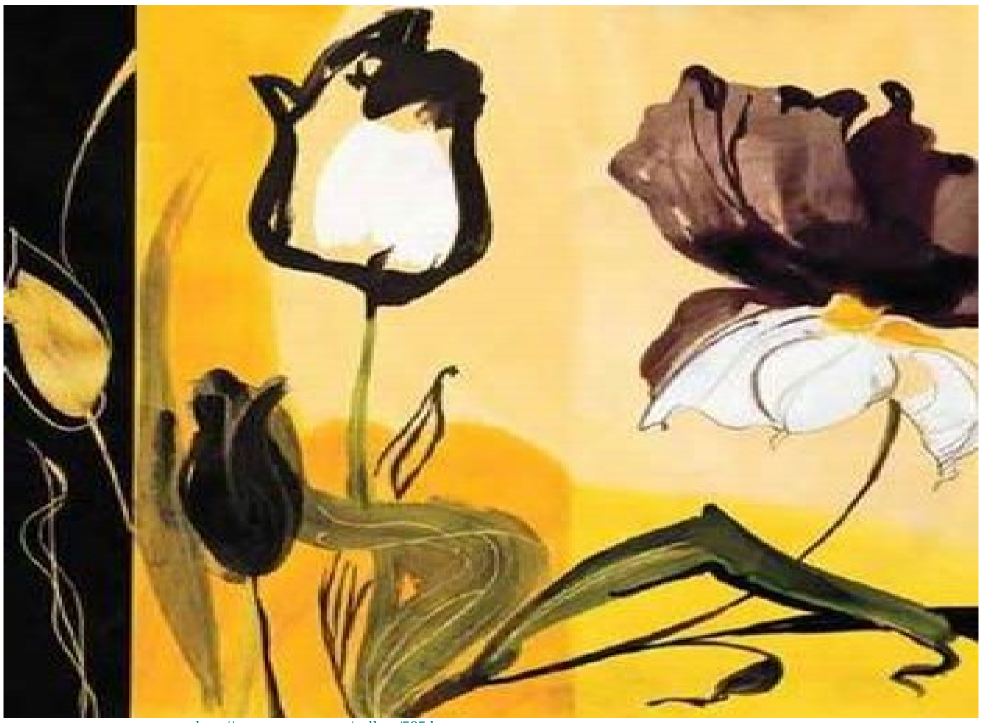
ObjectiveView

A Magazine for the Professional Software Developer


http://www.canvaz.com/gallery/585.htm

New Kids on the Block!

Features:

Ruby

Ruby on Rails

AspectJ

AJAX

Opinion:

Prefactor *and* be agile

Specs are bad

Abstraction

Goto considered harmful

Humour:

Glacial Development

published by

RATIO

Software Development, Training and Consultancy

www.ratio.co.uk/objectiveview.html for back copies

web distribution partner

ICONIX

www.iconixsw.com

Tel: +1 310 4580092

Fax: +1 310 3963454

Email: umltraining@iconixsw.com

web distribution partner

ThoughtWorks®

www.thoughtworks.com

web distribution partner

Software Reality

www.softwarereality.com

web distribution partner

CPTTM

www.cpttm.org.mo

web distribution partner

Ambysoft

www.ambysoft.com

web distribution partner

IASA

International Association of Software Architects

www.iasahome.org

web distribution partner

Dodeca Technologies

www.dodeca.co.uk

web distribution partner

OBTIVA

www.obtiva.com

ObjectiveView

CONTENTS

CONTACTS

Editor
Mark Collins-Cope
mcc@ratio.co.uk

Editorial Board
Scott Ambler
Kevlin Henney

Free subscription
PDF by email – email to:
objective.view@ratio.co.uk
(with subject: subscribe)

Feedback/Comments/Article Submission
info@ratio.co.uk

Circulation/Sponsorship Enquiries
info@ratio.co.uk

Authors may be contacted through the editor. All questions or messages will be passed on.

Have Your Say!
Join the ObjectiveView discussion and feedback group.
Simply visit:
groups.google.com/group/objectiveviewdiscussion

Subscribe to ObjectiveView
email: objective.view@ratio.co.uk subject: subscribe

Distribute ObjectiveView
It's easy. You link to ObjectiveView using our linkgif, and your logo will appear on *all* copies of the magazine. Contact info@ratio.co.uk for more info.

Welcome to Issue 9

If you're a regular reader of ObjectiveView, you'll know we don't rave about things very often. But for "Ruby on Rails" we've decided to break that rule. Having played with the Rails environment, we can honestly say that it lives up to its reputation as a truly high-productivity web/database development environment. The Ruby language itself takes some credit - it's compact, neat and in particular makes meta-programming (having code generated automatically on the fly) a truly viable option. The Rails framework itself takes full advantage of Ruby's meta-programming facilities to make the object/relational mapping a non-issue, generating web based UIs easy, taking the tedium out of issues such as validation, and enabling the manipulation of relationships between underlying tables (many to many, one to many, etc.) to be natural and easy. Meta-programming is the future - see the articles in this issue for more information.

It's also interesting to see how Ruby fairs against one of the other topics we cover in this issue - AspectJ. Aspect oriented programming has been gaining a following for a while now - albeit it hasn't hit the mainstream. Aspects deal with "cross-cutting" concerns like security checking calls to services in a service oriented architecture (to ensure the calling code has the correct permissions to make the call) - and Aspect/J certainly makes dealing with this type of concern (without duplicating code all over the place) much simpler.

But whereas with Java you need a new extended language to do this, Ruby's meta-programming features and the ability to extend a pre-existing class give you this for free (well almost – you'll have to do a bit of meta-programming, and it won't be as transparent). For my money, Ruby is the better option. But of course, not everyone will be able to use Ruby in their work. So AspectJ is still well worth a look. Aspects are a different way of looking at software – so even if you're not using AspectJ – you can still learn a lot from looking at the language.

The other major article in this issue covers AJAX. AJAX effectively opens the door to sophisticated and elegant direct manipulation user interfaces for the web. And about time too. One of the frustrations the web brought upon us (one we *had* to accept - mind you - given the benefits) was that user interface went back 20 years. Hopefully that is all about to change!

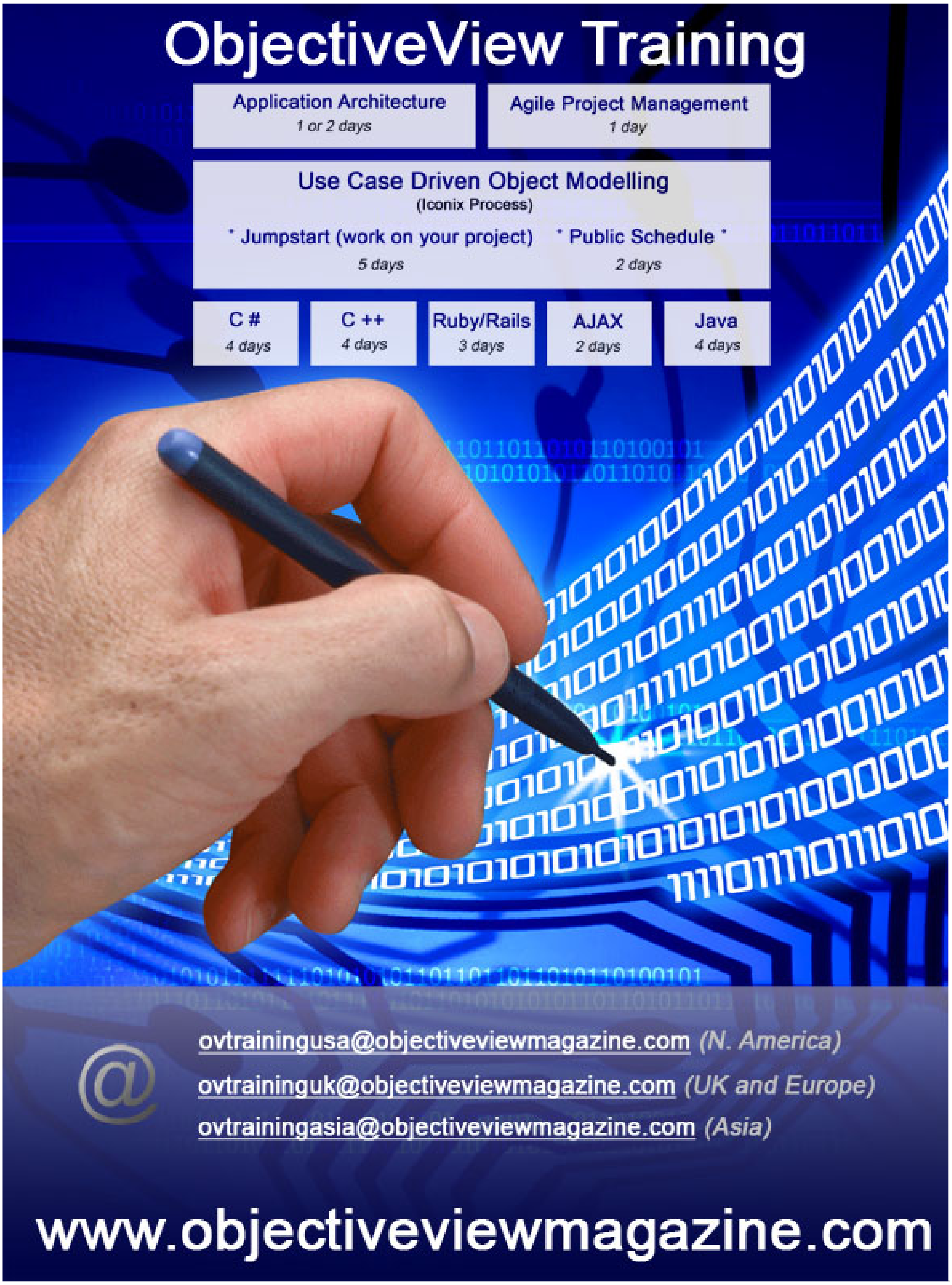
It's an interesting time for programming technology. It'll be even more interesting to see if AJAX, Ruby/Rails and AspectJ really do hit the big time.

We can but hope!

[illegible]

Mark Collins-Cope
London, February 2006.

advertisment


ObjectiveView Training
Application Architecture
1 or 2 days
Agile Project Management
1 day
Use Case Driven Object Modelling
(Iconix Process)
* Jumpstart (work on your project)
5 days
* Public Schedule *
2 days
C #
4 days
C ++
4 days
Ruby/Rails
3 days
AJAX
2 days
Java
4 days
ovtrainingusa@objectiveviewmagazine.com (N. America)
ovtraininguk@objectiveviewmagazine.com (UK and Europe)
ovtrainingasia@objectiveviewmagazine.com (Asia)
www.objectiveviewmagazine.com

http://www.objectiveviewmagazine.com/

Ruby’s a Gem

Amy Hoy with an amusing introduction the Ruby programming language…

Ah, Ruby. I'll skip the simpering introduction where I talk about how Ruby's the word on everyone's lips, worshippers and naysayers alike. You know this already, I know. But you may not have had time to give Ruby a thorough checking out, and that's a great reason to keep reading.

Help, What Am I Doing In This Nutshell?!

If I had to pick one word to describe Ruby—to sum it up in the shell of some kind of nut, as you might say—that word would have to be: elegance. Elegance, in this case, meaning as simple as it should be—and no simpler.

On the other hand, if "elegance," was unavailable as a choice for some reason, I'd probably pick "happiness." Ruby brings out the smiles, and it may just remind you of your salad days, when programming was fun and you enjoyed the tingly feel of photosynthesis in your leaves.

Contraindicators

You might be one of those few who have an unpleasant visceral reaction to Ruby. It's been known to happen in a small percentage of test subjects. Don't worry—even though you might get teased in the playground, you're reaction is completely within acceptable norms. Programming languages are, above all, an aesthetic choice.

Vitals—Stat!

Ruby's a great kid and it's got a lot going for it. For one, it's a very object-oriented programming language. Everything's an object, and there are no primitives.

Witness:

```
5.times { puts "Mice! " }
```

And:

```
"Elephants Like Peanuts".length
```

On the other hand, you don't have to encapsulate everything you write in classes—you're free to write procedural code if you so desire. Ruby classes may seem a bit strange at first, but we'll get to that later.

Feature Fantastic

Ruby also has a lot of very powerful features you've probably come to associate with "enterprise-class" development and typing lots and lots of extra characters, including operator overloading, a mature inheritance model (called mixins), and more.

Are You My Mommy?

Ruby descends from a dazzling array of languages, with names you might recognize: Smalltalk, Perl, and even Ada. It has such high-order programming features as blocks and closures, and so is an instant darling of folks who love the power of Lisp and yet find themselves violently allergic to anything with that much punctuation.

Ruby's also made to be readable, as in English:

```
Child.open('present') unless Child.bad?
```

You might recognize this as a Perlism. You would be right.

Zen is In

Ruby's daddy, Matz (Yukihiro Matsumoto) set out to "make programmers happy." Ruby made its first debut in 1995, and since then, Matz has repeatedly propounded the needs of humans over the needs of computers when it comes to language design. After all, the computers don't really care what the language looks like, and more processor power is easier to come by than joy.

Things in Ruby tend to work the way you'd think they would if you inhabited a sane, well-designed universe. Of course, you may not have inhabited such an environment till now, and may need to be *re-educated* to unlearn bad habits picked up elsewhere — cough — before things start just making sense.

This results in Ruby being a very simple-looking language. Perhaps even deceptively simple, as Ruby's arguably more powerful than a number of other popular languages (see Paul Graham's arguments in his book *Hackers and Painters* on what makes one language more powerful than another). While anyone can make a language look dense for an obfuscated <*insert-language-here*> contest, typically Ruby looks like no such thing.

Dive In—You Won't Break Your Neck

Learn by doing, I always say. Ruby's infamous crazy philosopher-king Why the Lucky Stiff clearly feels the same and, in his infinite insanity, has provided a spiffy web-based Ruby interpreter called TryRuby (http://tryruby.hobix.com/). He bills it as the cure for all ills, but I suggest it be used for trying out Ruby with no installation required.

Alternatively, if you are so inclined, open up your operating system's command-line interface (if it has one) and type “irb” to see if you already have Ruby installed. IRb stands for Interactive Ruby.

A Little Experimentation Never Hurt Anyone

Load up IRb or TryRuby and try these lines. Lines you type begin with >>, while responses from the Ruby interpreters begin with =>.

First, let's test your (and Ruby's) basic math skills:

```
>> 6 * 7
=> 42
```

Voila! You don't have to use any primitives to perform these elementary math operations. But just like in Java, those operators—and the others as well—are class methods which can be overloaded.

Now, something a little less like third grade math class:

```
>> 42.zero?
=> false
>> myvar = 0
=> 0
>> myvar.zero?
=> true
```

In addition to addition, the above code demonstrates that Ruby can answer direct questions! When you see a ? tacked on the end of a method, you can assume it will return a boolean value. This human-friendly convention makes code clearer with just a single character. Ruby's full of sweet little touches like this.

Now we leave numbers behind and enter the wild world of letters.

```
>> "Hi, I'm a String!".reverse
=> "!gnirtS a m'I ,iH"
>> "Monkeys! " * 3
=> "Monkeys! Monkeys! Monkeys! "
>> mystring = "cheese"
=> "cheese"
>> mystring[1..3]
=> "hee"
```

Ruby's *String* class is chock full of useful—or at least amusing—methods like the ones above. But you don't have to create a container variable to use class methods on an object. You need no intermediary to transform dairy product into pure glee:

```
>> "cheese"[1..3]
=> "hee"
```

Of course, not all methods available on one data type are available for another. When in doubt or in error, you can convert between them to get the result you want.

When you try to use a nonexistent method on a class, you'll get an error that looks like this:

```
>> 1337.reverse
NameError: undefined local variable or method `1337' for main:Object
            from (irb):12
            from (null):0
```

Whoops! That's a *String* method, but we tried to call it on a *Fixnum*. Bad mojo. Let's try that again:

```
>> 1337.to_s.reverse
=> "7331"
```

And, as you can see, you're able to stack up method calls like so many... precariously balanced things which are stackable.

```
>> (1..10).to_a
=> [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
```

This converts a *Range* object of *(1..10)* to an *Array* using the method *to_a*, *Range#to_a* comes from the mixin Enumerable, actually. Phew!

Basic Types

Your basic data types in Ruby are *Numeric* (subtypes include *Fixnum*, *Integer*, and *Float*), *String*, *Array*, *Hash*, *Object*, *Symbol*, *Range*, and *Regexp*. Ruby doesn't require you to use primitives when manipulating data of these types—if it looks like an integer, it's probably an integer; if it looks like a string, it's probably a string.

Which Type Are You?

And now, back to that idea of Ruby being brain-friendly. If you were assuming a sensible universe, how would you determine what type a given object was?

```
>> mysterytype = "hello"
>> mysterytype.class
=> String
>> (1..2).class
=> Range
```

You have to admit, it *does* make sense.

If you want to create an object which is of a specific type, and Ruby isn't inclined to do what you want, you can create it explicitly just like you create any other object from a class:

```
>> givemeastring = String.new("42")
=> "42"
```

Duck Typing

Another one of Ruby's vaunted features is its *duck typing* — it's not a statically typed language, far from it. Much like your social status in high school, an object's type (class) merely informs its very beginnings. Class is really not nearly as important as what an object can *do*, once it's out in the Real World.

If it looks like a duck, and quacks like a duck, we have at least to consider the possibility that we have a small aquatic bird of the family anatidae on our hands.

— Dirk Gently (as written by Douglas Adams) of Dirk Gently's Holistic Detective Agency

The essentials of duck typing can be boiled down to this: If you call a method, or reach for some capability, and the object in question has it, great—look no further, who cares what type it is! When put this way, it sounds less like a major programming theory thingamabobber and more like common sense, but there you have it.

Even so, duck typing can throw folks who have more static-typed language experience when, for example, an Array object responds to methods generally associated with Lists, Stacks, and Queues. If this happens to you, just relax, take a deep breath, and remember that that which does not kill you will only make you stronger. And I promise you, you won't be the first victim of terminal Ruby Exposure™.

Modules & Mixins

Ruby couldn't be outfitted with so many interesting features and then fall through on something as major as its inheritance paradigm. (Yeah, I said it — *paradigm!*) It'd be... well, a shameful lack of imagination!

And so, in Ruby, classes have only single inheritance—but you can get all the benefits (and only a few of the costs) of multiple inheritance through *mixins*. Mixins are a way of including a module of modular (get it?) code in one or more classes. Modules are very similar to classes, but are meant for extending classes, and creating namespaces, rather than standing on their own.

For example, the *Enumerable* module is mixed-in to both *Array* and *Hash*. It facilitates fun functions like *each* (for looping), *map* (for doing an array-walk kind of thing), *sort* (for sorting, natch), and so on. The best part is, the client code usage for all the methods in *Enumerable* is the same for both *Array* and *Hash*, so you just have to learn them the once and you're golden.

But mixins can be complex creatures, if they need be. The *Enumerable* module is a great example of this, because while the code for the module itself is fixed, it accomodates two rather different data types. How does it pull this off? Simple: *Enumerable* relies on the implementing class (*Array*, *Hash*) to provide its own *each* function to make everything work. It can delegate, while still maintaining the namespace. It's very purty.

Healthy eating?

