UML and design pattern

VU MINH HOANG

Outline

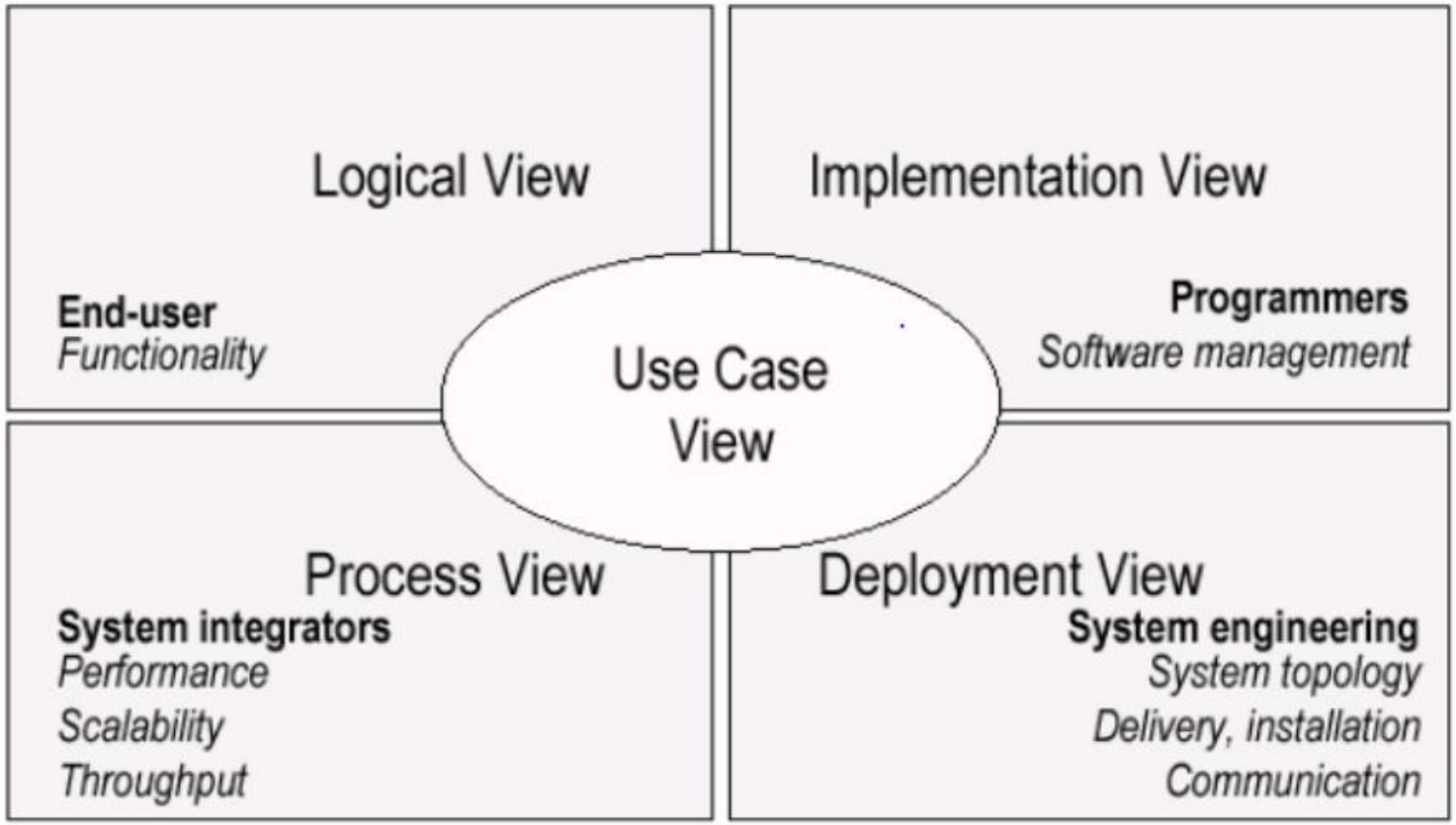
- What is UML
- Some basic diagrams
- Introduction design pattern

1. What is UML

- Unified modeling language
- Express and design software (OOP)
- Independent of implementation language
- From general to detailed design
- Increase understanding/communication of SW between customers and developers

