Paradigmas Avanzados de Programación

El modelo Orientado a Objetos

Juan Francisco Díaz Frias

Maestría en Ingeniería, Énfasis en Ingeniería de Sistemas y Computación
Escuela de Ingeniería de Sistemas y Computación,
home page: http://eisc.univalle.edu.co
Universidad del Valle - Cali, Colombia

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Plan

Plan

Un ejemplo de clase: sintaxis

```
class Contador
   attr val
   meth inic(Valor)
      val:=Valor
   end
   meth browse
      {Browse @val}
   end
   meth inc(Valor)
      val:=@val+Valor
   end
end
```

Supuestos

- `class`: constructor nuevo.
- Las clases son valores de primera clase.
- `class` es una abstracción lingüística.

Sintácticamente:

- Nombre `Contador` (podría ser anónima).
- Atributos: `val`.
- Métodos: `inic`, `browse`, e `inc`.
- Operadores: `:=` y `@`.

Un ejemplo de clase: sintaxis

```
class Contador
   attr val
   meth inic(Valor)
      val:=Valor
   end
   meth browse
      {Browse @val}
   end
   meth inc(Valor)
      val:=@val+Valor
   end
end
```

Supuestos

- **`class`**: constructor nuevo.
- Las clases son valores de primera clase.
- **`class`** es una abstracción lingüística.

Sintácticamente:

- Nombre `Contador` (podría ser anónima).
- Atributos: `val`.
- Métodos: `inic`, `browse`, e `inc`.
- Operadores: `:=` y `@`.

Un ejemplo de clase: sintaxis

```
class Contador
   attr val
   meth inic(Valor)
      val:=Valor
   end
   meth browse
      {Browse @val}
   end
   meth inc(Valor)
      val:=@val+Valor
   end
end
```

Supuestos

- **`class`**: constructor nuevo.
- Las clases son valores de primera clase.
- **`class`** es una abstracción lingüística.

Sintácticamente:

- Nombre `Contador` (podría ser anónima).
- Atributos: `val`.
- Métodos: `inic`, `browse`, e `inc`.
- Operadores: `:=` y `@`.

¿Es la diferencia con otros lenguajes sólo sintáctica?

Las apariencias engañan

- La declaración se ejecuta en tiempo de ejecución: se crea un valor de tipo clase y se liga a la variable `Contador`.
- La declaración al principio del programa se comporta familiarmente.
- Pero, la declaración puede colocarse en cualquier sitio donde pueda ir una declaración. Por ejemplo, si se coloca la declaración dentro de un procedimiento, se creará una clase nueva, distinta, cada vez que se invoca el procedimiento.

Ejemplo de uso

```
C={New Contador inic(0)}
{C inc(6)} {C inc(6)}
{C browse}

local X in {C inc(X)} X=5 end
{C browse}
```

```
declare S in
local X in
thread {C inc(X)} S=listo end
X=5
end
{Wait S} {C browse}
```

Plan

Un ejemplo de clase: semántica (1)

```
local
   proc {Inic M S}
      inic(Valor)=M in (S.val):=Valor
   end
   proc {Browse2 M S}
      {Browse @(S.val)}
   end
   proc {Inc M S}
      inc(Valor)=M in (S.val):=@(S.val)+Valor
   end
in
   Contador=c(atrbs:[val]
              métodos:m(inic:Inic browse:Browse2
inc:Inc))
end
```

Detalles

- Una clase es un registro: nombres de atributos; métodos.
- Nombre de atributo: literal.
- Método: procedimiento de dos argumentos: mensaje y estado.
- Asignación `val:=` y acceso `@val`.

Un ejemplo de clase: semántica (1)

```
local
   proc {Inic M S}
      inic(Valor)=M in (S.val):=Valor
   end
   proc {Browse2 M S}
      {Browse @(S.val)}
   end
   proc {Inc M S}
      inc(Valor)=M in (S.val):=@(S.val)+Valor
   end
in
   Contador=c(atrbs:[val]
              métodos:m(inic:Inic browse:Browse2
inc:Inc))
end
```

Detalles

- Una clase es un registro: nombres de atributos; métodos.
- Nombre de atributo: literal.
- Método: procedimiento de dos argumentos: mensaje y estado.
- Asignación `val:=` y acceso `@val`.

Un ejemplo de clase: semántica (2)

```
fun {New Clase Inic}
   Fs={Map Clase.atrbs fun {$ X} X#{NewCell _}}
end}
   S={List.toRecord estado Fs}
   proc {Obj M}
      {Clase.métodos.{Label M} M S}
   end
in
   {Obj Inic}
   Obj
end
```

La función `New`

- Crea el estado del objeto.
- Define `Obj` que es de hecho el objeto.
- Inicializa el objeto antes de devolverlo.
- Estado: registro `S`, oculto dentro de `Obj`.

Un ejemplo de clase: semántica (2)

```
fun {New Clase Inic}
   Fs={Map Clase.atrbs fun {$ X} X#{NewCell _}}
end}
   S={List.toRecord estado Fs}
   proc {Obj M}
      {Clase.métodos.{Label M} M S}
   end
in
   {Obj Inic}
   Obj
end
```

La función `New`

- Crea el estado del objeto.
- Define `Obj` que es de hecho el objeto.
- Inicializa el objeto antes de devolverlo.
- Estado: registro `S`, oculto dentro de `Obj`.

