Ruby trunk - Feature #7487

Cutting through the issues with Refinements

12/01/2012 08:55 AM - trans (Thomas Sawyer)

Status:	Rejected
Priority:	Normal
Assignee:	
Target version:	Next Major

Description

=begin
In issue #4085, there has been a long somewhat contentious discussion about Refinements. While it seems that everyone agrees they have merit, no one seems to have a concrete idea about how they should actually work. There are all sorts of complicated questions and edge case being flailed about. And though Matz is determined to stick to the feature freeze and include Refinements in Ruby 2.0, he has had to greatly peel back there capabilities and scope --it's quite a large change this close to release, and no one is still at all sure that this new limited design is right either. All this is rather unfortunate, not just because it means Ruby 2.0 is probably going to have a half-baked feature that is certain to change substantially by 2.1, but even more so because we were having pretty much the very same discussion six years ago!

In late 2003, there was a long discussion about method wrapping and aspect-oriented programming (AOP) on ruby-talk[1,2,3]. The conversation grew out of an early notion Matz (and maybe Jim Weirich?) had about wrapping methods for Rite. Remember Rite? That was the codename of the original Ruby 2.0. Back then Matz offered up the idea of using (({:pre})) and (({:post})) hooks to wrap methods. The notation was something like:

```
class C
  def foo
    print "foo"
  end

  def foo:pre
    print "before"
  end

  def foo:post
    print "after"
  end
end

C.new.foo
=> "beforefooafter"
```

Many of the same questions were asked about these method "hooks" that are now being asked about refinements -- "do they stack", "what happens if we remove the main method", "how are they applied to the object hierarchy?", and so on. While at first glance these yesteryear method hooks and today's refinements may seem quite different, they are actually quite related, which will become clear in a moment.

It was through these threads that Peter Vanbroekhoven and myself began an extensive conversation on AOP for Ruby, based originally on his idea of method wrapping via a module in much the same way as one uses include. He originally called the method that handled this simply "wrap". We know it today as (({prepend})). So you can thank Peter for that whole idea[4]. So, we were both very interested in the concept of AOP and with these early notions in mind we decided to take our conversation off-list with the hope of working out the ideal design for bringing AOP to Ruby. Truth is, we did even better than that.

Peter and I continued to discuss AOP over the following year trading hundreds of communiques exploring every nook and cranny of the concept. Indeed, at a certain point I think Peter was quite tired of it, as I had the tendency to review a concept again and again and again just to make sure we didn't miss anything. But as long and drawn out and as detailed as the whole process was, I think we were far the better for it. We developed a very good understanding of the whole matter. In the course of these conversations, I came upon the idea of the ((*transparent-subclass*)). It was little more than a variation on Peters original wrap idea but it had all the hallmarks of a fundamental OOP concept. With further discussion we agreed that this was a solid corner stone upon which to lay AOP --and not just for Ruby, but for OOP in general! I gave it a name, the "Cut".

From there Peter and I toiled to write an RCR (Ruby Change Request) to introduce the concept to the Ruby community. (Yes, in those days there was such a thing.) We wrote and edited our RCR on the old RubyGarden.org wiki. After many dozens of revisions and over a year after our original discussion!, we finally had our proposal. To top it off Peter even put together a preliminary implementation patch for Ruby, as I put together a pure Ruby (and thus limited) demonstration library. You can find that code and the

((|)) today on ((|)).

Now all of this pre-story leads up to what I want to suggest now. I would like the Cuts RCR to be reconsidered[5], on the merits that it is precisely the well thought out, solid foundation, with a real OOP design, that can serve as the basis of implementation for both prepend and refinements, as well as all other aspect-oriented designs patterns developers wish it construct.

So how does this work? How can prepend and refinements be implemented via cuts?

By simple analogy, prepend is to include as cuts are to classes. What this means implementation-wise is that just as modules are included in the class hierarchy via proxy classes, modules would be prepended into the hierarchy via proxy cuts. Its a simple symmetry that provides the proper behavior.

For refinements we need only add a conditional proviso to cuts --a cut would only be applicable if the pertinent scope is using the refinement. The condition could even be customizable for other uses lending a great deal of flexibility, power and convenience in aspect-oriented designs. To make this idea clear, here is example code for how a cut can be used as a refinement:

```
cut MyRefinement < String
  def self.apply?(binding)
    binding.using?(self)
  end
  def titlecase
    gsub(/\b\w/){ $`[-1,1] == "'" ? $& : $&.upcase }
  end
end
```

This is just a normal cut as described in the RCR, but we have added the idea of an ((*applicable callback*)) --a condition that determines if the cut is used or skipped-over in the method chain. What binding.using? checks exactly is up to Matz. Most recently Matz has said that the scope should be per-file, but it can just as easily be at a lower level, say pre-module, and it all works --because, cuts themselves have a well defined behavior.

There would be no more confusion about how refinements are supposed to work. Cuts provide the well-defined answer.