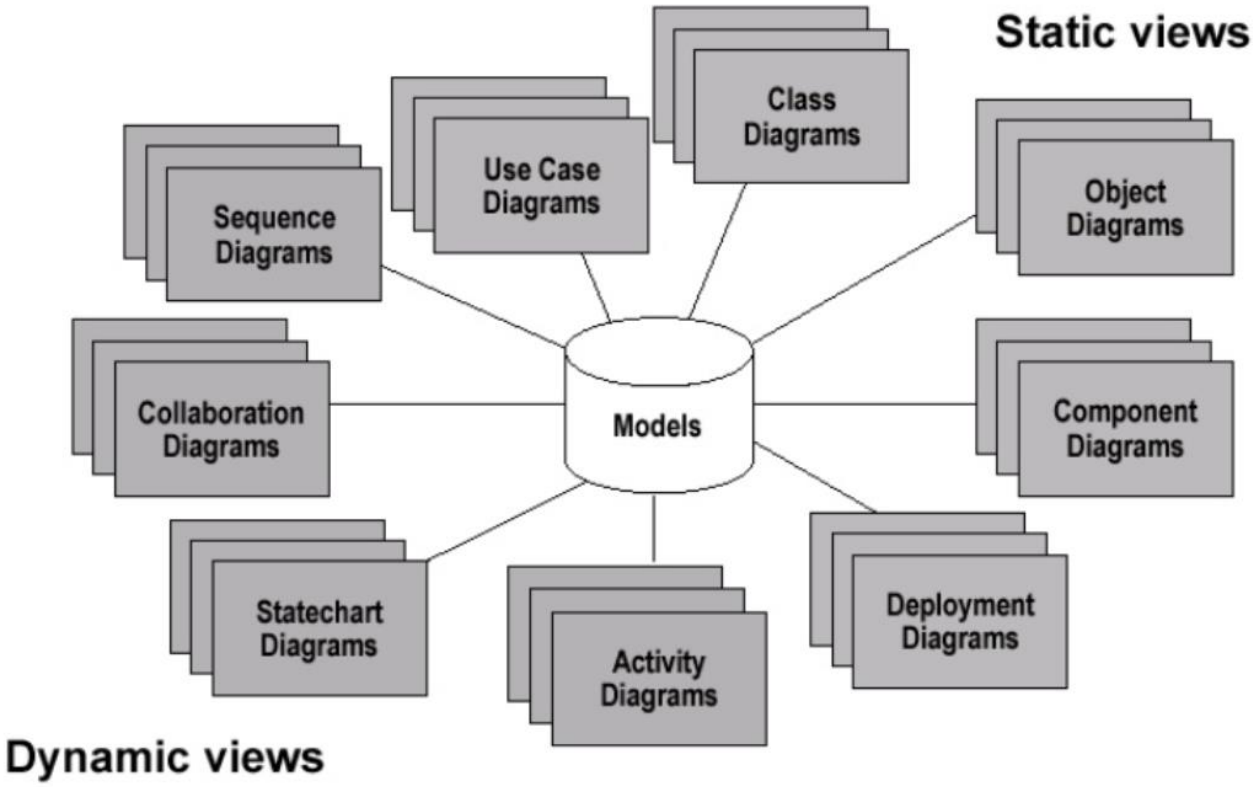
Models and views

- Model is an abstraction describing system
- View depicts selected aspects of a model
- UML is a multi-diagrammatic language



Static vs dynamic design

- Static: describe code structure and object relations, doesn't change
- Dynamic: show interaction between objects, follow sequences of events, change depending scenario

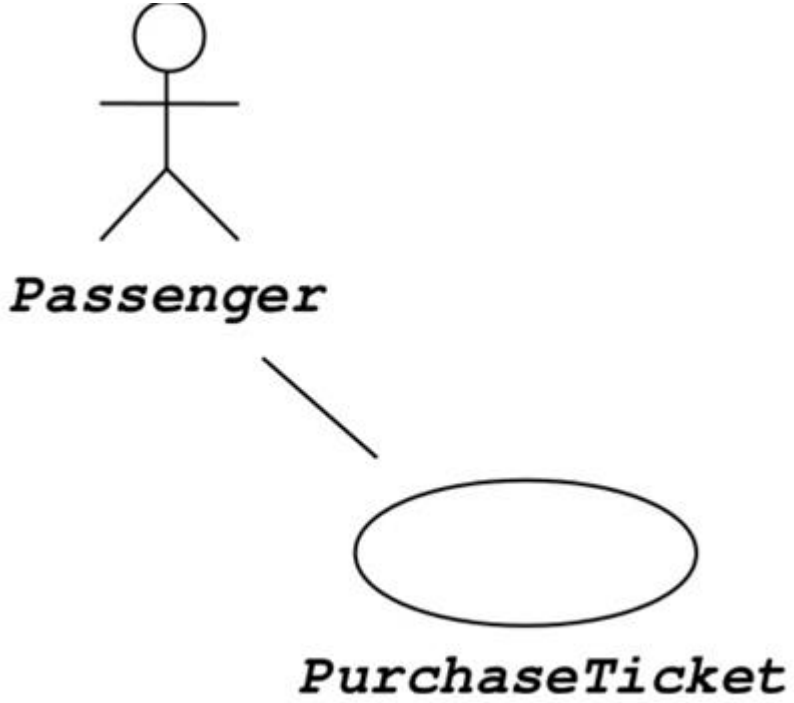


2. Basic diagrams

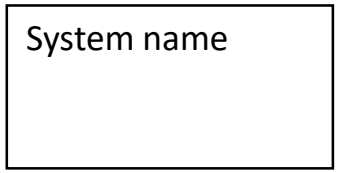
- Use Case
- Class
- Sequence

Use case diagram

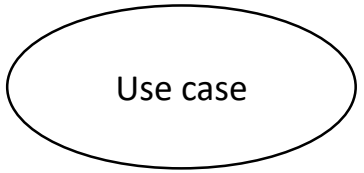
- Describe functionalities of system
- Requirements elicitation to represent external behavior
- Generating test case



