Tutorial - UML Introduce 1

> Designed by ZHU Yueming
>
> Reference
>
> 1. Bernd Bruegge and Allen H.Dutoit, Object Oriented Software Engineering Using UML, Patterns, and Java Third Edition
> 2. Zhang Yuqun, Slides of Object Oriented Analyze and Design
> 3. Wang Wenmin, Slides of Object Oriented Analyze and Design

Experimental Objective

1. Learn how to design use case diagram according to a paragraph of requirement description
2. Understand different relationship between use cases.
3. Learn how to design class diagram and the relationship between different class diagrams.
4. You need to understand how to write down the corresponding code to describe class diagrams and how to design class diagrams to describe the structure of your code.

General Description

Software engineers often have to communicate properties of the systems they develop with other stakeholders who may or may not be familiar with the technical details of the system that is being examined. UML provides a unified, consistent way to communicate information about software systems in a way designed to be intuitive for both technical and nontechnical individuals. Many tools exist that support working with UML that serve to support the development of a UML model and expressing that model by generating graphical artifacts as specified by the UML standard.

In this laboratory, you can use any plot tools to accomplish exercise. We recommend ProcessON or Xunjie as cartographic software. In this semester, we'll take two weeks to develop two most common UML diagrams:

- User case diagram
- Class model diagram

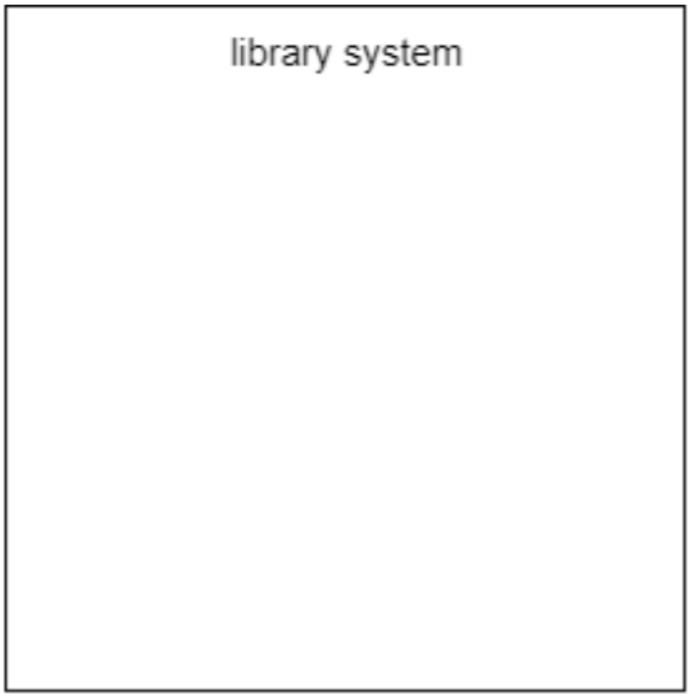
Topic 1. Use case diagram

1. General Introduction

Actor: An actor in a use case is an **external** agent that uses or interacts with the system. An actor can be a user or a role, such as a person or an external system characteristic of the environmental, such as time or temperature change.

Use case: A use case is a collection of related scenarios, including normal and alternative scenarios. The scenarios can be regarded as a behaviorally related sequence of steps, automated or manual, for the purpose of completing a business task.

