Classboxes, nested methods, and real private methods

Shugo Maeda

2010-11-12

Self introduction

- Shugo Maeda
- A Ruby committer
- A director of Network Applied Communication Laboratory Ltd. (NaCl)
- The co-chairperson of the Ruby Association

Where am I from?

- Matsue, Shimane, Japan
  - A sister city of New Orleans

We are different

Please accept us

Topics

- were supposed to be:
  - classboxes,
  - nested methods,
  - and real private methods
- But...

Topic

- A new feature "Refinements"

What are Classboxes?

- A way to extend classes

How to extend classes in Ruby?

- Subclassing
- Mix-in
- Singleton methods
- Open classes

Subclassing

```
class Person
  attr_accessor :name
end

class Employee < Person
  attr_accessor :monthly_salary
end
```

Aspects of subclassing

- Normal single inheritance
- Subclassing affects only instances of the subclasses
- Implementation-only inheritance
  - Violations of LSP

LSP

- Liskov Substitution Principle
- An instance of a subtype must behave like an instance of the supertype of the subtype
  - An instance of the supertype can be substituted with an instance of the subtype

An example of LSP

```
def print_name(person)
  puts person.name
end

shugo = Person.new
shugo.name = "Shugo Maeda"
print_name(shugo) #=> Shugo Maeda
matz = Employee.new
matz.name = "Yukihiro Matsumoto"
print_name(matz) #=> Yukihiro Matsumoto
```


A typical LSP violation

```ruby
class Rectangle
  attr_accessor :width, :height
end

class Square < Rectangle
  def set_size(x) @height = @width = x end
  alias width= set_size
  alias height= set_size
end

def set_size(rect)
  rect.width = 80; rect.height = 60
end
square = Square.new
set_size(square)
p square.width #=> not 80, but 60!
```

A Ruby-specific LSP violation

```ruby
class Employee < Person
  undef name
end

def print_name(person)
  puts person.name
end

matz = Employee.new
matz.name = "Yukihiro Matsumoto"
print_name(matz) #=> undefined method `name'...
```

Subclassing != Subtyping

- Implementation-only inheritance
- Duck typing

Mix-in

```
class; Stream; ... end
module Readable; ... end
module Writable; ... end

class ReadStream < Stream
  include Readable
end
class WriteStream < Stream
  include Writable
end
class ReadWriteStream
  include Writable, Readable
end
```

Aspects of mix-in

- Limited multiple inheritance
  - Only modules can be multiply inherited
  - A module has no instances
- Modules are also used as namespaces for constants

Singleton methods

```
matz = Person.new
def matz.design_ruby
  ...
end
matz.design_ruby
shugo = Person.new
shugo.design_ruby #=> NoMethodError
```

Aspects of singleton methods

- Clients of a class can extend the behavior of an instance of the class
- A singleton method defines the behavior of only one particular instance
- Some objects cannot have singleton methods
  - e.g., instances of Integer

Open classes

```ruby
# reopen Person, and add code
class Person
  attr_accessor :age
end
shugo = Person.new
shugo.name = "Shugo Maeda"
shugo.age = 34
```

Aspects of open classes

- Clients of a class can extend the behavior of instances of the class
- Classes are extended globally

Applications of open classes

- Ruby on Rails
  - ActiveSupport
  - Plugins
- jcode
- mathn

LSP and open classes

- s/subtype/class after reopen/g
- s/supertype/class before reopen/g
- Instances of a class after a reopen must behave like instances of the class before the reopen

an LSP violation

```
p 1 / 2 #=> 0
require "mathn"
p 1 / 2 #=> (1/2)
```

Summary

Subclassing	not by clients
Mix-in	not by clients
Singleton methods	per object
Open classes	global

Extensibility and Modularity

- Subclassing, mix-in, and singleton methods are less extensible
- Open classes are less modular

What we need

- Class extensions
  - by clients
  - per class
  - local

Possible solutions

- selector namespace
- Classboxes

selector namespace

- Implemented in SmallScript and ECMAScript 4
- A namespace of method names (selectors)
- A namespace can be imported into other namespaces
- **Lexically scoped**

Classboxes

- Implemented in Squeak and Java
- A classbox is a module where classes are defined and/or extended
- A classbox can be imported into other classboxes
- **Dynamically scoped**
  - called **local rebinding**

An example of Classbox/J

```
package Foo;
public class Foo { ... }

package Bar;
import Foo;
refine Foo { public void bar() { ... } }

package Baz;
import Bar;
public class Baz {
  public static void main(String[] args) {
    new Foo().bar();
  }
}
```

An example of local rebinding

```
package Foo;
public class Foo {
  public void bar() { System.out.println("original"); }
  public void call_bar() { bar(); }
}

package Bar;
import Foo;
refine Foo { public void bar() { System.out.println("refined"); } }

package Baz;
import Bar;
public class Baz {
  public static void main(String[] args) {
    new Foo().call_bar();
  }
}
```


Is local rebinding needed?

