Chapter 4

Introduction to Objects

(Continued)

Zhang Shuang, zhangs@swc.neu.edu.cn

Data Encapsulation

- **Example**
  - **Design an operating system for a large mainframe computer. It has been decided that batch jobs submitted to the computer will be classified as high priority, medium priority, or low priority. There must be three queues for incoming batch jobs, one for each job type. When a job is submitted by a user, the job is added to the appropriate queue, and when the operating system decides that a job is ready to be run, it is removed from its queue and memory is allocated to it.**

m_1

definition of job-queue

```
initialize_job_queue()
{
      ……
}
```

m_2

definition of job-queue

```
add_job_to_queue(job j)
{
      ……
}
```

m_3

definition of job-queue

```
remove_job_from_queue(job j)
{
      ……
}
```

m_123

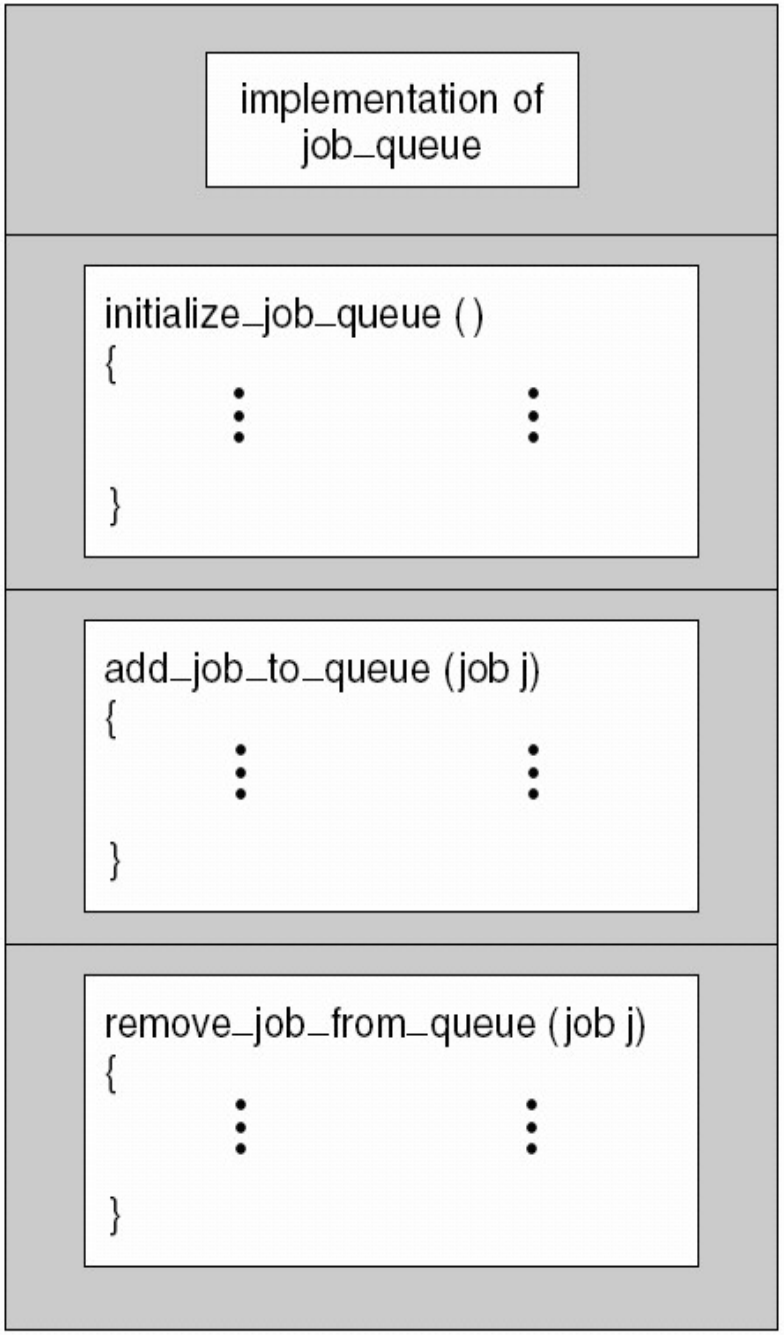
definition of job-queue

```
{
  job job_a, job_b;

  initialize_job_queue() {
      ……
  }
  add_job_to_queue (job_a) {
      ……
  }
  remove_job_from_queue (job_b) {
      ……
  }
}
```

m_123

```
{
    job   job_a, job_b;
          :              :

    initialize_job_queue ( );
          :              :

    add_job_to_queue (job_a);
          :              :

    remove_job_from_queue (job_b);
          :              :
}
```

m_encapsulation



Data Encapsulation

- **m_encapsulation has informational cohesion**
- **m_encapsulation is an implementation of *data encapsulation*, that is, a data structure, together with the operations to be performed on that data structure.**

Advantage of Data Encapsulation

1. **Data encapsulation & Development**

- **Data encapsulation is an example of *abstraction***
- **Job queue example**
  - **Data structure**
    - **job_queue**
  - **Three operations**
    - **initialize_job_queue**
    - **add_job_to_queue**
    - **delete_job_from_queue**

Data Abstraction

Advantage of Data Encapsulation

- **Data abstraction allows the designer to think at the level of the data structure and the operations performed on it and only later be concerned with the details of how the data structure and operations are implemented.**

Advantage of Data Encapsulation

2. **Data encapsulation & Maintenance**

- **Approaching data encapsulation from the viewpoint of maintenance, a basic issue is to identify the aspects of a product likely to change and design the product to minimize the effects of future changes.**

Advantage of Data Encapsulation


```
class JobQueueClass{
    public int queueLength;   // length of job queue
    public int queue[] = new int[25];
    public void initializeJobQuieue(){
        queueLength = 0;
    }
    public void addJobToQueue(int jobNumber){
        ......
    }
    public void removeJobFromQueue(){
        ......
    }
}
```

Advantage of Data Encapsulation

2. **Data encapsulation & Maintenance**

- **Invoking class doesn't have any knowledge as to how the job queue is implemented; the only information needed to use *JobQueueClass* is interface information regarding the three methods.**
- **So, data encapsulation supports the implementation of data abstraction in a way that simplifies maintenance and reduces the chance of a regression fault.**

Abstract Data Types

- **Abstract data type ---- a data type together with the actions to be performed on instantiations of that data type.**

