NCTU ECE

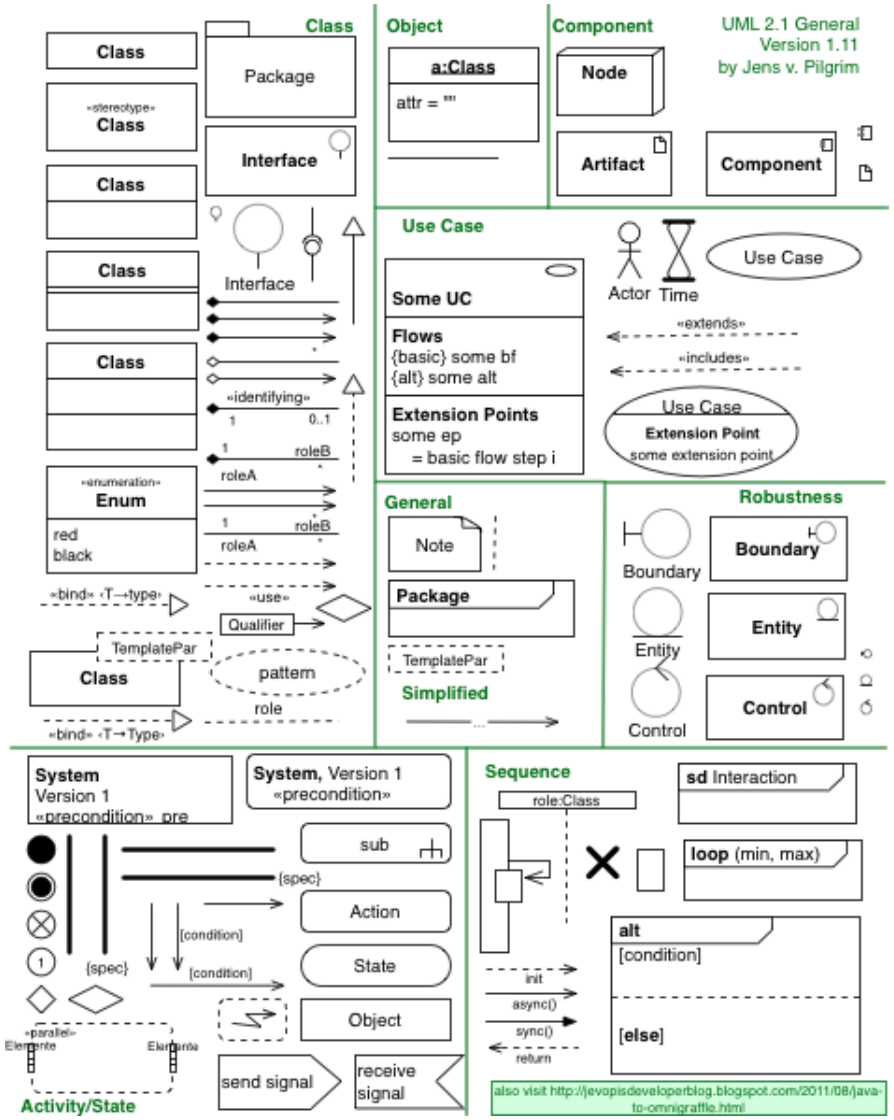
# Introduction to UML Class Diagrams

## Advanced Object Oriented Programming

Alphar Juan

2019. 11. 13/14

## Introduction to UML




# WHAT IS UML ?

UML (Unified Modeling Language) is a standard language for specifying, visualizing, constructing, and documenting the artifacts of software systems. UML was created by the Object Management Group (OMG) and UML 1.0 specification draft was proposed to the OMG in January 1997. It was initially started to capture the behavior of complex software and non-software system and now it has become an OMG standard.

### OO system Pictures

Programming languages are not abstract enough for OO (Object-Oriented) design. UML is a pictorial language used to make software blueprints.

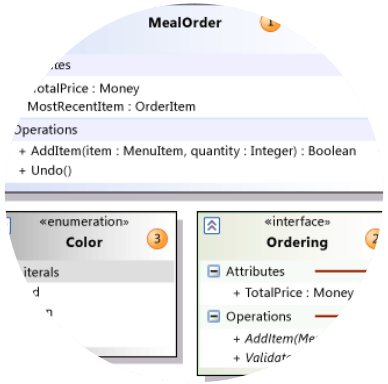
### Open Standard

Lots of companies use it. Very big, but a nice standard that has been embraced by the industry.

### No Limitation

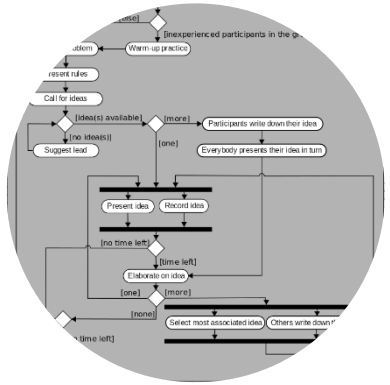
It is also used to model non-software systems as well. For example, the process flow in a manufacturing unit, etc.

# WHAT CAN YOU MODEL WITH UML 2.0?



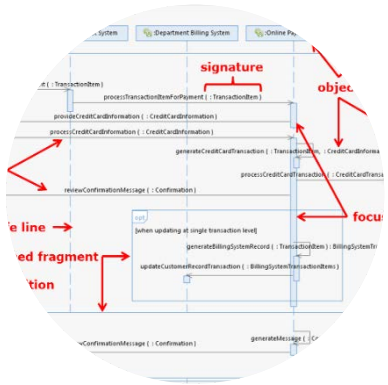

## Structure Diagrams

Class Diagram, Object Diagram, Component Diagram, Composite Structure Diagram, Package Diagram, and Deployment Diagram.




