Modulos

Módulos

Introducción

- Uno de los principios aceptados sobre buen diseño es la eliminación de duplicados innecesarios.
- Trataremos de lograr que cada concepto en nuestra aplicación sea expresado sólo una vez en el código.

Módulos

Herencia

- Permite crear clases que son un refinamiento o especialización de otra clase.
- A esta clase se la llama *subclase* de la original.
- A la clase original se la llama *superclase* de la subclase.
- También se utilizan los términos: clase padre y clase hija.
- El mecanismo de herencia es simple:
  - Se heredan las capacidades del padre.
  - Los métodos de instancia y clase del padre estarán disponibles en los hijos.

Módulos

Ejemplo de herencia

```ruby
class Parent
  def say_hello
    puts "Hello from #{self}"
  end
end
```

```ruby
p = Parent.new
p.say_hello
```

```ruby
class Child < Parent
end
```

```ruby
c = Child.new
c.say_hello
```

Módulos

Conociendo la herencia

El método `superclass` devuelve la clase padre

```
puts "The superclass of Child is #{Child.superclass}"
puts "The superclass of Parent is #{Parent.superclass}"
puts "The superclass of Object is #{Object.superclass}"
```

- *Si no se define superclase, Ruby asume* `Object`
- `to_s` *está definido aquí*
- `BasicObject` *es utilizado en metaprogramación.*
- *Su padre es* `nil`
- *Es la raíz: todas las clases lo tendran como ancestro*

Módulos

Ejemplo

- `GServer` es un servidor TCP/IP genérico.
- Agregaremos funcionalidad básica a nuestro servicio subclaseando GServer.
  - El servicio mostrará las últimas lineas del archivo de logs del sistema: `/var/log/syslog`
- GServer manipula todo lo relacionado a sockets TCP.
  - Sólo indicaremos el puerto en la inicialización.
  - Cuando un cliente se conecte, el objeto GServer invocará al método `serve`.
  - GServer no hace nada en el método que implementa `serve`.

Módulos

```
require 'gserver'
class LogServer < GServer

  def initialize
    super(12345)
  end

  def serve(client)
    remote_ip = client.peeraddr
    log "Connected from #{remote_ip[2]}:#{remote_ip[1]}"
    client.puts get_end_of_log_file
  end
```

Módulos

```
  private
    def get_end_of_log_file
      File.open("/var/log/syslog") do |log|
        # back up 1000 characters from end
        log.seek(-1000, IO::SEEK_END)
        # ignore partial line
        log.gets
        # and return rest
        log.read
      end
    end
end

server = LogServer.new
server.start.join
```

Módulos

¿Cómo hemos usado la herencia?

- `LogServer` hereda de `GServer`.
- Esto indica que:
  - LogServer es un GServer, compartiendo toda su funcionalidad.
  - LogServer es una especialización.

Módulos

¿Cómo usamos la herencia?

`initialize`

- Se fuerza el puerto a 12345.
- El puerto es un parámetro del constructor de GServer.
- Para invocar el método constructor del padre, utilizamos `super`.

`serve`

- El padre asume que será subclaseado invocando un método que será implementado por sus hijos.
- Esto permite a la clase padre implementar lo más pesado del procesamiento y delegar a los hijos mediante callbacks funcionalidad extra.

- *Veremos más adelante que esta práctica **muy común en OO** no la convierte en **un buen diseño***
- *En su lugar veremos **mixins***
- *Pero para explicar mixins, antes tenemos que explicar **módulos***

Módulos

- Los módulos son una forma de agrupar métodos, clases y constantes.
- Proveen dos beneficios:
  - Proveen **namespaces** y previenen el solapamiento de nombres.
  - Son la clave de los **mixins**.

Módulos

Namespaces

- A medida que los programas crecen, surge código reusable.
- Es así como aparecen las librerías.
- Deseamos agrupar en archivos diferentes estas rutinas de forma tal de poder reusarlas en programas distintos.
- Generalmente estas rutinas pertenecerán a una clase, o grupos de clases interrelacionadas, que podríamos disponer en un único archivo.
  - Sin embargo, a veces queremos agrupar cosas que no necesariamente forman una clase.

Módulos

Namespaces

- Como una primer idea, podríamos pensar en disponer todos los archivos que componen nuestra librería y luego cargar el archivo en nuestro programa cuando lo necesite.
- Esta idea tiene un problema si definimos funciones con nombres que son iguales a los de otra librería.

