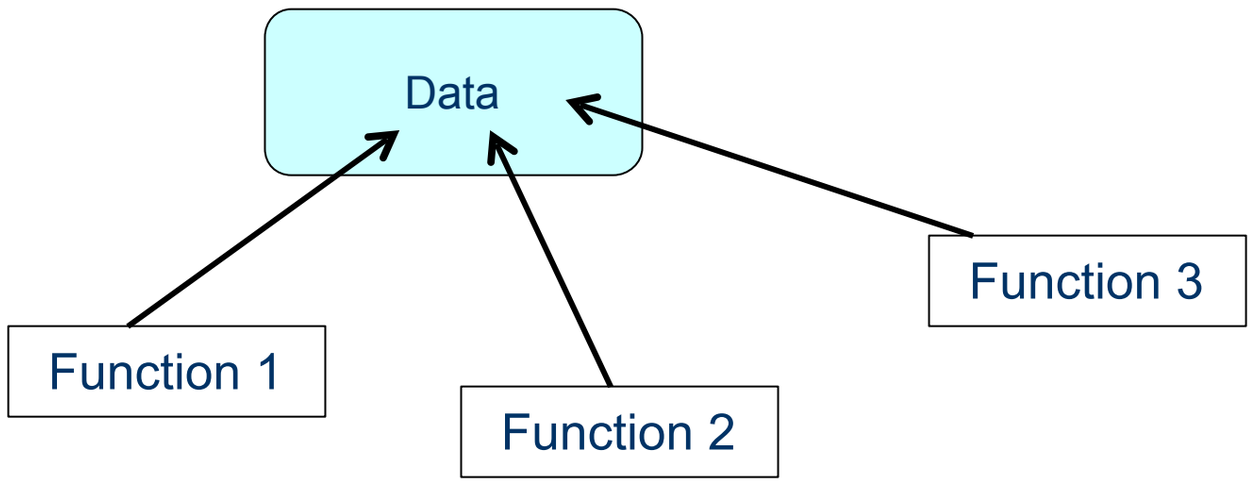
Implementation

- Reminders of object-oriented programming
- From design to implementation

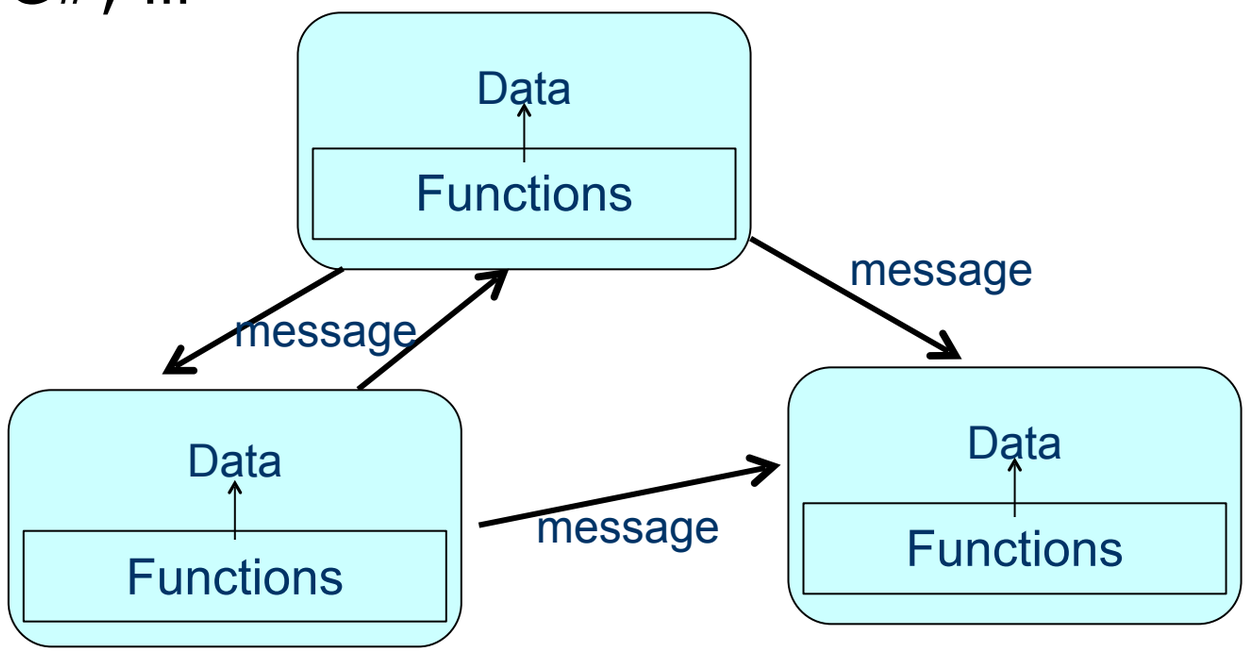
Object-oriented programming

- Functional/imperative programming
  - C/Pascal



Object-oriented programming

- Object-oriented programming
  - C++, Java, C#, …



- Encapsulation : class
  - Attributes
  - Methods
  - Constructors et destructors
- Inheritance
- Abstract classes et interfaces
- Polymorphism