Abstract Data Type


```
class JobQueueClass{
    public int queueLength;   // length of job queue
    public int queue[] = new int[25];
    public void initializeJobQuieue(){
        queueLength = 0;
    }
    public void addJobToQueue(int jobNumber){
        ......
    }
    public void removeJobFromQueue(){
        ......
    }
}
```

Information Hiding

- **Data abstraction**
  - **Designer thinks at level of an Abstract Data Type**
- ***Information hiding → detail hiding***
  - **Design the modules in way that implementation details are hidden from other modules**
  - **Future change is localized**
  - **Changes cannot affect other modules**

Information Hiding

- Java abstract data type implementation with information hiding

```
class JobQueueClass{
        private int queueLength;
        private int queue[] = new int[25];
        public void initializeJobQuieue(){
                queueLength = 0;
        }
        public void addJobToQuieue(int
                                        jobNumber){
                ......
        }
        public void removeJobFromQuieue(){
                ......
        }
}
```

Information Hiding

- **Effect of information hiding via *private* attributes**

Scheduler

```
{
    int                 job1, job2;
        :                   :
    highPriorityQueue.initializeJobQueue ();
        :                   :
    mediumPriorityQueue.addJobToQueue (job1);
        :                   :
    job2 = lowPriorityQueue.removeJobFromQueue ();
        :                   :
}
```

JobQueue

Implementation details of:
queue
queueLength
initializeJobQueue
addJobToQueue
removeJobFromQueue

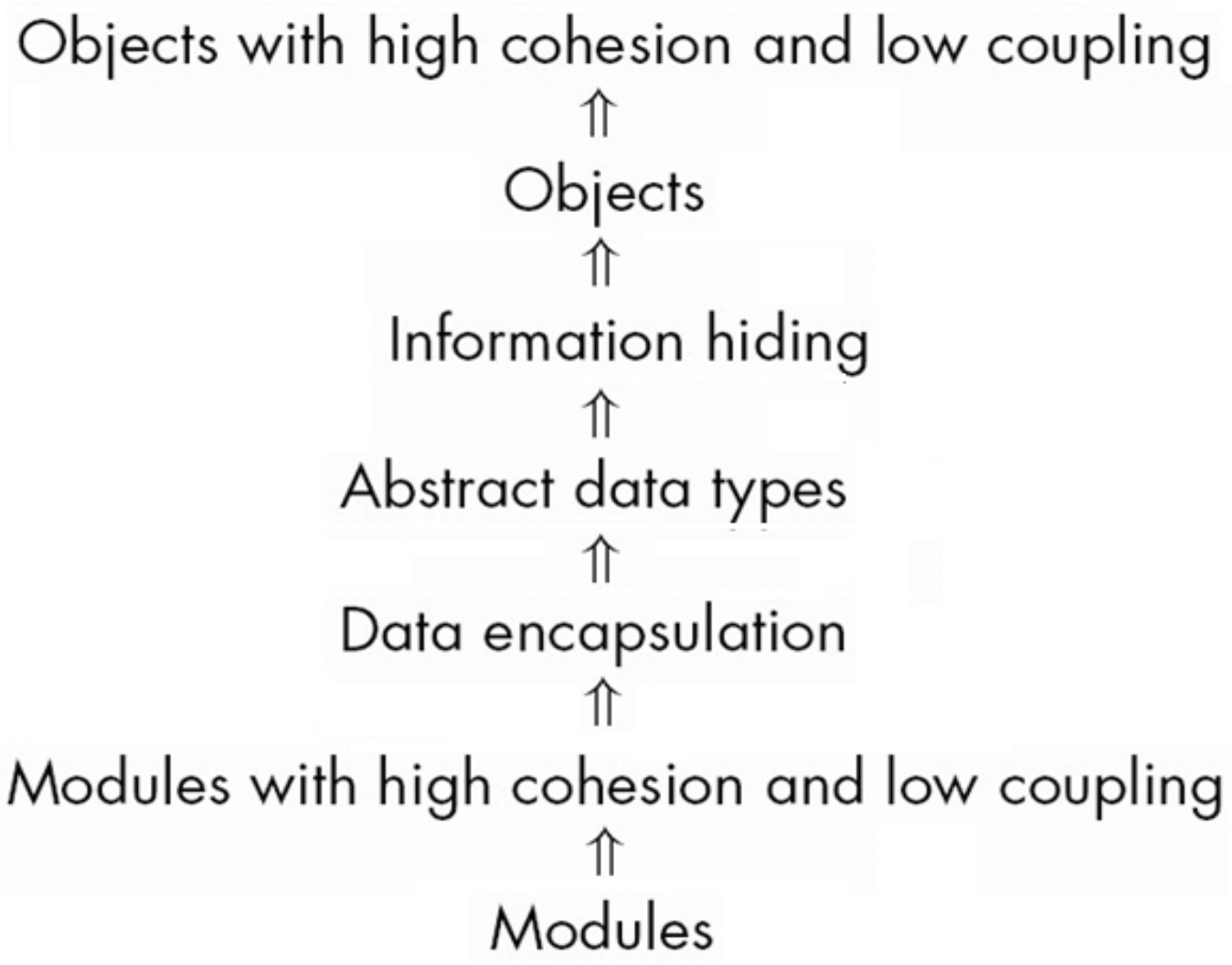
Interface information regarding:

initializeJobQueue
addJobToQueue
removeJobFromQueue

Invisible outside JobQueue

Visible outside JobQueue

Major Concepts



Objects

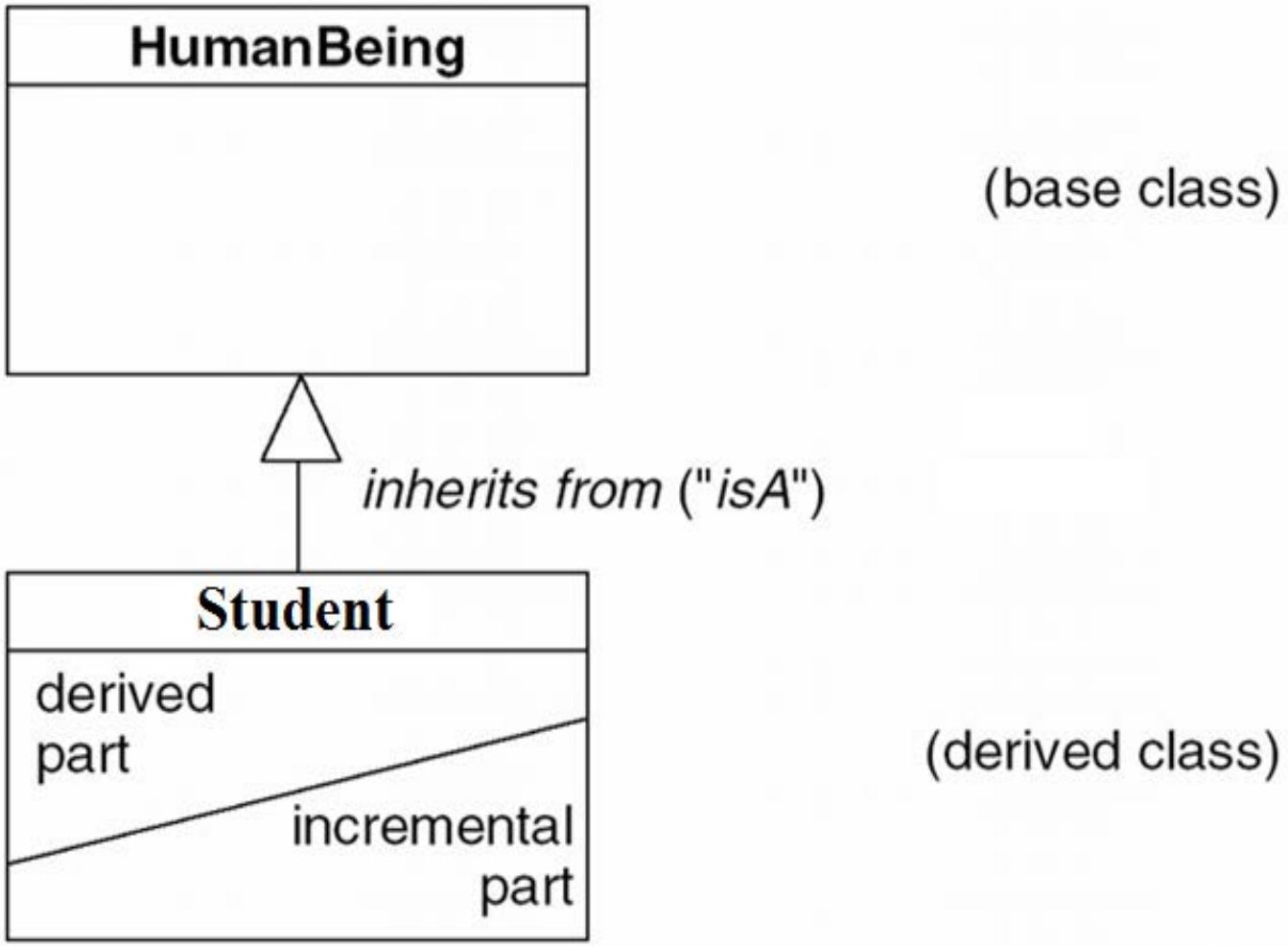
- ***Class*: abstract data type that supports *inheritance*.**
- ***Objects* are instantiations of classes.**

Inheritance

- **Define *HumanBeing* to be a *class***
  - **A *HumanBeing* has *attributes*, e.g., name, ID, and so on.**
  - **Assign values to attributes when describing object.**
- **Define *Student* to be a *subclass* of *HumanBeing***
  - **A *Student* has all attributes of a *HumanBeing*, plus attributes of his/her own (e.g., School, StudentNo).**
  - **A *Student* inherits all attributes of *HumanBeing*.**