Plan

Sintaxis de clase (1)

⟨declaración⟩ ::= **class** ⟨variable⟩ { ⟨descriptorClase⟩ }
{ **meth** ⟨cabezaMétodo⟩ ['=' ⟨variable⟩]
(⟨expresiónEn⟩ | ⟨declaraciónEn⟩) **end** }
end
| **lock** [⟨expresión⟩ **then**] ⟨declaraciónEn⟩ **end**
| ⟨expresión⟩ ':=' ⟨expresión⟩
| ⟨expresión⟩ ',' ⟨expresión⟩
| ...

Sintaxis de clase (2)

⟨expresión⟩ ::= **class** '$' { ⟨descriptorClase⟩ }
{ **meth** ⟨cabezaMétodo⟩ ['=' ⟨variable⟩]
(⟨expresiónEn⟩ | ⟨declaraciónEn⟩) **end** }
end
| **lock** [⟨expresión⟩ **then**] ⟨expresiónEn⟩ **end**
| ⟨expresión⟩ ':=' ⟨expresión⟩
| ⟨expresión⟩ ',' ⟨expresión⟩
| '@' ⟨expresión⟩
| **self**
| ...

Sintaxis de clase (3)

⟨descriptorClase⟩ ::= **from** { ⟨expresión⟩ }+ | **prop** { ⟨expresión⟩ }+
| **attr** { ⟨inicAtrb⟩ }+
⟨inicAtrb⟩ ::= ([' ! '] ⟨variable⟩ | ⟨átomo⟩ | **unit** | **true** | **false**)
[' : ' ⟨expresión⟩]
⟨cabezaMétodo⟩ ::= ([' ! '] ⟨variable⟩ | ⟨átomo⟩ | **unit** | **true** | **false**)
[' (' { ⟨argMétodo⟩ } [' ... '] ') ']
[' = ' ⟨variable⟩]
⟨argMétodo⟩ ::= [⟨feature⟩ ' : '] (⟨variable⟩ | ' _ ' | ' $ ') [' <= ' ⟨expresión⟩]

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

¿Qué es una clase?

Una clase es una estructura de datos que define un estado interno de un objeto (atributos), su comportamiento (métodos), las clases de las cuales hereda, y otras propiedades y operaciones.

Características

- Número de objetos ilimitado.
- Objeto $\equiv$ instancia. Cada instancia tiene identidad única.
- Cada objeto se comporta según la definición de clase.

Creación, invocación

```
MiObj={New MiClase Inic}
```

crea un objeto `MiObj`, de la clase `MiClase` e invoca el objeto con el mensaje `Inic`.
Con la sintaxis `{MiObj M}` se invoca el objeto.

¿Qué es una clase?

Una clase es una estructura de datos que define un estado interno de un objeto (atributos), su comportamiento (métodos), las clases de las cuales hereda, y otras propiedades y operaciones.

Características

- Número de objetos ilimitado.
- Objeto $\equiv$ instancia. Cada instancia tiene identidad única.
- Cada objeto se comporta según la definición de clase.

Creación, invocación

```
MiObj={New MiClase Inic}
```

crea un objeto `MiObj`, de la clase `MiClase` e invoca el objeto con el mensaje `Inic`.
Con la sintaxis `{MiObj M}` se invoca el objeto.

¿Qué es una clase?

Una clase es una estructura de datos que define un estado interno de un objeto (atributos), su comportamiento (métodos), las clases de las cuales hereda, y otras propiedades y operaciones.

Características

- Número de objetos ilimitado.
- Objeto $\equiv$ instancia. Cada instancia tiene identidad única.
- Cada objeto se comporta según la definición de clase.

Creación, invocación

```
MiObj={New MiClase Inic}
```

crea un objeto `MiObj`, de la clase `MiClase` e invoca el objeto con el mensaje `Inic`.
Con la sintaxis `{MiObj M}` se invoca el objeto.

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Miembros de una clase (1)

Atributos: palabra reservada "`attr`"

O variables de instancia: celda que contiene parte del estado de la instancia.

Operaciones:

- $\langle \text{expr} \rangle_1 := \langle \text{expr} \rangle_2$
- $@\langle \text{expr} \rangle$
- $\langle \text{expr} \rangle_3 = \langle \text{expr} \rangle_1 := \langle \text{expr} \rangle_2$.

Miembros de una clase (2)

Métodos: palabra reservada "`meth`"

Procedimiento que se invoca en el contexto de un objeto particular y que tiene acceso a los atributos del objeto.

- Consta de una cabeza y un cuerpo.
- La cabeza consta de una etiqueta, la cual debe ser un átomo o un nombre, y de un conjunto de argumentos.
- Los argumentos deben ser variables diferentes; de otra manera sería un error de sintaxis.
- Cabeza≡Patrón y Mensaje≡Registro.

Miembros de una clase (3)

Propiedades: palabra reservada "`prop`"

Una propiedad modifica cómo se comporta un objeto:

- La propiedad `locking` crea un candado nuevo con cada instancia de objeto.
- La propiedad `final` hace que la clase sea una clase final

Miembros de una clase (4)

Atributos como átomos o como identificadores

- Las etiquetas de los atributos y de los métodos son literales.
- Si se definen utilizando la sintaxis de átomo, entonces son átomos.
- Si se definen con la sintaxis de identificadores (e.g., en mayúscula), entonces el sistema creará nombres nuevos para ellos, cuyo alcance es la definición de la clase.