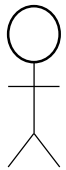
Use case diagram



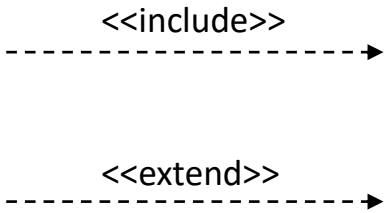
System boundary



Use case

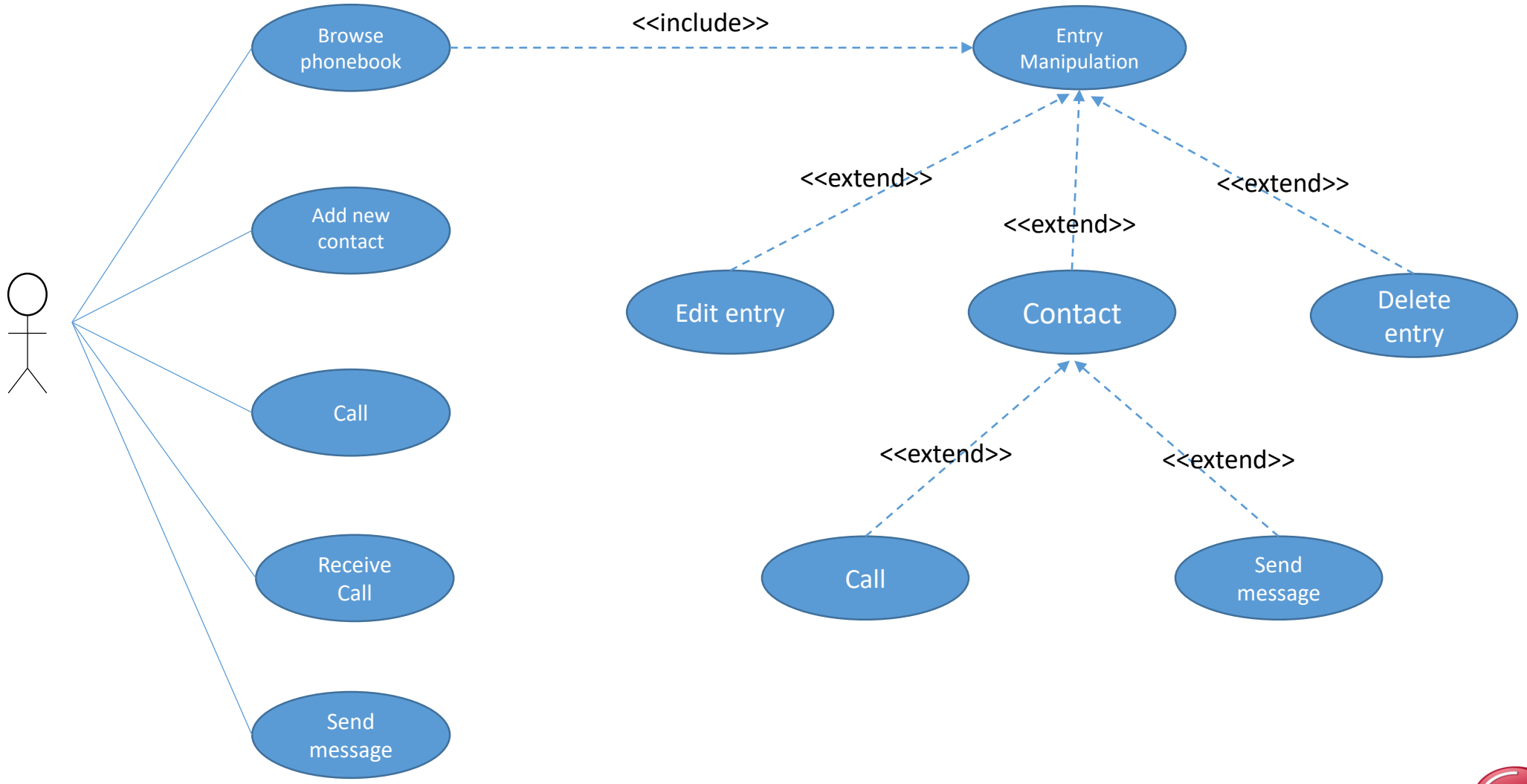


Actor



Relationships

Use case diagram

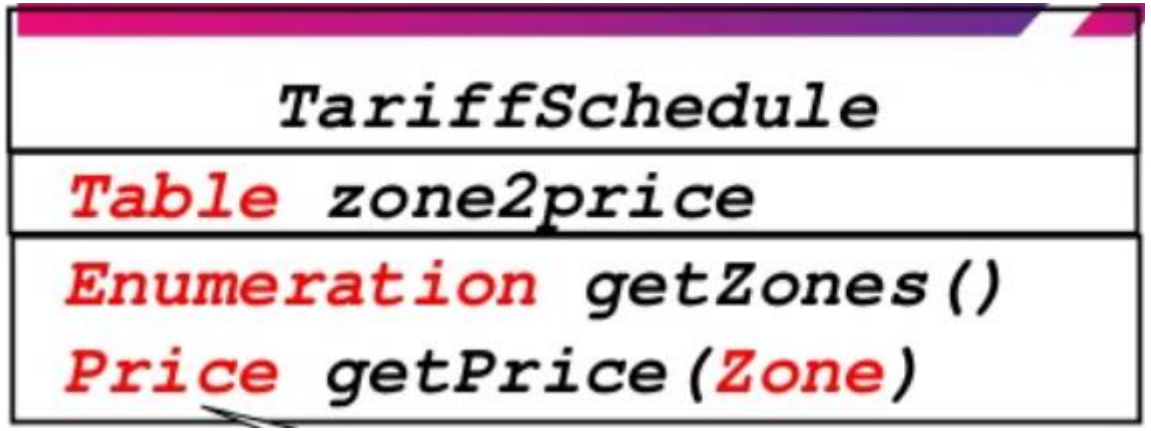


Class diagram

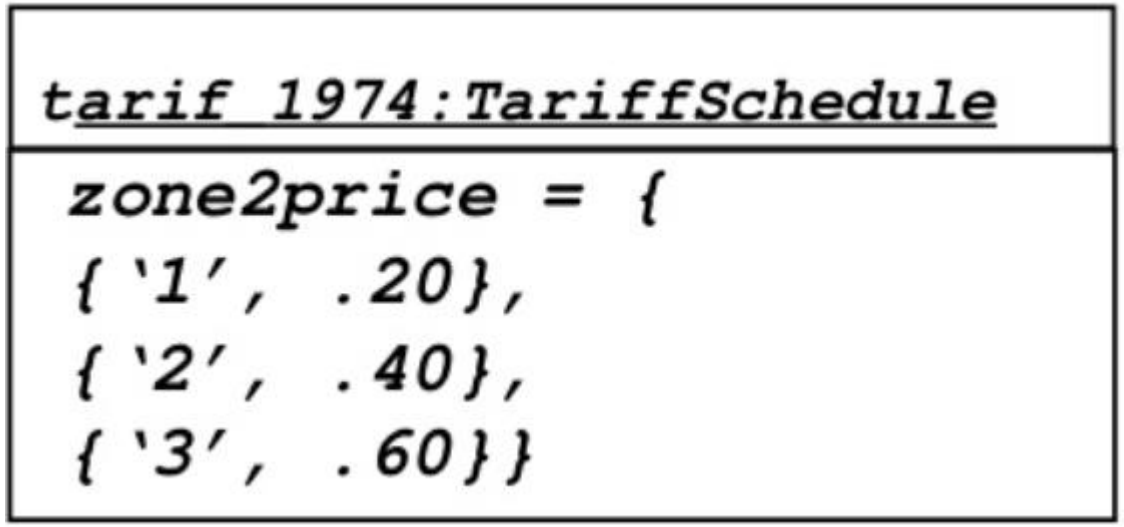
- Overview of classes and relationships
- Static diagram, don’t show what happens when interact
- Shows attributes and operations

Class and instance