- Local rebinding is less modular
  - Callees might expect the original behavior
- Singleton methods and open classes can be alternatives
  - However, effective scopes are different

Refinements

- A newly implemented feature of Ruby
  - Not merged into the official Ruby repository
- Refinements of classes are defined per module
- Effective scopes are explicitly specified
- no local rebinding
- Classbox/J like syntax

An example of Refinements

```
module MathN
  refine Fixnum do
    def /(other) quo(other) end
  end
end

class Foo
  using MathN

  def bar
    p 1 / 2 #=> (1/2)
  end
end
p 1 / 2 #=> 0
```

Demo

Module#refine

- refine(klass, &block)
- Additional or overriding methods of klass are defined in block
  - a set of such methods is called a **refinement**
- Activated only in the receiver module, and scopes where the moduleis imported by **using**
- refine can also be invoked on classes

Class local refinements

```
class Foo
  refine Fixnum do
    def /(other) quo(other) end
  end

  def bar
    p 1 / 2 #=> (1/2)
  end
end
p 1 / 2 #=> 0
```


Kernel#using

- using(mod)
- using imports refinements defined in mod
- Refinements are activated only in a file, module, class, or method where using is invoked
  - lexically scoped

An example of using

```
using A  # A is activated in this file

module Foo
  using B  # B is activated in Foo (including Foo::Bar)

  class Bar
    using C  # C is activated in Foo::Bar

    def baz
      using D  # D is activated in this method
    end
  end
end
```


Module#using

- using(mod)
- Module#using overrides Kernel#using
- The basic behavior is the same as Kernel#using
- Besides, Module#using supports reopen and inheritance

An example of Module#using

```
module A; refine(X) { ... } end
module B; refine(X) { ... } end
class Foo; using A end
class Foo
  # A is activated in a reopened definition of Foo
end
module Bar
  using B
  class Baz < Foo
    # A is activated in a subclass Baz of Foo
    # A has higher precedence than B
  end
end
```


using and include

```
module A; refine(X) { ... } end
module Foo; using A end
class Bar
  include Foo
  # include does not activate A
end
```


Precedence of refinements

- Refinements imported in subclasses have higher precedence
- Later imported refinements have higher precedence
- Refinements imported in the current class or its superclasses have higher precedence than refinements imported in outer scopes
- If a refined class has a subclass, methods in the subclass have higher precedence than those in the refinement

An example of precedence

```
class Foo; end
module Bar; refine Foo do end end
module Baz; refine Foo do end end
class Quux < Foo; end
class Quuux
  using Bar
end
module Quuuux
  using Baz
  class  Quuuuux < Quuux
    def foo
      # Quux -> Bar -> Baz -> Foo
      Quux.new.do_something
    end
  end
end
```

Using original features

- super in a refined method invokes the original method, if any
- If there is a method with the same name in a previously importedrefinements, super invokes the method
- In a refined method, constants and class variables in the original class is also accessible

An example of super

```
module FloorExtension
  refine Float do
    def floor(d=nil)
      if d
        x = 10 ** d
        return (self * x).floor.to_f / x
      else
        return super()
      end
    end
  end
end
using FloorExtension
p 1.234567890.floor #=> 1
p 1.234567890.floor(4) #=> 1.2345
```


special eval

- Refinements are also activated in instance_eval, module_eval, and class_eval

An example of special eval

```
class Foo
  using MathN
end
Foo.class_eval do
  p 1 / 2 #=> (1/2)
end
Foo.new.instance_eval do
  p 1 / 2 #=> (1/2)
end
```


Compatibility

- No syntax extensions
  - No new keywords
- The behavior of code without refinements never change
- However, if existing code has a method named refine or using, it may cause some problems

Applications of refinements

- Refinements of built-in classes
- Internal DSLs
- Nested methods

Refinements of built-in classes

- Refinements are activated in particular scopes
- So you can violate LSP like MathN
- Refinement inheritance is useful for frameworks

Example

```
class ApplicationController < ActionController::Base
  using ActiveSupport::All
  protect_from_forgery
end

class ArticlesController < ApplicationController
  def index
    @articles = Article.where("created_at > ?", 3.days.ago)
  end
end
```


Internal DSLs

- Methods for DSLs need not be available outside DSLs
- So these methods can be defined in refinements
- instance_eval and module_eval are useful for DSLs

Example

```
module Expectations
  refine Object do
    def should ... end
    ...
  end
end

def it(msg, &block)
  Expectations.module_eval(&block)
end

it "returns 0 for all gutter game" do
  bowling = Bowling.new
  20.times { bowling.hit(0) }
  bowling.score.should == 0
end
```


Nested methods

```
def fact(n)
  # fact_iter is defined in refinements
  # available only in fact
  def fact_iter(product, counter, max_count)
    if counter > max_count
      product
    else
      fact_iter(counter * product,
                counter + 1, max_count)
    end
  end

  fact_iter(1, 1, n)
end
```