Plan

Inicialización de atributos (1)

Inicialización por instancia

Un atributo puede tener un valor inicial diferente para cada instancia. Esto se logra no inicializándolos en la definición de clase.

Ejemplo

```
class UnApart
   attr nombreCalle
   meth inic(X) @nombreCalle=X end
end
Apt1={New UnApart inic(pasoancho)}
Apt2={New UnApart inic(calleQuinta)}
```

Inicialización de atributos (2)

Inicialización por clase

Un atributo puede tener un valor que sea el mismo para todas las instancias de la clase. Esto se hace, inicializándolo con “:” en la definición de clase.

Ejemplo

```
class ApartQuinta
   attr
      nombreCalle:calleQuinta
      númeroCalle:100
      colorPared:_
      superficiePiso:madera
   meth inic skip end
end
Apt3={New ApartQuinta inic}
Apt4={New ApartQuinta inic}
```

Inicialización de atributos (3)

Inicialización por marca

Esta es otra manera de utilizar la inicialización por clase. Una marca es un conjunto de clases relacionadas de alguna manera, pero no por herencia.

Ejemplo

```
L=linux
class RedHat attr tiposo:L end
class SuSE attr tiposo:L end
class Debian attr tiposo:L end
```

Plan

Mensajes de primera clase (1)

Principio

Los mensajes son registros y las cabezas de los métodos son patrones que reconocen un registro.

Invocación de objeto `{Obj M}`

- *Registro estático como mensaje*. En el caso más sencillo, `M` es un registro que se conoce en tiempo de compilación, e.g., como en la invocación `{Contador inc(X)}`.
- *Registro dinámico como mensaje*. Es posible invocar `{Obj M}` donde `M` es una variable que referencia un registro que se calcula en tiempo de ejecución. Como el tipamiento es dinámico, es posible crear nuevos tipos de registros en tiempo de ejecución (e.g., con `Adjoin` o `List.toRecord`).

Mensajes de primera clase (1)

Principio

Los mensajes son registros y las cabezas de los métodos son patrones que reconocen un registro.

Invocación de objeto `{Obj M}`

- *Registro estático como mensaje*. En el caso más sencillo, `M` es un registro que se conoce en tiempo de compilación, e.g., como en la invocación `{Contador inc(X)}`.
- *Registro dinámico como mensaje*. Es posible invocar `{Obj M}` donde `M` es una variable que referencia un registro que se calcula en tiempo de ejecución. Como el tipamiento es dinámico, es posible crear nuevos tipos de registros en tiempo de ejecución (e.g., con `Adjoin` o `List.toRecord`).

Mensajes de primera clase (2)

Definición de un método (1)

- Lista fija de argumentos:

```
meth foo(a:A b:B c:C)
   % Cuerpo del método
end
```

- Lista flexible de argumentos:

```
meth foo(a:A b:B c:C ...)
   % Cuerpo del método
end
```

- Referencia variable a la cabeza del método:

```
meth foo(a:A b:B c:C ...)=M
   % Cuerpo del método
end
```

Mensajes de primera clase (3)

Definición de un método (2)

- Argumento opcional:valor por defecto, se usa sólo si el argumento no viene en el mensaje.

```
meth foo(a:A b:B<=V)
   % Cuerpo del método
end
```

`foo(a:1 b:2)` ignora `V`, `foo(a:1)` $\equiv$ `foo(a:1 b:V)`

- Etiqueta privada de método: `A` se liga a un nombre fresco cuando se define la clase.

```
meth A(bar:X)
   % Cuerpo del método
end
```

- Etiqueta dinámica de método:

```
meth !A(bar:X)
   % Cuerpo del método
end
```

La etiqueta del método debe ser conocida en el momento en que la definición de clase sea ejecutada.

Mensajes de primera clase (4)

Definición de un método (3)

- Etiqueta dinámica de método:

```
meth !A(bar:X)
   % Cuerpo del método
end
```

La etiqueta del método debe ser conocida en el momento en que la definición de clase sea ejecutada.

- El método `otherwise`: acepta cualquier mensaje para el cual no exista ningún otro método.

```
meth otherwise(M)
   % Cuerpo del método
end
```

¿Cómo hace el compilador con la invocación `{Obj M}`?

Estáticamente ...

Determinar `Obj` y `M`. Si puede, compila a una instrucción muy rápida y especializada.

Si no puede ...

- Compila a un instrucción general de invocación de un objeto.
- La instrucción general utiliza el ocultamiento.
- La primera invocación es lenta, pero las subsiguientes son casi tan rápidas como las invocaciones especializadas.

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Tres conjuntos de conceptos para la herencia:

Grafo de herencia
Define cuáles clases preexistentes se extenderán. Se permitirá tanto herencia sencilla como herencia múltiple.

Control de acceso a los métodos
Define cómo se accede a los métodos en particular, tanto en la clase nueva como en las clases preexistentes. Esto se logra con ligaduras estática y dinámica y con el concepto de `self`.

Control de la encapsulación
Define qué parte de un programa puede ver los atributos y métodos de una clase.

Otros conceptos ...
Reenvío, delegación, y reflexión.

