Subtipos, herencia y programación orientada a objetos

Fernando Orejas

Introducción

¿Qué es un subtipo?

Definición 1:

t es subtipo de t' si todos los elementos de t son elementos de t'. Por ejemplo, en Pearl:

```
subset Pares of Int where {$_ % 2 == 0}
```

Esta clase de subtipos no son habituales en los lenguajes de programación

¿Qué es un subtipo?

Definición 2:

t es subtipo de t' si en todo contexto en que se puede usar un objeto de tipo t', se puede usar un objeto de tipo t.

Esta la definición en que se basa la programación orientada a objetos

```
class Punto{
  private:
     double x,y;
  public:
  ...
     void mover (double a, double b){
        x = x + a; y = y + b;
     }
  ...
}
```

```
class Punto_color: public Punto {
   private:
       int color;
    public:
    ...
}

    ...
    Punto_color p;
    ...
    p.mover(2.1,3.4)
// Válido porque Puntocolor es subtipo
// (subclase) de Punto
```

Herencia y subclases

La herencia y la relación de subclase permiten:

- Estructurar mejor el código
- Tener una mayor reutilización de código
- Simplificar el diseño

Ejemplo (programación "clásica"):

```
double sueldo (Empleado e);
    if (es_vendedor(e)){
       ...}
    else if (es_contable(e)){
       ...}
    else if (es_ejecutivo(e)){
       ...}
    ...
    }
}
```

Ejemplo (programación OO):

```
class Empleado{
    double sueldo ();
    ...
}
class Vendedor: public Empleado {
    double sueldo ();
    ...
}
class Contable: public Empleado {
    double sueldo ();
    ...
}
```

Herencia y subclases

- En cada subclase podemos redefinir operaciones de la clase base
- Si e es un empleado del tipo que sea e.sueldo() calcularía el sueldo que corresponde a su tipo, ya que el tipo de e se decide en ejecución (No exactamente en C++)

Herencia y subclases

Si cambia la estructura salarial: por ejemplo, hay algún nuevo tipo de empleado, desaparece algun tipo y a otros tipos se les cambia el cálculo del sueldo:

- En la programación clasica hay que rehacer entera la función de sueldo (y quizá lo mismo para otras operaciones)
- En el caso OO, se introducirían nuevas subclases, se eliminarían algunas y se redefiniría el cálculo del sueldo en otras

Lenguaje de programación OO

=

Modularidad (abstracción de datos)

+

Herencia

+

Vinculación dinámica o tardía (Late binding)

Comprobación e inferencia de tipos con subtipos

La estructura de tipos

- Si e:: t y t ≤ t' entonces e:: t'
- Si e:: t, t ≤ t' y f:: t' -> t'' entonces f e:: t''
- En el caso de la asignación, si x:: t, e:: t' y t' ≤ t entonces:

  x = e es correcto

  pero si t ≤ t' entonces

  x = e NO es correcto

Covarianza y contravarianza

- Si $t \leq s$ y $s' \leq t'$ entonces $(s \rightarrow s') \leq (t \rightarrow t')$

Si f:: (t -> t') y g:: (s -> s') y f a es correcto, entonces también es correcto g a, porque como a:: t, entonces también a::s y g a:: s' y, por tanto g a:: t'

- Si $s \leq t$ y $s' \leq t'$ entonces NO $(s \rightarrow s') \leq (t \rightarrow t')$

Si f:: (t -> t') y g:: (s -> s') y f a es correcto, entonces en general g a no es correcto, porque si a:: t, entonces no podemos asegurar que a::s

Covarianza y contravarianza

- Un constructor de tipos C, C es covariante si s ≤ t implica que C<s> ≤ C<t>.
- C es contravariante si s ≤ t implica que C<t> ≤ C<s>.
- C es invariante si s ≤ t no implica C<s> ≤ C<t> ni lo contrario.
- El constructor -> es contravariante en el primer argumento y covariante en el segundo.
- ¿El constructor List es covariante, contravariante o invariante?

Covarianza y contravarianza

Supongamos que List es covariante, es decir que si s ≤ t

entonces List <s> ≤ List <t>:

```
void push (List& <Empleado> L, Empleado e){
   L.insert(L.end(),e)
}
...
List <contable> L; Vendedor e;
...
push(L,e) // ¿es correcto esto?:
```

Otros problemas

Sabemos que si p::Punto_color, p.mover(1.2, 3.4) es correcto ya que mover:: Punto -> double -> double ->Punto, pero entonces:

p.mover(1.2, 3.4):: Punto.