## Behavior Diagrams

Use Case Diagram (used by some methodologies during requirements gathering); Activity Diagram, and State Machine Diagram.




## Interaction Diagrams

All derived from the more general Behavior Diagram, include the Sequence Diagram, Communication Diagram, Timing Diagram, and Interaction Overview Diagram.

## As a sketch

**to communicate aspects of system**

- forward design: doing UML before coding
- backward design: doing UML after coding as documentation
- often done on whiteboard or paper
- used to get rough selective ideas

## As a blueprint

**a complete design to be implemented**

- sometimes done with CASE (Computer-Aided Software Engineering) tools.

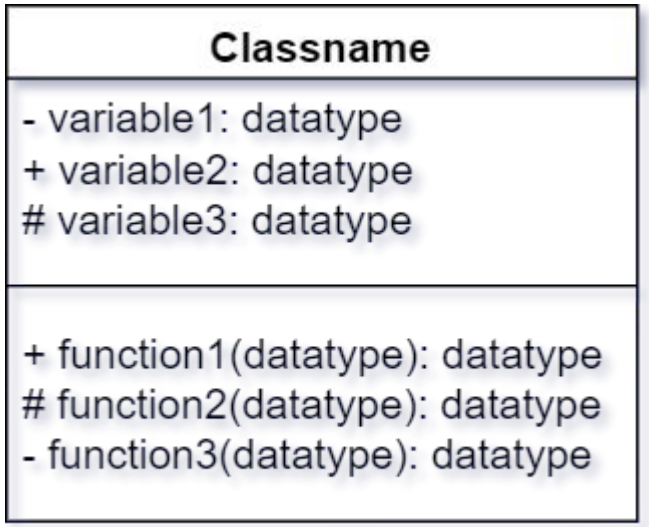
## As a programming language

**with the right tools, code can be auto-generated and executed from UML**

- only good if this is faster than coding in a "real" language.

# CLASS REPRESENTATION

A class representation can be divided into three part, class name, variable with datatype, and functions.




### Different visibility of the class can be represented as

- +: Public
- -: Private
- #: Protected

### Different Parameter direction

- in: An input Parameter (may not be modified).
- inout: An output Parameter (may be modified to communicate information to the caller).
- out: An input Parameter that may be modified.
- return: A return value of a call.

EX: -radius: in double

### Different type of members in a class

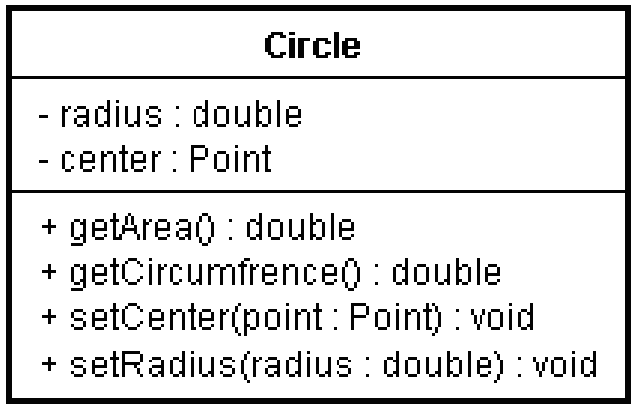
- Static members are represented as underlined.
- Pure virtual functions are represented as italics.

# Class Representation

Example: a class representation

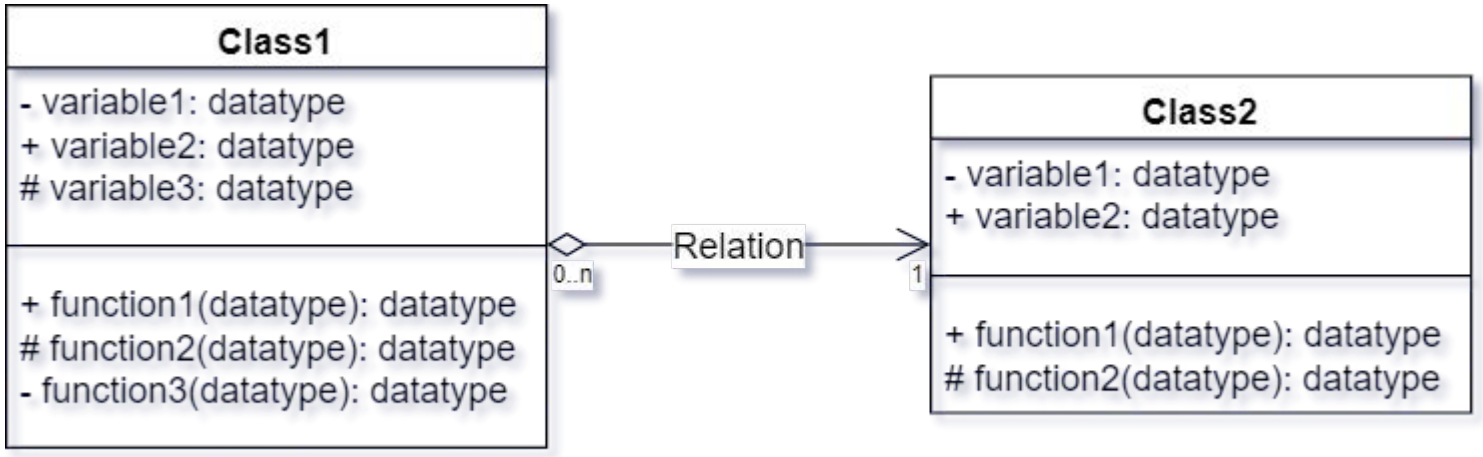
```
class Circle {
    private:
        double radius;
        Point center;

    public:
        void setRadius(double radius);
        void setCenter(Point center);
        double getArea();
        double getCircumfrence();
};
```

Class diagram for the class is shown below.