Tres conjuntos de conceptos para la herencia:

Grafo de herencia

Define cuáles clases preexistentes se extenderán. Se permitirá tanto herencia sencilla como herencia múltiple.

Control de acceso a los métodos

Define cómo se accede a los métodos en particular, tanto en la clase nueva como en las clases preexistentes. Esto se logra con ligaduras estática y dinámica y con el concepto de `self`.

Control de la encapsulación

Define qué parte de un programa puede ver los atributos y métodos de una clase.

Otros conceptos . . .

Reenvío, delegación, y reflexión.

Tres conjuntos de conceptos para la herencia:

Grafo de herencia

Define cuáles clases preexistentes se extenderán. Se permitirá tanto herencia sencilla como herencia múltiple.

Control de acceso a los métodos

Define cómo se accede a los métodos en particular, tanto en la clase nueva como en las clases preexistentes. Esto se logra con ligaduras estática y dinámica y con el concepto de `self`.

Control de la encapsulación

Define qué parte de un programa puede ver los atributos y métodos de una clase.

Otros conceptos . . .

Reenvío, delegación, y reflexión.

Tres conjuntos de conceptos para la herencia:

Grafo de herencia

Define cuáles clases preexistentes se extenderán. Se permitirá tanto herencia sencilla como herencia múltiple.

Control de acceso a los métodos

Define cómo se accede a los métodos en particular, tanto en la clase nueva como en las clases preexistentes. Esto se logra con ligaduras estática y dinámica y con el concepto de `self`.

Control de la encapsulación

Define qué parte de un programa puede ver los atributos y métodos de una clase.

Otros conceptos . . .

Reenvío, delegación, y reflexión.

Modelos y Paradigmas de Programación

Universidad del Valle

Plan

Grafo de herencia (1)

Disponibilidad de métodos y atributos

La herencia es una forma de construir clases nuevas a partir de clases existentes.
La herencia define qué atributos y métodos están disponibles en la clase nueva.

Relación de precedencia: Relación de anulación

Un método (atributo) en la clase C anula cualquier método (atributo) con la misma etiqueta (el mismo nombre) en todas las superclases de C.

Grafo de herencia (1)

Disponibilidad de métodos y atributos

La herencia es una forma de construir clases nuevas a partir de clases existentes.
La herencia define qué atributos y métodos están disponibles en la clase nueva.

Relación de precedencia: Relación de anulación

Un método (atributo) en la clase C anula cualquier método (atributo) con la misma etiqueta (el mismo nombre) en todas las superclases de C.

Grafo de herencia (2)

Tipos de herencia

Herencia sencilla (se hereda de una sola clase) o Herencia múltiple (se hereda de varias clases. Sintácticamente: **`from`**.

Superclases

Una clase `B` es una superclase de la clase `A` si

- `B` aparece en la parte **`from`** de la declaración de `A`, o
- `B` es una superclase de una clase que aparece en la parte **`from`** de la declaración de `A`.

Grafo de herencia (2)

Tipos de herencia

Herencia sencilla (se hereda de una sola clase) o Herencia múltiple (se hereda de varias clases. Sintácticamente: **`from`**.

Superclases

Una clase `B` es una superclase de la clase `A` si

- `B` aparece en la parte **`from`** de la declaración de `A`, o
- `B` es una superclase de una clase que aparece en la parte **`from`** de la declaración de `A`.

Grafo de herencia (3)

Jerarquía de clases

Grafo dirigido de la relación de superclase cuya raíz es la clase actual. Las aristas son dirigidas hacia las subclases.

Legalidad de la herencia

- La relación de herencia es dirigida y acíclica.

```
class A from B ... end
class B from A ... end
```

- Cada método (salvo los anulados) debe tener una etiqueta única y debe estar definido en una sola clase en la jerarquía.

```
class A1 meth m(...) ... end end
class B1 meth m(...) ... end end
class A from A1 end
class B from B1 end
class C from A B end
```

Grafo de herencia (3)

Jerarquía de clases

Grafo dirigido de la relación de superclase cuya raíz es la clase actual. Las aristas son dirigidas hacia las subclases.

Legalidad de la herencia

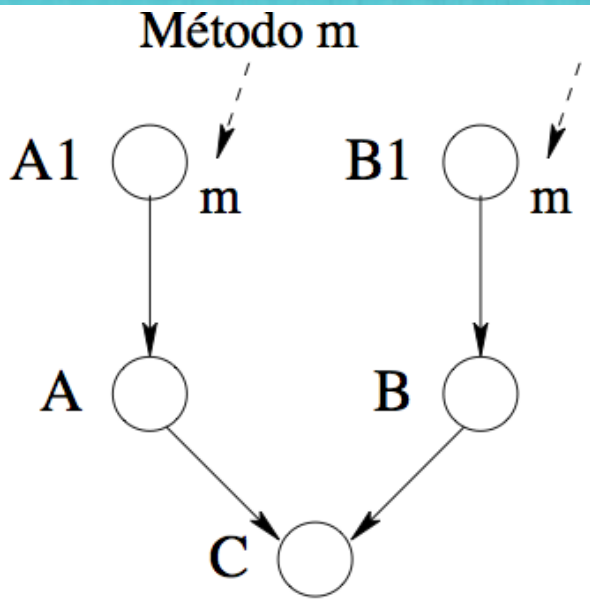
- La relación de herencia es dirigida y acíclica.

```
class A from B ... end
class B from A ... end
```