Here’s some example code using a mixin:

```
module Digestion
  def  eat(*args)
    args.each { |food|
      #look for foods the including class cannot eat -- instance variable
      puts "#{food} is delicious!" unless self.inedible_stuff.include?(food)
    }
  end
end

class Dog
  include Digestion
  attr :inedible_stuff
  def initialize
    #dogs can't eat this stuff
    @inedible_stuff = ['rocks','rubber','chocolate','people']
  end
end

class MonsterUnderMyBed
  include Digestion
  attr :inedible_stuff
  def initialize
    #monsters can eat anything
    @inedible_stuff = []
  end
end

# examples:
# @inedible_stuff accessible from including class (dogs can't eat rocks)

irb(main):025:0> Dog.new.eat('rocks','dog food','stuff from the trash can')
dog food is delicious!
stuff from the trash can is delicious!
=> ["rocks", "dog food", "stuff from the trash can"]

irb(main):028:0> MonsterUnderMyBed.new.eat('your sock','your foot','your thigh')your sock is delicious!
your foot is delicious!
your thigh is delicious!
=> ["your sock", "your foot", "your thigh"]
```

Fancier Stuff

Ruby has blocks, which let you pass in segments of code as an argument to some functions. If you've ever dallied with Lisp, you'll recognize block as another word for *closure*. For example, a simple block is used in this code, which uses the *each* method described in the previous section:

```
>> ['monkey','cheese','pants'].each { |thing| puts "I put #{thing} on my head!" }
I put monkey on my head!
I put cheese on my head!
I put pants on my head!
=> ["monkey", "cheese", "pants"]
```

Here's how this works: When somebody wrote the *each* method, he used the *yield* statement, which tells Ruby to execute any code supplied in a block. In this case, the programmer used *yield* to put an object in local scope for the block—the object in this case being the array element in question.

You too can use *yield*. It can simply execute code in the block, or you can use it to pass around data as *each* does. And if *yield* is supplying variables, you access them in the block by giving them names in pipes, one for each object, separated by commas.

Here's a simple example of how to use *yield* in a method of your own:

```
>> def using_yield
..   number = 2;
..   yield('yeehaw!', number)
.. end
=> nil
```

Now that the method is written, give it a call. As you can see, *yield* is yielding two different variables, so make sure to catch them both in the block:

```
>> using_yield { |word,num| puts "#{word} -- #{num} times!" }
yeehaw! -- 2 times!
=> nil
```

Neat, huh? This is how easy Ruby makes higher-order programming. (The console still returns *nil* in addition to the output because *using_yield()* does not in fact *return* anything.)

An Object Lesson

Classes in Ruby can be an interesting mix of straight-up code and other code wrapped in methods.

Here's a sample class to get you started:

```
class Junk
  attr_reader :socks, :glass, :book

  def initialize
    @socks, @glass, @book = 'floor', 'table', 'under the chair'
    @mappings = {'one' => 'socks', 'two' => 'glass', 'three' => 'book'}
  end

  def clean!
    @socks = @glass = @book = 'put away'
  end

  ['one','two','three'].each do |name|
     define_method(name) do
        "The #{@mappings[name]} is " + self.send(@mappings[name])
     end
  end
end
```

If you're feeling so inclined, you can use the *do..end* keywords instead of curly braces when using blocks, as I did in this example.

The section at the end (*['one', 'two', 'three'] …]*) is an example of meta-programming in Ruby. This code is the equivalent of writing:

```
def one
  "The #{@mappings['one']} is "+ self.send(@mappings['one'])
end

def two
  "The #{@mappings['two']} is "+ self.send(@mappings['two'])
end

def three
 "The #{@mappings['three']} is"+self.send(@mappings['three'])
end
```

[editor's note: to see meta-programming being put to very effective use, see the use of macros like "validate_length_of" in the Ruby on Rails later]

Paste this code into your TryRuby console and then give it a spin:

```
>> stuff = Junk.new
=> #<Junk:0x224234 @socks="floor", @mappings={"three"=>"book",
"two"=>"glass", "one"=>"socks"}, @book="under the chair",
@glass="table">
```

That's certainly an ugly mishmash, but somewhat informative nonetheless, with the new object's class name, ID, and all the instance variables. Now, let's try out that dynamic method magic:

```
>> stuff.three
=> "The book is under the chair"
```

Crazy! And what's *this* do?

```
>> stuff.clean!
=> ["put away", "put away", "put away"]
```

I've never felt so excited about cleaning! That's another Rubyism for you: a method ending in a *!* will typically alter the object itself or otherwise do something that might be destructive or unexpected. Ruby programmers believe in putting punctuation to good use. (Remember, kids, this is a convention—not a rule.)

So, we modified the object. Where's the book now?

```
>> stuff.three
=> "The book is put away"
```

Readin' & Writin' in One Line (Each)

You may have noticed that lone line at the top of the class which begins with *attr_reader*. That's a Ruby method to automatically generate accessors for class and instance variables. If you want write access as well, do up a line with *attr_writer*, too. You use symbols (they're a whole 'nother article) to denote the variables you want accessible. Use the format *:variablename.*

Methods to the Madness

This simple class has two regular methods (*initialize* is what you call the constructor), and some procedural code which actually defines new methods on the fly. A simple loop through an array creates three methods which would otherwise involve lots of duplicated lines—methods called *one*, *two*, and *three*, which spit out info corresponding to their mapped instance variables.

The *define_method* method will create a new method in the current context (e.g., our class); it takes a single argument, the name for the new method, and then in the block you supply what the contents of the method would be. If the method takes arguments, you put them in the block in |*pipes*|, just like you do for blocks elsewhere.

While this *particular* code does isn't *particularly* interesting because of its extreme simplicity (also, it's useless), it becomes quite intriguing when you consider that you can have any number of behaviors trigger method generation.

You may have thought I'd make it through the whole article without mentioning Ruby on Rails, but I hope you didn't make any bets. Ruby on Rails is one project that uses dynamic method creation to great effect, by catching exceptions thrown for missing constants and then making new methods as necessary. This is what allows you to use such as-yet-non-existent methods as *Model.find_by_column1_and_column2*. Instead of spending the startup time creating these kinds of goodies, it waits until they're needed. Code creation, on demand!

Message in a Parenthesis

Near the end of our little *Junk* class, there's the little bit of code *self.send(@mappings[name])*. Conceptually, it's a surprisingly dense snippet, especially if you don't already know Ruby. But explaining those dense concepts—why, that's what I'm here for!

Ruby's got a message-based system for interacting with objects. If you want to access a data member or method of a class, you can use the *send* method which belongs to the base Object class (which everything subsequently inherits).

And the reason this works here is because when you append a variable name to the end of a Ruby object using the dot notation, you're actually calling a function—an accessor method. So you can use the same method, *send*, to access methods *and* variables—because in reality, they're the same.

One Last Trick

One of my favorite Ruby features is *open classes*. You can redefine a class anywhere, and instead of getting angry messages about how a class is already defined, you can *modify* it. (but note: you will get a warning you put the earlier code and the new code in a text file and run them with Ruby's warnings enabled).

```
class Junk
   def clean!
      @socks, @glass, @book = ['donated']*3
   end
end
```

Pop this little extension into your console and try *stuff.clean!* again. Get rid of that clutter... all the items will now report themselves as being *donated*. It's very freeing, you know.

Caveat Coder: Like “the force”, open classes can be used for good or evil. Or programming practices so suspect they take on the patina of evil, anyway. Use “the force” for good and Ruby will treat you right.

So Long, Farewell...

Our time together is coming to an end! I hope your interest is tickled, or better yet piqued, and you're excited to pursue Ruby further. I promise you, you won't regret it. Except maybe when you have to use other languages. If you are so inclined towards further Ruby scholarship, here are some great resources for you to contemplate:

Programming Ruby v2 (Pickaxe), http://www.pragprog.com/

Why's Poignant Guide to Ruby, http://poignantguide.net/ruby/

Ruby Idioms, http://renaud.waldura.com/doc/ruby/idioms.shtml

Unofficial Ruby Style Guide, http://www.caliban.org/ruby/rubyguide.shtml

Things that Newcomers to Ruby Should Know, http://www.glue.umd.edu/~billtj/ruby.html

And of course, please feel free to stop by my site **((24)Slash7, http://www.slash7.com)** and leave comments or drop me an email. Til then, happy Rubying!

Amy Hoy is a self-proclaimed geek girl who writes about Rails and interface design at http://www.slash7.com. She works full time as developer and designer for OmniTI, Inc in Columbia, MD.

advertisment

IASA

international association of software architects

The IASA was created by architects for architects to solve the major problems facing IT architects today. Our resources are scarce: Architects have very few resources to workwith. It is often very difficult to find the materials we need to do ourjobs.It is difficult to judge quality: Very few people agree on thefundamental components of architecture or what it takes to become aqualified architect.

Our experience isn't portable: Every employer defines and uses architecture differently. Lack of community: There is no way to quickly find peers of our own calibre to discuss the issues which drive us.

Joining the IASA is very simple. You can register at your local chapter or on the IASA website at www.IASAhome.org.

advertisment

Training in Ruby and Ruby on Rails

5 day hands on workshop (RAT 801)

You've heard the hype, now experience the reality!

Ruby on Rails really is the high-productivity web-application development environment you've heard about. In this five day course, you'll be taken through the basics of Ruby, before moving on to developing a real web application using Ruby on Rails. You'll be amazed how much you'll be able to achieve!

Pre-requisites:
A good knowledge of at least one object-oriented programming language, including the use of OO design techniques. Some experience of web development.

At the end of this course, you will be able to:

- Develop programs using the Ruby programming language
- Fully understand the structure of a Rails application
- Interface with relational databases using Rails' ActiveRecord module
- Develop web based UIs using Rails' ActionView templates and eRuby (embedded Ruby)
- Control applications using Rails' ActionController classes
- Undertake test-driven development using Rails
- Be able to undertake productive real-world development using Rails

Syllabus

Day 1:
- introduction
- names, methods and classes
- modules
- arrays and hashes
- control structures
- regular expressions
- hands on worked examples

Day 2:
- blocks and iterators
- exceptions
- marshalling objects
- rails directory structure
- rails configuration
- tables, classes, primary and foreign keys
- CRUD
- relationships (1-1, *-1, etc.) between tables and inhertance
- hands on worked examples

Day 3:
- one to one relationshiops
- one to many relationships
- page routing
- controllers
- automated functional testing
- hands on worked examples

Day 4:
- action methods
- sessions and cookies
- filters and verification
- redirecting
- web pages using action views
- rhtml templates (embedded ruby)
- hands on worked examples

Day 5:
- helpers and linking to pages
- pagination
- sending / receiving email with action mailer
- introduction to web services with Ruby
- introduction to Ajax with Ruby
- hands on worked examples

For more information Email: info@ratio.co.uk or Call: +44 (0) 208 579 7900

advertisment

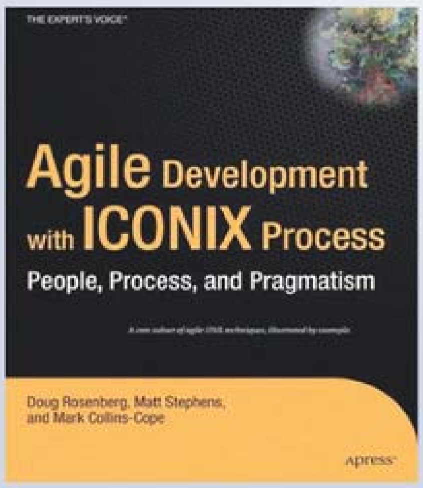
Agile UML®

Leverage the Power of UML 2.0 and the Agile/ICONIX Process to spec your software designs. Design a real mapping example from use cases through C#/.Net code in our open enrollment, public training workshops.

ENTERPRISE ARCHITECT

Optimized for Microsoft Visual Studio

+


THE EXPERT'S VOICE
Agile Development with ICONIX Process
People, Process, and Pragmatism
Doug Rosenberg, Matt Stephens, and Mark Collins-Cope
Apress

UML
Solution Partners

SPARX SYSTEMS
www.sparxsystems.com

ICONIX
Leaders in Object Technology
www.iconixsw.com

All product names are owned by their respective owners

Opinion ♦ Abstraction - Down on the Upside ♦ Kevlin Henney ♦ Opinion

When those involved in software development are sometimes accused of not living in the real world, there may actually be a case to answer. There are many things about software and its development that appear to be the wrong way around or the wrong way up…

… Take, for example, the case of the common tree. We have hierarchies for directories, for method calls and for class inheritance. We refer to these as trees. And where is the root? Highly visible and at the top. The leaves proliferate at the bottom, often obscured and hidden from immediate view.

Like the classic experiment where subjects are asked to where prism glasses that invert their view of the world, we adjust. What used to appear upside down now appears normal. We cease to notice the difference. In the case of trees, this is not really a problem: the properties of the abstraction are more useful to us than perfect fidelity against the metaphor. The inversion is simple enough not to be confusing and it is rarely misleading.

Which leads us neatly to the concept of abstraction and the way that it is often used. This flipped view of the world is not always so free of problems. Consider first of all what abstraction is: the act of omitting or taking away. This is an incredibly useful tool: it allows us to consider problems, solutions and models of problems and solutions without becoming lost in unnecessary detail. Different abstractions take different points of view: they omit certain kinds of detail in order to emphasise others. In this sense the practice of abstraction is intrinsically neither good nor bad; it is just more or less useful. A good abstraction is one that allows us to develop a piece of software more effectively; a poor abstraction is one that misleads us by omitting or including the wrong kind of detail.

Abstraction is a form of economy that at best supports clarity and focus and at worst can be confusing and misleading, so we can speak of quality of abstraction.

Often it is not so much a property of the abstraction as its inappropriate application that can lead us astray: using a crowded class diagram beset with operation and attribute minutiae where a sketched package diagram would have been more useful; using a programming language designed for science and engineering to do systems programming; using the London Underground map as a guide to London's geography.

So, what is so upside down about abstraction? In constructing a system we often stack different abstractions on one another as appropriate, leading to a layering of abstractions. The issue arises when we shift from speaking about "layers of abstractions" to "levels of abstraction". These two expressions are not synonymous and they have quite different implications. The latter implies that there is an intrinsic ordering to abstraction, which in one sense is true: one perspective leaves more out than another perspective is more abstract. It might be better to refer to this as "degrees of abstraction". One other aspect that might be ordered is granularity. Often when people speak of "levels of abstraction" they are actually referring to granularity of abstraction.

However, one of the most common tendencies is to equate "higher level" with "better" and "lower level" with "worse", which is where the inverted view of the world becomes noticeable. People often speak of domain-specific languages or analysis models as higher-level abstractions that are closer to the domain of the users than lower-level abstractions, such as bits and bytes, and so therefore better for working closely with users. The stated goal is that we should be thinking more in terms of the problem domain - high level - than the solution domain - low level. Wait a minute. If this is the goal, then it's an own goal: the terminology is the wrong way up and reveals an ingrained prejudice. What is it that is being left out? Machine-level details. Techies might never question this but, from the point of view of users, the world in which they carry out their business is significantly more concrete than the abstract realm of software.

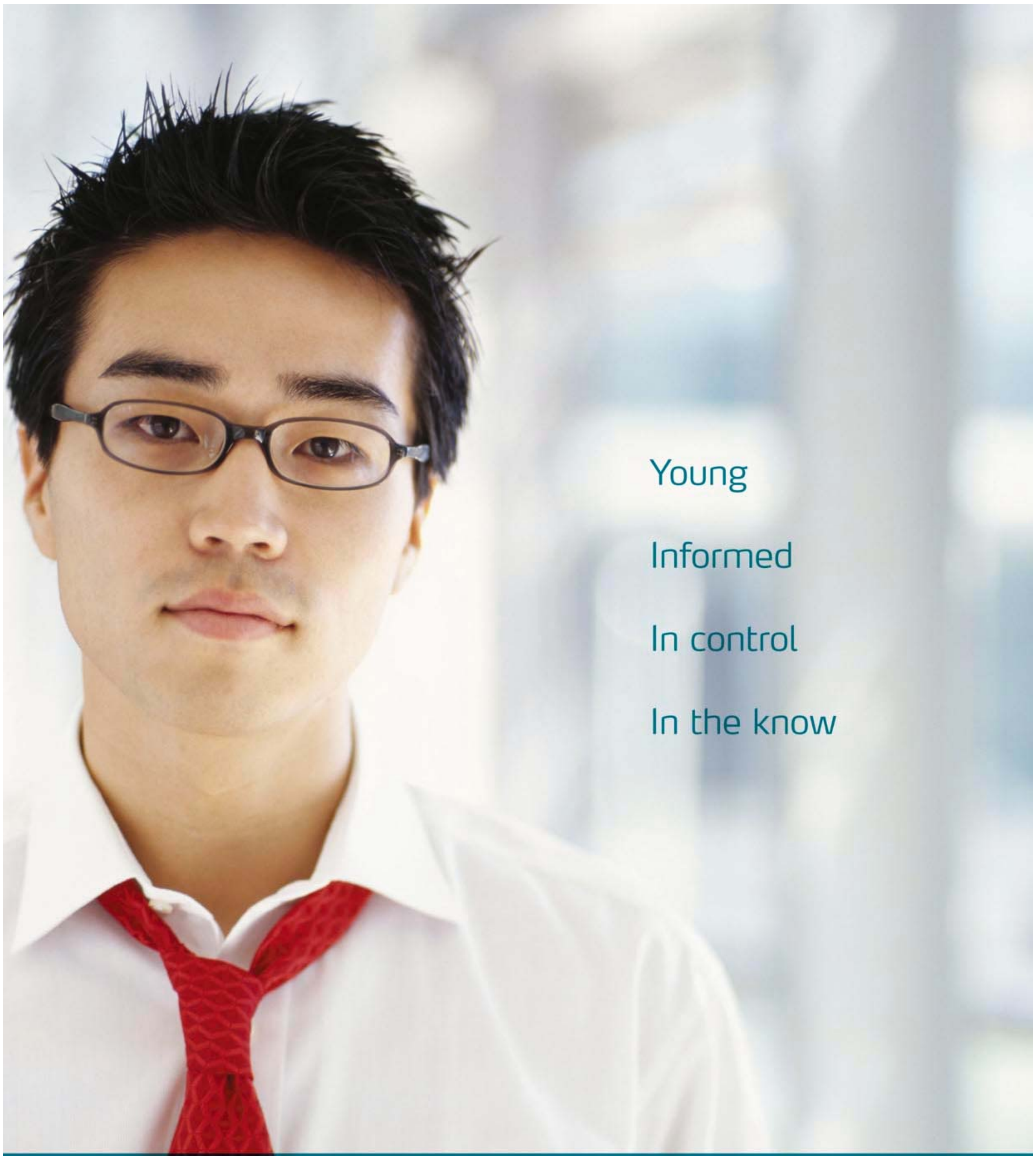
The deeper you go into the software the less real and more abstract things become. The real world (or problem domain, if you want to be precise) is what is gradually omitted. We need to be more careful in our terminology: abstraction is not so much a question of altitude as a matter of perspective and proximity.

Kevlin Henney is an independent consultant and trainer based in the UK. He specialises in programming languages and techniques, design and development process.

Have Your Say!
Join the ObjectiveView discussion and feedback group.
Simply visit:
groups.google.com/group/objectiveviewdiscussion

Subscribe to ObjectiveView
email: objective.view@ratio.co.uk subject: subscribe

advertisment



Meet David Tran, 26, Senior Software Engineer

As a senior software engineer with a multi-national company, David manages a wide number of projects from inception to completion, typically under strict timelines. And with the added responsibility of mentoring and developing a team of junior developers, David's time is precious to say the least.

That's why he's a regular visitor to the InternationalDeveloper.com website. Through the site David has access to a broad range of relevant industry news, reviews and resources that help him stay on top of the big issues in minimal time. So if your time is precious and you want to stay in the know, visit InternationalDeveloper.com today.

international ::developer

InternationalDeveloper.com

Ruby on Rails

David Heinemeier Hansson - the primary author of Ruby on Rails - loves to describe his wildly successful framework as an example of opinionated software, which basically means: software written as a faithful reflection of the author's opinions and desires. ***Obie Fernandez*** *explains…*

Here is a brief list of opinions that I see expressed clearly in Rails:

- Programming should be agile too
- Frameworks should be extracted from working applications
- Developing web applications should be fun and move rapidly
- Declarative aspects of code should follow logical conventions instead of requiring configuration
- Application developers should concentrate on delivering business value
- Reuse of application-level components is an overrated goal if your framework is powerful and productive enough

Agile and pragmatic programming principles dominate the philosophy and design of Rails, but perhaps most importantly, the authors of Rails have deep experience writing webapps. That experience shines through loud and clear in the design of the framework and the APIs.

