Delsit

Object Oriented Analysis Design and Programming

Presenter :Lloyd Rixon

OOAD and OOPL Topics

- Inheritance
- Polymorphism
- Information Hiding
- Encapsulation
- Extensible Software Design

Copyright © Delsit 2006

Introduction

- Object modelling is about not separating processes from data. It is about:
- encapsulation: separating interfaces from implementation
- polymorphism: inheritance and pluggability.
- design by contract: constraints and rules.

Copyright © Delsit 2006

- OO principles must be consistently applied to object-oriented programming, object-oriented analysis, business process modelling, distributed object design and components.

Copyright © Delsit 2006

Inheritance

- A special kind of object-oriented association.
- Modeled as the generalization association.

Copyright © Delsit 2006

Reuse

When you need a new class you can:

Write the class completely from scratch.

Find an existing class that exactly match your requirements.

Built it from well-tested, well-documented existing classes.

- (A very typical reuse, called composition reuse!)

Reuse an existing class with inheritance

- Requires more knowledge than composition reuse.
- (it is what we are doing).

Copyright © Delsit 2006

Class Specialisation

In specialisation a class is considered an Abstract Data Type (ADT).

The ADT is defined as a set of coherent values on which a set of operations is defined.

A specialisation of a class C1 is a new class C2 where the instances of C2 are a subset of the instances of C1.

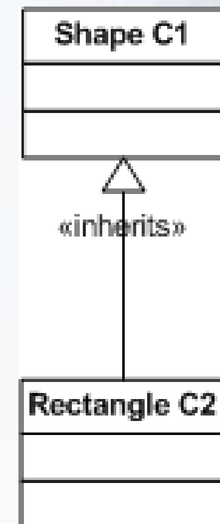
Operations defined of C1 are also defined on C2.

Operations defined on C1 can be redefined in C2.

Copyright © Delsit 2006

Extension (Is-A Relationship)

- The extension of a specialised class C2 is a subset of the extension of the general class C1.
- A C2 object is a C1 object (but not vice-versa).
- There is an "is-a" relationship between C1 and C2.



Copyright © Delsit 2006

Specialisation

«type»
Shape C1
-draw
-resize

«inherits»

«inherits»

«inherits»

«implementation class»
Rectangle C2

Circle C3

Line C4

«inherits»

Square :object is a run-time instance of the the Square Class C5

Note the inheritance via Generalisation

«implementation class»
Square C5

«implementation class»
:square : Square C5
draw
resize

Copyright © Delsit 2006

Class Extension

- In extension a class is considered a module.
- A module is a syntactical frame where a number of variables and method are defined, found in, e.g., PL/SQL.
- Extension is important in the context of reuse.
- Extension makes it possible for several modules to share code, i.e., avoid to have to copy code between modules.
- An extension of a class Rectangle C2 is a new class Square C5
- In C5 new properties (variables and methods) may be added.
- The properties of C2 are also properties of C5.

Copyright © Delsit 2006

Intension

- The intension of an extended class C5 is a superset of the intension of C2.

Copyright © Delsit 2006

Inheritance

Inheritance is a way to derive a new class from an existing class.

Inheritance can be used for:

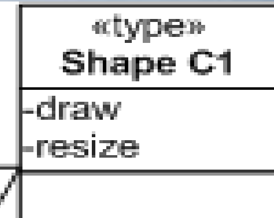
- Specialising an ADT.
- Extending an existing class.
- Often both specialisation and extension takes place when a class inherits from an existing class

Copyright © Delsit 2006

Composition and Inheritance

```
class Square extends Rectangle {
        //<class body>;
}
```

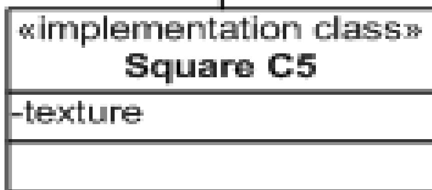
Superclass

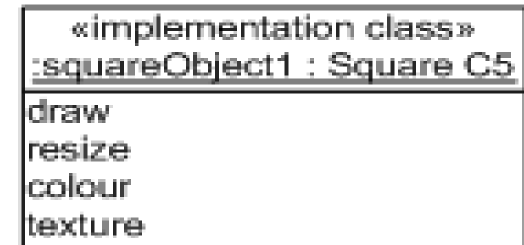


«inherits»

«inherits»