Object-oriented programming

- Class : C++

```
class User {
public:
        User(string n, int a):name(n), age(a) {}
        string getName() {return name;}
        int  getAge() const {return age;}
        void setName(string n) {name = n;}
        void setName(int a) {age = a;}
        void print() const ;
        …
[private:]
        string name;
        int age;
};

void    User::print() const {
        cout << "name: " << name << " age: " << age
<< endl;
}


…
User u(" Nguyen Van A ", 35);
User* p = new User( " Nguyen Van A ", 35 );
…
delete p;
…
```

Object-oriented programming

- Class : Java

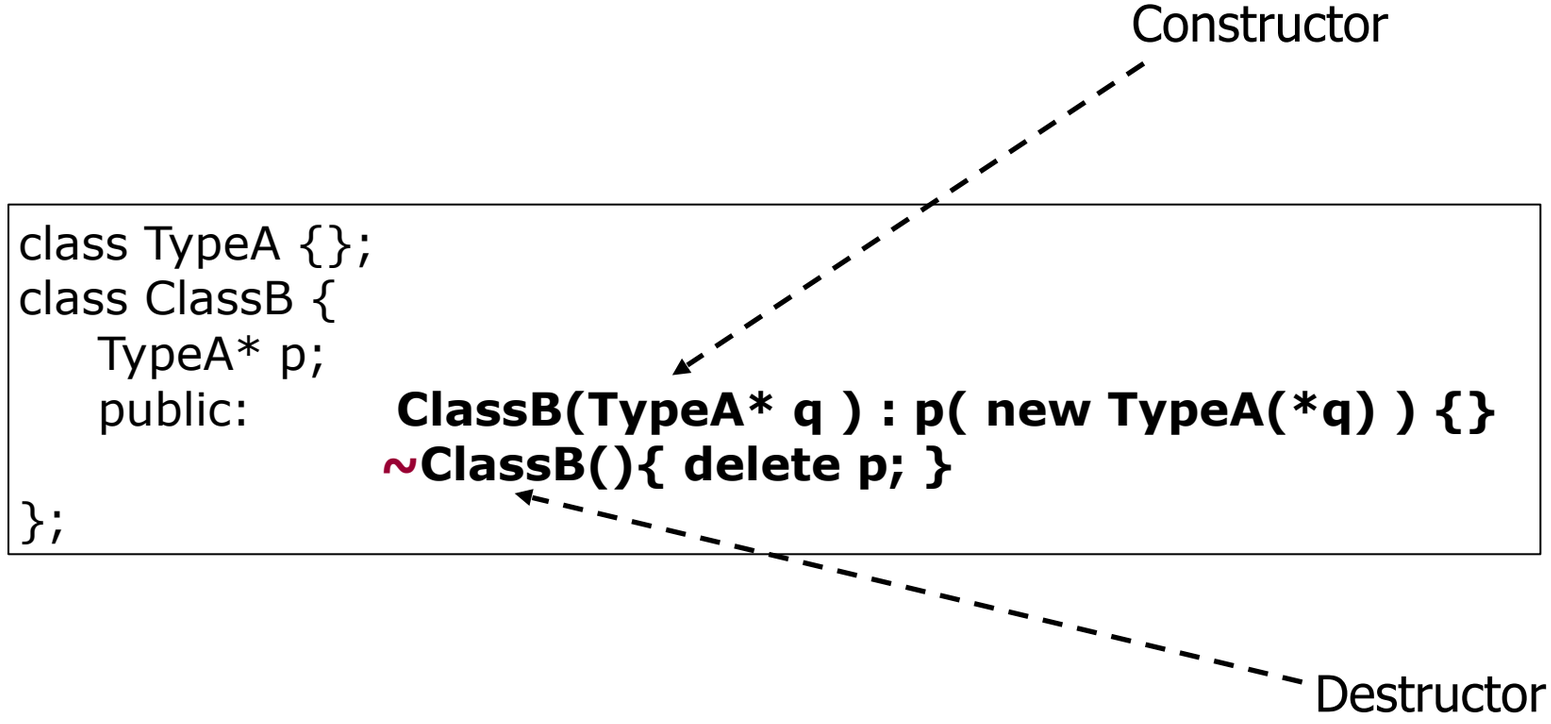
```
class User {
      public        User(String n, int a) {name = n; age = a;}
      public        String getName() {return name;}
      public        int  getAge() {return age;}
      public        void setName(String n) {name = n;}
      public        void setName(int a) {age = a;}
      public        void print(){
                System.out.println( "name: " + name + " age: " + age );
      }
            …

      private       String name;
      private       int age;
}

…
User u = new User (“Nguyen Van A", 35 );
…
```