Módulos

Ejemplo

```ruby
module Trig
  PI = 3.141592654


  def self.sin(x)
  # ..
  end

  def self.cos(x)
  # ..
  end
end
```

```ruby
module Moral
  VERY_BAD = 0
  BAD = 1

  def self.sin(badness)
  # ...
  end

end
```

Módulos

¿Como se usa?

```
y = Trig.sin(Trig::PI/4)
wrongdoing = Moral.sin(Moral::VERY_BAD)
```

- *Así como en los métodos de clase, se invocan los métodos de un módulo precediéndolos con el nombre del módulo y un punto.*
- *Las constantes se referencian con el nombre del módulo y doble dos puntos (::).*

Módulos

Mixins

- En el ejemplo reciente, definimos métodos del módulo que prefijábamos con el nombre del módulo: `self.cos`.
- La primer asociación es que los métodos de un módulo son como métodos de clase.
- La siguiente pregunta sería: *Si los métodos del módulo son como métodos de clase, qué serían los métodos de instancia de un módulo?*.

Módulos

Mixins

- Un módulo **no puede tener instancias** porque no es una clase.
- Podremos **incluir** un módulo a una definición de clase.
- Cuando esto sucede, los métodos de instancia definidos en el módulo son incluidos como métodos de instancia de la clase. Se **mezclan** (mixed in).
- En efecto, los módulos mixins se comportan como superclases.

Módulos - Ejemplo

```ruby
module Debug
  def who_am_i?
    "#{self.class.name}(\##{self.object_id}):#{self.to_s}"
  end
end

class Phonograph
  include Debug
  def initialize(n); @n=n; end
  def to_s; @n; end
end
```

Módulos - Ejemplo

```
class EightTrack
  include Debug
  def initialize(n); @n=n; end
  def to_s; @n; end
end

ph = Phonograph.new("West End Blues")
et = EightTrack.new("Surrealistic Pillow")
ph.who_am_i?
et.who_am_i?
```

Módulos

El uso de include

- El `include` en Ruby agrega una referencia al módulo que agregará nuevos métodos a nuestra clase.
- Si varias clases incluyen el mismo módulo, todas tendran referencias al mismo.
- Si modificamos el módulo durante la ejecución del programa, todas las clases que incluían el módulo tomarán los cambios automáticamente.

Módulos

El potencial

- El potencial real de los mixins se obtiene cuando el código de un mixin interactúa con código de una clase que lo utiliza.
- Analizamos el caso de un mixin que es parte de la librería estándar de Ruby, `Comparable`:
  - Agrega los operadores de comparación: `<`, `<=`, `==`, `>=`, `>`.
  - Agrega el método `between?`.
  - Asume que la clase que utilice este mixin, implementará el método `<=>`.

Módulos

Probamos con Person

```ruby
class Person
  include Comparable
  attr_reader :name
  def initialize(name)
    @name = name
  end
  def to_s
    "#{@name}"
  end
  def <=>(other)
    self.name <=> other.name
  end
end
```

```ruby
p1 = Person.new("Matz")
p2 = Person.new("Guido")
p3 = Person.new("Larry")
[p1, p2, p3].sort
```

Iteradores y Enumerable

- Si queremos que nuestra clase entienda los iteradores `each`, `include?`, `find_all?`, `map`, `inject`, `count`, etc.
  - Incluimos el módulo `Enumerable`.
  - Implementamos el iterador `each`.
- Si además los elementos de nuestra colección implementan `<=>` entonces dispondremos de:
  - `min`
  - `max`
  - `sort`

Composición de módulos

Creamos nuestra clase Enumerable

```
class VowelFinder
  include Enumerable
  def initialize(string)
    @string = string
  end
  def each
    @string.scan(/[aeiou]/i) do |vowel|
      yield vowel
    end
  end
end
vf = VowelFinder.new "El murcielago tiene todas"
vf.inject(:+)
```

Ahora nuestra clase funciona igual que otras colecciones:

```
[ 1, 2, 3, 4, 5 ].inject(:+)
( 'a'..'m').inject(:+)
```

Creamos el módulo Summable

```ruby
module Summable
  def sum
    inject(:+)
  end
end
```

Lo aplicamos a las clases del ejemplo

```ruby
class Array; include Summable; end
class Range; include Summable; end
class VowelFinder; include Summable; end

[ 1, 2, 3, 4, 5 ].sum
('a'..'m').sum
vf.sum
```

Variables en mixins

```ruby
module Observable

  def observers
    @observer_list ||= []
  end

  def add_observer(obj)
    observers << obj
  end

  def notify_observers
    observers.each {|o| o.update }
  end
end
```

- *En ruby las variables de instancia se crean cuando se nombran por primera vez.*
- *Esto significa que un Mixin podrá crear variables de instancia si las nombra por primera vez en la clase.*

Módulos

Variables en mixins

- Sin embargo, este uso es **riesgoso**.
- Los nombres de las variables pueden colisionar con otro nombre de la clase u otros módulos.
- Un programa que caiga en este escenario dará resultados erróneos y difíciles de rastrear.