Inheritance

- **UML notation ---- Inheritance is represented by a large open triangle.**



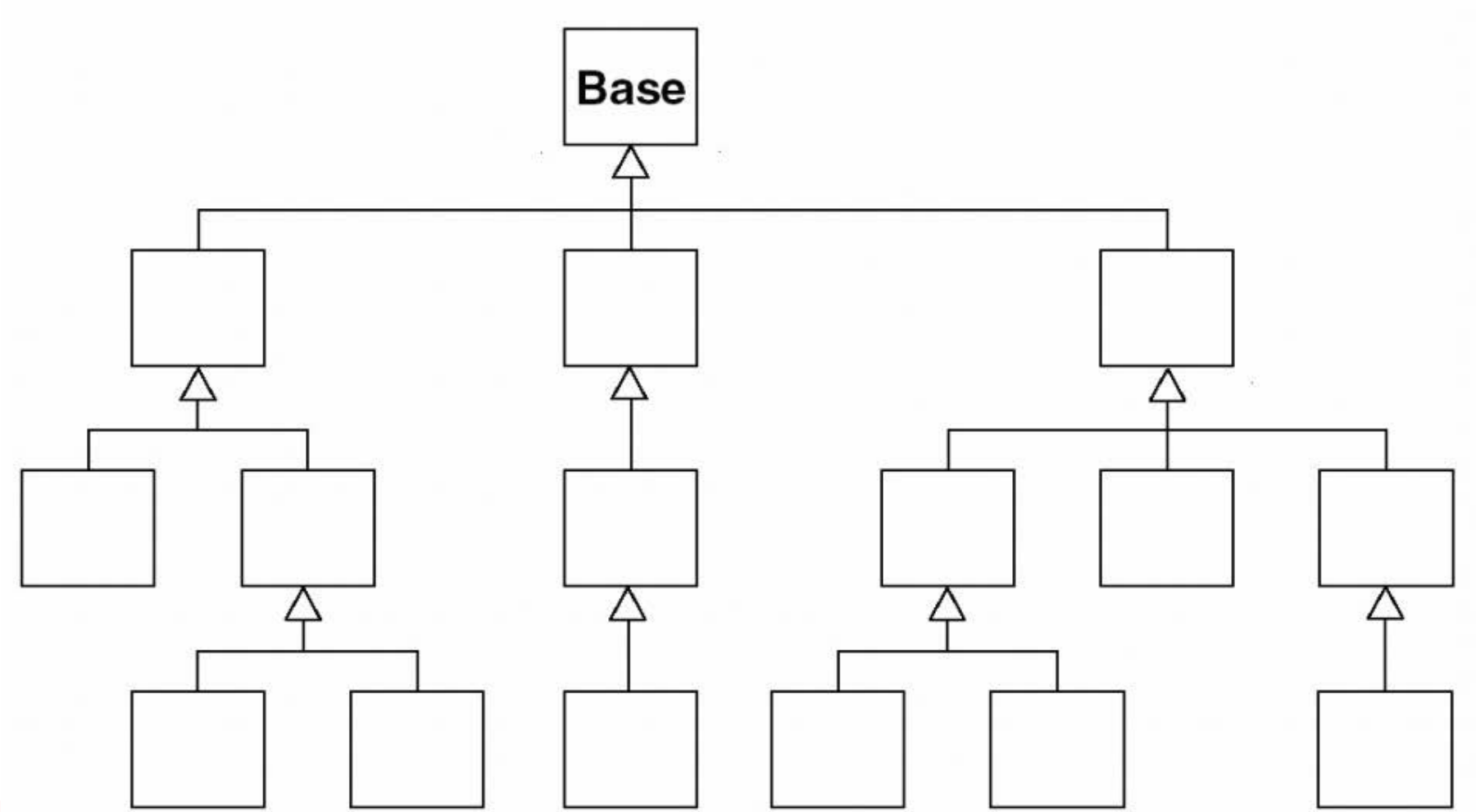
Java Implementation

- The property of inheritance is an essential feature of object-oriented languages such as Java, Smalltalk, C++ (but not C, Fortran)

```
class Humanbeing
{
        String Name;
        Date Birthday;

    // public declarations of operations on HumanBeing
}

class Student extends HumanBeing
{
        String School;
        Strong StudentNo;

    // public declarations of operations on Student
}
```

