Programación orientada a objetos - Herencia

Estructura de Datos/Algoritmos I

Depto. de computación-UNRC
Prof. Pablo Castro

Desarrollo Orientado a Objetos

La orientación a objetos es una metodología de diseño de software basada en la noción de TAD:

- Se utilizan módulos para encapsular datos y comportamiento.
- Se combina la noción de modulo con la noción de tipos.
- Se combina la estructura de tipos (subtipos) con la noción de extensión de módulos.

Bien utilizada, la programación orientada a objetos puede llevar a la construcción de software de calidad.

CLASES

Es el concepto central de orientación a objetos:

- Cada clase corresponde a un módulo.
- Cada clase define un tipo.
- A diferencia de los lenguajes procedurales, las clases encapsulan tanto comportamiento como datos.

En los llamados lenguajes OO puros, cada tipo corresponde a una clase (esto no pasa en JAVA).

SISTEMAS OO

Recordar que:

> Un sistema OO es una colección de clases, en la cual una de ellas es conocida como la clase raíz.

Estructura de los sistemas OO:

- La ejecución empieza por la clase raíz.
- Primero se crea una instancia de la clase raíz, y se ejecutan sus rutinas de creación.
- La clase raíz tiene una sola instancia.

RELACIONES ENTRE CLASES

En programación orientada a objetos existen dos tipos de relaciones entre clases:

- Una clase puede ser *cliente* de otra (relación de uso).
- Una clase puede extender o especializar a otra (relación de herencia).

El segundo tipo de relación es la *herencia*:

- Si las clases se ven como módulos, la relación de herencia corresponde a una extensión.

- Si las clases se ven como tipos, la herencia corresponde a la noción de subtipo.

HERENCIA

Podemos hacer que una clase herede de otra por medio de la palabra reservada *extends:*

```
/**
* Simple example of inheritance
* @author Pablo Castro
**/
public class SwappablePair extends Pair{

   /**
   * Change X for Y and viceversa
   * @precondition x=X and y =Y
   * @postcondition x=Y and y=X
   **/
   public void swap(){
    int temp = fst();
    changeFst(snd());
    changeSnd(temp);
   }

}
```

Agrega nueva funcionalidad a la clase permitiendo intercambiar x e y

SwappablePair es un subtipo de Pair, es decir, a cualquier variable de tipo Pair le podemos asignar un SwappablePair

HERENCIA COMO EXTENSIÓN DE TIPOS

Al definir una clase Sub como subclase de una clase Super, indicamos que Sub es un subtipo de Super: cualquier instancia de Sub es una instancia de Super.

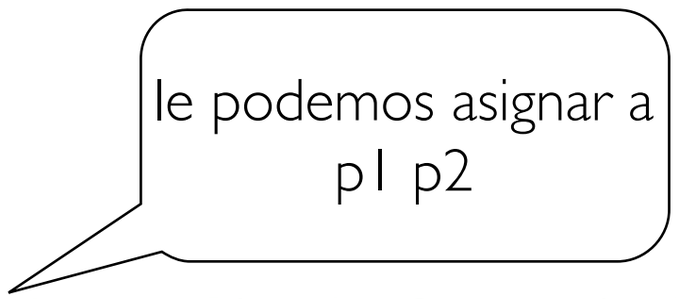
```
public boolean isZero(Pair p){
    return (p.fst() == 0) && (p.snd() == 0)
}

...

Pair p1;
Pair p2;
boolean b1, b2;

...

b1 = isZero(p1);
b2 = isZero(p2);
p1 = p2; // define a p1 como alias de p2 y se pierde la referencia original
```



HERENCIA Y POLIMORFISMO

La herencia está fuertemente relacionada con la noción de polimorfismo:

> Un método que toma como parámetro un objeto de tipo X es polimórfico, en el sentido que puede aplicarse a cualquier objeto de tipo Y, con Y subclase de X

Ejemplo: La función esZero() puede aplicarse a instancias de SwappablePair, y a cualquier subclase de esta.

REDEFINICIÓN DE MÉTODOS

Cuando declaramos una subclase podemos redefinir algunos de los métodos de la superclase.

Los lenguajes orientados a objetos poseen un mecanismo para invocar al método correcto en tiempo de ejecución, llamado *dynamic dispatch*

Dynamic dispatch permite utilizar los subtipos para proveer diferente funcionalidades.

EJEMPLO

```java
/**
* A simple class for pairs with undo
* @author Pablo Castro
*/

public class MemoryPair extends Pair{

    private int previousX; // the previous value of x
    private int previousY; // the previous value of y


    /**
    * It changes the x
    * @param v  the new value
    */
    public void changeFst(int v){
        previousX = fst();
        super.changeFst(v);
    }

    /**
    * It changes the y
    * @param v  the new value
    */
    public void changeSnd(int v){
        previousY = fst();
        super.changeSnd(v);
    }

    /**
    *   It recovers the old values of x and y
    */
    public void recover(){
        changeFst(previousX);
        changeSnd(previousY);
    }

}// end of class
```

```java
Pair p;
MemoryPair mp;
p.changeFst(4); // invoca la version original
p.changeFst(10); // invoca la version redefinida
```

Dynamic dispatching

El método changeFst está redefinido

Nuevo método para recuperar los valores anteriores

HERENCIA VS AGREGACIÓN

Además de usar herencia uno puede usar agregación para extender la funcionalidad de una clase.

```
public class SwappablePair{
    private Pair p;

    ...

    public void changeFst(int v){
        p.changeFst(v);
    }

    ...

    public void swap(){
        int aux = p.snd();
        p.changeSnd(p.fst());
        p.changeFst(aux);
    }

} // end of class
```

Le agregamos un atributo de tipo pair y de esta forma incorporamos su funcionalidad

En este caso no tenemos un relación de subtipos.