Módulos

Solución

- La mayoría de las veces, los modulos Mixins no usan variables de instancia, sino accessors.
- En caso de necesitarlo, utilizar nombres que se prefijen con el nombre del módulo por ejemplo.

Módulos

Resolución de nombres

¿Cómo se resuelve el nombre de un método que es el mismo en la clase, que es implementado en la superclase y además definido en uno o varios módulos incluidos?

Módulos

Resolución de nombres

- Primero se busca si la clase del objeto lo implementa.
- Luego en los mixins incluidos por la clase. *Si tiene varios módulos, el último será el considerado*.
- Luego en la superclase.

Caso 1 Child

```ruby
module MyModule
  def test
    "Module"
  end
end
class Parent
  def test
    "Parent"
  end
end
class Child < Parent
  include MyModule
  def test
    "Child"
  end
end
t = Child.new
p t.test
```

Caso 2
Module

```ruby
module MyModule
  def test
    "Module"
  end
end
class Parent
  def test
    "Parent"
  end
end
class Child < Parent
  include MyModule
end
t = Child.new
p t.test
```

Caso 3
Parent

```ruby
module MyModule
  def test1
    "Module"
  end
end
class Parent
  def test
    "Parent"
  end
end
class Child < Parent
  include MyModule
end
t = Child.new
p t.test
```

Invocación de métodos

Si analizamos la salida de `#ancestors` veremos la cadena Clases por la que se buscará por un método apropiado. Veamos el siguiente ejemplo:

```ruby
module Logging
  def log(level, message)
      puts "#{level}: #{message}"
  end
end

class Service
  include Logging
end

Service.ancestors
```

La salida es:

`[Service, Logging, Object, Kernel, BasicObject]`

Notar que Logging se interpone entre Object y Service

Ancestros y módulos

Si agregamos otro Modulo a la clase anterior:

```
Service.include Comparable
```

Podemos verificar que el último módulo incluido aparece detrás de la clase `Service`, explicando así las precedencias explicadas como casos anteriormente.

```
[Service, Comparable, Logging, Object, Kernel, BasicObject]
```

Extend

Utilizar `#extend` en una clase importará los métodos del módulo como métodos de clase.

En vez de actualizar la lista de ancestros, `#extend` modificará el singleton de la clase extendida, agregando métodos de clase.

En general, se utilizará `#include` en una clase para extender el comportamiento con métodos de instancia, pero a su vez podría necesitarse usar `#extend` para extender los métodos de clase. Entonces se necesitarían **dos modulos diferentes para cada caso.**

Extend

La siguiente estrategia permite crear dos módulos para extender clases y objetos en un mismo código:

```ruby
module Logging
  module ClassMethods
    def logging_enabled?
      true
    end
  end

  def self.included(base)
    base.extend(ClassMethods)
  end

  def log(level, message)
    puts "#{level}: #{message}"
  end
end
```

Extend vía include

Usando el ejemplo anterior, al realizar:

```
String.include Logging

String.logging_enabled?

'Test'.log 'ERROR', 'test message'
```

Conclusión

Herencia, Mixins y Diseño

Herencia y Mixins ambos permiten escribir código en un único lugar.

¿Cuándo usar cada uno?

Conclusión

Herencia, Mixins y Diseño

- El uso de herencia debe aplicarse cuando se cumple la propiedade **es un**.
- La herencia puede asociarse con la creación de clases, que sería como agregar nuevos tipos al lenguaje.
- Al usar herencia **deberíamos en todo momento poder reemplazar un objeto de la superclase por un objeto de la subclase**. *Los hijos deben hacer honor a los contratos asumidos por el padre*.

Conclusión

Herencia, Mixins y Diseño

- La herencia representa un gran acomplamiento entre dos componentes.
  - Cambiar la herencia sería algo complejo en cualquier programa mediano.
- **Debemos utilizar composición cada vez que encontramos una relación:** A *usa* B o A *tiene un* B