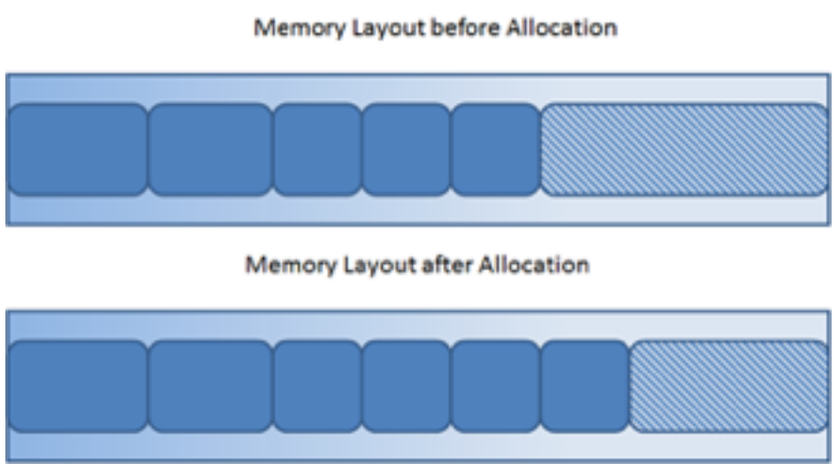
Object-oriented programming

- Constructor and Destructor: C++
  - Constructor
    - initialise attributes and then allocate memory for the attributes
  - Destructor
    - De-allocate the dynamic memory
    - Mandatory: if there are pointer attributes and the memory allocations

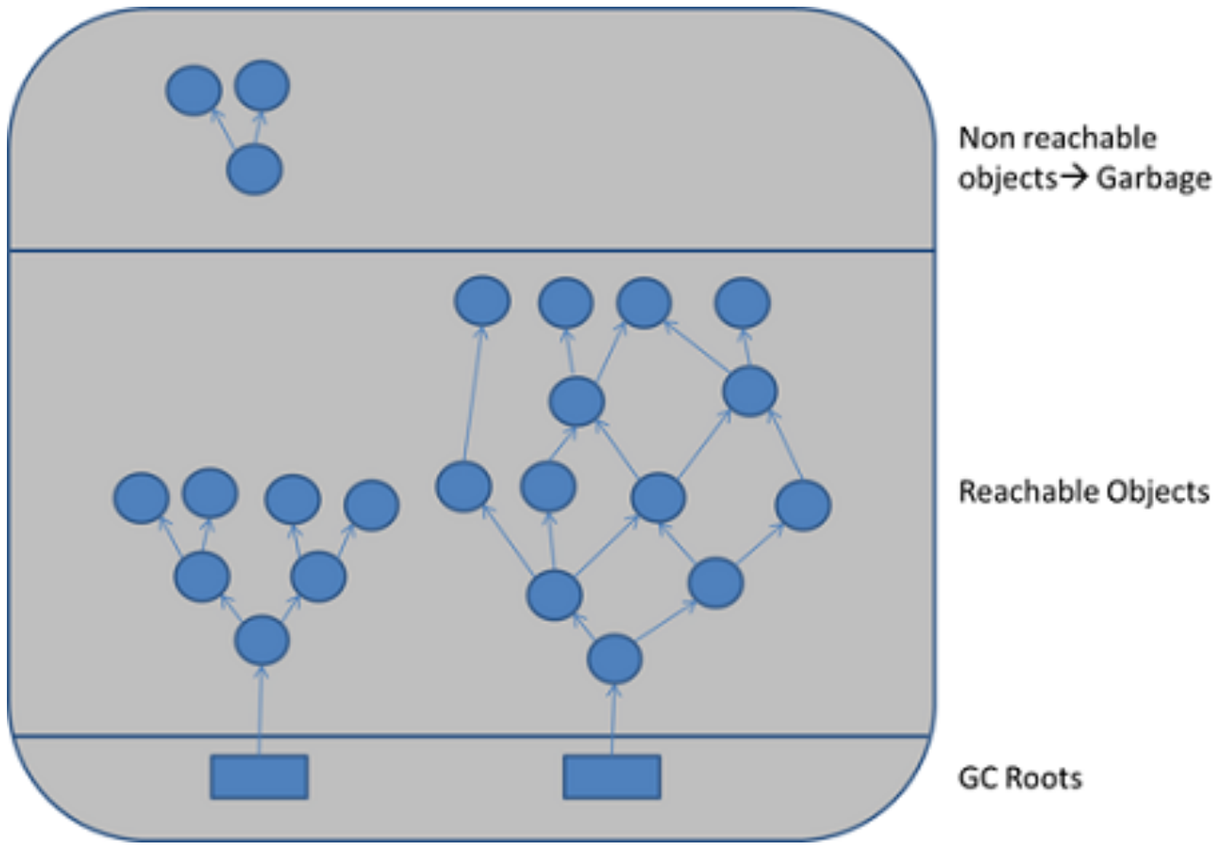


Object-oriented programming

- Constructor and destructor: Java
  - There are constructors
  - There are no destructor: The **garbage collector** is responsible for managing the heap memory