- Class name
- Attributes
- Operations

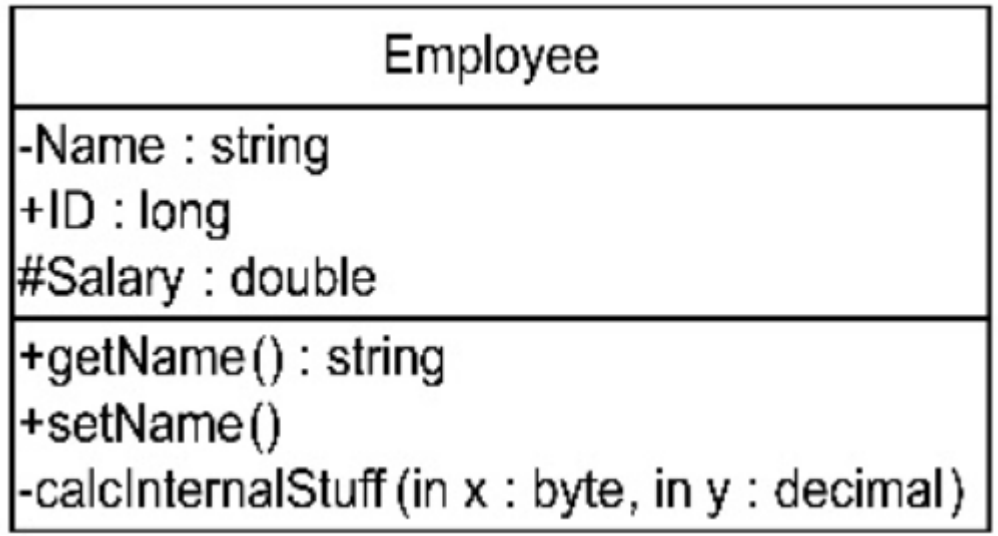


- Underlined name
- Specific values

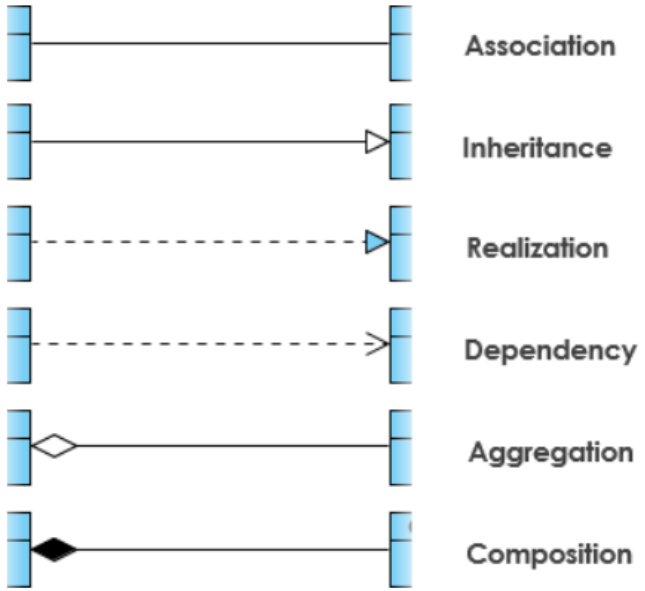


Class notations

- Modifiers:
  - Private: -
  - Public: +
  - Protected: #
  - Static: Underlined
  - Abstract class: *Name in italics*

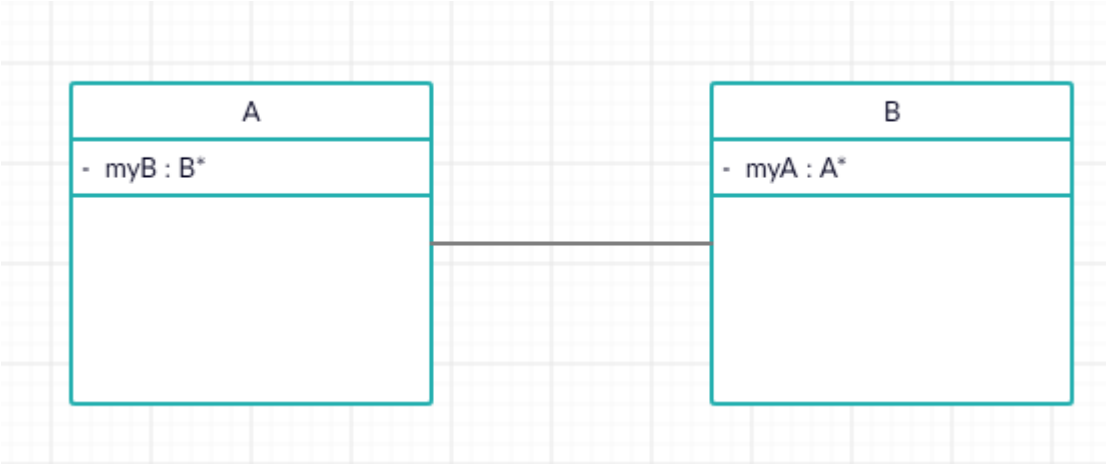


- Association (knows a)
- Inheritance (is a)
- Dependency (uses a)
- Aggregation (has a)
- Composition (has a)