I've been evangelizing Rails and working with Ruby professionally for close to a year. The best way to teach someone Rails and get them excited is to spend a little time pair programming in Rails with them, so they can really experience what it's all about. I'll try to simulate that experience as closely as possible while authoring a simple web application. I encourage you to setup Rails on your own computer and follow along.

Be advised that you'll need the following software installed:

- Ruby 1.8.2 (or higher)
- Rails 1.0
- SQLite3

I strongly recommend that Windows and Mac users new to Ruby look into the InstantRails and Locomotive single-download Rails installers, respectively. These simplify getting started with Rails tremendously.
Having problems getting your Rails installation to work correctly? Given all the online attention that Rails has gotten in the last six months, hit up Google for answers on resolving any problems you run into. I guarantee you that someone else has already figured out the solution and written about it.

An Example Application

For the example I picked a feature common to the vast majority of web applications out there: user authentication with an encryptedpassword. We'll need to write a User class which supports encrypted passwords and a login screen.

Getting Started

The requirements sound pretty easy, but since this is the first task of this project, we have some work ahead of us before we can start coding application logic. How much work exactly?

Traditionally, one of the hardest parts of kicking off a brand new project from scratch is deciding on directories to create and figuring out what parts of the application go where. Back when I was a junior programmer I'd typically keep a collection of codebases which I felt were structured well, just so that I could find one to clone for new projects whenever needed. Even that can be problematic though, depending on how closely the older project and new one match in terms of size and purpose.

Rails does a brilliant job of getting you past that first crucial obstacle of bootstrapping your application so you can get started with application development right away. Simply type rails <appname> at the command line. Throughout the rest of the article, I'll refer back to parts of the following list of directories and files and explain their purpose. In a burst of non-creativity, I've named my example app 'example'…

```
C:\workspace>rails example
    create  app/controllers
    create  app/helpers
    create  app/models
    create  app/views/layouts
    create  lib

   [snip!! … about 30 lines cut out here]

    create  public/javascripts
    create  public/javascripts/prototype.js
    create  public/javascripts/effects.js
    create  public/javascripts/dragdrop.js
    create  public/javascripts/controls.js
    create  public/stylesheets
    create  public/dispatch.fcgi
    create  public/index.html
    create  public/favicon.ico
    create  public/robots.txt
    create  doc/README_FOR_APP
```

That's a lot of files! I encourage you to take a moment and mount the example directory in your favorite editor or IDE. Browse through the files created – most of them are named in a somewhat self-explanatory way and contain easy to understand Ruby code.

Check out the script directory. A lot of helpful scripts were created for you in there, among them one that you will be using all the time: **script/server**. Try running it now. (Windows users type 'ruby script\server.)

advertisment

Revolutionising how software is delivered.

At ThoughtWorks, we've challenged conventional thinking – both in our approach to software development and in the way we run our business.

The result is a fast growing global IT consultancy that delivers on our promises to clients and our people.

A leader in the application of Agile delivery methods, we've built our reputation on delivering real world results for business in less time, with lower risk. And we've created an environment that liberates and motivates high calibre people, allowing them the freedom to think creatively and share knowledge openly, both internally and with their industry peers.

It's an unusual formula but it works. Although we now have over 700 people in six countries worldwide, we've stayed true to our principles and won't compromise on hiring only the very best.

To find out more about what makes ThoughtWorks a great place to work, visit **www.thoughtworks.co.uk**

Revolutionising your career.

ThoughtWorks®

```
C:\workspace\example>ruby script\server
=> Booting WEBrick...
=> Rails application started on http://0.0.0.0:3000

=> Ctrl-C to shutdown server; call with --help for options
[2006-01-26 20:37:40] INFO  WEBrick 1.3.1
[2006-01-26 20:37:40] INFO  ruby 1.8.2 (2004-12-25) [i386-mswin32]
[2006-01-26 20:37:40] INFO  WEBrick::HTTPServer#start: pid=1432 port=3000
```

Congrats! You've just fired up your first Rails application. Webrick is a simple HTTP server that is bundled with Ruby and it's the web server we use to test our application locally. Unless your installation is messed up somehow, you should be able to open up your browser and point it to http://localhost:3000 to see an introductory screen.

Database Configuration

The next step after bootstrapping your new Rails application is always to setup your database configuration. The **database.yml** file in the config directory has defaults in it that you modify to meet your needs. For our example application, blow away the provided defaults, and add the parameters necessary to run with SQLite3 file-based database instances.

```
# SQLite version 3.x
# gem install sqlite3-ruby

development:
  adapter: sqlite3
  database: db/development.sqlite3

test:
  adapter: sqlite3
  database: db/test.sqlite3

production:
  adapter: sqlite3
  database: db/production.sqlite3
```

In case you didn't guess it yourself, the format of this file is YAML (which stands for Yet Another Markup Language) and as you can see by how concise it is, YML won't give you nearly as much carpal-tunnel syndrome as it's much more popular older cousin XML. Why does Rails use YML instead of XML? The main reason is that YML is less verbose and easier to read than XML. It's also very well-supported in Ruby and you can serialize Ruby objects to YAML very easily.

Rails Modes (Development, Test and Production)

This is as good a time as any to mention that Rails has three modes of operation. The two most Rails developers will become most familiar with are development and test modes. During development, your Rails web server will use the development database instance and, as you might guess, the test database instance is used when running automated tests. In development mode, Rails reloads modified model and controller code with each request, which means you hardly ever need to restart your server to see changes reflected – just hit 'reload' on your browser and enjoy the fruits of your labor. There are a few exceptions though. One is the database.yml file that we just wrote and another is the **routes.rb** file that tells Rails how URLs are map to controller code.

HTTP Request Handling

Rails follows the Model View Controller (MVC) pattern for web applications popularized by web frameworks such as Struts. Describing the principles behind MVC is outside the scope of this article.
Rather than forcing the developer to maintain verbose and complicated configuration files defining explicit URL → Controller mappings, Rails routes incoming HTTP requests by convention, according to the rules established in **routes.rb**. Requests are routed according to rules and handed off to public *action* methods on controller classes.

```
config/routes.rb
```

For our example application, I've modified the default **routes.rb** slightly to map the root URL to a login controller that we will write a little later on in the article when we discuss the ActionController API in depth.

```
ActionController::Routing::Routes.draw do |map|
  # Add your own custom routes here.
  # The priority is based upon order of creation:
  # first created -> highest priority.
  # Here's a sample route:
  # map.connect 'products/:id',
  #            :controller => 'catalog', :action => 'view'
  # Note you can assign values other than :controller
  # and :action
  # Map root URL to the index action of the login
  # controller
  map.connect '', :controller => 'login',
                  :action => 'index'
  # Install the default route as the lowest priority.
  map.connect ':controller/:action/:id'
end
```

It is typical to define a URL root mapping for your application, but most simple Rails applications will rely on the default, lowest priority route. Now that we've edited the database and routes files we are done with configuration and we can start coding our application. If your Rails server is still running, hit Ctrl-C to shut it down and restart it again. Now if you access http://localhost:3000 again you will get an error screen along the lines of 'Controller not found'.

Application Code

Let's take a look at the first directories that were created during bootstrapping. They are where most of our application codebase will reside, and as you can tell from their names, they reflect the MVC architecture of Rails that we were just discussing.

```
app/controllers
app/helpers
app/models
app/views/layouts
lib
```

While we're at it, I'll mention the other application code directories. The *helpers* directory contains Ruby modules (a.k.a mixins) that Rails auto-magically mixes into your view templates and are convenient locations for commonly used template logic. Finally, the lib directory is home for code that doesn't belong to any particular model or controller.

So, let's write the model for our example application. Many web applications have the concept of a *User* that can login to access protected resources. Rails applications are typically designed in somewhat of a bottom-up fashion and we'll start our application development by creating a database schema with a user table.

The ActiveRecord Migrations API simplifies DDL operations between versions of your application and has a generator for creating new blank migrations classes. To invoke it, type the following at your command prompt.

```
C:\workspace\example>ruby script\generate migration create_users_table
      create  db/migrate
      create  db/migrate/001_create_users_table.rb
```

Pop open the newly created file and you should see two empty class method declarations *up* and *down*.

```
class CreateUsersTable < ActiveRecord::Migration
  def self.up
  end

  def self.down
  end
end
```

The documentation for the Migrations API is located at:

http://api.rubyonrails.com/classes/ActiveRecord/Migration.html

Using the API I'll describe my users table. Notice how Rails makes use of Ruby blocks (closures). Beneath the covers the create_table method issues a CREATE TABLE command to the database, but it also yields a table variable to the block provided. This style of programming is a common Ruby idiom.

```
class CreateUsersTable < ActiveRecord::Migration
  def self.up
    create_table "users" do |table|
      table.column "login", :string, :limit => 40
      table.column "created_at", :datetime
      table.column "updated_at", :datetime
    end
  end

  def self.down
   # It wouldn't make sense to put anything here
  end
end
```

When you're ready to apply the changes described in your migration file, you simply run the following at your command prompt:

```
C:\workspace\example>rake migrate
(in C:/workspace/example)
```

Introducing Rake

Rake is Ruby's build tool, comparable to Unix's *make* by merit of being dependency-based. Martin Fowler recently wrote a great introduction to Rake.. (http://www.martinfowler.com/articles/rake.html) Don't worry about similarities to make, Rakefiles are nowhere near as difficult to understand and maintain as makefiles. I'm sure Java programmers will find Rake somewhat similar in use to Apache Ant. When we bootstrapped our example application, one of the files generated for us was a Rakefile. Let's take a peek at the other tasks included…

```
C:\workspace\example>rake –T
(in C:/workspace/example)

rake add_new_scripts                # Add new scripts to the application script/ directory
rake apidoc                         # Build the apidoc HTML Files
rake appdoc                         # Build the appdoc HTML Files
rake clear_logs                     # Clears all *.log files in log/
rake clobber_apidoc                 # Remove rdoc products
rake clobber_appdoc                 # Remove rdoc products
rake clobber_plugindoc              # Remove plugin documentation
rake clone_schema_to_test           # Recreate test database from the current schema
rake clone_structure_to_test        # Recreate test database from the development structure
rake create_sessions_table          # Creates a sessions
rake db_schema_dump                 # Create a db/schema.rb file
rake db_schema_import               # Import a schema.rb file into the database.
rake db_structure_dump              # Dump the database structure to a SQL file
rake default                        # Run all the tests on a fresh test database
=rake drop_sessions_table           # Drop the sessions table
rake freeze_edge                    # Lock this application to the Edge
rake freeze_gems                    # Lock this application to the current gems
rake load_fixtures                  # Load fixtures into the current environment's database
rake migrate                        # Migrate the database according to the migrate scripts
                                    # (only supported on PG/MySQL).  A specific version can be targetted with VERSION=x
rake plugindoc                      # Generate documation for all installed plugins
rake prepare_test_database          # Prepare the test database and load the schema
rake purge_sessions_table           # Drop and recreate the session table
rake purge_test_database            # Empty the test database
rake reapidoc                       # Force a rebuild of the RDOC files
rake recent                         # Run recently modified tests
rake stats                          # Report code statistics
rake test_functional                # Run tests for test_functional
rake test_plugins                   # Run tests for test_pluginsenvironment
rake test_units                     # Run tests for test_unitsprepare_test_database
rake unfreeze_rails                 # Unlock this application from freeze of gems or edge
rake update_javascripts             # Update javascripts in your current Rails installation
```

If you're following along on your own computer you might notice that I shortened some of the task descriptions for size reasons.

What do Rake tasks look like? Well, to start with, Rakefiles are written in Ruby. One of the great attributes of Ruby is that it is very handy for creating internal domain-specific languages. So handy, in fact, that almost all programming in Ruby is done with Ruby code.

If it didn't quite make sense to you why I made that last assertion, ask yourself: Is all Java programming done with Java code? How about .NET? You'll find that most *serious enterprise* programming is actually done with lots and lots of XML or via complicated GUIs nowadays. Painful!

If we open up the Rakefile we can look for the migrate task that we ran a minute ago:

```
# Add your own tasks in files placed in lib/tasks ending in .rake,
# for example lib/tasks/switchtower.rake, and they
# will automatically be available to Rake.
require(
  File.join(File.dirname(__FILE__), 'config', 'boot')
)
require 'rake'
require 'rake/testtask'
require 'rake/rdoctask'

require 'tasks/rails'
```

Whoops! I forgot that as of a few releases ago the Rails authors wisely decided to un-clutter project Rakefiles by importing the default tasks from the Rails installation. We'll have to do some digging to find the migrate task. I won't bother with the first required file, **boot.rb**, since I know that it is responsible for loading both the Rails environment and your project files.

The next three require statements are for standard Rake library files included with the Ruby install. We'll open the **rails.rb** file and see what it contains.

```
# Load Rails rakefile extensions
Dir["#{File.dirname(__FILE__)}/*.rake"].each
  { |ext| load ext }

# Load any custom rakefile extensions
Dir["./lib/tasks/**/*.rake"].sort.each
  { |ext| load ext }

Dir["./vendor/plugins/*/tasks/**/*.rake"].sort.each
  { |ext| load ext }
```

No migrate task. But see how concise Ruby code can be? Remember, Ruby first gained a cult following as a powerful scripting language. I finally found the rake migrate task in a file that is loaded by the first line of code: **databases.rake**

```
desc "Migrate the database according to the migrate scripts in
db/migrate"

task :migrate => :environment do
  ActiveRecord::Migrator.migrate("db/migrate/",
    ENV["VERSION"] ? ENV["VERSION"].to_i : nil)

  Rake::Task[:db_schema_dump].invoke if
    ActiveRecord::Base.schema_format == :ruby
end
```

That's kind of complicated code, but if you take a moment to decipher it you'll notice that it takes an optional VERSION environment variable. If we access our Sqlite3 database file directly, we can see what the migrate task actually did.

```
C:\workspace\example>sqlite3 db\development.sqlite3
SQLite version 3.2.7
Enter ".help" for instructions
sqlite> .dump
BEGIN TRANSACTION;
CREATE TABLE schema_info (version integer);
INSERT INTO "schema_info" VALUES(1);
CREATE TABLE users ("id" INTEGER PRIMARY KEY NOT NULL,
"login" varchar(40),
"created_at" datetime, "updated_at" datetime);
COMMIT;

sqlite>
```

See the schema_info table? That's where Rails keeps the current migration version. I'm going to need another couple of columns in my database so that users can have encrypted passwords. Let's add them with another migration script.

```
C:\workspace\example>ruby script\generate migration
add_password_columns_to_users
      exists  db/migrate
      create  db/migrate/002_add_password_columns_to_users.rb
```

This time I'll use a method named add_column directly:

```
class AddPasswordColumnsToUsers < ActiveRecord::Migration
  def self.up
    add_column :users, :crypted_password, :string
    add_column :users, :salt, :string
  end

  def self.down
    remove_column :users, :crypted_password
    remove_column :users, :salt
  end
end
```

Then I'll run migrate one more time and we should be ready to write our first ActiveRecord model class in Ruby.

```
C:\workspace\example>rake migrate
(in C:/workspace/example)
```

ActiveRecord Models

Martin Fowler described the ActiveRecord pattern in *Patterns of Enterprise Application Architecture* as follows: An object that wraps a row in a database table or view, encapsulates the deatabase access, and adds domain logic on that data. The pattern was chosen for Rails in the interest of decreasing the amount of effort it takes to build a database-backed domain model. [http://www.martinfowler.com/eaaCatalog/activeRecord.html] Speaking from experience, ActiveRecord models are much easier to develop compared to using DataMapper-based solutions such as Hibernate. Since Ruby is single-inheritance like Java, it bothered me that to use ActiveRecord your persistent domain models must extend the ActiveRecord::Base class, but in practice I haven't found it to be a problem.

More Generators

We've already used the generator script to create migrations files. Rails provides code generation where it makes sense, especially to produce boilerplate code. Rails includes basic generators…

```
C:\workspace\example>ruby script\generate
Usage: script/generate [options] generator [args]

General Options:
   -p, --pretend              Run but do not make any changes.
   -f, --force                Overwrite files that already exist.
   -s, --skip                 Skip files that already exist.
   -q, --quiet                Suppress normal output.
   -t, --backtrace            Debugging: show backtrace on errors.
   -h, --help                 Show this help message.
   -c, --svn                  Modify files with subversion. (Note: svn must be
in path)


Installed Generators
  Builtin: controller, mailer, migration, model, plugin, scaffold,
session_migration, web_service
More are available at http://rubyonrails.org/show/Generators
```

We create our User model class using the model generator.

```
C:\workspace\example>ruby script\generate model User
      exists  app/models/
      exists  test/unit/
      exists  test/fixtures/
      create  app/models/user.rb
      create  test/unit/user_test.rb
      create  test/fixtures/users.yml
```

You're following along on your own computer right? Take a moment to browse through the generated files and get a feel for what Rails considers *boilerplate code*. (Hint: There's not much of it.) We'll come back to all these files in a moment.

The User Model

Since our database is ready, so we can go back to coding our User model. Let's take a look at the generated User class.

```
class User < ActiveRecord::Base
end
```

Seems empty, but this class is already usable to some extent. Rails provides a script wrapper around Ruby's interactive shell, irb, that automatically loads the Rails environment for your project. In the console session captured below, first I'll instantiate a user object, then I'll use the ActiveRecord method create to instantiate a new user object and persist it to the database in one line. Finally, I'll query the database using the find method to verify that the new User object was persisted.

```
C:\workspace\example>ruby script\console
Loading development environment.

>> User.new
=> #<User:0x381b770 @attributes={"salt"=>nil, "updated_at"=>nil,
"crypted_password"=>nil, "login"=>nil, "created_at"=>nil}, @new_record=true>

>> User.create(:login=>'obie')
=> #<User:0x37d31a0 @errors=#<ActiveRecord::Errors:0x37d2648 @errors={}, @base=#<User:0x37d31a0 ...>>, @attributes={"salt"=>nil,
"updated_at"=>Sun Jan 22 21:52:38 Central Standard Time 2006, "crypted_password"=>nil, "id"=>1, "login"=>"obie", "created_at"=>Sun Jan 22 21:52:38
Central Standard Time 2006}, @new_record=false>

>> User.find(:all)
=> [#<User:0x37caea8 @attributes={"salt"=>nil, "updated_at"=>"2006-01-22 21:52:38", "crypted_password"=>nil, "id"=>"1", "login"=>"obie",
"created_at"=>"2006-01-22 21:52:38"}>]
```

How does the User object know about the database if we haven't programmed any attributes into it yet? The secret is in how ActiveRecord interacts with the database, and the easiest way to describe it is to take a peek at the development.log file.