CLASES ABSTRACTAS

Con el objeto de explotar el polimorfismo, los lenguajes OO incorporan la noción de clase abstracta.

Una clase se dice abstracta si no posee instancias, se utilizan para definir interfaces que sus subclases deben implementar

En Java las clases abstractas se definen con la palabra *abstract*

El lenguaje Java provee la clase abstract Object, que cualquier clase extiende. Define las funcionalidades básicas de cualquier objeto

INTERFACES

Qué son las interfaces?

- Las interfaces son parecidas a las clases abstractas.
- A diferencia de las clases abstractas, todos sus métodos son abstractos.
- Las interfaces permiten herencia múltiple entre clases.

Si estamos caracterizando TADs abstractamente, podemos definir una interface con sus métodos

IMPLEMENTACIÓN DE TADS MEDIANTE OO

- Podemos aprovechar las interfaces para la especificación de TADs.

- Podemos aprovechar la herencia para proveer varias implementaciones de cada TAD.

- Se hace posible el uso de varias implementaciones diferentes para cada TAD.

- Se minimiza los cambios necesarios a realizar cuando un cliente cambia de implementación.

IMPLEMENTACIÓN DE TADS

Supongamos que queremos implementar un TAD X en un lenguaje orientado a objetos.

- Utilizamos una clase abstracta X para especificar las operaciones del TAD.
- Utilizamos clases concretas Imp1, Imp2, Imp3, que hereden de X para cada implementación del TAD

El lenguaje se encargará por medio de dynamic dispatching de invocar al método correspondiente para cada implementación. Además podemos aprovechar la herencia para definir el TAD polimorfico.

EJEMPLO: LISTAS

- Elementos: Listas finitas de un tipo C, un extremo de la lista se reconoce como su comienzo.

- Operaciones:
  - esVacia: dice si la lista es vacía.
  - vaciar: borra todos los elementos de la lista.
  - longitud: devuelve la longitud de la lista.
  - insertar: dado un elemento y una posición, inserta el elemento en esa posición.

TAD LISTA (CONT.)

- Eliminar: dada una posición de la lista elimina el elemento en esa posición.
- Obtener, dado una posición, elimina el elemento en esa posición.

Utilizaremos una interface para especificar el TAD lista y sus operaciones a un nivel abstracto, es decir, independiente de su implementaciones.

INTERFACE LISTA

```
public interface Lista {

/* esVacia(): Indica si la lista es vacia o no */
/* pre: true */
/* post: Retorna true ssi la lista no tiene elementos */
public abstract boolean esVacia();

/* vaciar(): Elimina todos los elementos de la lista */
/* pre: true */
/* post: elimina todos los elementos de la lista. */
public abstract void vaciar();

/* longitud(): Retorna la cantidad de elementos de la lista */
/* pre: true */
/* post: retorna la cantidad de elementos de la lista. */
public abstract int longitud();

/* eliminar(index): elimina el elemento en la posicion index de la lista.*/
/* pre: 1<=index<=longitud() */
/* post: si index es una posicion valida, el item de esa posicion. */
/* Si index es una posicion invalida, lanza una excepcion */
/* IndiceFueraDeRangoLista. */
public abstract void eliminar(int index) throws IndiceFueraDeRangoLista;
```

etc...

INTERFACE LISTA

```java
/**
* Interface de listas
* @author Pablo Castro
* Un ejemplo simple de interface en JAVA, el polimorfismo es obtenido por medio de herencia y la
* clase Object
**/
interface Lista{
    /**
     * Retorna true si la lista es vacia
     * @pre true
     * @post true ssi la lista es vacia
     * @return true cuando la lista es vacia
    **/
    public abstract boolean esVacia();
    /**
     * Elimina todos los elementos de la lista
     * @pre true
     * @post lista vacia
     **/
    public abstract void vaciar();
    /**
     * Retorna la longitud de la lista
     * @pre true
     * @post retorna la longitud
     * @return longitud de la lista
     **/
    public abstract int longitud();
    /**
     * Inserta un elemento en la posicion dada, suponemos que la primera posicion es 0
     * @pre 0 <= index
     * @post inserta el elemento
     * retorna una excepcion en el caso de que el indice sea negativo, o que no se pueda insertar el elemento
     **/
    public abstract void insertar(int index, Object item) throws ExcepcionLista, IndiceFueradeRangoLista;
     /**
     * Elimina el elemento en la posicion dada, suponemos que la primera posicion es 0
     * @pre 0 <= index
     * @post elimina el elemento
     * retorna una excepcion en el caso de que el indice sea negativo.
     **/
    public abstract void eliminar(int index) throws IndiceFueradeRangoLista;
    /**
     * Obtiene el elemento en la posicion dada, suponemos que la primera posicion es 0
     * @pre 0 <= index
     * @post devuelve el elemento
     * @return el elemento buscado
     * retorna una excepcion en el caso de que el indice sea negativo.
     **/
    public abstract Object obtener(int index) throws IndiceFueradeRangoLista;
}
```