New objects are allocated at the end of the used heap



GC roots are special objects referenced by the JVM.
Non reachable objects are garbaged-collected

Object-oriented programming

- Inheritance: C++

```
class StudentUser : public User {
        public: StudentUser(string n, int a, string school) : User(n, a){
                schoolEnrolled = school;
        }
        void print() {
                User::print();
                cout << "School Enrolled: " << schoolEnrolled << endl;
        }
         string schoolEnrolled;
};
```

- Multiple inheritance: C++

```
class StudentUser : public User, public Student { ... };
```

Object-oriented programming

- Inheritance: Java

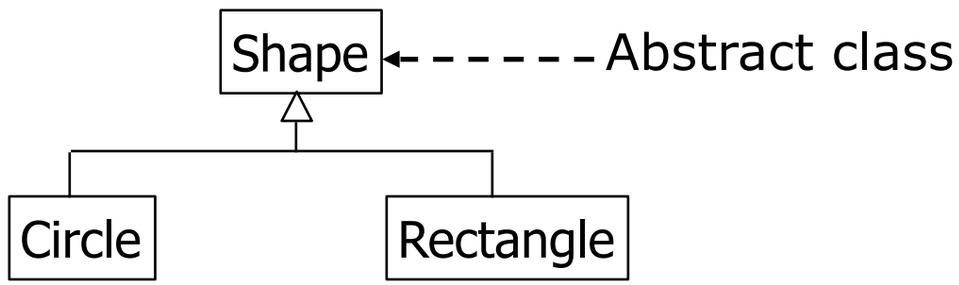
```
class StudentUser extends User {
   public StudentUser( String n, int a, String school ) {
         super(n, a);
         schoolEnrolled = school;
   }

   public void print() {
         super.print();
         System.out.print( " School: " + schoolEnrolled );
   }

   private String schoolEnrolled;
}
```

Object-oriented programming

- Abstract classes and interfaces
  - Java and C++ offer the abstract class concept



  - Additionally, Java offers the interface concept
    - An interface is similar a class abstract: no object can be created
    - An interface contains only the method declarations

Object-oriented programming

- Abstract class: C++

```
class Shape {
  public:
        virtual double area( ) = 0;
        virtual double circumference() = 0;
        ....
}
```

- Abstract class: Java

```
abstract class Shape {
        abstract public double area( );
        abstract public double circumference();
        ....
}
```

Object-oriented programming

- Java interface

```
interface MyInterface {
        public double area( );
        public double circumference();
        …
}

class MyClass implements MyInterface {
        // Implement the declared methods of MyInterface
}
```

- Multiple inheritance in Java

```
class MyClass extends SuperClass implements MyInterface1, MyInterface2 {
   // Implement the declared methods of MyInterface1 and MyInterface2
}
```

Object-oriented programming

- Polymorphism in C++

```
class User {
   string name;
   int age;
public:
   User(string nm, int a) {name=nm; age=a;}
   virtual  void print() {
       cout << "Name: " << name << " Age: " << age;
   }
};

class StudentUser : public User {
   string schoolEnrolled;
public:
   StudentUser(string name, int y, string school) : User(name, y) {
      schoolEnrolled = school;
   }
   void print() {
       User::print();
       cout << " School Enrolled: " << schoolEnrolled;
   }
};
```

Object-oriented programming

- Polymorphism in C++

```
int main()
{
        User* users[3];

        users[0] = new User( "Buster Dandy", 34 );
        users[1] = new StudentUser("Missy Showoff", 25, "Math");
        users[2] = new User( "Mister Meister", 28 );

        for (int i=0; i<3; i++)
        {
            users[i]->print();
            cout << endl;
        }

        delete [] users;
        return 0;
}
```

Object-oriented programming

- Polymorphism in Java

```
class User {
   public User( String str, int yy ) {
      name = str;
      age = yy;
   }
   public void print() {
       System.out.print( "name: " + name + " age: " + age );
   }

   private String name;
   private int age;
}

class StudentUser extends User {
   public StudentUser( String nam, int y, String sch ) {
       super(nam, y);
       schoolEnrolled = sch;
   }
   public void print() {
       super.print();
       System.out.print( " School: " + schoolEnrolled );
   }

    private String schoolEnrolled;
}
```

Object-oriented programming

- Polymorphism in Java

```
class Test
{
    public static void main( String[] args )
   {
          User[] users = new User [3];
          users[0] = new User( "Buster Dandy", 34 );
          users[1] = new StudentUser( "Missy Showoff",25, "Math");
          users[2] = new User( "Mister Meister", 28 );
          for (int i=0; i<3; i++)
          {
                users [i].print();
                System.out.println();
          }
    }
}
```