Benchmark

- make benchmark (5 times)
- Environment
  - CPU: Intel Core 2 Duo U7600 1.2GHz
  - RAM: 2GB
  - OS: Linux 2.6.34 (Ubuntu 10.04)

Additional benchmarks

- For refinements
  - bm_ref_factorial.rb
  - bm_ref_fib.rb
- For nested methods
  - bm_ref_factorial2.rb

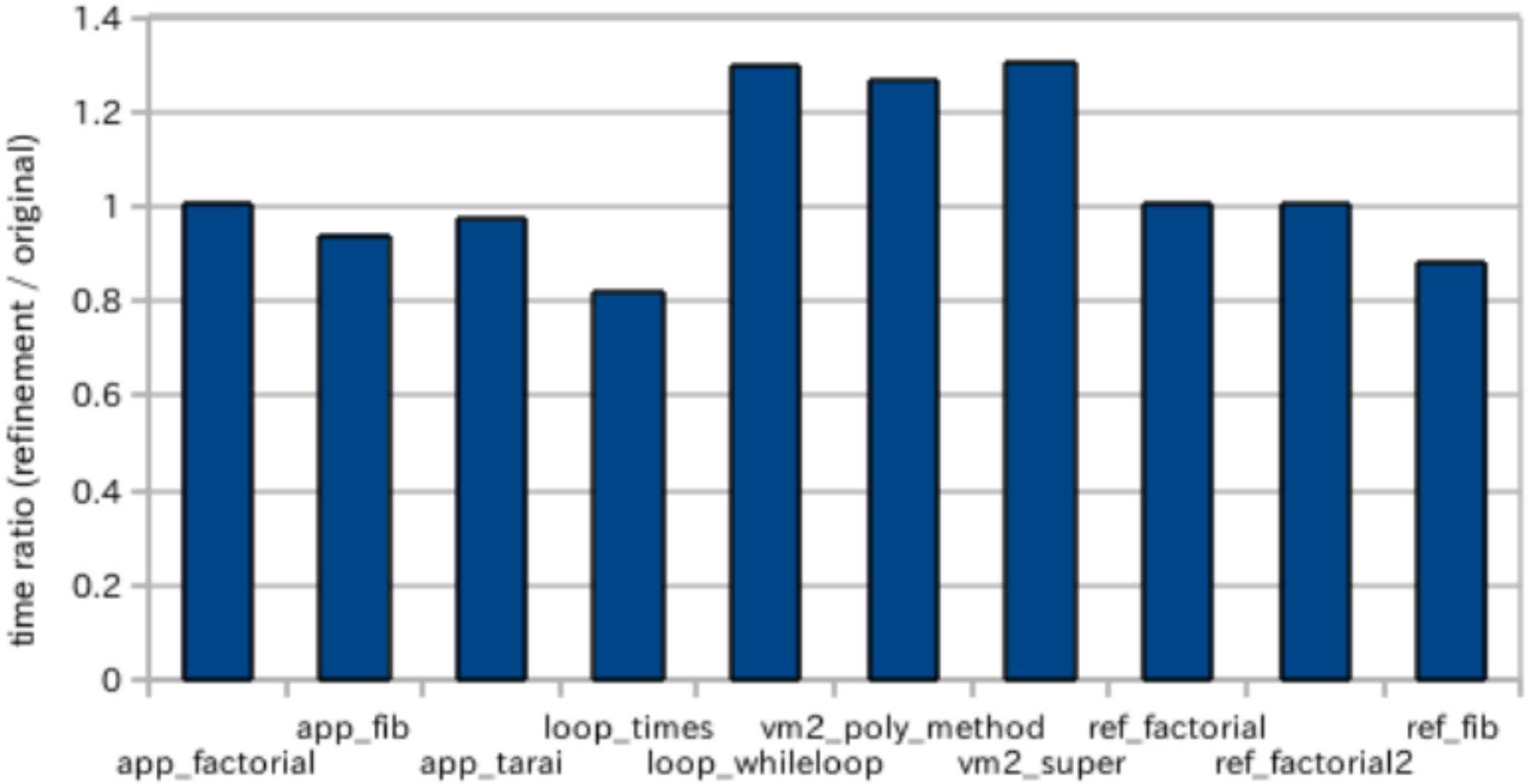
bm_ref_factorial.rb

```
if defined?(using)
  module Fact
    refine Integer do
      def fact ... end
    end
  end
  using Fact
else
  class Integer
    def fact ... end
  end
end
```


Benchmark result

- Average 2.5% slower than the original Ruby

Samples


time ratio (refinement / original)
1.4
1.2
1
0.8
0.6
0.4
0.2
0
app_factorial
app_fib
app_tarai
loop_times
loop_whileloop
vm2_poly_method
vm2_super
ref_factorial
ref_factorial2
ref_fib

Considerations

- Should include and using be integrated?
- Should modules be refinable?
- Should singleton methods be refinable?
- Implementation improvement

Patch

- http://shugo.net/tmp/refinement-r29498-20101109.diff

Conclusion

- Refinements achieve a good balance between extensibility and modularity

Thank you

- Any questions?