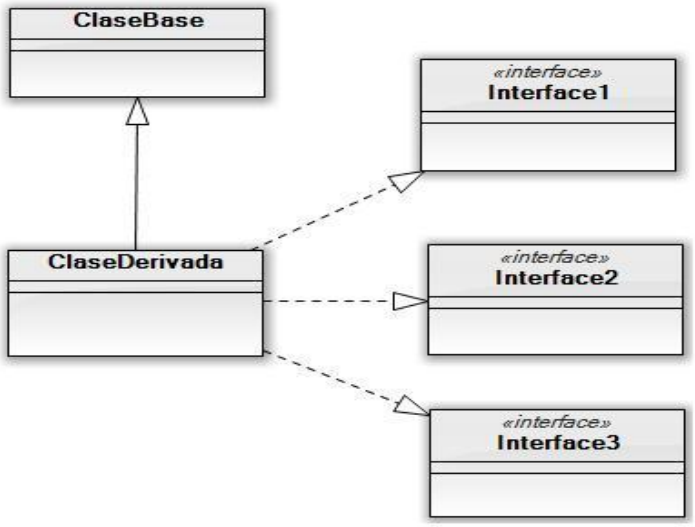
UNIVERSIDAD DISTRITAL
FRANCISCO JOSÉ DE CALDAS



Herencia de interfaz.

2013
Transversal Programación Orientada a Objetos
Proyecto Curricular de Ingeniería de Sistemas

Introducción

Ante la herencia múltiple alguna solución no controversial tenía que hacer… la herencia de interfaz.

Al darse diversos problemas con respecto a la herencia multiple, varios lenguajes optaron por implementar mecanismos de acción internos en caso de colisión, en el caso de Java, se planteó el uso de la interfaz para ese propósito.

1. "Herencia múltiple" y Java

La interfaz es básicamente una clase abstracta pura, sus métodos son todos abstractos y no maneja atributos generalmente, de hecho no tiene implementación (no asocia un espacio de almacenamiento), así que se pueden combinar varias interfaces cosa que no permiten las clases usuales, si se experimenta Java no permite que en la firma de una clase se pongan dos clases después de un extends pero si de los implements , cuantas se deseen, dando la facilidad de usar las interfaces que requiera. En este sentido es como si cada clase base que se desee manifestar en la herencia múltiple sea una interfaz. Claro está que se puede usar una clase concreta con extends y las demás con implements (interfaces).

Ejemplo:

```
interface SerPadre {
       void criar();
      }
interface SerEsposo {
       void amar();
      }
interface SerEmpleado {
       void trabajar();
      }
class EmpleadoDeOracle {
       public void trabajar(){}
      }
class Programador extends EmpleadoDeOracle implements serPadre,
serEsposo, serEmpleado{
      public void criar(){}
      public void amar(){}
      }
```

Como se observa se implementaron varias interfaces y Programador está comprometido a implementar los métodos de las interfaces, sin embargo, serEmpleado y EmpleadoDeOracle tienen el mismo método trabajar(), así que no se proporciona explícitamente trabajar() porque lo toma de la clase EmpleadoDeOracle, esto quiere decir que cuando se cree un objeto Programador, este tomará automáticamente ese método de EmpleadoDeOracle.

Ahora veamos cómo un objeto de Programador puede tomar comportamiento de cualquier interfaz o la clase.

```
public class DiaDeVida {
  static void comprobarA(SerPadre x){ x.criar();}
  static void comprobarB(SerEsposo x){x.amar();}
  static void comprobarC(SerEmpleado x){x.trabajar();}
  static void comprobarD(EmpleadoDeOracle x){x.trabajar();}
  public static void main (String [] args){
      Programador juan=new Programador();
      comprobarA(juan);// Se trata como un SerPadre
      comprobarB(juan);// Se trata como un SerEsposo
      comprobarC(juan);// Se trata como un SerEmpleado
      comprobarD(juan);// Se trata como un EmpleadoDeOracle
  }
}
```

Aquí se observa una de las labores principales de las interfaces poder hacer conversión hacia arriba a más de un tipo de clase base.

2. Colisiones

Se observó en el ejemplo anterior que el método trabajar() entre las clase EmpleadoDeOracle y la interface SerEmpleado, era igual, esto no generó problema, pero puede que no siempre se dé este caso y se de una colisión.

```
interface A {
   void f ();
}
interface B {
   int f (int i);
}
interface C {
   int f();
}
class D {
      public int f () {
       return 1;
      }
 }
class D2 implements A, C {
      public void f () { }
      public int f (int i) { return 1; } // sobrecargado
}
class D3 extends D implements B {
      public int f (int i) { return 1; } // sobrecargado
}
class D4 extends D implements C {
// Idéntica, sin problemas :
public int f () { return 1; }
}
```

Los anteriores fragmentos se pueden llevar a cabo sin ningún problema, en vista de que se presenta una sobrecarga de los métodos.

Si analizamos la clase D2, el implementa el método f() sin problema puesto que son iguales los métodos de la interface A y C.

En la clase D3 tampoco hay problema debido a que coincide el tipo de retorno (int).

```
//Los métodos sólo difieren en el tipo de retorno:
class D5 extends D implements A {}
 interface F extends A, C {}
```

Este fragmento si presentará problema, en vista de que el método de D y el método de A tienen diversos tipos de retorno (void e int).

Produciendo así el siguiente error.

```
ColisionInterfaces . java: 23 : f () in C cannot
implement f () in 11; attempting to use
incompatible return type
found : int
required: void
ColisionInterfaces.java:24: interfaces C and A are
incompatible; both define f
(), but with different return type
```

2. Herencia con interfaces

Las interfaces también pueden heredar de otras interfaces, consiguiendo así una nueva interfaz, se emplea el extends habitual.

Teniendo así que la clase que implementa a esa nueva interfaz recibirá los métodos tanto de la interfaz base como la derivada.

```
interface Galleta {
   void crearMasa ();
   void darForma ();
   void cocinar ();
}
interface GalletaConChispas extends Galleta{
   void anadirChispas();
}
class Cena implements GalletaConChispas{
   void crearMasa ();
   void darForma ();
   void cocinar ();
   void anadirChispas();
```

Bibliografía

- Eckel, Bruce. Thinking in Java. Prentice Hall. 1998. Estados Unidos.