Lenguajes de Programación I

Orientación a Objetos

Ernesto Hernández-Novich
<emhn@usb.ve>

Universidad "Simón Bolívar"

Módulos como abstracción de datos

Alcance controla la posibilidad de compartir datos

- Módulos llevan a programar aprovechando tipos de datos abstractos.
- Los módulos como tipos (**clase**) elevan el nivel de abstracción.
  - Crear y destruir instancias del módulo (**objetos**).
  - Invocar rutinas a través de una instancia (**métodos** o **mensajes**).
  - Refinar representación (**herencia**).
  - Refinar comportamiento (**asociación dinámica de métodos**).

Cualquier lenguaje que provea estas técnicas de programación será *Orientado a Objetos*.

Mitos y concepciones erróneas

. . . de la ignorancia o de la malicia

- **No** nació como evolución de los módulos sino directamente – Simula.
- Sin privacidad de datos **si puede** haber orientación a objetos – por técnica como en Simula o Perl.
- Con tipos dinámicos **si puede** haber orientación a objetos – por lenguaje como en Smalltalk o técnica como en Perl.
- En lenguajes funcionales **si puede** haber orientación a objetos.
  - Common LISP Object System (CLOS) – *lejos* el mejor modelo OO.
  - Moose para Perl es prácticamente idéntico.
- La herencia múltiple **no** es mala (del todo).

No es la solución ideal para todos los problemas de programación.

Ventajas de la Programación Orientada a Objetos

Espero estar predicándole al coro

- Reduce la carga conceptual – minimizar la cantidad de detalle que el programador tiene que conocer en un momento dado.
- Facilita la contención de problemas
  - Impidiendo el uso de componentes de manera equivocada.
  - Limitando la porción de programa a revisar al corregir errores.
- Eleva el nivel de independencia entre componentes del programa.
  - Facilita el desarrollo en equipo.
  - Simplifica la evolución interna sin alterar sistemas dependientes.
  - Fomenta la reutilización de código.

Mejora las oportunidades de reutilizar código facilitando el refinamiento de abstracciones existentes.

Terminología

- **Clase** – modelo de abstracción.
- Se **instancia** en **objetos**.
- Cada clase contiene **miembros**.
  - Estado – atributos, miembros, campos, *slots*, ...
  - Comportamiento – métodos o mensajes.
- Cada método necesita saber sobre cuál instancia particular operar
  - Referencia predefinida – `this`, `self`, `current`, ...
  - Referencia arbitraria pasada como argumento – Perl.
- Métodos especiales para la elaboración (**constructores**).
  - Usualmente se llaman igual que la clase – C++, Java, C#.
  - Usados explícitamente – `new`.
- Métodos especiales para la destrucción (**destructores**).
  - Típicos en lenguajes sin recolección de basura – C++.
  - Necesarios si los atributos son complejos.

Definición de clases

- Típicamente
  - Una clase *enteramente* contenida en un archivo.
  - Una clase visible por módulo.
  - Reglas de alcance estático se extienden considerando que una clase define un bloque léxicográfico.
- Declaración separada de definición – C++, Modula, Java.
  - Favorecer la compilación separada.
  - Reducir la polución del espacio de nombres.
  - Declaración debe ser suficiente para programador y compilador.
- Definición distribuida entre varios archivos – Ruby, Perl.
  - Posible extender la clase `Foo` desde *cualquier* punto de mi programa – básicamente agregar o redefinir métodos.
  - *Monkey-patching* – "la magia del hombre blanco".

Visibilidad

¿Qué se exporta hacia alcances externos?

- Permite decidir la disponibilidad de los atributos
  - **Públicos** – aquellos visibles al usuario de la abstracción.
  - **Privados** – aquellos requeridos por la implantación.
- Privado a menos que se indique lo contrario – Java, Ruby, C#.
- Público a menos que se indique lo contrario – C++, Python, Perl.
- Cuando hay compilación separada (C++) es necesario utilizar un operador de *resolución de alcance*.

```
void some_class::some_method( ... )
```

Acceso a los atributos

... un cálculo costoso de desplazamientos

- Los atributos **deberían** ser privados, y cuando lo son...
  - Métodos de acceso a los atributos – *accesor methods* o *setters/getters*.
    - Subrutinas muy pequeñas.
    - Ejecutan con mucha frecuencia – costo de llamada afecta desempeño.
    - *Inlining* – optimización en lenguajes compilados.
  - Generadas automáticamente vía sintaxis – C#, Ruby, Perl.

```
class Meta
  attr_reader  :foo  # Solamente leer
  attr_writer  :bar  # Solamente escribir
  attr_accesor :baz  # Leer y escribir
end
```

- Métodos de índice en clases contenedoras
  - Hacen lucir los objetos como arreglos.
  - Acceso *l-value* o *r-value* según el contexto.