# CLASS RELATIONSHIP

In a system a class may be related to different classes, following are the different relation ship.

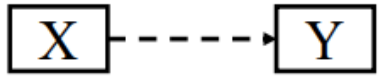


# CLASS RELATIONSHIP

In a system a class may be related to different classes, following are the different relation ship.

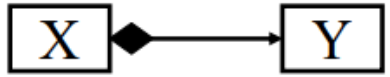

**Association**

knows a



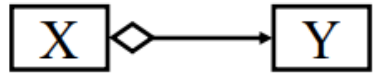

**Dependency**

uses a



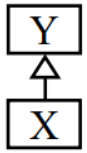

**Composition**

has a (own a)



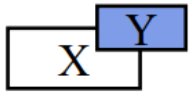

**Aggregation**

has a




**Inheritance**

is a




**Class Template**

parameterized class

# DIFFERENT MULTIPLICITY IN A RELATION

In a system a class may be related to different classes with different Multiplicity in a relation

**0..1**

No instances, or one instance (optional, may)

**1**

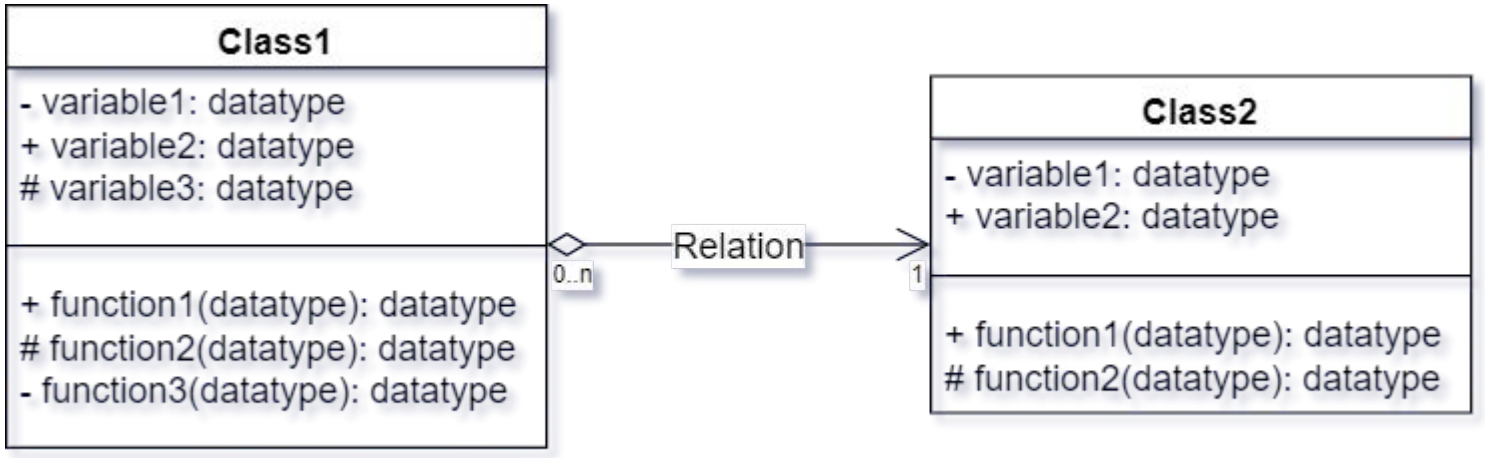
Exactly one instance

**0..* or ***

Zero or more instances

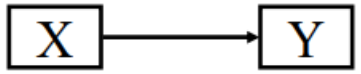
**1..***

One or more instances (at least one)



# ASSOCIATION

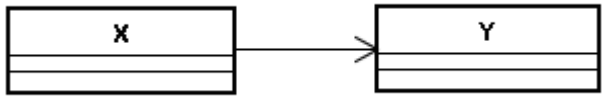
One object is aware of another; it contains a pointer or reference to another object.




**Association**

knows a

**Representation**

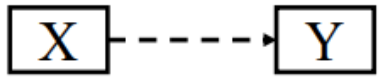



**Example Code**

```
class X {

    X(Y *y) : y_ptr(y) {}

    void SetY(Y *y) { y_ptr = y;   }

    void f()        { y_ptr->Foo();}
    ...
    Y *y_ptr; // pointer
};
```

# DEPENDENCY

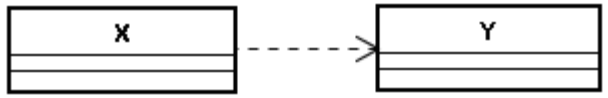
One class depends on another if the independent class is a parameter variable or local variable of a method of the dependent class. (One object issues a function call to a member function of another object.)




**Dependency**

use a

### Representation

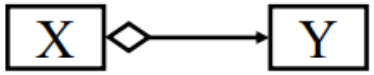



### Example Code

```
class X {
    ...
    void f1(Y y)  {…;  y.Foo();       }
    void f2(Y *y) {…;  y->Foo();      }
    void f3(Y &y) {…;  y.Foo();       }
    void f4()     {   Y y; y.Foo();  ...}
    void f5()     {…; Y::StaticFoo(); }
    ...
};
```

# AGGREGATION

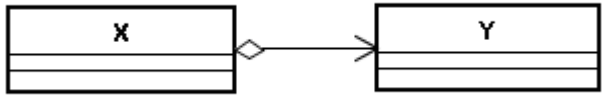
A class contains other classes as members. Aggregation can occur when a class is a collection or container of other classes, but where the contained classes do not have a strong life cycle dependency on the container—essentially, if the container is destroyed, its contents are not. You may have confusion between aggregation and association. Association differs from aggregation only in that it does not imply any containment.



