;
    int year;
};

class Person{
public:
     Person(string name);
    bool hasRead(Book b);
private:
    string name;
};
```

Realization

Realization is used when a class implements an interface. However, C++ does not explicitly define an interface type; instead, in C++ an interface is a class with no state and only pure virtual methods. Inheritance is represented as a dashed line with an empty triangle pointing from the derived class to the base class.

In the example below, the *Filter* interface establishes the protocol for filtering a vector of Items. Then, two sub-classes are defined: *FilterName* and *FilterPrice* which override the pure virtual method filter to filter the vector based on name and price, respectively.


Filter
<<interface>>
+filter(vector<Item>): vector<Item>
FilterName
- name: string
+ FilterName(name: string)
+filter(vector<Item>): vector<Item>
FilterPrice
- price: float
+FilterPrice(price: float)
+filter(vector<Item>): vector<Item>

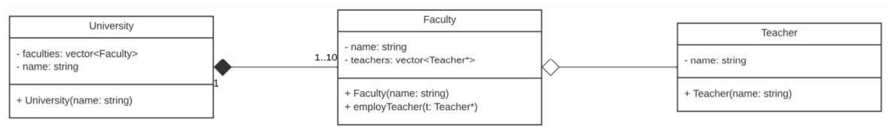
```
// class Item defined elsewhere
class Filter{
public:
    virtual vector<Item> filter(const vector<Item> &items) =
0;
};

class FilterName: public Filter {
public:
    vector<Item> filter(const vector<Item> &items) override;
private:
    string name;
};

class FilterPrice: public Filter {
public:
    vector<Item> filter(const vector<Item> &items) override;
private:
    float price;
};
```

<u>Aggregation and Composition</u>

The aggregation and composition are both used to model part-whole relationships ("has a"/"owns a"). But, there is a semantic difference between these two. As opposed to aggregation, composition implies strong ownership, and the parts cannot exist without the whole. In the composition relationship, the contained objects are destroyed when the owning object is destroyed.
The aggregation relationship is represented as a solid line with an unfilled diamond on the side of the aggregate (the whole), while the composition relationship is represented as a solid line with a filled diamond on the side of the composite (the whole).



```
class Teacher{
public:
    Teacher(string name);
private:
    string name;

};
```

```
class Faculty{
public:
    Faculty(string name);
    void employTeacher(Teacher t);
private:
    string name;
    vector<Teacher> teacher;
};

class University{
public:
    University(string name){
        this->name = name;
        faculties.push_back(Faculty{"Informatics"});
        faculties.push_back(Faculty{"History"});
        faculties.push_back(Faculty{"Chemistry"});
        faculties.push_back(Faculty{"Physics"});
    }

private:
    string name;
    vector<Faculty> faculties;
};
```

In the code example, a university owns/has various faculties (Informatics, History, Physics, etc.), and each faculty has several professors. If the university closes, the departments will no longer exist; therefore we have a composition relationship between the *University* and the *Faculty* class.
However, if the university/faculty closes, the professors will continue to exist. Therefore, the faculties have an aggregation of professors. In addition, a professor could work in more than one faculty, but a faculty cannot be part of more than one university.

Proposed problems

1. Draw the class diagrams for Problem 1 from Laboratory 7.
2. Draw the class diagrams for Problem 2 from Laboratory 7.
3. Draw the class diagrams for Problem 3 from Laboratory 7.
4. Draw the class diagrams for Problem 4 from Laboratory 7.