The first 9 lines show what happened during our migration a little while ago. I highlighted the lines that were logged during our console session.

```
SQL (0.000000) SQLite3::SQLException: no such table: schema_info: SELECT * FROM schema_info
SQL (0.000000) SELECT name FROM sqlite_master WHERE type = 'table'
SQL (0.079000) CREATE TABLE schema_info (version integer)
SQL (0.109000) INSERT INTO schema_info (version) VALUES(0)
SQL (0.000000) SQLite3::SQLException: table schema_info already exists: CREATE TABLE schema_info (version integer)
SQL (0.000000) SELECT version FROM schema_info
Migrating to CreateUserTable (1)
SQL (0.000000) SQLite3::SQLException: no such table: users: DROP TABLE users
SQL (0.110000) CREATE TABLE users ("id" INTEGER PRIMARY KEY NOT NULL, "login" varchar(40), "crypted_password" varchar(40), "salt"
varchar(40), "created_at" datetime, "updated_at" datetime)
SQL (0.156000) UPDATE schema_info SET version = 1

SQL (0.000000) PRAGMA table_info(users)
SQL (0.000000) INSERT INTO users ("salt", "updated_at", "crypted_password", "login", "created_at") VALUES(NULL, '2006-01-22 21:52:38', NULL,
'obie', '2006-01-22 21:52:38')
User Load (0.000000)
SELECT * FROM users
```

The PRAGMA table_info(users) statement sheds some light on what ActiveRecord is doing behind the scenes. Let's see what that statement returns using the sqlite3 console.

```
C:\workspace\example>sqlite3 db\development.sqlite3
SQLite version 3.2.7
Enter ".help" for instructions
sqlite> PRAGMA table_info(users);
0|id|INTEGER|99||1
1|login|varchar(40)|0||0
2|crypted_password|varchar(40)|0||0
3|salt|varchar(40)|0||0
4|created_at|datetime|0||0
5|updated_at|datetime|0||0
sqlite>
```

Rails is actually reading the schema of your database table and using that to dynamically add properties to your model. No more getter and setter methods! Many would argue that such tight binding to the database is a *bad thing*. I'll remind them that we're following the Active*Record* pattern – it might be a bad thing if you were trying to hide the fact that your persistent classes are backed by a database model, but we're not. Pragmatism and not repeating yourself (the DRY principle) are important opinions of Rails.

Testing

The importance of TDD (Test-Driven Design) is another important opinion of Ruby on Rails. Since we have no compiler safety-net in Ruby, it is especially crucial that we maintain full test-coverage of application code. Otherwise, the only way we'll find bugs is via manual testing (or reports from frustrated end-users.)

Rails encourages automated testing by including comprehensive testing capabilities built right into the framework. Functional tests verify correct controller code and unit tests verify domain logic included in ActiveRecord models.

```
test/fixtures
test/functional
test/mocks/development
test/mocks/test
test/unit
```

Rails expresses its TDD opinion by not giving you any excuses *not* to test. Look closely at the list of files created when we generated the User model. There is a **user_test.rb** in there…

```
require File.dirname(__FILE__) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  fixtures :users

  # Replace this with your real tests
  def test_truth
    assert_kind_of User, users(:first)
  end
end
```

The easiest unit test you ever wrote, huh? Let's run it and see what happens.

```
C:\workspace\example>rake
(in C:/workspace/example)

c:/ruby/bin/ruby -Ilib;test "c:/ruby/lib/ruby/gems/1.8/gems/rake-
0.6.2/lib/rake/rake_test_loader.rb" "test/unit/user_test.rb"

Loaded suite c:/ruby/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake/rake_test_loader
Started
.
Finished in 0.219 seconds.
1 tests, 1 assertions, 0 failures, 0 errors
```

Green bar! Of course, it doesn't really prove anything does it? What did that test actually try to do anyway? The method call *users(:first)* relies on the Fixtures system that Rails bakes directly into the Test::Unit library included with Ruby. There's much to be said about fixtures, but not by me in this article. I'll simply explain that it is the system by which you can easily create test data to be used in your tests. Again I'll point you at the Rails documentation for more information: http://api.rubyonrails.com/classes/Fixtures.html.

Back to the unit test to TDD the functionality we want to include in the User model. First, we shouldn't be able to create a user without a login attribute. I comment out the call to fixtures since we don't need it yet.

```
require File.dirname(__FILE__) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  #fixtures :users
```

```
  def test_should_require_login
    user = User.create(:login => nil)
    assert user.errors.on(:login)
  end
end
```

The create method is defined in ActiveRecord::Base. It takes a hash of attributes to initialize the object with and invokes save, which does not throw an exception if the save operation fails. (You can use save! instead if you'd like an exception thrown) As alluded to by the test case above, User has an errors object that contains any errors encountered during persistence operations. If user.errors.on(:login) returns nil the assert will fail.

Run the default rake task to execute your suite of automated tests:

```
C:\workspace\example>rake
(in C:/workspace/example)
c:/ruby/bin/ruby -Ilib;test "c:/ruby/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake/rake_test_loader.rb"
 "test/unit/user_test.rb"
Loaded suite c:/ruby/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake/rake_test_loader
Started
F
Finished in 0.468 seconds.

  1) Failure:
test_should_require_login(UserTest) [./test/unit/user_test.rb:8]:
<nil> is not true.

1 tests, 1 assertions, 1 failures, 0 errors
c:/ruby/bin/ruby -Ilib;test "c:/ruby/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake/rake_test_loader.rb"

rake aborted!
Test failures
```

The test fails because right now, because User is perfectly happy with a null login. We'll change that using one of the many validation methods available.

```
class User < ActiveRecord::Base
  validates_presence_of :login
end
```

Then run the test…

```
C:\workspace\example>rake recent
(in C:/workspace/example)
c:/ruby/bin/ruby -Ilib;test "c:/ruby/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake/rake_test_loader.rb"
 "test/unit/user_test.rb"
Loaded suite c:/ruby/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake/rake_test_loader
Started
.
Finished in 0.281 seconds.

1 tests, 1 assertions, 0 failures, 0 errors
```

We can validate other things as well. For instance, user logins should be unique. We can leverage fixtures for this test. What do we have in the **users.yml** file right now?

```
# Read about fixtures at
# http://ar.rubyonrails.org/classes/Fixtures.html
```

```
first:
  id: 1
another:
  id: 2
```

Boring! I’ll delete the defaults and put a more interesting record in there, taking advantage of the templating capacities available in fixtures.

```
quentin:
  id: 1
  login: quentin
  created_at: <%= 5.days.ago.to_s :db %>
  updated_at: <%= 5.days.ago.to_s :db %>
```

Now I can add the new test method.

```
class UserTest < Test::Unit::TestCase
  fixtures :users
```

```
def test_should_require_login
  user = User.create(:login => nil)
  assert user.errors.on(:login),
   "no errors creating user with nil login"
end

def test_should_require_unique_login
  user = User.create(:login => 'quentin')
  assert user.errors.on(:login),
   "no errors creating another quentin user"
end
```

Of course it fails. Notice I used the assert method’s optional message parameter this time.

```
C:\workspace\example>rake
(in C:/workspace/example)
c:/ruby/bin/ruby -Ilib;test "c:/ruby/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake/rake_test_loader.rb"
 "test/unit/user_test.rb"
Loaded suite c:/ruby/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake/rake_test_loader
Started
.F
Finished in 0.328 seconds.

  1) Failure:
test_should_require_unique_login(UserTest) [./test/unit/user_test.rb:13]:
no errors trying to create another quentin user.
<nil> is not true.

2 tests, 2 assertions, 1 failures, 0 errors
c:/ruby/bin/ruby -Ilib;test "c:/ruby/lib/ruby/gems/1.8/gems/rake-0.6.2/lib/rake/rake_test_loader.rb"

rake aborted!
Test failures
```

We’ll need another validation to make this test pass. Since you’re probably getting my drift by now I’ll go ahead and add the rest of the login validations.

```
class User < ActiveRecord::Base
  validates_presence_of     :login
  validates_uniqueness_of   :login
  validates_length_of       :login, :within => 3..40
end
```

[editor’s note: validates_presence_of is an example of a meta-programming macro – it generates code on the fly. See the Ruby article earlier for a description of meta-programming]

Notice how well the code reads? Many people have described the Rails APIs as a collection of domain-specific languages for creating web applications. Rails is proof that Ruby is a fantastic language for writing internal domain-specific languages. Martin Fowler covered the concept of ‘Internal DSL’ in his article about http://martinfowler.com/articles/languageWorkbench.html.

Okay, we should add password functionality to this User class now, but first we’ll add a couple more tests for **user_test.rb**

```
  def test_should_require_password
    user = User.create(:login=> 'obie', :password => nil)
    assert user.errors.on(:password)
```

```
end

def test_should_require_password_confirmation
  user = User.create((:login=> 'obie',
                :password => 'qwerty',
                :password_confirmation => nil)
  assert user.errors.on(:password_confirmation)
end
```

We didn’t add a ‘password’ column to User when we created the database table. Instead we give User a password attribute directly in its Ruby code using the *attr_accessor* method.

```
class User < ActiveRecord::Base
  attr_accessor :password

  validates_presence_of  :login
  validates_length_of  :login,    :within => 3..40
  validates_uniqueness_of  :login

  validates_presence_of
    :password,
    :password_confirmation,
    :if => :password_required?

  validates_length_of
    :password,
    :within => 3..40,
    :if => :password_required?

  validates_confirmation_of :password,
    :if => :password_required?
```

```
 protected

   def password_required?
     crypted_password.nil? or not password.blank?
   end

end
```

After refactoring the test code and extracting the common user creation code, it looks like this…

```
require File.dirname(__FILE__) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  fixtures :users

  def test_should_require_login
    user = create_user(:login => nil)
    assert user.errors.on(:login)
  end

  def test_should_ignore_blank_password
    users(:quentin).password = ''
    assert users(:quentin).valid?
    users(:quentin).password = 'foo'
    assert !users(:quentin).valid?
  end

  def test_should_require_unique_login
    user = create_user(:login => 'quentin')
    assert user.errors.on(:login)
  end

  def test_should_require_password
    user = create_user(:password => nil)
    assert user.errors.on(:password)
  end

  def test_should_require_password_confirmation
    user = create_user(:password_confirmation => nil)
    assert user.errors.on(:password_confirmation)
  end

  protected

    def create_user(override = {})
      User.create({:login => 'obie',
                   :password => 'qwerty',
                   :password_confirmation => 'qwerty'}.merge(override))
    end

end
```

The only thing left is to add the encryption and authentication methods. For the sake of brevity here is the complete User class including that functionality. Notice addition of a *before_save* callback declaration and an authenticate class method.

```
class User < ActiveRecord::Base
  attr_accessor :password

  validates_presence_of :login
  validates_length_of :login,    :within => 3..40
  validates_uniqueness_of :login

  validates_presence_of
    :password,
    :password_confirmation,
    :if => :password_required?

  validates_length_of
    :password,
    :within => 3..40,
    :if => :password_required?

  validates_confirmation_of
    :password,
    :if => :password_required?

  before_save :encrypt_password

  # Authenticates user by login name and
  # unencrypted password
  # and returns the user or nil
  def self.authenticate(login, password)
    user = find(:first,
                :select => 'id, salt',
                :conditions => ['login = ?', login])
    return nil if user.nil?

    find(:first, :conditions => ["id = ? AND crypted_password = ?",
                                 user.id, user.encrypt(password)])
  end

  # Encrypts the password with the user salt
  def encrypt(password)
    self.class.encrypt(password, salt)
  end

  # Encrypts some data with the salt provided
  def self.encrypt(password, salt)
    Digest::SHA1.hexdigest("--#{salt}--#{password}--")
  end

  # before filter
  def encrypt_password
    return if password.nil? # guard

    self.salt =
      Digest::SHA1.hexdigest("--#{Time.now.to_s}--#{login}--")
        if new_record?

    self.crypted_password = encrypt(password)
  end

  protected

  def password_required?
    crypted_password.nil? or not password.blank?
  end

end
```

My final **users.yml** and unit test.

```
quentin:
  id: 1
  login: quentin
  salt: 62a636a58d0648eadf7410aa2e4444866174c96e
  crypted_password: be61f3ff72492591afe5081857a8ff17a85b21f9 # quentin
  created_at: <%= 5.days.ago.to_s :db %>
  updated_at: <%= 5.days.ago.to_s :db %>
```

```
require File.dirname(__FILE__) + '/../test_helper'

class UserTest < Test::Unit::TestCase
  fixtures :users

  def test_should_require_login
    u = create_user(:login => nil)
    assert u.errors.on(:login)
  end

  def test_should_ignore_blank_password
    users(:quentin).password = ''
    assert users(:quentin).valid?
    users(:quentin).password = 'foo'
    assert !users(:quentin).valid?
  end

  def test_should_require_unique_login
    u = create_user(:login => 'quentin')
    assert u.errors.on(:login)
  end

  def test_should_require_password
    u = create_user(:password => nil)
    assert u.errors.on(:password)
  end

  def test_should_require_password_confirmation
    u = create_user(:password_confirmation => nil)
    assert u.errors.on(:password_confirmation)
  end

  def test_should_reset_password
    users(:quentin).update_attributes(
                  :password => 'new password',
                  :password_confirmation => 'new password')

    assert_equal users(:quentin),
             User.authenticate('quentin', 'new password')
  end

  def test_should_not_rehash_password
    users(:quentin).update_attribute(:login, 'quentin2')
    assert_equal users(:quentin),
       User.authenticate('quentin2', 'quentin')
  end

  def test_should_authenticate_user
    assert_equal users(:quentin),
     User.authenticate('quentin', 'quentin')
  end

  protected

  def create_user(override = {})
    User.create({:login => 'obie',
             :password => 'qwerty',
             :password_confirmation => 'qwerty'}.merge(override))
  end

end
```

The final test run.

```
C:\workspace\example>rake
(in C:/workspace/example)
c:/ruby/bin/ruby -Ilib;test "c:/ruby/lib/ruby/gems/1.8/gems/rake-
0.6.2/lib/rake/rake_test_loader.rb"
 "test/unit/user_test.rb"
Loaded suite c:/ruby/lib/ruby/gems/1.8/gems/rake-
0.6.2/lib/rake/rake_test_loader
Started
........
Finished in 0.282 seconds.

8 tests, 9 assertions, 0 failures, 0 errors
c:/ruby/bin/ruby -Ilib;test "c:/ruby/lib/ruby/gems/1.8/gems/rake-
0.6.2/lib/rake/rake_test_loader.rb"
```

User Interfaces in Rails

Now that the model is in place we can build a couple of screens, but due to size constraints for the article I will not delve into much detail about them. In particular, I've left out coverage of functional testing – Rails gives you a sophisticated, yet easy to use testing system for controllers. I recommend you read the great description how controllers function and the variables and methods available to them in the Rails docs at:

http://api.rubyonrails.com/classes/ActionController/Base.html

Authentication happens to be one of those aspects of an application that Rails opines should be left to the application developer because it varies too much between projects to be worth incorporating into its layers of abstraction.

A common way used to represent authenticated state in a Rails app is by capturing the current user's id in the session. A nil value for :person_id in the session hash represents unauthenticated state.

Controllers

Rails provides a base class for all application controllers in **application.rb** which is a perfect place for us to insert a before_filter that redirects unauthenticated users to the login page. Notice how naturally the program logic reads very naturally.

```
class ApplicationController
  before_filter :check_authentication

  protected
  def check_authentication
    if session[:person_id].nil? then
     redirect_to :controller => 'login'
  end
end
```

Unauthenticated requests will be redirected to **login_controller.rb**, which handles logging in and out of our application and does not extend ApplicationController.

```
class LoginController < ActionController::Base

  def index
    render :action => 'login'
  end

  def login
    if request.post?
      session[:person]
        = Person.authenticate(params[:login], params[:password])
      if session[:person]
        flash[:notice] = "Successfully logged in"
        redirect_to :controller => 'home'
      else
        flash[:notice] = "Login failed"
      end
    end
  end

  def logout
    reset_session
    redirect_to :action => 'login'
  end
end
```

As you can see above, upon successful login the login controller will redirect the user to the homepage.
Finally, here is a simple **home_controller.rb** which is protected by the filter in ApplicationController. Instance variables (prefixed with an '@' symbol) are copied from the controller instance to whatever templates are used to render the final HTML output.

```
class HomeController < ApplicationController
  def index
                @user = User.find(session[:person_id])
  end
end
```

Templates

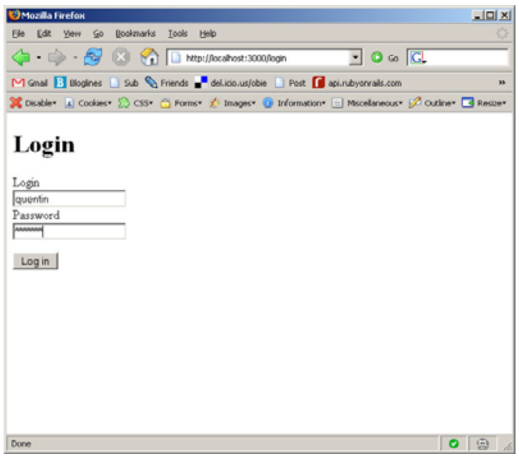
Finally, RHTML files are parsed by a simple Ruby templating engine named ERB. The syntax is very similar to ASP and JSP files.
Here is the relatively simple **views/home/index.rhtml** template:

```
<h1>Hello, <%= @user.login %></h1>
```

…and also the *views/login/login.rhtml* template. Notice how the folder and name of the template file follows the controller and action name.

```
<div class="centered">
 <div id="login" class="card">
 <h1>Login</h1>
 <%= form_tag :action => 'login' %>
 <p>
  <label for="login">Login</label><br/>
  <%= text_field_tag 'login' %><br/>
  <label for="password">Password</label><br/>
  <%= password_field_tag 'password' %>
 </p>
 <p><%= submit_tag 'Log in' %></p>
 <%= end_form_tag %>
 </div>
</div>
```

Here's the screen you get from this:


Mozilla Firefox
File Edit View Go Bookmarks Tools Help
http://localhost:3000/login
Go
Gmail Bloglines Sub Friends del.icio.us/obie Post api.rubyonrails.com
Disable Cookies CSS Forms Images Information Miscellaneous Outline Resize
Done

Login

Login
quentin
Password
Log in

Rails provides many helper methods such as form_tag and text_field_tag which simplify creation of HTML fields with dynamic content.

Conclusion

I hope that via this introductory article I've communicated some of the magic of Ruby on Rails and given you a taste for why it's quickly becoming the most popular web application framework out there. I tried to hit on the main points and tools that you will want to familiarize yourself with as you get started with Rails programming. If you're interested in learning more I strongly recommend the excellent Rails book from Pragmatic Programmers: *Agile Web Development with Rails* written by Dave Thomas and David Heinemeir Hansson.

Obie Fernandez is author of the forthcoming book, "Extending Ruby on Rails (Into the Enterprise)", which is scheduled to launch the new "Addison-Wesley Professional Ruby Series" in Fall, 2006, for which
Obie will also serve as Series Editor. "Extending Ruby on Rails" will be the flagship book in a robust library of learning tools for how to leverage Ruby-based technology in the enterprise, along with the newest agile development techniques.

advertisment

IASA

international association of software architects

The IASA was created by architects for architects to solve the major problems facing IT architects today. Our resources are scarce: Architects have very few resources to work with. It is often very difficult to find the materials we need to do our jobs. It is difficult to judge quality: Very few people agree on the fundamental components of architecture or what it takes to become a qualified architect. Our experience isn't portable: Every employer defines and uses architecture differently. Lack of community: There is no way to quickly find peers of our own calibre to discuss the issues which drive us.
Joining the IASA is very simple. You can register at your local chapter or on the IASA website at www.IASAhome.org.

advertisment

ObjectiveView Training
Application Architecture
1 or 2 days
Agile Project Management
1 day
Use Case Driven Object Modelling
(Iconix Process)
* Jumpstart (work on your project) *
5 days
* Public Schedule *
2 days
C #
4 days
C ++
4 days
Ruby/Rails
3 days
AJAX
2 days
Java
4 days
ovtrainingusa@objectiveviewmagazine.com (N. America)
ovtraininguk@objectiveviewmagazine.com (UK and Europe)
ovtrainingasia@objectiveviewmagazine.com (Asia)
www.objectiveviewmagazine.com

http://www.objectiveviewmagazine.com/

Opinion ♦ Specs Are Bad ♦ Rebecca Wirfs-Brock ♦ Opinion

It's official... specs are "bad" …according to Linus Torvalds. I have to chime in on Linus' newsgroup posting and the attendent buzz it sparked on the net recently (and on the Linux Kernel mailing list).