Subclass





Copyright © Delsit 2006

Inheritance Code example Java

```
public class Vehicle {
protected String make;
protected String model;
public Vehicle() {
        make = ""; model = "";
}
public String toString() {
return "Make: " + make + " Model: " + model;
}
}
public class Car extends Vehicle {
private double price;
public Car() {
super ();
price = 0.0;
}
public String toString() {
return "Make: " + make + " Model: " + model
+ " Price: " + price;
}
public double getPrice(){ return price; }
}
```

Copyright © Delsit 2006

Vehicle Specialisation and Class Extension

The Car type with respect to extension and intension:

Extension

- **Car is an extension of Vehicle.**
- **The intension of Car is increased with the variable price.**

Specialisation

- Car is an specialisation of Vehicle.
- The extension of Car is decreased compared to the class Vehicle.

Copyright © Delsit 2006

Inheritance and Constructors

- Constructors are not inherited.
- A constructor in a subclass must initialise variables in the class and variables in the superclass.
- *What about private fields in the super class?*
- It is possible to call the superclass' constructor in a subclass.

Copyright © Delsit 2006

Polymorphism (implemented by dynamic binding)

- The ability of a variable or argument to refer at run-time to instances of various classes.

```
Shape s = new Shape();
Circle c = new Circle();
Line l = new Line();
Rectangle r = new Rectangle();
s = l;
l = s;
```

The assignment s = l is legal if the static type of l is Shape or a subclass of Shape.

This is static type checking where the type comparison rules can be done at compile-time.

Copyright © Delsit 2006

Polymorphism - Dynamic Binding

```
class A {
        void doSomething(){
                ...
        }
}
```

```
class B extends A {
        void doSomething (){
                ...
        }
}
```

```
A x = new A();
B y = new B();
x = y;
x.doSomething(); // on class A or class B?
```

- **Dynamic binding is not possible without polymorphism.**

Copyright © Delsit 2006

Dynamic Binding - Example

```
class Shape {
        void draw() { System.out.println ("Shape"); }
}
class Circle extends Shape {
        void draw() { System.out.println ("Circle"); }
}
class Line extends Shape {
        void draw() { System.out.println ("Line"); }
}
class Rectangle extends Shape {
        void draw() {System.out.println ("Rectangle"); }
}
public static void main (String args[ ] ) {
         Shape[] s = new Shape[3];
         s[0] = new Circle();
         s[1] = new Line();
         s[2] = new Rectangle();
         for (int i = 0; i < s.length; i++){
                  s[i].draw(); // prints Circle, Line, Rectangle
        }
}
```

Copyright © Delsit 2006

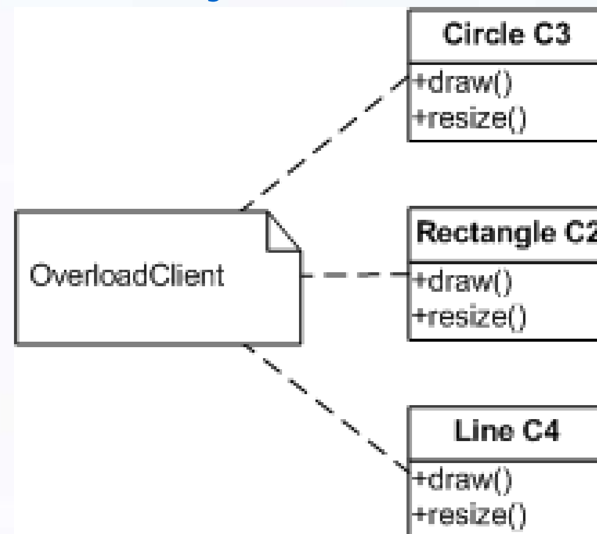
Why Polymorphism and Dynamic Binding?

- Separate interface from implementation.
  - (This is what we are trying to achieve in object-oriented design and programming)
- Allows programmers to isolate type specific details from the main part of the code.
- Code is simpler to write and to read.
- Can change types (and add new types) and this propagates through existing code.

Copyright © Delsit 2006

Overloading vs. Polymorphism

- Client has not yet discovered that the Circle, Line and Rectangle classes are related. (not very realistic but just to show the idea).