Main Activities of Software Development

Requirements Gathering

Define requirement specification

Analysis

Define the conceptual model

Design

Design the solution / software plan

Implementation

Code the system based on the design

Integration and Test

Prove that the system meets the requirements

Deployment

Installation and training

Maintenance

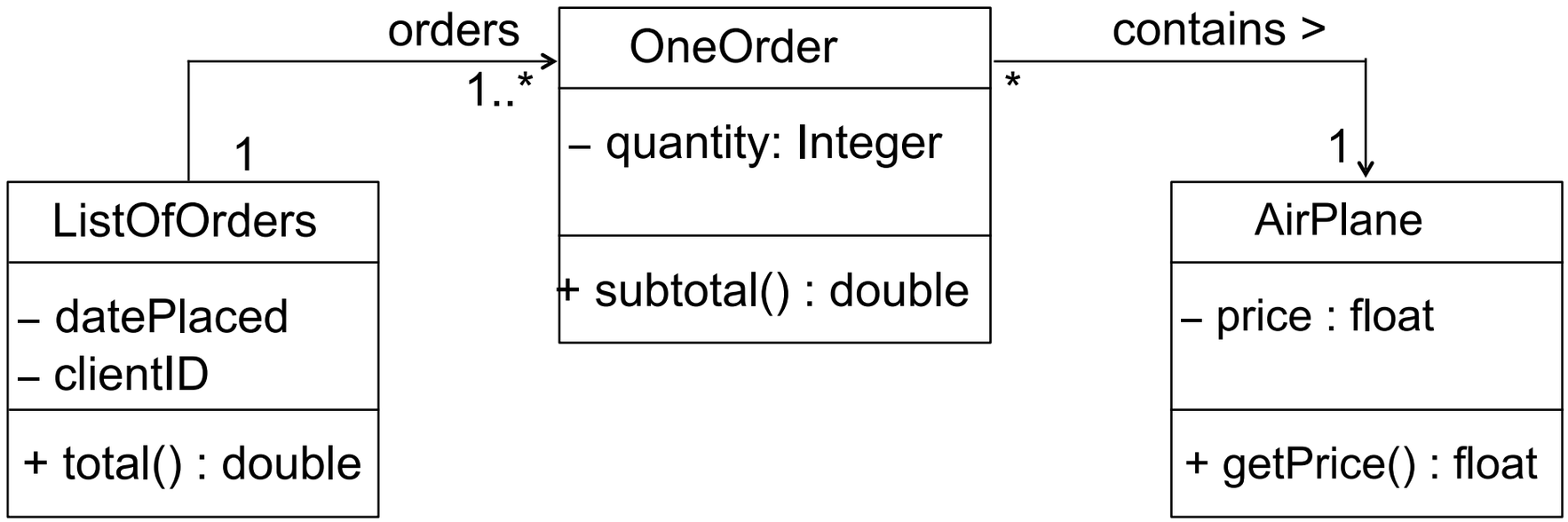
Post-install review
Support docs
Active support

From design to code

- Generation of source code from the design model
- Object-oriented code includes
  - Definitions of classes and interfaces
  - Definitions of methods
- The **class diagrams** are transformed to **classes** and **interfaces**
- The **interaction diagrams** are transformed to **code of methods**.
- Other diagrams allow to guide the programmer during coding