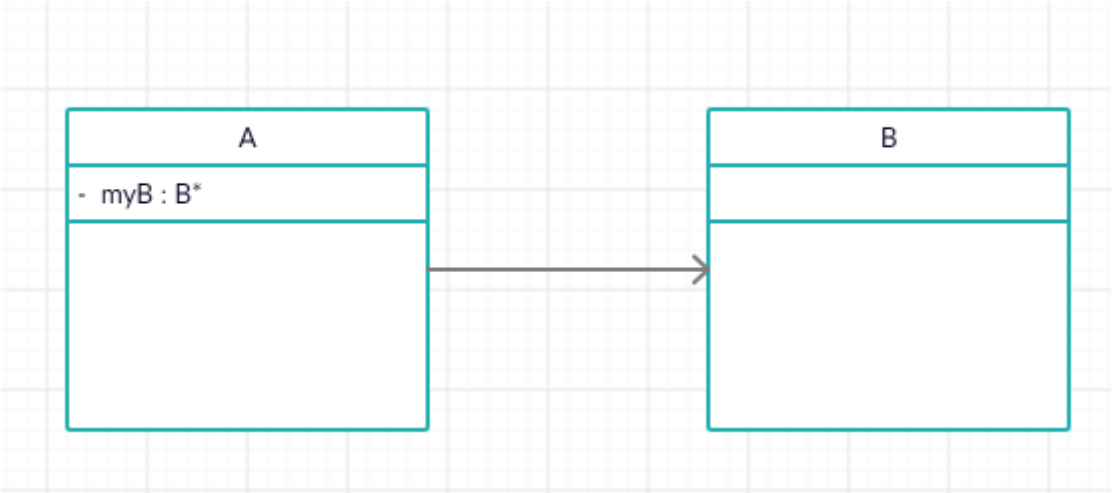
Association

- One contains a pointer or reference to another
- Binary:




```
class A
{
public:
    A(B* b);
    A();
    B* myB;
};
```

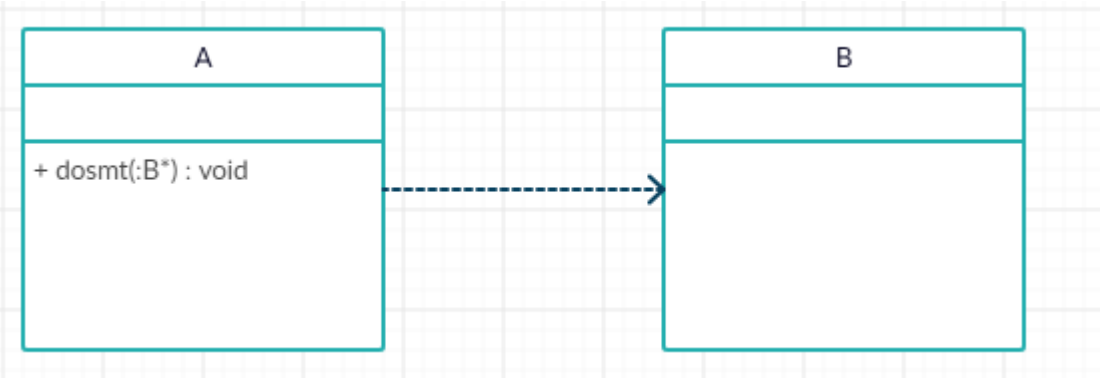
- Unary:




```
class B
{
public:
    B(A* a);
    A* myA;
};
```

Dependency

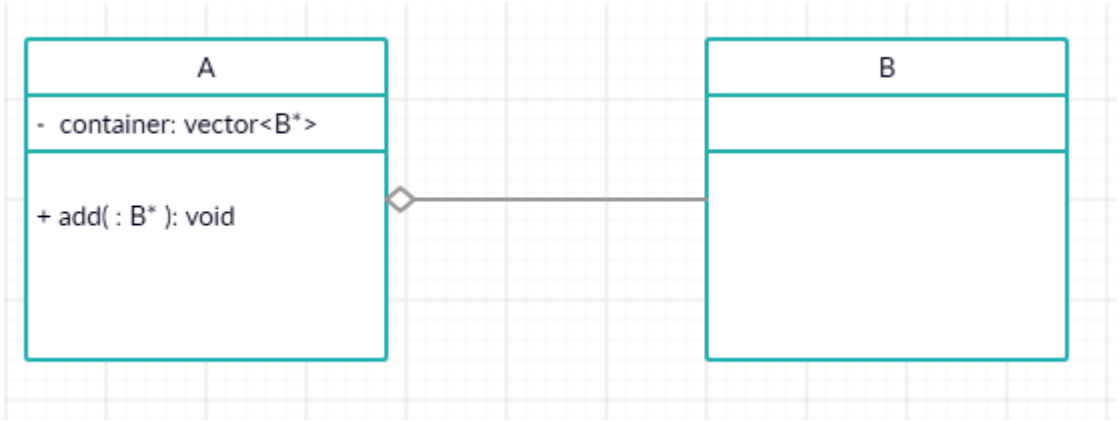
- One class is a parameter variable or local variable of a method of another




```
class A
{
public:
    A();
    void dosmt(B* b) {B* tmpB;};
};
```