Copyright © Delsit 2006

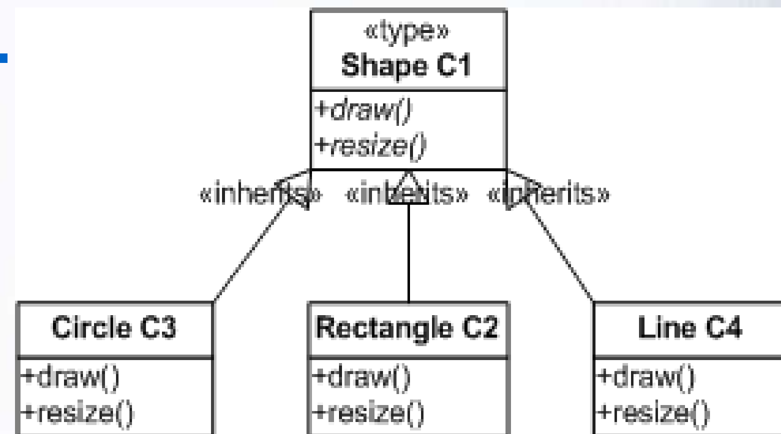
Usage code would look like this…

```
Class Circle {
      void draw() { System.out.println ("Circle"); }}
class Line {
      void draw() { System.out.println ("Line"); }}
class Rectangle {
      void draw() { System.out.println ("Rectangle"); }}
public class OverloadClient
{
       // make a flexible interface by overload and hard work
       public void doStuff(Circle c){ c.draw(); }
       public void doStuff(Line l){ l.draw(); }
       public void doStuff(Rectangle r){ r.draw(); }
       public static void main (String[] args){
             OverloadClient oc = new OverloadClient();
             Circle ci = new Circle();
             Line li = new Line();
             Rectangle re = new Rectangle();
             // shows encapsulation from client
             oc.doStuff (ci); oc.doStuff (li); oc.doStuff(re);
       }}
```

Copyright © Delsit 2006

Overloading vs. **Polymorphism**

- Discovered that the Circle, Line and Rectangle class are related are related via the general concept Shape
- Client only needs access to base class methods.



Copyright © Delsit 2006

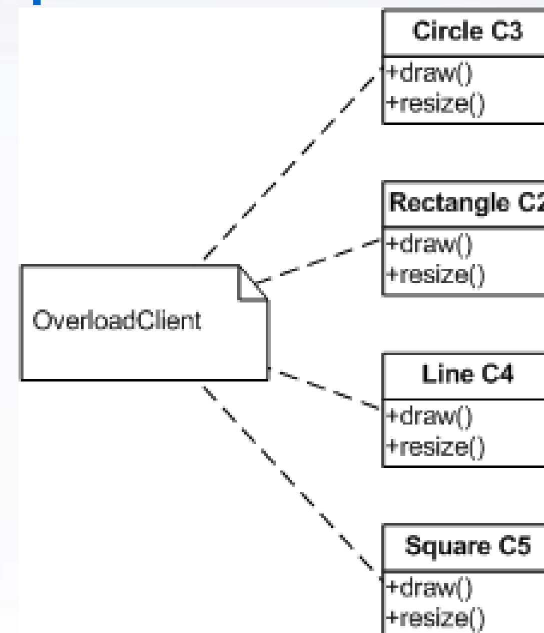
Inheritance code would look like this…

```
class Shape {
    void draw() { System.out.println ("Shape"); }}
class Circle extends Shape {
    void draw() { System.out.println ("Circle"); }}
class Line extends Shape {
    void draw() { System.out.println ("Line"); }}
class Rectangle extends Shape {
    void draw() { System.out.println ("Rectangle"); }}
public class PolymorphClient
{
    // make a really flexible interface by using polymorphism
    public void doStuff(Shape s){ s.draw(); }
    public static void main (String[ ] args){
        PolymorphClient pc = new PolymorphClient();
        Circle ci = new Circle();
        Line li = new Line();
        Rectangle re = new Rectangle();
        // still shows encapsulation from client
        pc.doStuff (ci); pc.doStuff (li); pc.doStuff(re);
    }
}
```

Copyright © Delsit 2006

Overloading vs. Polymorphism

- Must extend with a new class Square and the client has still not discovered that the Circle, Line, Rectangle, and Square classes are related.