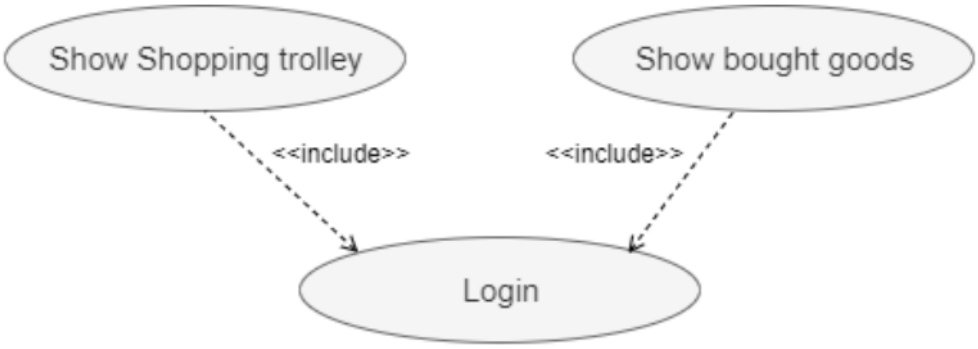
System boundary: System boundary refers to the boundary between different systems, and we can use it to distinguish elements within the system from the elements outside the system. Use case always serve as the function in the system boundary while actors refer to the external element that interact with system.



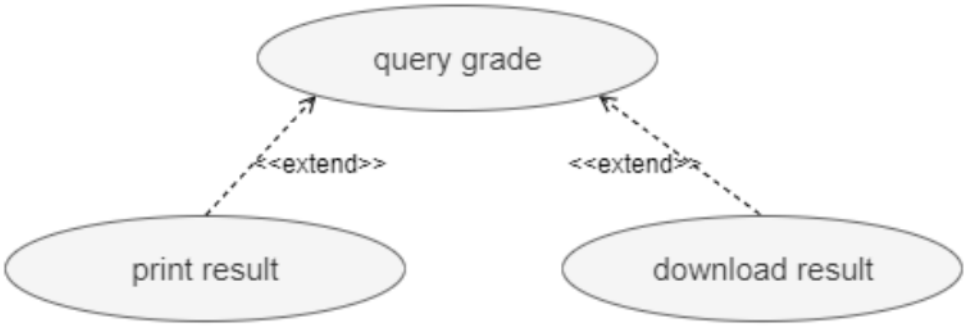
Relationships of Use case diagram:

When describing a complex system, its use case model can become quite complex and can contain redundancy. We use these three types of relationships including inclusion, extension, and inheritance to reduce the complexity of the model.

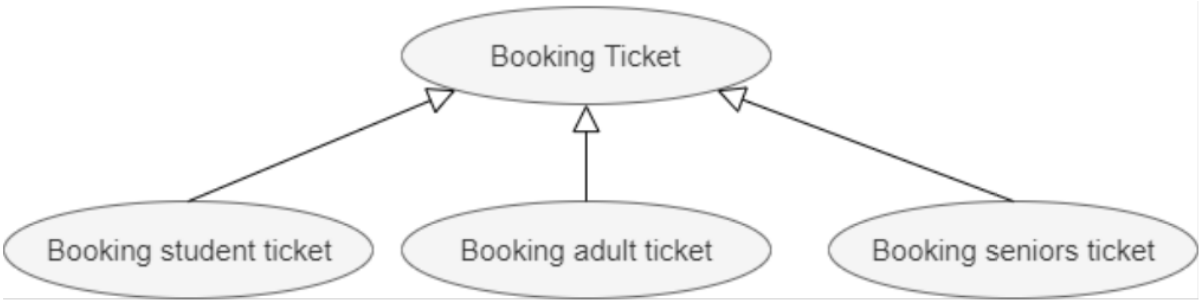
- **Include** (包含): Include is a directed relationship between two use cases, implying that the behavior of the included use case is inserted into the behavior of the including use case.



- **Extend** (扩展): This relationship specifies that the behavior of a use case may be extended by the behavior of another use case.



- **Generalization** (泛化): A generalization is a taxonomic relationship between a more general classifier and a more specific classifier. Each instance of the specific classifier is also an indirect instance of the general classifier. Thus, the specific classifier inherits the features of the more general classifier.



2. Basic Exercises

- **Exercise 1:**

  Translate the following English statements into a representative use case diagram: Library System: The system shall have the following actors: `Borrower` and `Librarian`. `Borrower` shall be able to borrow a book, and he/she can also return or extend the loan of book, if there is no book to return or to extend the loan, the system play nothing. The `Borrower` can also browse the book. A librarian shall be able to update the book catalog.