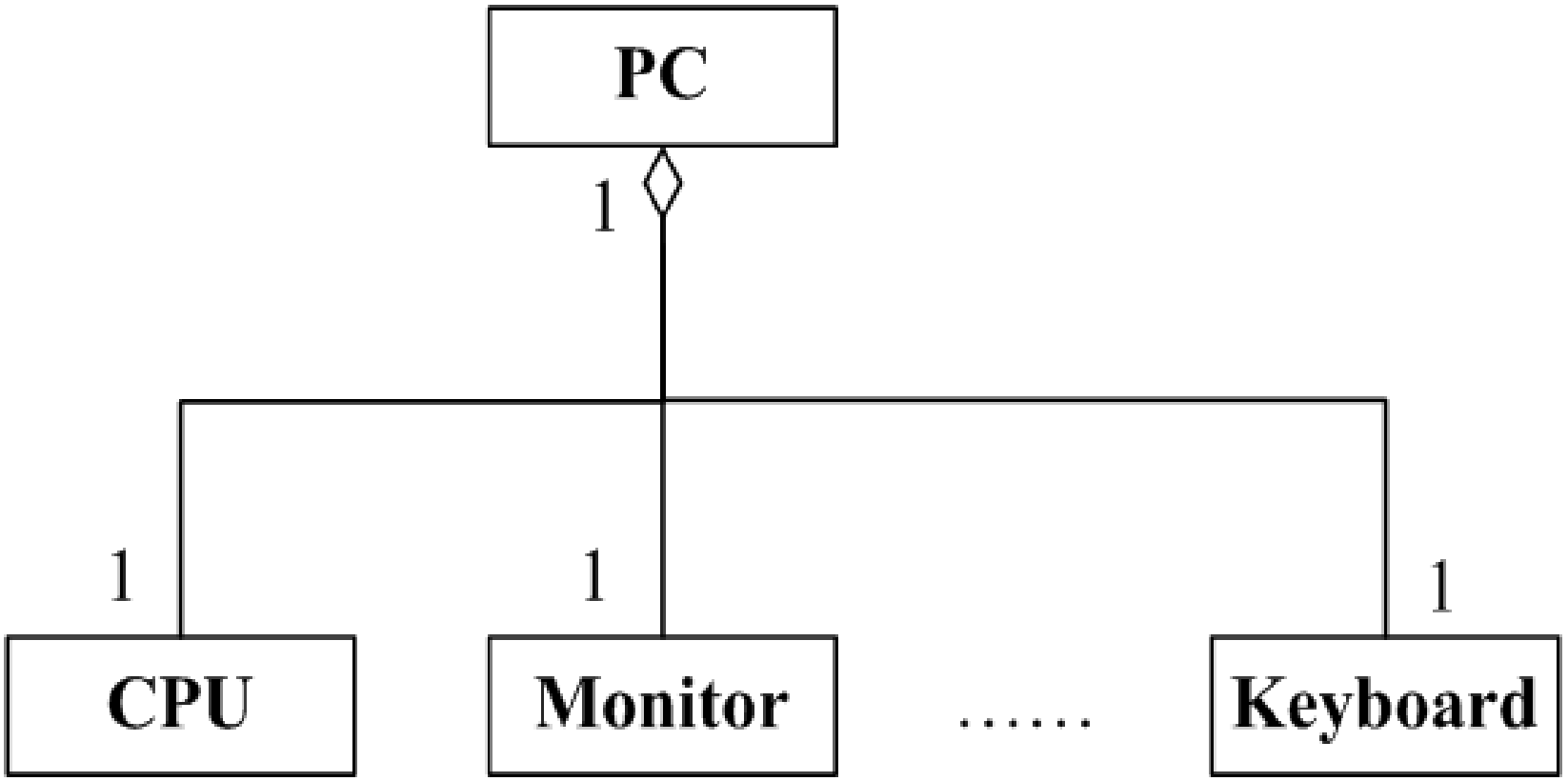
Inheritance

- **Thinking: Fragile base-class problem**



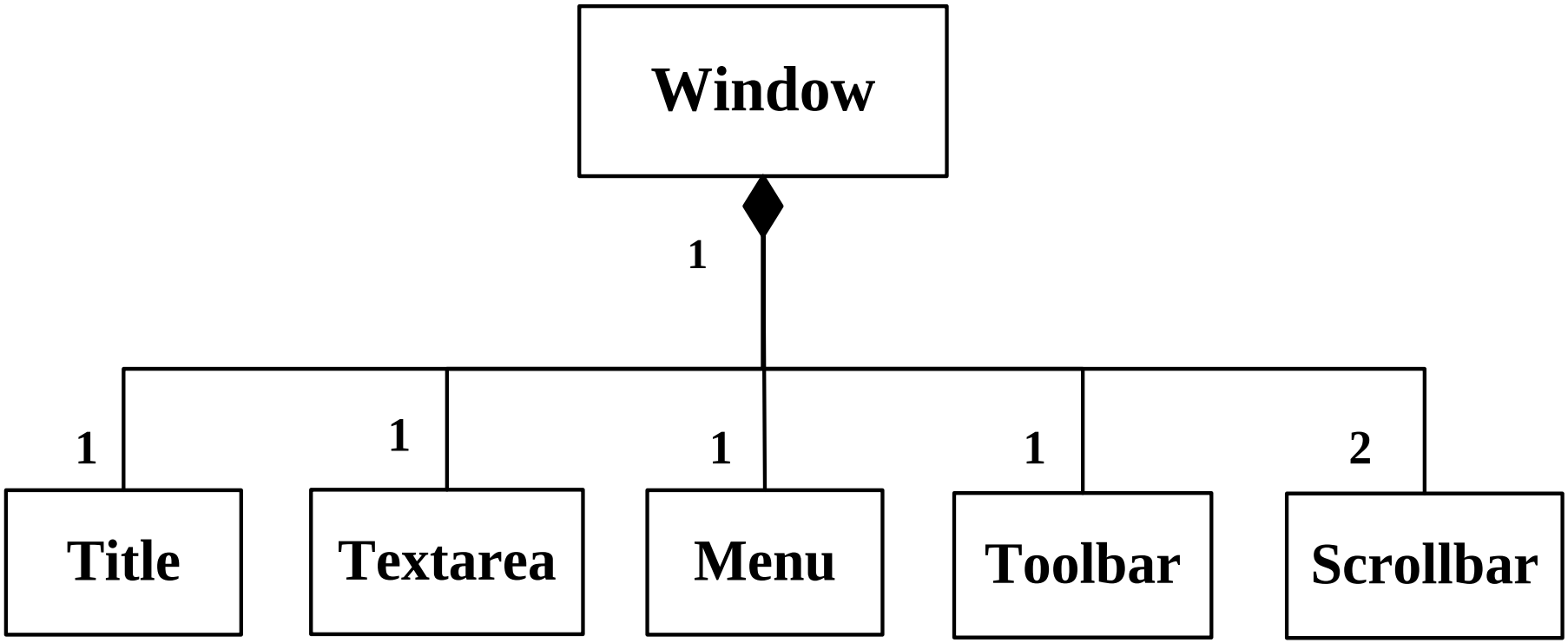
Aggregation

- **UML Notation**



Composition

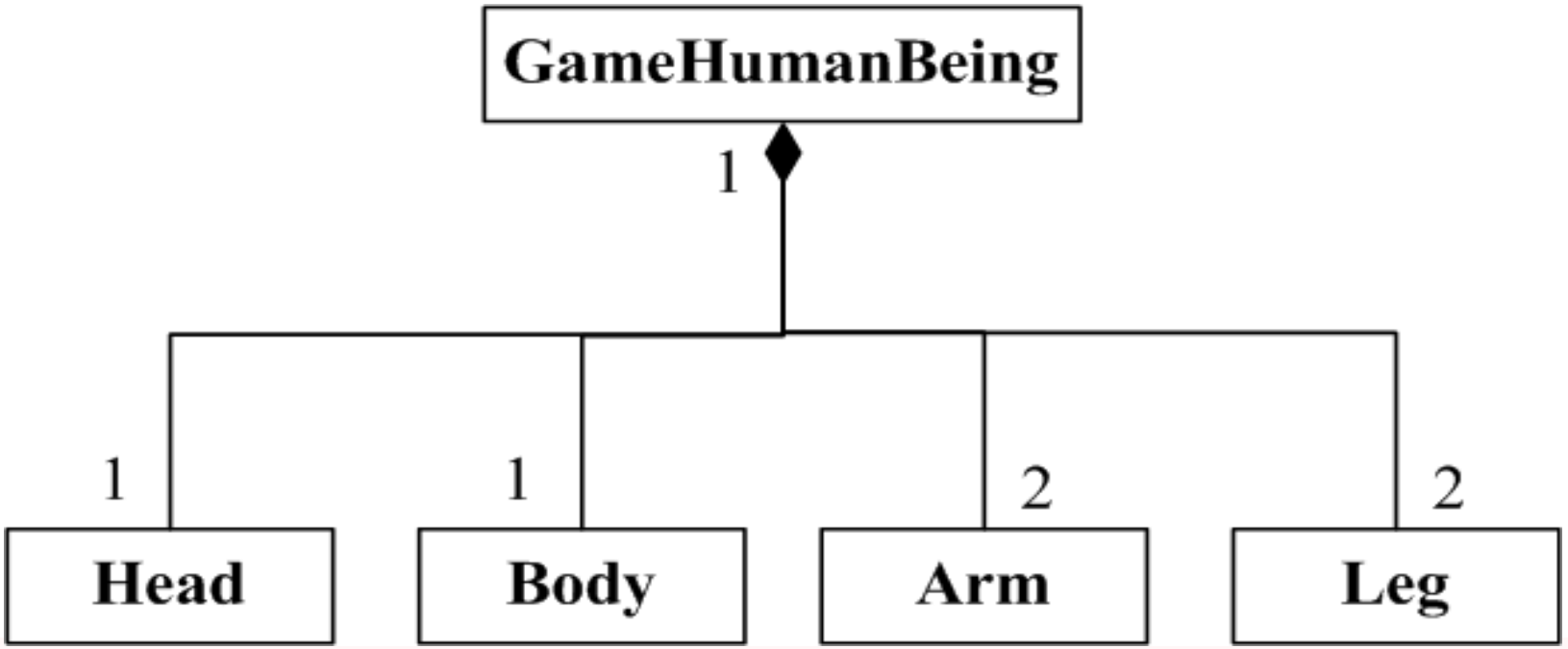
- **UML Notation**



Association

- **UML Notation**

Multiplicity



Multiplicity

- **Every class engaged in a relationship should have a multiplicity. (*except inheritance*)**
- **If the multiplicity is 1, indicate that it is 1; in order that reader know that multiplicity has been considered.**
- **UML Multiplicity Indicators**