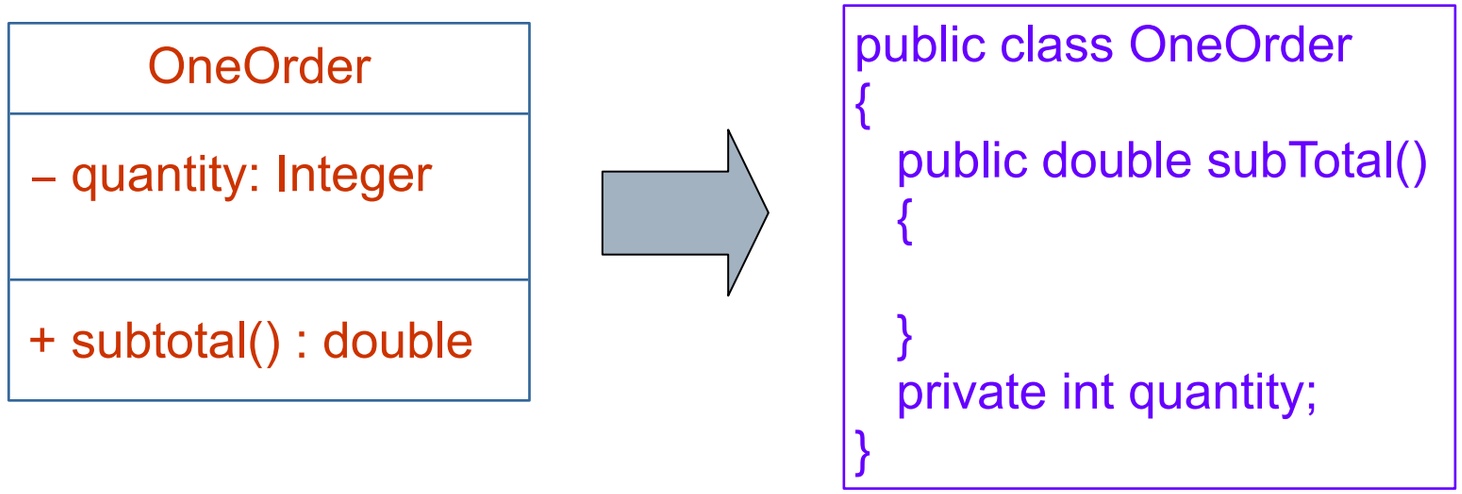
From design to code

- Class definition
  - Example of a part of class diagram



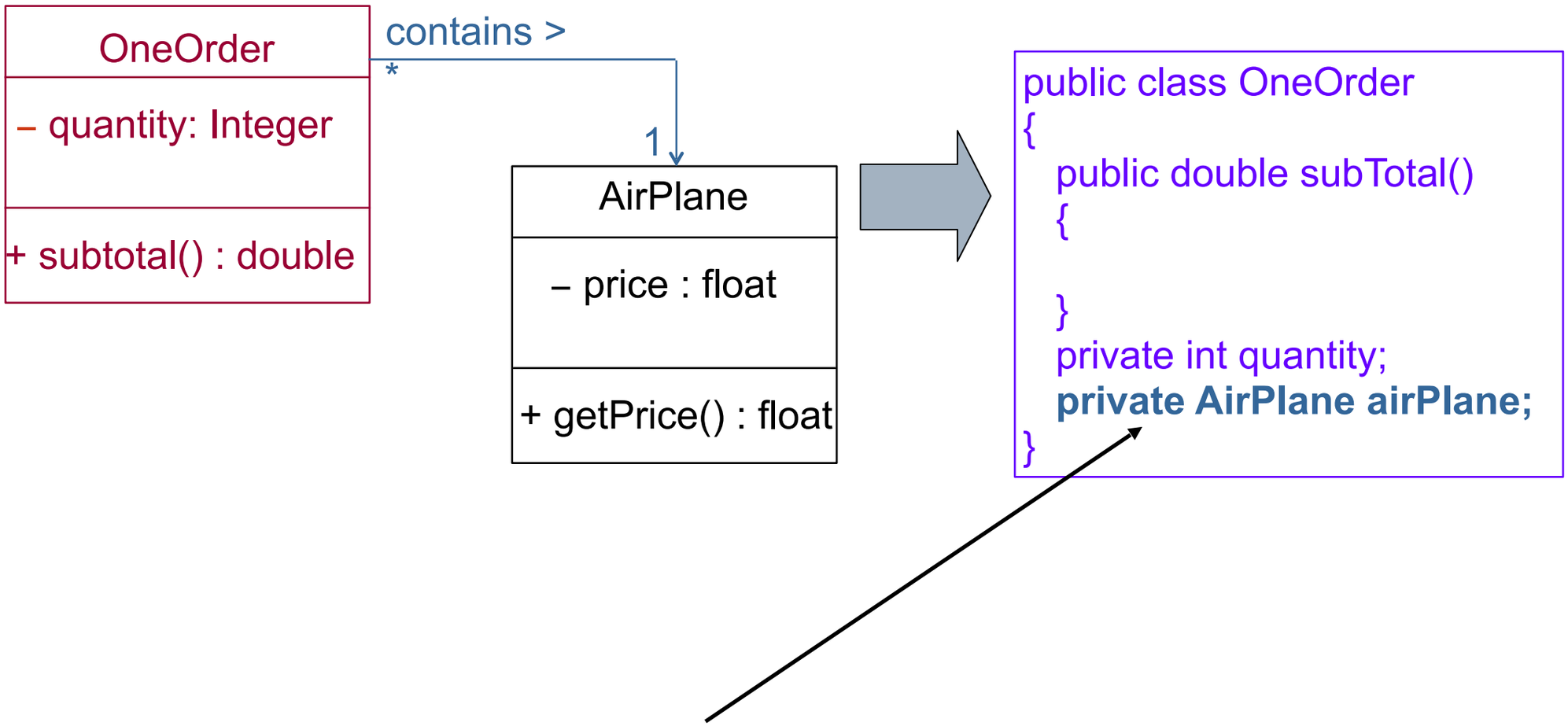
From design to code

- Class definition
  - Code of OneOrder class



From design to code

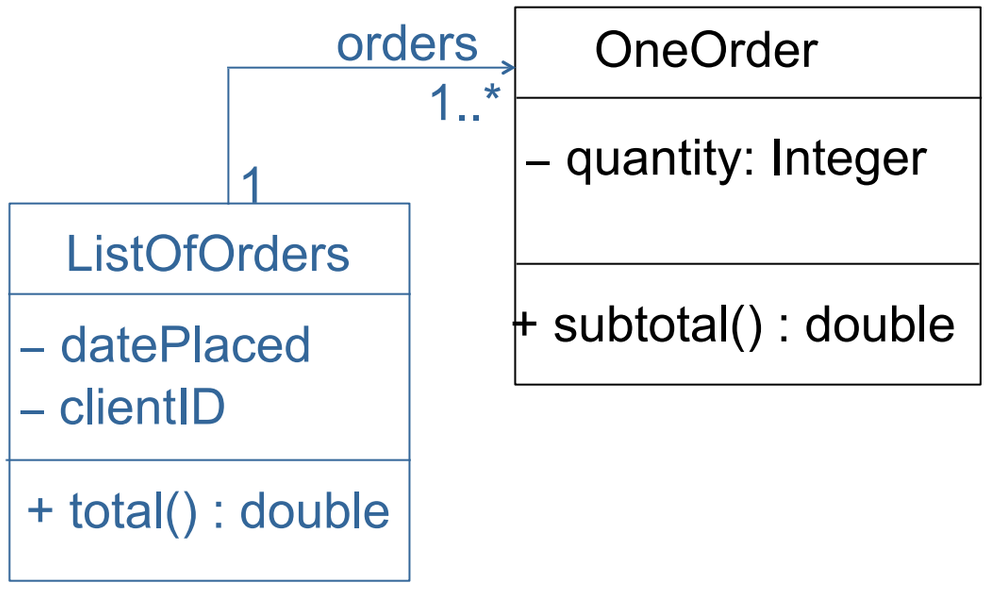
- Class definition
  - Code of OneOrder class