- **Exercise 2:**

  Translate the following English statements into a representative use case diagram. In this diagram you should add the use case relationship of inclusion and extension.

Fried System:

- When a fire occurs, the `supervisor` needs to report the disaster, and then the system will automatically send the disaster to `dispatcher`.
- `Dispatcher` can search the fire resource according to a specific location in this system, for example: how many fire extinguishers or fire trucks near to the specific (fire) location.And in this step, the `dispatcher` may choice to open the map to see the specific location of the fire and the remaining fire-fighting resources around the location.
- `Dispatcher` can allocate the fire-fighting resources in this system. At this point, after `dispatcher` submit the allocate report, the system will calculate whether to meet the needs of `dispatchers`. When the fire resources in the inventory information database meet the needs of the dispatcher, the dispatcher can allocate the corresponding fire resources including allocating fire extinguishers and allocating fire trucks.

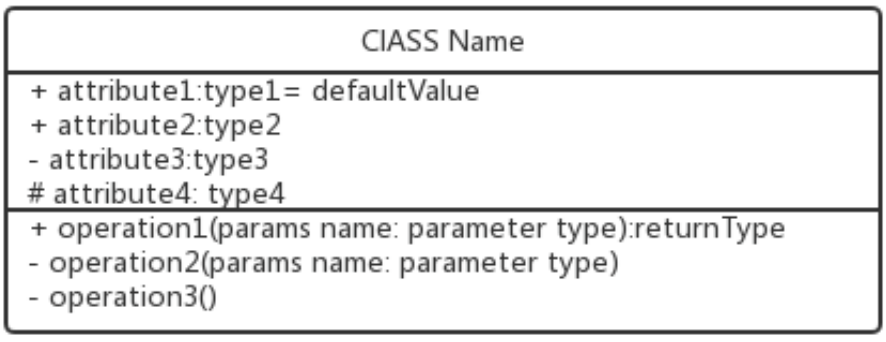
There are two actors in this use case diagram: `supervisor` and `dispatcher`. You need to find the use cases and the relationship of these use cases. Please begin to finish this use case diagram.

Topic 2. Class Diagram

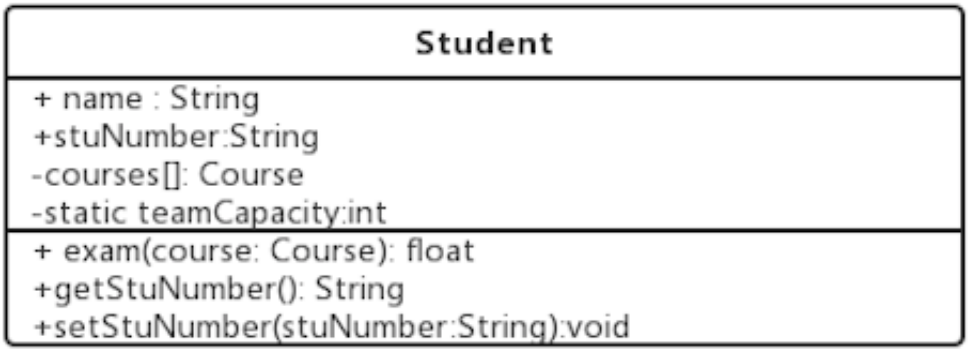
1. General Introduction

Class diagrams provide a way to document the structure of a system by defining what classes there are within it and how they are. In UML, classes and objects are depicted by boxes composed of three compartments. The top compartment displays the name of the class or object. The center compartment displays its attributes, and the bottom compartment displays it operations.

This is a model of class diagram.



This is an example of class diagram.