Polymorphism and Dynamic Binding

- **Structural paradigm**
  - **Must explicitly invoke correct version**

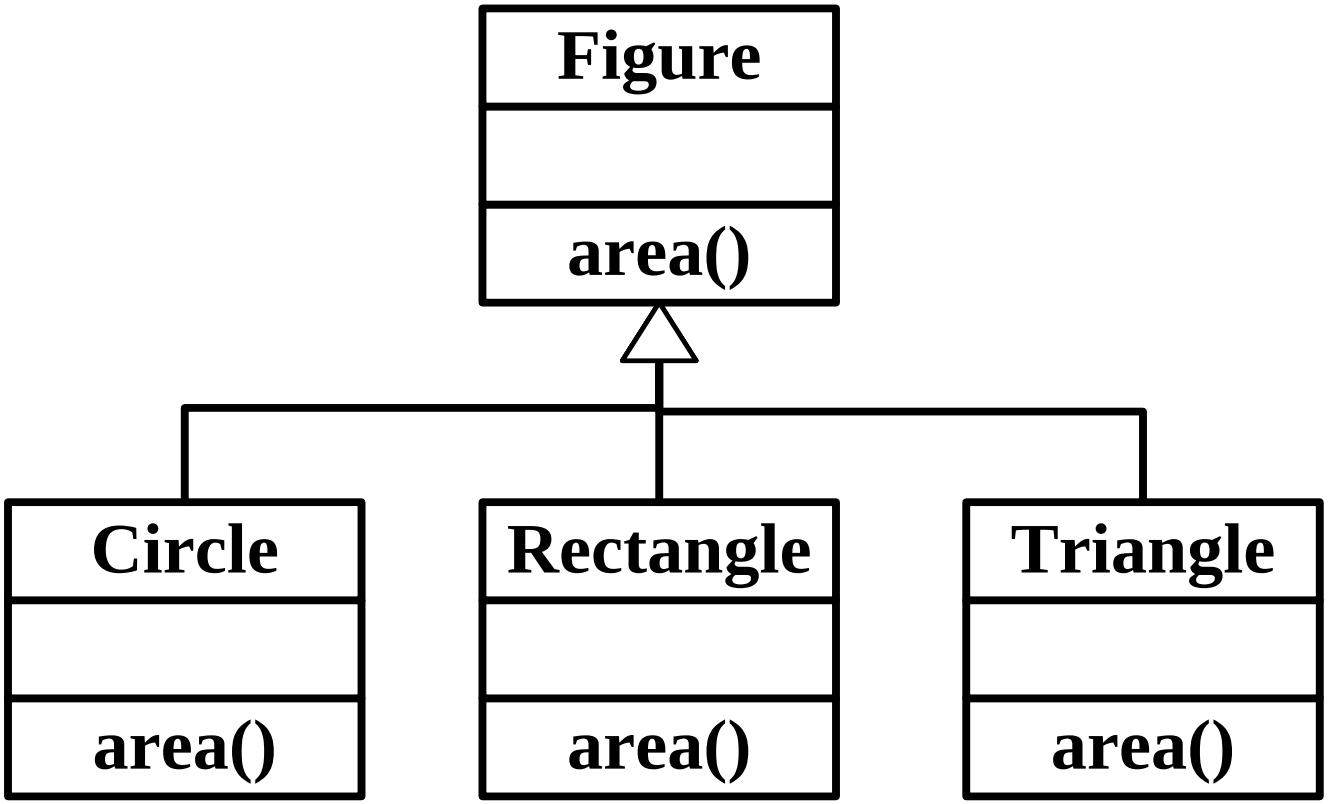
function area_circle()

function area_rectangle()

function area_triangle()

Polymorphism and Dynamic Binding

- Object-Oriented Paradigm



Polymorphism and Dynamic Binding

```
abstract class Figure{
      ......
      abstract double area();
}

class Circle extends Figure{
      double Radius;
      double area(){ ..... }
}

class Rectangle extends Figure{
      double Length, Width;
      double area(){ ..... }
}

class Triangle extends Figure{
       double area(){ ..... }
}
```

```
class Test{
      ......
      method_1(){
            Figure aFigure;
            ......
            aFigure = ……;
            double area = aFigure.area();
            ......
      }
}
```

Polymorphism and Dynamic Binding

- It is not necessary to determine which method to invoke to get area.
- Send only message *aFigure.area()* is sent.
  - Correct method invoked at run-time (dynamically)
  - Dynamic binding
- Method *area()* can be applied to objects of different classes
  - Polymorphic

Polymorphism and Dynamic Binding

- It can have a negative impact on maintenance
  - Code is hard to understand if there are multiple possibilities for a specific method.
- Both strength and weakness of the object-oriented paradigm

Cohesion and Coupling of Objects

- **The only feature unique to the object-oriented paradigm is *inheritance*.**
  - **Cohesion has nothing to do with inheritance.**
  - **Two objects with the same functionality have the same cohesion.**
  - **It does not matter if this functionality is inherited or not.**