---

[1] ((URL:http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/86071))

[2] ((URL:http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/86391))

[3] ((URL:http://blade.nagaokaut.ac.jp/cgi-bin/scat.rb/ruby/ruby-talk/86646))

[4] Not actually Yahuda Katz, who Matz has erroneously credited is his recent keynote speeches.

[5] Excluding the idea of the aspect given at the end of the RCR, that can be done via a 3rd party gem.
=end

History

#1 - 12/01/2012 09:13 AM - trans (Thomas Sawyer)

Here is a slightly more updated version of the RCR: https://github.com/rubyworks/cuts/blob/master/RCR.textile

#2 - 12/01/2012 09:55 AM - drbrain (Eric Hodel)

- Priority changed from 6 to Normal

- Target version changed from 2.0.0 to 2.6

Sorry, cuts are too late for ruby 2.0.0. We started feature freeze last month.

#3 - 12/01/2012 01:31 PM - matz (Yukihiro Matsumoto)

- Target version changed from 2.6 to Next Major

And it's too big to next minor.

Matz.

#4 - 12/01/2012 10:19 PM - trans (Thomas Sawyer)

This is too big, but refinements aren't?

#5 - 12/02/2012 12:29 AM - matz (Yukihiro Matsumoto)

Because we have made decision fore refinement before freeze.

Matz.

#6 - 12/02/2012 12:40 AM - trans (Thomas Sawyer)

I didn't understand such response at first. Now, I think I realize possible confusion. I am proposing that cuts be used as underlying implementation for refinements. The whole cut A < C construct does not need to be exposed --it is nothing that end developer's would even see at this point. The point is to give refinements a well grounded design framework.

As they currently stand refinements look like an arbitrarily implemented kludge. I don't think anyone in their right mind would go near them. And I fear they will severely hamper adoption of Ruby 2.0.

#7 - 12/02/2012 12:53 AM - The8472 (Aaron G)

On 01.12.2012 16:40, trans (Thomas Sawyer) wrote:

> Issue #7487 has been updated by trans (Thomas Sawyer).
>
> I didn't understand such response at first. Now, I think I realize possible confusion. I am proposing that cuts be used as underlying implementation for refinements. The whole cut A < C construct does not have to be exposed --It is nothing that end developer's would even see at this point. The point is to give refinements a well grounded design framework.

Ah, I missed this this thread and proposed something very similar (AST transforms) to achieve pretty much the same thing in the other one, see https://bugs.ruby-lang.org/issues/4085#note-211 (sorry for cross-posting)

That would be a more ruby-esque API on which refinements could be built.

> As they currently stand refinements look like an arbitrarily implemented kludge. I don't think anyone in there right mind would go near them. And I fear they would severely hamper adoption of Ruby 2.0.

I agree. We need a proper solution that integrates with the rest of the language.

#8 - 12/02/2012 01:23 AM - Anonymous

In message "Re: [ruby-core:50448] [ruby-trunk - Feature #7487] Cutting through the issues with Refinements"
on Sun, 2 Dec 2012 00:40:07 +0900, "trans (Thomas Sawyer)" transfire@gmail.com writes:

|I didn't understand such response at first. Now, I think I realize possible confusion. I am proposing that cuts be used as underlying implementation for refinements. The whole cut A < C construct does not have to be exposed --It is nothing that end developer's would even see at this point. The point is to give refinements a well grounded design framework.

I am not sure if I understand you correctly. If it's not exposed to end users, how should it be seen by refinement implementers?

```
                          matz.
```

#9 - 12/02/2012 02:39 AM - trans (Thomas Sawyer)

Same way as you have it now. The difference is behind the scenes.

```
module M
refine String do
def foo; "foo"; end
end
end
```

Under the hood this would create the equivalent of:

```
cut M::Refinements::String < String
def self.apply?(s)
s.include_refinement?(self)
end
def foo; "foo"; end
end
```

The name "M::Refinements::String" is arbitrary. Under the hood it doesn't even have to have a defined constant (though it might be helpful to have such a handle on it).

#10 - 12/02/2012 03:00 AM - matz (Yukihiro Matsumoto)

Probably I don't understand your true intention, but your proposal includes new syntax with new keyword, we cannot add it by 2.0. Besides that, there's no way to optimize 'cut' in compile-time, if I understand it correctly.

Matz.

#11 - 12/05/2012 06:56 PM - trans (Thomas Sawyer)

Well, that's is what I meant by "under the hood". There would be no new keyword exposed to Ruby and cuts could only be created via C code, at least until new major version. But, it doesn't matter anyway. headius (Charles Nutter) has made clear to me what you actually intend refinements to be (i.e. so called "method decorators"). This is not the same as cuts, which are a means of injecting/changing actual behavior of classes.

Cuts are still a powerful abstraction with plenty uses, but not as a means of implementing refinements. So I will have this issue closed and revisit cuts again at a later date.

#12 - 12/05/2012 08:57 PM - matz (Yukihiro Matsumoto)

- Status changed from Open to Rejected

OK, try again some time later.

Matz.