Following java code is a code framework for the class diagram above

```
public class Student {
        public String name;
        public String stuNumber;
        private Course courses[];
        private static int teamCapacity;

        public float exam(Course course){
            return 0;
        }
        public String getStuNumber(){
            return null;
        }
        public void setStuNumber(String stuNumber){
        }
  }
```

2. Relationship between classes

Classes are interrelated to each other in specific ways. In particular, relationships in class diagrams include different types of logical connections. The followings are such types of logical connections that are possible in UML:

(1) Dependency (依赖)

The UML *dependency* is the weakest relationship of them all. It means that one class uses the object of another class in method, and the object of another class is not declared in field.

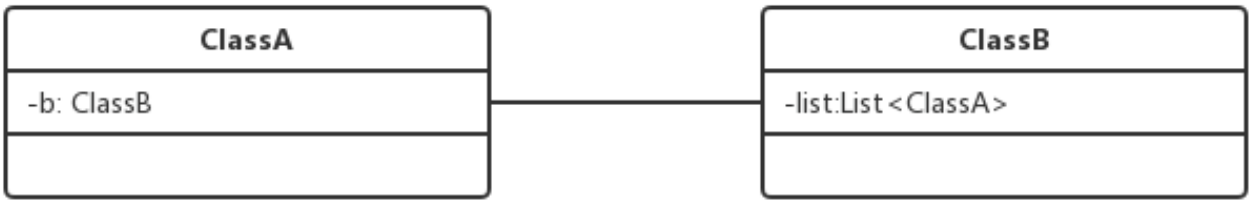
```
public class ClassA {
    public void depend(ClassB classB){}
}
public class ClassB {
}
```

(2) Association(关联)

`classA` has a `ClassB`

Unidirectional Associations(单向关联): The fields in `classA` contain at least one instances of `classB`, but the fields in `classB` does not contain any instances of `classA`.

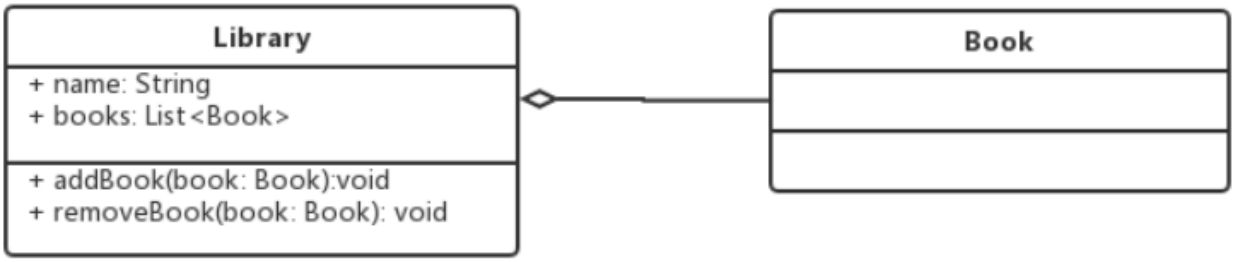
Bidirectional Associations(双向关联): The fields in both `classA` and `classB` contain at least one instances of each other's.



(3) Aggregation(聚合)

It is a specific case of **Association** (unidirectional association) that has following features:

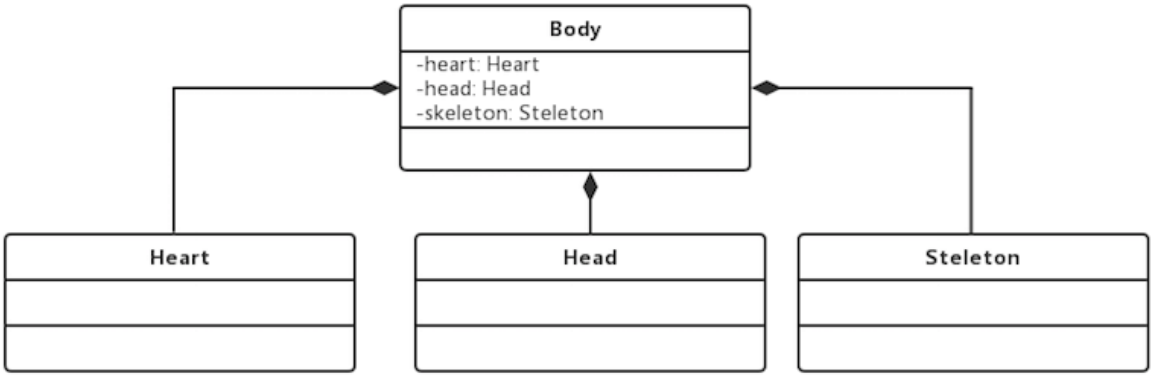
- The fields in `classA` contain at least one instances of `classB`
- The Relationship between a whole and a part. Partial objects can be created before the whole object is created, or they can be destroyed after the whole object is destroyed.
- Often describes "owns a" relationship. `ClassA` owns a `ClassB`, Library owns a Book etc.
- `ClassB` can exist dependency of `ClassA` which means when destroy `ClassA` the instance of `ClassB` still exist.



(4) Composition(组合)

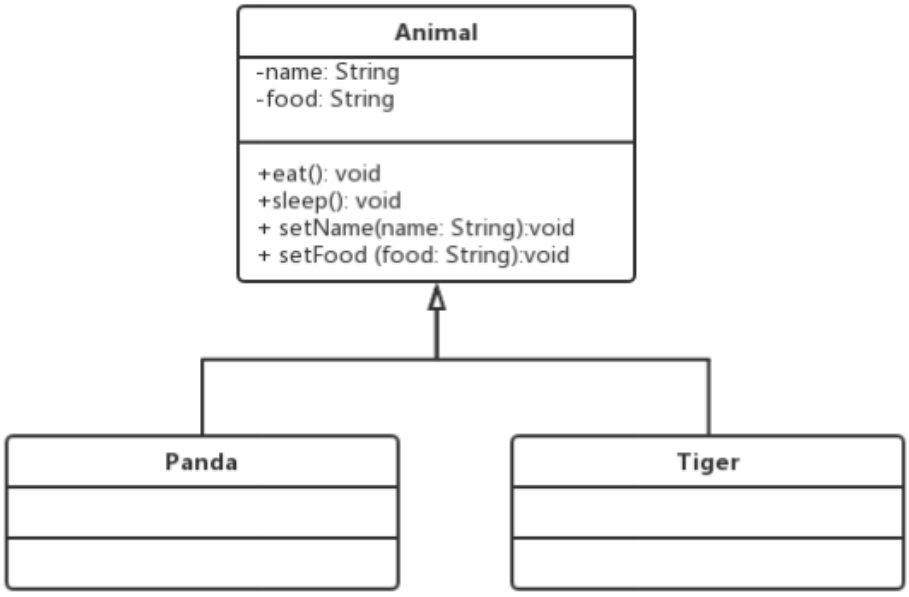
It is also a kind of **Association** but it describes a stronger dependency between two classes. Composition always has following features:

- The fields in `classA` contain at least one instances of `classB`.
- The Relationship between a whole and a part. The end of the whole life cycle also means the end of the partial life cycle. For example: `ClassB` cannot exist dependency of `ClassA` which means when destroy `classA` the instance of `classB` in `classA` would not exist anymore. For example, a heart can not exist without a body.
- Often represent a "is made up of" relationship. `ClassB` is made up of `ClassA`
- It often describe the inner class.