If the role of an association is not explicit, the created attribute takes the associated role.

From design to code

- Definition of classes
  - Code of ListOfOrders class

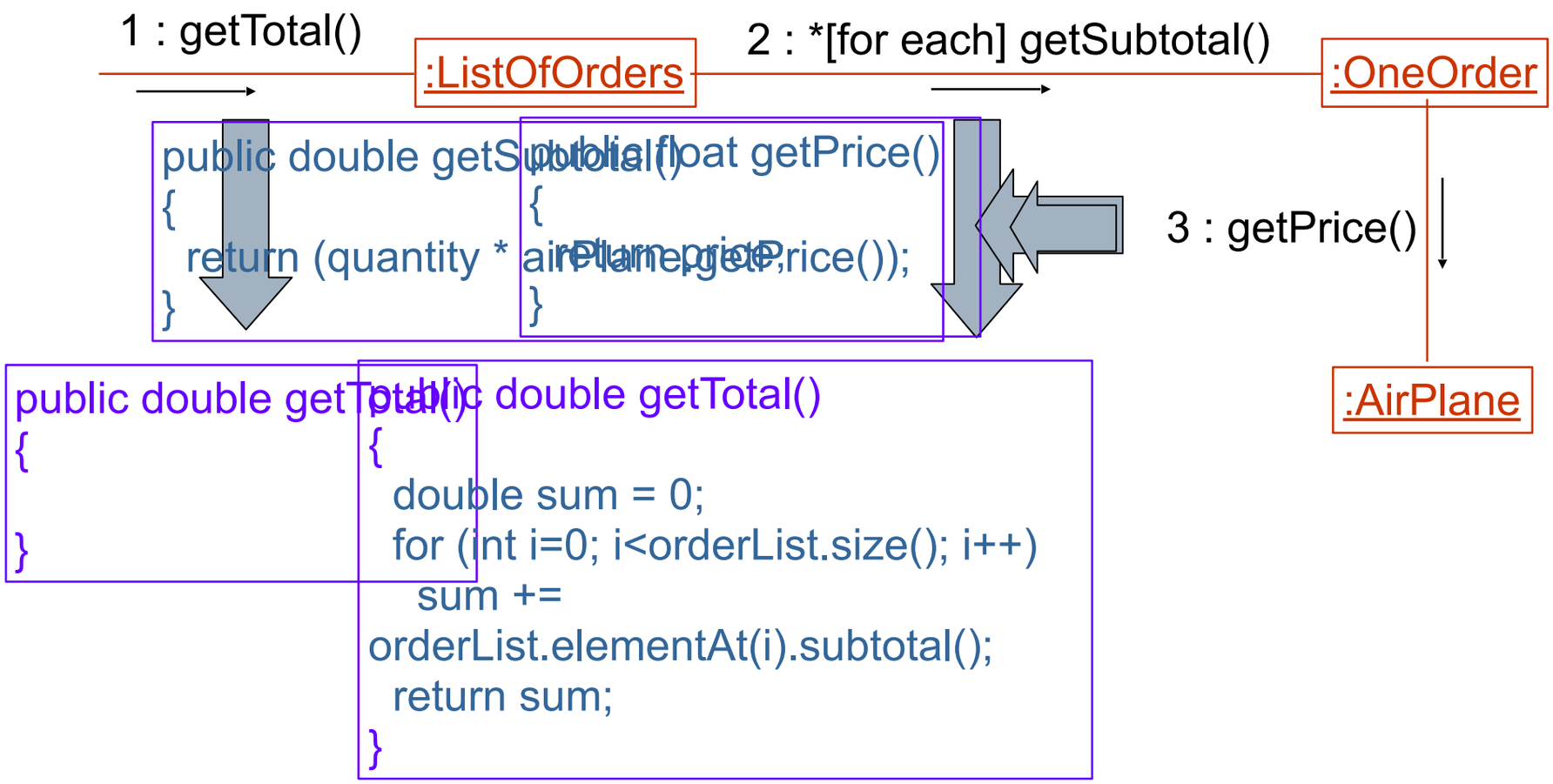



```
public class ListOfOrder
{
   public double total()
   {

   }
   private Date datePlaced;
   private int clientID;
   private List orders;
}
```