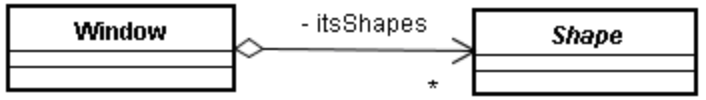

**Aggregation**

has a

## Representation




## A window class contains a list of its shapes

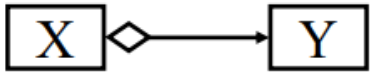



## Example Code 1

```
class Window
{
    public:
    //...
    private:
        vector itsShapes;
};
```

# AGGREGATION

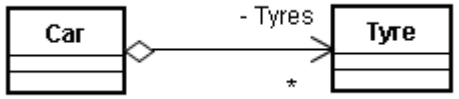
A class contains other classes as members. Aggregation can occur when a class is a collection or container of other classes, but where the contained classes do not have a strong life cycle dependency on the container—essentially, if the container is destroyed, its contents are not. You may have confusion between aggregation and association. Association differs from aggregation only in that it does not imply any containment.




**Aggregation**

has a

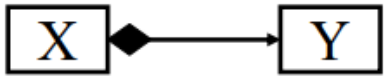
**A car has it’s tyres, and the scope of tyre doesn’t depend on a car since a tyre can be used for another car also.**




**A Rectangle class has its style, which may be shared by other shapes also; life time of style doesn’t depend on Rectangle class.**

# COMPOSITION

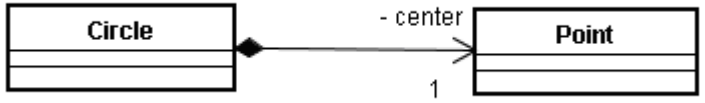
A class contains other classes as members. Composition is the stronger form of aggregation. Composition can occur when a class is a collection or container of other classes, but where the contained classes have a strong life cycle dependency on the container—essentially, if the container is destroyed, its contents are also destroyed.




**Composition**

has a (own a)

## Representation

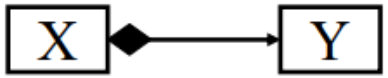



## Example Code 1

```
class Circle
{
    private:
        ...
        Point center;
        ....
};
```

# COMPOSITION

A class contains other classes as members. Composition is the stronger form of aggregation. Composition can occur when a class is a collection or container of other classes, but where the contained classes have a strong life cycle dependency on the container—essentially, if the container is destroyed, its contents are also destroyed.




**Composition**

has a (own a)

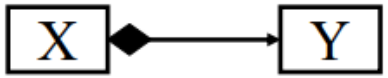
## Representation


## Example Code 2

```
class X {
    ...
    Y a; // 1; Composition
    Y b[10]; // 0..10; Composition
};

class X {
    X() { a = new Y[10]; }
    ~X(){ delete [] a; }
    ...
    Y *a; // 0..10; Composition
};
```

# COMPOSITION

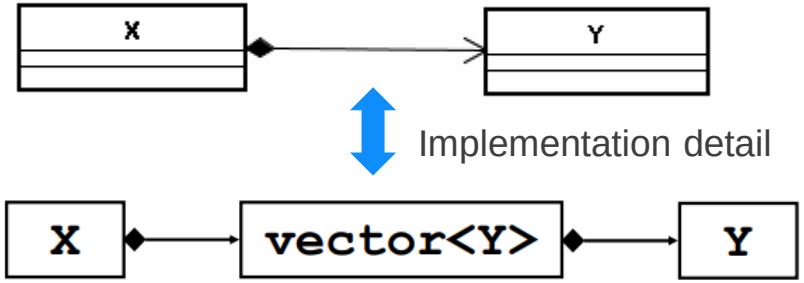
A class contains other classes as members. Composition is the stronger form of aggregation. Composition can occur when a class is a collection or container of other classes, but where the contained classes have a strong life cycle dependency on the container—essentially, if the container is destroyed, its contents are also destroyed.




**Composition**

has a (own a)

## Representation



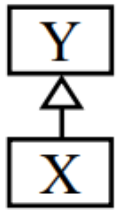

Composition of vector<Y>, not composition of Y

## Example Code 3

```
class X {
    ...
    vector<Y> a; // 0..*; Composition
};
```

# INHERITANCE

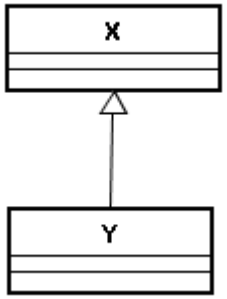
In Inheritance relationship, a class is derived from another class. It is a “is a” relationship between two classes.




**Inheritance**

is a

### Representation



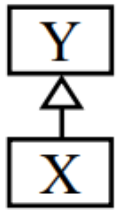

Here X and Y are normal classes.

### Example Code 1

```
class X {
    ...
};

class Y : public X {
    ...
};
```

# INHERITANCE

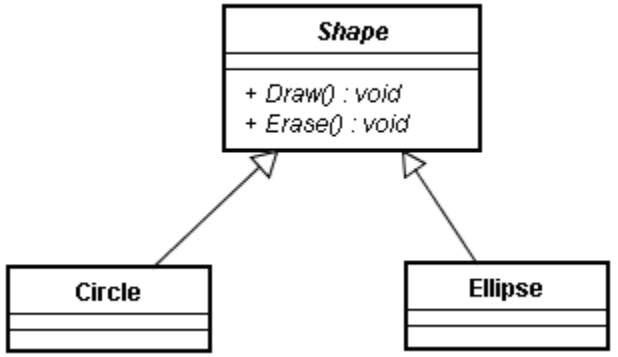
In Inheritance relationship, a class is derived from another class. It is a “is a” relationship between two classes.