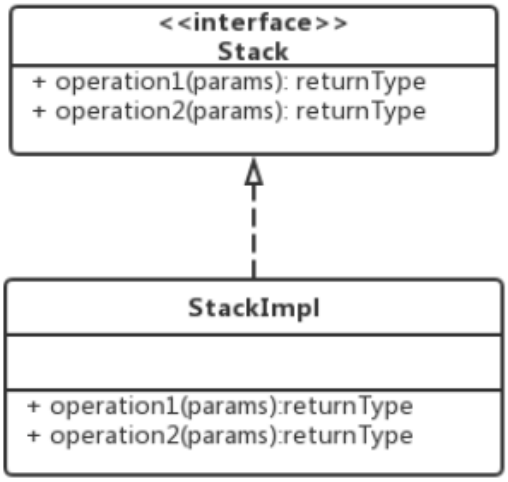
(5) Inheritance and Generalization(继承)

The inheritance relationship between classes.



(6) Realization(实现)

`classB` implements `classA`, so that `classA` always be an interface.



3. Basic Exercise

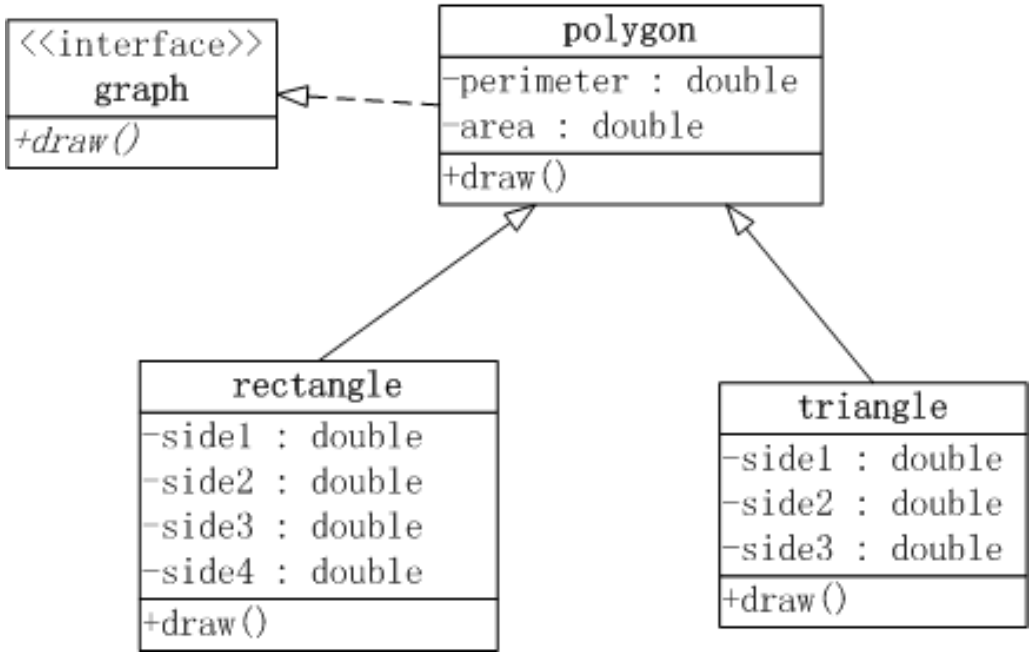
- **Exercise 1**

  According to the code framework below, please design the corresponding class diagram:

```
public class School {
  private List<Department>departments;

  public void SetUpDepart(){
    departments.add(new Department());
  }
}

public class Department{
  private Teacher[] teacher;
  public void displayTeacherName(){
  }
}

public class Teacher{
  private School school;
  private Department deparment;

  public void setSchool(School school){
    this.school=school;
  }
  public void setDepartment(Department department){
    this.deparment=department;
  }
}
```

- **Exercise 2**

  According to the class diagram below, please design the corresponding code framework.



4. ECB pattern

From the perspective of the functional differences, classes often divided into three different categories.

- **Entity class**:

  Entity class represents entities used to encapsulate data, transfer instance. (e. g. Customer, Record)

- **Control class**:

  Control class is the medium of interaction between entity class and boundary class, which is often used to describe a series of logical processes.

- **Boundary class**

  Boundary class represents the boundary of interaction with system actors. (e. g. Page)

For more detailed introduction: click here