IMPLEMENTACIÓN DEL TAD LISTA

```
/**
 * Implementacion de la Interface de listas con arreglos
 * @author Pablo Castro
 * Un ejemplo simple de implementacion de interface en JAVA, el polimorfismo es obtenido por medio de herencia y la
 * clase Object
 **/
public class ListaArray implements Lista{

   private static final int MAX_LIST = 100; // numero maximo de items que se pueden guardar en la lista
   private Object items[]; // arreglo que guarda los items de la lista
   private int numItems; // cantidad de items que tiene la lista
```

se implementan listas utilizando arreglos

numItems es la cantidad de elementos que hay guardados, es decir, apunta a la primera posición libre.

IMPLEMENTACIÓN LISTAS

```
/**
 * Constructor por defecto
 **/
public void ListaArray(){
 items = new Object[MAX_LIST];
 numItems = 0;
}

/**
 * Retorna si la lista es vacia
 * @pre true
 * @post true ssi la lista es vacia
 * @return true ssi la lista es vacia
 **/
public boolean esVacia(){
 return (numItems == 0);
}
```

Constructores

```
/**
    * Retorna la longitud de la lista
    * @pre true
    * @post retorna la longitud
    * @return longitud de la lista
    **/
   public int longitud(){
    return numItems;
   }

   /**
    * Elimina todos los elementos de la lista
    * @pre true
    * @post deja la lista vacia
    **/
   public void vaciar(){
    numItems = 0;
   }
```

Métodos de la clase

IMPLEMENTACIÓN LISTAS

```
/**
 * Inserta un item en la posicion dada
 * @param index indice en donde se inserta el elemento
 * @param item objeto a insertar
 * @pre 0<= index <= numItems
 * @post inserta el elemento en la posicion dada
 * Si el indice esta fuera de rango o no hay suficiente espacio
 * se retorna un error
 **/
public void insertar(int index, Object item) throws ExcepcionLista, IndiceFueradeRangoLista{
 if (numItems == MAX_LIST) {
      throw new ExcepcionLista("ListaSobreArreglos.insertar: Lista llena");
 }
 else{
      if ((index<=0) || (index>numItems+1))
           throw new IndiceFueradeRangoLista("ListaSobreArreglos.insertar: indice invalido");

      else{
           if (index == numItems+1) {
                // insertar item en la ultima posicion
                items[index-1] = item;
                numItems = numItems + 1;
           }
           else{
                // sino se hace una llamada recursiva
                Object temp = items[index-1];
                items[index-1] = item;
                insertar(index+1, temp);
           }
      }
 }

}
```

Inserta un Elemento

IMPLEMENTACIÓN LISTAS

```
/**
 * Elimina el item en la posicion dada
 * @param index indice en donde se encuentra el elemento a borrar
 * @param item objeto a insertar
 * @pre 0<= index <= numItems
 * @post es elimina el elemento en la posicion dada
 * Si el indice esta fuera de rango se dispara una excepcion
 **/
public void eliminar(int index) throws IndiceFueradeRangoLista {
 if ((index<=0) || (index>numItems))
      throw new IndiceFueradeRangoLista("ListSobreArreglos.eliminar: indice invalido");
  else{
      if (index == numItems)
          // eliminar el item en la ultima posicion
          numItems = numItems - 1;
      else{
          items[index-1] = items[index];
          eliminar(index+1);
      }
  }
}
```

Elimina un elemento

IMPLEMENTACIÓN LISTAS

```
/**
 * Obtiene el item en la posicion dada
 * @param index indice en donde se encuentra el elemento a obtener
 * @pre 0<= index <= numItems
 * @post se retorna el elemento correspondiente
 * @return elemento en el indice dado
 * Si el indice esta fuera de rango se dispara una excepcion
 **/
public Object obtener(int index) throws IndiceFueradeRangoLista{
 if ((index<=0) || (index>numItems)){
     throw new IndiceFueradeRangoLista("ListaSobreArreglos: indice invalido");
 }
 else {
     return (items[index-1]);
 }
}
```

Obtiene un elemento

CÓMO USAR LISTAS

Al usar las listas, podemos insertar elementos de cualquier tipo, pero cuando traemos un elemento debemos usar *casts*.

```
Lista miLista = new ListaArray();
String temp = new String("Hola Mundo");
miLista.insertar(1,temp);
Integer i = new Integer(5);
miLista.insertar(2,i);
String temp2 = (String) miLista.obtener(1);
Integer j = (Integer) miLista.obtener(2);
```