**Inheritance**

is a

## Representation



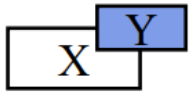

Here Shape is an abstract class that is why it is shown in Italics. Draw() and Erase() methods of Shape class is pure virtual function, so it is also shown as italics.

## Example Code 2

```
class Shape{
    virtual void Draw(){};
    virtual void Erase(){};
    ...
};

class Circle : public Shape{
    virtual void Draw(){};
    virtual void Erase(){};
    ...};

class Ellipse : public Shape{
    virtual void Draw(){};
    virtual void Erase(){};
    ...
};
```

# CLASS TEMPLATE

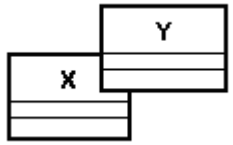
Template class mean generic classes. Languages like C++, java, C# supports generic programming.




**Class Template**

parameterized class

## Representation




## Example Code 1

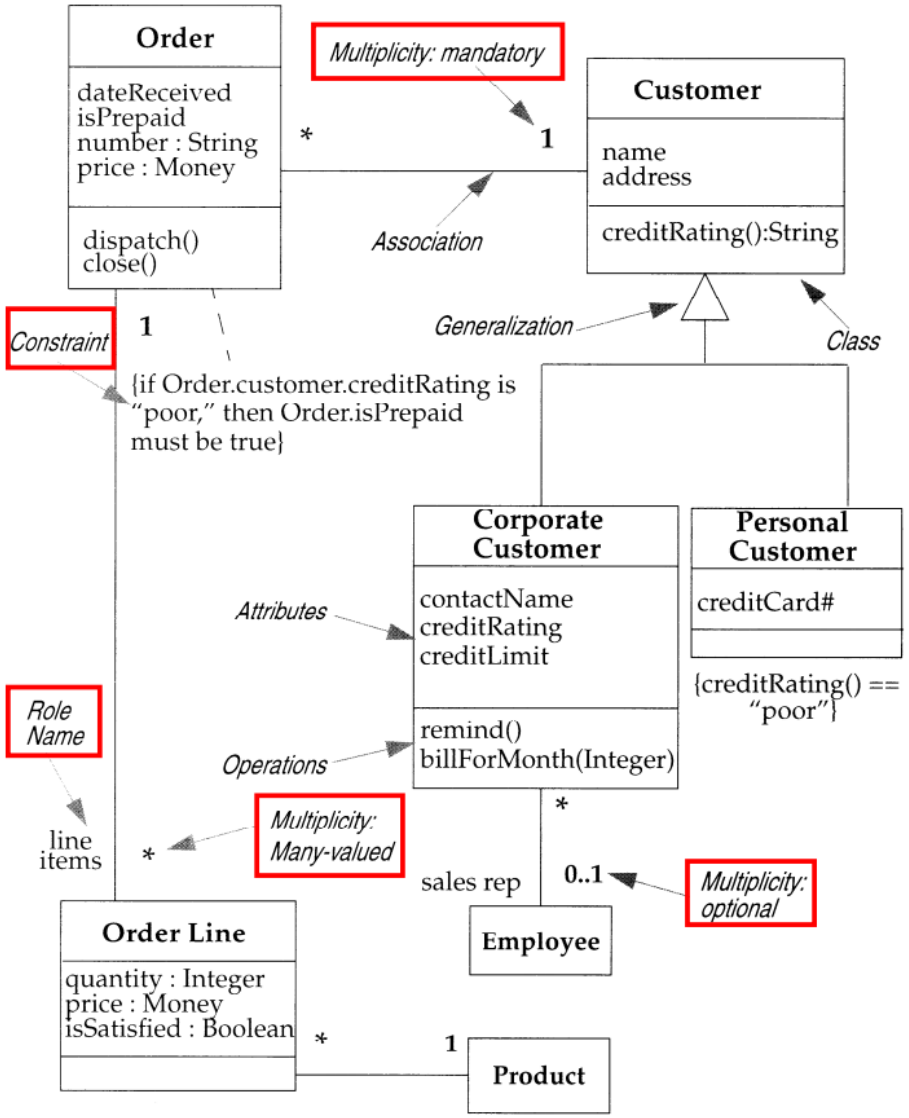
```
template <class Y>
class X {
    ...
    Y& add(const Y &a) {
        ...
    }
};

...
X<Y> a;
...
```