Linus stated:

"So there's two MAJOR reasons to avoid specs: they're dangerously wrong. Reality is different, and anybody who thinks specs matter over reality should get out of kernel programming NOW. When reality and specs clash, the spec has zero meaning. Zilch. Nada. None. It's like real science: if you have a theory that doesn't match experiments, it doesn't matter *how* much you like that theory. It's wrong. You can use it as an approximation, but you MUST keep in mind that it's an approximation. Specs have an inevitably tendency to try to introduce abstractions levels and wording and documentation policies that make sense for a written spec. Trying to implement actual code off the spec leads to the code looking and working like crap."

He went on to conclude:

"But the spec says ..." is pretty much always a sign of somebody who has just blocked out the fact that some device doesn't. So don't talk about specs. Talk about working code that is *readable* and *works*. There's an absolutely mindbogglingly huge difference between the two.

This posting launched an onslaught of discussion. Linus is right. Reality always differs from a specification of how software is supposed to behave. That's a reflection on how difficult it is to write precise specifications of behavior and on how many decisions during implementation are left open . Still, I'm not willing to say "no specs, ever" even though I'm a signer of the Agile Manifesto and on the board of the Agile Alliance. We need to get better at recognizing what types of descriptions do add value and under what circumstances. And become more aware of when and where precision is needed (and when it drags us down).

Linus points out that specs often introduce abstractions and concepts that shouldn't be directly implemented in code. I never expect to directly translate what someone writes into code without using my brain. I design and think before and during and after coding…and nothing substitutes for testing/proving out a design and implementation against the real environment it works in.

But that doesn't mean specs have no value. Working, readable code isn't the only thing that matters. It matters very much in the short and long term. But try understanding design rationale by just reading code. Or reading the test code. It's difficult, if not impossible. I find value in design documentation that explains the tricky bits. This type of documentation is especially valuable when those coding aren't going to hang around to offer explanations.

A spec. is an approximation of what is desired. I certainly don't expect it to tell me everything. There can be enormous value in writing descriptions of what software should do—especially when it is important to communicate design parameters and system behaviors instead of just providing an implementation. Most developers aren't good at writing specs, let alone descriptions/discussions about their code and design choices. But that doesn't mean they should stop writing them and resort to "organic code growth" in every situation. A firm believer in agile practices, I don't insist on writing merely for fun or because it is expected.

But if I need a spec, I write it. And if doesn't reflect reality or is misunderstood, I change it if there is value in keeping it up to date. There may not be. And if that's the case, I don't update it. It depends on the project and the need. It helps if I write these descriptions for someone who wants to read them (and will actually use it rather than toss it aside). I've got to know my audience. That often takes experimentation. Maybe I need to include sample prototype code in addition to design notes/models/sketches. Maybe I don't.

Communicating ideas to a diverse audience is especially hard. But specs aren't the problem. It's that effectively communicating how something works or should work is more difficult than cutting code. I prefer working code over piles of outdated, difficult diagrams and explanations. But that doesn't duck the issue. Specs aren't inherently bad. Most spec writers would rather be doing something else. And that is a problem…

Rebecca Wirfs-Brock, author of *Object Design: Roles, Responsibilities and Collaborations*, invented the way of thinking about objects known as Responsibility-Driven Design. Find out more out Rebecca and her services at http://www.wirfs-brock.com.

The Glacial Methodology: A Data-Oriented Approach to Software Development – Scott Ambler

Although object technology was introduced within the business community in the early 1990s, popularized in the late 1990s, and finally adopted as the defacto technology platform for new development in the early 2000s, it has clearly failed to deliver on its promise…

It is time to admit this to ourselves and abandon the "modern" software development techniques of the object community; Instead, we must re-embrace the tried and true approaches of the past. Now is the time for the Glacial Methodology™.

The Glacial Methodology is composed of seven distinct phases:

1. **Project Initiation**. During this phase you put together your project team and project plan. A Glacial team is typically comprised of a Project Manager, a Data Team lead (the person who is actually in charge), a Senior Data Architect, several Junior Data Architects, several Data Analysts, a Database Administrator (DBA), one or two application developers, and perhaps a tester or two. Your project plan will depict the milestones crucial to your project success: The delivery of the conceptual data model, the logical data model, and the physical data model.
2. **Conceptual Modeling**. Your conceptual data model forms the foundation upon which all development is based and that it is therefore imperative that you start with a highly detailed, reviewed, and accepted model. It is possible to finalize a conceptual data model within the first six to eight weeks of your project.
3. **Logical Data Modeling**. A logical data model (LDM) fleshes out business entities with detailed descriptions of their data attributes and the relationships between entities. Data requirements are gathered during this stage, information which is captured within your LDM as well as in your data requirements specification – Glacial data professionals understand the importance of comprehensive documentation.
4. **Physical Data Modeling**. You develop a detailed physical data model (PDM) based on your reviewed and accepted LDM. The PDM is used to generate your database schema and to help drive the modeling efforts of the application developers.
5. **Application Development**. Following the traditions of the traditional data community, the Glacial Methodology ignores trivial issues which are better left to application development teams. These issues include, but are not limited to understanding the usage requirements for the system, system usability, security, network architecture, hardware architecture, deployment, scheduling, estimation, component design, user interface design, overall functionality, and testing. These minor issues will be addressed by the development team when it goes rogue by choosing to ignore all of the previous modeling work.
6. **System and Acceptance Testing**. There is likely no time left at the end of the project, so there is little need to discuss this.
7. **Deployment**. Assuming the project hasn't been cancelled by this point, you will deploy whatever was actually built into production.

The Glacial Methodology is described in detail at www.ambysoft.com/essays/glacialMethodology.html, and yes it's an April Fool's joke (*editor's note: you are reading this one April 1 – aren't you?*). If your organization follows a Glacial approach you've likely got a serious problem, and you might want to visit www.agiledata.org/essays/differentStrategies.html for a detailed discussion of better options.

See also www.waterfall2006.com for similar items!

Have Your Say!
Join the ObjectiveView discussion and feedback group.
Simply visit:
groups.google.com/group/objectiveviewdiscussion

Subscribe to ObjectiveView
email: objective.view@ratio.co.uk subject: subscribe

Aspect-Oriented Programming with AspectJ

Object-oriented programming (OOP) is, without any doubt, one of the most important programming paradigms in the history of software engineering. In spite of this achievement, object-oriented programming has demonstrated to be inadequate in dealing with crosscutting concerns [2]. Alex Ruiz explains …

A *concern* is a particular area of interest in an application. A crosscutting concern is a system-wide concern which implementation might affect several modules.

Indicators of a crosscutting concern are:

- **Code tangling.** Code tangling is the result of having modules in a system that deal with more than one concern at the same time. A good example is the modules of a banking system that simultaneously interact with multiple requirements like business logic, performance, logging, transaction management and security. Implementing each concern adds too many elements to the system, which results in too complicated code.
- **Code scattering.** Code scattering happens when the implementation of a crosscutting concern spreads across several modules. There are two types of code scattering. The first type is code duplication, which increases code complexity. As an example, classic transaction management requires inserting nearly identical code to multiple modules in an application.

The second type of code scattering takes place when the implementations of *complementary* pieces of a concern are distributed loosely across multiple modules [3]. For example, in a common implementation of caching, some modules retrieve objects from the cache while other modules invalidate cache regions to prevent the storage of stale data.

Example – Caching as a Crosscutting Concern

In this article we are going to use the crosscutting concern of caching as example. Our target is the following implementation of a customer manager, which retrieves a customer from a data source given the customer's id – but apparently *contains no explicit caching code*:

```
public class CustomerManager() {

   private CustomerDao customerDao;

   public Customer getCustomer(String customerId) {
      return this.customerDao.getCustomer(customerId);
   }

   // rest of class implementation
}
```

Listing 1. Simple implementation of a customer manager.

Let's assume that the retrieval of the customer from the data source is a very expensive operation. One way to improve the performance of our application is storing the retrieved customer in a cache. In other words, we are going to apply caching to the return value of the method *CustomerDao.getCustomer(String):*

```
public class CustomerManager() {

   private static final Object NULL_OBJECT = new Object();

   private CacheManager cacheManager;
   private CustomerDao customerDao;

   public Customer getCustomer(String customerId) {
      String key = "customer." + customerId;
      Customer customer = null;
      Object cachedEntry = null;

      try {
         cachedEntry = this.cacheManager.get(key);
      } catch (CacheException e) {
        logCacheException(e);
      }

      if (cachedEntry == null) {
         customer = this.customerDao.getCustomer(customerId);

         Object objectToCache = customer;
         if (objectToCache == null) {
            objectToCache = NULL_OBJECT;
         }

         try {
            this.cacheManager.put(key, objectToCache);
         } catch (CacheException e) {
            logCacheException(e);
         }

      } else if (cachedEntry != NULL_OBJECT) {
         customer = (Customer) cachedEntry;
      }
      return customer;
   }

   // rest of class implementation
}
```

Listing 2. OOP based - non-aspect implementation of caching.

As we can see in listing 2, implementing caching user OOP alone added 20 lines of *entangled* code, none of them related to the main responsibility of the method. If we needed to apply caching to other parts of our system, we would end up having the same caching logic *scattered* all over our application. Code tangling and code scattering have the following (negative) impact on our software:

- **Complexity.** Code is harder to understand and the main concern is a lot harder to see. In our example, only the line in bold implements the real job of the method.
- **Maintenance.** Code is harder to maintain. Any maintenance work involving the implementation of one of the concerns might affect the rest of them. At the same time, modules implementing multiple concerns simultaneously tend to be maintained more frequently than modules implementing a single concern [4].

- **Testability.** Code is harder to test. In our example, we need a cache manager, or at least a mock object simulating it, to test the business logic of our method.
- **Code reuse.** Code is harder to reuse. It is difficult (and sometimes impossible) to take modules that implement multiple concerns and reuse them in systems requiring a different implementation of those concerns or a totally different set of concerns [3]. In our example, systems that need to retrieve customer information from a data source may not be able to reuse the `CustomerManager` class if they have different caching needs or no caching needs at all.

We can try to minimize code tangling and code scattering by refactoring the code in listing 2, creating a caching module that provides an abstract API which hides its implementation details. This is shown visually in figure 1 - and has somewhat minimized code duplication by centralizing the calls to the cache manager into a single module. The problem we try to solve is still present, just in slightly different form –the calls to the new caching API are tangled into the client modules and scattered through the system.

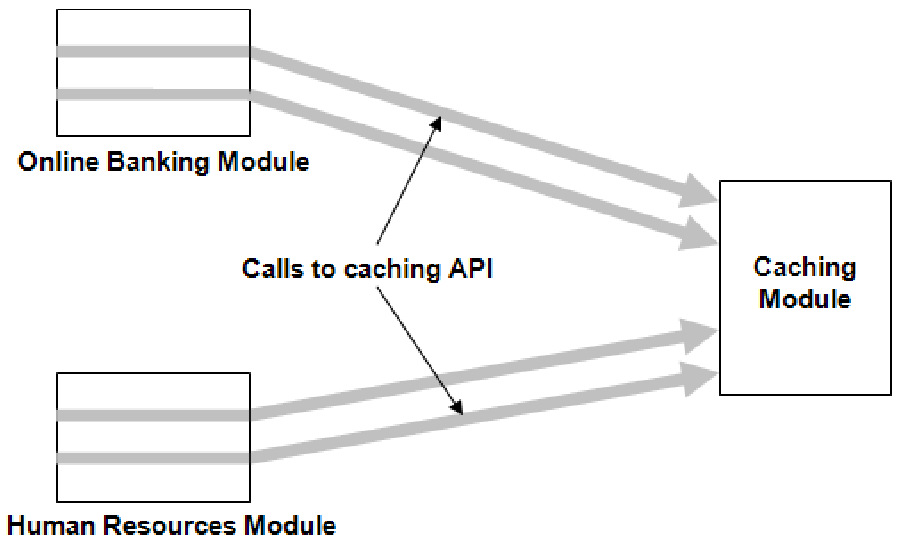


Figure 1. Implementation of caching using OOP [3]. The caching logic has been centralized in a module. Calls to the caching API are still embedded in the client modules and scattered across the system.

AspectJ to the Rescue

Aspect-oriented programming (AOP) helps us solve these problems by providing *separation of crosscutting concerns* [5]. AOP offers another way to structure our programs by assembling the code implementing each concern into its own module. Instead of having code scattered chaotically around an object model, AOP allows us to modularize crosscutting concerns into new units of work called *aspects* [2]. AspectJ, the most complete implementation of AOP, is a language extension to Java that treats AOP concepts as first-class citizens of the language [2].

Modularity is not the only benefit of aspects. Aspects also *encapsulate* the implementation details of a crosscutting concern. Unlike classes, aspects not only hide *how* something is done, but also *when* [4]. By modularizing and encapsulating the implementation details of a crosscutting concern, an aspect becomes a *unit of abstraction* for that concern. Modularity, encapsulation and abstraction make AOP a powerful tool against software complexity, a tool that *complements* OOP where it is weak –in the implementation of crosscutting concerns.

Figure 2 gives a visual representation of a caching concern modularized in an aspect. The core modules do not contain any calls to the caching API. In fact, the core modules are not even aware of the use of caching.

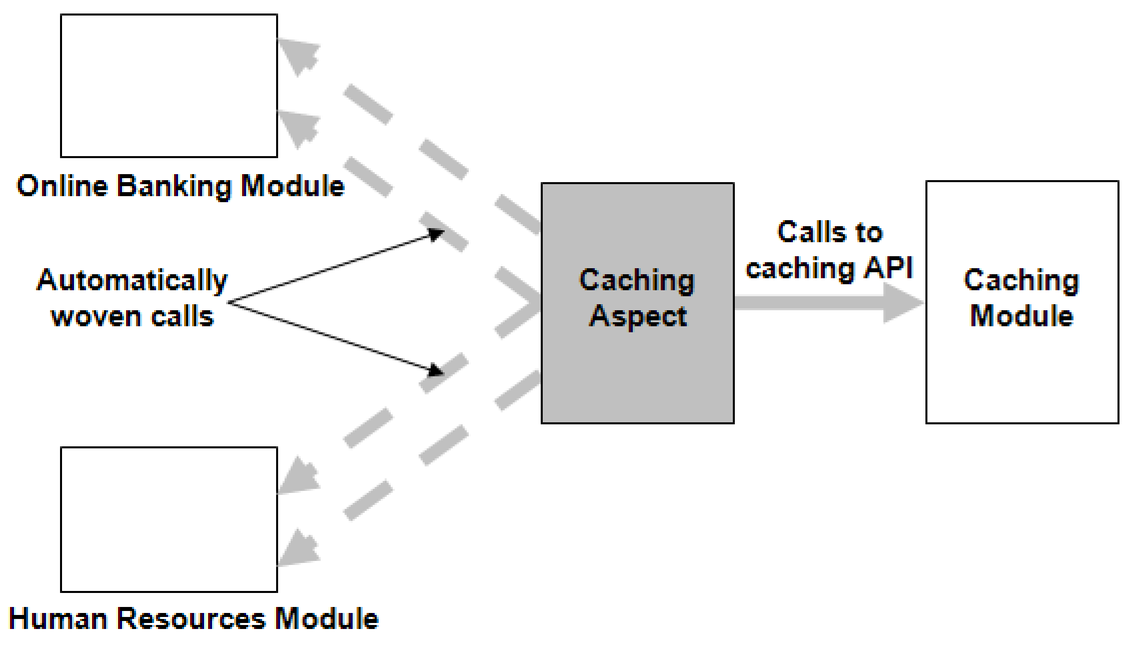


Figure 2. Implementation of caching using AOP [3]. The crosscutting caching requirement has been modularized into a single module: the caching aspect. This way, any changes made to the caching concern will only affect the caching aspect leaving the core modules intact.The AspectJ code for this is shown in the following listing, which we'll discuss in the following sections:

```
public aspect CachingAspect() {
   private static final Object NULL_OBJECT = new Object();

   private CacheManager cacheManager;

   pointcut getCustomerOperation(String customerId) :
      execution(public Customer CustomerManager.getCustomer(String))
      && args(customerId);


   Customer around(String customerId) :
      getCustomerOperation(customerId) {

      String key = “customer.” + customerId;
      Customer customer = null;
      Object cachedEntry = null;

      try {
         cachedEntry = this.cacheManager.get(key);
      } catch (CacheException e) {
        logCacheException(e);
      }
      if (cachedEntry == null) {
         customer = proceed(customerId);

         Object objectToCache = customer;
         if (objectToCache == null) {
            objectToCache = NULL_OBJECT;
         }

         try {
            this.cacheManager.put(key, objectToCache);
         } catch (CacheException e) {
            logCacheException(e);
         }

      } else if (cachedEntry != NULL_OBJECT) {
         customer = (Customer) cachedEntry;
      }
      return customer;
   }

   // rest of aspect implementation
}
```

Listing 3. Implementation of a caching aspect using AspectJ

As we can see, the caching aspect has effectively solved the problem of code tangling and code scattering. This approach raises a new issue – if the core modules are not aware of the caching concern, how does the system know where and when caching should be executed?

Weaving

The answer to this question relies in the AOP concept of *weaving rules*. Weaving rules specify how to integrate the implemented concerns to build the final system [3]. The weaving rules for our example identify the points during program execution where caching should be performed, the information to be stored in the cache, and so forth. The system then uses these rules to properly call the caching API from the specified operations. The process that integrates the concerns into the final system following the weaving rules is called *weaving*.

AspectJ's weaving rules are based on three concepts:

- **Join points.** A *join point* is a point during the execution of a program. For example, a method invocation or field access.
- **Advice.** An *advice* is an action taken at a particular join point. For example, the logic that performs caching.
- **Pointcut.** A *pointcut* describes a set of join points, defined to specify when an advice should execute. For example, a pointcut that identifies the join point defined by the method `getCustomer` in listing 2.

Weaving can occur at compile time, post-compile (to weave into existing class or jar files), or at load time (on class load) – though it's beyond the scope of this article to discuss this further.