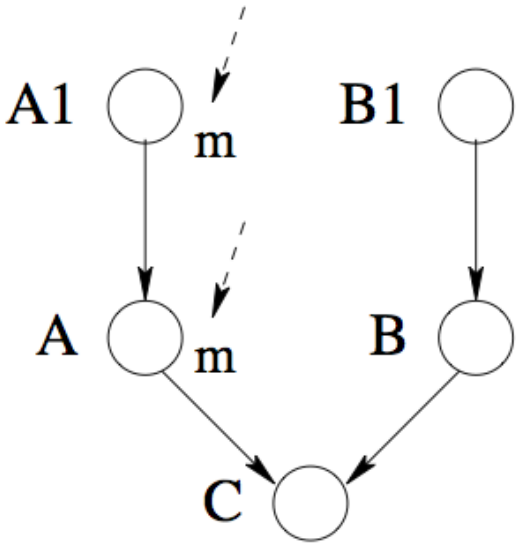
- Cada método (salvo los anulados) debe tener una etiqueta única y debe estar definido en una sola clase en la jerarquía.

```
class A1 meth m(...) ... end end
class B1 meth m(...) ... end end
class A from A1 end
class B from B1 end
class C from A B end
```

Grafo de herencia (4)



Jerarquía de clases ilegal
(dos ‘‘m’’ visibles desde C)

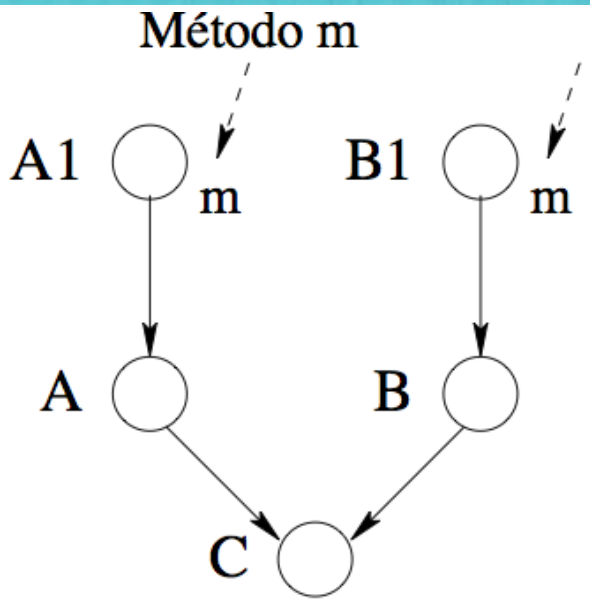


Jerarquía de clases legal
(un ‘‘m’’ visible desde C)

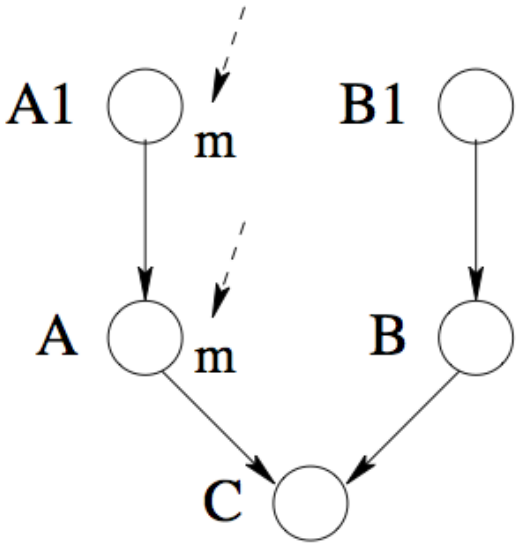
También es ilegal . . .

```
class A meth m(...) ... end end
```

Grafo de herencia (4)



Jerarquía de clases ilegal
(dos ‘‘m’’ visibles desde C)



Jerarquía de clases legal
(un ‘‘m’’ visible desde C)

También es ilegal . . .

```
class A meth m(...) ... end end
```

Grafo de herencia (5)

¿Cuando se detecta un error de estos?

```
fun {ClaseExtraña}
   class A meth foo(X) X=a end end
   class B meth foo(X) X=b end end
   class C from A B end
in C end
```

Principio:todo se hace en tiempo de ejecución

`ClaseExtraña` se puede compilar y ejecutar exitosamente. Sólo se lanzará una excepción en la invocación `{ClaseExtraña}`.

Grafo de herencia (5)

¿Cuando se detecta un error de estos?

```
fun {ClaseExtraña}
   class A meth foo(X) X=a end end
   class B meth foo(X) X=b end end
   class C from A B end
in C end
```

Principio:todo se hace en tiempo de ejecución

`ClaseExtraña` se puede compilar y ejecutar exitosamente. Sólo se lanzará una excepción en la invocación `{ClaseExtraña}`.

Plan

Control de acceso a los métodos (1)

Búsqueda del método correcto

¿Cuando se invoca un método de un objeto: cuál método se ejecuta? Esto parece bastante sencillo, pero se vuelve ligeramente más complicado cuando está involucrada la herencia.

Ligadura estática y dinámica

La herencia se utiliza para definir una clase nueva que extiende una clase existente. Ambas clases pueden tener métodos con el mismo nombre, y la clase nueva puede querer invocar cualquiera de ellos. Se necesitan dos maneras de invocar: ligadura estática y ligadura dinámica.