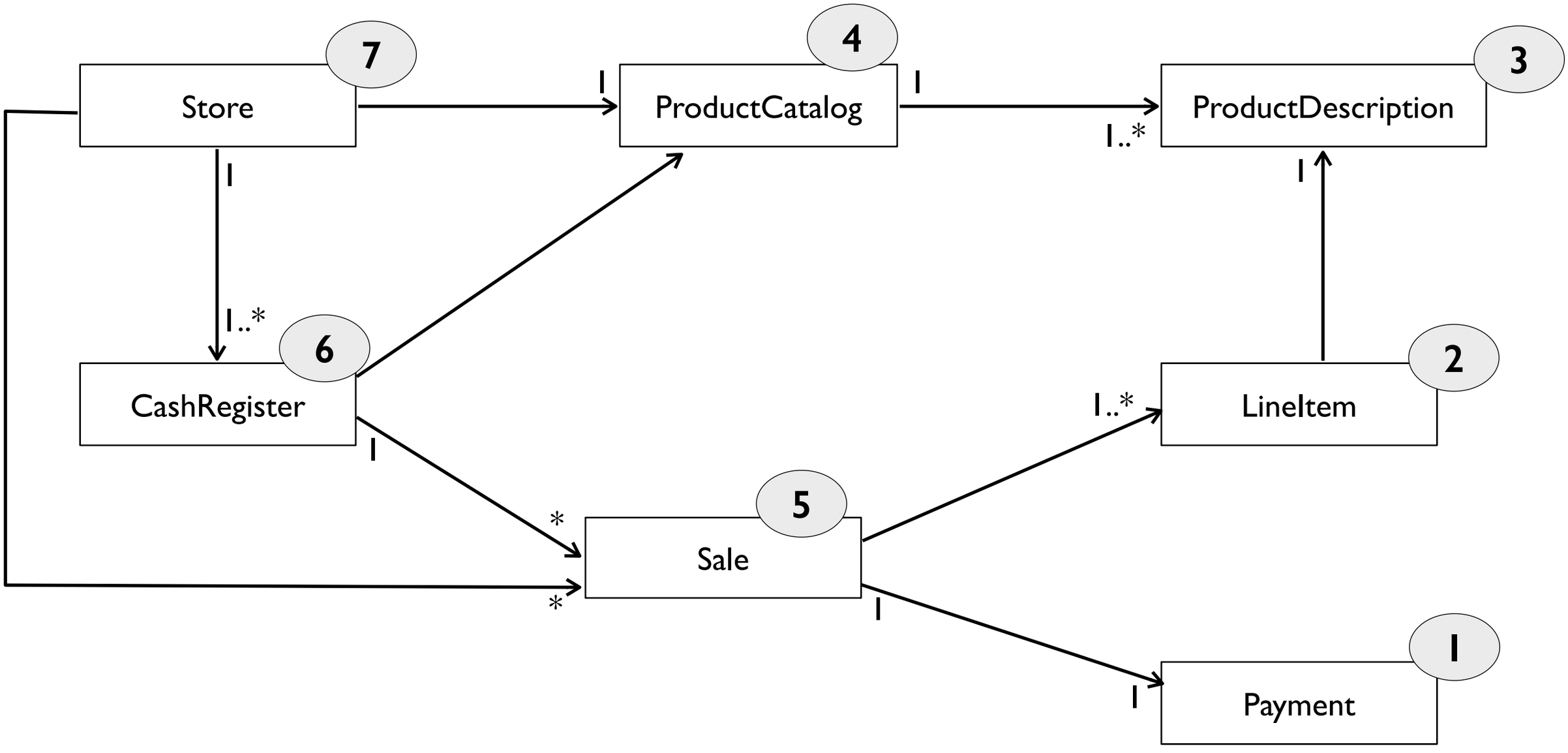
From design to code

- Method definition
  - Interaction diagram defines the **getTotal()** method



From design to code

- Implementation order
  - Class must be implemented from the least coupled/dependent to the most coupled/dependent



From design to code

- Several UML tools
  - Rational Rose, Dia ULM, Piosedon for UML, Umbrello, Power Design, Dia
    - Draw UML diagrams
    - Automatically generate source code: Java, C++, C#, …

- Automatically source code generation
  - Imperfect
  - Only the skeleton