Join points

AspectJ exposes a number of different types of *join point* within a program [3]:

- Method call and execution
- Constructor call and execution
- Read/write access to a field
- Initialization of classes and objects
- Execution of an exception handler

It is at these points that AspectJ may weave additional code to implement cross-cutting concerns. Figure 3 shows various join points that will be exposed during the execution of a *createCustomer* method in a *CustomerManager* class:

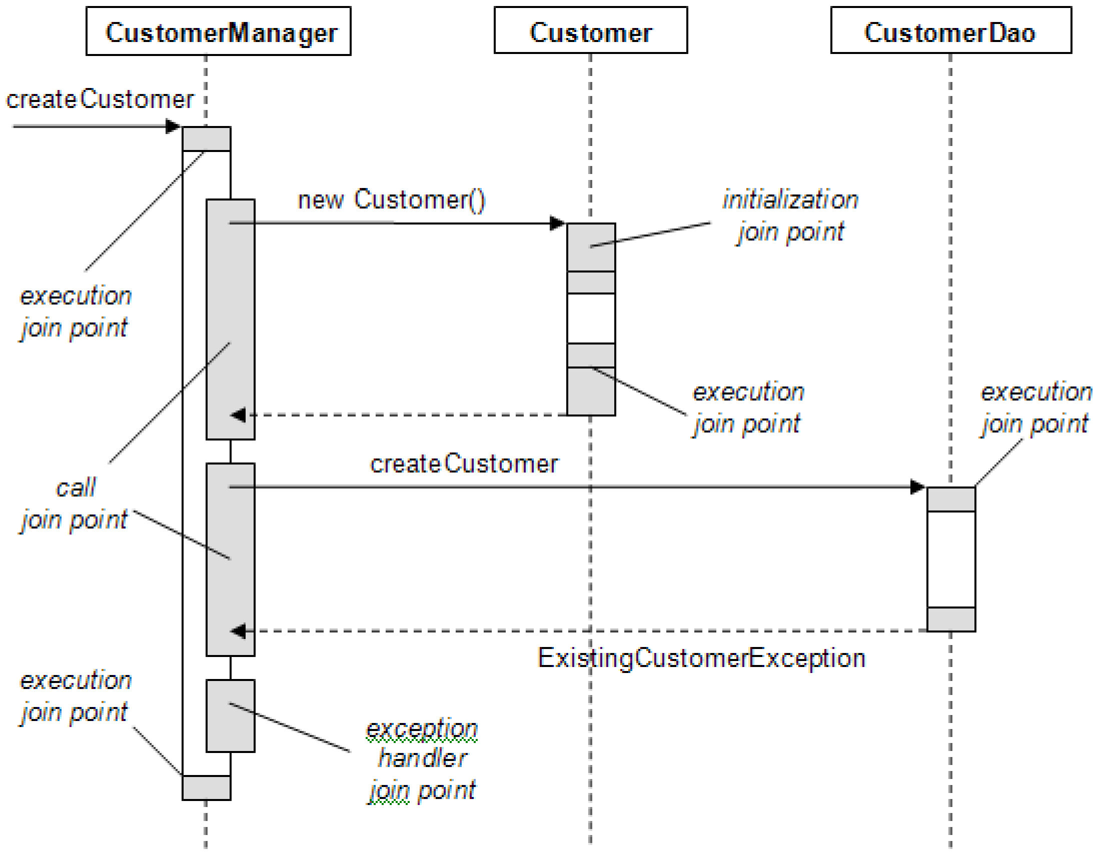


Figure 3. Join points in program execution. It is at these points that AspectJ may (subject to the correct pointcuts being used) weave in additional code.

Aspects, Pointcuts and Advice

Aspects

The *aspect* is the central unit of work in AspectJ, in the same way that a *class* is the central unit of work in Java. An aspect combines pointcuts, advices and declarations. In addition to those elements, aspects can contain data, methods and nested classes, just like regular Java classes [3]. The general form of an aspect is:

```
[access-modifier] aspect name [extends class-or-aspect]
   [implements interface-list] {

   ...aspect body
}
```

Here is the definition of the caching aspect from listing 3:

```
public aspect CachingAspect {
   // aspect body
}
```

Listing 4 – fragment of declaration of the caching aspect. As we can see it is very similar to the declaration of a class (taken from listing 3).

Aspects themselves are fairly complex creatures – and may be declared *abstract* (and inherited from) or declared as implementing a particular *interface* – but for the purposes of this article the main point is that they are used to encapsulate elements of a particular *concern* (such as caching) in an application. But note in particular that, unlike classes, it makes no sense to think about *instantiating* an aspect. The AspectJ compiler uses the aspect to weave behaviour into other parts of the application.

Pointcuts

Pointcuts are used to identify particular join points in the program flow for which we want AspectJ to weave in additional behaviour (using advice, as we'll see). Pointcuts can be declared inside an aspect, a class, or an interface. As with any other Java artifacts, pointcuts can have access modifiers (public, protected, private or default) to restrict access to them.

The general form of named pointcut definition AspectJ is:

```
[access-modifier] pointcut name([parameter-list]) :
  pointcut-expression;
```

Here is the declaration of a named pointcut taken from listing 3:

```
  pointcut getCustomerOperation(String customerId) :
     execution(public Customer CustomerManager.getCustomer(String))
     && args(customerId);
```

Listing 5. Definition of a named pointcut. The part after the colon defines the join point we want to identify: any methods starting with "get" in the class *CustomerManager*. **Note the use of the args() pattern to store a customerId for later use in** ***advice*****.**

Pointcuts may be used anonymously – without an explicit name – but most of the time it's sensible to name them. AspectJ provides a variety of different types of patterns (as in textual pattern matching) to use in contructing pointcuts – to enable the different types of join points discussed earlier to be matched. These include *type-signature patterns, method and constructor signature patterns, field signature patterns, lexical-flow based pointcuts, control-flow based pointcuts, argument pointcuts* and last but not least *execution object pointcuts* (phew!).

```
pointcut allGettersExceptCustomerOperations() :
  getterOperations() &&  !(* CustomerManager.get*(..))
```

An example pointcut using logical operators (&&, !, etc.) to match appropriate join points.

AspectJ also provides a fairly rich set of operators and wild-cards to enable wide or narrow matching as required.

Wildcard	Definition
*	Matches any number of characters except the period
..	Matches any number of characters including any number of periods
+	Matches any subclass or subinterface of given type

wildcards for use in constructing pointcuts

Advice

Once we've matched some join points with a pointcut, we need to specify the behaviour to be inserted – that's what *advices* are for. There are three types of advices:

- *Before advice*: specify behaviour to be executed before the join point is invoked
- *After advice*: specify behaviour to be executed after the join point is invoked
- *Around advice*: specify behaviour to be executed around the join point's execution. The advice is given control and responsibility for invoking the join point, in addition to doing its own work [2].

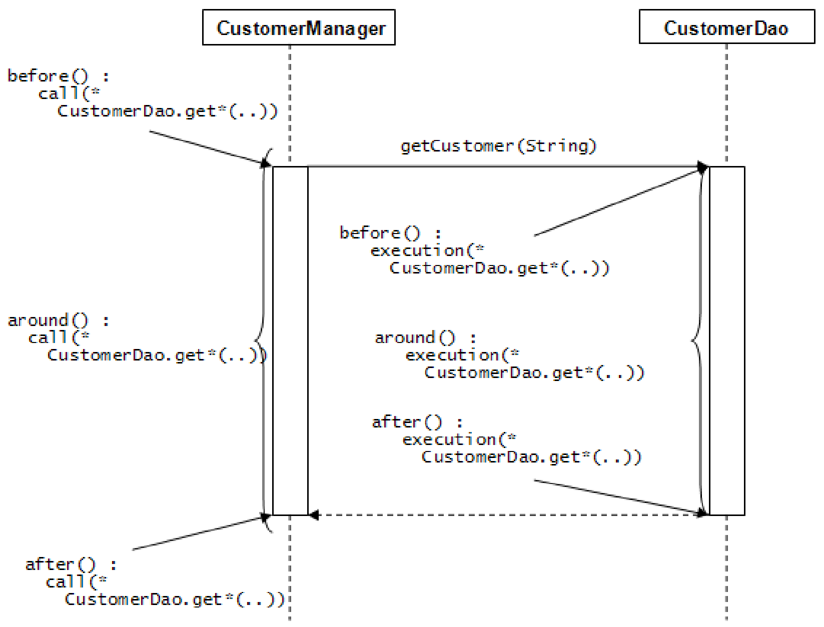

CustomerManager
CustomerDao
before() : call(* CustomerDao.get*(..))
getCustomer(String)
before() : execution(* CustomerDao.get*(..))
around() : call(* CustomerDao.get*(..))
around() : execution(* CustomerDao.get*(..))
after() : execution(* CustomerDao.get*(..))
after() : call(* CustomerDao.get*(..))

Figure 4. Before, after and around advices applied to different join points in the program flow.

All types of advice are declared using the same baic structure [4]:

```
[strictfp] advice-specification [throws type-list] :
        pointcut-expression {
    ...body of advice
}
```

Advices cannot be called explicitly; therefore there is no need for access modifiers. The only modifier allowed is `strictfp`, which makes all floating-point within the body of the advice FP-strict [4]. The optional `throws` clause specifies the exceptions that the advice may throw. Advices cannot throw checked exceptions that the clients invoking the join point are not expecting.

Let’s look at the different types of advice in more detail:

Before Advice

The before advice executes before the captured join point is invoked. In figure 4 we have an advice that executes before calling the method *getCustomer* in the class *CustomerDao*:

```
before() : call (* CustomerDao.get*(..)) {
   // advice body
}
```

If an exception is thrown in the before advice, the operation in the capture join point will not be executed [3].

After advice

The after advice executes after the captured join point is invoked. AspectJ offers three types of after advices [2]:

- **After returning advice** (executed after the successful completion of a call, in which no exception was thrown) in the form:

```
after() returning : call (* CustomerDao.get*(..)) {
   // advice body
}
```

- **After thowing advice.** It is executed after the join point throws a particular exception:

```
after() throwing : call (* CustomerDao.get*(..)) {
   // advice body
}
```

- **After advice.** It is executed after any call to the join point, regardless of whether it threw an exception:

```
after(): call (* CustomerDao.get*(..)) {
   // advice body
}
```

Around advice

The around advice surrounds the captured join point. It is the most powerful type of advice because:

- It can bypass the executed of the captured join point
- It can execute the captured join point with the same or different arguments
- It can execute the captured join point more than once, each time with the same or different arguments

To execute the operation inside the captured pointcut we need to use the keyword *proceed*`()` in the body of the advice [3]. If *proceed*`()` is not called, the execution of the join point is completely bypassed. Any call to *proceed*`()` should contain the same number and same type of parameters that the captured operation expects. At the same time, *proceed*`()` returns the same value returned by the captured operation [3].

Join Point Category	Pointcut Syntax
Method execution	execution(*method-signature*)
Method call	call(*method-signature*)
Constructor execution	execution(*constructor-signature*)
Constructor call	call(*constructor-signature*)
Class initialization	staticinitialization(*type-signature*)
Field read access	get(*field-signature*)
Field write access	set(*field-signature*)
Exception handler execution	Handler(*type-signature*)
Object initialization	initialization(*constructor-signature*)
Object pre-initialization	preinitialization(*constructor-signature*)
Advice execution	adviceexecution()

Pointcut syntax summary.

Here is the *around* advice taken from listing 3:

```
  Customer around(String customerId) :
     getCustomerOperation(customerId) {

     String key = “customer.” + customerId;
     Customer customer = null;
     Object cachedEntry = null;

     try {
        cachedEntry = this.cacheManager.get(key);
     } catch (CacheException e) {
       logCacheException(e);
     }

     if (cachedEntry == null) {
        customer = proceed(customerId);

        Object objectToCache = customer;
        if (objectToCache == null) {
           objectToCache = NULL_OBJECT;
        }

        try {
           this.cacheManager.put(key, objectToCache);
        } catch (CacheException e) {
           logCacheException(e);
        }

     } else if (cachedEntry != NULL_OBJECT) {
        customer = (Customer) cachedEntry;
     }
     return customer;
  }

  // rest of aspect implementation
}
```

Even though the code is listing 3 is far from perfect, it clearly demonstrates a complete concern being removed from the *Customer* class and being implement in a modularly distinct *CachingAspect*.

By letting AspectJ weave the aspect in listing 3 with the code in listing 1 – we get an aspect oriented version of code shown in listing 2 – with the two concerns clearly separated.

Inter-type declarations

Changing tack slightly, let’s now take a look at some other AspectJ capabilities, in particular those related to inter-type declarations (formally known as introductions):

Inter-type declarations are declarations that cut across classes and their hierarchies [7]. Inter-type declarations modify the static structure of types (classes, interfaces and aspects) and their compile-time behavior [3]. Again, they are used to assist in the seperation of cross-cutting concerns into distinct modular units.

Inter-type declarations can be used to introduce fields, methods and constructors to an existing class, as well as to manipular interface and inheritance hierarchies in a controlled manner.

Field introduction

Field introduction is used to add a new attribute to an existing class – but in a manner which separates the new code (a different concern) from the original class. The general form of an inter-type field is [4]:

```
[modifiers] field-type target-type.field-name;
```

In the following example, we are introducing the field *listeners* which “listen” to any change made to the properties of the class *Customer* (from figure 5):

```
private List<PropertyChangeListeners> Customer.listeners =
   new ArrayList<PropertyChangeListeners>();
```

The *listeners* field is not part of the state of a customer, and the declaration and use of this field clutter the implementation of the *Customer* class. Furthermore, listening to property changes is not the responsibility of the *Customer* class.

Method and constructor introduction

The general form of an inter-type method is [4]:

```
[modifiers] return-type target-type.method-name([parameter-list])
   [throws type-list] { method-body }
```

and an inter-type constructor:

```
[modifiers] target-type.new([parameter-list])
   [throws type-list] { method-body }
```

Like inter-type fields, inter-type methods can have *public*, *private* or default access, but cannot have *protected* access. The declaration of an inter-type method is similar to the declaration of any ordinary method, with the addition of *target-type*.

Here we add an inter-type constructor registration of the listeners to the example introduced previously:

```
public Customer.new(String customerId) {
   this.customerId = customerId;
}

private List<PropertyChangeListeners> Customer.listeners =
   new ArrayList<PropertyChangeListeners>();

public void Customer.addListener(PropertyChangeListener listener) {
   listeners.add(listener);
}

public Listener Customer.removeListener(int index) {
   return listener.remove(index);
}
```

As we can see, we can refer to the inter-type field *listeners* in the same way we would refer to any regular field declared in the *Customer* class. Because the introduced methods are *public*, they can be called from anywhere in the application.

Note again: what we’ve effectively done here is separated the implementation of a concern (listening) from an existing class (Customer) – thereby making a substantial improvement to the structure of our source code – separating out different concerns.

advertisment

ObjectiveView Training
Application Architecture
1 or 2 days
Agile Project Management
1 day
Use Case Driven Object Modelling
(Iconix Process)
* Jumpstart (work on your project) *
5 days
* Public Schedule *
2 days
C #
4 days
C ++
4 days
Ruby/Rails
3 days
AJAX
2 days
Java
4 days
ovtrainingusa@objectiveviewmagazine.com (N. America)
ovtraininguk@objectiveviewmagazine.com (UK and Europe)
ovtrainingasia@objectiveviewmagazine.com (Asia)
www.objectiveviewmagazine.com

http://www.objectiveviewmagazine.com/

Interface and super type manipulation

Let's assume we need to store one or more instances of *Customer* in a *List*. Later on, we realize we need to display in a web page the list of customers sorted by last name in descending order.

One way to do this is having the *Customer* class implement the *Comparable* interface. If we choose to take this route, we will have to implement the *compareTo* method too. Since this extra code is not part of the state of a *Customer*, we probably want to separate this functionality from the implementation of the *Customer* class. Here is how we could achieve this separation using AspectJ:

```
declare parents : Customer implements Comparable;

public int Customer.compareTo(Object obj) {
   // method implementation
}
```

In this simple example, we made the *Customer* class implement the *Comparable* interface. By using method introduction, we specified the implementation of the methods such interface.

We can also use the construct *declare parents* to change the super-type of a set of classes matching a type pattern [4]. For example:

```
declare parents : Customer extends Observable;
```

And finally… @AspectJ Annotations

In AspectJ 5 it is possible to write aspects using plain Java. AspectJ provides a set of Java 5 annotations that identify regular Java classes as aspects so they can be interpreted by the weaver. The AspectJ annotations are located in the *org.aspectj.lang.annotation* package.

Following are some examples of AspectJ elements rewritten using Java:

AspectJ language	Java with AspectJ Annotations

public aspect MyAspect {
  // aspect body
}
	@Aspect
	public class MyAspect {
	  // aspect body
	}

pointcut aCall() :
  call(* *.*(..));
	@Pointcut("call(* *.*(..))")
	void aCall() {}

before() : aCall() {
  // advice body
}
	@Before("aCall()")
	void aCallAdvice() {
	  // advice body
	}

Let's rewrite the aspect in listing 3 using plain Java:

```
@Aspect
public class CachingAspect() {

   private static final Object NULL_OBJECT = new Object();

   private CacheManager cacheManager;

   @Pointcut("execution(" +
     "public Customer CustomerManager.getCustomer(String)) " +
     "&& args(customerId)")
   void getCustomerOperation(String customerId) { }

   @Around("getCustomerOperation(customerId)")
   public Customer onCustomerOperation(
      ProceedingJoinPoint thisJoinPoint,String CustomerId) {

      String key = "customer." + customerId;
      Customer customer = null;
      Object cachedEntry = null;

      try {
         cachedEntry = this.cacheManager.get(key);
      } catch (CacheException e) {
        logCacheException(e);
      }

      if (cachedEntry == null) {
         customer =
            thisJoinPoint.proceed(new Object[] {customerId});

         Object objectToCache = customer;
         if (objectToCache == null) {
            objectToCache = NULL_OBJECT;
         }

         try {
            this.cacheManager.put(key, objectToCache);
         } catch (CacheException e) {
            logCacheException(e);
         }

      } else if (cachedEntry != NULL_OBJECT) {
         customer = (Customer) cachedEntry;
      }
      return customer;
   }

   // rest of aspect implementation
}
```

Listing 7. Aspect written in plain Java.

The aspect in listing 7 shows how to implement *proceed()* in Java. If we simply call *proceed()* like we did in the aspect in listing 5 we will get an compilation error because the method does not exist. We need to pass an instance of *org.aspectj.lang.ProceedingJoinPoint* as argument of the method annotated as an around advice. *ProceedingJoinPoint* exposes the *proceed(..)* method in order to support around advice.

Aspects created using plain Java are compiled by the regular Java 5 compiler, and then woven by the AspectJ weaver as an additional build step, for example. Writing aspects in plain Java comes in handy when working with IDEs that do not have support for the AspectJ language extensions (e.g. NetBeans.) Please note that aspects written in plain Java and AspectJ language can be mixed within the same application.

Conclusions

Aspect-Oriented Programming focuses on the modularization, encapsulation and abstraction of crosscutting concerns. AOP does not aim to compete with OOP. Instead, AOP complements OOP where it is weak –in the implementation of crosscutting concerns.

AspectJ provides the most complete implementation of AOP. It offers extensions for the Java language that define the elements necessary for the implementation of crosscutting concerns. Since AspectJ is an extension of Java, developers will find it easy to learn. In this article, we just scratched the surface of what can be done with AspectJ. The new features added in version 5 like load-time weaving and the ability to write aspects in plain Java make AspectJ one of the most powerful tools against software complexity.

References

[1] E. Gamma et al, “Design Patterns: Elements of Reusable Object-Oriented Software”, Addison-Wesley, 1995

[2] R. Johnson and J. Hoeller, “J2EE Development Without EJB”, Wrox Press, 2004

[3] R. Laddad, “AspectJ in Action: Practical Aspect-Oriented Programming”, Manning, 2003

[4] A. Colyer et al, “eclipse AspectJ”, Addison-Wesley, 2005

[5] I. Jacobson and P. NG, “Aspect-Oriented Software Development With Use Cases”, Addison-Wesley, 2004

[6] Wikipedia at: http://en.wikipedia.org/wiki/Aspect-oriented_programming (last visited, January 2006)

[7] The AspectJ Programming Guide at: http://www.eclipse.org/aspectj/doc/released/progguide/index.html (last visited, January 2006)

[8] S. Clarke and E. Baniassad, “Aspect-Oriented Analysis and Design. The Theme Approach”, Addison-Wesley, 2005

Alex Ruiz has designed and developed software for the past 7 years. After suffering a traumatic experience with waterfall software development, Alex has embraced XP and has carried it in his heart ever since. Alex enjoys reading anything related to Java, J2EE, AOP and lightweight containers and has programming as his second love. Alex is a proud employee of ThoughtWorks and is an active member and committer of the Spring Framework community.

