Herencia y Subtipos

Universidad de Chile

Departamento de Ciencias de la Computación

Prof.: Nancy Hitschfeld Kahler

Contenido

- Herencia, polimorfismo y enlace dinámico
- Tipos y subtipos
- Contraste entre herencia y subtipos
- Herencia entre clases
- Herencia múltiple

Herencia, polimorfismo y enlace dinámico

Un lenguaje es orientado a objetos si posee:

- Herencia
- Polimorfismo
- Enlace dinámico

Aplicación:

- Dibujar figuras
- Figuras: círculos, rectángulos, triángulos, etc
- Operaciones: mover, donde, rotar, dibujar

Herencia, polimorfismo y enlace dinámico

Usando enfoque sin herencia: Proposición 1:

```
enum Forma{TRIANGULO, RECTANGULO, CIRCULO}
class Figura{
        Punto centro;
        Color color;
        Forma  tipo;
        int radio, ancho, alto;
        Punto vertices[3];
   public:
      Figura(Punto p, Color c, int _ancho, int _alto);
      Figura(Punto p, Color c, int _radio);
      Punto donde();
      void mover(Punto hacia);
      void dibujar();
      void rotar(float angulo);
   }
```

```
void Figura::dibujar(){
  switch(tipo){
      case CIRCULO:
       //dibujar circulo
      case RECTANGULO:
      //dibujar rectangulo
      case TRIANGULO:
     // dibujar  triangulo
  }
}
```

Herencia, polimorfismo y enlace dinámico

Ventajas:

- Reusa métodos e implementaciones comunes

Desventajas:

- Métodos como Dibujar deben conocer todas las figuras
- Agregar una nueva figura modifica todos los métodos como Dibujar
- Agregar una nueva figura requiere del código fuente
- Posibilidad de introducir errores al código antiguo
- Almacenamiento conjunto de características particulares

¿Cuál es el problema?

No se puede hacer distinción entre las propiedades generales y particulares de una figura

Herencia, polimorfismo y enlace dinámico

Usando enfoque sin herencia: Proposición 2:

```
class Circulo{
      Punto centro;
      Color color;
      int radio;

   public:
       Circulo(Punto p, Color c, int r);
        Punto donde();

        void mover(Punto hacia);
        void dibujar();
        void rotar(float angulo);
   }
```

```
class Rectangulo{
      Punto centro;
      Color color;
      int  ancho;
       int alto;
   public:
      Rectangulo(Punto p, Color c,
                    int _ancho, int _alto);

      void  mover(Punto hacia);
      Punto donde();
       void dibujar();
       void rotar(float angulo);
  }
```

Herencia, polimorfismo y enlace dinámico

Ventajas:

- Permite separar propiedades particulares
- Métodos como Dibujar conoce solo las propiedades de la figura que debe dibujar

Desventajas:

- Propiedades comunes deben ser implementadas en cada clase

¿Cuál es el problema?

No se puede hacer uso eficiente de las propiedades generales y particulares de una figura al mismo tiempo

Herencia, polimorfismo y enlace dinámico

¿Qué caracteriza la programación orientada a objetos?

- La capacidad de expresar aspectos comunes y las diferencias, y sacar ventajas de ello
- Propiedades comunes: centro, color, donde, mover, dibujar(?) y rotar(?)
- Propiedades particulares:
  - Triángulo: tres vértices
  - Rectángulo: ancho, alto
  - Círculo: radio

Ejemplo en c++

Implementación usando orientación a objetos

```
class  Figura{
  protected:
       Punto centro;
       Color color;
  public:
      Figura(){// valores default}
      Figura(Punto p, Color c){ centro=p;  color = c;}
      Punto donde() {return centro;}
      void  mover (Punto hacia){centro=hacia; dibujar();}
      virtual void dibujar()=0;
      virtual void  rotar(float angulo)=0;
}
```

Ejemplo en c++

```
class Circulo: public Figura{
    int radio;
  public:
    Circulo(){radio=1;}
    Circulo(int r){radio=r;}
    Circulo(Punto p, Color c, int r): Figura(p,c)  {radio=r;}
     void dibujar(){//dibujar un circulo}
     void rotar(float angulo){//Rotar un circulo}
}
```

Ejemplo en c++

```
class Rectangulo: public Figura{
        int ancho, alto;
    public :
        Rectangulo(int _ancho, int _alto) {
                        ancho=_ancho; alto=_alto; }
        Rectangulo(Color c, Punto p,
                        int _ancho, int _alto):Figura(p,c){
              ancho =_ancho; alto = _alto;}
        void dibujar(){//Dibujar un rectangulo}
        void rotar(float angulo){//Rotar un rectangulo}
   }
```

Ejemplo en c++

Ejemplo 1 :

```
Color  rojo(1,0,0);
Color  verde(0,1,0);
Punto  centro_circulo(1,1);
Punto centro_rectangulo(1,0);



Rectangulo  rectangulo(
   centro_rectangulo,verde, 0.4,0.3);

rectangulo.mover(origen);
Figura* figura= new Circulo(centro_circulo,
                                    verde, 0.4);
 figura->dibujar();
```