Control de acceso a los métodos (1)

Búsqueda del método correcto

¿Cuando se invoca un método de un objeto: cuál método se ejecuta? Esto parece bastante sencillo, pero se vuelve ligeramente más complicado cuando está involucrada la herencia.

Ligadura estática y dinámica

La herencia se utiliza para definir una clase nueva que extiende una clase existente. Ambas clases pueden tener métodos con el mismo nombre, y la clase nueva puede querer invocar cualquiera de ellos. Se necesitan dos maneras de invocar: ligadura estática y ligadura dinámica.

Control de acceso a los métodos (2)

Ejemplo de clase: `Cuenta`

```
class Cuenta
   attr saldo:0
   meth transferir(Cant)
      saldo:=@saldo+Cant
   end
   meth pedirSaldo(Sal)
      Sal=@saldo
   end
   meth transferEnLote(CantList)
      for A in CantList do {self transferir(A)} end
   end
end
```

Control de acceso a los métodos (3)

Extensión de `Cuenta`

```
class CuentaVigilada from Cuenta
   meth transferir(Cant)
      {LogObj agregueEntrada(transferir(Cant))}
      ...
   end
end
```

Creación e invocación

```
CtaVig={New CuentaVigilada transferir(100)}
```

¿qué pasa cuando invocamos `transferEnLote`?
¿Se invoca `transferir` de `Cuenta` o `transferir` de `CuentaVigilada`?

Control de acceso a los métodos (3)

Extensión de `Cuenta`

```
class CuentaVigilada from Cuenta
   meth transferir(Cant)
      {LogObj agregueEntrada(transferir(Cant))}
      ...
   end
end
```

Creación e invocación

```
CtaVig={New CuentaVigilada transferir(100)}
```

¿qué pasa cuando invocamos `transferEnLote`?
¿Se invoca `transferir` de `Cuenta` o `transferir` de `CuentaVigilada`?

Control de acceso a los métodos (4)

¿Cuál debería invocarse?

`transferir` de `CuentaVigilada`.

Esto se denomina ligadura dinámica: `{self transferir(A)}`.

Ligadura dinámica: necesidad

Se conserva la funcionalidad de la anterior abstracción al tiempo que se añade una funcionalidad nueva.

Control de acceso a los métodos (4)

¿Cuál debería invocarse?

`transferir` de `CuentaVigilada`.
Esto se denomina ligadura dinámica: `{self transferir(A)}`.

Ligadura dinámica: necesidad

Se conserva la funcionalidad de la anterior abstracción al tiempo que se añade una funcionalidad nueva.

Control de acceso a los métodos (5)

Ligadura dinámica: limitación

```
class CuentaVigilada from Cuenta
   meth transferir(Cant)
      {LogObj agregueEntrada(transferir(Cant))}
      Cuenta,transferir(Cant)
   end
end
```

Dentro del nuevo método `transferir`, tenemos que invocar el antiguo método `transferir`. ¿Podemos usar ligadura dinámica?

Ligadura estática:necesidad

Invocamos un método señalando la clase del método:

`Cuenta,transferir(Cant)`

Control de acceso a los métodos (5)

Ligadura dinámica: limitación

```
class CuentaVigilada from Cuenta
   meth transferir(Cant)
      {LogObj agregueEntrada(transferir(Cant))}
      Cuenta,transferir(Cant)
   end
end
```

Dentro del nuevo método `transferir`, tenemos que invocar el antiguo método `transferir`. ¿Podemos usar ligadura dinámica?

Ligadura estática:necesidad

Invocamos un método señalando la clase del método:

`Cuenta,transferir(Cant)`

Control de acceso a los métodos (6)

Ligadura dinámica y estática

- La ligadura dinámica permite a la clase nueva extender correctamente la clase antigua dejando que los métodos antiguos invoquen los métodos nuevos, aunque los métodos nuevos no existan en el momento en que los métodos antiguos se definen.
- La ligadura estática permite que los nuevos métodos invoquen los métodos antiguos cuando tengan que hacerlo.

Control de acceso a los métodos (7)

Resumen ligadura dinámica

Se escribe `{self M}`. Este tipo de ligadura escoge el correspondiente método `M` visible en el objeto actual, tomando en cuenta la anulación que haya sido realizada.

Resumen ligadura estática

Se escribe `C, M` (con una coma), donde `C` es una clase donde se define el correspondiente método `M`. Este tipo de ligadura escoge el método `M` visible en la clase `C`, tomando en cuenta las anulaciones de métodos desde la clase raíz hasta la clase `C`, pero no más.
Si el objeto es de una subclase de `C` que ha anulado el método `M` de nuevo, entonces ésta anulación no es tenida en cuenta.

Control de acceso a los métodos (7)

Resumen ligadura dinámica

Se escribe `{self M}`. Este tipo de ligadura escoge el correspondiente método `M` visible en el objeto actual, tomando en cuenta la anulación que haya sido realizada.

Resumen ligadura estática

Se escribe `C, M` (con una coma), donde `C` es una clase donde se define el correspondiente método `M`. Este tipo de ligadura escoge el método `M` visible en la clase `C`, tomando en cuenta las anulaciones de métodos desde la clase raíz hasta la clase `C`, pero no más.
Si el objeto es de una subclase de `C` que ha anulado el método `M` de nuevo, entonces ésta anulación no es tenida en cuenta.