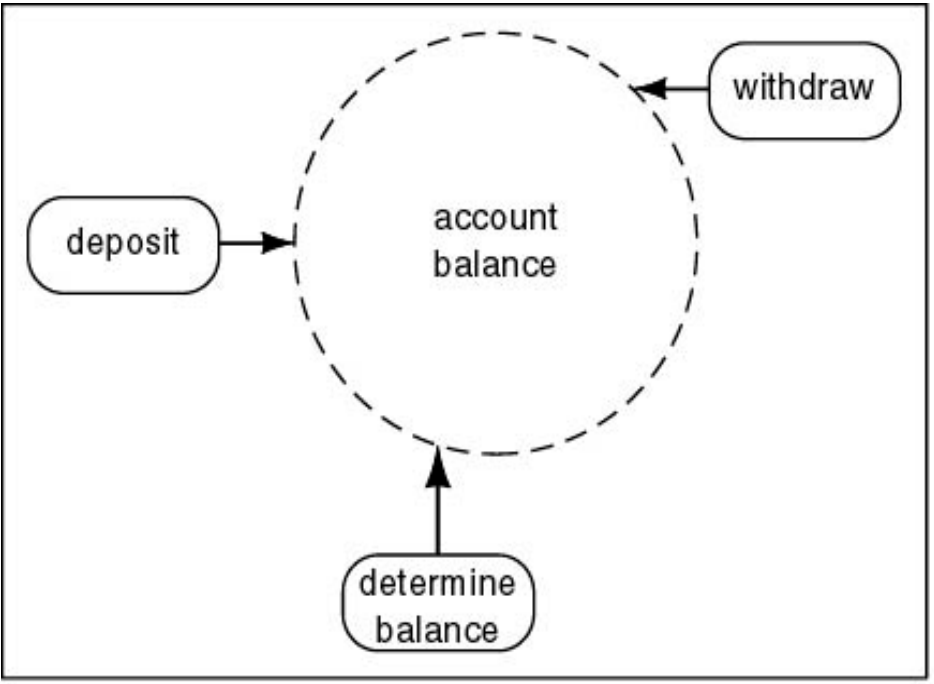
Advantages of Objects

- Same as advantages of abstract data types
  - Information hiding
  - Data abstraction
  - Procedural abstraction
- Inheritance provides further data abstraction
  - Easier and less error-prone product development
  - Easier maintenance
- Objects are more reusable than modules with functional cohesion

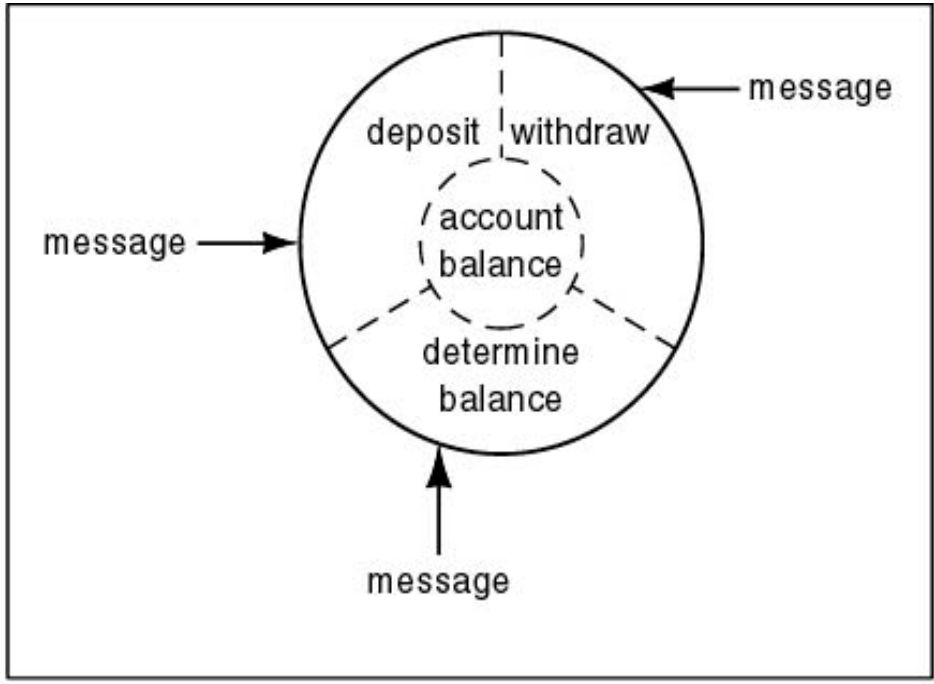
The Object-Oriented Paradigm

- **Both *data* and *actions* are of equal importance**
- **Structured paradigm does not take them important at the same time.**
- **Example:**
  - **Bank account**
    - **Data: account balance**
    - **Actions: deposit, withdraw, determine balance**

Structured vs. Object-Oriented Paradigm



(a)



(b)

- **Information hiding**
- **Responsibility-driven design**

Responsibility-Driven Design

- Also called “Design by Contract”
- Send flowers to your friend in Beijing
  - Call 086-*******(a flower shop), to make your demand.
  - After you pay, the contract will be fulfilled.
  - Who, how delivery the flowers?
  - *Information hiding*
- Object-oriented paradigm
  - “Send a message to a method [*action*] of an object”

Transition From Analysis to Design

- **Structured paradigm:**
  - **Jolt between analysis (what) and design (how)**
- **Object-oriented paradigm:**
  - **Objects enter from the very beginning**

Structured Paradigm	Object-OrientedParadigm
1. Requirements phase	1. Requirements phase
2. Specification (analysis) phase	2′. Object-oriented analysis phase
3. Design phase	3′. Object-oriented design phase
4. Implementation phase	4′. Object-oriented programming phase
5. Integration phase	5. Integration phase
6. Maintenance phase	6. Maintenance phase
7. Retirement	7. Retirement