¿Quiere esto decir que, al mover el punto, perdemos el campo del color?

Otros problemas

Si tenemos:

```
class Empleado{
   T f (Empleado x);   ...
}
class Vendedor: public Empleado {
   T f (Vendedor x);   ...
}
Empleado x; Vendedor y;
```

¿Es correcto?:

```
y.f(x);
```

Subclases y Herencia en C++ y Java

Declaración de subclases en C++

```
class Empleado{...}

class Vendedor: public Empleado {...}
```

Pero también

```
class Vendedor: private Empleado {...}

class Vendedor: protected Empleado {...}
```

La diferencia afecta a la visibilidad

Declaración de subclases en C++

Además en C++ tenemos herencia múltiple:

```
class Vendedor: public Empleado,
                public Comisionista {...}
```

Declaración de subclases en Java

```
class Empleado{...}

class Vendedor extends Empleado {...}
```

En Java no hay herencia múltiple con clases, pero sí con interfaces

```
class Vendedor: implements Empleado,
Comisionista {...}
```

Vinculación en Java y en C++

- En Java la vinculación es dinámica, dados:

```
class Empleado {
   public double sueldo() ... }
class Vendedor extends Empleado {
   public double sueldo()...}
class Contable extends Empleado {
   public double sueldo()...}
```

Si declaramos **`Empleado e;`** la llamada **`e.sueldo()`** ejecutaría la operación sueldo dependiendo del tipo de e en ese momento de la ejecución.

Vinculación en Java y en C++

- En C++ la vinculación "normal" es estática, dados:

```
class Empleado {
    double sueldo() ... }
class Vendedor: public Empleado {
    double sueldo()...}
class Contable: public Empleado {
    double sueldo()...}
```

Si declaramos **`Empleado e;`** la llamada **`e.sueldo()`** ejecutaría la operación sueldo de la clase **`Empleado`**, independientemente del tipo de **`e`** en ese momento de la ejecución.

Vinculación en Java y en C++

Si queremos tener vinculación dinámica en C++ debemos:

1. Declarar como **`virtual`** el método **`sueldo`**. de la clase **`empleado`**. Al hacer eso, la clase **`empleado`** se convierte en abstracta.
2. Como las clases abstractas no pueden tener instancias, hay que pasar como argumentos objetos dinámicos

```
class Empleado {
   virtual double sueldo(); ... }
class Vendedor: public Empleado {
    double sueldo()...}
class Contable: public Empleado {
    double sueldo()...}

Empleado* e;
```

En este caso, la llamada **`e->sueldo()`** ejecutaría la operación sueldo dependiendo del tipo del objeto apuntado por e en ese momento de la ejecución.

Vinculación en Java y en C++

Cuando tenemos vinculación dinámica, dado un objeto **`e`** y la operación **`e.op(...)`**:

1. En tiempo de ejecución, se calcula cual es la clase **`C`** de **`e`**.
2. Si **`C`** tiene declarada la operación **`op(...)`**, entonces se ejecuta esa operación.
3. En caso contrario, se ejecuta la operación **`op(...)`** de la superclase de **`C`** que la tenga declarada y que sea más cercana en la jerarquía de clases.

Ámbitos y visibilidad en C++ y Java

Dada la declaración:

```
class Vendedor: public Empleado {
```

- Los atributos y métodos privados en **`Empleado`** no son visibles en **`Vendedor`**.
- Los atributos y métodos públicos o protegidos (**`protected`**) en **`Empleado`** son, respectivamente, públicos o protegidos en **`Vendedor`**.

En Java la definición de subclase es similar a la **`public`** de C++

Ámbitos y visibilidad en C++

Dada la declaración:

```
class Vendedor: private Empleado {
```

- Los atributos y métodos privados en **`Empleado`** no son visibles en **`Vendedor`**.
- Los atributos y métodos públicos o protegidos en **`Empleado`** son privados en **`Vendedor`**.

Ámbitos y visibilidad en C++

Dada la declaración:

```
class Vendedor: protected Empleado {
```

- Los atributos y métodos privados en **`Empleado`** no son visibles en **`Vendedor`**.
- Los atributos y métodos públicos o protegidos en **`Empleado`** son protegidos en **`Vendedor`**.