Refinación de Abstracciones

Herencia

- Una clase B puede **derivar** a una clase A.
  - B es una **subclase** de A, y A es una **superclase** de B.
  - Todo objeto de la clase B es un objeto de la clase A.
  - B **hereda** atributos y métodos de A.
  - B **agrega** atributos y métodos – incluso los **cambia**
- Esto crea **Jerarquías de Clases**.
  - Jerarquía estándar provista por el lenguaje.
  - La mayoría ofrecen un árbol con una raíz común – Smalltalk, Java, C#.
  - Algunos lenguajes proveen un bosque – C++.
- Clases de Propósito General o Abstractas
  - Representar datos genéricos – refinar después.
  - Representar comportamientos genéricos – refinar después.
  - Para ambas cosas simultáneamente.

Redefinición

Refinación o extensión del comportamiento

- Para *redefinir* miembros basta con declararlos.
- No se pierde el acceso al miembro de la superclase.
  - Operador de resolución de alcance – C++
  - Palabra reservada – `super` o `base` en Java, Smalltalk.
  - Ambas opciones – Perl, Ruby.
  - Renombrar el método de la superclase dentro de la clase – Eiffel

Encapsulamiento con Módulos

- Declaración del módulo vs. implantación del módulo.
  - Si el módulo exporta un tipo T, el código usuario sólo puede usar objetos del tipo T con las funciones exportadas por el módulo.
  - Podrían implantarse asignaciones y comparaciones *bit-a-bit*.
  - Para compilación separada y modelo de referencia, basta tener *nombre* del tipo y firmas de subrutinas exportadas por el módulo.
- Uso de las reglas de alcance.
  - Variables globales al módulo, invisibles afuera.
  - Variables globales al módulo, visibles afuera.

Encapsulamiento con Clases

La herencia complica el acceso

- Un miembro público de una clase, ¿sigue siendo público en subclases?
- Un miembro privado de una clase, ¿es visible en una subclase?
- Lo habitual es que las subclases puedan *reducir* la visibilidad.
  - Salvo en Eiffel donde también pueden *aumentarla* gracias a la *visibilidad selectiva*...
  - ...o en Java y C#, que no permiten cambiar la visibilidad de miembros.
- Nada de privacidad *explícita* – Python, Perl.

C++ – heredar sin necesitar de publicar

. . . cooperación sin publicación

- Miembros "protegidos" (`protected`) en una clase – visibles a *métodos* de su clase o subclases.
- También se puede proteger una superclase al extenderla

```
class subclass : protected superclass { ... }
```

  - Miembros *públicos* de `superclass` se convierten en `protected`.
  - No pueden desprotegerse de nuevo.

C++ – privatizar partes públicas

. . . para reducir la vulnerabilidad

- Una clase puede usar a su superclase de forma privada – hereda de la superclase, pero la mantiene "invisible"

```
class subclass : private superclass {
  public:
    using superclass::foo;
    using superclass::bar;
    void baz(...);
};
```

  - Partes *públicas* de la superclase se vuelven privadas.
  - `using` para declarar explícitamente las que se utilizarán.
- El *especializador reduce* la visibilidad selectivamente.

C++ – acceso desde fuera de la jerarquía

"Amig@s con derecho"

- Función *cualquiera* con acceso a partes privadas

```
class Foo { friend void touchy(Foo f&); }
```

  - `touchy` no es un método de `Foo` – es una función cualquiera.
  - Función `touchy` puede acceder directamente a las partes privadas (y protegidas) de instancias de `Foo`.
- Clases `Foo` y `Bar` en jerarquías independientes.

```
class Foo { friend class Bar; }
```

  - Métodos de `Bar` pueden acceder directamente partes privadas (y protegidas) de `Foo`.
  - Las amistades no son recíprocas, ni transitivas.

Bibliografía

- [*Scott*]
  - Secciones 9.1 y 9.2
  - Ejercicios y Exploraciones
- Página de Wikipedia sobre Clases