Aggregation

- Independent existence: collection - member

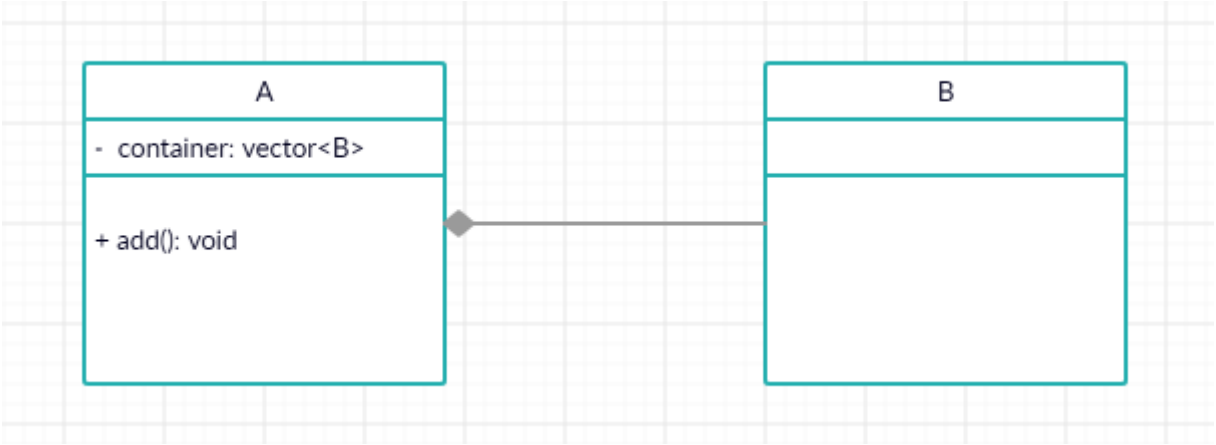



```
class A
{
public:
    A();
    void add(B* b) { container.push_back(b);
private:
    vector<B*> container;
};

class B {
    B();
};
```

Composition

- Dependent existence: lifetime control

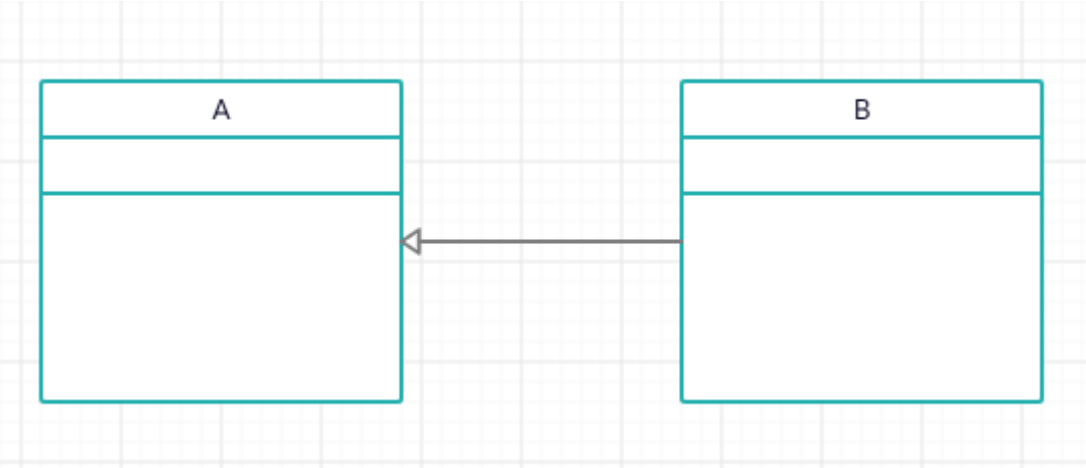



```
class A
{
public:
    A();
    void add() { B b;container.push_back(b);
private:
    vector<B> container;
};

class B  {
    B();
};
```