Advertisment

Agile UML®
Leverage the Power of UML 2.0 and the Agile/ICONIX Process to spec your software designs. Design a real mapping example from use cases through C#/.Net code in our open enrollment, public training workshops.
ENTERPRISE ARCHITECT
Agile Development with ICONIX Process
People, Process, and Pragmatism
Doug Rosenberg, Matt Stephens, and Mark Collins-Cope
Optimized for Microsoft Visual Studio
SPARX SYSTEMS
www.sparxsystems.com
ICONIX Leaders in Object Technology
www.iconixsw.com
UML Solution Partners
All product names are owned by their respective owners

Historical Perspectives ♦ Goto Considered Harmful ♦ Edsger W. Dijsktra

For a number of years I have been familiar with the observation that the quality of programmers is a decreasing function of the density of go to statements in the programs they produce…

More recently I discovered why the use of the go to statement has such disastrous effects, and I became convinced that the go to statement should be abolished from all "higher level" programming languages (i.e. everything except, perhaps, plain machine code). At that time I did not attach too much importance to this discovery; I now submit my considerations for publication because in very recent discussions in which the subject turned up, I have been urged to do so.

My first remark is that, although the programmer's activity ends when he has constructed a correct program, the process taking place under control of his program is the true subject matter of his activity, for it is this process that has to accomplish the desired effect; it is this process that in its dynamic behavior has to satisfy the desired specifications. Yet, once the program has been made, the "making" of the corresponding process is delegated to the machine.

My second remark is that our intellectual powers are rather geared to master static relations and that our powers to visualize processes evolving in time are relatively poorly developed. For that reason we should do (as wise programmers aware of our limitations) our utmost to shorten the conceptual gap between the static program and the dynamic process, to make the correspondence between the program (spread out in text space) and the process (spread out in time) as trivial as possible.

Let us now consider how we can characterize the progress of a process. (You may think about this question in a very concrete manner: suppose that a process, considered as a time succession of actions, is stopped after an arbitrary action, what data do we have to fix in order that we can redo the process until the very same point?) If the program text is a pure concatenation of, say, assignment statements (for the purpose of this discussion regarded the descriptions of single actions) it is sufficient to point in the program text to a point between two successive action descriptions. (In the absence of go to statements I can permit myself the syntactic ambiguity in the last three words of the previous sentence: if we parse them as "successive (action descriptions)" we mean successive in text space; if we parse as "(successive action) descriptions" we mean successive in time.) Let us call such a pointer to a suitable place in the- text a "textual index”.

When we include conditional clauses (if *B* then *A.),* alternative clauses (if *B* then *A1* else *A2),* choice clauses as introduced by C. A. R. Hoare (case[i] of (*A1*, *A2*, … , *An))* or conditional expressions as introduced by J: McCarthy *(B1 -> E1*, *B2 -> E2*, ... , *Bn -> En),* the fact remains that the progress of the process remains characterized by a single textual index.

As soon as we include in our language procedures we must admit that a single textual index is no longer sufficient. In the case that a textual index points 'to the interior of a procedure body the dynamic progress is only characterized when we also give to which call of the procedure we refer. With the inclusion of procedures we can characterize the progress of the process via a sequence of textual indices, the length of this sequence being equal to the dynamic depth of procedure calling.

Let us now consider repetition clauses (like, while *B* repeat *A* or repeat *A* until *B).* Logically speaking, such clauses are now superfluous, because we can express repetition with the aid of recursive procedures. For reasons of realism I don't wish to exclude them: on the one hand, repetition clauses can be implemented quite comfortably with present day finite equipment; on the other hand, the reasoning pattern known as "induction" makes us well equipped to retain our intellectual grasp on the processes generated by repetition clauses. With the inclusion of the repetition clauses textual indices are no longer sufficient to describe the dynamic progress of the process. With each entry into a repetition clause, however, we can associate a so-called "dynamic index," inexorably counting the ordinal number of the corresponding current repetition. As repetition clauses (just as procedure calls) may be applied nestedly, we find that now the progress of the process can always be uniquely characterized by a (mixed) sequence of textual and/or dynamic indices.

The main point is that the values of these indices are outside programmer's control; they are generated (either by the write-up of his program or by the dynamic evolution of the process) whether he wishes or not. They provide independent coordinates in which to describe the progress of the process.

Why do we need such independent coordinates? The reason is-and this seems to be inherent to sequential processes-that we can interpret the value of a variable only with respect to the progress of the process. If we wish to count the number, n say, of people in an initially empty room, we can achieve this by increasing n by one whenever we see someone entering the room. In the in-between moment that we have observed someone entering the room but have not yet performed the subsequent increase of *n* its value equals the number of people in the room minus one!

The unbridled use of the go to statement has an immediate consequence that it becomes terribly hard to find a meaningful set of coordinates in which to describe the process progress. Usually, people take into account as well the values of some well chosen variables, but this is out of the question because it is relative to the progress that the meaning of these values is to be understood! With the go to statement one can, of course, still describe the progress uniquely by a counter counting the number of actions performed since program start (viz. a kind of normalized clock). The difficulty is that such a coordinate, although unique, is utterly unhelpful. In such a coordinate system it becomes an extremely complicated affair to define all those points of progress where, say, n equals the number of persons in the room minus one!

The go to statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program. One can regard and appreciate the clauses considered as bridling its use. I do not claim that the clauses mentioned are exhaustive in the sense that they will satisfy all needs, but whatever clauses are suggested (e.g. abortion clauses) they should satisfy the requirement that a programmer independent coordinate system can be maintained to describe the process in a helpful and manageable way.

It is hard to end this with a fair acknowledgment. Am I to judge by whom my thinking has been influenced? It is fairly obvious that I am not uninfluenced by Peter Landin and Christopher Strachey. Finally I should like to record (as I remember it quite distinctly) how Heinz Zemanek at the pre-ALGOL meeting in early 1959 in Copenhagen quite explicitly expressed his doubts whether the go to statement should be treated on equal syntactic footing with the assignment statement. To a modest extent I blame myself for not having then drawn the consequences of his remark.

The remark about the undesirability of the go to statement is far from new. I remember having read the explicit recommendation to restrict the use of the go to statement to alarm exits, but I have not been able to trace it; presumably, it has been made by C. A. R. Hoare. In [1, Sec. 3.2.1.] Wirth and Hoare together make a remark in the same direction in motivating the case construction: "Like the conditional, it mirrors the dynamic structure of a program more clearly than go to statements and switches, and it eliminates the need for introducing a large number of labels in the program.

In [2] Guiseppe Jacopini seems to have proved the (logical) superfluousness of the go to statement. The exercise to translate an arbitrary flow diagram more or less mechanically into a jumpless one, however, is not to be recommended. Then the resulting flow diagram cannot be expected to be more transparent than the original one.

1. WIRTH, NIKLAUS, AND HOARE, C. A. R. A contribution to the development of ALGOL. *Comm. ACM* 9 (June 1966), 413-432.
2. BUHM, CORRADO, AND JACOPINI, GUISEPPE. Flow diagrams, Turing machines and languages with only two formation rules. *Comm. AGM* 9 (May 1966), 366-371.

This letter from Edsger W. Dijsktra was originally published in Communications of the ACM, Volume 11, Number 3 – March 1968 pp147-148. At the time it provoked some pretty heated debate, with developers adopting pro- and anti-goto positions. History, it seems, has proved Dijkstra right – no-one uses gotos now – do they?

Advertisment

Agile UML®
Leverage the Power of UML 2.0 and the Agile/ICONIX Process to spec your software designs. Design a real mapping example from use cases through C#/.Net code in our open enrollment, public training workshops.
ENTERPRISE ARCHITECT
Optimized for Microsoft Visual Studio
Agile Development with ICONIX Process
People, Process, and Pragmatism
Doug Rosenberg, Matt Stephens, and Mark Collins-Cope
Apress
UML Solution Partners
SPARX SYSTEMS
www.sparxsystems.com
ICONIX Leaders in Object Technology
www.iconixsw.com
All product names are owned by their respective owners

Opinion ♦ Prefactor and be Agile♦ Ken Pugh♦ Opinion

Ken Pugh's book "Prefactoring" caused some consternation with "agile" reviewers on the Amazon web site. Here he answers those critics …

Prefactoring evolved from a "Birds of a Feather" session at a Software Development conference in Washington, D.C. Martin Fowler, Ron Jeffries, myself, and several others discussed why software came to need refactoring and practices that might be used to lessen that need. Spurred by that discussion, I compiled a set of guidelines for creating software and entitled it Prefactoring.

Refactoring, according to Martin Fowler, "is a controlled technique for improving the design of an existing code base." "It does this by applying a series of particular code transformations, each of which are called refactorings." "*'Code smells' suggest refactoring.*"

Prefactoring guidelines emphasize things to think about before you start coding. Refactorings are code transformations performed after you have created code.

The prefactoring guidelines encompass some that are directly related to code transformations and others that are not. A commonly used refactoring is Extract Method. The "Separate Policy from Implementation" prefactoring guideline suggests organizing methods into ones that perform implementation (e.g. totaling a customer's purchases) and ones that perform policy (e.g. giving the customer a discount based on total purchases). If you think about this guideline as you develop the code, you may find yourself separating many methods, and therefore requiring fewer Extract Methods. Separating policy from implementation is not an operation you can perform mechanically. It's something that you have to do intentionally. Coding with intention makes your code start with less smells.

Other prefactoring guidelines do not deal with code transformations. "The Easiest Code to Debug is That Which is Not Written".guideline suggests that you use Google as a development tool. Before even writing a single line of code, google to see if the feature you are about to create exists in either open source or commercially available software. If it is, check it out. You may get away with not having to write a single line of code. Even if the software doesn't work out, you'll have an opportunity to see how others have approached a solution. Developing software is not just about writing code; it's about delivering solutions, regardless of how they are created.

There is no "big design up front" in prefactoring. The "Think About the Big Picture." guideline is not about big design. It suggests that you spend a little bit of time investigating the environment in which you are going to create your software. The environment (e.g. a corporate infrastructure, J2EE, or web services) may provide numerous services (e.g. security or error handling) that you don't need to develop or it may suggest ways for structuring your code to fit into the framework.

Following the prefactoring guidelines does not mean you don't refactor your code. The guidelines can help keep the smells out. But when code starts to smell, refactor it. As you go through development, you gain more knowledge about the total project, which can generate new ideas. Performing a retrospective after every release, as one guideline states, can help direct this generation of new ideas.

Agility is about delivering working software to the customer. Iterative and incremental development and customer communication, as shown in the example in the book, are key principles to enable that delivery. You can use multiple tools to achieve the goal – refactoring – performing transformations on created code is one tool; prefactoring – thinking about things before coding is another.

Ken Pugh of Pugh-Killeen Associates (www.pughkilleen.com) consults and trains on object-oriented programming, process development, real-time systems, and UNIX/ Linux.

An Analysis of AJAX

An Overview and Critique of Using XMLHTTPRequest in Client-Side Development

SOAP, Flash, AJAX – one senses a bathroom-cleanser trend these days in web-related nomenclature, akin to the fruity allusions that computer manufacturers have made over time, such as Apple, Apricot, Blackberry, Pearcom, Pineapple 6502, and …um… Olive-tti. ***Richard Vaughan*** *explains…*

Admittedly, the last one strains the point a little, nevertheless I am cooking up a new approach to web applications currently, called VIM – 'Validated Interface Management' (or something). This technology snoops users' browser-profiles on the sly, but guarantees a nice clean bathtub as compensation. The enterprise edition removes unsightly lime-scale from your boardroom and is called JIF, but I am thinking of changing this to CIF because nobody outside the UK seems to get the joke.

On a more restrained note, few software issues in the last ten years have generated quite so much interest, quite as quickly as AJAX – a moniker for the use of the XMLHTTPRequest class in script-based client-server communication – and this article examines the nature, implications and challenges of this surprise development of 2005.

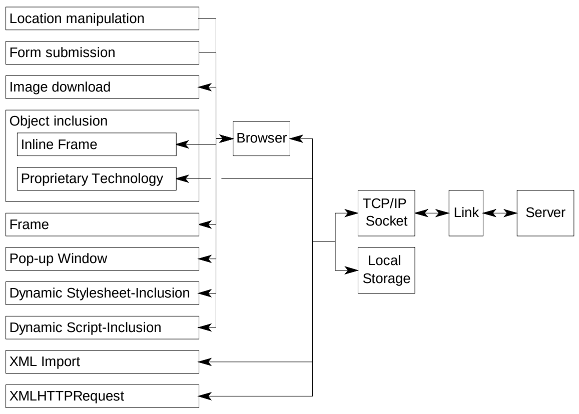
AJAX Defined

Asynchronous JavaScript and XML, or AJAX – an iconic and memorable acronym – possesses a marketable spirit of élan and dynamism, but is also rather non-representative. Firstly, there is no such thing as asynchronous JavaScript per se; this term refers to the asynchronous nature of most HTTP-based communication. Secondly, the technology behind AJAX is not limited to JavaScript, but is available from within VBScript as well.

Moreover, AJAX has nothing to do with XML intrinsically, as data may be exchanged in any format from raw character sequences, through proprietary formats, to the serious players such as JSON and XML (although scope for processing large binary-datasets is limited in client-side scripting). In fact, AJAX signifies fully bi-directional client-server transaction from within client-side scripts, using HTTP directly, and conducted through instances of the XMLHTTPRequest class. Critically, this is an improvement on other mechanisms.

Connection Spectrum

To clarify this, consider the (somewhat figurative) diagram. This lays out the range of client-server communication techniques available from within a script execution-context such as a web browser.



At the coarsest level, one can set the *location* attribute of the window object, which has the same effect as entering a literal URL in the address field of the browser. This embodies client-server communication, but a page refresh is implicit and, from that, re-initialisation of the script execution-context. It is only by recording the values of variables of a previous script, and retrieving them subsequently, can the developer maintain continuity between programs pertaining to different pages. This requires saving data in one or more cookies on the client.

The intermediate approaches, such as setting the *src* attribute of an image object, or using frames, preserve continuity of state, and can allow transaction of useful data. Moreover, they can be forced into service as a means of implementing relatively transparent (for users) communication with the server, the 'image cookie' technique being an example[1]. The problem is that this foists awkward and inelegant design on the programmer, with all the concomitant development, testing, and cross-browser issues that such dissonance carries.

An alternative is the use of proprietary technologies such as Java applets and Flash. While these too can preserve program state between client-server transactions, and allow great flexibility, they are disjoint in that they require languages, platforms and development environments that are quite distinct from the native calling-context. This can introduce intra-team impedance, causing design fragmentation, longer development times, bigger test regimens, and so on systemically. Then there is the problem of download times: waiting minutes for some unknown Flash-binary to arrive does not endear users with limited bandwidth.

Native Techniques

This leaves the truly native approaches. These comprise setting the *href* attribute of a style link from within a page's script – whereupon the property-set is downloaded automatically – or setting the *src* attribute of a script element, which causes the browser to fetch the relevant script file ('on-demand' JavaScript).

In the first case, the choice (and tradeoffs therein) are clear: a style-sheet can be downloaded according to the user agent in question, as opposed to the more algorithmic approach, where style properties are changed piece-meal. The next listing illustrates this:

```
<link id    = “StyleTag”
     rel   = "stylesheet"
     type  = "text/css"
     href  = “”
     media = "all" />


<script language="JavaScript">


if (navigator.userAgent.indexOf ("MSIE") != -1)      LoadStyle (“.../IE_Style.css”);
else
   {
   if (navigator.appName.indexOf ("Netscape" != -1)) LoadStyle (“.../NS_Style.css”);
   // Etc...
   }


function LoadStyle (StyleSheetName)
   {
   document.getElementById (“StyleTag”).href = StyleSheetName;
   }


</script>
```

In the second case, a script can download browser/functionality-specific scripts, with similar tradeoffs to the style-sheet technique.

These methods preserve execution state, yet as with the others, they work well when the server talks to the client but are impoverished bi-directionally.

XMLHTTPRequest

In contrast, the XMLHTTPRequest class is a wrapper for an underlying HTTP connection (although you can only communicate with the URL from which the script originated, unlike the on-demand technique). The commonly supported interface for the XMLHTTPRequest class is depicted in the next diagram.

XMLHTTPRequest		
onreadystatechange	: function reference	Points to event-handler function
readyState	: integer	Object status
responseText	: string	Server data as plain character string
responseXML	: DOM Node	Server data as DOM object-hierarchy (if applicable)
status	: integer	Numeric code returned by server
statusText	: string	Human readable string describing error states
abort	()	Terminates current transaction
getAllResponseHeaders	()	Returns HTTP headers sent by server
getResponseHeader	(HeaderLabel : string)	Returns a specific header
open	(method : string, URL : string [, asyncFlag : boolean [, userName : string [, password : string]]])	Opens the connection with the server
send	(content : string)	Dispatches data to server
setRequestHeader	(label : string, value : string)	Sets the value of a particular header

Using this class entails instantiating it, assigning a function reference to the *onreadystatechange* member, and then calling `open`, followed by a call to `send`. The parameters in the `open` call allow one to stipulate the HTTP method, the URL to send the request to, and optional parameters, such as a flag denoting a synchronous or asynchronous request (synchronous meaning the client waits until the response returns). Any data that must be transmitted is passed as a string during the call to `send`.

The reference assigned to *onreadystatechange* must point to a function that is called a number of times during the dialogue between server and client. This function should test the *readystate* and *status* members of the XMLHTTPRequest object and, assuming success, can process the data returned (if any) from there. If information has been sent then this can be accessed through the raw character string represented by the *responseText* attribute. However, if it is well-formed X(HT)ML then the XMLHTTPRequest object will have parsed the data on receipt (you have no choice – see below), resulting in an XML DOM-object hierarchy that is available through the *responseXML* attribute.

However, browser incompatibilities are the programmer's bane in much of client-side development, and XMLHTTPRequest is a (minor) challenge here too. All the major browsers, except IE, support the creation of XMLHTTPRequest objects as instances of native JavaScript classes, but Microsoft implement XMLHTTPRequest on the back of ActiveX, which complicates matters; moreover, two versions are possible. The listing depicts typical code for issuing a call, and for processing the response.

```
var RequestObj;

if     (window.XMLHttpRequest) RequestObj = new XMLHttpRequest
();
else if (window.ActiveXObject)
   {
   try { RequestObj = new ActiveXObject ("Msxml2.XMLHTTP"); }
   catch (e)
```

```
      {
      try      { RequestObj = new ActiveXObject ("Microsoft.XMLHTTP");
}
      catch (e) { throw e; }
      }

   }

RequestObj.onreadystatechange = Handler;

RequestObj.open (Method, URL, SyncFlag, ... );
RequestObj.send (Data);

function Handler ()
   {
   if (RequestObj.readyState == 4)
      {
      if (RequestObj.status == 200) // Do something here with
                                    // RequestObj.responseText
                                    // or RequestObj.responseXML
      else throw ("Error");

      }

   }
```

JSON

As pointed out above, XML is not mandatory, nor need it be de rigueur, and what is particularly exciting is the use of XMLHTTPRequest in conjunction with JSON or JavaScript Object Notation.

In an article in this journal in 2000[2], I suggested that a C-related format would have been preferable to the tag-based grammar and notation of XML; the argument being that syntactic and semantic parallels with the C-family of languages would make this easier to learn and more human-readable than XML, and would therefore represent true standards-unification.

```
<!-- An invoice element in XML -->

<Invoice InvNum  = “43508”
       InvDate = “23/07/2005”
       PONum   = “53098”>

  <Item Desc = “Bolts”   Num = “20” Price = “0.12” />
  <Item Desc = “Washers” Num = “10” Price = “0.13” />
  <Item Desc = “Screws”  Num = “15” Price = “0.05” />

</Invoice>


// The same data encoded in JavaScript object-literal syntax

Invoice =
  {
  InvNum  : “43508”,
  InvDate : “23/07/2005”,
  PONum   : “53098”,
  Items   : [ { Desc : “Bolts”,   Num : “20”, Price : “0.12” },
            { Desc : “Washers”, Num : “10”, Price : “0.13” },
            { Desc : “Screws”,  Num : “15”, Price : “0.05” } ]
  }
```

JSON is that idea incarnate: being a subset of JavaScript’s object-literal syntax, it relates implicitly to run-time objects, and thus removes all impedance between the communication format and object representation in client-side code. In addition, and in contrast with XML’s verbose syntax, JSON’s parsimony gives an improved ‘content to markup’ ratio, which makes it considerably more resource-efficient and human-readable than XML ever could be. The listing above shows the same data set encoded in XML and JSON.