Ejemplo 2:

```
Stack <Figura*> stack_de_figuras(20);

Figura* figura =
            new Circulo(centro_circulo,
                              verde, 0.8);
stack_de_figuras.push(figura);
figura = new Rectangulo(
             centro_circulo, azul,0.8,0.6);
stack_de_figuras.push(figura);
. . .
while( !stack_de_figuras.empty() ){
     figura = stack_de_figuras.top();
     figura->dibujar();
     stack_de_figuras.pop();
}
```

Ejemplo en Java

```
class Punto{

    private int x;
    private int y;

    public Punto(int _x, int _y){
         x = _x;
        y = _y;
    }

    public void asignar(int _x, int _y){
        x = _x;
        y = _y;
    }
   public int  obtenerX (){
       return x;
    }

    public int obtenerY(){
       return y;
    }
};
```

Ejemplo en Java

```
import java.awt.*;


abstract class Figura{

   protected Punto centro;
   protected Color color;

    public Figura(punto p, Color c){
        centro = new Punto(p.obtenerX(), p.obtenerY());
        color = new Color(c.getRed(), c.getGreen(), c.getBlue());
    }

    public void mover(Graphics g, Punto p){
        centro.asignar(p.obtenerX(), p.obtenerY());
        Dibujar(g);
    }

   public Punto donde(){ return centro;}

    abstract public void dibujar(Graphics g);
}
```

Ejemplo en Java

```
import java.awt.*;

class Rectangulo extends Figura{
    protected int ancho;
    protected int alto;

    public Rectangulo(Punto p, Color c, int _ancho, int _alto){
         super(p,c);
        ancho = _ancho;
        alto = _alto;
    }
    public void dibujar(Graphics g){
        g.setColor(color);
        g.drawRect(centro.obtenerX()-ancho/2, centro.obtenerY()-alto/2,
                   ancho, alto);

    }
}
```

Ejemplo en Java

```
import java.awt.*;


class Circulo extends Figura{
    protected float radio;

     public Circulo(Punto p, Color c, int _radio){
         super(p,c);
        radio = _radio;
    }
    public void dibujar(Graphics g){
        g.setColor(color);
        g.drawOval(centro.obtenerX()- radio/2,
                   centro.obtenerY()- radio/2, 2*radio, 2*radio);

    }
}
```

Ejemplo en Java

```
import java.awt.*;
public class Dibujo extends VisibleFrame{
    private int numero_figuras;
    private Figura[] figuras;

   public void paint(Graphics g){
        for(int i=0; i < numero_figuras; i++)
            figuras[i].dibujar(g);
  }
  public Dibujo(){
     figuras = new Figura[100];
     Punto centro = new Punto(20,40);
     Color color = new Color(255,0,0);
     figuras[0] = new Circulo(centro, color, 20);
```

Ejemplo en Java

```
        figuras[1] = new Rectangulo(new Punto(280,40), color, 10,40);

        figuras[2] = new Circulo(centro, new Color(0,150,255), 20);

        figuras[3] = new Rectangulo(new Punto(150,100),
                                        New Color(255,0,255), 50, 50);
         numero_figuras = 4;
   }


}
```

Ejemplo en Java

```
import java.awt.*;
public class Main_dibujo {

   public static void main(String[] args) {

         VisibleFrame mi_ventana = new Dibujo();
         mi_ventana.setBackground(Color.green);
         mi_ventana.setTitle(
                  mi_ventana.getClass().getName());
         mi_ventana.show();
    }
}

import java.awt.*;
class VisibleFrame extends Frame{
     public visibleFrame(){
          setSize(300,200); }
 }
```

Herencia, polimorfismo y enlace dinámico

Fijando conceptos:

- Herencia
  - compartir variables de instancia (clase base)
  - compartir métodos (clase base)
  - redefinir métodos (clase derivada)
  - agregar nuevas variables de instancias (clase derivada)
  - agregar nuevos métodos (clase derivada)
- Polimorfismo: ocultar distintas implementaciones bajo el mismo nombre
  - Antes: overloading y templates
  - Ahora: a través de la herencia

Herencia, polimorfismo y enlace dinámico

Mecanismos de llamadas

- Enlace dinámico: mecanismo que permite seleccionar la implementación correcta en tiempo de ejecución
- Llamada a un método cualquiera: en tiempo de compilación
  - Ej: figura→mover(); Punto p = figura→donde(); // c++
  - figura.mover(); Punto p = figura.donde(); // java
- Llamada a un método redefinido: en tiempo de ejecución
  - Ej: figura→dibujar(); // c++
  - figura.dibujar(); // java

Herencia, polimorfismo y enlace dinámico

Aspectos importantes en C++

- Control de lo que se hereda
  - public, protected, (private)
- Orden de llamada constructores: bottom-up
- Orden de llamada destructores: top-down
- Definir destructores virtuales en caso de herencia

Herencia, polimorfismo y enlace dinámico

Aspectos importantes en java

- Control de lo que se hereda
  - public, protected, (private)
- Orden de llamada constructores: bottom-up
- Destrucción de objetos a través de recolector de basura