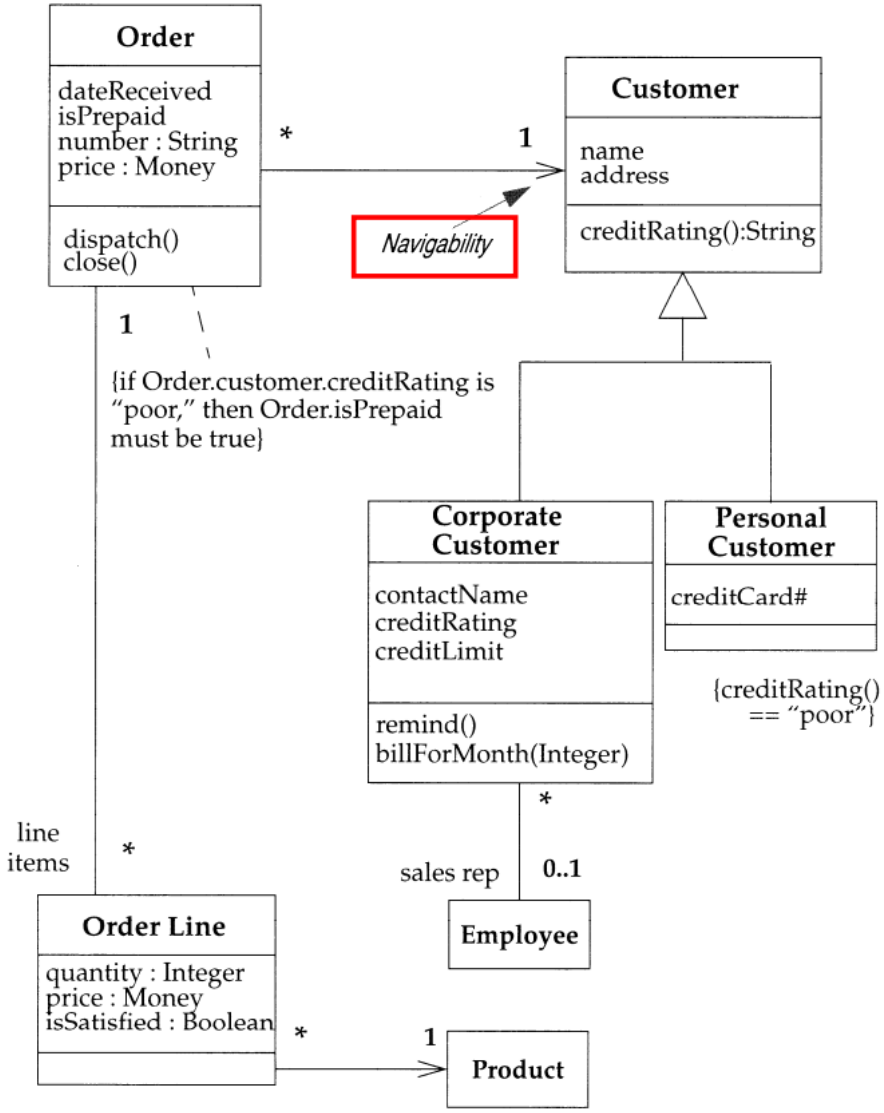
# RELATIONSHIP EXAMPLE

Template class mean generic classes. Languages like C++, java, C# supports generic programming.

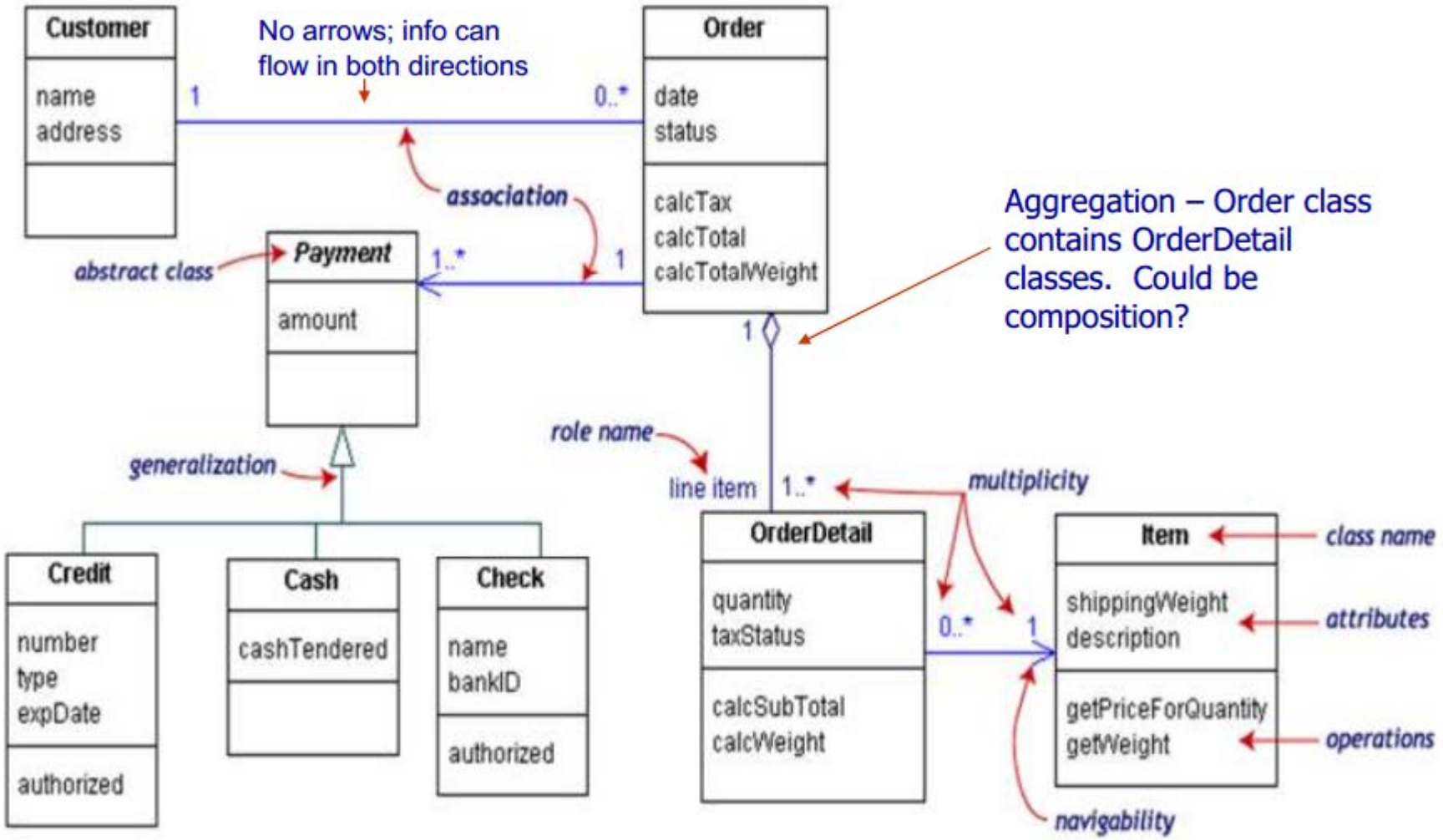
| | | |
|---|---|---|
| Car | Engine ? | Composition |
| Person | Cell Phone ? | Association/Composition/Dependency |
| Human | Brain ? | Composition |
| Fighter | Bomb ? | Association |
| Fighter | F16 ? | Inheritance |
| Bomb | Explosive ? | Composition/Association/Inheritance |
| Bomb | Nuclear Bomb ? | Inheritance |
| Airplane | Boeing 747 ? | Inheritance |
| Airplane | Seat ? | Composition |
| Hero | Life ? | Composition/Attribute |
| Hero | Score ? | Association/Dependency |
| Hero | Map ? | Association/Dependency |