A principal advantage to JSON is that no special parsing technology is required. JavaScript’s `eval` function (part of the underlying ECMAScript standard) allows direct invocation of the interpreter, and when passed a JSON string it will return a fully-fledged object that can be manipulated using standard notation. The next listing illustrates this.

```

var InvObj = eval(“(“
             + “{ InvNum : ‘43508’, InvDate : ‘23/07/2005’ }”
             + “)”);

alert (InvObj.InvDate); // Displays 23/07/2005
```

It follows that object-literal strings transmitted by the server can be passed to `eval` on the client, yielding data in native form. The reverse process – ‘stringifying’ an object – requires only a modest function[3], before data can be passed back to the server. Contrast the transparency of this approach with XML, which frequently requires complex parsing technologies and DOM operations, with all the inefficiencies therein.

First-Order Implications

The core implications of XMLHTTPRequest are clear. Form data can be validated on the fly, and without using awkward techniques, because the application can send the content of each input element to the server transparently as the user moves from field to field. For users, the page remains on screen, while their attention is drawn to any illegal values by means of DHTML techniques.

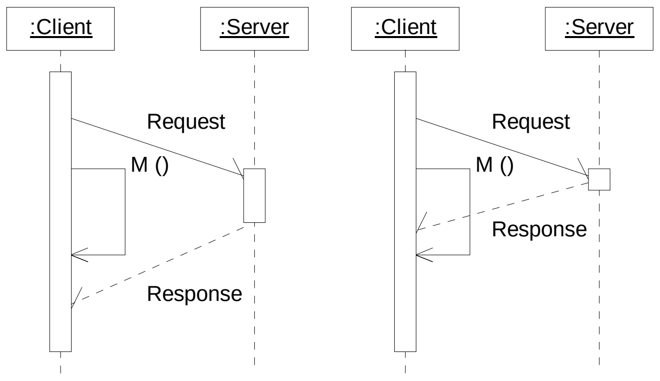
It also implies dynamic page-construction: X(HT)ML data can be requested, and when received the resulting object-hierarchy can be manipulated, as with any DOM tree, and/or included in part or whole into the structure of the page. Alternatively, text can be requested, and then placed as content into page elements upon receipt. The listing illustrates this, and shows the code required for displaying (on a rolling basis) the number of users logged on to a system.

```
<script language="JavaScript">

var NumUsers_RequestObj = null;
var NumUsers_Elem       = null;

function Init ()
  {
  NumUsers_Elem = document.getElementById ('NumUsers');
  GetNumUsers ();
  }

function GetNumUsers ()
  {
  try      { NumUsers_RequestObj = IssueXHR ("POST",
                                  ".../NumUsers.php",
                                  "",
                                  true,
                                  DisplayNumUsers); }

  catch (e) { alert ("Error sending request - "
              + e.name
              + ": "
              + e.message); }

  setTimeout ("GetNumUsers ()", 5000);  // Triggers every 5 seconds

  }

function DisplayNumUsers ()
  {
  if (NumUsers_RequestObj.readyState == 4)
    {
    if (NumUsers_RequestObj.status != 200) throw ("Error");
    NumUsers_Elem.innerText = NumUsers_RequestObj.responseText;
    }
  }

function IssueXHR (Method, URL, Data, ASync, Handler)
  {
  // Code for creating XMLHTTPRequest, then opening and sending
etc.
  }

</script>

<body onload = "Init ()">
  Users Logged on Currently = <span id = "NumUsers"></span>
</body>
```

Further to this, and while this is not part of JSON per se, JavaScript Object-Literal definitions can contain functions, meaning that the server can send procedures as and when they are needed. It follows that a page can change its functionality on the fly; implying applications that can be downloaded in a minimal form initially, and which develop their functionality contingently and dynamically from there. In the main, this is a more fine-grained alternative to the on-demand approach, but what is so attractive is that both techniques mean you need only pay for what you use. Clearly, this offers positive bandwidth and storage-requirements tradeoffs.

These points imply a major phase over the next few years of radical site-redesign in preference for user interfaces that resemble conventional desktop applications. Visitors to, for example, a railway-timetable site will be able to enter places, dates and times of departure, while the appropriate service-details appear automatically in adjacent parts of the page. Moreover, and beyond site re-design, XMLHTTPRequest implies applications whose implementation (previously) would prove challenging if not completely impracticable.

However, there are also negative implications: the price of freedom is responsibility, and XMLHTTPRequest gives the irresponsible developer the freedom to implement user interfaces that are clever in principle but frustrating in practice. Imagine reading a block of content, only to find it disappears from view as the script receives some data from the server and then updates the page dynamically, thus causing the browser to scroll your point of interest out of sight.



Concurrency Concerns

An additional concern lies with concurrency because asynchronous communication with the server means the potential for race conditions. In the diagram, method M must execute before the request returns, because some of the objects it manipulates contain data that can be changed by the response from the server.

All is well in the left-hand scenario, because the response is sufficiently tardy. In the right-hand case, however, the client receives the response (and thus executes its response handler) before M has completed, thereby invoking a bug that may prove difficult to resolve. To compound this, multiple concurrent XMLHTTPRequests increase the potential for such problems exponentially with each communication thread that is added to the execution context. The solution is to implement some form of locking, although multiple concurrent threads then introduce the potential for deadlock. These points are not unique to XMLHTTPRequest, as they apply to Java applets etc, but as with those, and with traditional application development, the only realistic way to manage such problems is good design.

The Ethical Dimension

There are, however, other implications, some of which hold out exciting possibilities, and some that are less desirable. One example is the potential to capture users' site-navigation patterns through mouse and keystroke events, and return these transparently to the server. By such means, it would be possible to build a statistical picture of the way that users interact with a site, thus allowing remodelling and refinement to reflect this, and thereby provide a better user-experience. This is clearly of benefit to all.

However, other potential applications of XMLHTTPRequest are more cynical. It is entirely possible to capture browsing patterns very precisely, which suggests that certain types of application could allow estimation of a given user's demographic profile in real time. In principle this would allow adverts that were targeted at that particular user-type to be placed within a page dynamically (indeed, before the user's very eyes).

Heat or Light

Understandably, XMLHTTPRequest also has its critics, and one area of debate (that seems to generate more heat than light) is over issues of URL-linearity, site indexing and the so-called 'broken back-button'. Some have used these points to decry the viability of AJAX-style techniques (on-demand JavaScript is in much the same position), yet the fact is that you cannot have your static browsing-cake and eat it. By definition, to introduce a dynamic component into web-based systems *is* to depart from the fixed-page model upon which these concepts rely.

In the case of the back-button, this was never more than a browser control that unwinds the URL-visitation history for a given window. It was never intended to support arbitrary undo-mechanisms; therefore one cannot really bemoan the loss of functionality that was never present originally. In the case of site indexing, it is obvious that the data-sets that traditional desktop-applications manipulate (web pages, for example) can be catalogued, as can binary executables themselves (this really does happen[4]). Yet the concept of a running program embodies the very notion of ever-changing state, and in this respect executable code constitutes an index into a particular state-space; one that search engines can never be suited to cataloguing.

Conflation

A lasting criticism, however, is that XMLHTTPRequest was designed rather poorly. Class names with numerous syllables often reflect conflation of abstraction and, from that, functionality and XMLHTTPRequest (nine syllables – tedious to rattle-off repeatedly when teaching AJAX courses) is a good example. As the fact of JSON shows, XML need not be the preferred medium, yet XMLHTTPRequest unifies XML formatting with HTTP connectivity, which are two different issues entirely – XML-processing capability comes along for the ride anyway, whether one desires it or not.

More subtly, one may actually wish to transact X(HT)ML data, but wish to avoid automatic parsing, for reasons of resource management when striking a balance between lazy and eager evaluation. Alternatively, developers may wish to implement some form of proprietary XML-parsing that is better suited to the application – a DOM parser is a relatively heavyweight affair, and mobile devices put a squeeze on resources. To stir politics into the mix, IT-related but non-technical colleagues (management) are likely to assume instantly that XMLHTTPRequest is about XML intrinsically, making it all the harder to convince them that XML is not all that it is cracked up to be, and that better alternatives exist.

A preferable approach, therefore, to implementing the spirit of XMLHTTPRequest would be a simple HTTP-connection class. This would implement client-server communication independently from the format in which data was transferred, and would have no fixed association with XML, thus leaving client-code response handlers free to process the data returned as they saw fit.

Performance and Redundancy

Another criticism is that the exception-handling technique that is required to work around browser incompatibilities can be considered an abuse of the exception-handling mechanism. However, the real problem with this is that it is slow. Obviously, ultra-performant code may not be the prime mover in client-side programming, but the inherent inelegance of this approach does tend to set one's teeth on edge.

Further to this, the response handler must check the `readyState` and `status` members of the XMLHTTPRequest object, and only proceed with processing the response if the transaction has both completed, and completed successfully. It seems that no meaningful processing can be done before satisfaction of both these conditions, which suggests that this checking should be the responsibility of the XMLHTTPRequest object, not client-code, and which would result in simplified response-handlers were the class implemented this way.

Polymorphism and Closure

Happily, a polymorphic solution to the performance problem is possible, where the correct creation-statement is determined on the first call to instantiate an XMLHTTPRequest object (using the exception-handling approach, in part), and is then called through a function reference whenever a connection object is required[5].

In the case of the redundant checking-logic in the response handler: this can be solved using the rather more exotic technique of JavaScript closures, and the listing illustrates the technique.

```
var NumUsers_RequestObj = IssueXHR ("POST",
                                    ".../NumUsers.php",
                                    "",
                                    true,
                                    DisplayNumUsers);

function IssueXHR (Method, URL, Data, ASync, ResponseHandler)
  {
  var XHRObj = ...  // Code for creating XMLHTTPRequest object

  XHRObj.onreadystatechange = function ()
    {
    if (XHRObj.readyState == 4)
      {
      if (XHRObj.status == 200) ResponseHandler (XHRObj);
      else                      throw ("Transaction complete but unsuccessful");
      }

    };

  XHRObj.open (Method, URL, ASync);
  XHRObj.send (Data);
  return XHRObj;
  }

function DisplayNumUsers (XHRObj)
  {
  NumUsers_Elem.innerText = XHRObj.responseText;
  }
```

Here a 'base' response handler is defined as an anonymous inner-function within `IssueXHR`, and every XMLHTTPRequest object created refers to that function through its `onreadystatechange` member. As before, the XMLHTTPRequest returned by `IssueXHR` is not garbage collected because it is referred to at global scope. However, the XMLHTTPRequest object's reference to the base response-handler also ensures preservation of the scope chain stretching from that function back to the global execution-context. In other words, the parameters passed into a given call to `IssueXHR`, along with its local reference to the XMLHTTPRequest object that it returns, persist as long as the XMLHTTPRequest object does – this is a closure.

The key factor is that each invocation of `IssueXHR` creates a distinct scope-chain. This means that when the base response-handler is executed, the XMLHTTPRequest object in whose context it is being called is visible, as is the reference to the client's response handler. This allows the base response-handler to check the `readyState` and `status` members of the XMLHTTPRequest object, and then call the correct client response-handler (passing the XMLHTTPRequest object) on successful completion of the transaction.

Gratifyingly, this means the client's response handler can be reduced to just the code for manipulating the data returned by the server, and if client code is not interested in the data, or if no data is returned, then the handler can be a simple empty function (which is an instance of the Null Object design-pattern)

On Balance

The reality of AJAX is that XMLHTTPRequest – given JSON, on-demand JavaScript, and dynamic style-sheet loading – is but one (albeit powerful) resource in web-based client-server development. It follows that it should be viewed in combination and on balance with these other techniques. Moreover, there are problems that developers will attack using XMLHTTPRequest that may far more soluble using one or more of the related approaches.

Finally, and for non-UK readers who may be perplexed by the introduction to this piece: Ajax is one particular brand of bathroom/kitchen cleanser, as is/was the case with Flash, Cif (neé Jif, in the UK), and Vim. Validated… virtual… I'll think of it soon.

References

[1] Image-Cookie technique www.ashleyit.com/rs/rslite/
[2] A Profile of XML Richard Vaughan - ObjectiveView Issue 4 (February 2000) - www.ratio.co.uk
[3] JSON - www.json.org
[4] Finding Binary Clones with Opstrings & Function Digests
Andrew Schulman - Dr Dobbs Journal of Programming - July, August & September 2005

Further Reading

www.ajaxinfo.com
www.ajaxian.com
www.ajaxmatters.com
www.ajaxpatterns.org

Richard Vaughan is a software developer, trainer and consultant.

advertisment

Spa 2006

Software Practice Advancement 2006

Technology
Practice
Process
People

26 - 29 March 2006
The Robinson Centre,
Bedfordshire, England

Keynote Speakers
Norman Kerth on *Retrospectives: Five years beyond the book*
Dave Thomas on *An Introduction to Pragmatic Programming*

SPA brings together experts and practitioners from around the world to exchange the latest ideas and skills in software architecture, design and development.

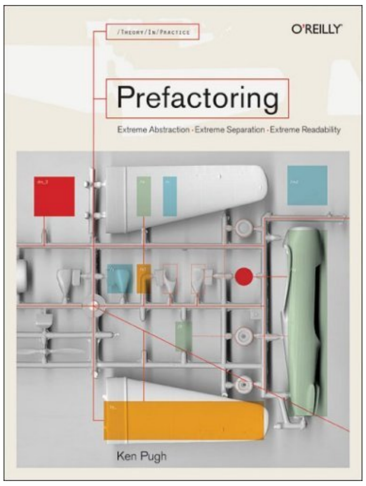
SPA provides a unique high-energy learning experience that explores a broad range of subjects from lead-edge technology, through pioneering software development and deployment practices, to innovative techniques for managing projects and the people that make up the project team.

SPA2006 is run by the British Computer Society's Software Practice Advancement group. We have a passion for advancing the art of software development - join us at

Find out more at
www.spaconference.org
or call +44 (0) 870 760 6863

BCS

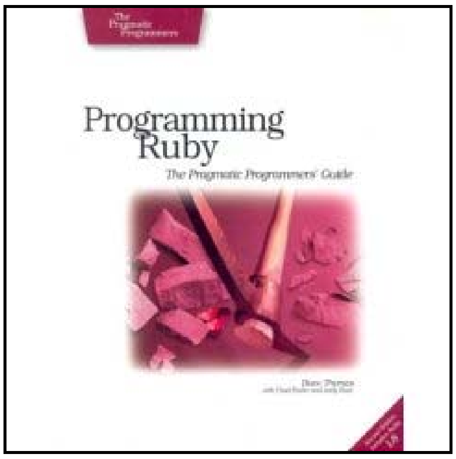
Books to look out for …


O'REILLY
Prefactoring
Extreme Abstraction · Extreme Separation · Extreme Readability
Ken Pugh

Prefactoring – Ken Pugh

I have to admit having a smile on my face when I saw the title of this book. Not that our industry goes round in circles or anything - but it seemed inevitable that given the rise of "refactoring" as technique, that someone would finally say - "hey, why don't we write our code so we don't have to refactor it!" Of course **it is** more complicated than that - and as Ken says in the book - many of his prefactoring techniques were learned during refactoring.

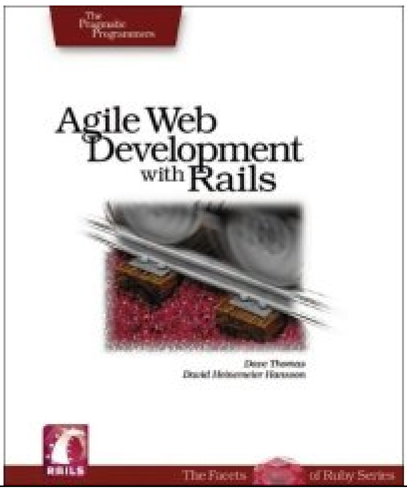
Anyhow, certainly worth a look at this book - it containts a lot of good design advice that you can apply in a pre- or post-factoring manner. Perhaps inevitably - some Amazon reviews see this book as an attack on "agile" methods but I think they miss a fundamental point - good design advice is good design advice - regardless of whether you emphasise
more design up front or refactoring existing code. As with all books - keep an open mind - and take what you think is valuable from it!



Programming Ruby - Dave Thomas

Dave Thomas has written an excellent introduction to the Ruby programming language - covering all aspects from classes and object through exception handling, modularity,
threads and processes to unit testing and using Ruby in a Windows environment. At over 800 pages, it's certainly a tome, but each page is put to good use and there's an invaluable full library reference thrown in to boot!

If you're serious about using Ruby you need this book.


The Pragmatic Programmers
Agile Web Development with Rails
Dave Thomas
David Heinemeier Hansson
RAILS
The Facets of Ruby Series

Agile Web Development with Rails
Dave Thomas and David Heinemeier Hansson

Another excellent book from Dave Thomas, this time teaming up with Rails creator David Heinemeier Hansson.
Learning Rails is not such a difficult job, and with this book it's made even easier. The book covers the Action Views (presentation), Active Record (object-relational mapping) and Action Controller subsystems, and takes the reader step by step (agile style) through the development of an web application.

Rails is well worth a look - and this book is a welcome addition to available Rails resources.

advertisment

AJAX and Advanced Javascript Training

(2 days)

While JavaScript lacks many features found in more powerful languages, and escapes the heavy demands that they experience, the complexity of modern web sites and their functionality means that the serious JavaScript developer benefits from acquiring as many advanced approaches and techniques as possible. This two-day course establishes fundamental aspects of design theory, and from there, develops powerful principles; including the use of the Strategy pattern to circumvent cross-browser coding headaches, the use of object decoration in performing complex processing of user input, and 'AJAX' or Asynchronous JavaScript and XML.

Who Should Attend

This course is appropriate to client and server-side developers alike. Advanced knowledge and experience of JavaScript and HTML is not essential, although some basic understanding of these languages is assumed.

Benefits

On completion of this course delegates will be able to:

- Understand browser implementation fundamentals
- Understand the role of AJAX in client-server development
- Instantiate and use XMLHTTPRequest objects
- Understand and use the Hyper-Text Transfer Protocol
- Select and implement appropriate data encoding and formats
- Understand and use JSON encoding techniques
- Avoid potential threading-problems and race conditions
- Work around AJAX/JavaScript limitations
- Implement user interfaces that employ AJAX techniques

Syllabus

Client-server programming overview

Basic JavaScript Recap
- Built-in types, operators
- Flow control, functions and object scope

Advanced JavaScript
- Objects and classes
- Object and class extension, prototype properties
- The global object and predefined classes
- User-defined classes
- Hard-coded class methods
- Parameterised methods – polymorphism and the strategy pattern
- Exception handling

Client-side JavaScript
- XML in brief, the XML DOM
- Browser objects and the HTML DOM
- CSS and the HTML DOM
- Events and timers
- Using the Decorator pattern to implement complex functionality
- Using the Strategy pattern to resolve browser incompatibilities

AJAX
- Client to server connection-alternatives
- XMLHTTPRequest basics
- Object instantiation, sending and receiving data
- HTTP in depth
- Synchronous and asynchronous requests
- Accessing third-party feeds
- Caching and latency considerations
- Avoiding race conditions
- Binary streams
- User interface considerations
- Usage-pattern and error-logging techniques
- Data transfer formats – plain text, JSON, XML, HTML
- Further uses of XMLHTTPRequest (URL detection, ‘last modified’ detection etc.)

For more information email info@ratio.co.uk or call +44 (0) 208 579 7900.