Inheritance

- A class is derived from another class



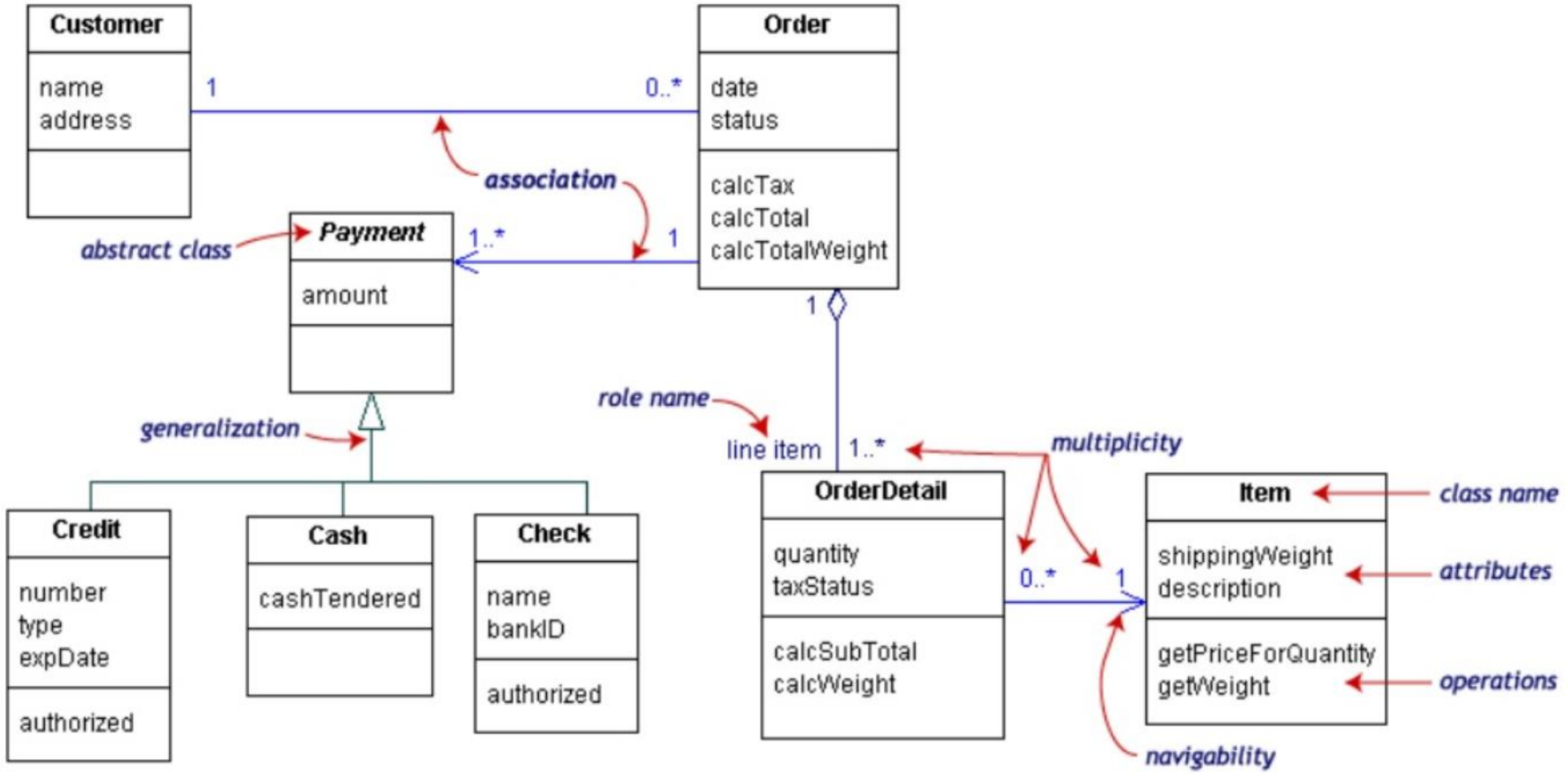

```
class A
{
public:
    A();
};

class B : public A {
    B();
};
```

Multiplicity

Multiplicities	Meaning
0..1	zero or one instance. The notation ***n*** **. .** ***m*** indicates ***n*** to ***m*** instances.
0..* *or* *	no limit on the number of instances (including none).
1	exactly one instance
1..*	at least one instance

Example

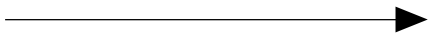


Sequence diagram

- Shows object interactions arranged in time sequence
- Depicts the sequence of messages exchanged between the objects in functional scenario