# Case Study 1

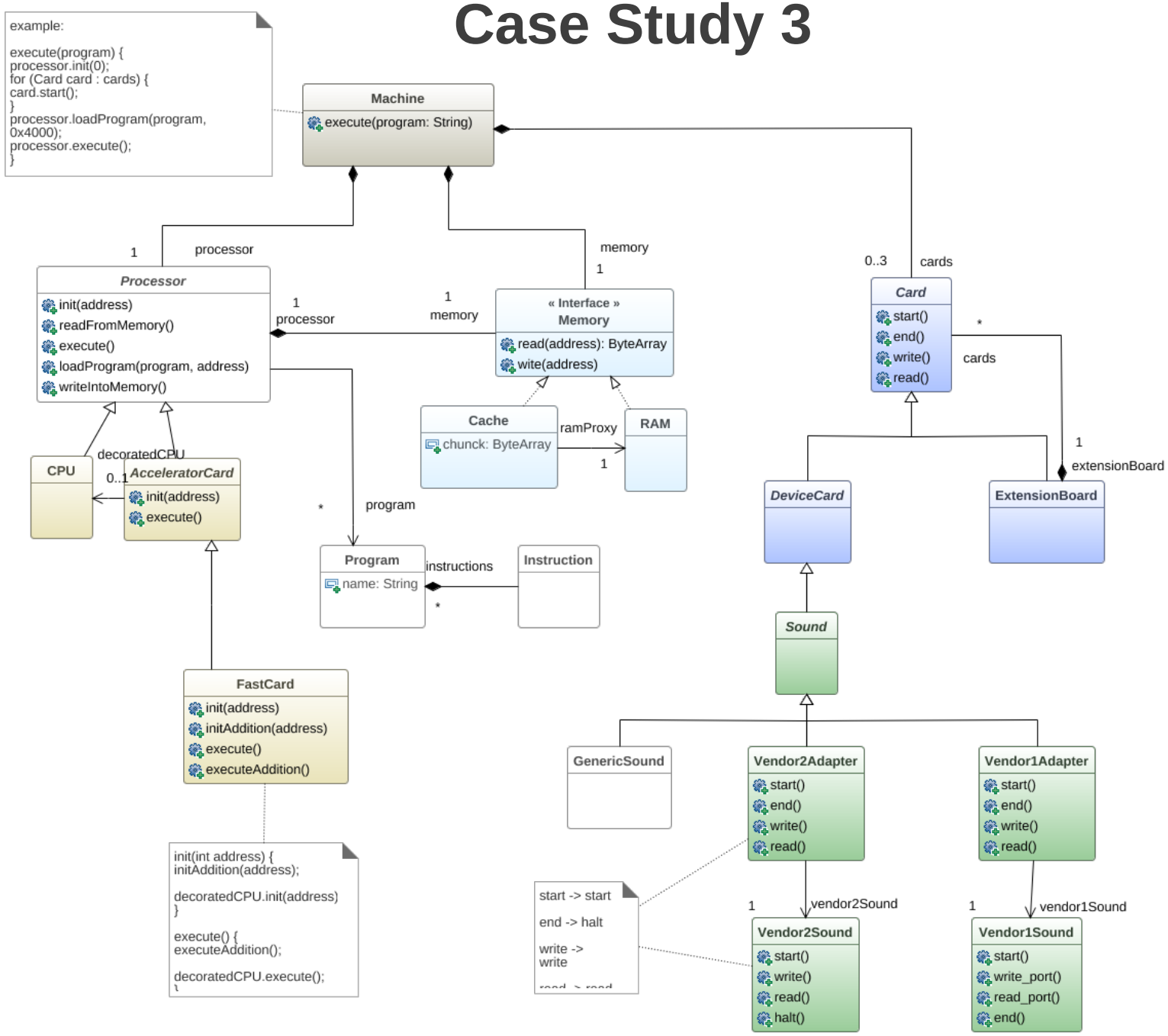

Order
dateReceived
isPrepaid
number : String
price : Money
dispatch()
close()
Multiplicity: mandatory
*
1
Customer
name
address
creditRating():String
Association
Constraint
1
{if Order.customer.creditRating is "poor," then Order.isPrepaid must be true}
Generalization
Class
Corporate Customer
contactName
creditRating
creditLimit
remind()
billForMonth(Integer)
Personal Customer
creditCard#
{creditRating() == "poor"}
Attributes
Role Name
Operations
Multiplicity: Many-valued
line items
*
sales rep
0..1
Multiplicity: optional
Employee
Order Line
quantity : Integer
price : Money
isSatisfied : Boolean
*
1
Product