Plan

Control de encapsulación (1)

Principio

Limitar el acceso a los miembros de la clase, a saber, atributos y métodos, de acuerdo a los requerimientos de la arquitectura de la aplicación.

Alcance de un miembro

Cada miembro se define con un alcance: El alcance es la parte del texto del programa en el cual el miembro es visible, i.e., donde, al mencionar su nombre, se accede a él.

Tipos de alcance

Por defecto, o modificado por palabras reservadas como `public`, `private`, y `protected`.
Cada lenguaje usa estos alcances con significados ligeramente diferentes.

Control de encapsulación (1)

Principio

Limitar el acceso a los miembros de la clase, a saber, atributos y métodos, de acuerdo a los requerimientos de la arquitectura de la aplicación.

Alcance de un miembro

Cada miembro se define con un alcance: El alcance es la parte del texto del programa en el cual el miembro es visible, i.e., donde, al mencionar su nombre, se accede a él.

Tipos de alcance

Por defecto, o modificado por palabras reservadas como `public`, `private`, y `protected`.

Cada lenguaje usa estos alcances con significados ligeramente diferentes.

Control de encapsulación (1)

Principio

Limitar el acceso a los miembros de la clase, a saber, atributos y métodos, de acuerdo a los requerimientos de la arquitectura de la aplicación.

Alcance de un miembro

Cada miembro se define con un alcance: El alcance es la parte del texto del programa en el cual el miembro es visible, i.e., donde, al mencionar su nombre, se accede a él.

Tipos de alcance

Por defecto, o modificado por palabras reservadas como `public`, `private`, y `protected`.
Cada lenguaje usa estos alcances con significados ligeramente diferentes.

Control de encapsulación (2)

Alcance privado

Un miembro privado es aquel que sólo es visible en la instancia correspondiente al objeto. Esta instancia puede ver todos los miembros definidos en su clase y en sus superclases. Visibilidad vertical.

Alcance público

Un miembro público es aquel que es visible en cualquier parte del programa.

¿Qué es lo natural?

Que los atributos sean privados y los métodos sean públicos, como en Oz y Smalltalk.

Control de encapsulación (2)

Alcance privado

Un miembro privado es aquel que sólo es visible en la instancia correspondiente al objeto. Esta instancia puede ver todos los miembros definidos en su clase y en sus superclases. Visibilidad vertical.

Alcance público

Un miembro público es aquel que es visible en cualquier parte del programa.

¿Qué es lo natural?

Que los atributos sean privados y los métodos sean públicos, como en Oz y Smalltalk.

Control de encapsulación (2)

Alcance privado

Un miembro privado es aquel que sólo es visible en la instancia correspondiente al objeto. Esta instancia puede ver todos los miembros definidos en su clase y en sus superclases. Visibilidad vertical.

Alcance público

Un miembro público es aquel que es visible en cualquier parte del programa.

¿Qué es lo natural?

Que los atributos sean privados y los métodos sean públicos, como en Oz y Smalltalk.

Control de encapsulación (3)

Atributos privados

Los atributos son internos a la abstracción de datos y deben ser invisibles desde el exterior.

Métodos públicos

Los métodos conforman la interfaz externa de la abstracción de datos, por lo tanto deberían ser visibles para todas las entidades que referencian la abstracción.

Control de encapsulación (3)

Atributos privados

Los atributos son internos a la abstracción de datos y deben ser invisibles desde el exterior.

Métodos públicos

Los métodos conforman la interfaz externa de la abstracción de datos, por lo tanto deberían ser visibles para todas las entidades que referencian la abstracción.

Control de encapsulación (4)

Dos conceptos básicos para controlar la encapsulación

Alcance léxico y valores de tipo nombre. Con ellos se implementa:

- Alcane público.
- Alcance privado.
- Alcance privado y protegido de C++ y Java.

Técnica fundamental

La técnica fundamental es permitir que las cabezas de los métodos sean valores de tipo nombre en lugar de átomos.

Control de encapsulación (4)

Dos conceptos básicos para controlar la encapsulación

Alcance léxico y valores de tipo nombre. Con ellos se implementa:

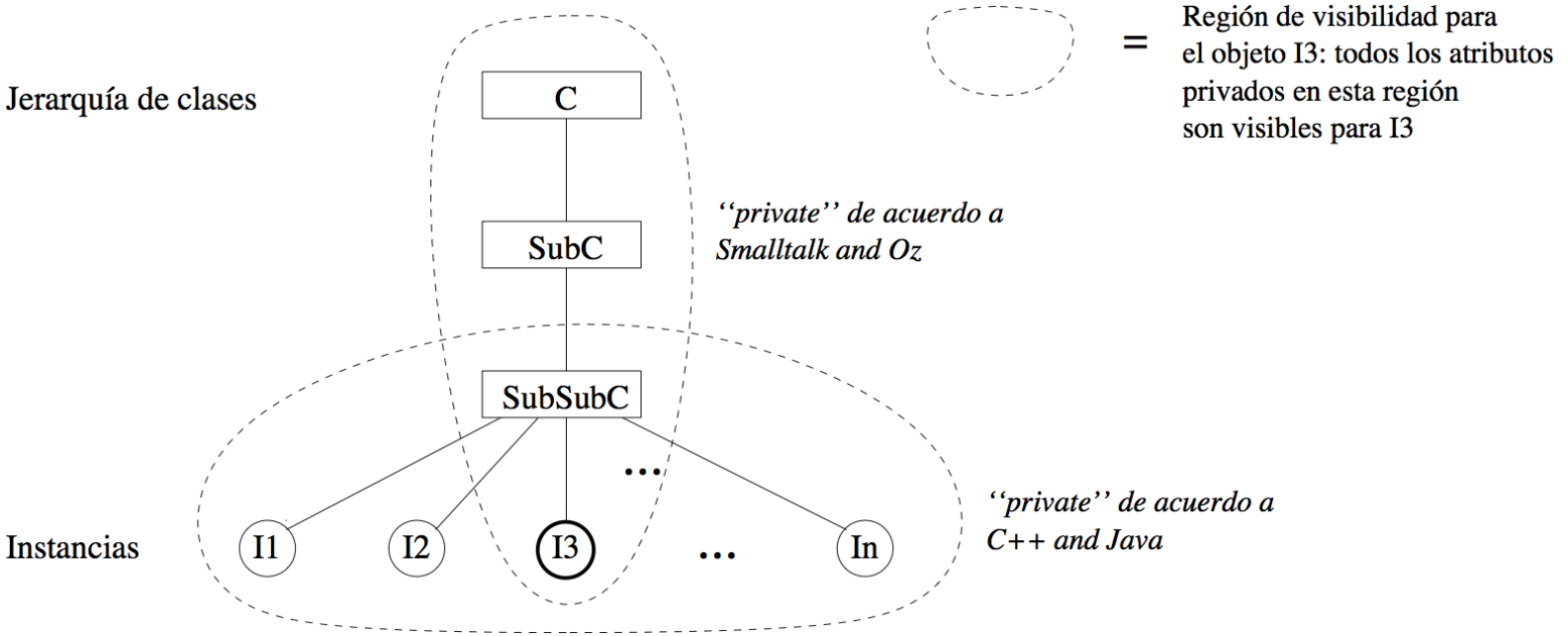
- Alcane público.
- Alcance privado.
- Alcance privado y protegido de C++ y Java.

Técnica fundamental

La técnica fundamental es permitir que las cabezas de los métodos sean valores de tipo nombre en lugar de átomos.

Control de encapsulación (5)

Métodos privados (en el sentido de C++ y Java)



Control de encapsulación (6)

Implementación vía nombres (1)

Utilizando un identificador de variable como la cabeza del método.

```
class C
   meth A(X)
      % Cuerpo del método
   end
end
```

Implementación vía nombres (2)

```
local
   A={NewName}
in
   class C
      meth !A(X)
         % Cuerpo del método
      end
   end
end
```

Control de encapsulación (6)

Implementación vía nombres (1)

Utilizando un identificador de variable como la cabeza del método.

```
class C
   meth A(X)
      % Cuerpo del método
   end
end
```

Implementación vía nombres (2)

```
local
   A={NewName}
in
   class C
      meth !A(X)
         % Cuerpo del método
      end
   end
end
```

Control de encapsulación (7)

Métodos protegidos (en el sentido de C++)

En C++, un método protegido solamente es accesible desde la clase donde fue definido o en las clases descendientes (y en todas las instancias de objetos de esas clases).

Implementación

```
class C
   attr pa:A
   meth A(X) skip end
   meth foo(...) {self A(5)} end
end
class C1 from C
   meth b(...) A=@pa in {self A(5)} end
end
```

Control de encapsulación (7)

Métodos protegidos (en el sentido de C++)

En C++, un método protegido solamente es accesible desde la clase donde fue definido o en las clases descendientes (y en todas las instancias de objetos de esas clases).

Implementación

```
class C
   attr pa:A
   meth A(X) skip end
   meth foo(...) {self A(5)} end
end
class C1 from C
   meth b(...) A=@pa in {self A(5)} end
end
```

Control de encapsulación (8)

Átomos o nombres como cabezas de métodos

- Los átomos son visibles a través de todo el programa y los nombres sólo son visibles en el alcance léxico de su creación.
- Use nombres para métodos internos, y use átomos para métodos externos.

¿Cómo lo hacen los lenguajes más populares?

Smalltalk, C++, y Java sólo soportan átomos como cabezas de métodos; no tienen soporte para los nombres.
El enfoque basado en nombres no es popular todavía.

Control de encapsulación (8)

Átomos o nombres como cabezas de métodos

- Los átomos son visibles a través de todo el programa y los nombres sólo son visibles en el alcance léxico de su creación.
- Use nombres para métodos internos, y use átomos para métodos externos.

¿Cómo lo hacen los lenguajes más populares?

Smalltalk, C++, y Java sólo soportan átomos como cabezas de métodos; no tienen soporte para los nombres.
El enfoque basado en nombres no es popular todavía.

Control de encapsulación (9)

Sencillez de los átomos

- Los átomos se identifican únicamente por sus representaciones impresas.
- Con los nombres esto es más difícil: el programa mismo tiene que pasar de alguna manera el nombre a quien desea invocar el método.

Ventajas de los nombres

- Imposible tener conflictos con la herencia (sencilla o múltiple).
- La encapsulación se puede manejar mejor, pues una referencia a un objeto no da necesariamente derecho a invocar los métodos del objeto.
- Los nombres pueden tener soporte sintáctico para simplificar su uso.