Sequence diagram

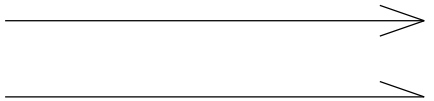
component



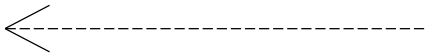
Synchronous message

Lifeline

Activation

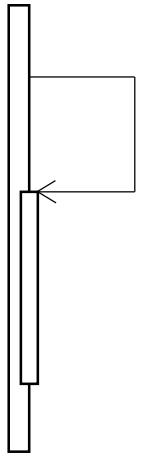


Asynchronous message

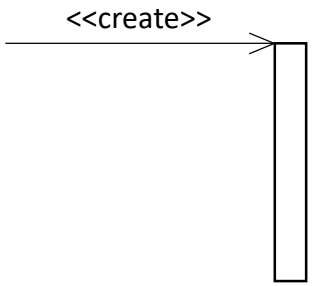


Return message

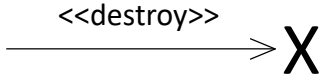
Sequence diagram



Self message



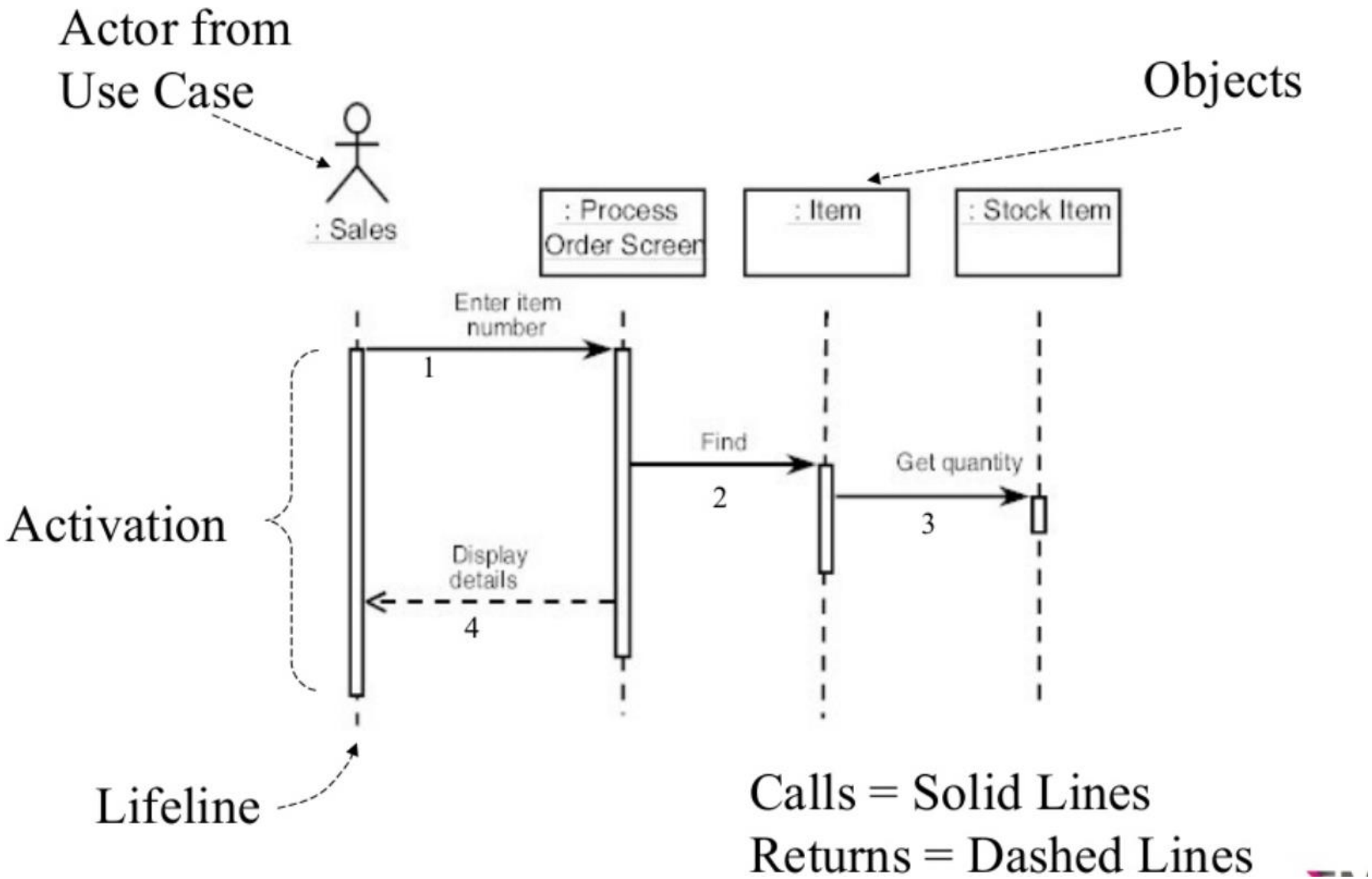
Create message



Delete message

End point/unknown recipent

Sequence diagram



3. Design pattern

- Good, simple and reusable design
- Provides solution for common problems occurs in software design

Pattern elements

- Name
- Problem description
- Solution
- Consequences

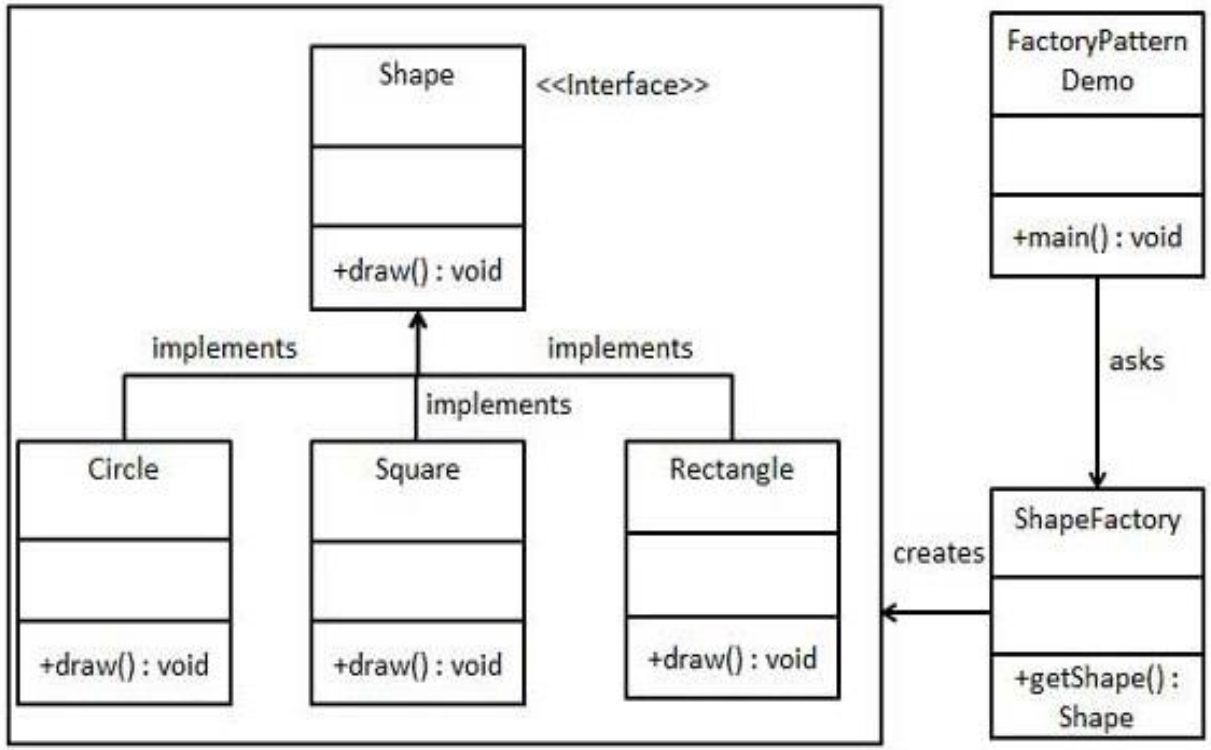
Types

- Creational: concern the process of object creation
- Structural: composition of classes or object
- Behavioral: ways in which class interact and distribute responsibility

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (class) (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory (87) Builder (97) Prototype (117) Singleton (127)	Adapter (object) (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Flyweight (195) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Observer (293) State (305) Strategy (315) Visitor (331)

Example

- Factory method
- To create object without exposing the creation logic to the client and refer to newly created object using common interface



Thank you