Order
dateReceived
isPrepaid
number : String
price : Money
dispatch()
close()
*
1
Customer
name
address
creditRating():String
Navigability
1
{if Order.customer.creditRating is "poor," then Order.isPrepaid must be true}
Corporate Customer
contactName
creditRating
creditLimit
remind()
billForMonth(Integer)
Personal Customer
creditCard#
{creditRating() == "poor"}
line items
*
sales rep
0..1
Employee
Order Line
quantity : Integer
price : Money
isSatisfied : Boolean
*
1
Product

# Case Study 2


Customer
name
address
No arrows; info can flow in both directions
1
0..*
Order
date
status
association
calcTax
calcTotal
calcTotalWeight
abstract class
Payment
amount
1..*
1
Aggregation – Order class contains OrderDetail classes. Could be composition?
1
role name
line item
1..*
multiplicity
generalization
Credit
number
type
expDate
authorized
Cash
cashTendered
Check
name
bankID
authorized
OrderDetail
quantity
taxStatus
calcSubTotal
calcWeight
0..*
1
Item
shippingWeight
description
getPriceForQuantity
getWeight
class name
attributes
operations
navigability

# Case Study 3


example:
execute(program) {
processor.init(0);
for (Card card : cards) {
card.start();
}
processor.loadProgram(program, 0x4000);
processor.execute();
}
Machine
execute(program: String)
1
processor
memory
1
0..3
cards
Processor
init(address)
readFromMemory()
execute()
loadProgram(program, address)
writeIntoMemory()
1
processor
1
memory
« Interface »
Memory
read(address): ByteArray
wite(address)
Card
start()
end()
write()
read()
*
cards
Cache
chunck: ByteArray
ramProxy
1
RAM
1
extensionBoard
CPU
decoratedCPU
0..1
AcceleratorCard
init(address)
execute()
*
program
DeviceCard
ExtensionBoard
Program
name: String
instructions
*
Instruction
Sound
FastCard
init(address)
initAddition(address)
execute()
executeAddition()
GenericSound
Vendor2Adapter
start()
end()
write()
read()
Vendor1Adapter
start()
end()
write()
read()
init(int address) {
initAddition(address);
decoratedCPU.init(address)
}
execute() {
executeAddition();
decoratedCPU.execute();
start -> start
end -> halt
write ->
write
1
vendor2Sound
1
vendor1Sound
Vendor2Sound
start()
write()
read()
halt()
Vendor1Sound
start()
write_port()
read_port()
end()

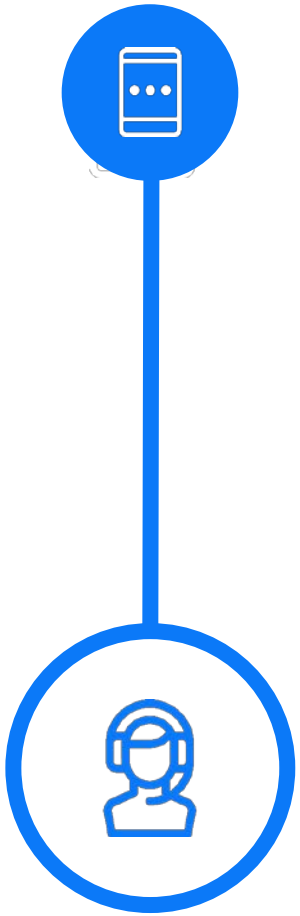
# UML TOOLS

## Microsoft Painter

You can use painter in Windows to draw the class diagrams. Painter is powerful and can make you powerful.

## Microsoft Visio

A diagramming and vector graphics application and is part of the Microsoft Office family. The product was first introduced in 1992, made by the Shapeware Corporation. It was acquired by Microsoft in 2000. You can get in NCTU CA.



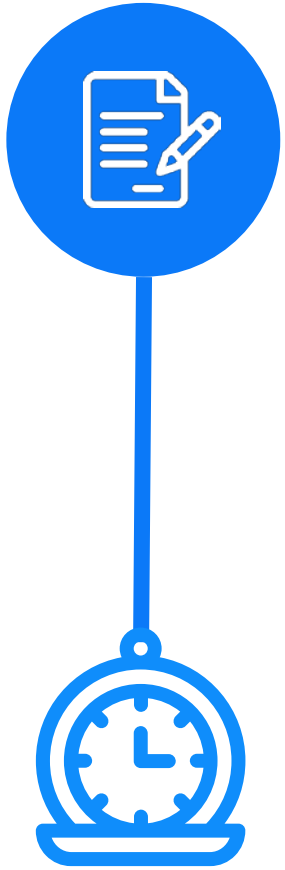
## Draw.io

Is free online diagram software for making flowcharts, process diagrams, org charts, UML, ER and network diagrams.
online version: https://www.draw.io/
offline version: https://github.com/jgraph/drawio-desktop/releases

## Google “UML Tools”

There are many UML tools and you can google by yourself. Choose one which you like.

# HOMEWORK



## Requirement

Please follow the final project specification to design the class hierarchy, and draw the UML class diagrams. Please hand in one page word file, and demo your design in the lab class. If your design is very bad, you will not pass about this homework.

## Deadline

2019.11/27(Wed).11/28(Thu) in lab class.