In More Detail

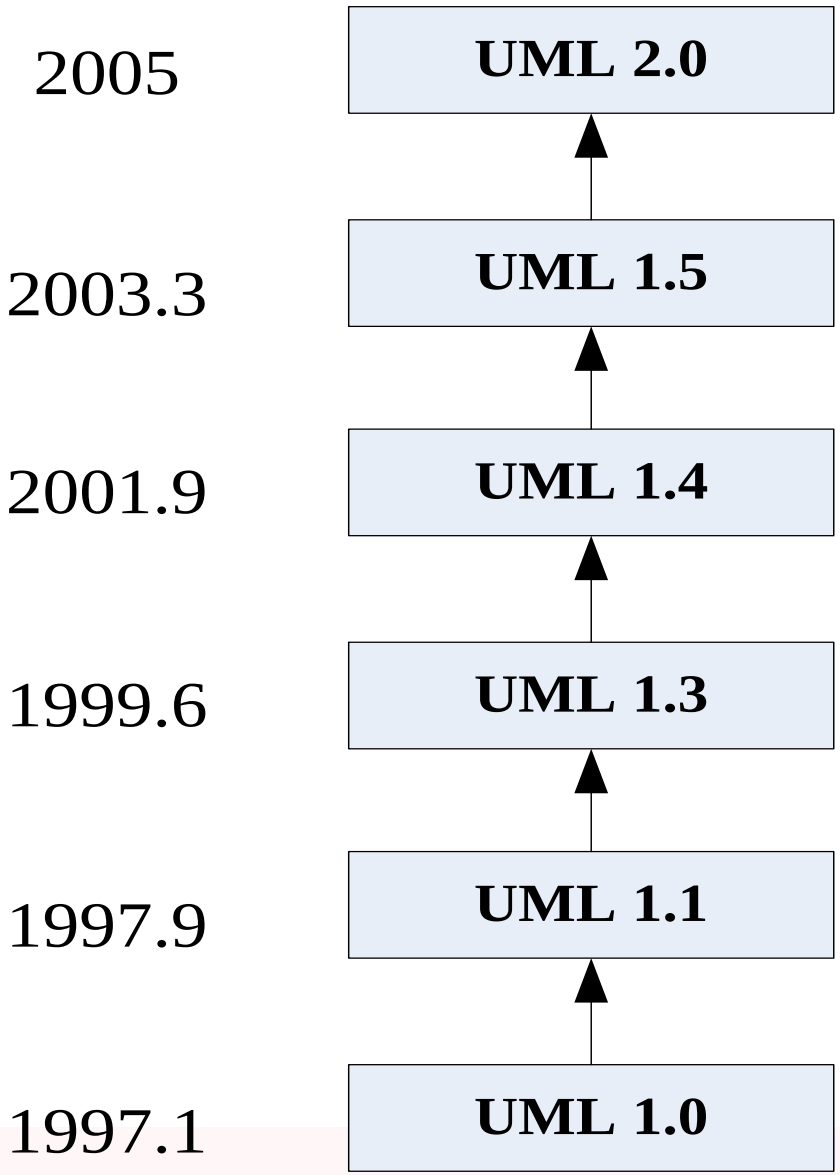
- **Objects enter here**

Structured Paradigm	Object-Oriented Paradigm
2. Specification (analysis) phase • Determine what the product is to do	2′. Object-oriented analysis phase • Determine what the product is to do • Extract the objects
3. Design phase • Architectural design (extract the modules) • Detailed design	3′. Object-oriented design phase • Detailed design
4. Implementation phase • Implement in appropriate programming language	4′. Object-oriented programming phase • Implement in appropriate object-oriented programminglanguage

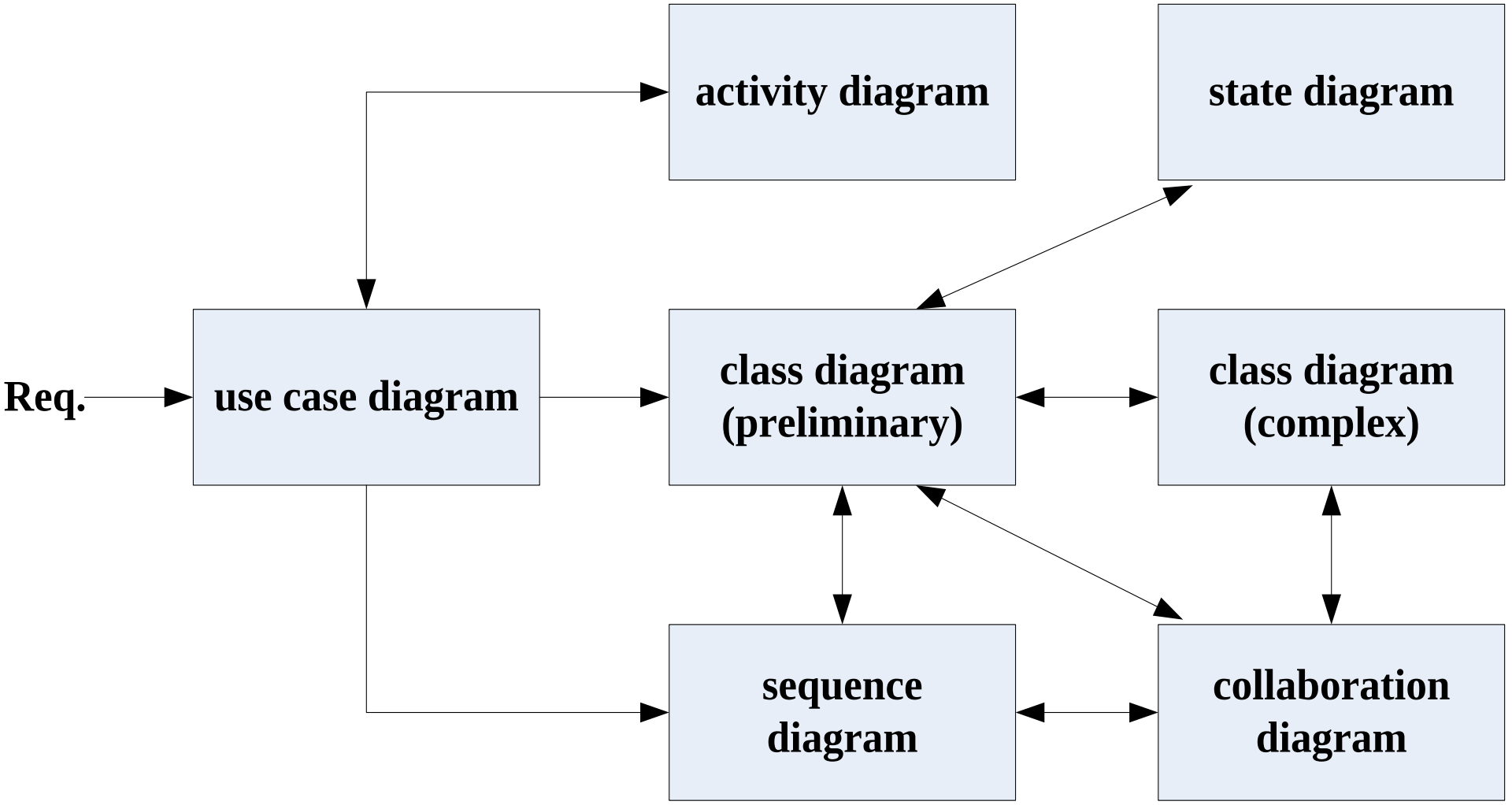
Introduction to UML

- UML----Unified Modeling Language
- Three amigos
  - Grady Booch ---- Booch Method, 1991
  - James Rumbaugh ---- OMT (Object Modeling Technique), 1991
  - Ivar Jacobson ---- OOSE (Object-Oriented Software Engineering), 1992

UML History


2005
UML 2.0
2003.3
UML 1.5
2001.9
UML 1.4
1999.6
UML 1.3
1997.9
UML 1.1
1997.1
UML 1.0

UML Diagrams



UML diagrams relationship

UML Tools

- Microsoft Visio
- IBM Rational Rose
- MagicDraw
- Together
- ArgoUML
- Clear Case
- RequisitePro


Thank You !