Copyright © Delsit 2006

Adding Square Class by Overloading would look like this …

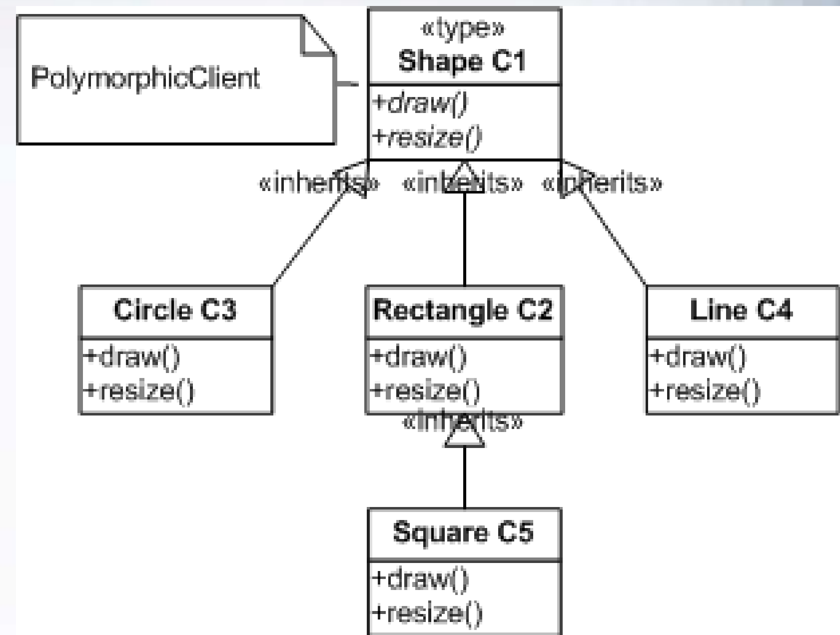
```
class Circle {
        void draw() { System.out.println ("Circle"); }}
class Line {
        void draw() { System.out.println ("Line"); }}
class Rectangle {
        void draw() { System.out.println ("Rectangle"); }}
class Square {
        void draw() { System.out.println ("Square"); }}

public class OverloadClient
{
         // make a flexible interface by overload and hard work
         public void doStuff(Circle c){ c.draw(); }
         public void doStuff(Line l){ l.draw(); }
         public void doStuff(Rectangle r){ r.draw(); }
         public void doStuff(Square s){ s.draw(); }
         public static void main (String[ ] args){
                 // encapsulation from client
                 oc.doStuff (ci); oc.doStuff (li); oc.doStuff(re);
         }
}
```

Copyright © Delsit 2006

Overloading vs. **Polymorphism**

- Must extend with a new class Square that is a subclass to Shape.



Copyright © Delsit 2006

Adding Square Class by Polymorphism would look like this …

```
class Shape {
      void draw() { System.out.println ("Shape"); }}
class Circle extends Shape {
      void draw() { System.out.println ("Circle"); }}
class Line extends Shape {
      void draw() { System.out.println ("Line"); }}
class Rectangle extends Shape {
      void draw() { System.out.println ("Rectangle"); }}
class Square extends Rectangle {
      void draw() { System.out.println ("Square"); }}
public class PolymorphClient
{
      // make a really flexible interface by using polymorphism
      public void doStuff(Shape s){ s.draw(); }
      public static void main (String[ ] args){
            // still encapsulation from client
            pc.doStuff (ci); pc.doStuff (li); pc.doStuff(re);
      }
}
```

Copyright © Delsit 2006

Opened/Closed Principle

- Open
- The class hierarchy can be extended with new specialized classes.
- Closed
  - The new classes added do not affect old clients.
  - The superclass interface of the new classes can be used by old clients.
- This is made possible via
  - Polymorphism
  - Dynamic binding

Copyright © Delsit 2006

Method Redefinition

- Redefinition: A method/variable in a subclass has the same as a method/variable in the superclass.
- Redefinition should change the implementation of a method, not its semantics.
- Redefinition in .Net (or Java) class B inherits from class A if
  - Method: Both versions of the method is available in instances of B. Can be accessed in B via **super.**
  - Variable: Both versions of the variable is available in instances of B. Can be accessed in B via **super.**

Copyright © Delsit 2006

The Final Keyword

- Final fields
  - Compile time constant
    - **final static double PI = 3.14**
  - Run-time constant
    - **final int RAND = (int) Math.random * 10**
- Final arguments
  - **double foo (final int i)**
- Final methods
  - Prevents overwriting in a subclass
  - Private methods are implicitly final
- Final class
  - Cannot inherit from the class

Copyright © Delsit 2006

Recap

- Designing good reusable classes is challenging!
- Reuse
  - Use composition when ever possible more flexible and easier to understand.
- Java and .Net support specialisation and extension via inheritance
  - Specialisation and extension can be combined.
- Polymorphism is a prerequisite for dynamic binding and central to the object-oriented programming paradigm.

Copyright © Delsit 2006

Questions